

The L^AT_EX3 Sources

The L^AT_EX3 Project*

November 19, 2013

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>	5
II	The <code>l3bootstrap</code> package: Bootstrap code	6
1	Using the <code>L^AT_EX3</code> modules	6
1.1	Internal functions and variables	7
III	The <code>l3names</code> package: Namespace for primitives	8
1	Setting up the <code>L^AT_EX3</code> programming language	8
IV	The <code>l3basics</code> package: Basic definitions	9
1	No operation functions	9
2	Grouping material	9
3	Control sequences and functions	10
3.1	Defining functions	10
3.2	Defining new functions using parameter text	11
3.3	Defining new functions using the signature	13
3.4	Copying control sequences	15
3.5	Deleting control sequences	16
3.6	Showing control sequences	16
3.7	Converting to and from control sequences	17
4	Using or removing tokens and arguments	18
4.1	Selecting tokens from delimited arguments	20

5	Predicates and conditionals	20
5.1	Tests on control sequences	21
5.2	Testing string equality	22
5.3	Engine-specific conditionals	23
5.4	Primitive conditionals	23
6	Internal kernel functions	24
V	The <code>l3expan</code> package: Argument expansion	27
1	Defining new variants	27
2	Methods for defining variants	28
3	Introducing the variants	28
4	Manipulating the first argument	29
5	Manipulating two arguments	30
6	Manipulating three arguments	31
7	Unbraced expansion	32
8	Preventing expansion	32
9	Internal functions and variables	34
VI	The <code>l3prg</code> package: Control structures	35
1	Defining a set of conditional functions	35
2	The boolean data type	37
3	Boolean expressions	39
4	Logical loops	40
5	Producing n copies	41
6	Detecting <code>TeX</code> 's mode	41
7	Primitive conditionals	42
8	Internal programming functions	42

VII	The <code>l3quark</code> package: Quarks	44
1	Introduction to quarks and scan marks	44
	1.1 Quarks	44
2	Defining quarks	45
3	Quark tests	45
4	Recursion	46
5	Clearing quarks away	47
6	An example of recursion with quarks	47
7	Internal quark functions	48
8	Scan marks	48
VIII	The <code>l3token</code> package: Token manipulation	49
1	All possible tokens	49
2	Character tokens	50
3	Generic tokens	53
4	Converting tokens	54
5	Token conditionals	54
6	Peeking ahead at the next token	58
7	Decomposing a macro definition	61
IX	The <code>l3int</code> package: Integers	62
1	Integer expressions	62
2	Creating and initialising integers	63
3	Setting and incrementing integers	64
4	Using integers	65
5	Integer expression conditionals	65

6	Integer expression loops	67
7	Integer step functions	69
8	Formatting integers	69
9	Converting from other formats to integers	71
10	Viewing integers	72
11	Constant integers	73
12	Scratch integers	73
13	Primitive conditionals	74
14	Internal functions	74
X	The <code>l3skip</code> package: Dimensions and skips	76
1	Creating and initialising <code>dim</code> variables	76
2	Setting <code>dim</code> variables	77
3	Utilities for dimension calculations	77
4	Dimension expression conditionals	78
5	Dimension expression loops	80
6	Using <code>dim</code> expressions and variables	81
7	Viewing <code>dim</code> variables	82
8	Constant dimensions	82
9	Scratch dimensions	82
10	Creating and initialising <code>skip</code> variables	83
11	Setting <code>skip</code> variables	83
12	Skip expression conditionals	84
13	Using <code>skip</code> expressions and variables	84
14	Viewing <code>skip</code> variables	85

15	Constant skips	85
16	Scratch skips	85
17	Inserting skips into the output	86
18	Creating and initialising muskip variables	86
19	Setting muskip variables	87
20	Using muskip expressions and variables	87
21	Viewing muskip variables	88
22	Constant muskips	88
23	Scratch muskips	88
24	Primitive conditional	88
25	Internal functions	89
XI	The l3tl package: Token lists	90
1	Creating and initialising token list variables	91
2	Adding data to token list variables	92
3	Modifying token list variables	92
4	Reassigning token list category codes	93
5	Reassigning token list character codes	93
6	Token list conditionals	94
7	Mapping to token lists	96
8	Using token lists	97
9	Working with the content of token lists	98
10	The first token from a token list	99
11	Viewing token lists	102
12	Constant token lists	103

13	Scratch token lists	103
14	Internal functions	103
XII	The l3seq package: Sequences and stacks	104
1	Creating and initialising sequences	104
2	Appending data to sequences	105
3	Recovering items from sequences	105
4	Recovering values from sequences with branching	106
5	Modifying sequences	107
6	Sequence conditionals	108
7	Mapping to sequences	108
8	Using the content of sequences directly	110
9	Sequences as stacks	110
10	Constant and scratch sequences	112
11	Viewing sequences	112
12	Internal sequence functions	112
XIII	The l3clist package: Comma separated lists	113
1	Creating and initialising comma lists	113
2	Adding data to comma lists	114
3	Modifying comma lists	115
4	Comma list conditionals	115
5	Mapping to comma lists	116
6	Using the content of comma lists directly	119
7	Comma lists as stacks	119
8	Viewing comma lists	121

9	Constant and scratch comma lists	121
XIV The l3prop package: Property lists		122
1	Creating and initialising property lists	122
2	Adding entries to property lists	123
3	Recovering values from property lists	123
4	Modifying property lists	124
5	Property list conditionals	124
6	Recovering values from property lists with branching	124
7	Mapping to property lists	125
8	Viewing property lists	126
9	Scratch property lists	127
10	Constants	127
11	Internal property list functions	127
XV The l3box package: Boxes		128
1	Creating and initialising boxes	128
2	Using boxes	129
3	Measuring and setting box dimensions	129
4	Box conditionals	130
5	The last box inserted	131
6	Constant boxes	131
7	Scratch boxes	131
8	Viewing box contents	131
9	Horizontal mode boxes	132
10	Vertical mode boxes	133

11	Primitive box conditionals	135
XVI	The l3coffins package: Coffin code layer	136
1	Creating and initialising coffins	136
2	Setting coffin content and poles	136
3	Joining and using coffins	138
4	Measuring coffins	138
5	Coffin diagnostics	139
	5.1 Constants and variables	139
XVII	The l3color package: Colour support	140
1	Colour in boxes	140
XVIII	The l3msg package: Messages	141
1	Creating new messages	141
2	Contextual information for messages	142
3	Issuing messages	143
4	Redirecting messages	145
5	Low-level message functions	146
6	Kernel-specific functions	147
7	Expandable errors	148
8	Internal l3msg functions	149
XIX	The l3keys package: Key–value interfaces	150
1	Creating keys	151
2	Sub-dividing keys	155
3	Choice and multiple choice keys	155

4	Setting keys	157
5	Handling of unknown keys	158
6	Selective key setting	159
7	Utility functions for keys	160
8	Low-level interface for parsing key-val lists	160
XX	The l3file package: File and I/O operations	162
1	File operation functions	162
1.1	Input-output stream management	163
1.2	Reading from files	164
2	Writing to files	165
2.1	Wrapping lines in output	167
2.2	Constant input-output streams	168
2.3	Primitive conditionals	168
2.4	Internal file functions and variables	168
2.5	Internal input-output functions	169
XXI	The l3fp package: floating points	170
1	Creating and initialising floating point variables	171
2	Setting floating point variables	171
3	Using floating point numbers	172
4	Floating point conditionals	173
5	Floating point expression loops	174
6	Some useful constants, and scratch variables	175
7	Floating point exceptions	176
8	Viewing floating points	177
9	Floating point expressions	178
9.1	Input of floating point numbers	178
9.2	Precedence of operators	179
9.3	Operations	179

10	Disclaimer and roadmap	185
XXII	The l3luatex package: LuaTeX-specific functions	188
1	Breaking out to Lua	188
2	Category code tables	189
XXIII	The l3candidates package: Experimental additions to l3kernel	191
1	Additions to l3basics	191
2	Additions to l3box	191
	2.1 Affine transformations	191
	2.2 Viewing part of a box	192
	2.3 Internal variables	193
3	Additions to l3clist	194
4	Additions to l3coffins	194
5	Additions to l3file	195
6	Additions to l3fp	196
7	Additions to l3prop	196
8	Additions to l3seq	197
9	Additions to l3skip	198
10	Additions to l3tl	199
11	Additions to l3tokens	200
XXIV	Implementation	201

1	l3bootstrap implementation	201
1.1	Format-specific code	201
1.2	Package-specific code part one	202
1.3	The <code>\pdfstrcmp</code> primitive in \LaTeX	203
1.4	Engine requirements	203
1.5	Package-specific code part two	204
1.6	Dealing with package-mode meta-data	205
1.7	The \LaTeX 3 code environment	207
2	l3names implementation	209
3	l3basics implementation	219
3.1	Renaming some \TeX primitives (again)	219
3.2	Defining some constants	221
3.3	Defining functions	222
3.4	Selecting tokens	222
3.5	Gobbling tokens from input	224
3.6	Conditional processing and definitions	224
3.7	Dissecting a control sequence	230
3.8	Exist or free	232
3.9	Defining and checking (new) functions	234
3.10	More new definitions	235
3.11	Copying definitions	237
3.12	Undefining functions	238
3.13	Generating parameter text from argument count	238
3.14	Defining functions from a given number of arguments	239
3.15	Using the signature to define functions	240
3.16	Checking control sequence equality	242
3.17	Diagnostic functions	242
3.18	Engine specific definitions	243
3.19	Doing nothing functions	244
3.20	String comparisons	244
3.21	Breaking out of mapping functions	246
3.22	Deprecated functions	247
4	l3expansion implementation	247
4.1	General expansion	248
4.2	Hand-tuned definitions	251
4.3	Definitions with the automated technique	253
4.4	Last-unbraced versions	254
4.5	Preventing expansion	256
4.6	Defining function variants	256
4.7	Variants which cannot be created earlier	263

5	l3prg implementation	264
5.1	Primitive conditionals	264
5.2	Defining a set of conditional functions	264
5.3	The boolean data type	264
5.4	Boolean expressions	266
5.5	Logical loops	272
5.6	Producing n copies	273
5.7	Detecting TeX's mode	274
5.8	Internal programming functions	275
6	l3quark implementation	277
6.1	Quarks	277
6.2	Scan marks	280
6.3	Deprecated quark functions	281
7	l3token implementation	282
7.1	Character tokens	282
7.2	Generic tokens	284
7.3	Token conditionals	285
7.4	Peeking ahead at the next token	295
7.5	Decomposing a macro definition	300
8	l3int implementation	301
8.1	Integer expressions	302
8.2	Creating and initialising integers	304
8.3	Setting and incrementing integers	305
8.4	Using integers	306
8.5	Integer expression conditionals	306
8.6	Integer expression loops	310
8.7	Integer step functions	312
8.8	Formatting integers	313
8.9	Converting from other formats to integers	318
8.10	Viewing integer	321
8.11	Constant integers	322
8.12	Scratch integers	323
8.13	Deprecated functions	323

9	l3skip implementation	323
9.1	Length primitives renamed	323
9.2	Creating and initialising <code>dim</code> variables	324
9.3	Setting <code>dim</code> variables	325
9.4	Utilities for dimension calculations	325
9.5	Dimension expression conditionals	326
9.6	Dimension expression loops	328
9.7	Using <code>dim</code> expressions and variables	329
9.8	Viewing <code>dim</code> variables	330
9.9	Constant dimensions	330
9.10	Scratch dimensions	331
9.11	Creating and initialising <code>skip</code> variables	331
9.12	Setting <code>skip</code> variables	332
9.13	Skip expression conditionals	332
9.14	Using <code>skip</code> expressions and variables	333
9.15	Inserting skips into the output	333
9.16	Viewing <code>skip</code> variables	334
9.17	Constant skips	334
9.18	Scratch skips	334
9.19	Creating and initialising <code>muskip</code> variables	334
9.20	Setting <code>muskip</code> variables	335
9.21	Using <code>muskip</code> expressions and variables	336
9.22	Viewing <code>muskip</code> variables	336
9.23	Constant muskips	337
9.24	Scratch muskips	337
9.25	Deprecated functions	337
10	l3tl implementation	337
10.1	Functions	337
10.2	Constant token lists	339
10.3	Adding to token list variables	340
10.4	Reassigning token list category codes	341
10.5	Reassigning token list character codes	342
10.6	Modifying token list variables	343
10.7	Token list conditionals	345
10.8	Mapping to token lists	349
10.9	Using token lists	350
10.10	Working with the contents of token lists	351
10.11	Token by token changes	353
10.12	The first token from a token list	355
10.13	Viewing token lists	360
10.14	Scratch token lists	361
10.15	Deprecated functions	361

11	l3seq implementation	361
11.1	Allocation and initialisation	362
11.2	Appending data to either end	364
11.3	Modifying sequences	365
11.4	Sequence conditionals	367
11.5	Recovering data from sequences	368
11.6	Mapping to sequences	371
11.7	Using sequences	373
11.8	Sequence stacks	374
11.9	Viewing sequences	375
11.10	Scratch sequences	376
12	l3clist implementation	376
12.1	Allocation and initialisation	376
12.2	Removing spaces around items	378
12.3	Adding data to comma lists	378
12.4	Comma lists as stacks	379
12.5	Modifying comma lists	381
12.6	Comma list conditionals	383
12.7	Mapping to comma lists	384
12.8	Using comma lists	387
12.9	Viewing comma lists	388
12.10	Scratch comma lists	388
13	l3prop implementation	388
13.1	Allocation and initialisation	389
13.2	Accessing data in property lists	390
13.3	Property list conditionals	394
13.4	Recovering values from property lists with branching	396
13.5	Mapping to property lists	396
13.6	Viewing property lists	397
14	l3box implementation	398
14.1	Creating and initialising boxes	398
14.2	Measuring and setting box dimensions	399
14.3	Using boxes	400
14.4	Box conditionals	400
14.5	The last box inserted	401
14.6	Constant boxes	401
14.7	Scratch boxes	401
14.8	Viewing box contents	401
14.9	Horizontal mode boxes	402
14.10	Vertical mode boxes	404

15	l3coffins Implementation	406
15.1	Coffins: data structures and general variables	406
15.2	Basic coffin functions	407
15.3	Measuring coffins	412
15.4	Coffins: handle and pole management	412
15.5	Coffins: calculation of pole intersections	415
15.6	Aligning and typesetting of coffins	418
15.7	Coffin diagnostics	422
15.8	Messages	428
16	l3color Implementation	429
17	l3msg implementation	430
17.1	Creating messages	430
17.2	Messages: support functions and text	431
17.3	Showing messages: low level mechanism	432
17.4	Displaying messages	435
17.5	Kernel-specific functions	442
17.6	Expandable errors	447
17.7	Showing variables	448
18	l3keys Implementation	450
18.1	Low-level interface	450
18.2	Constants and variables	454
18.3	The key defining mechanism	455
18.4	Turning properties into actions	457
18.5	Creating key properties	461
18.6	Setting keys	465
18.7	Utilities	469
18.8	Messages	470
18.9	Deprecated functions	471
19	l3file implementation	472
19.1	File operations	473
19.2	Input operations	477
19.2.1	Variables and constants	478
19.2.2	Stream management	478
19.2.3	Reading input	480
19.3	Output operations	481
19.3.1	Variables and constants	481
19.4	Stream management	482
19.4.1	Deferred writing	483
19.4.2	Immediate writing	484
19.4.3	Special characters for writing	484
19.4.4	Hard-wrapping lines to a character count	484
19.5	Messages	490

20	l3fp implementation	490
21	l3fp-aux implementation	491
22	Internal representation	491
23	Internal storage of floating points numbers	492
	23.1 Using arguments and semicolons	492
	23.2 Constants, and structure of floating points	493
	23.3 Overflow, underflow, and exact zero	495
	23.4 Expanding after a floating point number	496
	23.5 Packing digits	497
	23.6 Decimate (dividing by a power of 10)	499
	23.7 Functions for use within primitive conditional branches	501
	23.8 Small integer floating points	503
	23.9 Length of a floating point array	504
	23.10x-like expansion expandably	504
	23.11Messages	504
24	l3fp-traps Implementation	505
	24.1 Flags	505
	24.2 Traps	506
	24.3 Errors	510
	24.4 Messages	510
25	l3fp-round implementation	511
	25.1 Rounding tools	511
	25.2 The round function	514
26	l3fp-parse implementation	516
	26.1 Work plan	516
	26.1.1 Storing results	517
	26.1.2 Precedence and infix operators	518
	26.1.3 Prefix operators, parentheses, and functions	521
	26.1.4 Numbers and reading tokens one by one	522
	26.2 Main auxiliary functions	524
	26.3 Helpers	524
	26.4 Parsing one number	526
	26.4.1 Numbers: trimming leading zeros	531
	26.4.2 Number: small significand	532
	26.4.3 Number: large significand	534
	26.4.4 Number: beyond 16 digits, rounding	536
	26.4.5 Number: finding the exponent	539
	26.5 Constants, functions and prefix operators	542
	26.5.1 Prefix operators	542
	26.5.2 Constants	544
	26.5.3 Functions	546

26.6	Main functions	548
26.7	Infix operators	549
26.7.1	Closing parentheses and commas	550
26.7.2	Usual infix operators	551
26.7.3	Juxtaposition	552
26.7.4	Multi-character cases	553
26.7.5	Ternary operator	554
26.7.6	Comparisons	554
26.8	Messages	556
27	l3fp-logic Implementation	557
27.1	Syntax of internal functions	557
27.2	Existence test	557
27.3	Comparison	557
27.4	Floating point expression loops	560
27.5	Extrema	561
27.6	Boolean operations	562
27.7	Ternary operator	563
28	l3fp-basics Implementation	564
28.1	Common to several operations	565
28.2	Addition and subtraction	565
28.2.1	Sign, exponent, and special numbers	566
28.2.2	Absolute addition	568
28.2.3	Absolute subtraction	571
28.3	Multiplication	575
28.3.1	Signs, and special numbers	575
28.3.2	Absolute multiplication	577
28.4	Division	579
28.4.1	Signs, and special numbers	579
28.4.2	Work plan	580
28.4.3	Implementing the significand division	583
28.5	Unary operations	588
29	l3fp-extended implementation	589
29.1	Description of fixed point numbers	589
29.2	Helpers for numbers with extended precision	589
29.3	Multiplying a fixed point number by a short one	590
29.4	Dividing a fixed point number by a small integer	591
29.5	Adding and subtracting fixed points	592
29.6	Multiplying fixed points	593
29.7	Combining product and sum of fixed points	594
29.8	Extended-precision floating point numbers	596
29.9	Dividing extended-precision numbers	599
29.10	Inverse square root of extended precision numbers	602
29.11	Converting from fixed point to floating point	604

30	l3fp-expo implementation	606
30.1	Logarithm	607
30.1.1	Work plan	607
30.1.2	Some constants	607
30.1.3	Sign, exponent, and special numbers	607
30.1.4	Absolute ln	608
30.2	Exponential	615
30.2.1	Sign, exponent, and special numbers	615
30.3	Power	620
31	l3fp-trig Implementation	627
31.1	Direct trigonometric functions	627
31.1.1	Filtering special cases	627
31.1.2	Distinguishing small and large arguments	630
31.1.3	Small arguments	631
31.1.4	Argument reduction in degrees	632
31.1.5	Argument reduction in radians	633
31.1.6	Computing the power series	639
31.2	Inverse trigonometric functions	642
31.2.1	Arctangent and arccotangent	643
31.2.2	Arcsine and arccosine	649
31.2.3	Arccosecant and arcsecant	651
32	l3fp-convert implementation	652
32.1	Trimming trailing zeros	652
32.2	Scientific notation	653
32.3	Decimal representation	654
32.4	Token list representation	656
32.5	Formatting	657
32.6	Convert to dimension or integer	657
32.7	Convert from a dimension	658
32.8	Use and eval	659
32.9	Convert an array of floating points to a comma list	659
33	l3fp-assign implementation	660
33.1	Assigning values	660
33.2	Updating values	661
33.3	Showing values	661
33.4	Some useful constants and scratch variables	662
34	l3fp-old implementation	662
34.1	Compatibility	662
35	l3luatex implementation	665
35.1	Category code tables	666
35.2	Messages	669

36	l3candidates Implementation	670
36.1	Additions to l3box	670
36.2	Affine transformations	670
36.3	Viewing part of a box	677
36.4	Additions to l3clist	680
36.5	Additions to l3coffins	683
36.6	Rotating coffins	683
36.7	Resizing coffins	687
36.8	Additions to l3file	690
36.9	Additions to l3fp	691
36.10	Additions to l3prop	691
36.11	Additions to l3seq	692
36.12	Additions to l3skip	696
36.13	Additions to l3tl	697
36.14	Additions to l3tokens	700
	Index	703

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 `\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 `TeX parameters`. Normally this will be used for delimited functions as `expl3` provides better methods for creating simple sequential arguments.
- w** Finally, there is the **w** specifier for *weird* arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some arbitrary string).

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

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

- c** Constant: global parameters whose value should not be changed.
- g** Parameters whose value should only be set globally.
- l** Parameters whose value should only be set locally.

Each variable name is then build up in a similar way to that of a function, typically starting with the module¹ name and then a descriptive part. Variables end with a short identifier to show the variable type:

bool Either true or false.

box Box register.

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, `TeX` is a macro processor, and functions are simply macros that may or may not take arguments and expand to their replacement text. Many of the common variables are *also* macros, and if placed into the input stream will simply expand to their definition as well — a “function” with no arguments and a “token list variable” are in truth one and the same. On the other hand, some “variables” are actually registers that must be initialised and their values set and retrieved with specific functions.

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

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

2 Documentation conventions

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

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

`\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>TF</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`.

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 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.)

`\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

`_expl_package_check:`

`_expl_package_check:`

Used to ensure that all parts of `expl3` are loaded together (*i.e.* as part of `expl3`). Issues an error if a kernel package is loaded independently of the bundle.

`\l_kernel_expl_bool`

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

Part III

The 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, *<code>* is therefore used as a shorthand for “replacement text”.

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

3.1 Defining functions

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

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

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

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

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

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

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

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

Finally, the functions in Subsections 3.2 and 3.3 are primarily meant to define *base functions* only. Base functions can only have the following argument specifiers:

N and n No manipulation.

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

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 ??.

3.2 Defining new functions using parameter text

```
\cs_new:Npn
\cs_new:(cpn|Npx|cpx)
```

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

Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The definition is global and an error will result if the `<function>` is already defined.

```
\cs_new_nopar:Npn
\cs_new_nopar:(cpn|Npx|cpx)
```

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

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

```
\cs_new_protected:Npn
\cs_new_protected:(cpn|Npx|cpx)
```

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

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

```
\cs_new_protected_nopar:Npn
\cs_new_protected_nopar:(cpn|Npx|cpx)
```

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

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

```
\cs_set:Npn
\cs_set:(cpn|Npx|cpx)
```

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

Sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the `<function>` is restricted to the current \TeX group level.

`\cs_set_nopar:Npn`
`\cs_set_nopar:(cpn|Npx|cpx)`

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

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level.

`\cs_set_protected:Npn`
`\cs_set_protected:(cpn|Npx|cpx)`

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

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The 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.

`\cs_set_protected_nopar:Npn`
`\cs_set_protected_nopar:(cpn|Npx|cpx)`

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

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

`\cs_gset:Npn`
`\cs_gset:(cpn|Npx|cpx)`

`\cs_gset:Npn <function> <parameters> {<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. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current \TeX group level: the assignment is global.

`\cs_gset_nopar:Npn`
`\cs_gset_nopar:(cpn|Npx|cpx)`

`\cs_gset_nopar:Npn <function> <parameters> {<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.

`\cs_gset_protected:Npn`
`\cs_gset_protected:(cpn|Npx|cpx)`

`\cs_gset_protected:Npn <function> <parameters> {<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. 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.

$\backslash\text{cs_gset_protected_nopar:Npn}$ $\backslash\text{cs_gset_protected_nopar: (cpn Npx cpx)}$	$\backslash\text{cs_gset_protected_nopar:Npn } \langle\text{function}\rangle \langle\text{parameters}\rangle \{ \langle\text{code}\rangle \}$
---	--

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

3.3 Defining new functions using the signature

$\backslash\text{cs_new:Nn}$ $\backslash\text{cs_new: (cn Nx cx)}$	$\backslash\text{cs_new:Nn } \langle\text{function}\rangle \{ \langle\text{code}\rangle \}$
---	--

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

$\backslash\text{cs_new_nopar:Nn}$ $\backslash\text{cs_new_nopar: (cn Nx cx)}$	$\backslash\text{cs_new_nopar:Nn } \langle\text{function}\rangle \{ \langle\text{code}\rangle \}$
---	---

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

$\backslash\text{cs_new_protected:Nn}$ $\backslash\text{cs_new_protected: (cn Nx cx)}$	$\backslash\text{cs_new_protected:Nn } \langle\text{function}\rangle \{ \langle\text{code}\rangle \}$
---	---

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

$\backslash\text{cs_new_protected_nopar:Nn}$ $\backslash\text{cs_new_protected_nopar: (cn Nx cx)}$	$\backslash\text{cs_new_protected_nopar:Nn } \langle\text{function}\rangle \{ \langle\text{code}\rangle \}$
---	--

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

<hr/> <code>\cs_set:Nn</code> <hr/> <code>\cs_set:(cn Nx cx)</code>	<code>\cs_set:Nn <function> {<code>}</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 T _E X group level.
<hr/> <code>\cs_set_nopar:Nn</code> <hr/> <code>\cs_set_nopar:(cn Nx cx)</code>	<code>\cs_set_nopar:Nn <function> {<code>}</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 T _E X group level.
<hr/> <code>\cs_set_protected:Nn</code> <hr/> <code>\cs_set_protected:(cn Nx cx)</code>	<code>\cs_set_protected:Nn <function> {<code>}</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 T _E X group level.
<hr/> <code>\cs_set_protected_nopar:Nn</code> <hr/> <code>\cs_set_protected_nopar:(cn Nx cx)</code>	<code>\cs_set_protected_nopar:Nn <function> {<code>}</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 T _E X group level.
<hr/> <code>\cs_gset:Nn</code> <hr/> <code>\cs_gset:(cn Nx cx)</code>	<code>\cs_gset:Nn <function> {<code>}</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> <hr/> <code>\cs_gset_nopar:(cn Nx cx)</code>	<code>\cs_gset_nopar:Nn <function> {<code>}</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: 2012-09-09

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` `\cs_if_exist_use:cTF`

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:N` ★ `\cs_if_exist_use:NTF ⟨control sequence⟩ {⟨true code⟩} {⟨false code⟩}`

`\cs_if_exist_use:c` ★ `\cs_if_exist_use:cTF` ★

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 followed by the *⟨true code⟩*.

`\cs:w` ★ `\cs:w ⟨control sequence name⟩ \cs_end:`

`\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 the situation in force when first function absorbs the token).

```

\use:n ★ \use:n {\group1}
\use:(nn|nnn|nnnn) ★ \use:nn {\group1} {\group2}
\use:nnn {\group1} {\group2} {\group3}
\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.

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

<code>\use_ii:nn</code>	★	These functions absorb two arguments from the input stream. The function <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.
-------------------------	---	---

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

<code>\use_ii:nnn</code>	★	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.
<code>\use_iii:nnn</code>	★	

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

<code>\use_ii:nnnn</code>	★	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.
<code>\use_iii:nnnn</code>	★	
<code>\use_iv:nnnn</code>	★	

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

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

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

will result in the input stream containing

```
abc { def }
```

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

<code>\use_none:n</code>	★	<code>\use_none:n {⟨group₁⟩}</code>
--------------------------	---	--

<code>\use_none:(nn nnn nnnn nnnnn nnnnnn nnnnnnn nnnnnnnn nnnnnnnnn)</code>	★	
--	---	--

These functions absorb between one and nine groups from the input stream, leaving nothing on the resulting input stream. These functions work after a single expansion. One or more of the `n` arguments may be an unbraced single token (*i.e.* an `N` argument).

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

<hr/> <code>\use_none_delimit_by_q_nil:w</code> <hr/>	★	<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⟩* from the input stream delimited by the marker given in the function name, leaving nothing in the input stream.

<hr/> <code>\use_i_delimit_by_q_nil:nw</code> <hr/>	★	<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⟩* from 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 $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ and $\mathrm{L}^{\mathrm{A}}\mathrm{T}_{\mathrm{E}}\mathrm{X}\ 2_{\varepsilon}$. 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

```
\cs_if_eq_p:NN ★ \cs_if_eq_p:NN {\langle cs_1 \rangle} {\langle cs_2 \rangle}
\cs_if_eq:NNTF ★ \cs_if_eq:NNTF {\langle cs_1 \rangle} {\langle cs_2 \rangle} {\langle true code \rangle} {\langle false code \rangle}
```

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

```
\cs_if_exist_p:N ★ \cs_if_exist_p:N \langle control sequence \rangle
\cs_if_exist_p:c ★ \cs_if_exist:NNTF \langle control sequence \rangle {\langle true code \rangle} {\langle false code \rangle}
\cs_if_exist:NNTF ★
\cs_if_exist:cTF ★
```

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

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

5.2 Testing string equality

<code>\str_if_eq_p:nn</code>	★	<code>\str_if_eq_p:nn</code>	$\{\langle t_1 \rangle\}$ $\{\langle t_2 \rangle\}$
<code>\str_if_eq_p:(Vn on no nV VV)</code>	★	<code>\str_if_eq:nnTF</code>	$\{\langle t_1 \rangle\}$ $\{\langle t_2 \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\str_if_eq:nnTF</code>	★		
<code>\str_if_eq:(Vn on no nV VV)TF</code>	★		

Compares the two $\langle token\ lists \rangle$ 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</code>	$\{\langle t_1 \rangle\}$ $\{\langle t_2 \rangle\}$
<code>\str_if_eq_x:nnTF</code>	★	<code>\str_if_eq_x:nnTF</code>	$\{\langle t_1 \rangle\}$ $\{\langle t_2 \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

New: 2012-06-05

Compares the full expansion of two $\langle token\ lists \rangle$ 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.

<code>\str_case:nnTF</code>	★	<code>\str_case:nnTF</code>	$\{\langle test\ string \rangle\}$
<code>\str_case:onTF</code>	★	{	
		$\{\langle string\ case_1 \rangle\}$	$\{\langle code\ case_1 \rangle\}$
		$\{\langle string\ case_2 \rangle\}$	$\{\langle code\ case_2 \rangle\}$
		...	
		$\{\langle string\ case_n \rangle\}$	$\{\langle code\ case_n \rangle\}$
		}	
		$\{\langle true\ code \rangle\}$	
		$\{\langle false\ code \rangle\}$	

New: 2013-07-24

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.

<code>\str_case_x:nnTF</code> ★	<code>\str_case_x:nnn {<test string>}</code>
New: 2013-07-24	<pre> { {<string case₁>} {<code case₁>} {<string case₂>} {<code case₂>} ... {<string case_n>} {<code case_n>} } {<true code>} {<false code>} </pre>

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.

5.3 Engine-specific conditionals

<code>\luatex_if_engine_p:</code> ★	<code>\luatex_if_engine:TF {<true code>} {<false code>}</code>
<code>\luatex_if_engine:TF</code> ★	Detects is the document is being compiled using LuaTeX.
Updated: 2011-09-06	
<code>\pdftex_if_engine_p:</code> ★	<code>\pdftex_if_engine:TF {<true code>} {<false code>}</code>
<code>\pdftex_if_engine:TF</code> ★	Detects is the document is being compiled using pdfTeX.
Updated: 2011-09-06	
<code>\xetex_if_engine_p:</code> ★	<code>\xetex_if_engine:TF {<true code>} {<false code>}</code>
<code>\xetex_if_engine:TF</code> ★	Detects is the document is being compiled using XeTeX.
Updated: 2011-09-06	

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

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

<code>\if_meaning:w</code>	★	<code>\if_meaning:w <arg₁₂</code>
----------------------------	---	---

`\if_meaning:w` executes *<true code>* when *<arg_{1 and *<arg_{2 are the same, otherwise it executes *<false code>*. *<arg_{1 and *<arg_{2 could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.}*}*}*}*

T_EXhackers note: This is T_EX's `\ifx`.

<code>\if:w</code>	★	<code>\if:w <token₁₂</code>
<code>\if_catcode:w</code>	★	<code>\if_catcode:w <token₁₂</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>	★	Execute <i><true code></i> if currently in horizontal mode, otherwise execute <i><false code></i> . Similar for the other functions.
<code>\if_mode_math:</code>	★	
<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 <i><cs></i> exists according to the criteria for <code>\cs_if_exist_p:N</code> , and if not raises a kernel-level error.

<hr/> <code>__chk_if_free_cs:N</code> <code>__chk_if_free_cs:c</code> <hr/>	<code>__chk_if_free_cs:N</code> $\langle cs \rangle$ This function checks that $\langle cs \rangle$ is free according to the criteria for <code>\cs_if_free_p:N</code> , and if not raises a kernel-level error.
<hr/> <code>__cs_count_signature:N</code> ★ <code>__cs_count_signature:c</code> ★ <hr/>	<code>__cs_count_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 $\langle number \rangle$ 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> ★ <hr/>	<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> ★ <hr/>	<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> ★ <hr/>	<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> <hr/>	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:c</code> <hr/>	<code>__kernel_register_show:N</code> $\langle register \rangle$ 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> <hr/>	<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.

`_str_if_eq_x_return:nn` `_str_if_eq_x_return:nn {\langle t1 \rangle} {\langle t2 \rangle}`

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.

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: 2013-07-09

`\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 which 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 T_EX 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. Let's pretend we want to set the control sequence whose name is given by `b \l_tmpa_tl b` equal to the list of tokens `\aaa a`. Furthermore we want to store the execution of it in a `<tl var>`. In this example we assume `\l_tmpa_tl` contains the text string `lur`. The straightforward approach is

```
\tl_set:No \l_tmpb_tl { \tl_set:cn { b \l_tmpa_tl b } { \aaa a } }
```

Unfortunately this only puts `\exp_args:Nc \tl_set:Nn {b \l_tmpa_tl b} { \aaa a }` into `\l_tmpb_tl` and not `\tl_set:Nn \blurb { \aaa a }` as we probably wanted. Using `\tl_set:Nx` is not an option as that will die horribly. Instead we can do a

```
\tl_set:Nf \l_tmpb_tl { \tl_set:cn { b \l_tmpa_tl b } { \aaa a } }
```

which puts the desired result in `\l_tmpb_tl`. It requires `\tl_set:Nf` to be defined as

```
\cs_set_nopar:Npn \tl_set:Nf { \exp_args:NNf \tl_set:Nn }
```

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!

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.

```
\exp_args:No ★ \exp_args:No <function> {<tokens>} ...
```

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

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

This function absorbs two arguments (the `<function>` name and the `<tokens>`). The `<tokens>` are expanded until only characters remain, and are then turned into a control sequence. (An internal error will occur if such a conversion is not possible). The result is inserted into the input stream *after* reinsertion of the `<function>`. Thus the `<function>` may take more than one argument: all others will be left unchanged.

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

<hr/> <hr/>	<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/>	<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/>	<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.
<hr/> <hr/>	<hr/> <hr/>
<code>\exp_args:Nx</code>	<code>\exp_args:Nx</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$
	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$) and exhaustively expands the $\langle tokens \rangle$ second. 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.

5 Manipulating two arguments

<hr/> <hr/>	<hr/> <hr/>
<code>\exp_args:NNo</code> ★	<code>\exp_args:NNo</code> $\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle tokens \rangle\}$
<code>\exp_args:(NNc NNv NNf Nco Ncf Ncc 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.
<hr/> <hr/>	<hr/> <hr/>
<code>\exp_args:Nno</code> ★	<code>\exp_args:Nno</code> $\langle token \rangle$ $\{\langle tokens_1 \rangle\}$ $\{\langle tokens_2 \rangle\}$
<code>\exp_args:(NnV Nnf Noo Nof Noc Nff Nfo 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 <token₁> <token₂> {\tokens}</code>
<code>\exp_args:(Nnx Ncx 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 Nccc NcNc NcNo Ncco)</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:NNNo <token₁> <token₂> <token₃> {\tokens}</code>
<code>\exp_args:(NNno Nnno Nnnc Nooo)</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:(NNox Nnnx Nnox Noox Ncnx Nccx)</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.*

7 Unbraced expansion

<code>\exp_last_unbraced:Nf</code>	★	<code>\exp_last_unbraced:Nno</code>	$\langle token \rangle$
<code>\exp_last_unbraced:(NV No Nv Nco NcV NNV NNo Nno Noo Nfo NNNV NNNo NnNo)</code>	★		$\langle tokens_1 \rangle$ $\langle tokens_2 \rangle$

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 \mypkg_foo: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>	$\langle function \rangle$	$\{\langle tokens \rangle\}$
------------------------------------	------------------------------------	----------------------------	------------------------------

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

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

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>	$\langle token_1 \rangle$	$\langle token_2 \rangle$
----------------------------	---	----------------------------	---------------------------	---------------------------

Carries out a single expansion of $\langle token_2 \rangle$ (which may consume arguments) prior to the expansion of $\langle token_1 \rangle$. If $\langle token_2 \rangle$ is a T_EX primitive, it will be executed rather than expanded, while if $\langle token_2 \rangle$ has not expansion (for example, if it is a character) then it will be left unchanged. It is important to notice that $\langle token_1 \rangle$ may be *any* single token, including group-opening and -closing tokens (`{` or `}` assuming normal T_EX category codes). Unless specifically required, 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.
<hr/> <hr/>	<code>\exp_stop_f:</code> ★	<code>\function:f</code> $\langle tokens \rangle$ <code>\exp_stop_f:</code> $\langle more tokens \rangle$
Updated: 2011-06-03		This function terminates an f -type expansion. Thus if a function <code>\function:f</code> starts an f -type expansion and all of $\langle tokens \rangle$ are expandable <code>\exp_stop_f</code> will terminate the expansion of tokens even if $\langle more tokens \rangle$ 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 (\sqcup).

9 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.

`\::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 $\text{\LaTeX}3$ 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 \<name₁>:<arg spec₁> \<name₂>:<arg spec₂></code>
<code>\prg_set_eq_conditional:NNn</code>	<code>{<conditions>}</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 `<conditions>`, which should be one or more of **p**, **T**, **F** and **TF**.

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

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

<code>\bool_new:N</code>	<code>\bool_new:N <boolean></code>
<code>\bool_new:c</code>	

Creates a new `<boolean>` or raises an error if the name is already taken. The declaration is global. The `<boolean>` will initially be **false**.

<code>\bool_set_false:N</code>	<code>\bool_set_false:N <boolean></code>
<code>\bool_set_false:c</code>	
<code>\bool_gset_false:N</code>	
<code>\bool_gset_false:c</code>	Sets <code><boolean></code> logically false .

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

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.

$\backslash\text{bool_if_p:n}$ ★	$\backslash\text{bool_if_p:n} \{ \langle boolean\ expression \rangle \}$
$\backslash\text{bool_if:nTF}$ ★	$\backslash\text{bool_if:nTF} \{ \langle boolean\ expression \rangle \} \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$

Updated: 2012-07-08

Tests the current truth of $\langle boolean\ expression \rangle$, and continues expansion based on this result. The $\langle boolean\ expression \rangle$ 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 $\backslash\text{int_compare_p:nNn} \{ 1 \} = \{ \text{\error} \}$. The logical Not applies to the next predicate or group.

<hr/>	
<code>\bool_not_p:n</code> ☆	<code>\bool_not_p:n {<boolean expression>}</code>
Updated: 2012-07-08	Function version of <code>!(<boolean expression>)</code> within a boolean expression.
<hr/>	
<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.
<hr/>	

4 Logical loops

Loops using either boolean expressions or stored boolean values.

<hr/>	
<code>\bool_do_until:Nn</code> ☆	<code>\bool_do_until:Nn <boolean> {<code>}</code>
<code>\bool_do_until:cn</code> ☆	Places the <code><code></code> in the input stream for T _E X to process, and then checks the logical value of the <code><boolean></code> . If it is <code>false</code> then the <code><code></code> will be inserted into the input stream again and the process will loop until the <code><boolean></code> is <code>true</code> .
<hr/>	
<code>\bool_do_while:Nn</code> ☆	<code>\bool_do_while:Nn <boolean> {<code>}</code>
<code>\bool_do_while:cn</code> ☆	Places the <code><code></code> in the input stream for T _E X to process, and then checks the logical value of the <code><boolean></code> . If it is <code>true</code> then the <code><code></code> will be inserted into the input stream again and the process will loop until the <code><boolean></code> is <code>false</code> .
<hr/>	
<code>\bool_until_do:Nn</code> ☆	<code>\bool_until_do:Nn <boolean> {<code>}</code>
<code>\bool_until_do:cn</code> ☆	This function firsts checks the logical value of the <code><boolean></code> . If it is <code>false</code> the <code><code></code> is placed in the input stream and expanded. After the completion of the <code><code></code> the truth of the <code><boolean></code> is re-evaluated. The process will then loop until the <code><boolean></code> is <code>true</code> .
<hr/>	
<code>\bool_while_do:Nn</code> ☆	<code>\bool_while_do:Nn <boolean> {<code>}</code>
<code>\bool_while_do:cn</code> ☆	This function firsts checks the logical value of the <code><boolean></code> . If it is <code>true</code> the <code><code></code> is placed in the input stream and expanded. After the completion of the <code><code></code> the truth of the <code><boolean></code> is re-evaluated. The process will then loop until the <code><boolean></code> is <code>false</code> .
<hr/>	
<code>\bool_do_until:nn</code> ☆	<code>\bool_do_until:nn {<boolean expression>} {<code>}</code>
Updated: 2012-07-08	Places the <code><code></code> in the input stream for T _E X to process, and then checks the logical value of the <code><boolean expression></code> as described for <code>\bool_if:nTF</code> . If it is <code>false</code> then the <code><code></code> will be inserted into the input stream again and the process will loop until the <code><boolean expression></code> evaluates to <code>true</code> .
<hr/>	
<code>\bool_do_while:nn</code> ☆	<code>\bool_do_while:nn {<boolean expression>} {<code>}</code>
Updated: 2012-07-08	Places the <code><code></code> in the input stream for T _E X to process, and then checks the logical value of the <code><boolean expression></code> as described for <code>\bool_if:nTF</code> . If it is <code>true</code> then the <code><code></code> will be inserted into the input stream again and the process will loop until the <code><boolean expression></code> evaluates to <code>false</code> .
<hr/>	

<hr/> <code>\bool_until_do:nn</code> ☆	<code>\bool_until_do:nn {\boolean expression} {\code}</code>
Updated: 2012-07-08	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 .

<hr/> <code>\bool_while_do:nn</code> ☆	<code>\bool_while_do:nn {\boolean expression} {\code}</code>
Updated: 2012-07-08	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 .

5 Producing n copies

<hr/> <code>\prg_replicate:nn</code> ☆	<code>\prg_replicate:nn {\integer expression} {\tokens}</code>
Updated: 2011-07-04	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.

6 Detecting T_EX's mode

<hr/> <code>\mode_if_horizontal_p:</code> ☆	<code>\mode_if_horizontal_p:</code>
<code>\mode_if_horizontal:TF</code> ☆	<code>\mode_if_horizontal:TF {\true code} {\false code}</code>
	Detects if T _E X is currently in horizontal mode.

<hr/> <code>\mode_if_inner_p:</code> ☆	<code>\mode_if_inner_p:</code>
<code>\mode_if_inner:TF</code> ☆	<code>\mode_if_inner:TF {\true code} {\false code}</code>
	Detects if T _E X is currently in inner mode.

<hr/> <code>\mode_if_math_p:</code> ☆	<code>\mode_if_math:TF {\true code} {\false code}</code>
<code>\mode_if_math:TF</code> ☆	
Updated: 2011-09-05	Detects if T _E X is currently in maths mode.

<hr/> <code>\mode_if_vertical_p:</code> ☆	<code>\mode_if_vertical_p:</code>
<code>\mode_if_vertical:TF</code> ☆	<code>\mode_if_vertical:TF {\true code} {\false code}</code>
	Detects if T _E X is currently in vertical mode.

7 Primitive conditionals

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

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

<code>\if_bool:N</code>	★	<code>\if_bool:N <boolean> <true code> \else: <false code> \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 \TeX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the `&` token inside `\halign`. This is necessary to allow grabbing of tokens for testing purposes, as \TeX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as `\peek_after:Nw` will result in a forbidden comparison of the internal `\endtemplate` token, yielding a fatal error. Each `\group_align_safe_begin:` must be matched by a `\group_align_safe_end:`, although this does not have to occur within the same function.

<code>\scan_align_safe_stop:</code>	<code>\scan_align_safe_stop:</code>
-------------------------------------	-------------------------------------

Updated: 2011-09-06

Stops \TeX 's scanner looking for expandable control sequences at the beginning of an alignment cell. This function is required, for example, to obtain the expected output when testing `\mode_if_math:TF` at the start of a math array cell: placing `\scan_align_safe_stop:` before `\mode_if_math:TF` will give the correct result. This function does not destroy any kerning if used in other locations, but *does* render functions non-expandable.

\TeX hackers note: This is a protected version of `\prg_do_nothing:`, which therefore stops \TeX 's scanner in the circumstances described without producing any affect on the output.

<code>__prg_variable_get_scope:N</code>	★	<code>__prg_variable_get_scope:N <variable></code>
--	---	---

Returns the scope (g for global, blank otherwise) for the *<variable>*.

<code>__prg_variable_get_type:N</code>	★	<code>__prg_variable_get_type:N <variable></code>
---	---	--

Returns the type of *<variable>* (tl, int, etc.)

<u><code>__prg_break_point:Nn</code></u> ★	<code>__prg_break_point:Nn \<type>_map_break: <tokens></code> <p>Used to mark the end of a recursion or mapping: the functions <code>\<type>_map_break:</code> and <code>\<type>_map_break:n</code> use this to break out of the loop. After the loop ends, the <code><tokens></code> are inserted into the input stream. This occurs even if the break functions are <i>not</i> applied: <code>__prg_break_point:Nn</code> is functionally-equivalent in these cases to <code>\use_ii:nn</code>.</p>
<u><code>__prg_map_break:Nn</code></u> ★	<code>__prg_map_break:Nn \<type>_map_break: {(user code)}</code> <code>...</code> <code>__prg_break_point:Nn \<type>_map_break: {(ending code)}</code> <p>Breaks a recursion in mapping contexts, inserting in the input stream the <code><user code></code> after the <code><ending code></code> for the loop. The function breaks loops, inserting their <code><ending code></code>, until reaching a loop with the same <code><type></code> as its first argument. This <code>\<type>_map_break:</code> argument is simply used as a recognizable marker for the <code><type></code>.</p>
<u><code>\g__prg_map_int</code></u>	<p>This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions <code>__prg_map_1:w</code>, <code>__prg_map_2:w</code>, <i>etc.</i>, labelled by <code>\g__prg_map_int</code> hold functions to be mapped over various list datatypes in inline and variable mappings.</p>
<u><code>__prg_break_point:</code></u> ★	<p>This copy of <code>\prg_do_nothing:</code> is used to mark the end of a fast short-term recursions: the function <code>__prg_break:n</code> uses this to break out of the loop.</p>
<u><code>__prg_break:</code></u> ★ <u><code>__prg_break:n</code></u> ★	<code>__prg_break:n {(tokens)} ... __prg_break_point:</code> <p>Breaks a recursion which has no <code><ending code></code> and which is not a user-breakable mapping (see for instance <code>\prop_get:Nn</code>), and inserts <code><tokens></code> in the input stream.</p>

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. 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 6.

<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 Clearing quarks away

```
\use_none_delimit_by_q_recursion_stop:w \use_none_delimit_by_q_recursion_stop:w <tokens>
\q_recursion_stop
```

Used to prematurely terminate a recursion using `\q_recursion_stop` as the end marker, removing any remaining `<tokens>` from the input stream.

```
\use_i_delimit_by_q_recursion_stop:nw \use_i_delimit_by_q_recursion_stop:nw {<insertion>}
<tokens> \q_recursion_stop
```

Used to prematurely terminate a recursion using `\q_recursion_stop` as the end marker, removing any remaining `<tokens>` from the input stream. The `<insertion>` is then made into the input stream after the end of the recursion.

6 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.

```
1 \cs_new:Npn \my_map_dbl:nn #1#2
2 {
3   \cs_set:Npn \__my_map_dbl_fn:nn ##1 ##2 {#2}
4   \__my_map_dbl:nn #1 \q_recursion_tail \q_recursion_tail \q_recursion_stop
5 }
```

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.

```
6 \cs_new:Nn \__my_map_dbl:nn
7 {
8   \quark_if_recursion_tail_stop:n {#1}
9   \quark_if_recursion_tail_stop:n {#2}
10  \__my_map_dbl_fn:nn {#1} {#2}
```

Finally, recurse:

```
11 \__my_map_dbl:nn
12 }
```

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`.

7 Internal quark functions

<code>_quark_if_recursion_tail_break:NN</code>	<code>_quark_if_recursion_tail_break:nN</code> $\langle\textit{token list}\rangle$
<code>_quark_if_recursion_tail_break:nN</code>	<code>\langle\textit{type}\rangle_map_break:</code>

Tests if $\langle\textit{token list}\rangle$ contains only `_q_recursion_tail`, and if so terminates the recursion using `\langle\textit{type}\rangle_map_break:`. The recursion end should be marked by `\prg_break_point:Nn \langle\textit{type}\rangle_map_break:`.

8 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence will never expand in an expansion context and will be (largely) invisible if they are encountered in a typesetting context.

Like quarks, they can be used as delimiters in weird functions and are often safer to use for this purpose. Since they are harmless when executed by T_EX in non-expandable contexts, they can be used to mark the end of a set of instructions. This allows to skip to that point if the end of the instructions should not be performed (see `l3regex`).

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

<code>_scan_new:N</code>	<code>_scan_new:N</code> $\langle\textit{scan mark}\rangle$
----------------------------	---

Creates a new $\langle\textit{scan mark}\rangle$ which is set equal to `\scan_stop:`. The $\langle\textit{scan mark}\rangle$ 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> $\langle\textit{tokens}\rangle$ <code>\s_stop</code>
--	--

Removes the $\langle\textit{tokens}\rangle$ and `\s_stop` from the input stream. This leads to a low-level T_EX 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 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>	

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

`\char_set_catcode_other:N \%`

The assignment is local.

<code>\char_set_catcode_escape:n</code>	<code>\char_set_catcode_letter:n</code> $\{ \langle integer\ expression \rangle \}$
<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>	

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.

<hr/> <hr/> <code>\char_set_catcode:nn</code>	<code>\char_set_catcode:nn {⟨integer₁⟩} {⟨integer₂⟩}</code>
	These functions set the category code of the <i>⟨character⟩</i> which has character code as given by the <i>⟨integer expression⟩</i> . The first <i>⟨integer expression⟩</i> is the character code and the second is the category code to apply. The setting applies within the current T _E X group. In general, the symbolic functions <code>\char_set_catcode_⟨type⟩</code> should be preferred, but there are cases where these lower-level functions may be useful.
<hr/> <hr/> <code>\char_value_catcode:n</code> ★	<code>\char_value_catcode:n {⟨integer expression⟩}</code>
	Expands to the current category code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> .
<hr/> <hr/> <code>\char_show_value_catcode:n</code>	<code>\char_show_value_catcode:n {⟨integer expression⟩}</code>
	Displays the current category code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> on the terminal.
<hr/> <hr/> <code>\char_set_lccode:nn</code>	<code>\char_set_lccode:nn {⟨integer₁⟩} {⟨integer₂⟩}</code>
	This function set up the behaviour of <i>⟨character⟩</i> when found inside <code>\tl_to_lowercase:n</code> , such that <i>⟨character₁⟩</i> will be converted into <i>⟨character₂⟩</i> . The two <i>⟨characters⟩</i> may be specified using an <i>⟨integer expression⟩</i> for the character code concerned. This may include the T _E X ‘ <i>⟨character⟩</i> ’ 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/> <hr/> <code>\char_value_lccode:n</code> ★	<code>\char_value_lccode:n {⟨integer expression⟩}</code>
	Expands to the current lower case code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> .
<hr/> <hr/> <code>\char_show_value_lccode:n</code>	<code>\char_show_value_lccode:n {⟨integer expression⟩}</code>
	Displays the current lower case code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> on the terminal.

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

This function set up the behaviour of $\langle character \rangle$ when found inside `\tl_to_uppercase:n`, such that $\langle character_1 \rangle$ will be converted into $\langle character_2 \rangle$. The two $\langle characters \rangle$ may be specified using an $\langle integer expression \rangle$ for the character code concerned. This may include the T_EX ‘ $\langle character \rangle$ ’ method for converting a single character into its character code:

```
\char_set_uccode:nn { '\a } { '\A } % Standard behaviour
\char_set_uccode:nn { '\A } { '\A - 32 }
\char_set_uccode:nn { 60 } { 50 }
```

The setting applies within the current T_EX group.

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

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

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

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

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

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_EX group.

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

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

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

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

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

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_EX group.

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

Expands to the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.

<hr/> <hr/>	<hr/>
<code>\char_show_value_sfcode:n</code>	<code>\char_show_value_sfcode:n {\langle integer expression \rangle}</code>
<hr/>	<hr/>
	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 will require special handling at the document level as they are 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.
<hr/>	<hr/>
New: 2012-01-23	

<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.
<hr/>	<hr/>
New: 2012-01-23	

3 Generic tokens

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

<hr/>	<hr/>
<code>\c_group_begin_token</code> <code>\c_group_end_token</code> <code>\c_math_toggle_token</code> <code>\c_alignment_token</code> <code>\c_parameter_token</code> <code>\c_math_superscript_token</code> <code>\c_math_subscript_token</code> <code>\c_space_token</code>	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.
<hr/>	<hr/>

<hr/>	<hr/>
<code>\c_catcode_letter_token</code> <code>\c_catcode_other_token</code>	These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.
<hr/>	<hr/>

<hr/>	<hr/>
<code>\c_catcode_active_tl</code>	A token list containing an active token. This is used internally for test purposes and should not be used other than in appropriately-constructed category code tests.
<hr/>	<hr/>

4 Converting tokens

<code>\token_to_meaning:N</code>	★	<code>\token_to_meaning:N <token></code>
<code>\token_to_meaning:c</code>	★	

Inserts the current meaning of the $\langle token \rangle$ into the input stream as a series of characters of category code 12 (other). This 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`.

<code>\token_to_str:N</code>	★	<code>\token_to_str:N <token></code>
<code>\token_to_str:c</code>	★	

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

<code>\token_if_group_begin_p:N</code>	★	<code>\token_if_group_begin_p:N <token></code>
<code>\token_if_group_begin:NTF</code>	★	<code>\token_if_group_begin:NTF <token> {\true code} {\false code}</code>

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 <token></code>
<code>\token_if_group_end:NTF</code>	★	<code>\token_if_group_end:NTF <token> {\true code} {\false code}</code>

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

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

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

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

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

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

Updated: 2011-05-23

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

<code>\token_if_cs_p:N</code>	★	<code>\token_if_cs_p:N</code>	$\langle token \rangle$
<code>\token_if_cs:NNTF</code>	★	<code>\token_if_cs:NNTF</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:NNTF</code>	★	<code>\token_if_expandable:NNTF</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:NNTF</code>	★	<code>\token_if_long_macro:NNTF</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:NNTF</code>	★	<code>\token_if_protected_macro:NNTF</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:NNTF</code>	★	<code>\token_if_protected_long_macro:NNTF</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:NNTF</code>	★	<code>\token_if_chardef:NNTF</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.

\TeX hackers 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 a integer register.

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

<code>\token_if_muskip_register_p:N</code>	★	<code>\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.

`\peek_after:Nw`

`\peek_after:Nw` $\langle function \rangle$ $\langle token \rangle$

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.

`\peek_gafter:Nw`

`\peek_gafter:Nw` $\langle function \rangle$ $\langle token \rangle$

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.

`\l_peek_token`

Token set by `\peek_after:Nw` and available for testing as described above.

`\g_peek_token`

Token set by `\peek_gafter:Nw` and available for testing as described above.

`\peek_catcode:NTF`

`\peek_catcode:NTF` $\langle test token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

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).

`\peek_catcode_ignore_spaces:NTF`

`\peek_catcode_ignore_spaces:NTF` $\langle test token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-12-20

Tests if the next non-space $\langle token \rangle$ 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`). 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 left in the input stream after the $\langle true code \rangle$ or $\langle false code \rangle$ (as appropriate to the result of the test).

\peek_catcode_remove:NTF

Updated: 2012-12-20

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

Tests if the next *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the *<token>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

\peek_catcode_remove_ignore_spaces:NTF

Updated: 2012-12-20

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

Tests if the next non-space *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test `\token_if_eq_catcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

\peek_charcode:NTF

Updated: 2012-12-20

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

Tests if the next *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the *<token>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

\peek_charcode_ignore_spaces:NTF

Updated: 2012-12-20

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

Tests if the next non-space *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test `\token_if_eq_charcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

\peek_charcode_remove:NTF

Updated: 2012-12-20

`\peek_charcode_remove:NTF <test token> {<true code>} {<false code>}`

Tests if the next *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the *<token>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<code>\peek_charcode_remove_ignore_spaces:NTF</code>	<code>\peek_charcode_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 character code as the *<test token>* (as defined by the test `\token_if_eq_charcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

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

Updated: 2011-07-02

Tests if the next *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the *<token>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

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

Updated: 2012-12-05

Tests if the next non-space *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

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

Updated: 2011-07-02

Tests if the next *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the *<token>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

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

Updated: 2012-12-05

Tests if the next non-space *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

7 Decomposing a macro definition

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

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

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

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

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

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

`\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 `)`. After two expansions, `\int_eval:n` yields an *⟨integer denotation⟩* which is left in the input stream. This 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.

`\int_div_round:nn` ★ `\int_div_round:nn {⟨intexpr1⟩} {⟨intexpr2⟩}`

Updated: 2012-09-26

Evaluates the two *⟨integer expressions⟩* as described earlier, then calculates the result of dividing the first value by the second, rounding any remainder. Ties are rounded away from zero. Note that this is identical to using `/` directly in an *⟨integer expression⟩*. The result is left in the input stream as an *⟨integer denotation⟩* after two expansions.

<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 $\langle integer \text{ expressions} \rangle$ as described earlier, then calculates the result of dividing the first value by the second, truncating any remainder. Note that division using / rounds the result. The result is left in the input stream as an $\langle integer \text{ denotation} \rangle$ after two expansions.

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

<hr/> <code>\int_mod:nn</code> ★	<code>\int_mod:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-09-26	Evaluates the two $\langle integer \text{ expressions} \rangle$ as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is left in the input stream as an $\langle integer \text{ denotation} \rangle$ after two expansions.

2 Creating and initialising integers

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

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

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

<hr/> <code>\int_zero_new:N</code>	<code>\int_zero_new:N \langle integer \rangle</code>
<hr/> <code>\int_zero_new:c</code>	Ensures that the $\langle integer \rangle$ exists globally by applying <code>\int_new:N</code> if necessary, then applies <code>\int_(g)zero:N</code> to leave the $\langle integer \rangle$ set to zero.
<hr/> <code>\int_gzero_new:N</code>	
<hr/> <code>\int_gzero_new:c</code>	
New: 2011-12-13	

<hr/> <code>\int_set_eq:NN</code>	<code>\int_set_eq:NN \langle integer_1 \rangle \langle integer_2 \rangle</code>
<hr/> <code>\int_set_eq:(cN Nc cc)</code>	Sets the content of $\langle integer_1 \rangle$ equal to that of $\langle integer_2 \rangle$.
<hr/> <code>\int_gset_eq:NN</code>	
<hr/> <code>\int_gset_eq:(cN Nc cc)</code>	

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

New: 2012-03-03

3 Setting and incrementing integers

<code>\int_add:Nn</code>	<code>\int_add:Nn</code> $\langle integer \rangle$ $\{\langle integer\ expression \rangle\}$
<code>\int_add:cn</code>	
<code>\int_gadd:Nn</code>	Adds the result of the $\langle integer\ expression \rangle$ to the current content of the $\langle integer \rangle$.
<code>\int_gadd:cn</code>	

Updated: 2011-10-22

<code>\int_decr:N</code>	<code>\int_decr:N</code> $\langle integer \rangle$
<code>\int_decr:c</code>	
<code>\int_gdecr:N</code>	Decreases the value stored in $\langle integer \rangle$ by 1.
<code>\int_gdecr:c</code>	

<code>\int_incr:N</code>	<code>\int_incr:N</code> $\langle integer \rangle$
<code>\int_incr:c</code>	
<code>\int_gincr:N</code>	Increases the value stored in $\langle integer \rangle$ by 1.
<code>\int_gincr:c</code>	

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

Updated: 2011-10-22

<code>\int_sub:Nn</code>	<code>\int_sub:Nn</code> $\langle integer \rangle$ $\{\langle integer\ expression \rangle\}$
<code>\int_sub:cn</code>	
<code>\int_gsub:Nn</code>	Subtracts the result of the $\langle integer\ expression \rangle$ from the current content of the $\langle integer \rangle$.
<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 {<intexpr₁>} <relation> {<intexpr₂>} {<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,elation>} {<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: 2012-06-29

`\int_step_function:nnnN` { $\langle initial\ value \rangle$ } { $\langle step \rangle$ } { $\langle final\ value \rangle$ } $\langle function \rangle$

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. The $\langle function \rangle$ is then placed in front of each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$). Thus $\langle function \rangle$ should absorb one numerical argument. For example

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

would print

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

`\int_step_inline:nnnn`

New: 2012-06-04
Updated: 2012-06-29

`\int_step_inline:nnnn` { $\langle initial\ value \rangle$ } { $\langle step \rangle$ } { $\langle final\ value \rangle$ } { $\langle code \rangle$ }

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. The $\langle code \rangle$ is then placed in front of each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$). Thus the $\langle code \rangle$ should define a function of one argument ($\#1$).

`\int_step_variable:nnnNn`

New: 2012-06-04
Updated: 2012-06-29

`\int_step_variable:nnnNn`
{ $\langle initial\ value \rangle$ } { $\langle step \rangle$ } { $\langle final\ value \rangle$ } $\langle tl\ var \rangle$ { $\langle code \rangle$ }

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. The $\langle code \rangle$ is inserted into the input stream, with the $\langle tl\ var \rangle$ defined as the current $\langle value \rangle$. Thus the $\langle code \rangle$ should make use of the $\langle tl\ var \rangle$.

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` { $\langle integer\ expression \rangle$ }

Places the value of the $\langle integer\ expression \rangle$ in the input stream as digits, with category code 12 (other).

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

<code>\int_to_Alph:n</code> ★

Updated: 2011-09-17

Evaluates the *⟨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.

<code>\int_to_symbols:nnn</code> ★

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 }
  }
}
```

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

Updated: 2011-09-17

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

<code>\int_to_hexadecimal:n</code> ★	<code>\int_to_hexadecimal:n {⟨integer expression⟩}</code>
Updated: 2011-09-17	Calculates the value of the <i>⟨integer expression⟩</i> and places the hexadecimal (base 16) representation of the result in the input stream. Upper case letters are used for digits beyond 9.

<code>\int_to_octal:n</code> ★	<code>\int_to_octal:n {⟨integer expression⟩}</code>
Updated: 2011-09-17	Calculates the value of the <i>⟨integer expression⟩</i> and places the octal (base 8) representation of the result in the input stream.

<code>\int_to_base:nn</code> ★	<code>\int_to_base:nn {⟨integer expression⟩} {⟨base⟩}</code>
Updated: 2011-09-17	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 the upper case letters from the English alphabet. The maximum <i>⟨base⟩</i> value is 36.

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

<code>\int_to_roman:n</code> ★	<code>\int_to_roman:n {⟨integer expression⟩}</code>
<code>\int_to_Roman:n</code> ★	Places the value of the <i>⟨integer expression⟩</i> in the input stream as Roman numerals, either lower case (<code>\int_to_roman:n</code>) or upper case (<code>\int_to_Roman:n</code>). The Roman numerals are letters with category code 11 (letter).
Updated: 2011-10-22	

9 Converting from other formats to integers

<code>\int_from_alph:n</code> ★	<code>\int_from_alph:n {⟨letters⟩}</code>
	Converts the <i>⟨letters⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨letters⟩</i> are treated using the English alphabet only, with “a” equal to 1 through to “z” equal to 26. Either lower or upper case letters may be used. This is the inverse function of <code>\int_to_alph:n</code> .

<code>\int_from_binary:n</code> ★	<code>\int_from_binary:n {⟨binary number⟩}</code>
	Converts the <i>⟨binary number⟩</i> into the integer (base 10) representation and leaves this in the input stream.

<code>\int_from_hexadecimal:n</code> ★	<code>\int_from_hexadecimal:n {⟨hexadecimal number⟩}</code>
	Converts the <i>⟨hexadecimal number⟩</i> into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the <i>⟨hexadecimal number⟩</i> by upper or lower case letters.

<hr/> <code>\int_from_octal:n</code> ★ <hr/>	<code>\int_from_octal: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.
<hr/> <code>\int_from_roman:n</code> ★ <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 $\langle roman numeral \rangle$ may be in upper or lower case; if the numeral is not valid then the resulting value will be -1 .
<hr/> <code>\int_from_base:nn</code> ★ <hr/>	<code>\int_from_base:nn {\langle number \rangle} {\langle base \rangle}</code> Converts the $\langle number \rangle$ in $\langle base \rangle$ into the appropriate value in base 10. 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.

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> New: 2011-11-22 Updated: 2012-05-27 <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>	★	<code>\if_int_compare:w</code> $\langle integer_1 \rangle$ $\langle relation \rangle$ $\langle integer_2 \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code> Compare two integers using $\langle relation \rangle$, which must be one of =, < or > with category code 12. The <code>\else:</code> branch is optional.
--------------------------------	---	---

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

<code>\if_case:w</code>	★	<code>\if_case:w</code> $\langle integer \rangle$ $\langle case_0 \rangle$
<code>\or:</code>	★	$\langle case_1 \rangle$ $\langle \dots \rangle$ <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, <i>etc.</i> The $\langle integer \rangle$ may be a literal, a constant or an integer expression (<i>e.g.</i> using <code>\int_eval:n</code>).

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

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

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

14 Internal functions

<code>__int_to_roman:w</code>	★	<code>__int_to_roman:w</code> $\langle integer \rangle$ $\langle space \rangle$ or $\langle non-expandable\ token \rangle$ Converts $\langle integer \rangle$ to its lower case Roman representation. Expansion ends when a space or non-expandable token is found. Note that this function produces a string of letters with category code 12 and that protected functions <i>are</i> 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` $\langle dimension \rangle$ $\{ \langle dimension \text{ expression} \rangle \}$

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 $\langle dimension \text{ expression} \rangle$.

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` $\langle dimension \rangle$ $\{ \langle true \text{ code} \rangle \} \{ \langle false \text{ code} \rangle \}$

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 $\langle dimension\ expression \rangle$ to the current content of the $\langle dimension \rangle$.
<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 $\langle dimension \rangle$ to the value of $\langle dimension\ expression \rangle$, 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 <dimension₁> <dimension₂></code>
<code>\dim_set_eq:(cN Nc cc)</code>	
<code>\dim_gset_eq:NN</code>	Sets the content of $\langle dimension_1 \rangle$ equal to that of $\langle dimension_2 \rangle$.
<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 $\langle dimension\ expression \rangle$ from the current content of the $\langle dimension \rangle$.
<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 $\langle dimexpr \rangle$ to its absolute value, leaving the result in the input stream as a $\langle dimension\ denotation \rangle$.

<code>\dim_max:nn</code> ★	<code>\dim_max:nn {<dimexpr₁>} {<dimexpr₂>}</code>
<code>\dim_min:nn</code> ★	<code>\dim_min:nn {<dimexpr₁>} {<dimexpr₂>}</code>
New: 2012-09-09	
Updated: 2012-09-26	Evaluates the two $\langle dimension\ expressions \rangle$ and leaves either the maximum or minimum value in the input stream as appropriate, as a $\langle dimension\ denotation \rangle$.

<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> .

<code>\dim_use:N</code>	★	<code>\dim_use:N</code> $\langle dimension \rangle$
-------------------------	---	---

<code>\dim_use:c</code>	★	
-------------------------	---	--

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.

7 Viewing dim variables

<code>\dim_show:N</code>	<code>\dim_show:N</code> $\langle dimension \rangle$
--------------------------	--

<code>\dim_show:c</code>	
--------------------------	--

Displays the value of the $\langle dimension \rangle$ on the terminal.

<code>\dim_show:n</code>	<code>\dim_show:n</code> $\langle dimension expression \rangle$
--------------------------	---

New: 2011-11-22	
Updated: 2012-05-27	

Displays the result of evaluating the $\langle dimension expression \rangle$ on the terminal.

8 Constant dimensions

<code>\c_max_dim</code>	
-------------------------	--

The maximum value that can be stored as a dimension. This can also be used as a component of a skip.

<code>\c_zero_dim</code>	
--------------------------	--

A zero length as a dimension. This can also be used as a component of a skip.

9 Scratch dimensions

<code>\l_tmpa_dim</code>	
--------------------------	--

<code>\l_tmpb_dim</code>	
--------------------------	--

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.

<code>\g_tmpa_dim</code>	
--------------------------	--

<code>\g_tmpb_dim</code>	
--------------------------	--

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

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

<code>\skip_const:Nn</code>	<code>\skip_const:Nn <skip> {(skip expression)}</code>
<code>\skip_const:cn</code>	Creates a new constant $\langle skip \rangle$ or raises an error if the name is already taken. The value of the $\langle skip \rangle$ will be set globally to the $\langle skip expression \rangle$.

New: 2012-03-05

<code>\skip_zero:N</code>	<code>\skip_zero:N <skip></code>
<code>\skip_zero:c</code>	Sets $\langle skip \rangle$ to 0 pt.
<code>\skip_gzero:N</code>	
<code>\skip_gzero:c</code>	

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

New: 2012-01-07

<code>\skip_if_exist_p:N</code> *	<code>\skip_if_exist_p:N <skip></code>
<code>\skip_if_exist_p:c</code> *	<code>\skip_if_exist:Ntf <skip> {(true code)} {(false code)}</code>
<code>\skip_if_exist:Ntf</code> *	Tests whether the $\langle skip \rangle$ is currently defined. This does not check that the $\langle skip \rangle$ really is a skip variable.
<code>\skip_if_exist:cNtf</code> *	

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>	Adds the result of the $\langle skip expression \rangle$ to the current content of the $\langle skip \rangle$.
<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>	Sets $\langle skip \rangle$ to the value of $\langle skip expression \rangle$, which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm).
<code>\skip_gset:Nn</code>	
<code>\skip_gset:cn</code>	

Updated: 2011-10-22

```
\skip_set_eq:NN
\skip_set_eq:(cN|Nc|cc)
\skip_gset_eq:NN
\skip_gset_eq:(cN|Nc|cc)
```

```
\skip_set_eq:NN <skip1> <skip2>
```

Sets the content of $\langle skip_1 \rangle$ equal to that of $\langle skip_2 \rangle$.

```
\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn
```

```
\skip_sub:Nn <skip> {\skip expression}
```

Subtracts the result of the $\langle skip expression \rangle$ from the current content of the $\langle skip \rangle$.

Updated: 2011-10-22

12 Skip expression conditionals

```
\skip_if_eq_p:n  $\star$ 
\skip_if_eq:nTF  $\star$ 
```

```
\skip_if_eq_p:n {<skipexpr1>} {<skipexpr2>}
\dim_compare:nTF
  {<skipexpr1>} {<skipexpr2>}
  {\true code} {\false code}
```

This function first evaluates each of the $\langle skip expressions \rangle$ as described for `\skip_eval:n`. The two results are then compared for exact equality, *i.e.* both the fixed and rubber components must be the same for the test to be true.

```
\skip_if_finite_p:n  $\star$ 
\skip_if_finite:nTF  $\star$ 
```

New: 2012-03-05

```
\skip_if_finite_p:n {<skipexpr>}
\skip_if_finite:nTF {<skipexpr>} {\true code} {\false code}
```

Evaluates the $\langle skip expression \rangle$ as described for `\skip_eval:n`, and then tests if all of its components are finite.

13 Using skip expressions and variables

```
\skip_eval:n  $\star$ 
```

Updated: 2011-10-22

```
\skip_eval:n {\skip expression}
```

Evaluates the $\langle skip expression \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ to their content (without requiring `\skip_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle glue denotation \rangle$ after two expansions. This will be expressed in points (`\pt`), and will require suitable termination if used in a T_EX-style assignment as it is *not* an $\langle internal glue \rangle$.

<hr/> <code>\skip_use:N</code> ★	<code>\skip_use:N</code> $\langle skip \rangle$
<hr/> <code>\skip_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>\skip_eval:n</code>).

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

14 Viewing skip variables

<hr/> <code>\skip_show:N</code>	<code>\skip_show:N</code> $\langle skip \rangle$
<hr/> <code>\skip_show:c</code>	Displays the value of the $\langle skip \rangle$ on the terminal.
<hr/> <code>\skip_show:n</code>	<code>\skip_show:n</code> $\langle skip \text{ expression} \rangle$
<hr/> New: 2011-11-22 Updated: 2012-05-27	Displays the result of evaluating the $\langle skip \text{ expression} \rangle$ on the terminal.

15 Constant skips

<hr/> <code>\c_max_skip</code>	The maximum value that can be stored as a skip (equal to <code>\c_max_dim</code> in length), with no stretch nor shrink component.
<hr/> Updated: 2012-11-02	
<hr/> <code>\c_zero_skip</code>	A zero length as a skip, with no stretch nor shrink component.
<hr/> Updated: 2012-11-01	

16 Scratch skips

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

17 Inserting skips into the output

```
\skip_horizontal:N
\skip_horizontal:(c|n)
```

Updated: 2011-10-22

```
\skip_horizontal:N <skip>
\skip_horizontal:n {\<skipexpr>}
```

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|n)
```

Updated: 2011-10-22

```
\skip_vertical:N <skip>
\skip_vertical:n {\<skipexpr>}
```

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 <muskip>
```

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 <muskip> {\<muskip expression>}
```

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
```

```
\skip_zero:N <muskip>
```

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 `\muskip_new:N` if necessary, then applies `\muskip_(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

<code>\muskip_add:Nn</code>	<code>\muskip_add:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_add:cn</code>	
<code>\muskip_gadd:Nn</code>	Adds the result of the $\langle muskip expression \rangle$ to the current content of the $\langle muskip \rangle$.
<code>\muskip_gadd:cn</code>	
Updated: 2011-10-22	
<code>\muskip_set:Nn</code>	<code>\muskip_set:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_set:cn</code>	
<code>\muskip_gset:Nn</code>	Sets $\langle muskip \rangle$ to the value of $\langle muskip expression \rangle$, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu).
<code>\muskip_gset:cn</code>	
Updated: 2011-10-22	

<code>\muskip_set_eq:NN</code>	<code>\muskip_set_eq:NN <muskip₁> <muskip₂></code>
<code>\muskip_set_eq:(cN Nc cc)</code>	
<code>\muskip_gset_eq:NN</code>	Sets the content of $\langle muskip_1 \rangle$ equal to that of $\langle muskip_2 \rangle$.
<code>\muskip_gset_eq:(cN Nc cc)</code>	

<code>\muskip_sub:Nn</code>	<code>\muskip_sub:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_sub:cn</code>	
<code>\muskip_gsub:Nn</code>	Subtracts the result of the $\langle muskip expression \rangle$ from the current content of the $\langle skip \rangle$.
<code>\muskip_gsub:cn</code>	
Updated: 2011-10-22	

20 Using muskip expressions and variables

<code>\muskip_eval:n</code> ★	<code>\muskip_eval:n {<muskip expression>}</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 mu, and will require suitable termination if used in a TeX-style assignment as it is <i>not</i> an $\langle internal muglue \rangle$.
<code>\muskip_use:N</code> ★	<code>\muskip_use:N <muskip></code>
<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>).

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

21 Viewing muskip variables

<hr/> <code>\muskip_show:N</code> <hr/>	<code>\muskip_show:N <muskip></code>
<code>\muskip_show:c</code> <hr/>	Displays the value of the $\langle muskip \rangle$ on the terminal.
<hr/> <code>\muskip_show:n</code> <hr/>	<code>\muskip_show:n <muskip expression></code>
New: 2011-11-22 Updated: 2012-05-27 <hr/>	Displays the result of evaluating the $\langle muskip expression \rangle$ on the terminal.

22 Constant muskips

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

23 Scratch muskips

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

24 Primitive conditional

<hr/> <code>\if_dim:w</code> <hr/>	<code>\if_dim:w <dimen₁> <relation> <dimen₂></code> <code> <true code></code> <code>\else:</code> <code> <false></code> <code>\fi:</code>
	Compare two dimensions. The $\langle relation \rangle$ is one of $<$, $=$ or $>$ with category code 12.

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

25 Internal functions

<code>_dim_eval:w</code> ★ <code>_dim_eval_end:</code> ★	<code>_dim_eval:w</code> $\langle dimexpr \rangle$ <code>_dim_eval_end:</code>
---	--

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.

TeXhackers note: This is the ε -TeX primitive `\dimexpr`.

<code>_dim_strip_bp:n</code> ★ <code>_dim_strip_pt:n</code> ★	<code>_dim_strip_bp:n</code> $\{ \langle dimension expression \rangle \}$ <code>_dim_strip_pt:n</code> $\{ \langle dimension expression \rangle \}$
--	--

New: 2011-11-11

Evaluates the $\langle dimension expression \rangle$, expanding any dimensions and token list variables within the $\langle expression \rangle$ to their content (without requiring `\dim_use:N/\tl_use:N`) and applying the standard mathematical rules. The magnitude of the result, expressed in big points (**bp**) or points (**pt**), will be left in the input stream with *no units*. If the decimal part of the magnitude is zero, this will be omitted.

If the $\{ \langle dimension expression \rangle \}$ contains additional units, these will be ignored, so for example

`_dim_strip_pt:n { 1 bp pt }`

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

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

```
\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: 2011-12-18

```
\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. This allows the $\langle tl var \rangle$ to contain material with category codes other than those that apply when $\langle tokens \rangle$ are absorbed. See also $\backslash tl_rescan:nn$.

```
\tl_rescan:nn
```

Updated: 2011-12-18

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

Rescans $\langle tokens \rangle$ applying the category code régime specified in the $\langle setup \rangle$, and leaves the resulting tokens in the input stream. See also $\backslash tl_set_rescan:Nnn$.

5 Reassigning token list character codes

```
\tl_to_lowercase:n
```

Updated: 2012-09-08

```
\tl_to_lowercase:n {<tokens>}
```

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

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

\tl_to_uppercase:nUpdated: 2012-09-08

\tl_to_uppercase:n $\{\langle tokens \rangle\}$

Works through all of the $\langle tokens \rangle$, 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 $\langle tokens \rangle$.

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

6 Token list conditionals

\tl_if_blank_p:n ***\tl_if_blank_p:(V|o)** ***\tl_if_blank:nTF** ***\tl_if_blank:(V|o)TF** ***\tl_if_blank_p:n** $\{\langle token list \rangle\}$ **\tl_if_blank:nTF** $\{\langle token list \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

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

\tl_if_empty_p:N ***\tl_if_empty_p:c** ***\tl_if_empty:NTF** ***\tl_if_empty:cTF** ***\tl_if_empty_p:N** $\langle tl var \rangle$ **\tl_if_empty:NTF** $\langle tl var \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle token list variable \rangle$ is entirely empty (*i.e.* contains no tokens at all).

\tl_if_empty_p:n ***\tl_if_empty_p:(V|o)** ***\tl_if_empty:nTF** ***\tl_if_empty:(V|o)TF** ***\tl_if_empty_p:n** $\{\langle token list \rangle\}$ **\tl_if_empty:nTF** $\{\langle token list \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle token list \rangle$ is entirely empty (*i.e.* contains no tokens at all).

New: 2012-05-24

Updated: 2012-06-05

\tl_if_eq_p:NN ***\tl_if_eq_p:(Nc|cN|cc)** ***\tl_if_eq:NNTF** ***\tl_if_eq:(Nc|cN|cc)TF** ***\tl_if_eq_p:NN** $\{\langle tl var_1 \rangle\}$ $\{\langle tl var_2 \rangle\}$ **\tl_if_eq:NNTF** $\{\langle tl var_1 \rangle\}$ $\{\langle tl var_2 \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Compares the content of two $\langle token list variables \rangle$ and is logically **true** if the two contain the same list of tokens (*i.e.* identical in both the list of characters they contain and the category codes of those characters). Thus for example

```
\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Nx \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq:NNTF \l_tmpa_tl \l_tmpb_tl { true } { false }
```

yields **false**.

\tl_if_eq:nnTF**\tl_if_eq:nnTF** $\langle token list_1 \rangle$ $\{\langle token list_2 \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token list_1 \rangle$ and $\langle token list_2 \rangle$ contain the same list of tokens, both in respect of character codes and category codes.

<code>\tl_if_in:NnTF</code> <code>\tl_if_in:cnTF</code>	<code>\tl_if_in:NnTF <tl var> {<token list>} {<true code>} {<false code>}</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:(Vn on no)TF</code>	<code>\tl_if_in:nnTF {<token list₁>} {<token list₂>} {<true code>} {<false code>}</code>
--	--

Tests if *<token list₂>* is found inside *<token list₁>*. The *<token list₂>* cannot contain the tokens `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

<code>\tl_if_single_p:N</code> ★ <code>\tl_if_single_p:c</code> ★ <code>\tl_if_single:NnTF</code> ★ <code>\tl_if_single:cnTF</code> ★	<code>\tl_if_single_p:N <tl var></code> <code>\tl_if_single:NnTF <tl var> {<true code>} {<false code>}</code>
--	--

Updated: 2011-08-13

Tests if the content of the *<tl var>* consists of a single item, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\tl_count:N`.

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

New: 2013-07-24

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

<hr/> <code>\tl_map_function:NN</code> ☆ <code>\tl_map_function:cN</code> ☆ <hr/> Updated: 2012-06-29	<code>\tl_map_function:NN</code> $\langle tl\ var \rangle$ $\langle function \rangle$ Applies $\langle function \rangle$ to every $\langle item \rangle$ in the $\langle tl\ var \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. This may be a number of tokens if the $\langle item \rangle$ was stored within braces. Hence the $\langle function \rangle$ should anticipate receiving n-type arguments. See also <code>\tl_map_function:nN</code> .
<hr/> <code>\tl_map_function:nN</code> ☆ <hr/> Updated: 2012-06-29	<code>\tl_map_function:nN</code> $\langle token\ list \rangle$ $\langle function \rangle$ Applies $\langle function \rangle$ to every $\langle item \rangle$ in the $\langle token\ list \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. This may be a number of tokens if the $\langle item \rangle$ was stored within braces. Hence the $\langle function \rangle$ should anticipate receiving n-type arguments. See also <code>\tl_map_function:NN</code> .
<hr/> <code>\tl_map_inline:Nn</code> <code>\tl_map_inline:cN</code> <hr/> Updated: 2012-06-29	<code>\tl_map_inline:Nn</code> $\langle tl\ var \rangle$ $\{\langle inline\ function \rangle\}$ Applies the $\langle inline\ function \rangle$ to every $\langle item \rangle$ stored within the $\langle tl\ var \rangle$. 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. See also <code>\tl_map_function:NN</code> .
<hr/> <code>\tl_map_inline:nn</code> <hr/> Updated: 2012-06-29	<code>\tl_map_inline:nn</code> $\langle token\ list \rangle$ $\{\langle inline\ function \rangle\}$ Applies the $\langle inline\ function \rangle$ to every $\langle item \rangle$ stored within the $\langle token\ list \rangle$. 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. See also <code>\tl_map_function:nN</code> .
<hr/> <code>\tl_map_variable:NNn</code> <code>\tl_map_variable:cNn</code> <hr/> Updated: 2012-06-29	<code>\tl_map_variable:NNn</code> $\langle tl\ var \rangle$ $\langle variable \rangle$ $\{\langle function \rangle\}$ Applies the $\langle function \rangle$ to every $\langle item \rangle$ stored within the $\langle tl\ var \rangle$. The $\langle function \rangle$ should consist of code which will receive the $\langle item \rangle$ stored in the $\langle variable \rangle$. One variable mapping can be nested inside another. See also <code>\tl_map_inline:Nn</code> .
<hr/> <code>\tl_map_variable:nNn</code> <hr/> Updated: 2012-06-29	<code>\tl_map_variable:nNn</code> $\langle token\ list \rangle$ $\langle variable \rangle$ $\{\langle function \rangle\}$ Applies the $\langle function \rangle$ to every $\langle item \rangle$ stored within the $\langle token\ list \rangle$. The $\langle function \rangle$ should consist of code which will receive the $\langle item \rangle$ stored in the $\langle variable \rangle$. One variable mapping can be nested inside another. See also <code>\tl_map_inline:nn</code> .

\tl_map_break: ☆

Updated: 2012-06-29

\tl_map_break:

Used to terminate a `\tl_map...` function before all entries in the *⟨token list variable⟩* have been processed. This will normally take place within a conditional statement, for example

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

See also `\tl_map_break:n`. Use outside of a `\tl_map...` scenario will lead to low level TeX errors.

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

\tl_map_break:n ☆

Updated: 2012-06-29

\tl_map_break:n {*⟨tokens⟩*}

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

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

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

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

8 Using token lists

\tl_to_str:N ☆

\tl_to_str:c ☆**\tl_to_str:N** *⟨tl var⟩*

Converts the content of the *⟨tl var⟩* into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This *⟨string⟩* is then left in the input stream.

<code>\tl_to_str:n</code>	★	<code>\tl_to_str:n {(tokens)}</code>
---------------------------	---	--------------------------------------

Converts the given $\langle tokens \rangle$ 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. Note that this function requires only a single expansion.

TeXhackers note: This is the ε -TeX primitive `\detokenize`. Hence its argument *must* be given within braces.

<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 $\{ \dots \}$. 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 $\{ \dots \}$. 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`.

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.

<code>\tl_reverse:N</code>	<code>\tl_reverse:N <tl var></code>
<code>\tl_reverse:c</code>	
<code>\tl_greverse:N</code>	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 <code>\tl_reverse:n</code> , and, for improved performance, <code>\tl_reverse_items:n</code> .
<code>\tl_greverse:c</code>	
Updated: 2012-01-08	

<code>\tl_reverse_items:n</code> ★	<code>\tl_reverse_items:n {\langle token list \rangle}</code>
New: 2012-01-08	Reverses the order of the $\langle items \rangle$ stored in $\langle tl var \rangle$, so that $\{\langle item_1 \rangle\} \{\langle item_2 \rangle\} \{\langle item_3 \rangle\} \dots \{\langle item_n \rangle\}$ becomes $\{\langle item_n \rangle\} \dots \{\langle item_3 \rangle\} \{\langle item_2 \rangle\} \{\langle item_1 \rangle\}$. This process 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 `\exp_not:n`, which means that the token list will not expand further when appearing in an `x`-type argument expansion.

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

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

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 {⟨token list⟩}</code>
<code>\tl_head:(n V v f)</code> ★	Leaves in the input stream the first <i>⟨item⟩</i> in the <i>⟨token list⟩</i> , discarding the rest of the <i>⟨token list⟩</i> . All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example
Updated: 2012-09-09	

`\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> ★	<code>\tl_head:w ⟨token list⟩ { } \q_stop</code>
	Leaves in the input stream the first <i>⟨item⟩</i> in the <i>⟨token list⟩</i> , discarding the rest of the <i>⟨token list⟩</i> . All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank <i>⟨token list⟩</i> (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, <code>\tl_if_blank:nF</code> 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 <i>o</i> -type expansion. In general, <code>\tl_head:n</code> should be preferred if the number of expansions is not critical.

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

`\tl_tail:(n|V|v|f)` ★

Updated: 2012-09-01

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first *⟨item⟩* in the *⟨token list⟩*, and leaves the remaining tokens in the input stream. Thus for example

`\tl_tail:n { a ~ {bc} d }`

and

`\tl_tail:n { ~ a ~ {bc} d }`

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

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

`\str_head:n` ★ `\str_head:n {⟨token list⟩}`

`\str_tail:n` ★ `\str_tail:n {⟨token list⟩}`

New: 2011-08-10

Converts the *⟨token list⟩* into a string, as described for `\tl_to_str:n`. The `\str_head:n` function then leaves the first character of this string in the input stream. The `\str_tail:n` function leaves all characters except the first in the input stream. The first character may be a space. If the *⟨token list⟩* argument is entirely empty, nothing is left in the input stream.

`\tl_if_head_eq_catcode_p:nN` ★ `\tl_if_head_eq_catcode_p:nN {⟨token list⟩} ⟨test token⟩`

`\tl_if_head_eq_catcode:nNTF` ★ `\tl_if_head_eq_catcode:nNTF {⟨token list⟩} ⟨test token⟩`
`{⟨true code⟩} {⟨false code⟩}`

Updated: 2012-07-09

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

`\tl_if_head_eq_charcode_p:nN` ★ `\tl_if_head_eq_charcode_p:nN {⟨token list⟩} ⟨test token⟩`

`\tl_if_head_eq_charcode_p:fN` ★ `\tl_if_head_eq_charcode:nNTF {⟨token list⟩} ⟨test token⟩`

`\tl_if_head_eq_charcode:nNTF` ★ `{⟨true code⟩} {⟨false code⟩}`

`\tl_if_head_eq_charcode:fNTF` ★

Updated: 2012-07-09

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

`\tl_if_head_eq_meaning_p:nN` ★ `\tl_if_head_eq_meaning_p:nN {⟨token list⟩} ⟨test token⟩`

`\tl_if_head_eq_meaning:nNTF` ★ `\tl_if_head_eq_meaning:nNTF {⟨token list⟩} ⟨test token⟩`

`{⟨true code⟩} {⟨false code⟩}`

Updated: 2012-07-09

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

<code>\tl_if_head_is_group_p:n</code> ★	<code>\tl_if_head_is_group_p:n {⟨token list⟩}</code>
<code>\tl_if_head_is_group:nTF</code> ★	<code>\tl_if_head_is_group:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>
New: 2012-07-08	Tests if the first <i>⟨token⟩</i> in the <i>⟨token list⟩</i> is an explicit begin-group character (with category code 1 and any character code), in other words, if the <i>⟨token list⟩</i> starts with a brace group. In particular, the test is false if the <i>⟨token list⟩</i> starts with an implicit token such as <code>\c_group_begin_token</code> , 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 <i>⟨token⟩</i> in the <i>⟨token list⟩</i> is an explicit space character (explicit token with character code 12 and category code 10). In particular, the test is false if the <i>⟨token list⟩</i> starts with an implicit token such as <code>\c_space_token</code> , or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

11 Viewing token lists

<code>\tl_show:N</code>	<code>\tl_show:N ⟨tl var⟩</code>
<code>\tl_show:c</code>	Displays the content of the <i>⟨tl var⟩</i> on the terminal.
Updated: 2012-09-09	T_EXhackers note: This is similar to the T _E X primitive <code>\show</code> , wrapped to a fixed number of characters per line.

<code>\tl_show:n</code>	<code>\tl_show:n ⟨token list⟩</code>
Updated: 2012-09-09	Displays the <i>⟨token list⟩</i> on the terminal.
	T_EXhackers note: This is similar to the ε-T _E X primitive <code>\showtokens</code> , wrapped to a fixed number of characters per line.

12 Constant token lists

<hr/> <hr/> <code>\c_empty_tl</code> <hr/> <hr/>	Constant that is always empty.
<hr/> <hr/> <code>\c_job_name_tl</code> <hr/> <hr/>	Constant that gets the “job name” assigned when TeX starts.
<hr/> <hr/> <code>Updated: 2011-08-18</code> <hr/> <hr/>	TeXhackers note: This copies the contents of the primitive <code>\jobname</code> . It is a constant that is set by TeX and should not be overwritten by the package.
<hr/> <hr/> <code>\c_space_tl</code> <hr/> <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.

13 Scratch token lists

<hr/> <hr/> <code>\l_tmpa_tl</code> <code>\l_tmpb_tl</code> <hr/> <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/> <hr/> <code>\g_tmpa_tl</code> <code>\g_tmpb_tl</code> <hr/> <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.

14 Internal functions

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

Part XII

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

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

`\seq_new:N` *⟨sequence⟩*

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

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

`\seq_clear:N` *⟨sequence⟩*

Clears all items from the *⟨sequence⟩*.

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

`\seq_clear_new:N` *⟨sequence⟩*

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

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

`\seq_set_eq:NN` *⟨sequence₁⟩* *⟨sequence₂⟩*

Sets the content of *⟨sequence₁⟩* equal to that of *⟨sequence₂⟩*.

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

`\seq_set_split:Nnn` *⟨sequence⟩* *{⟨delimiter⟩}* *{⟨token list⟩}*

Splits the *⟨token list⟩* into *⟨items⟩* separated by *⟨delimiter⟩*, and assigns the result to the *⟨sequence⟩*. Spaces on both sides of each *⟨item⟩* are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of l3clist functions. Empty *⟨items⟩* are preserved by `\seq_set_split:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn` *⟨sequence⟩* *{⟨⟩}*. The *⟨delimiter⟩* may not contain `{`, `}` or `#` (assuming T_EX’s normal category code régime). If the *⟨delimiter⟩* is empty, the *⟨token list⟩* is split into *⟨items⟩* as a *⟨token list⟩*.

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

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

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

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.

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

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

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

New: 2012-03-03

2 Appending data to sequences

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

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

Appends the $\langle item \rangle$ to the left of the $\langle sequence \rangle$.

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

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

Appends the $\langle item \rangle$ 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`.

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

Updated: 2012-05-14

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

Stores the left-most item from a $\langle sequence \rangle$ 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 `\q_no_value`.

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

Updated: 2012-05-19

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

Stores the right-most item from a $\langle sequence \rangle$ 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 `\q_no_value`.

`\seq_pop_left:NN`
`\seq_pop_left:cN`
 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.e.* removes the item from the sequence and stores it in the $\langle token list variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ will contain the special marker `\q_no_value`.

`\seq_gpop_left:NN`
`\seq_gpop_left:cN`
 Updated: 2012-05-14

`\seq_gpop_left:NN` $\langle sequence \rangle$ $\langle token list variable \rangle$

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

`\seq_pop_right:NN`
`\seq_pop_right:cN`
 Updated: 2012-05-19

`\seq_pop_right:NN` $\langle sequence \rangle$ $\langle token list variable \rangle$

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

`\seq_gpop_right:NN`
`\seq_gpop_right:cN`
 Updated: 2012-05-19

`\seq_gpop_right:NN` $\langle sequence \rangle$ $\langle token list variable \rangle$

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

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.

<code>\seq_pop_left:NNTF</code> <code>\seq_pop_left:cNTF</code>	<code>\seq_pop_left:NNTF <sequence> <token list variable> {\true code} {\false code}</code>
<p style="margin: 0;">New: 2012-05-14</p> <p style="margin: 0;">Updated: 2012-05-19</p>	<p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, pops the left-most item from a <i><sequence></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from a <i><sequence></i>. Both the <i><sequence></i> and the <i><token list variable></i> are assigned locally.</p>

<code>\seq_gpop_left:NNTF</code> <code>\seq_gpop_left:cNTF</code>	<code>\seq_gpop_left:NNTF <sequence> <token list variable> {\true code} {\false code}</code>
<p style="margin: 0;">New: 2012-05-14</p> <p style="margin: 0;">Updated: 2012-05-19</p>	<p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, pops the left-most item from a <i><sequence></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from a <i><sequence></i>. The <i><sequence></i> is modified globally, while the <i><token list variable></i> is assigned locally.</p>

<code>\seq_pop_right:NNTF</code> <code>\seq_pop_right:cNTF</code>	<code>\seq_pop_right:NNTF <sequence> <token list variable> {\true code} {\false code}</code>
<p style="margin: 0;">New: 2012-05-19</p>	<p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, pops the right-most item from a <i><sequence></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from a <i><sequence></i>. Both the <i><sequence></i> and the <i><token list variable></i> are assigned locally.</p>

<code>\seq_gpop_right:NNTF</code> <code>\seq_gpop_right:cNTF</code>	<code>\seq_gpop_right:NNTF <sequence> <token list variable> {\true code} {\false code}</code>
<p style="margin: 0;">New: 2012-05-19</p>	<p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, pops the right-most item from a <i><sequence></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from a <i><sequence></i>. The <i><sequence></i> is modified globally, while the <i><token list variable></i> is assigned locally.</p>

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.

<code>\seq_remove_duplicates:N</code> <code>\seq_remove_duplicates:c</code> <code>\seq_gremove_duplicates:N</code> <code>\seq_gremove_duplicates:c</code>	<code>\seq_remove_duplicates:N <sequence></code>
	<p>Removes duplicate items from the <i><sequence></i>, leaving the left most copy of each item in the <i><sequence></i>. The <i><item></i> comparison takes place on a token basis, as for <code>\tl_if_eq:nn(TF)</code>.</p>

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

<hr/> <code>\seq_remove_all:Nn</code>	<code>\seq_remove_all:Nn <sequence> {<item>}</code>
<code>\seq_remove_all:cn</code>	
<code>\seq_gremove_all:Nn</code>	Removes every occurrence of <code><item></code> from the <code><sequence></code> . The <code><item></code> comparison takes place on a token basis, as for <code>\tl_if_eq:nn(TF)</code> .
<code>\seq_gremove_all:cn</code>	

6 Sequence conditionals

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

<hr/> <code>\seq_if_in:NnTF</code>	<code>\seq_if_in:NnTF <sequence> {<item>} {<true code>} {<false code>}</code>
<code>\seq_if_in:(NV Nv No Nx cn cV cv co cx)TF</code>	

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

7 Mapping to sequences

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

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

<hr/> <code>\seq_map_variable:NNn</code>	<code>\seq_map_variable:NNn <sequence> <tl var.> {<function using tl var.>}</code>
<code>\seq_map_variable:(Ncn cN ccn)</code>	
Updated: 2012-06-29	

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

`\seq_map_break:` ☆

Updated: 2012-06-29

`\seq_map_break:`

Used to terminate a `\seq_map...` function before all entries in the $\langle sequence \rangle$ 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 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.

`\seq_map_break:n` ☆

Updated: 2012-06-29

`\seq_map_break:n {<tokens>}`

Used to terminate a `\seq_map...` function before all entries in the $\langle sequence \rangle$ 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 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.

`\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 Constant and scratch sequences

`\c_empty_seq` Constant that is always empty.

New: 2012-07-02

`\l_tmpa_seq` Scratch sequences for local assignment. These are never used by the kernel code, and so
`\l_tmpb_seq` 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

`\g_tmpa_seq` Scratch sequences for global assignment. These are never used by the kernel code, and
`\g_tmpb_seq` 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

11 Viewing sequences

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

`\seq_show:c`

Displays the entries in the $\langle sequence \rangle$ in the terminal.

Updated: 2012-09-09

12 Internal sequence functions

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

`__seq_item:n` ★ `__seq_item:n` $\{\langle item \rangle\}$

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.

`__seq_push_item_def:n` `__seq_push_item_def:n` $\{\langle code \rangle\}$

`__seq_push_item_def:x`

Saves the definition of `__seq_item:n` and redefines it to accept one parameter and expand to $\langle code \rangle$. This function should always be balanced by use of `__seq_pop_item_def:`.

`__seq_pop_item_def:` `__seq_pop_item_def:`

Restores the definition of `__seq_item:n` most recently saved by `__seq_push_item_def:n`. This function should always be used in a balanced pair with `__seq_push_item_def:n`.

Part XIII

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

<code>\clist_new:N</code>	<code>\clist_new:N <comma list></code>
<code>\clist_new:c</code>	

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

<code>\clist_clear:N</code>	<code>\clist_clear:N <comma list></code>
<code>\clist_clear:c</code>	
<code>\clist_gclear:N</code>	Clears all items from the <i><comma list></i> .
<code>\clist_gclear:c</code>	

<code>\clist_clear_new:N</code>	<code>\clist_clear_new:N <comma list></code>
<code>\clist_clear_new:c</code>	
<code>\clist_gclear_new:N</code>	Ensures that the <i><comma list></i> 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.

```
\clist_set_eq:NN
\clist_set_eq:(cN|Nc|cc)
\clist_gset_eq:NN
\clist_gset_eq:(cN|Nc|cc)
```

```
\clist_set_eq:NN <comma list1> <comma list2>
```

Sets the content of $\langle comma list_1 \rangle$ equal to that of $\langle comma list_2 \rangle$.

```
\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc
```

```
\clist_concat:NNN <comma list1> <comma list2> <comma list3>
```

Concatenates the content of $\langle comma list_2 \rangle$ and $\langle comma list_3 \rangle$ together and saves the 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.

```
\clist_if_exist_p:N ★
\clist_if_exist_p:c ★
\clist_if_exist:NTF ★
\clist_if_exist:cTF ★
```

```
\clist_if_exist_p:N <comma list>
\clist_if_exist:NNTF <comma list> {\true code} {\false code}
```

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

New: 2012-03-03

2 Adding data to comma lists

```
\clist_set:Nn
\clist_set:(NV|No|Nx|cn|cV|co|cx)
\clist_gset:Nn
\clist_gset:(NV|No|Nx|cn|cV|co|cx)
```

New: 2011-09-06

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

```
\clist_put_left:Nn
\clist_put_left:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_left:Nn
\clist_gput_left:(NV|No|Nx|cn|cV|co|cx)
```

Updated: 2011-09-05

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

```
\clist_put_right:Nn
\clist_put_right:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_right:Nn
\clist_gput_right:(NV|No|Nx|cn|cV|co|cx)
```

Updated: 2011-09-05

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 *<comma list>*, leaving the left most copy of each item in the *<comma list>*. The *<item>* comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

T_EXhackers note: This function iterates through every item in the *<comma list>* and does a comparison with the *<items>* already checked. It is therefore relatively slow with large comma lists. Furthermore, it will not work if any of the items in the *<comma list>* contains `{`, `}`, or `#` (assuming the usual 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>	

Removes every occurrence of *<item>* from the *<comma list>*. The *<item>* comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

Updated: 2011-09-06

T_EXhackers note: The *<item>* may not contain `{`, `}`, or `#` (assuming the usual T_EX category codes apply).

4 Comma list conditionals

<code>\clist_if_empty_p:N</code> ★	<code>\clist_if_empty_p:N <comma list></code>
<code>\clist_if_empty_p:c</code> ★	<code>\clist_if_empty:NNTF <comma list> {<true code>} {<false code>}</code>
<code>\clist_if_empty:NTF</code> ★	
<code>\clist_if_empty:cTF</code> ★	Tests if the <i><comma list></i> is empty (containing no items).

<code>\clist_if_in:NnTF</code> <code>\clist_if_in:(NV No cn cV co nn nV no)TF</code>	<code>\clist_if_in:NnTF <comma list> {<item>} {<true code>} {<false code>}</code>
---	---

Updated: 2011-09-06

Tests if the $\langle item \rangle$ is present in the $\langle comma list \rangle$. In the case of an **n**-type $\langle comma list \rangle$, 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 $\langle item \rangle$ 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.

<code>\clist_map_function:NN</code> <code>\clist_map_function:(cN nN)</code>	<code>\clist_map_function:NN <comma list> <function></code>
---	---

Updated: 2012-06-29

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.

<code>\clist_map_inline:Nn</code> <code>\clist_map_inline:(cn nn)</code>	<code>\clist_map_inline:Nn <comma list> {<inline function>}</code>
---	--

Updated: 2012-06-29

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.

<code>\clist_map_variable:Nn</code>	<code>\clist_map_variable:Nn <comma list> <tl var.> {<function using tl var.>}</code>
<code>\clist_map_variable:(cNn nNn)</code>	

Updated: 2012-06-29

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

<code>\clist_map_break: ☆</code>	<code>\clist_map_break:</code>
----------------------------------	--------------------------------

Updated: 2012-06-29

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 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.

`\clist_map_break:n` ☆

Updated: 2012-06-29

`\clist_map_break:n` { \langle tokens \rangle }

Used to terminate a `\clist_map...` function before all entries in the \langle comma list \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

```
\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 TeX errors.

TeXhackers 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.

`\clist_count:N` ☆

`\clist_count:(c|n)` ☆

New: 2012-07-13

`\clist_count:N` \langle comma list \rangle

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

6 Using the content of comma lists directly

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

New: 2013-05-26

Places the contents of the $\langle\textit{clist var}\rangle$ in the input stream, with the appropriate $\langle\textit{separator}\rangle$ between the items. Namely, if the comma list has more than two items, the $\langle\textit{separator between more than two}\rangle$ is placed between each pair of items except the last, for which the $\langle\textit{separator between final two}\rangle$ is used. If the comma list has exactly two items, then they are placed in the input stream separated by the $\langle\textit{separator between two}\rangle$. 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 $\langle\textit{items}\rangle$ will not expand further when appearing in an `x`-type argument expansion.

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

New: 2013-05-26

Places the contents of the $\langle\textit{clist var}\rangle$ in the input stream, with the $\langle\textit{separator}\rangle$ between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error 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 $\langle\textit{items}\rangle$ 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.

<u>\clist_get:NN</u> <u>\clist_get:cN</u> Updated: 2012-05-14	<p>\clist_get:NN <i><comma list></i> <i><token list variable></i></p> <p>Stores the left-most item from a <i><comma list></i> in the <i><token list variable></i> without removing it from the <i><comma list></i>. The <i><token list variable></i> is assigned locally. If the <i><comma list></i> is empty the <i><token list variable></i> will contain the marker value <code>\q_no_value</code>.</p>
<u>\clist_get:NNTF</u> <u>\clist_get:cNTF</u> New: 2012-05-14	<p>\clist_get:NNTF <i><comma list></i> <i><token list variable></i> <i>{<true code>}</i> <i>{<false code>}</i></p> <p>If the <i><comma list></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><comma list></i> is non-empty, stores the top item from the <i><comma list></i> in the <i><token list variable></i> without removing it from the <i><comma list></i>. The <i><token list variable></i> is assigned locally.</p>
<u>\clist_pop:NN</u> <u>\clist_pop:cN</u> Updated: 2011-09-06	<p>\clist_pop:NN <i><comma list></i> <i><token list variable></i></p> <p>Pops the left-most item from a <i><comma list></i> into the <i><token list variable></i>, <i>i.e.</i> removes the item from the comma list and stores it in the <i><token list variable></i>. Both of the variables are assigned locally.</p>
<u>\clist_gpop:NN</u> <u>\clist_gpop:cN</u>	<p>\clist_gpop:NN <i><comma list></i> <i><token list variable></i></p> <p>Pops the left-most item from a <i><comma list></i> into the <i><token list variable></i>, <i>i.e.</i> removes the item from the comma list and stores it in the <i><token list variable></i>. The <i><comma list></i> is modified globally, while the assignment of the <i><token list variable></i> is local.</p>
<u>\clist_pop:NNTF</u> <u>\clist_pop:cNTF</u> New: 2012-05-14	<p>\clist_pop:NNTF <i><sequence></i> <i><token list variable></i> <i>{<true code>}</i> <i>{<false code>}</i></p> <p>If the <i><comma list></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><comma list></i> is non-empty, pops the top item from the <i><comma list></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from the <i><comma list></i>. Both the <i><comma list></i> and the <i><token list variable></i> are assigned locally.</p>
<u>\clist_gpop:NNTF</u> <u>\clist_gpop:cNTF</u> New: 2012-05-14	<p>\clist_gpop:NNTF <i><comma list></i> <i><token list variable></i> <i>{<true code>}</i> <i>{<false code>}</i></p> <p>If the <i><comma list></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><comma list></i> is non-empty, pops the top item from the <i><comma list></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from the <i><comma list></i>. The <i><comma list></i> is modified globally, while the <i><token list variable></i> is assigned locally.</p>

<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 `{<items>}` to the top of the `<comma list>`. Spaces are removed from both sides of each item.

8 Viewing comma lists

<code>\clist_show:N</code>	<code>\clist_show:N <comma list></code>
<code>\clist_show:c</code>	Displays the entries in the <code><comma list></code> in the terminal.
Updated: 2012-09-09	

<code>\clist_show:n</code>	<code>\clist_show:n {<tokens>}</code>
Updated: 2012-09-09	Displays the entries in the comma list in the terminal.

9 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>	Scratch comma lists for local assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
New: 2011-09-06	

<code>\g_tmpa_clist</code> <code>\g_tmpb_clist</code>	Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
New: 2011-09-06	

Part XIV

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

4 Modifying property lists

```
\prop_remove:Nn
\prop_remove:(NV|cn|cV)
\prop_gremove:Nn
\prop_gremove:(NV|cn|cV)
```

New: 2012-05-12

```
\prop_remove:Nn <property list> {<key>}
```

Removes the entry listed under $\langle key \rangle$ from the $\langle property list \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ no change occurs, *i.e* there is no need to test for the existence of a key before deleting it.

5 Property list conditionals

```
\prop_if_exist_p:N *
\prop_if_exist_p:c *
\prop_if_exist:NTF *
\prop_if_exist:cTF *
```

New: 2012-03-03

```
\prop_if_exist_p:N <property list>
```

```
\prop_if_exist:NTF <property list> {<true code>} {<false code>}
```

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

```
\prop_if_empty_p:N *
\prop_if_empty_p:c *
\prop_if_empty:NTF *
\prop_if_empty:cTF *
```

```
\prop_if_empty_p:N <property list>
```

```
\prop_if_empty:NTF <property list> {<true code>} {<false code>}
```

Tests if the $\langle property list \rangle$ is empty (containing no entries).

```
\prop_if_in_p:Nn *
\prop_if_in_p:(NV|No|cn|cV|co) *
\prop_if_in:NnTF *
\prop_if_in:(NV|No|cn|cV|co)TF *
```

Updated: 2011-09-15

```
\prop_if_in:NnTF <property list> {<key>} {<true code>} {<false code>}
```

Tests if the $\langle key \rangle$ 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.

`\prop_get:NnNTF`
`\prop_get:(NVN|NoN|cnN|cVN|coN)TF`

Updated: 2012-05-19

`\prop_get:NnNTF` $\langle\textit{property list}\rangle$ $\{\langle\textit{key}\rangle\}$ $\langle\textit{token list variable}\rangle$
 $\{\langle\textit{true code}\rangle\}$ $\{\langle\textit{false code}\rangle\}$

If the $\langle\textit{key}\rangle$ is not present in the $\langle\textit{property list}\rangle$, leaves the $\langle\textit{false code}\rangle$ in the input stream. The value of the $\langle\textit{token list variable}\rangle$ is not defined in this case and should not be relied upon. If the $\langle\textit{key}\rangle$ is present in the $\langle\textit{property list}\rangle$, stores the corresponding $\langle\textit{value}\rangle$ in the $\langle\textit{token list variable}\rangle$ without removing it from the $\langle\textit{property list}\rangle$, then leaves the $\langle\textit{true code}\rangle$ in the input stream. The $\langle\textit{token list variable}\rangle$ is assigned locally.

`\prop_pop:NnNTF`
`\prop_pop:cnNTF`

New: 2011-08-18
Updated: 2012-05-19

`\prop_pop:NnNTF` $\langle\textit{property list}\rangle$ $\{\langle\textit{key}\rangle\}$ $\langle\textit{token list variable}\rangle$ $\{\langle\textit{true code}\rangle\}$
 $\{\langle\textit{false code}\rangle\}$

If the $\langle\textit{key}\rangle$ is not present in the $\langle\textit{property list}\rangle$, leaves the $\langle\textit{false code}\rangle$ in the input stream. The value of the $\langle\textit{token list variable}\rangle$ is not defined in this case and should not be relied upon. If the $\langle\textit{key}\rangle$ is present in the $\langle\textit{property list}\rangle$, pops the corresponding $\langle\textit{value}\rangle$ in the $\langle\textit{token list variable}\rangle$, *i.e.* removes the item from the $\langle\textit{property list}\rangle$. Both the $\langle\textit{property list}\rangle$ and the $\langle\textit{token list variable}\rangle$ are assigned locally.

`\prop_gpop:NnNTF`
`\prop_gpop:cnNTF`

New: 2011-08-18
Updated: 2012-05-19

`\prop_gpop:NnNTF` $\langle\textit{property list}\rangle$ $\{\langle\textit{key}\rangle\}$ $\langle\textit{token list variable}\rangle$ $\{\langle\textit{true code}\rangle\}$
 $\{\langle\textit{false code}\rangle\}$

If the $\langle\textit{key}\rangle$ is not present in the $\langle\textit{property list}\rangle$, leaves the $\langle\textit{false code}\rangle$ in the input stream. The value of the $\langle\textit{token list variable}\rangle$ is not defined in this case and should not be relied upon. If the $\langle\textit{key}\rangle$ is present in the $\langle\textit{property list}\rangle$, pops the corresponding $\langle\textit{value}\rangle$ in the $\langle\textit{token list variable}\rangle$, *i.e.* removes the item from the $\langle\textit{property list}\rangle$. The $\langle\textit{property list}\rangle$ is modified globally, while the $\langle\textit{token list variable}\rangle$ is assigned locally.

7 Mapping to property lists

`\prop_map_function:NN` ☆
`\prop_map_function:cn` ☆

Updated: 2013-01-08

`\prop_map_function:NN` $\langle\textit{property list}\rangle$ $\langle\textit{function}\rangle$

Applies $\langle\textit{function}\rangle$ to every $\langle\textit{entry}\rangle$ stored in the $\langle\textit{property list}\rangle$. The $\langle\textit{function}\rangle$ will receive two argument for each iteration: the $\langle\textit{key}\rangle$ and associated $\langle\textit{value}\rangle$. The order in which $\langle\textit{entries}\rangle$ are returned is not defined and should not be relied upon.

`\prop_map_inline:Nn`
`\prop_map_inline:cn`

Updated: 2013-01-08

`\prop_map_inline:Nn` $\langle\textit{property list}\rangle$ $\{\langle\textit{inline function}\rangle\}$

Applies $\langle\textit{inline function}\rangle$ to every $\langle\textit{entry}\rangle$ stored within the $\langle\textit{property list}\rangle$. The $\langle\textit{inline function}\rangle$ should consist of code which will receive the $\langle\textit{key}\rangle$ as #1 and the $\langle\textit{value}\rangle$ as #2. The order in which $\langle\textit{entries}\rangle$ are returned is not defined and should not be relied upon.

\prop_map_break: ☆

Updated: 2012-06-29

\prop_map_break:

Used to terminate a `\prop_map...` function before all entries in the *⟨property list⟩* have been processed. This will normally take place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break: }
  {
    % Do something useful
  }
}
```

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

\prop_map_break:n ☆

Updated: 2012-06-29

\prop_map_break:n {*⟨tokens⟩*}

Used to terminate a `\prop_map...` function before all entries in the *⟨property list⟩* have been processed, inserting the *⟨tokens⟩* after the mapping has ended. This will normally take place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break:n { <tokens> } }
  {
    % Do something useful
  }
}
```

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

8 Viewing property lists

\prop_show:N**\prop_show:c**

Updated: 2012-09-09

\prop_show:N *⟨property list⟩*

Displays the entries in the *⟨property list⟩* in the terminal.

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 XV

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</code> $\langle box \rangle$
<code>\box_ht:c</code>	Calculates the height (above the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

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

<code>\box_wd:N</code>	<code>\box_wd:N</code> $\langle box \rangle$
<code>\box_wd:c</code>	Calculates the width of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

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

<code>\box_set_dp:Nn</code>	<code>\box_set_dp:Nn</code> $\langle box \rangle$ $\{ \langle dimension expression \rangle \}$
<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</code> $\langle box \rangle$ $\{ \langle dimension expression \rangle \}$
<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</code> $\langle box \rangle$ $\{ \langle dimension expression \rangle \}$
<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</code> $\langle box \rangle$
<code>\box_if_empty_p:c</code> ★	<code>\box_if_empty:N</code> $\langle box \rangle$ $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$
<code>\box_if_empty:N</code> ★	Tests if $\langle box \rangle$ is a empty (equal to <code>\c_empty_box</code>).
<code>\box_if_empty:c</code> ★	

<code>\box_if_horizontal_p:N</code> ★	<code>\box_if_horizontal_p:N</code> $\langle box \rangle$
<code>\box_if_horizontal_p:c</code> ★	<code>\box_if_horizontal:N</code> $\langle box \rangle$ $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$
<code>\box_if_horizontal:N</code> ★	Tests if $\langle box \rangle$ is a horizontal box.
<code>\box_if_horizontal:c</code> ★	

<code>\box_if_vertical_p:N</code> ★	<code>\box_if_vertical_p:N</code> $\langle box \rangle$
<code>\box_if_vertical_p:c</code> ★	<code>\box_if_vertical:N</code> $\langle box \rangle$ $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$
<code>\box_if_vertical:N</code> ★	Tests if $\langle box \rangle$ is a vertical box.
<code>\box_if_vertical:c</code> ★	

5 The last box inserted

<hr/> <code>\box_set_to_last:N</code> <hr/>	<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> <hr/>	

6 Constant boxes

<hr/> <code>\c_empty_box</code> <hr/>	This is a permanently empty box, which is neither set as horizontal nor vertical.
Updated: 2012-11-04 <hr/>	

7 Scratch boxes

<hr/> <code>\l_tmpa_box</code> <hr/>	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> <hr/>	
Updated: 2012-11-04 <hr/>	

<hr/> <code>\g_tmpa_box</code> <hr/>	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> <hr/>	

8 Viewing box contents

<hr/> <code>\box_show:N</code> <hr/>	<code>\box_show:N</code> $\langle box \rangle$
<code>\box_show:c</code> <hr/>	Shows full details of the content of the $\langle box \rangle$ in the terminal.
Updated: 2012-05-11 <hr/>	
<hr/> <code>\box_show:Nnn</code> <hr/>	<code>\box_show:Nnn</code> $\langle box \rangle$ $\langle intexpr_1 \rangle$ $\langle intexpr_2 \rangle$
<code>\box_show:cnn</code> <hr/>	Display the contents of $\langle box \rangle$ in the terminal, showing the first $\langle intexpr_1 \rangle$ items of the box, and descending into $\langle intexpr_2 \rangle$ group levels.
New: 2012-05-11 <hr/>	
<hr/> <code>\box_log:N</code> <hr/>	<code>\box_log:N</code> $\langle box \rangle$
<code>\box_log:c</code> <hr/>	Writes full details of the content of the $\langle box \rangle$ to the log.
New: 2012-05-11 <hr/>	

<hr/> <code>\box_log:Nnn</code> <hr/>	<code>\box_show:Nnn</code> $\langle box \rangle$ $\langle intexpr_1 \rangle$ $\langle intexpr_2 \rangle$
<code>\box_log:cnn</code>	Writes the contents of $\langle box \rangle$ to the log, showing the first $\langle intexpr_1 \rangle$ items of the box, and descending into $\langle intexpr_2 \rangle$ group levels.
<hr/> <code>New: 2012-05-11</code> <hr/>	

9 Horizontal mode boxes

<hr/> <code>\hbox:n</code> <hr/>	<code>\hbox:n</code> $\{\langle contents \rangle\}$
	Typesets the $\langle contents \rangle$ into a horizontal box of natural width and then includes this box in the current list for typesetting.

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

<hr/> <code>\hbox_to_wd:nn</code> <hr/>	<code>\hbox_to_wd:nn</code> $\{\langle dimexpr \rangle\}$ $\{\langle contents \rangle\}$
	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</code> $\{\langle contents \rangle\}$
	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</code> $\langle box \rangle$ $\{\langle contents \rangle\}$
<code>\hbox_set:cn</code>	Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$.
<code>\hbox_gset:Nn</code>	
<code>\hbox_gset:cn</code> <hr/>	

<hr/> <code>\hbox_set_to_wd:Nnn</code> <hr/>	<code>\hbox_set_to_wd:Nnn</code> $\langle box \rangle$ $\{\langle dimexpr \rangle\}$ $\{\langle contents \rangle\}$
<code>\hbox_set_to_wd:cnn</code>	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>	
<code>\hbox_gset_to_wd:cnn</code> <hr/>	

<hr/> <code>\hbox_overlap_right:n</code> <hr/>	<code>\hbox_overlap_right:n</code> $\{\langle contents \rangle\}$
	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</code> $\{\langle contents \rangle\}$
	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>	
<code>\hbox_set_end:</code>	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_gset:Nw</code>	
<code>\hbox_gset:cw</code>	
<code>\hbox_gset_end:</code>	

<hr/> <code>\hbox_unpack:N</code> <hr/>	<code>\hbox_unpack:N <box></code>
<code>\hbox_unpack:c</code>	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`.

<hr/> <code>\hbox_unpack_clear:N</code> <hr/>	<code>\hbox_unpack_clear:N <box></code>
<code>\hbox_unpack_clear:c</code>	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.

<hr/> <code>\vbox:n</code> <hr/>	<code>\vbox:n {<contents>}</code>
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`.

<hr/> <code>\vbox_top:n</code> <hr/>	<code>\vbox_top:n {<contents>}</code>
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 <i>first</i> item added to the box.

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

<hr/> <code>\vbox_to_ht:nn</code> <hr/>	<code>\vbox_to_ht:nn {<dimexpr>} {<contents>}</code>
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.
<hr/> <code>\vbox_to_zero:n</code> <hr/>	<code>\vbox_to_zero:n {<contents>}</code>
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.
<hr/> <code>\vbox_set:Nn</code> <code>\vbox_set:cn</code> <code>\vbox_gset:Nn</code> <code>\vbox_gset:cn</code> <hr/>	<code>\vbox_set:Nn <box> {<contents>}</code> Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$.
Updated: 2011-12-18	
<hr/> <code>\vbox_set_top:Nn</code> <code>\vbox_set_top:cn</code> <code>\vbox_gset_top:Nn</code> <code>\vbox_gset_top:cn</code> <hr/>	<code>\vbox_set_top:Nn <box> {<contents>}</code> Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. The baseline of the box will be equal to that of the <i>first</i> item added to the box.
Updated: 2011-12-18	
<hr/> <code>\vbox_set_to_ht:Nnn</code> <code>\vbox_set_to_ht:cnn</code> <code>\vbox_gset_to_ht:Nnn</code> <code>\vbox_gset_to_ht:cnn</code> <hr/>	<code>\vbox_set_to_ht:Nnn <box> {<dimexpr>} {<contents>}</code> Typesets the $\langle contents \rangle$ to the height given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$.
Updated: 2011-12-18	
<hr/> <code>\vbox_set:Nw</code> <code>\vbox_set:cw</code> <code>\vbox_set_end:</code> <code>\vbox_gset:Nw</code> <code>\vbox_gset:cw</code> <code>\vbox_gset_end:</code> <hr/>	<code>\vbox_set:Nw <box> <contents> \vbox_set_end:</code> Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. In contrast to <code>\vbox_set:Nn</code> this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.
Updated: 2011-12-18	
<hr/> <code>\vbox_set_split_to_ht:NNn</code> <hr/>	<code>\vbox_set_split_to_ht:NNn <box₁₂</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).

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

`\vbox_unpack:N`
`\vbox_unpack:c`

`\vbox_unpack:N` $\langle box \rangle$

Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.

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

`\vbox_unpack_clear:N`
`\vbox_unpack_clear:c`

`\vbox_unpack:N` $\langle box \rangle$

Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The $\langle box \rangle$ is then cleared globally.

T_EXhackers note: This is the T_EX 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 XVI

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>
<hr/>	<hr/>
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/>	<hr/>

<hr/> <code>\coffin_ht:N</code> <hr/>	<code>\coffin_ht:N</code> $\langle\textit{coffin}\rangle$
<code>\coffin_ht:c</code> <hr/>	Calculates the height (above the baseline) of the $\langle\textit{coffin}\rangle$ in a form suitable for use in a $\langle\textit{dimension expression}\rangle$.
<hr/> <code>\coffin_wd:N</code> <hr/>	<code>\coffin_wd:N</code> $\langle\textit{coffin}\rangle$
<code>\coffin_wd:c</code> <hr/>	Calculates the width of the $\langle\textit{coffin}\rangle$ in a form suitable for use in a $\langle\textit{dimension expression}\rangle$.

5 Coffin diagnostics

<hr/> <code>\coffin_display_handles:Nn</code> <hr/>	<code>\coffin_display_handles:Nn</code> $\langle\textit{coffin}\rangle$ $\{\langle\textit{colour}\rangle\}$
<code>\coffin_display_handles:cn</code> <hr/>	This function first calculates the intersections between all of the $\langle\textit{poles}\rangle$ of the $\langle\textit{coffin}\rangle$ to give a set of $\langle\textit{handles}\rangle$. It then prints the $\langle\textit{coffin}\rangle$ at the current location in the source, with the position of the $\langle\textit{handles}\rangle$ marked on the coffin. The $\langle\textit{handles}\rangle$ will be labelled as part of this process: the locations of the $\langle\textit{handles}\rangle$ and the labels are both printed in the $\langle\textit{colour}\rangle$ specified.
Updated: 2011-09-02 <hr/>	

<hr/> <code>\coffin_mark_handle:Nnnn</code> <hr/>	<code>\coffin_mark_handle:Nnnn</code> $\langle\textit{coffin}\rangle$ $\{\langle\textit{pole}_1\rangle\}$ $\{\langle\textit{pole}_2\rangle\}$ $\{\langle\textit{colour}\rangle\}$
<code>\coffin_mark_handle:cnnn</code> <hr/>	This function first calculates the $\langle\textit{handle}\rangle$ for the $\langle\textit{coffin}\rangle$ as defined by the intersection of $\langle\textit{pole}_1\rangle$ and $\langle\textit{pole}_2\rangle$. It then marks the position of the $\langle\textit{handle}\rangle$ on the $\langle\textit{coffin}\rangle$. The $\langle\textit{handle}\rangle$ will be labelled as part of this process: the location of the $\langle\textit{handle}\rangle$ and the label are both printed in the $\langle\textit{colour}\rangle$ specified.
Updated: 2011-09-02 <hr/>	

<hr/> <code>\coffin_show_structure:N</code> <hr/>	<code>\coffin_show_structure:N</code> $\langle\textit{coffin}\rangle$
<code>\coffin_show_structure:c</code> <hr/>	This function shows the structural information about the $\langle\textit{coffin}\rangle$ 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: 2012-09-09 <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 <hr/>	

Part XVII

The l3color package

Colour 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 Colour 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 XVIII

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> 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.
--	---

<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> 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 .
-----------------------------------	--

<code>\msg_line_number:</code> ★	<code>\msg_line_number:</code> Prints the current line number when a message is given.
----------------------------------	---

<code>\msg_fatal_text:n</code> ★	<code>\msg_fatal_text:n {<module>}</code> Produces the standard text <div style="text-align: center; margin: 10px 0;"> Fatal <i><module></i> error </div> 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.
----------------------------------	--

<code>\msg_critical_text:n</code> ★	<code>\msg_critical_text:n {<module>}</code> Produces the standard text <div style="text-align: center; margin: 10px 0;"> Critical <i><module></i> error </div> 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.
-------------------------------------	--

<code>\msg_error_text:n</code> ★	<code>\msg_error_text:n {<module>}</code> Produces the standard text <div style="text-align: center; margin: 10px 0;"> <i><module></i> error </div> 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.
----------------------------------	--

<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>}</code>
<code>\msg_fatal:(nnnnn nnnn nnn nn nnxxxx nnxxx nnxx nnx)</code>	<code>{<arg two>} {<arg three>} {<arg four>}</code>

Updated: 2012-08-11

Issues `<module> error <message>`, passing `<arg one>` to `<arg four>` to the text-creating functions. After issuing a fatal error the `TEX` run will halt.

```
\msg_critical:nnnnnn  
\msg_critical:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

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:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

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:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

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:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

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.

```
\msg_log:nnnnnn  
\msg_log:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

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: the output is briefer than `\msg_info:nnnnnn`.

<code>\msg_none:nnnnnn</code> <code>\msg_none:(nnnnn nnnn nnn nn nnxxxx nnxxx nnxx nnx)</code>	<code>\msg_none:nnnnnn {<module>} {<message>} {<arg one>}</code> <code>{<arg two>} {<arg three>} {<arg four>}</code>
---	---

Updated: 2012-08-11

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

4 Redirecting messages

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

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

to define a message, with

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

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

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

to turn all errors into warnings, or with

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

to alter only messages from that module, or even

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

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

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

<code>\msg_redirect_class:nn</code>	<code>\msg_redirect_class:nn {<class one>} {<class two>}</code>
-------------------------------------	---

Updated: 2012-04-27

Changes the behaviour of messages of *<class one>* so that they are processed using the code for those of *<class two>*.

<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:

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

<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:

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

5 Low-level message functions

The lower-level message functions should usually be accessed from the higher-level system. However, there are occasions where direct access to these functions is desirable.

<hr/> <code>\msg_interrupt:nnn</code> <hr/>	<code>\msg_interrupt:nnn {<first line>} {<text>} {<extra text>}</code>
New: 2012-06-28	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:nnn`; the documentation for the latter should be consulted for full details.

\msg_log:n**\msg_log:n** {<text>}

New: 2012-06-28

Writes to the log file with the <text> laid out in the format

```
.....  
. <text>  
.....
```

where the <text> will be wrapped to fit within the current line length. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

\msg_term:n**\msg_term:n** {<text>}

New: 2012-06-28

Writes to the terminal and log file with the <text> laid out in the format

```
*****  
* <text>  
*****
```

where the <text> will be wrapped to fit within the current line length. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

6 Kernel-specific functions

Messages from L^AT_EX3 itself are handled by the general message system, but have their own functions. This allows some text to be pre-defined, and also ensures that serious errors can be handled properly.

__msg_kernel_new:nnnn**__msg_kernel_new:nnn**

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:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues kernel $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing a fatal error the T_EX run will halt. Cannot be redirected.

```
\_msg_kernel_error:nnnnnn
\_msg_kernel_error:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues kernel $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The error will stop processing and issue the text at the terminal. After user input, the run will continue. Cannot be redirected.

```
\_msg_kernel_warning:nnnnnn
\_msg_kernel_warning:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

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 T_EX run will not be interrupted.

```
\_msg_kernel_info:nnnnnn
\_msg_kernel_info:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

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.

```
\_msg_kernel_fatal:nnnnnn {\langle module \rangle}
{\langle message \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg
three \rangle} {\langle arg four \rangle}
```

```
\_msg_kernel_error:nnnnnn {\langle module \rangle}
{\langle message \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg
three \rangle} {\langle arg four \rangle}
```

```
\_msg_kernel_warning:nnnnnn {\langle module \rangle}
{\langle message \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg
three \rangle} {\langle arg four \rangle}
```

```
\_msg_kernel_info:nnnnnn {\langle module \rangle}
{\langle message \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg
three \rangle} {\langle arg four \rangle}
```

7 Expandable errors

In a few places, the L^AT_EX3 kernel needs to produce errors in an expansion only context. This must be handled internally very differently from normal error messages, as none of the tools to print to the terminal or the log file are expandable. However, the interface is similar, with the important caveat that the message text and arguments are not expanded, and messages should be very short.

```
\_msg_kernel_expandable_error:nnnnnn ★
\_msg_kernel_expandable_error:(nnnnn|nnnn|nnn|nn) ★
```

New: 2011-11-23

```
\_msg_kernel_expandable_error:nnnnnn {\langle module \rangle}
{\langle message \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg
three \rangle} {\langle arg four \rangle}
```

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.

<code>_msg_expandable_error:n</code> ★	<code>_msg_expandable_error:n {⟨error message⟩}</code>
---	---

New: 2011-08-11

Updated: 2011-08-13

Issues an “Undefined error” message from T_EX itself, and prints the *⟨error message⟩*. The *⟨error message⟩* must be short: it is cropped at the end of one line.

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.

<code>_msg_term:nnnnnn</code>	<code>_msg_term:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}</code>
<code>_msg_term:(nnnnnV nnnnn nnn nn)</code>	

Prints the *⟨message⟩* from *⟨module⟩* in the terminal without formatting. Used in messages which print complex variable contents completely.

<code>_msg_show_variable:Nnn</code>	<code>_msg_show_variable:Nnn ⟨variable⟩ {⟨type⟩} {⟨formatted content⟩}</code>
--------------------------------------	--

Updated: 2012-09-09

Displays the *⟨formatted content⟩* of the *⟨variable⟩* of *⟨type⟩* in the terminal. The *⟨formatted content⟩* will be processed as the first argument in a call to `\iow_wrap:nnnN`, hence `\`, `_` and other formatting sequences can be used. Once expanded and processed, the *⟨formatted content⟩* must either be empty or contain `>`; everything until the first `>` will be removed.

<code>_msg_show_variable:n</code>	<code>_msg_show_variable:n {⟨formatted text⟩}</code>
------------------------------------	---

Updated: 2012-09-09

Shows the *⟨formatted text⟩* on the terminal. After expansion, unless it is empty, the *⟨formatted text⟩* must contain `>`, and the part of *⟨formatted text⟩* before the first `>` is removed. Failure to do so causes low-level T_EX errors.

<code>_msg_show_item:n</code>	<code>_msg_show_item:n ⟨item⟩</code>
<code>_msg_show_item:nn</code>	<code>_msg_show_item:nn ⟨item-key⟩ ⟨item-value⟩</code>
<code>_msg_show_item_unbraced:nn</code>	

Updated: 2012-09-09

Auxiliary functions used within the argument of `_msg_show_variable:Nnn` 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.

Part XIX

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:
}

```

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.

```
.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](#).

```
.clist_set:N
.clist_set:c
.clist_gset:N
.clist_gset:c
```

New: 2011-09-11

$\langle key \rangle$.clist_set:N = $\langle comma list variable \rangle$

Defines $\langle key \rangle$ to set $\langle comma list variable \rangle$ to $\langle value \rangle$. Spaces around commas and empty items will be stripped. If the variable does not exist, it will be created globally at the point that the key is set up.

```
.code:n
```

Updated: 2013-07-10

$\langle key \rangle$.code:n = $\langle code \rangle$

Stores the $\langle code \rangle$ for execution when $\langle key \rangle$ is used. The $\langle code \rangle$ can include one parameter (**#1**), which will be the $\langle value \rangle$ given for the $\langle key \rangle$. The **x**-type variant will expand $\langle code \rangle$ at the point where the $\langle key \rangle$ is created.

<hr/> <code>.default:n</code> <hr/>	<code><key> .default:n = <default></code>
<code>.default:V</code>	
<code>.default:o</code>	Creates a <i><default></i> value for <i><key></i> , which is used if no value is given. This will be used
<code>.default:x</code> <hr/>	if only the key name is given, but not if a blank <i><value></i> is given:
Updated: 2013-07-09	<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>
<hr/> <code>.dim_set:N</code> <hr/>	<code><key> .dim_set:N = <dimension></code>
<code>.dim_set:c</code>	
<code>.dim_gset:N</code>	Defines <i><key></i> to set <i><dimension></i> to <i><value></i> (which must a dimension expression). If the
<code>.dim_gset:c</code> <hr/>	variable does not exist, it will be created globally at the point that the key is set up.
<hr/> <code>.fp_set:N</code> <hr/>	<code><key> .fp_set:N = <floating point></code>
<code>.fp_set:c</code>	
<code>.fp_gset:N</code>	Defines <i><key></i> to set <i><floating point></i> to <i><value></i> (which must a floating point expression).
<code>.fp_gset:c</code> <hr/>	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>
New: 2013-07-14	Defines <i><key></i> as belonging to the <i><groups></i> 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>	
<code>.initial:o</code>	Initialises the <i><key></i> with the <i><value></i> , equivalent to
<code>.initial:x</code> <hr/>	<code>\keys_set:nn {<module>} { <key> = <value> }</code>
Updated: 2013-07-09	
<hr/> <code>.int_set:N</code> <hr/>	<code><key> .int_set:N = <integer></code>
<code>.int_set:c</code>	
<code>.int_gset:N</code>	Defines <i><key></i> to set <i><integer></i> to <i><value></i> (which must be an integer expression). If the
<code>.int_gset:c</code> <hr/>	variable does not exist, it will be created globally at the point that the key is set up.

<hr/> <code>.meta:n</code> <hr/>	<code><key> .meta:n = {<keyval list>}</code>
Updated: 2013-07-10	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	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	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> <code>.multichoices:Vn</code> <code>.multichoices:on</code> <code>.multichoices:xn</code> <hr/>	<code><key> .multichoices:nn <choices> <code></code>
New: 2011-08-21 Updated: 2013-07-10	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.
<hr/> <code>.skip_set:N</code> <code>.skip_set:c</code> <code>.skip_gset:N</code> <code>.skip_gset:c</code> <hr/>	<code><key> .skip_set:N = <skip></code> 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.
<hr/> <code>.tl_set:N</code> <code>.tl_set:c</code> <code>.tl_gset:N</code> <code>.tl_gset:c</code> <hr/>	<code><key> .tl_set:N = <token list variable></code> Defines <code><key></code> to set <code><token list variable></code> to <code><value></code> . If the variable does not exist, it will be created globally at the point that the key is set up.
<hr/> <code>.tl_set_x:N</code> <code>.tl_set_x:c</code> <code>.tl_gset_x:N</code> <code>.tl_gset_x:c</code> <hr/>	<code><key> .tl_set_x:N = <token list variable></code> Defines <code><key></code> to set <code><token list variable></code> to <code><value></code> , which will be subjected to an x-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.
<hr/> <code>.value_forbidden:</code> <hr/>	<code><key> .value_forbidden:</code> 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.
<hr/> <code>.value_required:</code> <hr/>	<code><key> .value_required:</code> 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.

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
```

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:(nVN|nvN|noN|nn|nV|nv|no)
```

```
\keys_set_known:nnN {<module>} {<keyval list>} <tl>
```

New: 2011-08-23

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.

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</code>
<code>\keys_set_filter:(nnVN nnvN nnoN nnn nnV nnv nno)</code>	<code>list)} <tl></code>

New: 2013-07-14

Activates key filtering in an “opt-out” sense: keys assigned to any of the *<groups>* specified will be ignored. The *<groups>* are given as a comma-separated list. Unknown keys are not assigned to any group and will thus always be set. The key-value pairs for each key which is filtered out will be stored in the *<tl>* in a comma-separated form (*i.e.* an edited version of the *<keyval list>*). The `\keys_set_filter:nnn` version skips this 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

Activates key filtering in an “opt-in” sense: only keys assigned to one or more of the *<groups>* specified will be set. The *<groups>* are given as a comma-separated list. Unknown keys are not assigned to any group and will thus never be set.

7 Utility functions for keys

<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 <module> <key> <choice></code>
<code>\keys_if_choice_exist:nnnTF</code> ★	<code>\keys_if_choice_exist:nnnTF <module> <key> <choice> {<true code>} {<false code>}</code>

New: 2011-08-21

Tests if the *<choice>* is defined for the *<key>* within the *<module>*, *i.e.* if any code has been defined for *<key>/<choice>*. The test is **false** if the *<key>* itself is not defined.

<code>\keys_show:nn</code>	<code>\keys_show:nn {<module>} {<key>}</code>
----------------------------	---

Shows the function which is used to actually implement a *<key>* for a *<module>*.

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 *<key–value list>* into *<keys>* and associated *<values>*. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and

a second function is required for keys given without any value (it is called with a single argument).

The parser does not double # tokens or expand any input. Active tokens = and , appearing at the outer level of braces are converted to category “other” (12) so that the parser does not “miss” any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces 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 XX

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. Spaces are not allowed in file names.

1 File operation functions

<hr/> <code>\g_file_current_name_tl</code> <hr/>	Contains the name of the current L ^A T _E X file. This variable should not be modified: it is intended for information only. It will be equal to <code>\c_job_name_tl</code> at the start of a L ^A T _E X 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/> <div>Updated: 2012-02-10</div> <hr/>	<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 TeX search path and the additional paths controlled by <code>\file_path_include:n</code> .
<hr/> <code>\file_add_path:nN</code> <hr/> <div>Updated: 2012-02-10</div> <hr/>	<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/> <div>Updated: 2012-02-17</div> <hr/>	<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 L ^A T _E X 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 <code>x</code> -type expansion except active characters. Spaces are not allowed in the $\langle path \rangle$.
<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 <code>x</code> -type expansion except active characters. Spaces are not allowed in the $\langle path \rangle$.
<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 \TeX is limited to 16 input streams and 16 output streams, direct use of the streams by the programmer is not supported in $\text{\LaTeX}3$. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

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

<hr/> <code>\ior_new:N</code> <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.
<code>\iow_new:c</code>	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>
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 \TeX 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 \TeX run ends. The $\langle true\ code \rangle$ is then inserted into the input stream. If the file is not found, no error is raised and the $\langle false\ code \rangle$ is inserted into the input stream.
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: 2012-09-09

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**

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**.

\iow_shipout:Nn**\iow_shipout:Nx****\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.* **\iow_shipout_x:Nn**).

\iow_shipout_x:Nn**\iow_shipout_x:Nx**

Updated: 2012-09-08

\iow_shipout_x:Nn $\langle stream \rangle$ $\{\langle tokens \rangle\}$

This function 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).

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

\iow_char:N ★**\iow_char:N** $\langle char \rangle$

Inserts $\langle char \rangle$ into the output stream. Useful when trying to write difficult characters such as %, {, }, *etc.* in messages, for example:

$$\backslash\mathrm{iow_now:Nx} \backslash\mathrm{g_my_iow} \{ \backslash\mathrm{iow_char:N} \{ \mathrm{text} \backslash\mathrm{iow_char:N} \} \}$$

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

\iow_newline: ★**\iow_newline:**

Function to add a new line within the $\langle tokens \rangle$ written to a file. The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

2.1 Wrapping lines in output

`\iow_wrap:nnnN`

New: 2012-06-28

`\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 $\langle run-on text \rangle$. 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 message.

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 context 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_TE_X systems.

<hr/> <hr/> <code>\c_catcode_other_space_tl</code> <hr/> <hr/>	Token list containing one character with category code 12, (“other”), and character code 32 (space).
<hr/> <hr/> <small>New: 2011-09-05</small> <hr/> <hr/>	

2.2 Constant input–output streams

<hr/> <hr/> <code>\c_term_ior</code> <hr/> <hr/>	Constant input stream for reading from the terminal. Reading from this stream using <code>\ior_get:NN</code> or similar will result in a prompt from T _E X of the form <code><tl>=</code>
--	---

<hr/> <hr/> <code>\c_log_ior</code> <code>\c_term_ior</code> <hr/> <hr/>	Constant output streams for writing to the log and to the terminal (plus the log), respectively.
---	--

2.3 Primitive conditionals

<hr/> <hr/> <code>\if_eof:w</code> ★ <hr/> <hr/>	<code>\if_eof:w <stream></code> <code><true code></code> <code>\else:</code> <code><false code></code> <code>\fi:</code> Tests if the <code><stream></code> returns “end of file”, which is true for non-existent files. The <code>\else:</code> branch is optional.
--	---

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

2.4 Internal file functions and variables

<hr/> <hr/> <code>\l__file_internal_name_ior</code> <hr/> <hr/>	Used to test for the existence of files when opening.
<hr/> <hr/> <code>\l__file_internal_name_tl</code> <hr/> <hr/>	Used to return the full name of a file for internal use.
<hr/> <hr/> <code>__file_name_sanitizenn</code> <hr/> <hr/>	<code>__file_name_sanitizenn {<name>} {<tokens>}</code>
<hr/> <hr/> <small>New: 2012-02-09</small> <hr/> <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

<code>__ior_open:Nn</code>	<code>__ior_open:Nn <stream> {<file name>}</code>
<code>__ior_open:No</code>	

New: 2012-01-23

This function has identical syntax to the public version. However, it does not take precautions against active characters in the *<file name>*, and it does not attempt to add a *<path>* to the *<file name>*: it is therefore intended to be used by higher-level functions which have already fully expanded the *<file name>* and which need to perform multiple open or close operations. See for example the implementation of `\file_add_path:nN`,

Part XXI

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 , 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 $\text{sind } x$, $\text{cosd } x$, $\text{tand } x$, $\text{cotd } x$, $\text{secd } x$, $\text{cscd } x$ expecting their arguments in degrees.
 - Inverse trigonometric functions: $\text{asin } x$, $\text{acos } x$, $\text{atan } x$, $\text{acot } x$, $\text{asec } x$, $\text{acsc } x$ giving a result in radians, and $\text{asind } x$, $\text{acosd } x$, $\text{atand } x$, $\text{acotd } x$, $\text{asecd } x$, $\text{acscd } x$ giving a result in degrees.
- (not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\text{sech } x$, $\text{csch } x$, and $\text{asinh } x$, $\text{acosh } x$, $\text{atanh } x$, $\text{acoth } x$, $\text{asech } x$, $\text{acsch } x$.
- Extrema: $\max(x, y, \dots)$, $\min(x, y, \dots)$, $\text{abs}(x)$.
 - Rounding functions: $\text{round}(x, n)$ round to closest, $\text{round0}(x, n)$ round towards zero, $\text{round}\pm(x, n)$ round towards $\pm\infty$. 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 `\langle type \rangle_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 zero.
<code>\fp_gzero_new:N</code>	
<code>\fp_gzero_new:c</code>	
Updated: 2012-05-08	

2 Setting floating point variables

<code>\fp_set:Nn</code>	<code>\fp_set:Nn <fp var> {<floating point expression>}</code>
<code>\fp_set:cn</code>	Sets <i><fp var></i> equal to the result of computing the <i><floating point expression></i> .
<code>\fp_gset:Nn</code>	
<code>\fp_gset:cn</code>	
Updated: 2012-05-08	

```
\fp_set_eq:NN
\fp_set_eq:(cN|Nc|cc)
\fp_gset_eq:NN
\fp_gset_eq:(cN|Nc|cc)
```

Updated: 2012-05-08

```
\fp_set_eq:NN <fp var1> <fp var2>
```

Sets the floating point variable $\langle fp\ var_1 \rangle$ equal to the current value of $\langle fp\ var_2 \rangle$.

```
\fp_add:Nn
\fp_add:cn
\fp_gadd:Nn
\fp_gadd:cn
```

Updated: 2012-05-08

```
\fp_add:Nn <fp var> {<floating point expression>}
```

Adds the result of computing the $\langle floating\ point\ expression \rangle$ to the $\langle fp\ var \rangle$.

```
\fp_sub:Nn
\fp_sub:cn
\fp_gsub:Nn
\fp_gsub:cn
```

Updated: 2012-05-08

```
\fp_sub:Nn <fp var> {<floating point expression>}
```

Subtracts the result of computing the $\langle floating\ point\ expression \rangle$ from the $\langle fp\ var \rangle$.

3 Using floating point numbers

```
\fp_eval:n ★
```

New: 2012-05-08

Updated: 2012-07-08

```
\fp_eval:n {<floating point expression>}
```

Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result as a decimal number with 16 significant figures and 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|n) ★
```

New: 2012-05-08

Updated: 2012-07-08

```
\fp_to_decimal:N <fp var>
```

```
\fp_to_decimal:n {<floating point expression>}
```

Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result as a decimal number with 16 significant figures and no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and `nan` trigger an “invalid operation” exception.

```
\fp_to_dim:N ★
\fp_to_dim:(c|n) ★
```

Updated: 2012-07-08

```
\fp_to_dim:N <fp var>
```

```
\fp_to_dim:n {<floating point expression>}
```

Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result as a dimension (in `pt`) suitable for use in dimension expressions. The output is identical to `\fp_to_decimal:n`, with an additional trailing `pt`. In particular, the result may be outside the range $[-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]$ of valid \TeX dimensions, leading to overflow errors if used as a dimension. The values $\pm\infty$ and `nan` trigger an “invalid operation” exception.

<code>\fp_to_int:N</code> ★	<code>\fp_to_int:N</code> $\langle fp\ var \rangle$
<code>\fp_to_int:(c n)</code> ★	<code>\fp_to_int:n</code> $\{\langle floating\ point\ expression \rangle\}$
Updated: 2012-07-08	Evaluates the $\langle floating\ point\ expression \rangle$, and rounds the result to the closest integer, with ties rounded to an even integer. The result may be outside the range $[-2^{31} + 1, 2^{31} - 1]$ of valid TeX integers, triggering TeX errors if used in an integer expression. The values $\pm\infty$ and <code>nan</code> trigger an “invalid operation” exception.

<code>\fp_to_scientific:N</code> ★	<code>\fp_to_scientific:N</code> $\langle fp\ var \rangle$
<code>\fp_to_scientific:(c n)</code> ★	<code>\fp_to_scientific:n</code> $\{\langle floating\ point\ expression \rangle\}$
New: 2012-05-08 Updated: 2012-07-08	Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result in scientific notation with 16 significant figures:

$$\langle optional\ - \rangle \langle digit \rangle . \langle 15\ digits \rangle e \langle optional\ sign \rangle \langle exponent \rangle$$

The leading $\langle digit \rangle$ is non-zero except in the case of ± 0 . The values $\pm\infty$ and `nan` trigger an “invalid operation” exception.

<code>\fp_to_tl:N</code> ★	<code>\fp_to_tl:N</code> $\langle fp\ var \rangle$
<code>\fp_to_tl:(c n)</code> ★	<code>\fp_to_tl:n</code> $\{\langle floating\ point\ expression \rangle\}$
Updated: 2012-07-08	Evaluates the $\langle floating\ point\ expression \rangle$ 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 (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 <code>nan</code> are rendered as <code>0</code> , <code>-0</code> , <code>inf</code> , <code>-inf</code> , and <code>nan</code> respectively.

<code>\fp_use:N</code> ★	<code>\fp_use:N</code> $\langle fp\ var \rangle$
<code>\fp_use:c</code> ★	Inserts the value of the $\langle fp\ var \rangle$ into the input stream as a decimal number with 16 significant figures and no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values $\pm\infty$ and <code>nan</code> trigger an “invalid operation” exception. 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:NTF</code> $\langle fp\ var \rangle$ $\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
<code>\fp_if_exist:NTF</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:cTF</code> ★	
Updated: 2012-05-08	

<code>\fp_compare_p:nNn</code> ★	<code>\fp_compare_p:nNn {<fpexpr₁>} <relation> {<fpexpr₂>}</code>
<code>\fp_compare_p:n</code> ★	<code>\fp_compare:nNnTF {<fpexpr₁>} <relation> {<fpexpr₂>} {<true code>} {<false code>}</code>
<code>\fp_compare:nNnTF</code> ★	<code>\fp_compare_p:n {<fpexpr₁>} <relation> {<fpexpr₂>}</code>
<code>\fp_compare:nTF</code> ★	<code>\fp_compare:nTF {<fpexpr₁>} <relation> {<fpexpr₂>} {<true code>} {<false code>}</code>

Updated: 2012-05-08

Compares the $\langle fpexpr_1 \rangle$ and the $\langle fpexpr_2 \rangle$, and returns **true** if the $\langle relation \rangle$ is obeyed. Two floating 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 relations is denoted by the symbol **?**. The **nNn** functions support the $\langle relations \rangle$ **<**, **=**, **>**, and **?**. The **n** functions support as a $\langle relation \rangle$ any non-empty string of those four symbols, plus optional leading **!** (which negate the $\langle relation \rangle$), with the restriction that the $\langle relation \rangle$ may not start with **?**. Common choices of $\langle relation \rangle$ include **>=** (greater or equal), **!=** (not equal), **!?** (comparable). Note that a **nan** is distinct from any value, even another **nan**, hence $x = x$ is not true for a **nan**. Since a **nan** is not comparable to any floating point, to test if a value is **nan**, one can use the following, where 0 is an arbitrary floating point.

```
\fp_compare:nNnTF { <value> } ? { 0 }
{ } % <value> is nan
{ } % <value> is not nan
```

5 Floating point expression loops

<code>\fp_do_until:nNnn</code> ☆	<code>\fp_do_until:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
----------------------------------	--

New: 2012-08-16

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 $\langle code \rangle$ 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 **true** then the $\langle code \rangle$ will be inserted into the input stream again and a loop will occur until the $\langle relation \rangle$ is **false**.

<code>\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 $\langle floating\ point\ expressions \rangle$ as described for `\fp_compare:nNnTF`, and then places the $\langle code \rangle$ in the input stream if the $\langle relation \rangle$ is **false**. After the $\langle code \rangle$ has been processed by T_EX the test will be repeated, and a loop will occur until the test is **true**.

<hr/> <code>\fp_while_do:nNnn</code> ☆ <hr/>	<code>\fp_while_do:nNnn {<fpexpr1>} <relation> {<fpexpr2>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .
<hr/> <code>\fp_do_until:nn</code> ☆ <hr/>	<code>\fp_do_until:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16	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 .
<hr/> <code>\fp_do_while:nn</code> ☆ <hr/>	<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 .
<hr/> <code>\fp_until_do:nn</code> ☆ <hr/>	<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 .
<hr/> <code>\fp_while_do:nn</code> ☆ <hr/>	<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/>	Zero, with either sign.
New: 2012-05-08	
<hr/> <code>\c_one_fp</code> <hr/>	One as an fp: useful for comparisons in some places.
New: 2012-05-08	

<hr/> <code>\c_inf_fp</code> <code>\c_minus_inf_fp</code> <hr/>	Infinity, with either sign. These can be input directly in a floating point expression as <code>inf</code> and <code>-inf</code> .
<hr/> New: 2012-05-08 <hr/>	
<hr/> <code>\c_e_fp</code> <hr/>	The value of the base of the natural logarithm, $e = \exp(1)$.
<hr/> Updated: 2012-05-08 <hr/>	
<hr/> <code>\c_pi_fp</code> <hr/>	The value of π . This can be input directly in a floating point expression as <code>pi</code> .
<hr/> Updated: 2013-11-17 <hr/>	
<hr/> <code>\c_one_degree_fp</code> <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/> New: 2012-05-08 Updated: 2013-11-17 <hr/>	
<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 <i><exception></i> is on, which normally means the given <i><exception></i> 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 <i><exception></i> 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 <i><exception></i> has occurred. Note that this function is expandable: it is used internally by l3fp 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	All occurrences of the <i><exception></i> (<i>invalid_operation</i> , <i>division_by_zero</i> , <i>overflow</i> , or <i>underflow</i>) within the current group are treated as <i><trap type></i> , which can be <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 n)</code>	<code>\fp_show:n {<floating point expression>}</code>
New: 2012-05-08 Updated: 2012-08-14	Evaluates the <i><floating point expression></i> and displays the result in the terminal.

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:

- $\langle sign \rangle$: a possibly empty string of + and - characters;
- $\langle significand \rangle$: a non-empty string of digits together with zero or one dot;
- $\langle exponent \rangle$ optionally: the character **e**, followed by a possibly empty string of + and - tokens, and a non-empty string of digits.

The sign of the resulting number is + if $\langle sign \rangle$ contains an even number of -, and - otherwise, hence, an empty $\langle sign \rangle$ denotes a non-negative input. The stored significand is obtained from $\langle significand \rangle$ by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input $\langle significand \rangle$ has at most 16 digits. The stored $\langle exponent \rangle$ is obtained by combining the input $\langle exponent \rangle$ (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting $\langle exponent \rangle$ is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by $\pm \infty$), or an underflow (resulting in ± 0).

The result is thus ± 0 if and only if $\langle significand \rangle$ contains no non-zero digit (*i.e.*, consists only in 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.

- Implicit multiplication by juxtaposition (`2pi`, `3(4+5)`, *etc*).
- Function calls (`sin`, `ln`, *etc*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Binary `*`, `/` and `%`.
- 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

$$\begin{aligned}\text{sin2pi} &= \sin(2\pi) = 0, \\ 2^{2\text{max}(3,4)} &= 2^{2 \max(3,4)} = 256.\end{aligned}$$

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.

```
TWOBARS \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 { <operand1> <comparison> <operand2> }
```

The $\langle comparison \rangle$ consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by `!`. It may not start with `?`. This evaluates to `+1` if the $\langle comparison \rangle$ between the $\langle operand_1 \rangle$ and $\langle operand_2 \rangle$ is true, and `+0` otherwise.

```
+ \fp_eval:n { <operand1> + <operand2> }
- \fp_eval:n { <operand1> - <operand2> }
```

Computes the sum or the difference of its two $\langle operands \rangle$. The “invalid operation” exception occurs for $\infty - \infty$. “Underflow” and “overflow” occur when appropriate.

```
* \fp_eval:n { <operand1> * <operand2> }
/ \fp_eval:n { <operand1> / <operand2> }
```

Computes the product or the ratio of its two $\langle operands \rangle$. The “invalid operation” exception occurs for ∞/∞ , $0/0$, or $0 * \infty$. “Division by zero” occurs when dividing a finite non-zero number by ± 0 . “Underflow” and “overflow” occur when appropriate.

```

+ \fp_eval:n { + <operand> }
- \fp_eval:n { - <operand> }
! \fp_eval:n { ! <operand> }

```

The unary + does nothing, the unary - changes the sign of the $\langle operand \rangle$, and ! $\langle operand \rangle$ 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 { <operand1> ** <operand2> }
^ \fp_eval:n { <operand1> ^ <operand2> }

```

Raises $\langle operand_1 \rangle$ to the power $\langle operand_2 \rangle$. This operation is right associative, hence `2 ** 2 ** 3` equals $2^{2^3} = 256$. 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( <fpexpr> ) }

```

Computes the absolute value of the $\langle fpexpr \rangle$. This function does not raise any exception beyond those raised when computing its operand $\langle fpexpr \rangle$. See also `\fp_abs:n`.

```

exp \fp_eval:n { exp( <fpexpr> ) }

```

Computes the exponential of the $\langle fpexpr \rangle$. “Underflow” and “overflow” occur when appropriate.

```

ln \fp_eval:n { ln( <fpexpr> ) }

```

Computes the natural logarithm of the $\langle fpexpr \rangle$. Negative numbers have no (real) logarithm, hence the “invalid operation” is raised in that case, including for $\ln(-0)$. “Division by zero” occurs when evaluating $\ln(+0) = -\infty$. “Underflow” and “overflow” occur when appropriate.

```

max \fp_eval:n { max( <fpexpr1> , <fpexpr2> , ... ) }
min \fp_eval:n { min( <fpexpr1> , <fpexpr2> , ... ) }

```

Evaluates each $\langle fpexpr \rangle$ and computes the largest (smallest) of those. If any of the $\langle fpexpr \rangle$ is a **nan**, the result is **nan**. Those operations do not raise exceptions.

<code>round</code>	<code>\fp_eval:n { round <option> (<fpexpr>) }</code>
<code>round0</code>	<code>\fp_eval:n { round <option> (<fpexpr₁> , <fpexpr₂>) }</code>
<code>round+</code>	Rounds $\langle fpexpr_1 \rangle$ to $\langle fpexpr_2 \rangle$ places. When $\langle fpexpr_2 \rangle$ is omitted, it is assumed to be 0, <i>i.e.</i> , $\langle fpexpr_1 \rangle$ is rounded to an integer. The $\langle option \rangle$ controls the rounding direction:
<code>round-</code>	

- by default, the operation rounds to the closest allowed number (rounding ties to even);
- with 0, the operation rounds towards 0, *i.e.*, truncates;
- with +, the operation rounds towards $+\infty$;
- with -, the operation rounds towards $-\infty$.

If $\langle fpexpr_2 \rangle$ does not yield an integer less than 10^8 in absolute value, then an “invalid operation” exception is raised. “Overflow” may occur if the result is infinite (this cannot happen unless $\langle fpexpr_2 \rangle < -9984$).

<code>sin</code>	<code>\fp_eval:n { sin(<fpexpr>) }</code>
<code>cos</code>	<code>\fp_eval:n { cos(<fpexpr>) }</code>
<code>tan</code>	<code>\fp_eval:n { tan(<fpexpr>) }</code>
<code>cot</code>	<code>\fp_eval:n { cot(<fpexpr>) }</code>
<code>csc</code>	<code>\fp_eval:n { csc(<fpexpr>) }</code>
<code>sec</code>	<code>\fp_eval:n { sec(<fpexpr>) }</code>

Updated: 2013-11-17

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$ given in radians. For arguments given in degrees, see `sind`, `cosd`, *etc.* Note that since π is irrational, $\sin(8\pi)$ is not quite zero, while its analog `sind`(8×180) is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate.

<code>sind</code>	<code>\fp_eval:n { sind(<fpexpr>) }</code>
<code>cosd</code>	<code>\fp_eval:n { cosd(<fpexpr>) }</code>
<code>tand</code>	<code>\fp_eval:n { tand(<fpexpr>) }</code>
<code>cotd</code>	<code>\fp_eval:n { cotd(<fpexpr>) }</code>
<code>cscd</code>	<code>\fp_eval:n { cscd(<fpexpr>) }</code>
<code>secd</code>	<code>\fp_eval:n { secd(<fpexpr>) }</code>

New: 2013-11-02

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$ given in degrees. For arguments given in radians, see `sin`, `cos`, *etc.* Note that since π is irrational, $\sin(8\pi)$ is not quite zero, while its analog `sind`(8×180) is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate.

<code>asin</code>	<code>\fp_eval:n { asin(<fpexpr>) }</code>
<code>acos</code>	<code>\fp_eval:n { acos(<fpexpr>) }</code>
<code>acsc</code>	<code>\fp_eval:n { acsc(<fpexpr>) }</code>
<code>asec</code>	<code>\fp_eval:n { asec(<fpexpr>) }</code>

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fpexpr \rangle$ and returns the result in radians, in the range $[-\pi/2, \pi/2]$ for `asin` and `acsc` and $[0, \pi]$ for `acos` and `asec`. For a result in degrees, use `asind`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate.

<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>
	<code>\fp_eval:n { acot(<fpexpr₁> , <fpexpr₂>) }</code>

New: 2013-11-02

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.

<hr/>	<code>\fp_eval:n { atand(<fpepr>) }</code>
<code>atand</code>	<code>\fp_eval:n { atand(<fpepr1> , <fpepr2>) }</code>
<code>acotd</code>	<code>\fp_eval:n { acotd(<fpepr>) }</code>
<hr/>	<code>\fp_eval:n { acotd(<fpepr1> , <fpepr2>) }</code>
<hr/>	

New: 2013-11-02

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 `<fpepr>`: 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 $(\langle fpepr_2 \rangle, \langle fpepr_1 \rangle)$: this is the arctangent of $\langle fpepr_1 \rangle / \langle fpepr_2 \rangle$, possibly shifted by 180 depending on the signs of $\langle fpepr_1 \rangle$ and $\langle fpepr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpepr_1 \rangle, \langle fpepr_2 \rangle)$, equal to the arccotangent of $\langle fpepr_1 \rangle / \langle fpepr_2 \rangle$, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio $\langle fpepr_1 \rangle / \langle fpepr_2 \rangle$ 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/>	
<code>inf</code>	The special values $+\infty$, $-\infty$, and <code>nan</code> are represented as <code>inf</code> , <code>-inf</code> and <code>nan</code> (see <code>\c_-</code>
<code>nan</code>	<code>inf_fp</code> , <code>\c_minus_inf_fp</code> and <code>\c_nan_fp</code>).
<hr/>	

<hr/>	
<code>pi</code>	The value of π (see <code>\c_pi_fp</code>).
<hr/>	

<hr/>	
<code>deg</code>	The value of 1° in radians (see <code>\c_one_degree_fp</code>).
<hr/>	

<hr/>	
<code>em</code>	Those units of measurement are equal to their values in <code>pt</code> , namely
<code>ex</code>	
<code>in</code>	
<code>pt</code>	
<code>pc</code>	
<code>cm</code>	
<code>mm</code>	
<code>dd</code>	
<code>cc</code>	
<code>nd</code>	
<code>nc</code>	
<code>bp</code>	
<code>sp</code>	
<hr/>	

$$\begin{aligned}
1\text{in} &= 72.27\text{pt} \\
1\text{pt} &= 1\text{pt} \\
1\text{pc} &= 12\text{pt} \\
1\text{cm} &= \frac{1}{2.54}\text{in} = 28.45275590551181\text{pt} \\
1\text{mm} &= \frac{1}{25.4}\text{in} = 2.845275590551181\text{pt} \\
1\text{dd} &= 0.376065\text{mm} = 1.07000856496063\text{pt} \\
1\text{cc} &= 12\text{dd} = 12.84010277952756\text{pt} \\
1\text{nd} &= 0.375\text{mm} = 1.066978346456693\text{pt} \\
1\text{nc} &= 12\text{nd} = 12.80374015748031\text{pt} \\
1\text{bp} &= \frac{1}{72}\text{in} = 1.00375\text{pt} \\
1\text{sp} &= 2^{-16}\text{pt} = 1.52587890625e - 5\text{pt}.
\end{aligned}$$

The values of the (font-dependent) units `em` and `ex` are gathered from T_EX when the surrounding floating point expression is evaluated.

<hr/> true false <hr/>	Other names for 1 and +0.
<hr/> \dim_to_fp:n ★ New: 2012-05-08 <hr/>	\dim_to_fp:n { <i><dimexpr></i> } Expands to an internal floating point number equal to the value of the <i><dimexpr></i> in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, \dim_to_fp:n can be used to speed up parts of a computation where a low precision is acceptable.
<hr/> \fp_abs:n ★ New: 2012-05-14 Updated: 2012-07-08 <hr/>	\fp_abs:n { <i><floating point expression></i> } Evaluates the <i><floating point expression></i> as described for \fp_eval:n and leaves the absolute value of the result in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, abs() can be used.
<hr/> \fp_max:nn ★ \fp_min:nn ★ New: 2012-09-26 <hr/>	\fp_max:nn { <i><fp expression 1></i> } { <i><fp expression 2></i> } \fp_min:nn { <i><floating point expressions></i> } Evaluates the <i><floating point expressions></i> 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 TeX primitive conditional affected by **\exp_not:N**.

The following need to be done. I'll try to time-order the items.

- Rename **round0** to **trunc**, **round+** to **ceil**, and **round-** to **floor**.
- Decide what exponent range to consider.
- Improve the treatment of signalling versus quiet **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.)

`! -3 < -2 < -1` is wrongly parsed as `(-3 < -2) < -1`.

- Check that functions are monotonic when they should.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- `round` should accept any integer as its second argument.
- 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 $(\pm 0) + (\pm 0)$ should depend on the rounding mode.
- `0e9999999999` gives a `TEX` “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?
- (Likely not.) Change the internal representation of fp, by replacing braced groups of 4 digits by delimited arguments. Also consider changing the fp structure a bit to allow using `\pdfTeX_strcmp:D` to compare (not in LuaT_EX: it is too slow)?
- Support multi-character prefix operators (*e.g.*, `@/` or whatever)? Perhaps for including comments inside the computation itself??

Part XXII

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 `\luatex_if_engine:T` to avoid this. Details of coding the LuaTeX engine are detailed in the LuaTeX manual.

<code>\lua_now:n</code>	★	<code>\lua_now:n {⟨token list⟩}</code>
-------------------------	---	--

<code>\lua_now:x</code>	★
-------------------------	---

Updated: 2012-08-02

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.

<code>\lua_now_x:n</code>	★	<code>\lua_now_x:n {⟨token list⟩}</code>
---------------------------	---	--

<code>\lua_now_x:x</code>	★
---------------------------	---

New: 2012-08-02

The `⟨token list⟩` is first tokenized and expanded 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_x:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the `⟨Lua input⟩` immediately, and in an expandable manner.

TeXhackers note: `\lua_now_x:n` is the LuaTeX primitive `\directlua` renamed.

<code>\lua_shipout:n</code>		<code>\lua_shipout:n {⟨token list⟩}</code>
-----------------------------	--	--

<code>\lua_shipout:x</code>	
-----------------------------	--

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.

TeXhackers note: At a TeX level, the `⟨Lua input⟩` is stored as a “whatsit”.

<code>\lua_shipout_x:n</code> <code>\lua_shipout_x:x</code>	<code>\lua_shipout:n {⟨token list⟩}</code> <p>The <i>⟨token list⟩</i> 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 <i>⟨Lua input⟩</i> is passed to the Lua interpreter when the current page is finalised (<i>i.e.</i> at shipout). Each <code>\lua_shipout:n</code> block is treated by Lua as a separate chunk. The Lua interpreter will execute the <i>⟨Lua input⟩</i> during the page-building routine: the <i>⟨Lua input⟩</i> is expanded during this process in addition to any expansion when the argument was read. This makes these functions suitable for including material finalised during the page building process (such as the page number).</p>
--	--

\TeX hackers note: `\lua_shipout_x:n` is the Lua \TeX primitive `\latelua` named using the \LaTeX 3 scheme.

At a \TeX level, the *⟨Lua input⟩* is stored as a “whatsit”.

2 Category code tables

As well as providing methods to break out into Lua, there are places where additional \LaTeX 3 functions are provided by the Lua \TeX engine. In particular, Lua \TeX provides category code tables. These can be used to ensure that a set of category codes are in force in a more robust way than is possible with other engines. These are therefore used by `\ExplSyntaxOn` and `\ExplSyntaxOff` when using the Lua \TeX engine.

<code>\cctab_new:N</code>	<code>\cctab_new:N ⟨category code table⟩</code> <p>Creates a new category code table, initially with the codes as used by <code>\iniTeX</code>.</p>
---------------------------	--

<code>\cctab_gset:Nn</code>	<code>\cctab_gset:Nn ⟨category code table⟩ {⟨category code set up⟩}</code> <p>Sets the <i>⟨category code table⟩</i> to apply the category codes which apply when the prevailing régime is modified by the <i>⟨category code set up⟩</i>. Thus within a standard code block the starting point will be the code applied by <code>\c_code_cctab</code>. The assignment of the table is global: the underlying primitive does not respect grouping.</p>
-----------------------------	---

<code>\cctab_begin:N</code>	<code>\cctab_begin:N ⟨category code table⟩</code> <p>Switches the category codes in force to those stored in the <i>⟨category code table⟩</i>. The prevailing codes before the function is called are added to a stack, for use with <code>\cctab_end:</code>.</p>
-----------------------------	---

<code>\cctab_end:</code>	<code>\cctab_end:</code> <p>Ends the scope of a <i>⟨category code table⟩</i> started using <code>\cctab_begin:N</code>, retuning the codes to those in force before the matching <code>\cctab_begin:N</code> was used.</p>
--------------------------	---

<code>\c_code_cctab</code>	<p>Category code table for the code environment. This does not include setting the behaviour of the line-end character, which is only altered by <code>\ExplSyntaxOn</code>.</p>
----------------------------	--

<u><code>\c_document_cctab</code></u>	Category code table for a standard L ^A T _E X document. This does not include setting the behaviour of the line-end character, which is only altered by <code>\ExplSyntaxOff</code> .
<u><code>\c_initex_cctab</code></u>	Category code table as set up by <code>iniT_EX</code> .
<u><code>\c_other_cctab</code></u>	Category code table where all characters have category code 12 (other).
<u><code>\c_str_cctab</code></u>	Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space).

Part XXIII

The l3candidates package

Experimental additions to l3kernel

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.

1 Additions to l3basics

`\cs_if_exist_use:NTF` ★
`\cs_if_exist_use:cTF` ★

`\cs_if_exist_use:NTF` $\langle control\ sequence \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

If the $\langle control\ sequence \rangle$ exists, leave it in the input stream, followed by the $\langle true\ code \rangle$ (unbraced). Otherwise, leave the $\langle false \rangle$ code in the input stream. For example,

```
\cs_set:Npn \mypkg_use_character:N #1
{ \cs_if_exist_use:cF { mypkg_#1:n } { \mypkg_default:N #1 } }
```

calls the function `\mypkg_#1:n` if it exists, and falls back to a default action otherwise. This could also be done (more slowly) using `\str_case_x:nnn`.

TeXhackers note: The `c` variants do not introduce the $\langle control\ sequence \rangle$ in the hash table if it is not there.

2 Additions to l3box

2.1 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in TeX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.

`\box_resize:Nnn`
`\box_resize:cnn`

`\box_resize:Nnn` $\langle box \rangle$ $\{\langle x-size \rangle\}$ $\{\langle y-size \rangle\}$

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. The resizing applies within the current TeX 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. The resizing applies within the current \TeX 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. The resizing applies within the current \TeX group level.

<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. The scaling applies within the current \TeX group level.

2.2 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 \TeX group level.

These functions require the \LaTeX 3 native drivers: they will not work with the \LaTeX 2_ϵ graphics drivers!

\TeX hackers note: Clipping is implemented by the driver, and as such the full content of the box is places 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 `<box>` `<left>` is removed from the left-hand edge of the bounding box, `<right>` 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 `<box>` will be an hbox, irrespective of the nature of the `<box>` 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 `<box>` such that it has lower-left co-ordinates (`<llx>`, `<lly>`) and upper-right co-ordinates (`<urx>`, `<ury>`). 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 `<box>` will be an hbox, irrespective of the nature of the `<box>` before the viewport operation is applied. The adjustment applies within the current T_EX group level.

2.3 Internal variables

<code>\l__box_angle_fp</code>	The angle through which a box is rotated by <code>\box_rotate:Nn</code> , given in degrees counter-clockwise. This value is required by the underlying driver code in <code>l3driver</code> to carry out the driver-dependent part of box rotation.
-------------------------------	---

<code>\l__box_cos_fp</code>	The sine and cosine of the angle through which a box is rotated by <code>\box_rotate:Nn</code> : the values refer to the angle counter-clockwise. 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_sin_fp</code>	

<code>\l__box_scale_x_fp</code>	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>	

<code>\l__box_internal_box</code>	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.
-----------------------------------	---

3 Additions to l3clist

<hr/> <code>\clist_item:Nn</code> ★	<code>\clist_item:Nn <comma list> {<integer expression>}</code>
<code>\clist_item:(cn nn)</code> ★	Indexing items in the <i><comma list></i> from 1 at the top (left), this function will evaluate the <i><integer expression></i> and leave the appropriate item from the comma list in the input stream. If the <i><integer expression></i> is negative, indexing occurs from the bottom (right) of the comma list. When the <i><integer expression></i> is larger than the number of items in the <i><comma list></i> (as calculated by <code>\clist_count:N</code>) 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 *<item>* will not expand further when appearing in an x-type argument expansion.

<hr/> <code>\clist_set_from_seq:NN</code>	<code>\clist_set_from_seq:NN <comma list> <sequence></code>
<code>\clist_set_from_seq:(cn Nc cc)</code>	
<hr/> <code>\clist_gset_from_seq:NN</code>	
<code>\clist_gset_from_seq:(cn Nc cc)</code>	

Sets the *<comma list>* to be equal to the content of the *<sequence>*. Items which contain either spaces or commas are surrounded by braces.

<hr/> <code>\clist_const:Nn</code>	<code>\clist_const:Nn <clist var> {<comma list>}</code>
<code>\clist_const:(Nx cn cx)</code>	Creates a new constant <i><clist var></i> or raises an error if the name is already taken. The value of the <i><clist var></i> will be set globally to the <i><comma list></i> .

<hr/> <code>\clist_if_empty_p:n</code> ★	<code>\clist_if_empty_p:n {<comma list>}</code>
<code>\clist_if_empty:nTF</code> ★	<code>\clist_if_empty:nTF {<comma list>} {<true code>} {<false code>}</code>

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.

4 Additions to l3coffins

<hr/> <code>\coffin_resize:Nnn</code>	<code>\coffin_resize:Nnn <coffin> {<width>} {<total-height>}</code>
<code>\coffin_resize:cnn</code>	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>	<code>\coffin_rotate:Nn <coffin> {<angle>}</code>
<code>\coffin_rotate:cn</code>	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.

`\coffin_scale:Nnn`
`\coffin_scale:cnn`

`\coffin_scale:Nnn` $\langle coffin \rangle$ $\{\langle x-scale \rangle\}$ $\{\langle y-scale \rangle\}$

Scales the $\langle coffin \rangle$ by a factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

5 Additions to l3file

`\ior_map_inline:Nn`

New: 2012-02-11

`\ior_map_inline:Nn` $\langle stream \rangle$ $\{\langle inline function \rangle\}$

Applies the $\langle inline function \rangle$ to $\langle lines \rangle$ obtained by reading one or more lines (until an equal number of left and right braces are found) from the $\langle stream \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle line \rangle$ as #1. Note that T_EX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. T_EX also ignores any trailing new-line marker from the file it reads.

`\ior_str_map_inline:Nn`

New: 2012-02-11

`\ior_str_map_inline:Nn` $\{\langle stream \rangle\}$ $\{\langle inline function \rangle\}$

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

`\ior_map_break:`

New: 2012-06-29

`\ior_map_break:`

Used to terminate a `\ior_map...` function before all lines from the $\langle stream \rangle$ have been processed. This 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 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.

\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 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.

6 Additions to l3fp

\fp_set_from_dim:Nn**\fp_set_from_dim:cn****\fp_gset_from_dim:Nn****\fp_gset_from_dim:cn**

\fp_set_from_dim:Nn <floating point variable> {<dimexpr>}

Sets the *<floating point variable>* to the distance represented by the *<dimension expression>* in the units points. This means that distances given in other units are first converted to points before being assigned to the *<floating point variable>*.

7 Additions to l3prop

\prop_map_tokens:Nn ☆**\prop_map_tokens:cn** ☆

\prop_map_tokens:Nn <property list> {<code>}

Analogue of `\prop_map_function:NN` which maps several tokens instead of a single function. The *<code>* receives each key-value pair in the *<property list>* as two trailing brace groups. For instance,

```
\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }
```

will expand to the value corresponding to `mykey`: for each pair in `\l_my_prop` the function `\str_if_eq:nnT` receives `mykey`, the *<key>* and the *<value>* as its three arguments. For that specific task, `\prop_get:Nn` is faster.

<code>\prop_get:Nn</code> ★	<code>\prop_get:Nn</code> $\langle property\ list \rangle$ $\{\langle key \rangle\}$
<code>\prop_get: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.

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.

8 Additions to l3seq

<code>\seq_item:Nn</code> ★	<code>\seq_item:Nn</code> $\langle sequence \rangle$ $\{\langle integer\ expression \rangle\}$
<code>\seq_item:cn</code> ★	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 <code>\seq_count:N</code>) 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.

<code>\seq_mapthread_function:NNN</code> ★	<code>\seq_mapthread_function:NNN</code> $\langle seq_1 \rangle$ $\langle seq_2 \rangle$ $\langle function \rangle$
<code>\seq_mapthread_function:(NcN cNN ccN)</code> ★	

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).

<code>\seq_set_from_clist:NN</code>	<code>\seq_set_from_clist:NN</code> $\langle sequence \rangle$ $\langle comma\text{-}list \rangle$
<code>\seq_set_from_clist:(cN Nc cc Nn cn)</code>	
<code>\seq_gset_from_clist:NN</code>	
<code>\seq_gset_from_clist:(cN Nc cc Nn cn)</code>	

Sets the $\langle sequence \rangle$ within the current T_EX group to be equal to the content of the $\langle comma\text{-}list \rangle$.

<code>\seq_reverse:N</code>	<code>\seq_reverse:N</code> $\langle sequence \rangle$
<code>\seq_greverse:N</code>	Reverses the order of items in the $\langle sequence \rangle$, and assigns the result to $\langle sequence \rangle$, locally or globally according to the variant chosen.

`\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.

9 Additions to l3skip

`\dim_to_pt:n` ★

New: 2013-05-06

`\dim_to_pt:n` $\{\langle dimexpr \rangle\}$

Evaluates the $\langle dimension\ expression \rangle$, and leaves the result, expressed in points (`pt`) in the input stream, with *no units*. The result is rounded by 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.

If the $\langle dimension\ expression \rangle$ contains additional tokens such as redundant units, these will be ignored, so for example

`\dim_to_pt:n { 1 bp pt }`

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

`\dim_to_unit:nn` ★

New: 2013-05-06

`\dim_to_unit:nn` $\{\langle dimexpr_1 \rangle\}$ $\{\langle dimexpr_2 \rangle\}$

Evaluates the $\langle dimension\ expressions \rangle$, and leaves the value of $\langle dimexpr_1 \rangle$, expressed in a unit given by $\langle dimexpr_2 \rangle$, 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.

If the $\langle dimension\ expressions \rangle$ contain additional tokens such as redundant units, these will be ignored, so for example

`\dim_to_unit:nn { 1 bp pt } { 1 mm }`

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

<code>\skip_split_finite_else_action:nnNN</code>	<code>\skip_split_finite_else_action:nnNN {<skipexpr>} {<action>}</code>
	<code><dimen₁> <dimen₂></code>

Checks if the `<skipexpr>` contains finite glue. If it does then it assigns `<dimen1>` the stretch component and `<dimen2>` the shrink component. If it contains infinite glue set `<dimen1>` and `<dimen2>` to 0pt and place #2 into the input stream: this is usually an error or warning message of some sort.

10 Additions to l3tl

<code>\tl_if_single_token_p:n</code> ★	<code>\tl_if_single_token_p:n {<token list>}</code>
<code>\tl_if_single_token:nTF</code> ★	<code>\tl_if_single_token:nTF {<token list>} {<true code>} {<false code>}</code>

Tests if the token list consists of exactly one token, *i.e.* is either a single space character or a single “normal” token. Token groups (`{...}`) are not single tokens.

<code>\tl_reverse_tokens:n</code> ★	<code>\tl_reverse_tokens:n {<tokens>}</code>
-------------------------------------	--

This function, which works directly on T_EX tokens, reverses the order of the `<tokens>`: the first will be the last and the last will become first. Spaces are preserved. The reversal also operates within brace groups, but the braces themselves are not exchanged, as this would lead to an unbalanced token list. For instance, `\tl_reverse_tokens:n {a~{b()}}` leaves `{(b)~a` 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.

<code>\tl_count_tokens:n</code> ★	<code>\tl_count_tokens:n {<tokens>}</code>
-----------------------------------	--

Counts the number of T_EX tokens in the `<tokens>` and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of `a~{bc}` is 6. This function requires three expansions, giving an *<integer denotation>*.

<code>\tl_expandable_uppercase:n</code> ★	<code>\tl_expandable_uppercase:n {<tokens>}</code>
<code>\tl_expandable_lowercase:n</code> ★	<code>\tl_expandable_lowercase:n {<tokens>}</code>

The `\tl_expandable_uppercase:n` function works through all of the `<tokens>`, replacing characters in the range a–z (with arbitrary category code) by the corresponding letter in the range A–Z, with category code 11 (letter). Similarly, `\tl_expandable_lowercase:n` replaces characters in the range A–Z by letters in the range a–z, and leaves other tokens unchanged. This function requires two steps of expansion.

T_EXhackers note: Begin-group and end-group characters are normalized and become `{` and `}`, respectively. 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.

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

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.

11 Additions to l3tokens

<code>\char_set_active:Npn</code>	<code>\char_set_active:Npn <char> <parameters> {<code>}</code>
<code>\char_set_active:Npx</code>	

Makes *<char>* an active character to expand to *<code>* as replacement text. Within the *<code>*, the *<parameters>* (*#1*, *#2*, *etc.*) will be replaced by those absorbed. The *<char>* is made active within the current T_EX group level, and the definition is also local.

<code>\char_gset_active:Npn</code>	<code>\char_gset_active:Npn <char> <parameters> {<code>}</code>
<code>\char_gset_active:Npx</code>	

Makes *<char>* an active character to expand to *<code>* as replacement text. Within the *<code>*, the *<parameters>* (*#1*, *#2*, *etc.*) will be replaced by those absorbed. The *<char>* is made active within the current T_EX group level, but the definition is global. This function is therefore suited to cases where an active character definition should be applied only in some context (where the *<char>* is again made active).

<code>\char_set_active_eq:NN</code>	<code>\char_set_active_eq:NN <char> <function></code>
-------------------------------------	---

Makes *<char>* an active character equivalent in meaning to the *<function>* (which may itself be an active character). The *<char>* is made active within the current T_EX group level, and the definition is also local.

<code>\char_gset_active_eq:NN</code>	<code>\char_gset_active_eq:NN <char> <function></code>
--------------------------------------	--

Makes *<char>* an active character equivalent in meaning to the *<function>* (which may itself be an active character). The *<char>* is made active within the current T_EX group level, but the definition is global. This function is therefore suited to cases where an active character definition should be applied only in some context (where the *<char>* is again made active).

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

Updated: 2012-12-20

Tests if the next $\langle token \rangle$ in the input stream can be safely grabbed as an N-type argument. The test will be $\langle false \rangle$ if the next $\langle token \rangle$ is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), or an outer token (never used in L^AT_EX3) and $\langle true \rangle$ in all other cases. Note that a $\langle true \rangle$ result ensures that the next $\langle token \rangle$ is a valid N-type argument. However, if the next $\langle token \rangle$ is for instance `\c_space_token`, the test will take the $\langle false \rangle$ branch, even though the next $\langle token \rangle$ is in fact a valid N-type argument. The $\langle token \rangle$ will be left in the input stream after the $\langle true code \rangle$ or $\langle false code \rangle$ (as appropriate to the result of the test).

Part XXIV

Implementation

1 l3bootstrap implementation

```
1 \*initex | package>
2 \@@=expl>
```

1.1 Format-specific code

The very first thing to do is to bootstrap the iniT_EX system so that everything else will actually work. T_EX does not start with some pretty basic character codes set up.

```
3 \*initex>
4 \catcode '\{ = 1 \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 LuaT_EX the extra primitives need to be enabled before they can be used. No `\ifdefined` yet, so do it the old-fashioned way. The primitive `\strcmp` is simulated using some Lua code, which currently has to be applied to every job as the Lua code is not part of the format. Thanks to Taco Hoekwater for this code. The odd `\csname` business is needed so that the later deletion code will work.

```
12 \*initex>
13 \begingroup\expandafter\expandafter\expandafter\endgroup
14 \expandafter\ifx\csname directlua\endcsname\relax
15 \else
16 \directlua
17 {
```

```

18 tex.enableprimitives('',tex.extraprimitives ())
19 lua.bytecode[1] = function ()
20   function strcmp (A, B)
21     if A == B then
22       tex.write("0")
23     elseif A < B then
24       tex.write("-1")
25     else
26       tex.write("1")
27     end
28   end
29 end
30 lua.bytecode[1]()
31 }
32 \everyjob\expandafter
33 {\csname\detokenize{luatex_directlua:D}\endcsname{lua.bytecode[1]()}}
34 \long\edef\pdfstrcmp#1#2%
35 {%
36   \expandafter\noexpand\csname\detokenize{luatex_directlua:D}\endcsname
37   {%
38     strcmp%
39     (%
40       "\noexpand\luaescapestring{#1}",%
41       "\noexpand\luaescapestring{#2}"%
42     )%
43   }%
44 }
45 \fi
46 \</initex>

```

1.2 Package-specific code part one

The package starts by identifying itself: the information itself is taken from the SVN Id string at the start of the source file.

```

47 \<*package>
48 \ProvidesPackage{l3bootstrap}
49 [%
50   \ExplFileDate\space v\ExplFileVersion\space
51   L3 Experimental bootstrap code%
52 ]
53 \</package>

```

For LuaTeX the functionality of the `\pdfstrcmp` primitive needs to be provided: the `pdftexmcds` package is used to do this if necessary. At present, there is also a need to deal with some low-level allocation stuff that could usefully be added to `lualatex.ini`. As it is currently not, load Heiko Oberdiek's `luatex` package instead.

```

54 \<*package>
55 \def\@tempa%
56 {%

```

```

57 \def\@tempa{}%
58 \RequirePackage{luatex}%
59 \RequirePackage{pdftexcmds}%
60 \let\pdfstrcmp\pdf@strcmp
61 }
62 \begingroup\expandafter\expandafter\expandafter\endgroup
63 \expandafter\ifx\csname directlua\endcsname\relax
64 \else
65 \expandafter\@tempa
66 \fi
67 \end{package}

```

1.3 The `\pdfstrcmp` primitive in XeTeX

Only pdfTeX has a primitive called `\pdfstrcmp`. The XeTeX version is just `\strcmp`, so there is some shuffling to do.

```

68 \begingroup\expandafter\expandafter\expandafter\endgroup
69 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
70 \let\pdfstrcmp\strcmp
71 \fi

```

1.4 Engine requirements

The code currently requires functionality equivalent to `\pdfstrcmp` in addition to ε -TeX. The former is therefore used as a test for a suitable engine.

```

72 \begingroup\expandafter\expandafter\expandafter\endgroup
73 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
74 \*package
75 \PackageError{expl3}{Required primitives not found}
76 {%
77 LaTeX3 requires the e-TeX primitives and \string\pdfstrcmp.\MessageBreak
78 \MessageBreak
79 These are available in engine versions:\MessageBreak
80 - pdfTeX 1.30\MessageBreak
81 - XeTeX 0.9994\MessageBreak
82 - LuaTeX 0.40\MessageBreak
83 or later.\MessageBreak
84 \MessageBreak
85 Loading of expl3 will abort!%
86 }
87 \expandafter\endinput
88 \end{package}
89 \*initex
90 \newlinechar'\^^J\relax
91 \errhelp{%
92 LaTeX3 requires the e-TeX primitives and \pdfstrcmp.\^^J%
93 \^^J%
94 These are available in engine versions:\^^J%
95 - pdfTeX 1.30\^^J%

```

```

96      - XeTeX 0.9994^^J%
97      - LuaTeX 0.40^^J%
98      or later.^^J%
99      ^^J%
100     For pdfTeX and XeTeX the '-etex' command-line switch is also
101     needed.^^J%
102     ^^J%
103     Format building will abort!%
104 }
105 \errmessage{Required primitives not found}%
106 \expandafter\end
107 \</initex>
108 \fi

```

1.5 Package-specific code part two

`\ExplSyntaxOff` Experimental syntax switching is set up here for the package-loading process. These are redefined in `expl3` for the package and in `l3final` for the format.

```

109 <*package>
110 \protected\edef\ExplSyntaxOff
111 {%
112   \catcode 9 = \the\catcode 9\relax
113   \catcode 32 = \the\catcode 32\relax
114   \catcode 34 = \the\catcode 34\relax
115   \catcode 38 = \the\catcode 38\relax
116   \catcode 58 = \the\catcode 58\relax
117   \catcode 94 = \the\catcode 94\relax
118   \catcode 95 = \the\catcode 95\relax
119   \catcode 124 = \the\catcode 124\relax
120   \catcode 126 = \the\catcode 126\relax
121   \endlinechar = \the\endlinechar\relax
122   \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0 \relax
123 }
124 \protected\edef\ExplSyntaxOn
125 {
126   \catcode 9 = 9 \relax
127   \catcode 32 = 9 \relax
128   \catcode 34 = 12 \relax
129   \catcode 58 = 11 \relax
130   \catcode 94 = 7 \relax
131   \catcode 95 = 11 \relax
132   \catcode 124 = 12 \relax
133   \catcode 126 = 10 \relax
134   \endlinechar = 32 \relax
135   \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 1 \relax
136 }
137 </package>

```

(End definition for `\ExplSyntaxOff` and `\ExplSyntaxOn`. These functions are documented on page 6.)

`\l__kernel_expl_bool` The status for experimental code syntax: this is off at present. This code is used by both the package and the format.

```

138 \expandafter\chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0 \relax

```

(End definition for `\l__kernel_expl_bool`. This variable is documented on page 7.)

1.6 Dealing with package-mode meta-data

`\GetIdInfo` This is implemented right at the start of `l3bootstrap.dtx`.
(End definition for `\GetIdInfo`. This function is documented on page 6.)

`\ProvidesExplPackage`
`\ProvidesExplClass`
`\ProvidesExplFile` For other packages and classes building on this one it is convenient not to need `\ExplSyntaxOn` each time.

```

139 <*package>
140 \protected\def\ProvidesExplPackage
141   {%
142     \@ifpackageloaded{expl3}
143     {}
144     {%
145       \PackageError{expl3}
146         {Cannot load the expl3 modules separately}
147         {%
148           The expl3 modules cannot be loaded separately;\MessageBreak
149           please \string\usepackage\string{expl3\string} instead.%
150         }%
151     }%
152   \protected\def\ProvidesExplPackage##1##2##3##4%
153     {%
154       \ProvidesPackage{##1}[##2 v##3 ##4]%
155       \ExplSyntaxOn
156     }%
157   \ProvidesExplPackage
158   }
159 \protected\def\ProvidesExplClass#1#2#3#4%
160   {%
161     \ProvidesClass{#1}[#2 v#3 #4]%
162     \ExplSyntaxOn
163   }
164 \protected\def\ProvidesExplFile#1#2#3#4%
165   {%
166     \ProvidesFile{#1}[#2 v#3 #4]%
167     \ExplSyntaxOn
168   }
169 </package>

```

(End definition for `\ProvidesExplPackage`, `\ProvidesExplClass`, and `\ProvidesExplFile`. These functions are documented on page 6.)

`\@pushfilename`
`\@popfilename` The idea here is to use L^AT_EX 2_ε's `\@pushfilename` and `\@popfilename` to track the current syntax status. This can be achieved by saving the current status flag at each

push to a stack, then recovering it at the pop stage and checking if the code environment should still be active.

```

170 <*package>
171 \edef\@pushfilename
172 {%
173   \edef\expandafter\noexpand
174   \csname\detokenize{l__expl_status_stack_tl}\endcsname
175   {%
176     \noexpand\ifodd\expandafter\noexpand
177     \csname\detokenize{l__kernel_expl_bool}\endcsname
178     1%
179     \noexpand\else
180     0%
181     \noexpand\fi
182     \expandafter\noexpand
183     \csname\detokenize{l__expl_status_stack_tl}\endcsname
184   }%
185   \ExplSyntaxOff
186   \unexpanded\expandafter{\@pushfilename}%
187 }
188 \edef\@popfilename
189 {%
190   \unexpanded\expandafter{\@popfilename}%
191   \noexpand\if a\expandafter\noexpand\csname
192     \detokenize{l__expl_status_stack_tl}\endcsname a%
193   \ExplSyntaxOff
194   \noexpand\else
195     \noexpand\expandafter
196     \expandafter\noexpand\csname
197       \detokenize{__expl_status_pop:w}\endcsname
198     \expandafter\noexpand\csname
199       \detokenize{l__expl_status_stack_tl}\endcsname
200     \noexpand\@nil
201   \noexpand\fi
202 }
203 </package>

```

(End definition for \@pushfilename and \@popfilename. These functions are documented on page ??.)

`\l__expl_status_stack_tl` As expl3 itself cannot be loaded with the code environment already active, at the end of the package `\ExplSyntaxOff` can safely be called.

```

204 <*package>
205 \@namedef{\detokenize{l__expl_status_stack_tl}}{0}
206 </package>

```

(End definition for `\l__expl_status_stack_tl`. This function is documented on page ??.)

`__expl_status_pop:w` The pop auxiliary function removes the first item from the stack, saves the rest of the stack and then does the test. As `\ExplSyntaxOff` is already defined as a protected macro, there is no need for `\noexpand` here.

```

207 <*package>

```

```

208 \expandafter\edef\csname\detokenize{__expl_status_pop:w}\endcsname#1#2\@nil
209 {%
210   \def\expandafter\noexpand
211     \csname\detokenize{l__expl_status_stack_tl}\endcsname{#2}%
212   \noexpand\ifodd#1\space
213     \noexpand\expandafter\noexpand\ExplSyntaxOn
214   \noexpand\else
215     \noexpand\expandafter\ExplSyntaxOff
216   \noexpand\fi
217 }
218 \</package>
(End definition for \__expl_status_pop:w.)

```

`__expl_package_check:` We want the expl3 bundle to be loaded “as one”; this command is used to ensure that one of the 13 packages isn’t loaded on its own.

```

219 \<*package>
220 \expandafter\protected\expandafter\def
221   \csname\detokenize{__expl_package_check:}\endcsname
222   {%
223     \@ifpackageloaded{expl3}
224     {}
225     {%
226       \PackageError{expl3}
227         {Cannot load the expl3 modules separately}
228         {%
229           The expl3 modules cannot be loaded separately;\MessageBreak
230           please \string\usepackage\string{expl3\string} instead.%
231         }%
232     }%
233   }
234 \</package>
(End definition for \__expl_package_check:.)

```

1.7 The L^AT_EX3 code environment

The code environment is now set up for the format: the package deals with this using `\ProvidesExplPackage`.

```

235 \<*initex>
236 \catcode 9 = 9 \relax
237 \catcode 32 = 9 \relax
238 \catcode 34 = 12 \relax
239 \catcode 58 = 11 \relax
240 \catcode 94 = 7 \relax
241 \catcode 95 = 11 \relax
242 \catcode 124 = 12 \relax
243 \catcode 126 = 10 \relax
244 \endlinechar = 32 \relax
245 \</initex>

```

`\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.

```

246 <*initex>
247 \protected \def \ExplSyntaxOn
248 {
249   \bool_if:NF \l__kernel_expl_bool
250   {
251     \cs_set_protected_nopar:Npx \ExplSyntaxOff
252     {
253       \char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
254       \char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
255       \char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
256       \char_set_catcode:nn { 38 } { \char_value_catcode:n { 38 } }
257       \char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
258       \char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
259       \char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }
260       \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
261       \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
262       \tex_endlinechar:D =
263       \tex_the:D \tex_endlinechar:D \scan_stop:
264       \bool_set_false:N \l__kernel_expl_bool
265       \cs_set_protected_nopar:Npn \ExplSyntaxOff { }
266     }
267   }
268   \char_set_catcode_ignore:n { 9 } % tab
269   \char_set_catcode_ignore:n { 32 } % space
270   \char_set_catcode_other:n { 34 } % double quote
271   \char_set_catcode_alignment:n { 38 } % ampersand
272   \char_set_catcode_letter:n { 58 } % colon
273   \char_set_catcode_math_superscript:n { 94 } % circumflex
274   \char_set_catcode_letter:n { 95 } % underscore
275   \char_set_catcode_other:n { 124 } % pipe
276   \char_set_catcode_space:n { 126 } % tilde
277   \tex_endlinechar:D = 32 \scan_stop:
278   \bool_set_true:N \l__kernel_expl_bool
279 }
280 \protected \def \ExplSyntaxOff { }
281 </initex>

```

(End definition for `\ExplSyntaxOn` and `\ExplSyntaxOff`. These functions are documented on page 6.)

`\l__kernel_expl_bool` A flag to show the current syntax status.

```

282 <*initex>
283 \chardef \l__kernel_expl_bool = 0 ~
284 </initex>

```

(End definition for `\l__kernel_expl_bool`. This variable is documented on page 7.)

```

285 </initex | package>

```

2 l3names implementation

```

286 <*initex | package>
287 <*package>
288 \ProvidesExplPackage
289   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
290 </package>

```

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.

```

291 \let \tex_global:D \global
292 \let \tex_let:D \let

```

Everything is inside a (rather long) group, which keeps `__expl_primitive:NN` trapped.

```

293 \begingroup

```

`__expl_primitive:NN` A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

```

294 \long \def \__expl_primitive:NN #1#2
295 {
296   \tex_global:D \tex_let:D #2 #1
297 <*initex>
298   \tex_global:D \tex_let:D #1 \tex_undefined:D
299 </initex>
300 }

```

(End definition for __expl_primitive:NN.)

In the current incarnation of this package, all T_EX 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.

```

301 \__expl_primitive:NN \tex_space:D
302 \__expl_primitive:NN \tex_italiccorrection:D
303 \__expl_primitive:NN \tex_hyphen:D

```

Now all the other primitives.

```

304 \__expl_primitive:NN \tex_let:D
305 \__expl_primitive:NN \tex_def:D
306 \__expl_primitive:NN \tex_edef:D
307 \__expl_primitive:NN \tex_gdef:D
308 \__expl_primitive:NN \tex_xdef:D
309 \__expl_primitive:NN \tex_chardef:D
310 \__expl_primitive:NN \tex_countdef:D

```

311	_expl_primitive:NN	\dimendef	\tex_dimendef:D
312	_expl_primitive:NN	\skipdef	\tex_skipdef:D
313	_expl_primitive:NN	\muskipdef	\tex_muskipdef:D
314	_expl_primitive:NN	\mathchardef	\tex_mathchardef:D
315	_expl_primitive:NN	\toksdef	\tex_toksdef:D
316	_expl_primitive:NN	\futurelet	\tex_futurelet:D
317	_expl_primitive:NN	\advance	\tex_advance:D
318	_expl_primitive:NN	\divide	\tex_divide:D
319	_expl_primitive:NN	\multiply	\tex_multiply:D
320	_expl_primitive:NN	\font	\tex_font:D
321	_expl_primitive:NN	\fam	\tex_fam:D
322	_expl_primitive:NN	\global	\tex_global:D
323	_expl_primitive:NN	\long	\tex_long:D
324	_expl_primitive:NN	\outer	\tex_outer:D
325	_expl_primitive:NN	\setlanguage	\tex_setlanguage:D
326	_expl_primitive:NN	\globaldefs	\tex_globaldefs:D
327	_expl_primitive:NN	\afterassignment	\tex_afterassignment:D
328	_expl_primitive:NN	\aftergroup	\tex_aftergroup:D
329	_expl_primitive:NN	\expandafter	\tex_expandafter:D
330	_expl_primitive:NN	\noexpand	\tex_noexpand:D
331	_expl_primitive:NN	\begingroup	\tex_begingroup:D
332	_expl_primitive:NN	\endgroup	\tex_endgroup:D
333	_expl_primitive:NN	\halign	\tex_halign:D
334	_expl_primitive:NN	\valign	\tex_valign:D
335	_expl_primitive:NN	\cr	\tex_cr:D
336	_expl_primitive:NN	\crcr	\tex_crcr:D
337	_expl_primitive:NN	\noalign	\tex_noalign:D
338	_expl_primitive:NN	\omit	\tex_omit:D
339	_expl_primitive:NN	\span	\tex_span:D
340	_expl_primitive:NN	\tabskip	\tex_tabskip:D
341	_expl_primitive:NN	\everycr	\tex_everycr:D
342	_expl_primitive:NN	\if	\tex_if:D
343	_expl_primitive:NN	\ifcase	\tex_ifcase:D
344	_expl_primitive:NN	\ifcat	\tex_ifcat:D
345	_expl_primitive:NN	\ifnum	\tex_ifnum:D
346	_expl_primitive:NN	\ifodd	\tex_ifodd:D
347	_expl_primitive:NN	\ifdim	\tex_ifdim:D
348	_expl_primitive:NN	\ifeof	\tex_ifeof:D
349	_expl_primitive:NN	\ifhbox	\tex_ifhbox:D
350	_expl_primitive:NN	\ifvbox	\tex_ifvbox:D
351	_expl_primitive:NN	\ifvoid	\tex_ifvoid:D
352	_expl_primitive:NN	\ifx	\tex_ifx:D
353	_expl_primitive:NN	\iffalse	\tex_iffalse:D
354	_expl_primitive:NN	\iftrue	\tex_iftrue:D
355	_expl_primitive:NN	\ifhmode	\tex_ifhmode:D
356	_expl_primitive:NN	\ifmmode	\tex_ifmmode:D
357	_expl_primitive:NN	\ifvmode	\tex_ifvmode:D
358	_expl_primitive:NN	\ifinner	\tex_ifinner:D
359	_expl_primitive:NN	\else	\tex_else:D
360	_expl_primitive:NN	\fi	\tex_fi:D

361	_expl_primitive:NN	\or	\tex_or:D
362	_expl_primitive:NN	\immediate	\tex_immediate:D
363	_expl_primitive:NN	\closeout	\tex_closeout:D
364	_expl_primitive:NN	\openin	\tex_openin:D
365	_expl_primitive:NN	\openout	\tex_openout:D
366	_expl_primitive:NN	\read	\tex_read:D
367	_expl_primitive:NN	\write	\tex_write:D
368	_expl_primitive:NN	\closein	\tex_closein:D
369	_expl_primitive:NN	\newlinechar	\tex_newlinechar:D
370	_expl_primitive:NN	\input	\tex_input:D
371	_expl_primitive:NN	\endinput	\tex_endinput:D
372	_expl_primitive:NN	\inputlineno	\tex_inputlineno:D
373	_expl_primitive:NN	\errmessage	\tex_errmessage:D
374	_expl_primitive:NN	\message	\tex_message:D
375	_expl_primitive:NN	\show	\tex_show:D
376	_expl_primitive:NN	\showthe	\tex_showthe:D
377	_expl_primitive:NN	\showbox	\tex_showbox:D
378	_expl_primitive:NN	\showlists	\tex_showlists:D
379	_expl_primitive:NN	\errhelp	\tex_errhelp:D
380	_expl_primitive:NN	\errorcontextlines	\tex_errorcontextlines:D
381	_expl_primitive:NN	\tracingcommands	\tex_tracingcommands:D
382	_expl_primitive:NN	\tracinglostchars	\tex_tracinglostchars:D
383	_expl_primitive:NN	\tracingmacros	\tex_tracingmacros:D
384	_expl_primitive:NN	\tracingonline	\tex_tracingonline:D
385	_expl_primitive:NN	\tracingoutput	\tex_tracingoutput:D
386	_expl_primitive:NN	\tracingpages	\tex_tracingpages:D
387	_expl_primitive:NN	\tracingparagraphs	\tex_tracingparagraphs:D
388	_expl_primitive:NN	\tracingrestores	\tex_tracingrestores:D
389	_expl_primitive:NN	\tracingstats	\tex_tracingstats:D
390	_expl_primitive:NN	\pausing	\tex_pausing:D
391	_expl_primitive:NN	\showboxbreadth	\tex_showboxbreadth:D
392	_expl_primitive:NN	\showboxdepth	\tex_showboxdepth:D
393	_expl_primitive:NN	\batchmode	\tex_batchmode:D
394	_expl_primitive:NN	\errorstopmode	\tex_errorstopmode:D
395	_expl_primitive:NN	\nonstopmode	\tex_nonstopmode:D
396	_expl_primitive:NN	\scrollmode	\tex_scrollmode:D
397	_expl_primitive:NN	\end	\tex_end:D
398	_expl_primitive:NN	\csname	\tex_csname:D
399	_expl_primitive:NN	\endcsname	\tex_endcsname:D
400	_expl_primitive:NN	\ignorespaces	\tex_ignorespaces:D
401	_expl_primitive:NN	\relax	\tex_relax:D
402	_expl_primitive:NN	\the	\tex_the:D
403	_expl_primitive:NN	\mag	\tex_mag:D
404	_expl_primitive:NN	\language	\tex_language:D
405	_expl_primitive:NN	\mark	\tex_mark:D
406	_expl_primitive:NN	\topmark	\tex_topmark:D
407	_expl_primitive:NN	\firstmark	\tex_firstmark:D
408	_expl_primitive:NN	\botmark	\tex_botmark:D
409	_expl_primitive:NN	\splitfirstmark	\tex_splitfirstmark:D
410	_expl_primitive:NN	\splitbotmark	\tex_splitbotmark:D

411	_expl_primitive:NN	\fontname	\tex_fontname:D
412	_expl_primitive:NN	\escapechar	\tex_escapechar:D
413	_expl_primitive:NN	\endlinechar	\tex_endlinechar:D
414	_expl_primitive:NN	\mathchoice	\tex_mathchoice:D
415	_expl_primitive:NN	\delimiter	\tex_delimiter:D
416	_expl_primitive:NN	\mathaccent	\tex_mathaccent:D
417	_expl_primitive:NN	\mathchar	\tex_mathchar:D
418	_expl_primitive:NN	\mskip	\tex_mskip:D
419	_expl_primitive:NN	\radical	\tex_radical:D
420	_expl_primitive:NN	\vcenter	\tex_vcenter:D
421	_expl_primitive:NN	\mkern	\tex_mkern:D
422	_expl_primitive:NN	\above	\tex_above:D
423	_expl_primitive:NN	\abovewithdelims	\tex_abovewithdelims:D
424	_expl_primitive:NN	\atop	\tex_atop:D
425	_expl_primitive:NN	\atopwithdelims	\tex_atopwithdelims:D
426	_expl_primitive:NN	\over	\tex_over:D
427	_expl_primitive:NN	\overwithdelims	\tex_overwithdelims:D
428	_expl_primitive:NN	\displaystyle	\tex_displaystyle:D
429	_expl_primitive:NN	\textstyle	\tex_textstyle:D
430	_expl_primitive:NN	\scriptstyle	\tex_scriptstyle:D
431	_expl_primitive:NN	\scriptscriptstyle	\tex_scriptscriptstyle:D
432	_expl_primitive:NN	\nonscript	\tex_nonscript:D
433	_expl_primitive:NN	\eqno	\tex_eqno:D
434	_expl_primitive:NN	\leqno	\tex_leqno:D
435	_expl_primitive:NN	\abovedisplayshortskip	\tex_abovedisplayshortskip:D
436	_expl_primitive:NN	\abovedisplayskip	\tex_abovedisplayskip:D
437	_expl_primitive:NN	\belowdisplayshortskip	\tex_belowdisplayshortskip:D
438	_expl_primitive:NN	\belowdisplayskip	\tex_belowdisplayskip:D
439	_expl_primitive:NN	\displaywidowpenalty	\tex_displaywidowpenalty:D
440	_expl_primitive:NN	\displayindent	\tex_displayindent:D
441	_expl_primitive:NN	\displaywidth	\tex_displaywidth:D
442	_expl_primitive:NN	\everydisplay	\tex_everydisplay:D
443	_expl_primitive:NN	\predisplaysize	\tex_predisplaysize:D
444	_expl_primitive:NN	\predisplaypenalty	\tex_predisplaypenalty:D
445	_expl_primitive:NN	\postdisplaypenalty	\tex_postdisplaypenalty:D
446	_expl_primitive:NN	\mathbin	\tex_mathbin:D
447	_expl_primitive:NN	\mathclose	\tex_mathclose:D
448	_expl_primitive:NN	\mathinner	\tex_mathinner:D
449	_expl_primitive:NN	\mathop	\tex_mathop:D
450	_expl_primitive:NN	\displaylimits	\tex_displaylimits:D
451	_expl_primitive:NN	\limits	\tex_limits:D
452	_expl_primitive:NN	\nolimits	\tex_nolimits:D
453	_expl_primitive:NN	\mathopen	\tex_mathopen:D
454	_expl_primitive:NN	\mathord	\tex_mathord:D
455	_expl_primitive:NN	\mathpunct	\tex_mathpunct:D
456	_expl_primitive:NN	\mathrel	\tex_mathrel:D
457	_expl_primitive:NN	\overline	\tex_overline:D
458	_expl_primitive:NN	\underline	\tex_underline:D
459	_expl_primitive:NN	\left	\tex_left:D
460	_expl_primitive:NN	\right	\tex_right:D

461	_expl_primitive:NN	\binoppenalty	\tex_binoppenalty:D
462	_expl_primitive:NN	\relpenalty	\tex_relpenalty:D
463	_expl_primitive:NN	\delimitershortfall	\tex_delimitershortfall:D
464	_expl_primitive:NN	\delimiterfactor	\tex_delimiterfactor:D
465	_expl_primitive:NN	\nulldelimiterspace	\tex_nulldelimiterspace:D
466	_expl_primitive:NN	\everymath	\tex_everymath:D
467	_expl_primitive:NN	\mathsurround	\tex_mathsurround:D
468	_expl_primitive:NN	\medmuskip	\tex_medmuskip:D
469	_expl_primitive:NN	\thinmuskip	\tex_thinmuskip:D
470	_expl_primitive:NN	\thickmuskip	\tex_thickmuskip:D
471	_expl_primitive:NN	\scriptspace	\tex_scriptspace:D
472	_expl_primitive:NN	\noboundary	\tex_noboundary:D
473	_expl_primitive:NN	\accent	\tex_accent:D
474	_expl_primitive:NN	\char	\tex_char:D
475	_expl_primitive:NN	\discretionary	\tex_discretionary:D
476	_expl_primitive:NN	\hfil	\tex_hfil:D
477	_expl_primitive:NN	\hfilneg	\tex_hfilneg:D
478	_expl_primitive:NN	\hfill	\tex_hfill:D
479	_expl_primitive:NN	\hskip	\tex_hskip:D
480	_expl_primitive:NN	\hss	\tex_hss:D
481	_expl_primitive:NN	\vfil	\tex_vfil:D
482	_expl_primitive:NN	\vfilneg	\tex_vfilneg:D
483	_expl_primitive:NN	\vfill	\tex_vfill:D
484	_expl_primitive:NN	\vskip	\tex_vskip:D
485	_expl_primitive:NN	\vss	\tex_vss:D
486	_expl_primitive:NN	\unskip	\tex_unskip:D
487	_expl_primitive:NN	\kern	\tex_kern:D
488	_expl_primitive:NN	\unkern	\tex_unkern:D
489	_expl_primitive:NN	\hrule	\tex_hrule:D
490	_expl_primitive:NN	\vrule	\tex_vrule:D
491	_expl_primitive:NN	\leaders	\tex_leaders:D
492	_expl_primitive:NN	\cleaders	\tex_cleaders:D
493	_expl_primitive:NN	\xleaders	\tex_xleaders:D
494	_expl_primitive:NN	\lastkern	\tex_lastkern:D
495	_expl_primitive:NN	\lastskip	\tex_lastskip:D
496	_expl_primitive:NN	\indent	\tex_indent:D
497	_expl_primitive:NN	\par	\tex_par:D
498	_expl_primitive:NN	\noindent	\tex_noindent:D
499	_expl_primitive:NN	\vadjust	\tex_vadjust:D
500	_expl_primitive:NN	\baselineskip	\tex_baselineskip:D
501	_expl_primitive:NN	\lineskip	\tex_lineskip:D
502	_expl_primitive:NN	\lineskiplimit	\tex_lineskiplimit:D
503	_expl_primitive:NN	\clubpenalty	\tex_clubpenalty:D
504	_expl_primitive:NN	\widowpenalty	\tex_widowpenalty:D
505	_expl_primitive:NN	\exhyphenpenalty	\tex_exhyphenpenalty:D
506	_expl_primitive:NN	\hyphenpenalty	\tex_hyphenpenalty:D
507	_expl_primitive:NN	\linepenalty	\tex_linepenalty:D
508	_expl_primitive:NN	\doublehyphendemerits	\tex_doublehyphendemerits:D
509	_expl_primitive:NN	\finalhyphendemerits	\tex_finalhyphendemerits:D
510	_expl_primitive:NN	\adjdemerits	\tex_adjdemerits:D

511	_expl_primitive:NN	\hangafter	\tex_hangafter:D
512	_expl_primitive:NN	\hangindent	\tex_hangindent:D
513	_expl_primitive:NN	\parshape	\tex_parshape:D
514	_expl_primitive:NN	\hsize	\tex_hsize:D
515	_expl_primitive:NN	\lefthyphenmin	\tex_lefthyphenmin:D
516	_expl_primitive:NN	\righthyphenmin	\tex_righthyphenmin:D
517	_expl_primitive:NN	\leftskip	\tex_leftskip:D
518	_expl_primitive:NN	\rightskip	\tex_rightskip:D
519	_expl_primitive:NN	\looseness	\tex_looseness:D
520	_expl_primitive:NN	\parskip	\tex_parskip:D
521	_expl_primitive:NN	\parindent	\tex_parindent:D
522	_expl_primitive:NN	\uchyph	\tex_uchyph:D
523	_expl_primitive:NN	\emergencystretch	\tex_emergencystretch:D
524	_expl_primitive:NN	\pretolerance	\tex_pretolerance:D
525	_expl_primitive:NN	\tolerance	\tex_tolerance:D
526	_expl_primitive:NN	\spaceskip	\tex_spaceskip:D
527	_expl_primitive:NN	\xspaceskip	\tex_xspaceskip:D
528	_expl_primitive:NN	\parfillskip	\tex_parfillskip:D
529	_expl_primitive:NN	\everypar	\tex_everypar:D
530	_expl_primitive:NN	\prevgraf	\tex_prevgraf:D
531	_expl_primitive:NN	\spacefactor	\tex_spacefactor:D
532	_expl_primitive:NN	\shipout	\tex_shipout:D
533	_expl_primitive:NN	\vsize	\tex_vsize:D
534	_expl_primitive:NN	\interlinepenalty	\tex_interlinepenalty:D
535	_expl_primitive:NN	\brokenpenalty	\tex_brokenpenalty:D
536	_expl_primitive:NN	\topskip	\tex_topskip:D
537	_expl_primitive:NN	\maxdeadcycles	\tex_maxdeadcycles:D
538	_expl_primitive:NN	\maxdepth	\tex_maxdepth:D
539	_expl_primitive:NN	\output	\tex_output:D
540	_expl_primitive:NN	\deadcycles	\tex_deadcycles:D
541	_expl_primitive:NN	\pagedepth	\tex_pagedepth:D
542	_expl_primitive:NN	\pagestretch	\tex_pagestretch:D
543	_expl_primitive:NN	\pagefilstretch	\tex_pagefilstretch:D
544	_expl_primitive:NN	\pagefillstretch	\tex_pagefillstretch:D
545	_expl_primitive:NN	\pagefilllstretch	\tex_pagefilllstretch:D
546	_expl_primitive:NN	\pageshrink	\tex_pageshrink:D
547	_expl_primitive:NN	\pagegoal	\tex_pagegoal:D
548	_expl_primitive:NN	\pagetotal	\tex_pagetotal:D
549	_expl_primitive:NN	\outputpenalty	\tex_outputpenalty:D
550	_expl_primitive:NN	\hoffset	\tex_hoffset:D
551	_expl_primitive:NN	\voffset	\tex_voffset:D
552	_expl_primitive:NN	\insert	\tex_insert:D
553	_expl_primitive:NN	\holdinginserts	\tex_holdinginserts:D
554	_expl_primitive:NN	\floatingpenalty	\tex_floatingpenalty:D
555	_expl_primitive:NN	\insertpenalties	\tex_insertpenalties:D
556	_expl_primitive:NN	\lower	\tex_lower:D
557	_expl_primitive:NN	\moveleft	\tex_moveleft:D
558	_expl_primitive:NN	\moveright	\tex_moveright:D
559	_expl_primitive:NN	\raise	\tex_raise:D
560	_expl_primitive:NN	\copy	\tex_copy:D

561	_expl_primitive:NN	\lastbox	\tex_lastbox:D
562	_expl_primitive:NN	\vsplit	\tex_vsplit:D
563	_expl_primitive:NN	\unhbox	\tex_unhbox:D
564	_expl_primitive:NN	\unhcopy	\tex_unhcopy:D
565	_expl_primitive:NN	\unvbox	\tex_unvbox:D
566	_expl_primitive:NN	\unvcopy	\tex_unvcopy:D
567	_expl_primitive:NN	\setbox	\tex_setbox:D
568	_expl_primitive:NN	\hbox	\tex_hbox:D
569	_expl_primitive:NN	\vbox	\tex_vbox:D
570	_expl_primitive:NN	\vtop	\tex_vtop:D
571	_expl_primitive:NN	\prevdepth	\tex_prevdepth:D
572	_expl_primitive:NN	\badness	\tex_badness:D
573	_expl_primitive:NN	\hbadness	\tex_hbadness:D
574	_expl_primitive:NN	\vbadness	\tex_vbadness:D
575	_expl_primitive:NN	\hfuzz	\tex_hfuzz:D
576	_expl_primitive:NN	\vfuzz	\tex_vfuzz:D
577	_expl_primitive:NN	\overfullrule	\tex_overfullrule:D
578	_expl_primitive:NN	\boxmaxdepth	\tex_boxmaxdepth:D
579	_expl_primitive:NN	\splitmaxdepth	\tex_splitmaxdepth:D
580	_expl_primitive:NN	\splittopskip	\tex_splittopskip:D
581	_expl_primitive:NN	\everyhbox	\tex_everyhbox:D
582	_expl_primitive:NN	\everyvbox	\tex_everyvbox:D
583	_expl_primitive:NN	\nullfont	\tex_nullfont:D
584	_expl_primitive:NN	\textfont	\tex_textfont:D
585	_expl_primitive:NN	\scriptfont	\tex_scriptfont:D
586	_expl_primitive:NN	\scriptscriptfont	\tex_scriptscriptfont:D
587	_expl_primitive:NN	\fontdimen	\tex_fontdimen:D
588	_expl_primitive:NN	\hyphenchar	\tex_hyphenchar:D
589	_expl_primitive:NN	\skewchar	\tex_skewchar:D
590	_expl_primitive:NN	\defaultshyphenchar	\tex_defaultshyphenchar:D
591	_expl_primitive:NN	\defaultskewchar	\tex_defaultskewchar:D
592	_expl_primitive:NN	\number	\tex_number:D
593	_expl_primitive:NN	\romannumeral	\tex_romannumeral:D
594	_expl_primitive:NN	\string	\tex_string:D
595	_expl_primitive:NN	\lowercase	\tex_lowercase:D
596	_expl_primitive:NN	\uppercase	\tex_uppercase:D
597	_expl_primitive:NN	\meaning	\tex_meaning:D
598	_expl_primitive:NN	\penalty	\tex_penalty:D
599	_expl_primitive:NN	\unpenalty	\tex_unpenalty:D
600	_expl_primitive:NN	\lastpenalty	\tex_lastpenalty:D
601	_expl_primitive:NN	\special	\tex_special:D
602	_expl_primitive:NN	\dump	\tex_dump:D
603	_expl_primitive:NN	\patterns	\tex_patterns:D
604	_expl_primitive:NN	\hyphenation	\tex_hyphenation:D
605	_expl_primitive:NN	\time	\tex_time:D
606	_expl_primitive:NN	\day	\tex_day:D
607	_expl_primitive:NN	\month	\tex_month:D
608	_expl_primitive:NN	\year	\tex_year:D
609	_expl_primitive:NN	\jobname	\tex_jobname:D
610	_expl_primitive:NN	\everyjob	\tex_everyjob:D

611	<code>_expl_primitive:NN \count</code>	<code>\tex_count:D</code>
612	<code>_expl_primitive:NN \dimen</code>	<code>\tex_dimen:D</code>
613	<code>_expl_primitive:NN \skip</code>	<code>\tex_skip:D</code>
614	<code>_expl_primitive:NN \toks</code>	<code>\tex_toks:D</code>
615	<code>_expl_primitive:NN \muskip</code>	<code>\tex_muskip:D</code>
616	<code>_expl_primitive:NN \box</code>	<code>\tex_box:D</code>
617	<code>_expl_primitive:NN \wd</code>	<code>\tex_wd:D</code>
618	<code>_expl_primitive:NN \ht</code>	<code>\tex_ht:D</code>
619	<code>_expl_primitive:NN \dp</code>	<code>\tex_dp:D</code>
620	<code>_expl_primitive:NN \catcode</code>	<code>\tex_catcode:D</code>
621	<code>_expl_primitive:NN \delcode</code>	<code>\tex_delcode:D</code>
622	<code>_expl_primitive:NN \sfcode</code>	<code>\tex_sfcode:D</code>
623	<code>_expl_primitive:NN \lccode</code>	<code>\tex_lccode:D</code>
624	<code>_expl_primitive:NN \uccode</code>	<code>\tex_uccode:D</code>
625	<code>_expl_primitive:NN \mathcode</code>	<code>\tex_mathcode:D</code>

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_`.

626	<code>_expl_primitive:NN \ifdefined</code>	<code>\etex_ifdefined:D</code>
627	<code>_expl_primitive:NN \ifcsname</code>	<code>\etex_ifcsname:D</code>
628	<code>_expl_primitive:NN \unless</code>	<code>\etex_unless:D</code>
629	<code>_expl_primitive:NN \eTeXversion</code>	<code>\etex_eTeXversion:D</code>
630	<code>_expl_primitive:NN \eTeXrevision</code>	<code>\etex_eTeXrevision:D</code>
631	<code>_expl_primitive:NN \marks</code>	<code>\etex_marks:D</code>
632	<code>_expl_primitive:NN \topmarks</code>	<code>\etex_topmarks:D</code>
633	<code>_expl_primitive:NN \firstmarks</code>	<code>\etex_firstmarks:D</code>
634	<code>_expl_primitive:NN \botmarks</code>	<code>\etex_botmarks:D</code>
635	<code>_expl_primitive:NN \splitfirstmarks</code>	<code>\etex_splitfirstmarks:D</code>
636	<code>_expl_primitive:NN \splitbotmarks</code>	<code>\etex_splitbotmarks:D</code>
637	<code>_expl_primitive:NN \unexpanded</code>	<code>\etex_unexpanded:D</code>
638	<code>_expl_primitive:NN \detokenize</code>	<code>\etex_detokenize:D</code>
639	<code>_expl_primitive:NN \scantokens</code>	<code>\etex_scantokens:D</code>
640	<code>_expl_primitive:NN \showtokens</code>	<code>\etex_showtokens:D</code>
641	<code>_expl_primitive:NN \readline</code>	<code>\etex_readline:D</code>
642	<code>_expl_primitive:NN \tracingassigns</code>	<code>\etex_tracingassigns:D</code>
643	<code>_expl_primitive:NN \tracingscantokens</code>	<code>\etex_tracingscantokens:D</code>
644	<code>_expl_primitive:NN \tracingnesting</code>	<code>\etex_tracingnesting:D</code>
645	<code>_expl_primitive:NN \tracingifs</code>	<code>\etex_tracingifs:D</code>
646	<code>_expl_primitive:NN \currentiflevel</code>	<code>\etex_currentiflevel:D</code>
647	<code>_expl_primitive:NN \currentifbranch</code>	<code>\etex_currentifbranch:D</code>
648	<code>_expl_primitive:NN \currentiftyp</code>	<code>\etex_currentiftyp:D</code>
649	<code>_expl_primitive:NN \tracinggroups</code>	<code>\etex_tracinggroups:D</code>
650	<code>_expl_primitive:NN \currentgrouplevel</code>	<code>\etex_currentgrouplevel:D</code>
651	<code>_expl_primitive:NN \currentgrouptyp</code>	<code>\etex_currentgrouptyp:D</code>
652	<code>_expl_primitive:NN \showgroups</code>	<code>\etex_showgroups:D</code>
653	<code>_expl_primitive:NN \showifs</code>	<code>\etex_showifs:D</code>
654	<code>_expl_primitive:NN \interactionmode</code>	<code>\etex_interactionmode:D</code>
655	<code>_expl_primitive:NN \lastnodetype</code>	<code>\etex_lastnodetype:D</code>
656	<code>_expl_primitive:NN \iffontchar</code>	<code>\etex_iffontchar:D</code>
657	<code>_expl_primitive:NN \fontcharht</code>	<code>\etex_fontcharht:D</code>

658	_expl_primitive:NN	\fontchardp	\etex_fontchardp:D
659	_expl_primitive:NN	\fontcharwd	\etex_fontcharwd:D
660	_expl_primitive:NN	\fontcharic	\etex_fontcharic:D
661	_expl_primitive:NN	\parshapeindent	\etex_parshapeindent:D
662	_expl_primitive:NN	\parshapelength	\etex_parshapelength:D
663	_expl_primitive:NN	\parshapedimen	\etex_parshapedimen:D
664	_expl_primitive:NN	\numexpr	\etex_numexpr:D
665	_expl_primitive:NN	\dimexpr	\etex_dimexpr:D
666	_expl_primitive:NN	\glueexpr	\etex_glueexpr:D
667	_expl_primitive:NN	\muexpr	\etex_muexpr:D
668	_expl_primitive:NN	\gluestretch	\etex_gluestretch:D
669	_expl_primitive:NN	\glueshrink	\etex_glueshrink:D
670	_expl_primitive:NN	\gluestretchorder	\etex_gluestretchorder:D
671	_expl_primitive:NN	\glueshrinkorder	\etex_glueshrinkorder:D
672	_expl_primitive:NN	\gluetomu	\etex_gluetomu:D
673	_expl_primitive:NN	\mutoglu	\etex_mutoglu:D
674	_expl_primitive:NN	\lastlinefit	\etex_lastlinefit:D
675	_expl_primitive:NN	\interlinepenalties	\etex_interlinepenalties:D
676	_expl_primitive:NN	\clubpenalties	\etex_clubpenalties:D
677	_expl_primitive:NN	\widowpenalties	\etex_widowpenalties:D
678	_expl_primitive:NN	\displaywidowpenalties	\etex_displaywidowpenalties:D
679	_expl_primitive:NN	\middle	\etex_middle:D
680	_expl_primitive:NN	\savinghyphcodes	\etex_savinghyphcodes:D
681	_expl_primitive:NN	\savingvdiscards	\etex_savingvdiscards:D
682	_expl_primitive:NN	\pagediscards	\etex_pagediscards:D
683	_expl_primitive:NN	\splitdiscards	\etex_splitdiscards:D
684	_expl_primitive:NN	\TeXeTstate	\etex_TeXeTstate:D
685	_expl_primitive:NN	\beginL	\etex_beginL:D
686	_expl_primitive:NN	\endL	\etex_endL:D
687	_expl_primitive:NN	\beginR	\etex_beginR:D
688	_expl_primitive:NN	\endR	\etex_endR:D
689	_expl_primitive:NN	\predisplaydirection	\etex_predisplaydirection:D
690	_expl_primitive:NN	\everyeof	\etex_everyeof:D
691	_expl_primitive:NN	\protected	\etex_protected: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. 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.

692	_expl_primitive:NN	\pdfcreationdate	\pdftex_pdfcreationdate:D
693	_expl_primitive:NN	\pdfcolorstack	\pdftex_pdfcolorstack:D
694	_expl_primitive:NN	\pdfcompresslevel	\pdftex_pdfcompresslevel:D
695	_expl_primitive:NN	\pdfdecimaldigits	\pdftex_pdfdecimaldigits:D
696	_expl_primitive:NN	\pdfhorigin	\pdftex_pdfhorigin:D
697	_expl_primitive:NN	\pdfinfo	\pdftex_pdfinfo:D
698	_expl_primitive:NN	\pdflastxform	\pdftex_pdflastxform:D
699	_expl_primitive:NN	\pdfliteral	\pdftex_pdfliteral:D
700	_expl_primitive:NN	\pdfminorversion	\pdftex_pdfminorversion:D
701	_expl_primitive:NN	\pdfobjcompresslevel	\pdftex_pdfobjcompresslevel:D

```

702 \__expl_primitive:NN \pdfoutput \pdftex_pdfoutput:D
703 \__expl_primitive:NN \pdfrefxform \pdftex_pdfrefxform:D
704 \__expl_primitive:NN \pdfrestore \pdftex_pdfrestore:D
705 \__expl_primitive:NN \pdfsave \pdftex_pdfsave:D
706 \__expl_primitive:NN \pdfsetmatrix \pdftex_pdfsetmatrix:D
707 \__expl_primitive:NN \pdfpkresolution \pdftex_pdfpkresolution:D
708 \__expl_primitive:NN \pdftexrevision \pdftex_pdftextrevision:D
709 \__expl_primitive:NN \pdfvorigin \pdftex_pdfvorigin:D
710 \__expl_primitive:NN \pdfxform \pdftex_pdfxform:D

```

While these are not.

```

711 \__expl_primitive:NN \pdfstrcmp \pdftex_strcmp:D

```

X_qTeX-specific primitives. Note that X_qTeX’s `\strcmp` is handled earlier and is “rolled up” into `\pdfstrcmp`.

```

712 \__expl_primitive:NN \XeTeXversion \xetex_XeTeXversion:D

```

Primitives from LuaTeX.

```

713 \__expl_primitive:NN \catcodetable \luatex_catcodetable:D
714 \__expl_primitive:NN \directlua \luatex_directlua:D
715 \__expl_primitive:NN \initcatcodetable \luatex_initcatcodetable:D
716 \__expl_primitive:NN \latelua \luatex_latelua:D
717 \__expl_primitive:NN \luatexversion \luatex_luatexversion:D
718 \__expl_primitive:NN \savecatcodetable \luatex_savecatcodetable:D

```

Slightly more awkward are the directional primitives in LuaTeX. These come from Omega *via* Aleph, but we do not support those engines and so it seems most sensible to treat them as LuaTeX primitives for prefix purposes.

```

719 \__expl_primitive:NN \bodydir \luatex_bodydir:D
720 \__expl_primitive:NN \mathdir \luatex_mathdir:D
721 \__expl_primitive:NN \pagedir \luatex_pagedir:D
722 \__expl_primitive:NN \pardir \luatex_pardir:D
723 \__expl_primitive:NN \textdir \luatex_textdir:D

```

The job is done: close the group (using the primitive renamed!).

```

724 \tex_endgroup:D

```

L^AT_EX 2_ε will have moved a few primitives, so these are sorted out.

```

725 <*package>
726 \tex_let:D \tex_end:D \@@end
727 \tex_let:D \tex_everydisplay:D \frozen@everydisplay
728 \tex_let:D \tex_everymath:D \frozen@everymath
729 \tex_let:D \tex_hyphen:D \@@hyph
730 \tex_let:D \tex_input:D \@@input
731 \tex_let:D \tex_italiccorrection:D \@@italiccorr
732 \tex_let:D \tex_underline:D \@@underline

```

That is also true for the luatex package for L^AT_EX 2_ε.

```

733 \tex_let:D \luatex_catcodetable:D \luatexcatcodetable
734 \tex_let:D \luatex_initcatcodetable:D \luatexinitcatcodetable
735 \tex_let:D \luatex_latelua:D \luatexlatelua
736 \tex_let:D \luatex_savecatcodetable:D \luatexsavecatcodetable

```

Which also covers those slightly odd ones.

```

737 \tex_let:D \luatex_bodydir:D      \luatexbodydir
738 \tex_let:D \luatex_mathdir:D      \luatexmathdir
739 \tex_let:D \luatex_pagedir:D      \luatexpagedir
740 \tex_let:D \luatex_pardir:D      \luatexpardir
741 \tex_let:D \luatex_textdir:D      \luatextextdir
742 </package>
743 </initex | package>

```

3 l3basics implementation

```

744 <*initex | package>
745 <*package>
746 \ProvidesExplPackage
747   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
748   \__expl_package_check:
749 </package>

```

3.1 Renaming some T_EX primitives (again)

Having given all the T_EX primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but do a few now, just to get started.²

```

\if_true: Then some conditionals.
\if_false: 750 \tex_let:D \if_true:      \tex_iftrue:D
\or:       751 \tex_let:D \if_false:    \tex_iffalse:D
\else:     752 \tex_let:D \or:      \tex_or:D
\fi:       753 \tex_let:D \else:     \tex_else:D
\reverse_if:N 754 \tex_let:D \fi:      \tex_fi:D
\if:w      755 \tex_let:D \reverse_if:N \etex_unless:D
\if_charcode:w 756 \tex_let:D \if:w      \tex_if:D
\if_catcode:w 757 \tex_let:D \if_charcode:w \tex_if:D
\if_meaning:w 758 \tex_let:D \if_catcode:w \tex_ifcat:D
              759 \tex_let:D \if_meaning:w \tex_ifx:D

```

(End definition for \if_true: and others. These functions are documented on page 24.)

```

\if_mode_math: TEX lets us detect some if its modes.
\if_mode_horizontal: 760 \tex_let:D \if_mode_math:    \tex_ifmmode:D
\if_mode_vertical:   761 \tex_let:D \if_mode_horizontal: \tex_ifhmode:D
\if_mode_inner:      762 \tex_let:D \if_mode_vertical:  \tex_ifvmode:D
                    763 \tex_let:D \if_mode_inner:    \tex_ifinner:D

```

(End definition for \if_mode_math: and others. These functions are documented on page 24.)

²This 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.

`\if_cs_exist:N` Building csnames and testing if control sequences exist.

```
\if_cs_exist:w 764 \tex_let:D \if_cs_exist:N \etex_ifdefined:D
\cs:w          765 \tex_let:D \if_cs_exist:w \etex_ifcsname:D
\cs_end:       766 \tex_let:D \cs:w \tex_csname:D
              767 \tex_let:D \cs_end: \tex_endcsname:D
```

(End definition for \if_cs_exist:N and others. These functions are documented on page 17.)

`\exp_after:wN` The three `\exp_` functions are used in the `l3expan` module where they are described.

```
\exp_not:N      768 \tex_let:D \exp_after:wN \tex_expandafter:D
\exp_not:n      769 \tex_let:D \exp_not:N \tex_noexpand:D
              770 \tex_let:D \exp_not:n \etex_unexpanded:D
```

(End definition for \exp_after:wN, \exp_not:N, and \exp_not:n. These functions are documented on page 33.)

`\token_to_meaning:N` Examining a control sequence or token.

```
\token_to_str:N 771 \tex_let:D \token_to_meaning:N \tex_meaning:D
\cs_meaning:N    772 \tex_let:D \token_to_str:N \tex_string:D
              773 \tex_let:D \cs_meaning:N \tex_meaning:D
```

(End definition for \token_to_meaning:N, \token_to_str:N, and \cs_meaning:N. These functions are documented on page 16.)

`\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: 774 \tex_let:D \scan_stop: \tex_relax:D
\group_end:    775 \tex_let:D \group_begin: \tex_begingroup:D
              776 \tex_let:D \group_end: \tex_endgroup:D
```

(End definition for \scan_stop:, \group_begin:, and \group_end:. These functions are documented on page 9.)

`\if_int_compare:w` For integers.

```
\__int_to_roman:w 777 \tex_let:D \if_int_compare:w \tex_ifnum:D
                  778 \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 74.)

`\group_insert_after:N` Adding material after the end of a group.

```
779 \tex_let:D \group_insert_after:N \tex_aftergroup:D
```

(End definition for \group_insert_after:N. This function is documented on page 9.)

`\exp_args:Nc` Discussed in `l3expan`, but needed much earlier.

```
\exp_args:cc 780 \tex_long:D \tex_def:D \exp_args:Nc #1#2
              781 { \exp_after:wN #1 \cs:w #2 \cs_end: }
              782 \tex_long:D \tex_def:D \exp_args:cc #1#2
              783 { \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 ??.)

`\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.

```

784 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
785 \tex_long:D \tex_def:D \cs_meaning:c #1
786 {
787   \if_cs_exist:w #1 \cs_end:
788     \exp_after:wN \use_i:nn
789   \else:
790     \exp_after:wN \use_ii:nn
791   \fi:
792   { \exp_args:Nc \cs_meaning:N {#1} }
793   { \tl_to_str:n {undefined} }
794 }
795 \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` 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.
`\c_zero`
`\c_sixteen` 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 as \TeX wants to reserve count registers 0–9, the first available one is 10 so we use that for `\c_minus_one`.
`\c_six`
`\c_seven`
`\c_twelve`

```

796 <*package>
797 \tex_let:D \c_minus_one \m@ne
798 </package>
799 <*initex>
800 \tex_countdef:D \c_minus_one = 10 ~
801 \c_minus_one = -1 ~
802 </initex>
803 \tex_chardef:D \c_sixteen = 16 ~
804 \tex_chardef:D \c_zero = 0 ~
805 \tex_chardef:D \c_six = 6 ~
806 \tex_chardef:D \c_seven = 7 ~
807 \tex_chardef:D \c_twelve = 12 ~

```

(End definition for `\c_minus_one`, `\c_zero`, and `\c_sixteen`. These functions are documented on page 73.)

`\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`.

```

808 \etex_ifdefined:D \luatex luatexversion:D
809 \tex_chardef:D \c_max_register_int = 65 535 ~
810 \tex_else:D

```

```

811 \tex_mathchardef:D \c_max_register_int = 32 767 ~
812 \tex_fi:D

```

(End definition for `\c_max_register_int`. This variable is documented on page 73.)

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.

```

813 \tex_let:D \cs_set_nopar:Npn \tex_def:D
814 \tex_let:D \cs_set_nopar:Npx \tex_edef:D
815 \etex_protected:D \cs_set_nopar:Npn \cs_set:Npn
816 { \tex_long:D \cs_set_nopar:Npn }
817 \etex_protected:D \cs_set_nopar:Npn \cs_set:Npx
818 { \tex_long:D \cs_set_nopar:Npx }
819 \etex_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npn
820 { \etex_protected:D \cs_set_nopar:Npn }
821 \etex_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npx
822 { \etex_protected:D \cs_set_nopar:Npx }
823 \cs_set_protected_nopar:Npn \cs_set_protected:Npn
824 { \etex_protected:D \tex_long:D \cs_set_nopar:Npn }
825 \cs_set_protected_nopar:Npn \cs_set_protected:Npx
826 { \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 ??.)

```

\cs_gset_nopar:Npn \cs_gset_nopar:Npx \cs_gset:Npn \cs_gset:Npx
\cs_gset_protected_nopar:Npn \cs_gset_protected_nopar:Npx
\cs_gset_protected:Npn \cs_gset_protected:Npx

```

Global versions of the above functions.

```

827 \tex_let:D \cs_gset_nopar:Npn \tex_gdef:D
828 \tex_let:D \cs_gset_nopar:Npx \tex_xdef:D
829 \cs_set_protected_nopar:Npn \cs_gset:Npn
830 { \tex_long:D \cs_gset_nopar:Npn }
831 \cs_set_protected_nopar:Npn \cs_gset:Npx
832 { \tex_long:D \cs_gset_nopar:Npx }
833 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npn
834 { \etex_protected:D \cs_gset_nopar:Npn }
835 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npx
836 { \etex_protected:D \cs_gset_nopar:Npx }
837 \cs_set_protected_nopar:Npn \cs_gset_protected:Npn
838 { \etex_protected:D \tex_long:D \cs_gset_nopar:Npn }
839 \cs_set_protected_nopar:Npn \cs_gset_protected:Npx
840 { \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 ??.)

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.

```

841 \cs_set_nopar:Npn \l__exp_internal_tl { }

```

(End definition for `\l__exp_internal_tl`. This variable is documented on page 34.)

`\use:c` This macro grabs its argument and returns a csname from it.

```
842 \cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }
```

(End definition for `\use:c`. This function is documented on page 17.)

`\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`.

```
843 \cs_set_protected:Npn \use:x #1
844 {
845   \cs_set_nopar:Npx \l__exp_internal_tl {#1}
846   \l__exp_internal_tl
847 }
```

(End definition for `\use:x`. This function is documented on page 20.)

`\use:n` These macros grab their arguments and returns them back to the input (with outer braces removed).

```
\use:nnn      848 \cs_set:Npn \use:n      #1      {#1}
\use:nnnn     849 \cs_set:Npn \use:nn     #1#2     {#1#2}
              850 \cs_set:Npn \use:nnn    #1#2#3    {#1#2#3}
              851 \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 ??.)

`\use_i:nn` The equivalent to L^AT_EX 2_ε's `\@firstoftwo` and `\@secondoftwo`.

```
\use_ii:nn    852 \cs_set:Npn \use_i:nn   #1#2 {#1}
              853 \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 19.)

`\use_i:nnn` We also need something for picking up arguments from a longer list.

```
\use_ii:nnn   854 \cs_set:Npn \use_i:nnn   #1#2#3 {#1}
\use_iii:nnn  855 \cs_set:Npn \use_ii:nnn  #1#2#3 {#2}
\use_i_ii:nnn 856 \cs_set:Npn \use_iii:nnn #1#2#3 {#3}
\use_i:nnnn   857 \cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}
\use_ii:nnnn  858 \cs_set:Npn \use_i:nnnn   #1#2#3#4 {#1}
\use_iii:nnnn 859 \cs_set:Npn \use_ii:nnnn  #1#2#3#4 {#2}
\use_iv:nnnn  860 \cs_set:Npn \use_iii:nnnn  #1#2#3#4 {#3}
              861 \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 19.)

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.

```
\use_none_delimit_by_q_stop:w 862 \cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
\use_none_delimit_by_q_recursion_stop:w 863 \cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
864 \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 47.)

`\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.

```
865 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
866 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
867 \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 47.)

3.5 Gobbling tokens from input

`\use_none:n` To gobble tokens from the input we use a standard naming convention: the number of
`\use_none:nn` tokens gobbled is given by the number of n's following the : in the name. Although we
`\use_none:nnn` could define functions to remove ten arguments or more using separate calls of `\use_-`
`\use_none:nnnn` `none:nnnnn`, this is very non-intuitive to the programmer who will assume that expanding
`\use_none:nnnnn` such a function once will take care of gobbling all the tokens in one go.
`\use_none:nnnnnn`

```
868 \cs_set:Npn \use_none:n #1 { }
869 \cs_set:Npn \use_none:nn #1#2 { }
870 \cs_set:Npn \use_none:nnn #1#2#3 { }
871 \cs_set:Npn \use_none:nnnn #1#2#3#4 { }
872 \cs_set:Npn \use_none:nnnnn #1#2#3#4#5 { }
873 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6 { }
874 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7 { }
875 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8 { }
876 \cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9 { }
```

(End definition for `\use_none:n` and others. These functions are documented on page ??.)

3.6 Conditional processing and definitions

Underneath any predicate function (`_p`) or other conditional forms (TF, etc.) is a built-in logic saying that it after all of the testing and processing must return the *state* this leaves TeX in. Therefore, a simple user interface could be something like

```
\if_meaning:w #1#2
  \prg_return_true:
\else:
  \if_meaning:w #1#3
    \prg_return_true:
  \else:
    \prg_return_false:
  \fi:
\fi:
```

Usually, a TeX programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the TeX programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

`\prg_return_true:` The idea here is that `__int_to_roman:w` will expand fully any `\else:` and the `\fi:` that are waiting to be discarded, before reaching the `\c_zero` 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.

```

877 \cs_set_nopar:Npn \prg_return_true:
878 { \exp_after:wN \use_i:nn \__int_to_roman:w }
879 \cs_set_nopar:Npn \prg_return_false:
880 { \exp_after:wN \use_ii:nn \__int_to_roman: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 37.)

`\prg_set_conditional:Npnn` The user functions for the types using parameter text from the programmer. The various functions only differ by which function is used for the assignment. For those `Npnn` type functions, we must grab the parameter text, reading everything up to a left brace before continuing. Then split the base function into name and signature, and feed `{<name>}` `{<signature>}` `<boolean>` `{<set or new>}` `{<maybe protected>}` `{<parameters>}` `{TF,...}` `{<code>}` to the auxiliary function responsible for defining all conditionals.

```

881 \cs_set_protected_nopar:Npn \prg_set_conditional:Npnn
882 { \__prg_generate_conditional_parm:nnNpnn { set } { } }
883 \cs_set_protected_nopar:Npn \prg_new_conditional:Npnn
884 { \__prg_generate_conditional_parm:nnNpnn { new } { } }
885 \cs_set_protected_nopar:Npn \prg_set_protected_conditional:Npnn
886 { \__prg_generate_conditional_parm:nnNpnn { set } { _protected } }
887 \cs_set_protected_nopar:Npn \prg_new_protected_conditional:Npnn
888 { \__prg_generate_conditional_parm:nnNpnn { new } { _protected } }
889 \cs_set_protected:Npn \__prg_generate_conditional_parm:nnNpnn #1#2#3#4#
890 {
891   \__cs_split_function:NN #3 \__prg_generate_conditional:nnNnnnnn
892   {#1} {#2} {#4}
893 }

```

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 35.)

`\prg_set_conditional:Nnn` The user functions for the types automatically inserting the correct parameter text based on the signature. The various functions only differ by which function is used for the assignment. Split the base function into name and signature. The second auxiliary generates the parameter text from the number of letters in the signature. Then feed `{<name>}` `{<signature>}` `<boolean>` `{<set or new>}` `{<maybe protected>}` `{<parameters>}` `{TF,...}` `{<code>}` to the auxiliary function responsible for defining all conditionals. If the `<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.

```

894 \cs_set_protected_nopar:Npn \prg_set_conditional:Nnn

```

```

895 { \prg_generate_conditional_count:nnNnn { set } { } }
896 \cs_set_protected_nopar:Npn \prg_new_conditional:Nnn
897 { \prg_generate_conditional_count:nnNnn { new } { } }
898 \cs_set_protected_nopar:Npn \prg_set_protected_conditional:Nnn
899 { \prg_generate_conditional_count:nnNnn { set } { _protected } }
900 \cs_set_protected_nopar:Npn \prg_new_protected_conditional:Nnn
901 { \prg_generate_conditional_count:nnNnn { new } { _protected } }
902 \cs_set_protected:Npn \prg_generate_conditional_count:nnNnn #1#2#3
903 {
904   \cs_split_function:NN #3 \prg_generate_conditional_count:nnNnnnn
905   {#1} {#2}
906 }
907 \cs_set_protected:Npn \prg_generate_conditional_count:nnNnnnn #1#2#3#4#5
908 {
909   \cs_parm_from_arg_count:nnF
910   { \prg_generate_conditional:nnNnnnnnn {#1} {#2} #3 {#4} {#5} }
911   { \tl_count:n {#2} }
912   {
913     \msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
914     { \token_to_str:c { #1 : #2 } }
915     { \tl_count:n {#2} }
916     \use_none:nn
917   }
918 }

```

(End definition for `\prg_set_conditional:Nnn` and others. These functions are documented on page ??.)

`\prg_generate_conditional:nnNnnnnn`
`\prg_generate_conditional:nnnnnnnw`

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 `\etex_detokenize:D` makes the later loop more robust.

```

919 \cs_set_protected:Npn \prg_generate_conditional:nnNnnnnnn #1#2#3#4#5#6#7#8
920 {
921   \if_meaning:w \c_false_bool #3
922   \msg_kernel_error:nnx { kernel } { missing-colon }
923   { \token_to_str:c {#1} }
924   \exp_after:wN \use_none:nn
925   \fi:
926   \use:x
927   {
928     \exp_not:N \prg_generate_conditional:nnnnnnnw
929     \exp_not:n { {#4} {#5} {#1} {#2} {#6} {#8} }
930     \etex_detokenize:D {#7}
931     \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
932   }

```

```
933 }
```

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.

```
934 \cs_set_protected:Npn \__prg_generate_conditional:nnnnnnw #1#2#3#4#5#6#7 ,
935 {
936   \if_meaning:w \q_recursion_tail #7
937   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
938   \fi:
939   \use:c { __prg_generate_ #7 _form:wnnnnnn }
940   \tl_if_empty:nF {#7}
941   {
942     \__msg_kernel_error:nnxx
943     { kernel } { conditional-form-unknown }
944     {#7} { \token_to_str:c { #3 : #4 } }
945   }
946   \use_none:nnnnnnn
947   \q_stop
948   {#1} {#2} {#3} {#4} {#5} {#6}
949   \__prg_generate_conditional:nnnnnnw {#1} {#2} {#3} {#4} {#5} {#6}
950 }
```

(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 `\c_zero`: 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.

```
951 \cs_set_protected:Npn \__prg_generate_p_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
952 {
953   \if_meaning:w \scan_stop: #3 \scan_stop:
954   \exp_after:wN \use_i:nn
955   \else:
956   \exp_after:wN \use_ii:nn
957   \fi:
958   {
959     \exp_args:cc { cs_ #2 #3 :Npn } { #4 _p: #5 } #6
960     { #7 \c_zero \c_true_bool \c_false_bool }
961   }
962   {
963     \__msg_kernel_error:nnx { kernel } { protected-predicate }
964     { \token_to_str:c { #4 _p: #5 } }
965   }
966 }
```

```

967 \cs_set_protected:Npn \__prg_generate_T_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
968 {
969   \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 T } #6
970   { #7 \c_zero \use:n \use_none:n }
971 }
972 \cs_set_protected:Npn \__prg_generate_F_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
973 {
974   \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 F } #6
975   { #7 \c_zero { } }
976 }
977 \cs_set_protected:Npn \__prg_generate_TF_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
978 {
979   \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 TF } #6
980   { #7 \c_zero }
981 }

```

(End definition for __prg_generate_p_form:wnnnnnn and others.)

\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn
__prg_set_eq_conditional:NNn

The setting-equal functions. Split the two functions and feed a first auxiliary $\{\langle name_1 \rangle\}$ $\{\langle signature_1 \rangle\}$ $\langle boolean_1 \rangle$ $\{\langle name_2 \rangle\}$ $\{\langle signature_2 \rangle\}$ $\langle boolean_2 \rangle$ $\langle copying\ function \rangle$ $\langle conditions \rangle$, $\backslash q_recursion_tail$, $\backslash q_recursion_stop$

```

982 \cs_set_protected_nopar:Npn \prg_set_eq_conditional:NNn
983 { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
984 \cs_set_protected_nopar:Npn \prg_new_eq_conditional:NNn
985 { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
986 \cs_set_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
987 {
988   \use:x
989   {
990     \exp_not:N \__prg_set_eq_conditional:nnNnnNNw
991     \__cs_split_function:NN #2 \prg_do_nothing:
992     \__cs_split_function:NN #3 \prg_do_nothing:
993     \exp_not:N #1
994     \etex_detokenize:D {#4}
995     \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
996   }
997 }

```

(End definition for \prg_set_eq_conditional:NNn and \prg_new_eq_conditional:NNn. These functions are documented on page 37.)

__prg_set_eq_conditional:nnNnnNNw
__prg_set_eq_conditional_loop:nnnnNw
__prg_set_eq_conditional_p_form:nnn
__prg_set_eq_conditional_TF_form:nnn
__prg_set_eq_conditional_T_form:nnn
__prg_set_eq_conditional_F_form:nnn

Split the function to be defined, and setup a manual clist loop over argument #6 of the first auxiliary. The second auxiliary receives twice three arguments coming from splitting the function to be defined and the function to copy. Make sure that both functions contained a colon, otherwise we don't know how to build conditionals, hence abort. Call the looping macro, with arguments $\{\langle name_1 \rangle\}$ $\{\langle signature_1 \rangle\}$ $\{\langle name_2 \rangle\}$ $\{\langle signature_2 \rangle\}$ $\langle copying\ function \rangle$ and followed by the comma list. At each step in the loop, make sure that the conditional form we copy is defined, and copy it, otherwise abort.

```

998 \cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
999 {
1000   \if_meaning:w \c_false_bool #3

```



```

1001     \_msg_kernel_error:nnx { kernel } { missing-colon }
1002     { \token_to_str:c {#1} }
1003     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1004     \fi:
1005     \if_meaning:w \c_false_bool #6
1006     \_msg_kernel_error:nnx { kernel } { missing-colon }
1007     { \token_to_str:c {#4} }
1008     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1009     \fi:
1010     \_prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}
1011   }
1012 \cs_set_protected:Npn \_prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
1013 {
1014   \if_meaning:w \q_recursion_tail #6
1015   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1016   \fi:
1017   \use:c { \_prg_set_eq_conditional_ #6 _form:wNnnnn }
1018   \tl_if_empty:nF {#6}
1019   {
1020     \_msg_kernel_error:nnxx
1021     { kernel } { conditional-form-unknown }
1022     {#6} { \token_to_str:c { #1 : #2 } }
1023   }
1024   \use_none:nnnnnn
1025   \q_stop
1026   #5 {#1} {#2} {#3} {#4}
1027   \_prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
1028 }
1029 \cs_set:Npn \_prg_set_eq_conditional_p_form:wNnnnn #1 \q_stop #2#3#4#5#6
1030 {
1031   \_chk_if_exist_cs:c { #5 _p : #6 }
1032   #2 { #3 _p : #4 } { #5 _p : #6 }
1033 }
1034 \cs_set:Npn \_prg_set_eq_conditional_TF_form:wNnnnn #1 \q_stop #2#3#4#5#6
1035 {
1036   \_chk_if_exist_cs:c { #5 : #6 TF }
1037   #2 { #3 : #4 TF } { #5 : #6 TF }
1038 }
1039 \cs_set:Npn \_prg_set_eq_conditional_T_form:wNnnnn #1 \q_stop #2#3#4#5#6
1040 {
1041   \_chk_if_exist_cs:c { #5 : #6 T }
1042   #2 { #3 : #4 T } { #5 : #6 T }
1043 }
1044 \cs_set:Npn \_prg_set_eq_conditional_F_form:wNnnnn #1 \q_stop #2#3#4#5#6
1045 {
1046   \_chk_if_exist_cs:c { #5 : #6 F }
1047   #2 { #3 : #4 F } { #5 : #6 F }
1048 }

```

(End definition for _prg_set_eq_conditional:nnNnnNw and _prg_set_eq_conditional_loop:nnnnNw.
These functions are documented on page 37.)

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
1049 \tex_chardef:D \c_true_bool = 1 ~
1050 \tex_chardef:D \c_false_bool = 0 ~
```

(End definition for `\c_true_bool` and `\c_false_bool`. These variables are documented on page 21.)

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 `__int_to_roman:w`. 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 `__int_to_roman:w` 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 `__int_to_roman:w` then sees 0, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the argument of `__int_to_roman:w`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```
1051 \cs_set_nopar:Npn \cs_to_str:N
1052 {
```

```

1053 \__int_to_roman:w
1054 \if:w \token_to_str:N \__cs_to_str:w \fi:
1055 \exp_after:wN \__cs_to_str:N \token_to_str:N
1056 }
1057 \cs_set:Npn \__cs_to_str:N #1 { \c_zero }
1058 \cs_set:Npn \__cs_to_str:w #1 \__cs_to_str:N
1059 { - \__int_value:w \fi: \exp_after:wN \c_zero }

```

(End definition for `\cs_to_str:N`. This function is documented on page 18.)

```

\__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 $\langle true \rangle$ or $\langle false \rangle$ is returned with $\langle true \rangle$ for when there is a colon in the function and $\langle false \rangle$ if there is not. 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 can't use a literal `:` because it has the wrong catcode here, so it's transformed from `@` with `\tex_lowercase:D`.

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 $\langle processor \rangle$. The second auxiliary trims the trailing `\q_mark` from the function name if present (that is, if the original function had no colon).

```

1060 \group_begin:
1061 \tex_lccode:D '\@ = '\: \scan_stop:
1062 \tex_catcode:D '\@ = 12 ~
1063 \tex_lowercase:D
1064 {
1065 \group_end:
1066 \cs_set:Npn \__cs_split_function:NN #1
1067 {
1068 \exp_after:wN \exp_after:wN
1069 \exp_after:wN \__cs_split_function_auxi:w
1070 \cs_to_str:N #1 \q_mark \c_true_bool
1071 @ \q_mark \c_false_bool
1072 \q_stop
1073 }
1074 \cs_set:Npn \__cs_split_function_auxi:w #1 @ #2 \q_mark #3#4 \q_stop #5
1075 { \__cs_split_function_auxii:w #5 #1 \q_mark \q_stop {#2} #3 }
1076 \cs_set:Npn \__cs_split_function_auxii:w #1#2 \q_mark #3 \q_stop
1077 { #1 {#2} }
1078 }

```

(End definition for `__cs_split_function:NN`. This function is documented on page 25.)

```

\__cs_get_function_name:N
\__cs_get_function_signature:N

```

Simple wrappers.

```

1079 \cs_set:Npn \__cs_get_function_name:N #1

```

```

1080 { \_cs_split_function:NN #1 \use_i:nnn }
1081 \cs_set:Npn \_cs_get_function_signature:N #1
1082 { \_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 then if it is in the hash table. There is no problem when inputting something like `\else:` or `\fi:` as `TeX` will only ever skip input in case the token tested against is `\scan_stop:`.

```

\cs_if_exist_p:c
\cs_if_exist:NTF
\cs_if_exist:cTF
1083 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1084 {
1085   \if_meaning:w #1 \scan_stop:
1086     \prg_return_false:
1087   \else:
1088     \if_cs_exist:N #1
1089       \prg_return_true:
1090     \else:
1091       \prg_return_false:
1092     \fi:
1093   \fi:
1094 }

```

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.

```

1095 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1096 {
1097   \if_cs_exist:w #1 \cs_end:
1098     \exp_after:wN \use_i:nn
1099   \else:
1100     \exp_after:wN \use_ii:nn
1101   \fi:
1102   {
1103     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1104     \prg_return_false:
1105   \else:
1106     \prg_return_true:
1107   \fi:
1108 }
1109 \prg_return_false:
1110 }

```

(End definition for `\cs_if_exist:N` and `\cs_if_exist:c`. These functions are documented on page ??.)

`\cs_if_free_p:N` The logical reversal of the above.

```

\cs_if_free_p:c 1111 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
\cs_if_free:NTF 1112 {
\cs_if_free:cTF 1113   \if_meaning:w #1 \scan_stop:
                  1114   \prg_return_true:
                  1115   \else:
                  1116     \if_cs_exist:N #1
                  1117     \prg_return_false:
                  1118     \else:
                  1119     \prg_return_true:
                  1120   \fi:
                  1121 \fi:
                  1122 }
                  1123 \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
                  1124 {
                  1125   \if_cs_exist:w #1 \cs_end:
                  1126   \exp_after:wN \use_i:nn
                  1127   \else:
                  1128   \exp_after:wN \use_ii:nn
                  1129   \fi:
                  1130   {
                  1131     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
                  1132     \prg_return_true:
                  1133     \else:
                  1134     \prg_return_false:
                  1135     \fi:
                  1136   }
                  1137   { \prg_return_true: }
                  1138 }

```

(End definition for `\cs_if_free:N` and `\cs_if_free:c`. These functions are documented on page ??.)

`\cs_if_exist_use:N` 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.

```

1139 \cs_set:Npn \cs_if_exist_use:NTF #1#2
1140 { \cs_if_exist:NTF #1 { #1 #2 } }
1141 \cs_set:Npn \cs_if_exist_use:NF #1
1142 { \cs_if_exist:NTF #1 { #1 } }
1143 \cs_set:Npn \cs_if_exist_use:NT #1 #2
1144 { \cs_if_exist:NTF #1 { #1 #2 } { } }
1145 \cs_set:Npn \cs_if_exist_use:N #1
1146 { \cs_if_exist:NTF #1 { #1 } { } }
1147 \cs_set:Npn \cs_if_exist_use:cTF #1#2
1148 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
1149 \cs_set:Npn \cs_if_exist_use:cF #1
1150 { \cs_if_exist:cTF {#1} { \use:c {#1} } }
1151 \cs_set:Npn \cs_if_exist_use:cT #1#2
1152 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }

```

```

1153 \cs_set:Npn \cs_if_exist_use:c #1
1154 { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }

```

(End definition for `\cs_if_exist_use:N` and `\cs_if_exist_use:c`. These functions are documented on page ??.)

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`.

```

1155 \cs_set_protected_nopar:Npn \iow_log:x
1156 { \tex_immediate:D \tex_write:D \c_minus_one }
1157 \cs_set_protected_nopar:Npn \iow_term:x
1158 { \tex_immediate:D \tex_write:D \c_sixteen }

```

(End definition for `\iow_log:x` and `\iow_term:x`. These functions are documented on page ??.)

`__msg_kernel_error:nxxx` If an internal error occurs before L^AT_EX3 has loaded `l3msg` then the code should issue a usable if terse error message and halt. This can only happen if a coding error is made by the team, so this is a reasonable response.

```

1159 \cs_set_protected:Npn \__msg_kernel_error:nxxx #1#2#3#4
1160 {
1161   \tex_errmessage:D
1162   {
1163     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
1164     Argh,~internal~LaTeX3~error! ^^J ^^J
1165     Module ~ #1 , ~ message-name~"#2": ^^J
1166     Arguments~'#3'~and~'#4' ^^J ^^J
1167     This~is~one~for~The~LaTeX3~Project:~bailing~out
1168   }
1169   \tex_end:D
1170 }
1171 \cs_set_protected:Npn \__msg_kernel_error:nxx #1#2#3
1172 { \__msg_kernel_error:nxxx {#1} {#2} {#3} { } }
1173 \cs_set_protected:Npn \__msg_kernel_error:nn #1#2
1174 { \__msg_kernel_error:nxxx {#1} {#2} { } { } }

```

(End definition for `__msg_kernel_error:nxxx`, `__msg_kernel_error:nxx`, and `__msg_kernel_error:nn`.)

`\msg_line_context:` Another one from `l3msg` which will be altered later.

```

1175 \cs_set_nopar:Npn \msg_line_context:
1176 { on~line~ \tex_the:D \tex_inputlineno:D }

```

(End definition for `\msg_line_context:`. This function is documented on page 142.)

`__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 `\scan_stop:.` Otherwise an error message is issued. We have to make sure we don't put the argument into the conditional processing since it may be an `\if...` type function!

```

1177 \cs_set_protected:Npn \__chk_if_free_cs:N #1
1178 {
1179   \cs_if_free:NF #1
1180   {
1181     \__msg_kernel_error:nxxx { kernel } { command-already-defined }
1182     { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1183   }
1184 }
1185 \*package
1186 \tex_ifodd:D \l@expl@log@functions@bool
1187 \cs_set_protected:Npn \__chk_if_free_cs:N #1
1188 {
1189   \cs_if_free:NF #1
1190   {
1191     \__msg_kernel_error:nxxx { kernel } { command-already-defined }
1192     { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1193   }
1194   \iow_log:x { Defining~\token_to_str:N #1~ \msg_line_context: }
1195 }
1196 \fi:
1197 \*package
1198 \cs_set_protected_nopar:Npn \__chk_if_free_cs:c
1199 { \exp_args:Nc \__chk_if_free_cs:N }
(End definition for \__chk_if_free_cs:N and \__chk_if_free_cs:c.)

```

`__chk_if_exist_cs:N` This function issues an error message when the control sequence in its argument does not exist.

```

1200 \cs_set_protected:Npn \__chk_if_exist_cs:N #1
1201 {
1202   \cs_if_exist:NF #1
1203   {
1204     \__msg_kernel_error:nxx { kernel } { command-not-defined }
1205     { \token_to_str:N #1 }
1206   }
1207 }
1208 \cs_set_protected_nopar:Npn \__chk_if_exist_cs:c
1209 { \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

```

```

1211 {
1212     \cs_set_protected:Npn #1 ##1
1213     {
1214         \__chk_if_free_cs:N ##1
1215         #2 ##1
1216     }
1217 }
1218 \__cs_tmp:w \cs_new_nopar:Npn          \cs_gset_nopar:Npn
1219 \__cs_tmp:w \cs_new_nopar:Npx          \cs_gset_nopar:Npx
1220 \__cs_tmp:w \cs_new:Npn                 \cs_gset:Npn
1221 \__cs_tmp:w \cs_new:Npx                 \cs_gset:Npx
1222 \__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1223 \__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1224 \__cs_tmp:w \cs_new_protected:Npn       \cs_gset_protected:Npn
1225 \__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 ??.)

\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.

```

1226 \cs_set:Npn \__cs_tmp:w #1#2
1227 { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
1228 \__cs_tmp:w \cs_set_nopar:cpn   \cs_set_nopar:Npn
1229 \__cs_tmp:w \cs_set_nopar:cpx   \cs_set_nopar:Npx
1230 \__cs_tmp:w \cs_gset_nopar:cpn  \cs_gset_nopar:Npn
1231 \__cs_tmp:w \cs_gset_nopar:cpx  \cs_gset_nopar:Npx
1232 \__cs_tmp:w \cs_new_nopar:cpn   \cs_new_nopar:Npn
1233 \__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. We may also do this globally.

```

1234 \__cs_tmp:w \cs_set:cpn   \cs_set:Npn
1235 \__cs_tmp:w \cs_set:cpx   \cs_set:Npx
1236 \__cs_tmp:w \cs_gset:cpn  \cs_gset:Npn
1237 \__cs_tmp:w \cs_gset:cpx  \cs_gset:Npx
1238 \__cs_tmp:w \cs_new:cpn   \cs_new:Npn
1239 \__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 the first arguments. We may also do this globally.

```

1240 \__cs_tmp:w \cs_set_protected_nopar:cpn   \cs_set_protected_nopar:Npn
1241 \__cs_tmp:w \cs_set_protected_nopar:cpx   \cs_set_protected_nopar:Npx

```



```
1242 \__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
1243 \__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1244 \__cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
1245 \__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	1246	__cs_tmp:w	\cs_set_protected:cpn	\cs_set_protected:Npn
\cs_gset_protected:cpx	1247	__cs_tmp:w	\cs_set_protected:cpx	\cs_set_protected:Npx
\cs_new_protected:cpn	1248	__cs_tmp:w	\cs_gset_protected:cpn	\cs_gset_protected:Npn
\cs_new_protected:cpx	1249	__cs_tmp:w	\cs_gset_protected:cpx	\cs_gset_protected:Npx
	1250	__cs_tmp:w	\cs_new_protected:cpn	\cs_new_protected:Npn
	1251	__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 sequence.
`\cs_set_eq:cN`

`\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 =.

<code>\cs_gset_eq:NN</code>	<code>\cs_set_eq:NN</code> is long to avoid problems with a literal argument of <code>\par</code> . While
<code>\cs_gset_eq:cN</code>	<code>\cs_new_eq:NN</code> will probably never be correct with a first argument of <code>\par</code> , define it
<code>\cs_gset_eq:Nc</code>	long in order to throw an “already defined” error rather than “runaway argument”.

```

\cs_gset_eq:cc 1252 \cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
\cs_new_eq:NN 1253 \cs_new_protected_nopar:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
\cs_new_eq:cN 1254 \cs_new_protected_nopar:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
\cs_new_eq:Nc 1255 \cs_new_protected_nopar:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
\cs_new_eq:cc 1256 \cs_new_protected_nopar:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }
1257 \cs_new_protected_nopar:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }
1258 \cs_new_protected_nopar:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
1259 \cs_new_protected_nopar:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }
1260 \cs_new_protected:Npn \cs_new_eq:NN #1
1261 {
1262   \__chk_if_free_cs:N #1
1263   \tex_global:D \cs_set_eq:NN #1
1264 }
1265 \cs_new_protected_nopar:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
1266 \cs_new_protected_nopar:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
1267 \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 ??.)

3.12 Undefining functions

`\cs_undefine:N` The following function is used to free the main memory from the definition of some function that isn't in use any longer. The `c` variant is careful not to add the control sequence to the hash table if it isn't there yet, and it also avoids nesting \TeX conditionals in case `#1` is unbalanced in this matter.

```

1268 \cs_new_protected:Npn \cs_undefine:N #1
1269 { \cs_gset_eq:NN #1 \tex_undefined:D }
1270 \cs_new_protected:Npn \cs_undefine:c #1
1271 {
1272   \if_cs_exist:w #1 \cs_end:
1273     \exp_after:wN \use:n
1274   \else:
1275     \exp_after:wN \use_none:n
1276   \fi:
1277   { \cs_gset_eq:cN {#1} \tex_undefined:D }
1278 }

```

(End definition for `\cs_undefine:N` and `\cs_undefine:c`. These functions are documented on page ??.)

3.13 Generating parameter text from argument count

`_cs_parm_from_arg_count:nnF` \LaTeX 3 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.

```

1279 \cs_set_protected:Npn \_cs_parm_from_arg_count:nnF #1#2
1280 {
1281   \exp_args:Nx \_cs_parm_from_arg_count_test:nnF
1282   {
1283     \exp_after:wN \exp_not:n
1284     \if_case:w \_int_eval:w #2 \_int_eval_end:
1285       { }
1286       \or: { ##1 }
1287       \or: { ##1##2 }
1288       \or: { ##1##2##3 }
1289       \or: { ##1##2##3##4 }
1290       \or: { ##1##2##3##4##5 }
1291       \or: { ##1##2##3##4##5##6 }
1292       \or: { ##1##2##3##4##5##6##7 }
1293       \or: { ##1##2##3##4##5##6##7##8 }
1294       \or: { ##1##2##3##4##5##6##7##8##9 }
1295       \else: { \c_false_bool }
1296     \fi:
1297   }

```

```

1298     {#1}
1299   }
1300   \cs_set_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
1301   {
1302     \if_meaning:w \c_false_bool #1
1303       \exp_after:wN \use_ii:nn
1304     \else:
1305       \exp_after:wN \use_i:nn
1306     \fi:
1307     { #2 {#1} }
1308   }

```

(End definition for `__cs_parm_from_arg_count:nnF`. This function is documented on page ??.)

3.14 Defining functions from a given number of arguments

```

\__cs_count_signature:N
\__cs_count_signature:c
\__cs_count_signature:nnN

```

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.

```

1309 \cs_new:Npn \__cs_count_signature:N #1
1310 { \int_eval:n { \__cs_split_function:NN #1 \__cs_count_signature:nnN } }
1311 \cs_new:Npn \__cs_count_signature:nnN #1#2#3
1312 {
1313   \if_meaning:w \c_true_bool #3
1314     \tl_count:n {#2}
1315   \else:
1316     \c_minus_one
1317   \fi:
1318 }
1319 \cs_new_nopar:Npn \__cs_count_signature:c
1320 { \exp_args:Nc \__cs_count_signature:N }

```

(End definition for `__cs_count_signature:N` and `__cs_count_signature:c`. These functions are documented on page ??.)

```

\cs_generate_from_arg_count:NNnn
\cs_generate_from_arg_count:cNnn
\cs_generate_from_arg_count:Ncnn

```

We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since TeX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```

1321 \cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
1322 {
1323   \__cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
1324   {
1325     \__msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
1326     { \token_to_str:N #1 } { \int_eval:n {#3} }
1327   }

```

```

1328         {#4}
1329     }

```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```

1330 \cs_new_protected_nopar:Npn \cs_generate_from_arg_count:cNnn
1331   { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
1332 \cs_new_protected_nopar:Npn \cs_generate_from_arg_count:Ncnn
1333   { \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 ??.)

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. We define some simpler functions with user interface `\cs_set:Nn \foo_bar:nn {#1,#2}`, *i.e.*, the number of arguments is read from the signature.

We want to define `\cs set:Nn` as

```

\cs_set:Nn \cs_set:Nn as
\cs_set:Nx
\cs_set_nopar:Nn \cs_set_protected:Npn \cs_set:Nn #1#2
\cs_set: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_gset_protected:Nx
\cs_gset_protected_nopar:Nn
\cs_gset_protected_nopar:Nx
\cs_new:Nn
\cs_new:Nx
\cs_new_nopar:Nn
\cs_new_nopar:Nx
\cs_new_protected:Nn
\cs_new_protected:Nx
\cs_new_protected_nopar:Nn
\cs_new_protected_nopar:Nx

```

We want to define `\cs_set:Nn` as

```

\cs_set:Npn \__cs_tmp:w #1#2#3
{
\cs_new_protected_nopar:cpx { cs_ #1 : #2 }
{
\exp_not:N \__cs_generate_from_signature:NNn
\exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
}
}
\cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
{
\__cs_split_function:NN #2 \__cs_generate_from_signature:nnNNNn
#1 #2
}
\cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
{
\bool_if:NTF #3
{
\cs_generate_from_arg_count:NNnn
#5 #4 { \tl_count:n {#2} } {#6}
}
}

```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```

1354     {
1355         \_msg_kernel_error:nxx { kernel } { missing-colon }
1356         { \token_to_str:N #5 }
1357     }
1358 }

```

Then we define the 24 variants beginning with N.

```

1359 \_cs_tmp:w { set } { Nn } { Npn }
1360 \_cs_tmp:w { set } { Nx } { Npx }
1361 \_cs_tmp:w { set_nopar } { Nn } { Npn }
1362 \_cs_tmp:w { set_nopar } { Nx } { Npx }
1363 \_cs_tmp:w { set_protected } { Nn } { Npn }
1364 \_cs_tmp:w { set_protected } { Nx } { Npx }
1365 \_cs_tmp:w { set_protected_nopar } { Nn } { Npn }
1366 \_cs_tmp:w { set_protected_nopar } { Nx } { Npx }
1367 \_cs_tmp:w { gset } { Nn } { Npn }
1368 \_cs_tmp:w { gset } { Nx } { Npx }
1369 \_cs_tmp:w { gset_nopar } { Nn } { Npn }
1370 \_cs_tmp:w { gset_nopar } { Nx } { Npx }
1371 \_cs_tmp:w { gset_protected } { Nn } { Npn }
1372 \_cs_tmp:w { gset_protected } { Nx } { Npx }
1373 \_cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
1374 \_cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
1375 \_cs_tmp:w { new } { Nn } { Npn }
1376 \_cs_tmp:w { new } { Nx } { Npx }
1377 \_cs_tmp:w { new_nopar } { Nn } { Npn }
1378 \_cs_tmp:w { new_nopar } { Nx } { Npx }
1379 \_cs_tmp:w { new_protected } { Nn } { Npn }
1380 \_cs_tmp:w { new_protected } { Nx } { Npx }
1381 \_cs_tmp:w { new_protected_nopar } { Nn } { Npn }
1382 \_cs_tmp:w { new_protected_nopar } { Nx } { Npx }

```

(End definition for \cs_set:Nn and others. These functions are documented on page ??.)

```

\cs_set:cn The 24 c variants simply use \exp_args:Nc.
\cs_set:cx
\cs_set_nopar:cn
\cs_set_nopar:cx
\cs_set_protected:cn
\cs_set_protected:cx
\cs_set_protected_nopar:cn
\cs_set_protected_nopar:cx
\cs_gset:cn
\cs_gset:cx
\cs_gset_nopar:cn
\cs_gset_nopar:cx
\cs_gset_protected:cn
\cs_gset_protected:cx
\cs_gset_protected_nopar:cn
\cs_gset_protected_nopar:cx
\cs_new:cn
\cs_new:cx
\cs_new_nopar:cn
\cs_new_nopar:cx
\cs_new_protected:cn
\cs_new_protected:cx
\cs_new_protected_nopar:cn
\cs_new_protected_nopar:cx

```

```

1399 \__cs_tmp:w { gset } { n }
1400 \__cs_tmp:w { gset } { x }
1401 \__cs_tmp:w { gset_nopar } { n }
1402 \__cs_tmp:w { gset_nopar } { x }
1403 \__cs_tmp:w { gset_protected } { n }
1404 \__cs_tmp:w { gset_protected } { x }
1405 \__cs_tmp:w { gset_protected_nopar } { n }
1406 \__cs_tmp:w { gset_protected_nopar } { x }
1407 \__cs_tmp:w { new } { n }
1408 \__cs_tmp:w { new } { x }
1409 \__cs_tmp:w { new_nopar } { n }
1410 \__cs_tmp:w { new_nopar } { x }
1411 \__cs_tmp:w { new_protected } { n }
1412 \__cs_tmp:w { new_protected } { x }
1413 \__cs_tmp:w { new_protected_nopar } { n }
1414 \__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`
`\cs_if_eq_p:cN`
`\cs_if_eq_p:Nc`
`\cs_if_eq_p:cc`
`\cs_if_eq:NNTF`
`\cs_if_eq:cNTF`
`\cs_if_eq:NcTF`
`\cs_if_eq:ccTF`

Check if two control sequences are identical.

```

1415 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
1416 {
1417     \if_meaning:w #1#2
1418     \prg_return_true: \else: \prg_return_false: \fi:
1419 }
1420 \cs_new_nopar:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
1421 \cs_new_nopar:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
1422 \cs_new_nopar:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }
1423 \cs_new_nopar:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
1424 \cs_new_nopar:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
1425 \cs_new_nopar:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
1426 \cs_new_nopar:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNT }
1427 \cs_new_nopar:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
1428 \cs_new_nopar:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
1429 \cs_new_nopar:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
1430 \cs_new_nopar:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
1431 \cs_new_nopar:Npn \cs_if_eq:ccF { \exp_args:Ncc \cs_if_eq:NNF }

```

(End definition for \cs_if_eq:NN and others. These functions are documented on page ??.)

3.17 Diagnostic functions

`__kernel_register_show:N`
`__kernel_register_show:c`

Check that the variable exists, then apply the \showthe primitive to the variable. The odd-looking \use:n gives a nicer output.

```

1432 \cs_new_protected:Npn \__kernel_register_show:N #1
1433 {
1434     \cs_if_exist:NTF #1
1435     { \tex_showthe:D \use:n {#1} }

```

```

1436     {
1437         \__msg_kernel_error:nmx { kernel } { variable-not-defined }
1438         { \token_to_str:N #1 }
1439     }
1440 }
1441 \cs_new_protected_nopar:Npn \__kernel_register_show:c
1442 { \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: a line-break is added after the first colon in the meaning (this is what TeX does for macros and five `\...mark` primitives). 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.

```

1443 \group_begin:
1444 \tex_lccode:D '?' = ': \scan_stop:
1445 \tex_catcode:D '?' = 12 \scan_stop:
1446 \tex_lowercase:D
1447 {
1448     \group_end:
1449     \cs_new_protected:Npn \cs_show:N #1
1450     {
1451         \__msg_show_variable:n
1452         {
1453             > ~ \token_to_str:N #1 =
1454             \exp_after:wN \__cs_show:www \cs_meaning:N #1
1455             \use_none:nn ? \prg_do_nothing:
1456         }
1457     }
1458     \cs_new:Npn \__cs_show:www #1 ? { #1 ? \ }
1459 }
1460 \cs_new_protected_nopar:Npn \cs_show:c
1461 { \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 ??.)

```

3.18 Engine specific definitions

`\xetex_if_engine_p:` In some cases it will be useful to know which engine we're running. This can all be hard-coded for speed.

```

\luatex_if_engine_p:
\pdfTeX_if_engine_p:
\xetex_if_engine:TF
\luatex_if_engine:TF
\pdfTeX_if_engine:TF
1462 \cs_new_eq:NN \luatex_if_engine:T \use_none:n
1463 \cs_new_eq:NN \luatex_if_engine:F \use:n
1464 \cs_new_eq:NN \luatex_if_engine:TF \use_i:nn
1465 \cs_new_eq:NN \pdfTeX_if_engine:T \use:n
1466 \cs_new_eq:NN \pdfTeX_if_engine:F \use_none:n
1467 \cs_new_eq:NN \pdfTeX_if_engine:TF \use_i:nn
1468 \cs_new_eq:NN \xetex_if_engine:T \use_none:n

```

```

1469 \cs_new_eq:NN \xetex_if_engine:F \use:n
1470 \cs_new_eq:NN \xetex_if_engine:TF \use_ii:nn
1471 \cs_new_eq:NN \luatex_if_engine_p: \c_false_bool
1472 \cs_new_eq:NN \pdftex_if_engine_p: \c_true_bool
1473 \cs_new_eq:NN \xetex_if_engine_p: \c_false_bool
1474 \cs_if_exist:NT \xetex_XeTeXversion:D
1475 {
1476   \cs_gset_eq:NN \pdftex_if_engine:T \use_none:n
1477   \cs_gset_eq:NN \pdftex_if_engine:F \use:n
1478   \cs_gset_eq:NN \pdftex_if_engine:TF \use_ii:nn
1479   \cs_gset_eq:NN \xetex_if_engine:T \use:n
1480   \cs_gset_eq:NN \xetex_if_engine:F \use_none:n
1481   \cs_gset_eq:NN \xetex_if_engine:TF \use_i:nn
1482   \cs_gset_eq:NN \pdftex_if_engine_p: \c_false_bool
1483   \cs_gset_eq:NN \xetex_if_engine_p: \c_true_bool
1484 }
1485 \cs_if_exist:NT \luatex_directlua:D
1486 {
1487   \cs_gset_eq:NN \luatex_if_engine:T \use:n
1488   \cs_gset_eq:NN \luatex_if_engine:F \use_none:n
1489   \cs_gset_eq:NN \luatex_if_engine:TF \use_i:nn
1490   \cs_gset_eq:NN \pdftex_if_engine:T \use_none:n
1491   \cs_gset_eq:NN \pdftex_if_engine:F \use:n
1492   \cs_gset_eq:NN \pdftex_if_engine:TF \use_ii:nn
1493   \cs_gset_eq:NN \luatex_if_engine_p: \c_true_bool
1494   \cs_gset_eq:NN \pdftex_if_engine_p: \c_false_bool
1495 }

```

(End definition for `\xetex_if_engine:`, `\luatex_if_engine:`, and `\pdftex_if_engine:`. These functions are documented on page 23.)

3.19 Doing nothing functions

`\prg_do_nothing:` This does not fit anywhere else!

```

1496 \cs_new_nopar:Npn \prg_do_nothing: { }

```

(End definition for `\prg_do_nothing:`. This function is documented on page 9.)

3.20 String comparisons

`\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. These should eventually move somewhere else.

`\str_if_eq_x_p:nn`
`\str_if_eq:nnTF`
`\str_if_eq_x:nnTF`

```

1497 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
1498 {
1499   \if_int_compare:w \pdftex_strcmp:D { \exp_not:n {#1} } { \exp_not:n {#2} }
1500     = \c_zero
1501     \prg_return_true: \else: \prg_return_false: \fi:
1502 }
1503 \prg_new_conditional:Npnn \str_if_eq_x:nn #1#2 { p , T , F , TF }

```



```

1504 {
1505   \if_int_compare:w \pdfutex_strcmp:D {#1} {#2} = \c_zero
1506   \prg_return_true: \else: \prg_return_false: \fi:
1507 }

```

(End definition for `\str_if_eq:nn` and `\str_if_eq_x:nn`. These functions are documented on page 22.)

`__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`, but hard-coded for speed.

```

1508 \cs_new:Npn \__str_if_eq_x_return:nn #1 #2
1509 {
1510   \if_int_compare:w \pdfutex_strcmp:D {#1} {#2} = \c_zero
1511   \prg_return_true:
1512   \else:
1513   \prg_return_false:
1514   \fi:
1515 }

```

(End definition for `__str_if_eq_x_return:nn`.)

`\str_case:nn` 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. `\str_case_x:nn` That is achieved by using the test input as the final case, as this will always be true. The `\str_case:nnTF` trick is then to tidy up the output such that the appropriate case code plus either the `\str_case_x:nnTF` true or false branch code is inserted. `__str_case:nnTF`

```

\__str_case_x:nnTF
\__prg_case_end:nw
\__str_case:nw
\__str_case_x:nw
\__str_case_end:nw
1516 \cs_new:Npn \str_case:nn #1#2
1517 {
1518   \tex_romannumeral:D
1519   \__str_case:nnTF {#1} {#2} { } { }
1520 }
1521 \cs_new:Npn \str_case:nnT #1#2#3
1522 {
1523   \tex_romannumeral:D
1524   \__str_case:nnTF {#1} {#2} {#3} { }
1525 }
1526 \cs_new:Npn \str_case:nnF #1#2
1527 {
1528   \tex_romannumeral:D
1529   \__str_case:nnTF {#1} {#2} { }
1530 }
1531 \cs_new:Npn \str_case:nnTF #1#2
1532 {
1533   \tex_romannumeral:D
1534   \__str_case:nnTF {#1} {#2}
1535 }
1536 \cs_new:Npn \__str_case:nnTF #1#2#3#4
1537 { \__str_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
1538 \cs_new:Npn \__str_case:nw #1#2#3
1539 {
1540   \str_if_eq:nnTF {#1} {#2}

```

```

1541     { \_str_case_end:nw {#3} }
1542     { \_str_case:nw {#1} }
1543   }
1544   \cs_new:Npn \str_case_x:nn #1#2
1545   {
1546     \tex_romannumeral:D
1547     \_str_case_x:nnTF {#1} {#2} { } { }
1548   }
1549   \cs_new:Npn \str_case_x:nnT #1#2#3
1550   {
1551     \tex_romannumeral:D
1552     \_str_case_x:nnTF {#1} {#2} {#3} { }
1553   }
1554   \cs_new:Npn \str_case_x:nnF #1#2
1555   {
1556     \tex_romannumeral:D
1557     \_str_case_x:nnTF {#1} {#2} { }
1558   }
1559   \cs_new:Npn \str_case_x:nnTF #1#2
1560   {
1561     \tex_romannumeral:D
1562     \_str_case_x:nnTF {#1} {#2}
1563   }
1564   \cs_new:Npn \_str_case_x:nnTF #1#2#3#4
1565   { \_str_case_x:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
1566   \cs_new:Npn \_str_case_x:nw #1#2#3
1567   {
1568     \str_if_eq_x:nnTF {#1} {#2}
1569     { \_str_case_end:nw {#3} }
1570     { \_str_case_x:nw {#1} }
1571   }

```

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).

```

1572   \cs_new:Npn \_prg_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
1573   { \c_zero #1 #4 }
1574   \cs_new_eq:NN \_str_case_end:nw \_prg_case_end:nw

```

(End definition for \str_case:nn and \str_case_x:nn. These functions are documented on page 25.)

3.21 Breaking out of mapping functions

_prg_break_point:Nn In inline mappings, the nesting level must be reset at the end of the mapping, even when
_prg_map_break:Nn the user decides to break out. This is done by putting the code that must be performed
as an argument of _prg_break_point:Nn. The breaking functions are then defined

to jump to that point and perform the argument of `__prg_break_point:Nn`, before the user’s code (if any). There is a check that we close the correct loop, otherwise we continue breaking.

```

1575 \cs_new_eq:NN \__prg_break_point:Nn \use_ii:nn
1576 \cs_new:Npn \__prg_map_break:Nn #1#2#3 \__prg_break_point:Nn #4#5
1577 {
1578   #5
1579   \if_meaning:w #1 #4
1580   \exp_after:wN \use_iii:nnn
1581   \fi:
1582   \__prg_map_break:Nn #1 {#2}
1583 }

```

(End definition for `__prg_break_point:Nn` and `__prg_map_break:Nn`. These functions are documented on page 43.)

`__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.

```

1584 \cs_new_eq:NN \__prg_break_point: \prg_do_nothing:
1585 \cs_new:Npn \__prg_break: #1 \__prg_break_point: { }
1586 \cs_new:Npn \__prg_break:n #1#2 \__prg_break_point: {#1}

```

(End definition for `__prg_break_point:..` This function is documented on page ??.)

3.22 Deprecated functions

`\str_case:nnn` Deprecated 2013-07-15.

```

\str_case_x:nnn
1587 \cs_new_eq:NN \str_case:nnn \str_case:nnF
1588 \cs_new_eq:NN \str_case_x:nnn \str_case_x:nnF

```

(End definition for `\str_case:nnn` and `\str_case_x:nnn`. These functions are documented on page ??.)

```

1589 </initex | package>

```

4 l3expan implementation

```

1590 <*initex | package>

```

```

1591 <@@=exp>

```

We start by ensuring that the required packages are loaded.

```

1592 <*package>
1593 \ProvidesExplPackage
1594   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
1595 \__expl_package_check:
1596 </package>

```

`\exp_after:wN` These are defined in `l3basics`.

`\exp_not:N` (End definition for `\exp_after:wN`. This function is documented on page 33.)

`\exp_not:n`

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 34.)

This code uses internal functions with names that start with `\::` to perform the expansions. All macros are **long** as this turned out to be desirable since the tokens undergoing expansion may be arbitrary user input.

An argument manipulator `\::⟨Z⟩` always has signature `#1\:::#2#3` where `#1` holds the remaining argument manipulations to be performed, `\:::` serves as an end marker for the list of manipulations, `#2` is the carried over result of the previous expansion steps and `#3` is the argument about to be processed. One exception to this rule is `\::p`, which has to grab an argument delimited by a left brace.

`__exp_arg_next:nnn` `#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.

```
1597 \cs_new:Npn \__exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
```

```
1598 \cs_new:Npn \__exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }
```

(End definition for `__exp_arg_next:nnn`. This function is documented on page 34.)

`\:::` The end marker is just another name for the identity function.

```
1599 \cs_new:Npn \::: #1 {#1}
```

(End definition for `\:::`.)

`\::n` This function is used to skip an argument that doesn’t need to be expanded.

```
1600 \cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(End definition for `\::n`.)

`\::N` This function is used to skip an argument that consists of a single token and doesn’t need to be expanded.

```
1601 \cs_new:Npn \::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(End definition for `\::N`.)

`\::p` This function is used to skip an argument that is delimited by a left brace and doesn’t need to be expanded. It should not be wrapped in braces in the result.

```
1602 \cs_new:Npn \::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(End definition for \::p.)

\::c This function is used to skip an argument that is turned into a control sequence without expansion.

```
1603 \cs_new:Npn \::c #1 \::: #2#3
1604 { \exp_after:wN \__exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(End definition for \::c.)

\::o This function is used to expand an argument once.

```
1605 \cs_new:Npn \::o #1 \::: #2#3
1606 { \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.
\exp_stop_f: The underlying `\romannumeral -'0` expands everything in its way to find something terminating the number and thereby expands the function in front of 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 case the scanner hits a number, this number also terminates the scanning and is left untouched. In the example shown earlier the scanning was stopped once $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ 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 `\romannumeral -'0` is $\langle null \rangle$, we wind up with a fully expanded list, only $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```
1607 \cs_new:Npn \::f #1 \::: #2#3
1608 {
1609   \exp_after:wN \__exp_arg_next:nnn
1610   \exp_after:wN { \tex_romannumeral:D -'0 #3 }
1611   {#1} {#2}
1612 }
1613 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(End definition for \::f. This function is documented on page 33.)

\::x This function is used to expand an argument fully.

```
1614 \cs_new_protected:Npn \::x #1 \::: #2#3
1615 {
1616   \cs_set_nopar:Npx \l__exp_internal_tl { {#3} }
1617   \exp_after:wN \__exp_arg_next:nnn \l__exp_internal_tl {#1} {#2}
1618 }
```

(End definition for \::x.)

\::v These functions return the value of a register, i.e., one of `tl`, `clist`, `int`, `skip`, `dim`
\::V and `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 primitive `\romannumeral` sets off an expansion similar to an `f` type expansion, which we will terminate using `\c_zero`. The argument is returned in braces.

```

1619 \cs_new:Npn \::V #1 \::: #2#3
1620 {
1621   \exp_after:wN \__exp_arg_next:nnn
1622   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #3 }
1623   {#1} {#2}
1624 }
1625 \cs_new:Npn \::v # 1\::: #2#3
1626 {
1627   \exp_after:wN \__exp_arg_next:nnn
1628   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:c {#3} }
1629   {#1} {#2}
1630 }

```

(End definition for \::v. This function is documented on page 34.)

`__exp_eval_register:N`
`__exp_eval_register:c`
`__exp_eval_error_msg:w`

This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in TeX register such as `\count`. For the TeX registers we have to utilize a `\the` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\the` and when not to. What we do here is try to find out whether the token will expand to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the register in question when it has been prefixed with `\exp_not:N` and the register itself. If it is a macro, the prefixed `\exp_not:N` will temporarily turn it into the primitive `\scan_stop:`.

```

1631 \cs_new:Npn \__exp_eval_register:N #1
1632 {
1633   \exp_after:wN \if_meaning:w \exp_not:N #1 #1

```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\scan_stop:`. In that case we throw an error. We could let TeX do it for us but that would result in the rather obscure

! You can't use '\relax' after \the.

which while quite true doesn't give many hints as to what actually went wrong. We provide something more sensible.

```

1634   \if_meaning:w \scan_stop: #1
1635   \__exp_eval_error_msg:w
1636   \fi:

```

The next bit requires some explanation. The function must be initiated by the primitive `\romannumeral` and we want to terminate this expansion chain by inserting the `\c_zero` integer constant. However, we have to expand the register `#1` before we do that. If it is a TeX register, we need to execute the sequence `\exp_after:wN \c_zero \tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN \c_zero #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```

1637   \else:
1638     \exp_after:wN \use_i_ii:nnn
1639   \fi:
1640   \exp_after:wN \c_zero \tex_the:D #1
1641 }

```

```

1642 \cs_new:Npn \__exp_eval_register:c #1
1643 { \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}

1644 \cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
1645 {
1646   \fi:
1647   \fi:
1648   \_msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
1649   \c_zero
1650 }

```

(End definition for `__exp_eval_register:N` and `__exp_eval_register:c`. These functions are documented on page ??.)

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!

```

1651 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
1652 \cs_new:Npn \exp_args:NNo #1#2#3
1653 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
1654 \cs_new:Npn \exp_args:NNNo #1#2#3#4
1655 { \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 31.)

`\exp_args:Nc` In l3basics.

`\exp_args:cc` (End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page ??.)

`\exp_args:NNc` Here are the functions that turn their argument into csnames but are expandable.

```

1656 \cs_new:Npn \exp_args:NNc #1#2#3
1657 { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
1658 \cs_new:Npn \exp_args:Ncc #1#2#3
1659 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
1660 \cs_new:Npn \exp_args:Nccc #1#2#3#4
1661 {
1662   \exp_after:wN #1
1663   \cs:w #2 \exp_after:wN \cs_end:
1664   \cs:w #3 \exp_after:wN \cs_end:
1665   \cs:w #4 \cs_end:
1666 }

```

(End definition for `\exp_args:Nnc`, `\exp_args:Ncc`, and `\exp_args:Nccc`. These functions are documented on page ??.)

```

\exp_args:Nf
\exp_args:Nv
\exp_args:Nv
1667 \cs_new:Npn \exp_args:Nf #1#2
1668 { \exp_after:wN #1 \exp_after:wN { \tex_romannumeral:D -'0 #2 } }
1669 \cs_new:Npn \exp_args:Nv #1#2
1670 {
1671   \exp_after:wN #1 \exp_after:wN
1672   { \tex_romannumeral:D \__exp_eval_register:c {#2} }
1673 }
1674 \cs_new:Npn \exp_args:Nv #1#2
1675 {
1676   \exp_after:wN #1 \exp_after:wN
1677   { \tex_romannumeral:D \__exp_eval_register:N #2 }
1678 }

```

(End definition for `\exp_args:Nf`, `\exp_args:Nv`, and `\exp_args:Nv`. These functions are documented on page 30.)

`\exp_args:NNV` Some more hand-tuned function with three arguments. If we forced that an `o` argument always has braces, we could implement `\exp_args:Nco` with less tokens and only two arguments.

```

\exp_args:NNV
\exp_args:NNv
\exp_args:NNf
\exp_args:NNV
\exp_args:Ncf
\exp_args:Nco
1679 \cs_new:Npn \exp_args:NNf #1#2#3
1680 {
1681   \exp_after:wN #1
1682   \exp_after:wN #2
1683   \exp_after:wN { \tex_romannumeral:D -'0 #3 }
1684 }
1685 \cs_new:Npn \exp_args:NNv #1#2#3
1686 {
1687   \exp_after:wN #1
1688   \exp_after:wN #2
1689   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:c {#3} }
1690 }
1691 \cs_new:Npn \exp_args:NNV #1#2#3
1692 {
1693   \exp_after:wN #1
1694   \exp_after:wN #2
1695   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #3 }
1696 }
1697 \cs_new:Npn \exp_args:Nco #1#2#3
1698 {
1699   \exp_after:wN #1
1700   \cs:w #2 \exp_after:wN \cs_end:
1701   \exp_after:wN {#3}
1702 }
1703 \cs_new:Npn \exp_args:Ncf #1#2#3
1704 {
1705   \exp_after:wN #1

```



```

1706     \cs:w #2 \exp_after:wN \cs_end:
1707     \exp_after:wN { \tex_romannumeral:D -'0 #3 }
1708   }
1709 \cs_new:Npn \exp_args:NVV #1#2#3
1710 {
1711   \exp_after:wN #1
1712   \exp_after:wN { \tex_romannumeral:D \exp_after:wN
1713     \__exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
1714   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #3 }
1715 }

```

(End definition for `\exp_args:NNV` and others. These functions are documented on page ??.)

`\exp_args:Ncco` A few more that we can hand-tune.

```

\exp_args:NcNc 1716 \cs_new:Npn \exp_args:NNNV #1#2#3#4
\exp_args:NcNo 1717 {
\exp_args:NNNV 1718   \exp_after:wN #1
1719   \exp_after:wN #2
1720   \exp_after:wN #3
1721   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #4 }
1722 }
1723 \cs_new:Npn \exp_args:NcNc #1#2#3#4
1724 {
1725   \exp_after:wN #1
1726   \cs:w #2 \exp_after:wN \cs_end:
1727   \exp_after:wN #3
1728   \cs:w #4 \cs_end:
1729 }
1730 \cs_new:Npn \exp_args:NcNo #1#2#3#4
1731 {
1732   \exp_after:wN #1
1733   \cs:w #2 \exp_after:wN \cs_end:
1734   \exp_after:wN #3
1735   \exp_after:wN {#4}
1736 }
1737 \cs_new:Npn \exp_args:Ncco #1#2#3#4
1738 {
1739   \exp_after:wN #1
1740   \cs:w #2 \exp_after:wN \cs_end:
1741   \cs:w #3 \exp_after:wN \cs_end:
1742   \exp_after:wN {#4}
1743 }

```

(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

```
1744 \cs_new_protected_nopar:Npn \exp_args:Nx { \::x \::: }
```

(End definition for \exp_args:Nx. This function is documented on page 30.)

\exp_args:Nnc Here are the actual function definitions, using the helper functions above.

```
\exp_args:Nfo 1745 \cs_new_nopar:Npn \exp_args:Nnc { \::n \::c \::: }
```

```
\exp_args:Nff 1746 \cs_new_nopar:Npn \exp_args:Nfo { \::f \::o \::: }
```

```
\exp_args:Nnf 1747 \cs_new_nopar:Npn \exp_args:Nff { \::f \::f \::: }
```

```
\exp_args:Nno 1748 \cs_new_nopar:Npn \exp_args:Nnf { \::n \::f \::: }
```

```
\exp_args:NnV 1749 \cs_new_nopar:Npn \exp_args:Nno { \::n \::o \::: }
```

```
\exp_args:Noo 1750 \cs_new_nopar:Npn \exp_args:NnV { \::n \::V \::: }
```

```
\exp_args:Nof 1751 \cs_new_nopar:Npn \exp_args:Noo { \::o \::o \::: }
```

```
\exp_args:Noc 1752 \cs_new_nopar:Npn \exp_args:Nof { \::o \::f \::: }
```

```
\exp_args:NNx 1753 \cs_new_nopar:Npn \exp_args:Noc { \::o \::c \::: }
```

```
\exp_args:Ncx 1754 \cs_new_protected_nopar:Npn \exp_args:NNx { \::N \::x \::: }
```

```
\exp_args:Nnx 1755 \cs_new_protected_nopar:Npn \exp_args:Ncx { \::c \::x \::: }
```

```
\exp_args:Nox 1756 \cs_new_protected_nopar:Npn \exp_args:Nnx { \::n \::x \::: }
```

```
\exp_args:Nxo 1757 \cs_new_protected_nopar:Npn \exp_args:Nox { \::o \::x \::: }
```

```
\exp_args:Nxx 1758 \cs_new_protected_nopar:Npn \exp_args:Nxo { \::x \::o \::: }
```

```
1759 \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 1760 \cs_new_nopar:Npn \exp_args:NNno { \::N \::n \::o \::: }
```

```
\exp_args:NNnc 1761 \cs_new_nopar:Npn \exp_args:NNoo { \::N \::o \::o \::: }
```

```
\exp_args:NNno 1762 \cs_new_nopar:Npn \exp_args:NNnc { \::n \::n \::c \::: }
```

```
\exp_args:Nooo 1763 \cs_new_nopar:Npn \exp_args:NNno { \::n \::n \::o \::: }
```

```
\exp_args:NNnx 1764 \cs_new_nopar:Npn \exp_args:Nooo { \::o \::o \::o \::: }
```

```
\exp_args:NNox 1765 \cs_new_protected_nopar:Npn \exp_args:NNnx { \::N \::n \::x \::: }
```

```
\exp_args:NNnx 1766 \cs_new_protected_nopar:Npn \exp_args:NNox { \::N \::o \::x \::: }
```

```
\exp_args:Nnox 1767 \cs_new_protected_nopar:Npn \exp_args:NNnx { \::n \::n \::x \::: }
```

```
1768 \cs_new_protected_nopar:Npn \exp_args:Nnox { \::n \::o \::x \::: }
```

```
1769 \cs_new_protected_nopar:Npn \exp_args:Nccx { \::c \::c \::x \::: }
```

```
\exp_args:Ncnx 1770 \cs_new_protected_nopar:Npn \exp_args:Ncnx { \::c \::n \::x \::: }
```

```
\exp_args:Noox 1771 \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

```
1772 \cs_new:Npn \_exp_arg_last_unbraced:nn #1#2 { #2#1 }
```

```
1773 \cs_new:Npn \::f_unbraced \::: #1#2
```

```
1774 {
```

```
1775 \exp_after:wN \_exp_arg_last_unbraced:nn
```

```
1776 \exp_after:wN { \tex_romannumeral:D -'0 #2 } {#1}
```

```
1777 }
```

```
1778 \cs_new:Npn \::o_unbraced \::: #1#2
```

```

1779 { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
1780 \cs_new:Npn \::V_unbraced \::: #1#2
1781 {
1782   \exp_after:wN \__exp_arg_last_unbraced:nn
1783   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #2 } {#1}
1784 }
1785 \cs_new:Npn \::v_unbraced \::: #1#2
1786 {
1787   \exp_after:wN \__exp_arg_last_unbraced:nn
1788   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:c {#2} } {#1}
1789 }
1790 \cs_new_protected:Npn \::x_unbraced \::: #1#2
1791 {
1792   \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
1793   \l__exp_internal_tl
1794 }

```

(End definition for __exp_arg_last_unbraced:nn. This function is documented on page ??.)

\exp_last_unbraced:NV
 \exp_last_unbraced:Nv
 \exp_last_unbraced:Nf
 \exp_last_unbraced:No
 \exp_last_unbraced:Nco
 \exp_last_unbraced:NcV
 \exp_last_unbraced:NNV
 \exp_last_unbraced:NNo
 \exp_last_unbraced:NNNV
 \exp_last_unbraced:NNNo
 \exp_last_unbraced:Nno
 \exp_last_unbraced:Noo
 \exp_last_unbraced:Nfo
 \exp_last_unbraced:NnNo
 \exp_last_unbraced:Nx

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```

1795 \cs_new:Npn \exp_last_unbraced:NV #1#2
1796 { \exp_after:wN #1 \tex_romannumeral:D \__exp_eval_register:N #2 }
1797 \cs_new:Npn \exp_last_unbraced:Nv #1#2
1798 { \exp_after:wN #1 \tex_romannumeral:D \__exp_eval_register:c {#2} }
1799 \cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
1800 \cs_new:Npn \exp_last_unbraced:Nf #1#2
1801 { \exp_after:wN #1 \tex_romannumeral:D -'0 #2 }
1802 \cs_new:Npn \exp_last_unbraced:Nco #1#2#3
1803 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
1804 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
1805 {
1806   \exp_after:wN #1
1807   \cs:w #2 \exp_after:wN \cs_end:
1808   \tex_romannumeral:D \__exp_eval_register:N #3
1809 }
1810 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3
1811 {
1812   \exp_after:wN #1
1813   \exp_after:wN #2
1814   \tex_romannumeral:D \__exp_eval_register:N #3
1815 }
1816 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
1817 { \exp_after:wN #1 \exp_after:wN #2 #3 }
1818 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
1819 {
1820   \exp_after:wN #1
1821   \exp_after:wN #2
1822   \exp_after:wN #3
1823   \tex_romannumeral:D \__exp_eval_register:N #4

```

```

1824 }
1825 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
1826 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
1827 \cs_new_nopar:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \:: }
1828 \cs_new_nopar:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \:: }
1829 \cs_new_nopar:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \:: }
1830 \cs_new_nopar:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \:: }
1831 \cs_new_protected_nopar:Npn \exp_last_unbraced:Nx { \::x_unbraced \:: }

```

(End definition for `\exp_last_unbraced:NV`. This function is documented on page 32.)

```

\exp_last_two_unbraced:Noo
\__exp_last_two_unbraced:noN

```

If #2 is a single token then this can be implemented as

```

\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
{ \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }

```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```

1832 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
1833 { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
1834 \cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
1835 { \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
\exp_not:f
\exp_not:V
\exp_not:v
1836 \cs_new:Npn \exp_not:o #1 { \etex_unexpanded:D \exp_after:wN {#1} }
1837 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
1838 \cs_new:Npn \exp_not:f #1
1839 { \etex_unexpanded:D \exp_after:wN { \tex_romannumeral:D -'0 #1 } }
1840 \cs_new:Npn \exp_not:V #1
1841 {
1842   \etex_unexpanded:D \exp_after:wN
1843   { \tex_romannumeral:D \__exp_eval_register:N #1 }
1844 }
1845 \cs_new:Npn \exp_not:v #1
1846 {
1847   \etex_unexpanded:D \exp_after:wN
1848   { \tex_romannumeral:D \__exp_eval_register:c {#1} }
1849 }

```

(End definition for `\exp_not:o`. This function is documented on page 33.)

4.6 Defining function variants

```

1850 \<@@=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.

```

1851 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
1852 {
1853   \__chk_if_exist_cs:N #1
1854   \__cs_generate_variant:N #1
1855   \exp_after:wN \__cs_split_function:NN
1856   \exp_after:wN #1
1857   \exp_after:wN \__cs_generate_variant:nnNN
1858   \exp_after:wN #1
1859   \etex_detokenize:D {#2} , \scan_stop: , \q_recursion_stop
1860 }
```

(End definition for `\cs_generate_variant:Nn`. This function is documented on page 28.)

```

\__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`.

```

1861 \group_begin:
1862   \tex_catcode:D '\M = 12 \scan_stop:
1863   \tex_catcode:D '\A = 12 \scan_stop:
1864   \tex_catcode:D '\P = 12 \scan_stop:
1865   \tex_catcode:D '\R = 12 \scan_stop:
1866   \tex_lowercase:D
1867   {
1868     \group_end:
1869     \cs_new_protected:Npn \__cs_generate_variant:N #1
1870     {
1871       \exp_after:wN \if_meaning:w \exp_not:N #1 #1
1872       \cs_set_eq:NN \__cs_tmp:w \cs_new_protected_nopar:Npx
1873       \else:
1874       \exp_after:wN \__cs_generate_variant:ww
```

```

1875         \token_to_meaning:N #1 MA \q_mark
1876         \q_mark \cs_new_protected_nopar:Npx
1877         PR
1878         \q_mark \cs_new_nopar:Npx
1879         \q_stop
1880     \fi:
1881 }
1882 \cs_new_protected:Npn \__cs_generate_variant:ww #1 MA #2 \q_mark
1883 { \__cs_generate_variant:wwNw #1 }
1884 \cs_new_protected:Npn \__cs_generate_variant:wwNw
1885     #1 PR #2 \q_mark #3 #4 \q_stop
1886 {
1887     \cs_set_eq:NN \__cs_tmp:w #3
1888 }
1889 }

```

(End definition for `__cs_generate_variant:N`. This function is documented on page 28.)

`__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.

```

1890 \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
1891 {
1892     \if_meaning:w \c_false_bool #3
1893     \__msg_kernel_error:nnx { kernel } { missing-colon }
1894     { \token_to_str:c {#1} }
1895     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1896     \fi:
1897     \__cs_generate_variant:Nnnw #4 {#1}{#2}
1898 }

```

(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`.

```

1899 \cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
1900 {
1901   \if_meaning:w \scan_stop: #4
1902     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1903   \fi:
1904   \use:x
1905   {
1906     \exp_not:N \__cs_generate_variant:wwNN
1907     \__cs_generate_variant_loop:nNwN { }
1908     #4
1909     \__cs_generate_variant_loop_end:nwwwNNnn
1910     \q_mark
1911     #3 ~
1912     { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
1913     { }
1914     \q_stop
1915     \exp_not:N #1 {#2} {#4}
1916   }
1917   \__cs_generate_variant:Nnnw #1 {#2} {#3}
1918 }
(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>		
<code>__cs_generate_variant_loop_end:nwwwNNnn</code>	#2 :	Next variant letter.
<code>__cs_generate_variant_loop_long:wNNnn</code>	#3 :	Remainder of variant form.
<code>__cs_generate_variant_loop_invalid:NNwNNnn</code>		

#4 : Next base letter.

The first argument is populated by `__cs_generate_variant_loop_same:w` when a variant letter and a base letter match. It is flushed into the input stream whenever the two letters are different: if the loop ends before, the argument is dropped, which means that trailing common letters are ignored.

The case where the two letters are different is only allowed with a base letter of `N` or `n`. Otherwise, call `__cs_generate_variant_loop_invalid:NNwNNnn` to remove the end of the loop, get arguments at the end of the loop, and place an appropriate error message as a second argument of `__cs_generate_variant:wwNN`. If the letters are distinct and the base letter is indeed `n` or `N`, leave in the input stream whatever argument was collected, and the next variant letter #2, then loop by calling `__cs_generate_variant_loop:nNwN`.

The loop can stop in three ways.

- If the end of the variant form is encountered first, #2 is `__cs_generate_variant_loop_end:nwwwNNnn` (expanded by the conditional `\if:w`), which inserts some tokens to end the conditional; grabs the *base name* as #7, the *variant signature* #8, the *next base letter* #1 and the part #3 of the base signature that wasn't read yet; and combines those into the *new function* to be defined.
- If the end of the base form is encountered first, #4 is `~{}~\fi:` which ends the conditional (with an empty expansion), followed by `__cs_generate_variant_loop_long:wNNnn`, which places an error as the second argument of `__cs_generate_variant:wwNN`.
- The loop can be interrupted early if the requested expansion is unavailable, namely when the variant and base letters differ and the base is neither `n` nor `N`. Again, an error is placed as the second argument of `__cs_generate_variant:wwNN`.

Note that if the variant form has the same length as the base form, #2 is as described in the first point, and #4 as described in the second point above. The `__cs_generate_variant_loop_end:nwwwNNnn` breaking function takes the empty brace group in #4 as its first argument: this empty brace group produces the correct signature for the full variant.

```
1919 \cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \q_mark #4
1920 {
1921   \if:w #2 #4
1922     \exp_after:wN \__cs_generate_variant_loop_same:w
1923   \else:
1924     \if:w N #4 \else:
1925       \if:w n #4 \else:
1926         \__cs_generate_variant_loop_invalid:NNwNNnn #4#2
1927       \fi:
1928     \fi:
1929   \fi:
1930   #1
1931   \prg_do_nothing:
1932   #2
```



```

1933     \_cs_generate_variant_loop:nNwN { } #3 \q_mark
1934 }
1935 \cs_new:Npn \_cs_generate_variant_loop_same:w
1936   #1 \prg_do_nothing: #2#3#4
1937 {
1938   #3 { #1 \_cs_generate_variant_same:N #2 }
1939 }
1940 \cs_new:Npn \_cs_generate_variant_loop_end:nwwwNNnn
1941   #1#2 \q_mark #3 ~ #4 \q_stop #5#6#7#8
1942 {
1943   \scan_stop: \scan_stop: \fi:
1944   \exp_not:N \q_mark
1945   \exp_not:N \q_stop
1946   \exp_not:N #6
1947   \exp_not:c { #7 : #8 #1 #3 }
1948 }
1949 \cs_new:Npn \_cs_generate_variant_loop_long:wNNnn #1 \q_stop #2#3#4#5
1950 {
1951   \exp_not:n
1952   {
1953     \q_mark
1954     \_msg_kernel_error:nxxx { kernel } { variant-too-long }
1955     {#5} { \token_to_str:N #3 }
1956     \use_none:nnnn
1957     \q_stop
1958     #3
1959     #3
1960   }
1961 }
1962 \cs_new:Npn \_cs_generate_variant_loop_invalid:NNwNNnn
1963   #1#2 \fi: \fi: \fi: #3 \q_stop #4#5#6#7
1964 {
1965   \fi: \fi: \fi:
1966   \exp_not:n
1967   {
1968     \q_mark
1969     \_msg_kernel_error:nxxxxx { kernel } { invalid-variant }
1970     {#7} { \token_to_str:N #5 } {#1} {#2}
1971     \use_none:nnnn
1972     \q_stop
1973     #5
1974     #5
1975   }
1976 }

```

(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.

```

1977 \cs_new:Npn \__cs_generate_variant_same:N #1
1978 {
1979   \if:w N #1
1980     N
1981   \else:
1982     \if:w p #1
1983       p
1984     \else:
1985       n
1986     \fi:
1987   \fi:
1988 }

```

(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.

```

1989 \cs_new_protected:Npn \__cs_generate_variant:wwNN
1990   #1 \q_mark #2 \q_stop #3#4
1991 {
1992   #2
1993   \cs_if_free:NTF #4
1994   {
1995     \group_begin:
1996     \__cs_generate_internal_variant:n {#1}
1997     \__cs_tmp:w #4 { \exp_not:c { exp_args:N #1 } \exp_not:N #3 }
1998     \group_end:
1999   }
2000   {
2001     \iow_log:x
2002     {
2003       Variant~\token_to_str:N #4~%
2004       already~defined;~ not~ changing~ it~on~line~%
2005       \tex_the:D \tex_inputlineno:D
2006     }
2007   }
2008 }

```

(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.

```

2009 \group_begin:
2010 \tex_catcode:D '\X = 12 \scan_stop:
2011 \tex_lccode:D '\N = '\N \scan_stop:
2012 \tex_lowercase:D
2013 {

```

```

2014 \group_end:
2015 \cs_new_protected:Npn \__cs_generate_internal_variant:n #1
2016 {
2017   \__cs_generate_internal_variant:wwnNwnn
2018   #1 \q_mark
2019   { \cs_set_eq:NN \__cs_tmp:w \cs_new_protected_nopar:Npx }
2020   \cs_new_protected_nopar:cpx
2021   X \q_mark
2022   { }
2023   \cs_new_nopar:cpx
2024   \q_stop
2025   { exp_args:N #1 }
2026   { \__cs_generate_internal_variant_loop:n #1 { : \use_i:nn } }
2027 }
2028 \cs_new_protected:Npn \__cs_generate_internal_variant:wwnNwnn
2029 #1 X #2 \q_mark #3 #4 #5 \q_stop #6 #7
2030 {
2031   #3
2032   \cs_if_free:cT {#6} { #4 {#6} {#7} }
2033 }
2034 }

```

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.

```

2035 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
2036 {
2037   \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
2038   \__cs_generate_internal_variant_loop:n
2039 }

```

(End definition for `__cs_generate_internal_variant:n`. This function is documented on page ??.)

4.7 Variants which cannot be created earlier

```

\str_if_eq_p:Vn These cannot come earlier as they need \cs_generate_variant:Nn.
\str_if_eq_p:on 2040 \cs_generate_variant:Nn \str_if_eq_p:nn { V , o }
\str_if_eq_p:nV 2041 \cs_generate_variant:Nn \str_if_eq_p:nn { nV , no , VV }
\str_if_eq_p:no 2042 \cs_generate_variant:Nn \str_if_eq:nnT { V , o }
\str_if_eq_p:VV 2043 \cs_generate_variant:Nn \str_if_eq:nnT { nV , no , VV }
\str_if_eq:VnTF 2044 \cs_generate_variant:Nn \str_if_eq:nnF { V , o }
\str_if_eq:onTF 2045 \cs_generate_variant:Nn \str_if_eq:nnF { nV , no , VV }
\str_if_eq:nVTF 2046 \cs_generate_variant:Nn \str_if_eq:nnTF { V , o }
\str_if_eq:noTF 2047 \cs_generate_variant:Nn \str_if_eq:nnTF { nV , no , VV }
\str_if_eq:VVTF 2048 \cs_generate_variant:Nn \str_case:nn { o }
\str_case:on 2049 \cs_generate_variant:Nn \str_case:nnT { o }
\str_case:onTF 2050 \cs_generate_variant:Nn \str_case:nnF { o }
2051 \cs_generate_variant:Nn \str_case:nnTF { o }

```

(End definition for `\str_if_eq:Vn` and others. These functions are documented on page ??.)

```

\str_case:onnn Deprecated 2013-07-15.
2052 \cs_new_eq:NN \str_case:onnn \str_case:onF
(End definition for \str_case:onnn. This function is documented on page ??.)
2053 </initex | package>

```

5 l3prg implementation

The following test files are used for this code: *m3prg001.lvt,m3prg002.lvt,m3prg003.lvt*.

```

2054 <*initex | package>
2055 <*package>
2056 \ProvidesExplPackage
2057   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
2058   \__expl_package_check:
2059 </package>

```

5.1 Primitive conditionals

`\if_bool:N` Those two primitive TeX conditionals are synonyms. They should not be used outside the kernel code.

```

2060 \tex_let:D \if_bool:N \tex_ifodd:D
2061 \tex_let:D \if_predicate:w \tex_ifodd:D
(End definition for \if_bool:N. This function is documented on page 42.)

```

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

```

2062 <@@=bool>

```

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```

2063 \cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
2064 \cs_generate_variant:Nn \bool_new:N { c }
(End definition for \bool_new:N and \bool_new:c. These functions are documented on page ??.)

```

Setting is already pretty easy.

```

2065 \cs_new_protected:Npn \bool_set_true:N #1
2066   { \cs_set_eq:NN #1 \c_true_bool }
2067 \cs_new_protected:Npn \bool_set_false:N #1
2068   { \cs_set_eq:NN #1 \c_false_bool }
2069 \cs_new_protected:Npn \bool_gset_true:N #1
2070   { \cs_gset_eq:NN #1 \c_true_bool }

```

```

2071 \cs_new_protected:Npn \bool_gset_false:N #1
2072 { \cs_gset_eq:NN #1 \c_false_bool }
2073 \cs_generate_variant:Nn \bool_set_true:N { c }
2074 \cs_generate_variant:Nn \bool_set_false:N { c }
2075 \cs_generate_variant:Nn \bool_gset_true:N { c }
2076 \cs_generate_variant:Nn \bool_gset_false:N { c }

```

(End definition for `\bool_set_true:N` and others. These functions are documented on page ??.)

`\bool_set_eq:NN` The usual copy code.

```

\bool_set_eq:cN 2077 \cs_new_eq:NN \bool_set_eq:NN \cs_set_eq:NN
\bool_set_eq:Nc 2078 \cs_new_eq:NN \bool_set_eq:Nc \cs_set_eq:Nc
\bool_set_eq:cc 2079 \cs_new_eq:NN \bool_set_eq:cN \cs_set_eq:cN
\bool_gset_eq:NN 2080 \cs_new_eq:NN \bool_set_eq:cc \cs_set_eq:cc
\bool_gset_eq:cN 2081 \cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
\bool_gset_eq:Nc 2082 \cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
\bool_gset_eq:cN 2083 \cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
\bool_gset_eq:cc 2084 \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 ??.)

`\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 2085 \cs_new_protected:Npn \bool_set:Nn #1#2
\bool_gset:cn 2086 { \tex_chardef:D #1 = \bool_if_p:n {#2} }
2087 \cs_new_protected:Npn \bool_gset:Nn #1#2
2088 { \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2} }
2089 \cs_generate_variant:Nn \bool_set:Nn { c }
2090 \cs_generate_variant:Nn \bool_gset:Nn { c }

```

(End definition for `\bool_set:Nn` and `\bool_set:cn`. These functions are documented on page ??.)

`\bool_if_p:N` Straight forward here. We could optimize here if we wanted to as the boolean can just
`\bool_if_p:c` be input directly.

```

\bool_if:NTF 2091 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
\bool_if:cTF 2092 {
2093   \if_meaning:w \c_true_bool #1
2094   \prg_return_true:
2095   \else:
2096   \prg_return_false:
2097   \fi:
2098 }
2099 \cs_generate_variant:Nn \bool_if_p:N { c }
2100 \cs_generate_variant:Nn \bool_if:NT { c }
2101 \cs_generate_variant:Nn \bool_if:NF { c }
2102 \cs_generate_variant:Nn \bool_if:NTF { c }

```

(End definition for `\bool_if:N` and `\bool_if:c`. These functions are documented on page ??.)

`\bool_show:N` Show the truth value of the boolean, as true or false. We use `_msg_show_variable:n`
`\bool_show:c` to get a better output; this function requires its argument to start with `>~`.

```

\bool_show:n 2103 \cs_new_protected:Npn \bool_show:N #1

```

```

2104 {
2105     \bool_if_exist:NTF #1
2106     { \bool_show:n {#1} }
2107     {
2108         \_msg_kernel_error:nxx { kernel } { variable-not-defined }
2109         { \token_to_str:N #1 }
2110     }
2111 }
2112 \cs_new_protected:Npn \bool_show:n #1
2113 {
2114     \bool_if:nTF {#1}
2115     { \_msg_show_variable:n { > ~ true } }
2116     { \_msg_show_variable:n { > ~ false } }
2117 }
2118 \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 38.)

`\l_tmpa_bool` A few booleans just if you need them.

```

\l_tmpb_bool 2119 \bool_new:N \l_tmpa_bool
\g_tmpa_bool 2120 \bool_new:N \l_tmpb_bool
\g_tmpb_bool 2121 \bool_new:N \g_tmpa_bool
2122 \bool_new:N \g_tmpb_bool

```

(End definition for `\l_tmpa_bool` and others. These variables are documented on page 38.)

`\bool_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\bool_if_exist_p:c 2123 \prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N { TF , T , F , p }
\bool_if_exist:NTF 2124 \prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c { TF , T , F , p }
\bool_if_exist:cTF

```

(End definition for `\bool_if_exist:N` and `\bool_if_exist:c`. These functions are documented on page ??.)

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 `<true>` or `<false>`.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

$\langle true \rangle$ 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.

```

2125 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
2126 {
2127   \if_predicate:w \bool_if_p:n {#1}
2128   \prg_return_true:
2129   \else:
2130   \prg_return_false:
2131   \fi:
2132 }
```

(End definition for `\bool_if:n`. These functions are documented on page 39.)

```

\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 T_EX. 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.

```

2133 \cs_new:Npn \bool_if_p:n #1
2134 {
2135   \group_align_safe_begin:
2136   \__bool_if_left_parentheses:wwwn \q_nil
2137   #1 \q_mark { }
2138   ( \q_mark { \__bool_if_right_parentheses:wwwn \q_nil }
2139   ) \q_mark { \__bool_if_or:wwwn \q_nil }
2140   || \q_mark \__bool_if_parse:NNNww
2141   \q_stop
2142 }
2143 \cs_new:Npn \__bool_if_left_parentheses:wwwn #1 \q_nil #2 ( #3 \q_mark #4
2144 { #4 \__bool_if_left_parentheses:wwwn #1 #2 (( \q_nil #3 \q_mark {#4} }
2145 \cs_new:Npn \__bool_if_right_parentheses:wwwn #1 \q_nil #2 ) #3 \q_mark #4
2146 { #4 \__bool_if_right_parentheses:wwwn #1 #2 )) \q_nil #3 \q_mark {#4} }
2147 \cs_new:Npn \__bool_if_or:wwwn #1 \q_nil #2 || #3 \q_mark #4
2148 { #4 \__bool_if_or:wwwn #1 #2 )||( \q_nil #3 \q_mark {#4} }

```

(End definition for `\bool_if_p:n`. This function is documented on page 39.)

`__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.

```

2149 \cs_new:Npn \__bool_if_parse:NNNww #1#2#3#4 \q_mark #5 \q_stop
2150 {
2151   \__bool_get_next:NN \use_i:nn (( #4 )) S
2152 }

```

(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.

```

2153 \cs_new:Npn \__bool_get_next:NN #1#2
2154 {
2155   \use:c
2156   {
2157     __bool_
2158     \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
2159     :Nw
2160   }
2161   #1 #2

```


2162 }
 (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.

2163 \cs_new:cpn { _bool_!:Nw } #1#2
 2164 { \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.

2165 \cs_new:cpn { _bool_(:Nw } #1#2
 2166 {
 2167 \exp_after:wN _bool_choose:NNN \exp_after:wN #1
 2168 _int_value:w _bool_get_next:NN \use_i:nn
 2169 }
 (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.

2170 \cs_new:cpn { _bool_p:Nw } #1
 2171 { \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.

2172 \cs_new:Npn _bool_choose:NNN #1#2#3
 2173 {
 2174 \use:c
 2175 {
 2176 _bool_ #3 _
 2177 #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: }
 2178 :w
 2179 }
 2180 }
 (End definition for _bool_choose:NNN.)

bool)_0:w Closing a group is just about returning the result. The Stop operation is similar except it closes the special alignment group before returning the boolean.

bool)_1:w
 _bool_S_0:w 2181 \cs_new_nopar:cpn { _bool_)_0:w } { \c_false_bool }
 _bool_S_1:w 2182 \cs_new_nopar:cpn { _bool_)_1:w } { \c_true_bool }
 2183 \cs_new_nopar:cpn { _bool_S_0:w } { \group_align_safe_end: \c_false_bool }
 2184 \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` 2185 `\cs_new_nopar:cpn { __bool_&_1:w } & { __bool_get_next:NN \use_i:nn }`
 2186 `\cs_new_nopar:cpn { __bool_|_0:w } | { __bool_get_next:NN \use_i:nn }`

(End definition for `__bool_&_1:w`. This function is documented on page 39.)

`__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` 2187 `\cs_new_nopar:cpn { __bool_&_0:w } & { __bool_eval_skip_to_end_auxi:Nw \c_false_bool }`
`__bool_eval_skip_to_end_auxiii:Nw` 2188 `\cs_new_nopar:cpn { __bool_|_1:w } | { __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.

```
2189 %% (
2190 \cs_new:Npn \__bool_eval_skip_to_end_auxi:Nw #1#2 )
2191 {
2192   \__bool_eval_skip_to_end_auxii:Nw #1#2 ( % )
2193   \q_no_value \q_stop
2194   {#2}
2195 }
```

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

```
2196 \cs_new:Npn \__bool_eval_skip_to_end_auxii:Nw #1#2 ( #3#4 \q_stop #5 % )
2197 {
2198   \quark_if_no_value:NTF #3
2199   {#1}
2200   { \__bool_eval_skip_to_end_auxiii:Nw #1 #5 }
2201 }
```

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!

```
2202 \cs_new:Npn \__bool_eval_skip_to_end_auxiii:Nw #1#2 ( #3 )
2203 { % (
2204   \__bool_eval_skip_to_end_auxi:Nw #1#3 )
2205 }
```

(End definition for __bool_&_0:w. This function is documented on page 39.)

\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.

```
2206 \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 40.)

\bool_xor_p:nn Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```
2207 \cs_new:Npn \bool_xor_p:nn #1#2
2208 {
2209   \int_compare:nNnTF { \bool_if_p:n {#1} } = { \bool_if_p:n {#2} }
2210   \c_false_bool
2211   \c_true_bool
2212 }
```

(End definition for \bool_xor_p:nn. This function is documented on page 40.)

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

```

2213 \cs_new:Npn \bool_while_do:Nn #1#2
2214 { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
2215 \cs_new:Npn \bool_until_do:Nn #1#2
2216 { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
2217 \cs_generate_variant:Nn \bool_while_do:Nn { c }
2218 \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 ??.)

\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

```

2219 \cs_new:Npn \bool_do_while:Nn #1#2
2220 { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
2221 \cs_new:Npn \bool_do_until:Nn #1#2
2222 { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
2223 \cs_generate_variant:Nn \bool_do_while:Nn { c }
2224 \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 ??.)

\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

```

2225 \cs_new:Npn \bool_while_do:nn #1#2
2226 {
2227   \bool_if:nT {#1}
2228   {
2229     #2
2230     \bool_while_do:nn {#1} {#2}
2231   }
2232 }
2233 \cs_new:Npn \bool_do_while:nn #1#2
2234 {
2235   #2
2236   \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
2237 }
2238 \cs_new:Npn \bool_until_do:nn #1#2
2239 {
2240   \bool_if:nF {#1}
2241   {
2242     #2
2243     \bool_until_do:nn {#1} {#2}
2244   }
2245 }
2246 \cs_new:Npn \bool_do_until:nn #1#2
2247 {

```

```
2248     #2
2249     \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
2250 }
```

(End definition for `\bool_while_do:nn` and others. These functions are documented on page 40.)

5.6 Producing n copies

2251 <@@=prg>

This function uses a cascading cname 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 `__int_to_roman: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:` An alternative approach is to create a string of m's with `__int_to_roman: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.

```

2252 \cs_new:Npn \prg_replicate:nn #1
2253 {
2254   \__int_to_roman:w
2255   \exp_after:wN \__prg_replicate_first:N
2256   \__int_value:w \__int_eval:w #1 \__int_eval_end:
2257   \cs_end:
2258 }
2259 \cs_new:Npn \__prg_replicate:N #1
2260 { \cs:w \__prg_replicate_#1 :n \__prg_replicate:N }
2261 \cs_new:Npn \__prg_replicate_first:N #1
2262 { \cs:w \__prg_replicate_first_#1 :n \__prg_replicate:N }

```

Then comes all the functions that do the hard work of inserting all the copies. The first function takes `:n` as a parameter.

```
2263 \cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
2264 \cs_new:cpn { __prg_replicate_0:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
2265 \cs_new:cpn { __prg_replicate_1:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1 }
2266 \cs_new:cpn { __prg_replicate_2:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1 }
2267 \cs_new:cpn { __prg_replicate_3:n } #1
```

Users shouldn't ask for something to be replicated once or even not at all but...

End definition for \prg_replicate:nn. This function is documented on page 41.)

5.7 Detecting T_EX's mode

```
\mode_if_vertical_p: For testing vertical mode. Strikes me here on the bus with David, that as long as
\mode_if_vertical:TF we are just talking about returning true and false states, we can just use the primitive
conditionals for this and gobbling the \c_zero in the input stream. However this requires
knowledge of the implementation so we keep things nice and clean and use the return
statements.
```

```
2296 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
2297 { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }
```

End definition for \mode_if_vertical:. These functions are documented on page 41.)

`\mode_if_horizontal_p:` For testing horizontal mode.

```

\mode_if_horizontal:TF
2298 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
2299 { \if mode horizontal: \prg_return true: \else: \prg_return false: \fi: }

```

End definition for \mode_if_horizontal:. These functions are documented on page 41.)

`\mode_if_inner_p:` For testing inner mode.
`\mode_if_inner:TF`

```

2300 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
2301 { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_inner:`. These functions are documented on page 41.)

`\mode_if_math_p:` For testing math mode. At the beginning of an alignment cell, the programmer should
`\mode_if_math:TF` insert `\scan_align_safe_stop:` before the test.

```

2302 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
2303 { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_math:`. These functions are documented on page 41.)

5.8 Internal programming functions

`\group_align_safe_begin:` `\group_align_safe_end:` T_EX’s alignment structures present many problems. As Knuth says himself in *T_EX: The Program*: “It’s sort of a miracle whenever `\halign` or `\valign` work, [...]” One problem relates to commands that internally issues a `\cr` but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a `&` with category code 4 we will get some sort of weird error message because the underlying `\futurelet` will store the token at the end of the alignment template. This could be a `&4` giving a message like `! Misplaced \cr.` or even worse: it could be the `\endtemplate` token causing even more trouble! To solve this we have to open a special group so that T_EX still thinks it’s on safe ground but at the same time we don’t want to introduce any brace group that may find its way to the output. The following functions help with this by using code documented only in Appendix D of *The T_EXbook*... We place the `\if_false: { \fi: }` part at that place so that the successive expansions of `\group_align_safe_begin/end:` are always brace balanced.

```

2304 \cs_new_nopar:Npn \group_align_safe_begin:
2305 { \if_int_compare:w \if_false: { \fi: ' } = \c_zero \fi: }
2306 \cs_new_nopar:Npn \group_align_safe_end:
2307 { \if_int_compare:w '{ = \c_zero } \fi: }

```

(End definition for `\group_align_safe_begin:` and `\group_align_safe_end:`.)

`\scan_align_safe_stop:` When T_EX is in the beginning of an align cell (right after the `\cr` or `&`) it is in a somewhat strange mode as it is looking ahead to find an `\omit` or `\noalign` and hasn’t looked at the preamble yet. Thus an `\ifmmode` test at the start of an array cell (where math mode is introduced by the preamble, not in the cell itself) will always fail unless we stop T_EX from scanning ahead. With ε -T_EX’s first version, this required inserting `\scan_stop:`, but not in all cases (see below). This is no longer needed with a newer ε -T_EX, since protected macros are not expanded anymore at the beginning of an alignment cell. We can thus use an empty protected macro to stop T_EX.

```

2308 \cs_new_protected_nopar:Npn \scan_align_safe_stop: { }

```

Let us now explain the earlier version. We don’t want to insert a `\scan_stop:` every time as that will destroy kerning between letters³ Unfortunately there is no way to detect if we’re in the beginning of an alignment cell as they have different characteristics depending

³Unless we enforce an extra pass with an appropriate value of `\pretolerance`.

on column number, *etc.* However we *can* detect if we're in an alignment cell by checking the current group type and we can also check if the previous node was a character or ligature. What is done here is that `\scan_stop:` is only inserted if and only if a) we're in the outer part of an alignment cell and b) the last node *wasn't* a char node or a ligature node. Thus an older definition here was

```
\cs_new_nopar:Npn \scan_align_safe_stop:
{
  \int_compare:nNnT \etex_currentgrouptype:D = \c_six
  {
    \int_compare:nNnF \etex_lastnodetype:D = \c_zero
    {
      \int_compare:nNnF \etex_lastnodetype:D = \c_seven
      { \scan_stop: }
    }
  }
}
```

However, this is not truly expandable, as there are places where the `\scan_stop:` ends up in the result.

(End definition for `\scan_align_safe_stop:.`)

2309 <@@=prg>

`__prg_variable_get_scope:N` Expandable functions to find the type of a variable, and to return g if the variable is global. The trick for `__prg_variable_get_scope:N` is the same as that in `__cs-split_function:NN`, but it can be simplified as the requirements here are less complex.

```
__prg_variable_get_scope:w
__prg_variable_get_type:N
__prg_variable_get_type:w
2310 \group_begin:
2311 \tex_lccode:D '*' = 'g \scan_stop:
2312 \tex_catcode:D '*' = \c_twelve
2313 \tl_to_lowercase:n
2314 {
2315   \group_end:
2316   \cs_new:Npn __prg_variable_get_scope:N #1
2317   {
2318     \exp_after:wN \exp_after:wN
2319     \exp_after:wN __prg_variable_get_scope:w
2320     \cs_to_str:N #1 \exp_stop_f: \q_stop
2321   }
2322   \cs_new:Npn __prg_variable_get_scope:w #1#2 \q_stop
2323   { \token_if_eq_meaning:NNT * #1 { g } }
2324 }
2325 \group_begin:
2326 \tex_lccode:D '*' = ' _ \scan_stop:
2327 \tex_catcode:D '*' = \c_twelve
2328 \tl_to_lowercase:n
2329 {
2330   \group_end:
2331   \cs_new:Npn __prg_variable_get_type:N #1
```



```

2332     {
2333         \exp_after:wN \__prg_variable_get_type:w
2334         \token_to_str:N #1 * a \q_stop
2335     }
2336     \cs_new:Npn \__prg_variable_get_type:w #1 * #2#3 \q_stop
2337     {
2338         \token_if_eq_meaning:NNTF a #2
2339         {#1}
2340         { \__prg_variable_get_type:w #2#3 \q_stop }
2341     }
2342 }

```

(End definition for __prg_variable_get_scope:N. This function is documented on page 42.)

\g__prg_map_int A nesting counter for mapping.

```

2343 \int_new:N \g__prg_map_int

```

(End definition for \g__prg_map_int. This variable is documented on page 43.)

__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 43.)

__prg_break_point: Also done in l3basics as in format mode these are needed within l3alloc.

__prg_break:

(End definition for __prg_break_point:. This function is documented on page ??.)

__prg_break:n

```

2344 </initex | package>

```

6 l3quark implementation

The following test files are used for this code: m3quark001.lvt.

```

2345 <*initex | package>
2346 <*package>
2347 \ProvidesExplPackage
2348   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
2349 \__expl_package_check:
2350 </package>

```

6.1 Quarks

\quark_new:N Allocate a new quark.

```

2351 \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 45.)

\q_nil Some “public” quarks. \q_stop is an “end of argument” marker, \q_nil is a empty value and \q_no_value marks an empty argument.

\q_mark

\q_no_value

\q_stop

```

2352 \quark_new:N \q_nil
2353 \quark_new:N \q_mark
2354 \quark_new:N \q_no_value
2355 \quark_new:N \q_stop

```

(End definition for `\q_nil` and others. These variables are documented on page 45.)

`\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.

```
2356 \quark_new:N \q_recursion_tail
2357 \quark_new:N \q_recursion_stop
```

(End definition for `\q_recursion_tail` and `\q_recursion_stop`. These variables are documented on page 46.)

`\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.

```
2358 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
2359 {
2360   \if_meaning:w \q_recursion_tail #1
2361   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2362   \fi:
2363 }
2364 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
2365 {
2366   \if_meaning:w \q_recursion_tail #1
2367   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2368   \else:
2369   \exp_after:wN \use_none:n
2370   \fi:
2371 }
```

(End definition for `\quark_if_recursion_tail_stop:N`. This function is documented on page 46.)

`\quark_if_recursion_tail_stop:n` The same idea applies when testing multiple tokens, but here we just compare the token list to `\q_recursion_tail` as a string.

```
\quark_if_recursion_tail_stop:o
\quark_if_recursion_tail_stop_do:nn
\quark_if_recursion_tail_stop_do:on
2372 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
2373 {
2374   \if_int_compare:w \pdfTeX_strcmp:D
2375   { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
2376   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2377   \fi:
2378 }
2379 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
2380 {
2381   \if_int_compare:w \pdfTeX_strcmp:D
2382   { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
2383   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2384   \else:
```

```

2385     \exp_after:wN \use_none:n
2386     \fi:
2387   }
2388   \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
2389   \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 ??.)

`_quark_if_recursion_tail_break:NN` Analogs of the `\quark_if_recursion_tail_stop...` functions. Break the mapping using #2.

```

2390   \cs_new:Npn \_quark_if_recursion_tail_break:NN #1#2
2391   {
2392     \if_meaning:w \q_recursion_tail #1
2393     \exp_after:wN #2
2394     \fi:
2395   }
2396   \cs_new:Npn \_quark_if_recursion_tail_break:nN #1#2
2397   {
2398     \if_int_compare:w \pdfTeX_strcmp:D
2399     { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
2400     \exp_after:wN #2
2401     \fi:
2402   }

```

(End definition for _quark_if_recursion_tail_break:NN. This function is documented on page ??.)

`\quark_if_nil_p:N` Here we test if we found a special quark as the first argument. We better start with `\q_no_value` as the first argument since the whole thing may otherwise loop if #1 is wrongly given a string like aabc instead of a single token.⁴

```

\quark_if_nil:NTF
\quark_if_no_value_p:N
\quark_if_no_value_p:c
\quark_if_no_value:NTF
\quark_if_no_value:cTF
2403   \prg_new_conditional:Nnn \quark_if_nil:N { p, T, F, TF }
2404   {
2405     \if_meaning:w \q_nil #1
2406     \prg_return_true:
2407     \else:
2408     \prg_return_false:
2409     \fi:
2410   }
2411   \prg_new_conditional:Nnn \quark_if_no_value:N { p, T, F, TF }
2412   {
2413     \if_meaning:w \q_no_value #1
2414     \prg_return_true:
2415     \else:
2416     \prg_return_false:
2417     \fi:
2418   }
2419   \cs_generate_variant:Nn \quark_if_no_value_p:N { c }
2420   \cs_generate_variant:Nn \quark_if_no_value:NT { c }
2421   \cs_generate_variant:Nn \quark_if_no_value:NF { c }
2422   \cs_generate_variant:Nn \quark_if_no_value:NTF { c }

```

⁴It may still loop in special circumstances however!

(End definition for `\quark_if_nil:N`. These functions are documented on page ??.)

```

\quark_if_nil_p:n These are essentially \str_if_eq:nn tests but done directly.
\quark_if_nil_p:V 2423 \prg_new_conditional:Nnn \quark_if_nil:n { p, T , F , TF }
\quark_if_nil_p:o 2424 {
\quark_if_nil:nTF 2425   \if_int_compare:w \pdfTeX_strcmp:D
\quark_if_nil:VTF 2426   { \exp_not:N \q_nil } { \exp_not:n {#1} } = \c_zero
\quark_if_nil:oTF 2427   \prg_return_true:
\quark_if_no_value_p:n 2428   \else:
\quark_if_no_value:nTF 2429   \prg_return_false:
2430   \fi:
2431 }
2432 \prg_new_conditional:Nnn \quark_if_no_value:n { p, T , F , TF }
2433 {
2434   \if_int_compare:w \pdfTeX_strcmp:D
2435   { \exp_not:N \q_no_value } { \exp_not:n {#1} } = \c_zero
2436   \prg_return_true:
2437   \else:
2438   \prg_return_false:
2439   \fi:
2440 }
2441 \cs_generate_variant:Nn \quark_if_nil_p:n { V , o }
2442 \cs_generate_variant:Nn \quark_if_nil:nTF { V , o }
2443 \cs_generate_variant:Nn \quark_if_nil:nT { V , o }
2444 \cs_generate_variant:Nn \quark_if_nil:nF { V , o }

```

(End definition for `\quark_if_nil:n`, `\quark_if_nil:V`, and `\quark_if_nil:o`. These functions are documented on page 45.)

`\q__tl_act_mark` These private quarks are needed by `l3tl`, but that is loaded before the quark module,
`\q__tl_act_stop` hence their definition is deferred.

```

2445 \quark_new:N \q__tl_act_mark
2446 \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

```

2447 <@@=scan>

```

`\g__scan_marks_tl` The list of all scan marks currently declared.

```

2448 \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.

```

2449 \cs_new_protected:Npn \__scan_new:N #1
2450 {
2451   \tl_if_in:NnTF \g__scan_marks_tl { #1 }
2452   {

```

```

2453     \_msg_kernel_error:nmx { kernel } { scanmark-already-defined }
2454     { \token_to_str:N #1 }
2455   }
2456   {
2457     \tl_gput_right:Nn \g__scan_marks_tl {#1}
2458     \cs_new_eq:NN #1 \scan_stop:
2459   }
2460 }
(End definition for \__scan_new:N.)

```

\s__stop We only declare one scan mark here, more can be defined by specific modules.

```

2461 \__scan_new:N \s__stop
(End definition for \s__stop. This variable is documented on page 48.)

```

__use_none_delimit_by_s__stop:w Similar to \use_none_delimit_by_q_stop:w.

```

2462 \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.

```

2463 \__scan_new:N \s__seq
(End definition for \s__seq. This variable is documented on page 112.)

```

\s_obj_end This scan mark is used both in l3seq and l3prop. This token ought to be defined in a l3obj module if we decide to go in that direction. In the mean-time, we need it to be set.

```

2464 \__scan_new:N \s_obj_end
(End definition for \s_obj_end. This variable is documented on page ??.)

```

6.3 Deprecated quark functions

\quark_if_recursion_tail_break:N It's not clear what breaking function we should be using here, so I'm picking one somewhat arbitrarily.

\quark_if_recursion_tail_break:n

```

2465 \cs_new:Npn \quark_if_recursion_tail_break:N #1
2466 { \__quark_if_recursion_tail_break:NN #1 \prg_break: }
2467 \cs_new:Npn \quark_if_recursion_tail_break:n #1
2468 { \__quark_if_recursion_tail_break:nN {#1} \prg_break: }
(End definition for \quark_if_recursion_tail_break:N and \quark_if_recursion_tail_break:n. These
functions are documented on page ??.)
2469 </initex | package>

```

7 l3token implementation

```

2470 <*initex | package>
2471 <@@=token>
2472 <*package>
2473 \ProvidesExplPackage
2474   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
2475   \_expl_package_check:
2476 </package>

```

7.1 Character tokens

Category code changes.

```

\char_set_catcode:nn
\char_value_catcode:n
\char_show_value_catcode:n
2477 \cs_new_protected:Npn \char_set_catcode:nn #1#2
2478   { \tex_catcode:D #1 = \_int_eval:w #2 \_int_eval_end: }
2479 \cs_new:Npn \char_value_catcode:n #1
2480   { \tex_the:D \tex_catcode:D \_int_eval:w #1 \_int_eval_end: }
2481 \cs_new_protected:Npn \char_show_value_catcode:n #1
2482   { \tex_showthe:D \tex_catcode:D \_int_eval:w #1 \_int_eval_end: }

```

(End definition for `\char_set_catcode:nn`. This function is documented on page 51.)

```

\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
2483 \cs_new_protected:Npn \char_set_catcode_escape:N #1
2484   { \char_set_catcode:nn { '#1 } \c_zero }
2485 \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
2486   { \char_set_catcode:nn { '#1 } \c_one }
2487 \cs_new_protected:Npn \char_set_catcode_group_end:N #1
2488   { \char_set_catcode:nn { '#1 } \c_two }
2489 \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
2490   { \char_set_catcode:nn { '#1 } \c_three }
2491 \cs_new_protected:Npn \char_set_catcode_alignment:N #1
2492   { \char_set_catcode:nn { '#1 } \c_four }
2493 \cs_new_protected:Npn \char_set_catcode_end_line:N #1
2494   { \char_set_catcode:nn { '#1 } \c_five }
2495 \cs_new_protected:Npn \char_set_catcode_parameter:N #1
2496   { \char_set_catcode:nn { '#1 } \c_six }
2497 \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
2498   { \char_set_catcode:nn { '#1 } \c_seven }
2499 \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
2500   { \char_set_catcode:nn { '#1 } \c_eight }
2501 \cs_new_protected:Npn \char_set_catcode_ignore:N #1
2502   { \char_set_catcode:nn { '#1 } \c_nine }
2503 \cs_new_protected:Npn \char_set_catcode_space:N #1
2504   { \char_set_catcode:nn { '#1 } \c_ten }
2505 \cs_new_protected:Npn \char_set_catcode_letter:N #1
2506   { \char_set_catcode:nn { '#1 } \c_eleven }
2507 \cs_new_protected:Npn \char_set_catcode_other:N #1
2508   { \char_set_catcode:nn { '#1 } \c_twelve }
2509 \cs_new_protected:Npn \char_set_catcode_active:N #1

```

```

2510 { \char_set_catcode:nn { '#1 } \c_thirteen }
2511 \cs_new_protected:Npn \char_set_catcode_comment:N #1
2512 { \char_set_catcode:nn { '#1 } \c_fourteen }
2513 \cs_new_protected:Npn \char_set_catcode_invalid:N #1
2514 { \char_set_catcode:nn { '#1 } \c_fifteen }

```

(End definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 50.)

```

\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
2515 \cs_new_protected:Npn \char_set_catcode_escape:n #1
2516 { \char_set_catcode:nn {#1} \c_zero }
2517 \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
2518 { \char_set_catcode:nn {#1} \c_one }
2519 \cs_new_protected:Npn \char_set_catcode_group_end:n #1
2520 { \char_set_catcode:nn {#1} \c_two }
2521 \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
2522 { \char_set_catcode:nn {#1} \c_three }
2523 \cs_new_protected:Npn \char_set_catcode_alignment:n #1
2524 { \char_set_catcode:nn {#1} \c_four }
2525 \cs_new_protected:Npn \char_set_catcode_end_line:n #1
2526 { \char_set_catcode:nn {#1} \c_five }
2527 \cs_new_protected:Npn \char_set_catcode_parameter:n #1
2528 { \char_set_catcode:nn {#1} \c_six }
2529 \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
2530 { \char_set_catcode:nn {#1} \c_seven }
2531 \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
2532 { \char_set_catcode:nn {#1} \c_eight }
2533 \cs_new_protected:Npn \char_set_catcode_ignore:n #1
2534 { \char_set_catcode:nn {#1} \c_nine }
2535 \cs_new_protected:Npn \char_set_catcode_space:n #1
2536 { \char_set_catcode:nn {#1} \c_ten }
2537 \cs_new_protected:Npn \char_set_catcode_letter:n #1
2538 { \char_set_catcode:nn {#1} \c_eleven }
2539 \cs_new_protected:Npn \char_set_catcode_other:n #1
2540 { \char_set_catcode:nn {#1} \c_twelve }
2541 \cs_new_protected:Npn \char_set_catcode_active:n #1
2542 { \char_set_catcode:nn {#1} \c_thirteen }
2543 \cs_new_protected:Npn \char_set_catcode_comment:n #1
2544 { \char_set_catcode:nn {#1} \c_fourteen }
2545 \cs_new_protected:Npn \char_set_catcode_invalid:n #1
2546 { \char_set_catcode:nn {#1} \c_fifteen }

```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 50.)

```

\char_set_mathcode:nn
\char_value_mathcode:n
\char_show_value_mathcode:n
  \char_set_lccode:nn
  \char_value_lccode:n
\char_show_value_lccode:n
  \char_set_uccode:nn
  \char_value_uccode:n
\char_show_value_uccode:n
  \char_set_sfcode:nn
  \char_value_sfcode:n
\char_show_value_sfcode:n
2547 \cs_new_protected:Npn \char_set_mathcode:nn #1#2
2548 { \tex_mathcode:D #1 = \__int_eval:w #2 \__int_eval_end: }
2549 \cs_new:Npn \char_value_mathcode:n #1
2550 { \tex_the:D \tex_mathcode:D \__int_eval:w #1 \__int_eval_end: }

```

Pretty repetitive, but necessary!

```

2551 \cs_new_protected:Npn \char_show_value_mathcode:n #1
2552 { \tex_showthe:D \tex_mathcode:D \__int_eval:w #1 \__int_eval_end: }
2553 \cs_new_protected:Npn \char_set_lccode:nn #1#2
2554 { \tex_lccode:D #1 = \__int_eval:w #2 \__int_eval_end: }
2555 \cs_new:Npn \char_value_lccode:n #1
2556 { \tex_the:D \tex_lccode:D \__int_eval:w #1 \__int_eval_end: }
2557 \cs_new_protected:Npn \char_show_value_lccode:n #1
2558 { \tex_showthe:D \tex_lccode:D \__int_eval:w #1 \__int_eval_end: }
2559 \cs_new_protected:Npn \char_set_uccode:nn #1#2
2560 { \tex_uccode:D #1 = \__int_eval:w #2 \__int_eval_end: }
2561 \cs_new:Npn \char_value_uccode:n #1
2562 { \tex_the:D \tex_uccode:D \__int_eval:w #1 \__int_eval_end: }
2563 \cs_new_protected:Npn \char_show_value_uccode:n #1
2564 { \tex_showthe:D \tex_uccode:D \__int_eval:w #1 \__int_eval_end: }
2565 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
2566 { \tex_sfcode:D #1 = \__int_eval:w #2 \__int_eval_end: }
2567 \cs_new:Npn \char_value_sfcode:n #1
2568 { \tex_the:D \tex_sfcode:D \__int_eval:w #1 \__int_eval_end: }
2569 \cs_new_protected:Npn \char_show_value_sfcode:n #1
2570 { \tex_showthe:D \tex_sfcode:D \__int_eval:w #1 \__int_eval_end: }

```

(End definition for \char_set_mathcode:nn. This function is documented on page 53.)

7.2 Generic tokens

`\token_to_meaning:N` These are all defined in `l3basics`, as they are needed “early”. This is just a reminder!
`\token_to_meaning:c` (End definition for `\token_to_meaning:N` and `\token_to_meaning:c`. These functions are documented on page ??.)
`\token_to_str:N`
`\token_to_str:c`
`\token_new:Nn`

Creates a new token.

```

2571 \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 53.)

`\c_group_begin_token` We define these useful tokens. We have to do it by hand with the brace tokens for obvious reasons.
`\c_group_end_token`
`\c_math_toggle_token`
`\c_alignment_token`
`\c_parameter_token`

```

2572 \cs_new_eq:NN \c_group_begin_token {
2573 \cs_new_eq:NN \c_group_end_token }
2574 \group_begin:
2575 \char_set_catcode_math_toggle:N \*
2576 \token_new:Nn \c_math_toggle_token { * }
2577 \char_set_catcode_alignment:N \*
2578 \token_new:Nn \c_alignment_token { * }
2579 \token_new:Nn \c_parameter_token { # }
2580 \token_new:Nn \c_math_superscript_token { ^ }
2581 \char_set_catcode_math_subscript:N \*
2582 \token_new:Nn \c_math_subscript_token { * }
2583 \token_new:Nn \c_space_token { ~ }
2584 \token_new:Nn \c_catcode_letter_token { a }
2585 \token_new:Nn \c_catcode_other_token { 1 }
2586 \group_end:

```


(End definition for `\c_group_begin_token` and others. These functions are documented on page 53.)

`\c_catcode_active_tl` Not an implicit token!

```

2587 \group_begin:
2588 \char_set_catcode_active:N \*
2589 \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
2590 \group_end:

```

(End definition for `\c_catcode_active_tl`. This variable is documented on page 53.)

`\l_char_active_seq` Two sequences for dealing with special characters. The first is characters which may be active, and contains the active characters themselves to allow easy redefinition. The
`\l_char_special_seq` 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`.

```

2591 \seq_new:N \l_char_active_seq
2592 \use:n
2593 {
2594   \group_begin:
2595   \char_set_catcode_active:N \"
2596   \char_set_catcode_active:N \$
2597   \char_set_catcode_active:N &
2598   \char_set_catcode_active:N ^
2599   \char_set_catcode_active:N _
2600   \char_set_catcode_active:N ~
2601   \use:nn
2602   {
2603     \group_end:
2604     \seq_set_split:Nnn \l_char_active_seq { }
2605   }
2606 }
2607 { { " $ & ^ _ ~ } } %$
2608 \seq_new:N \l_char_special_seq
2609 \seq_set_split:Nnn \l_char_special_seq { }
2610 { \ \ " \# \$ \% \& \ \ ^ \_ \{ \} \~ }

```

(End definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 53.)

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
`\token_if_group_begin:NTF` this.

```

2611 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
2612 {
2613   \if_catcode:w \exp_not:N #1 \c_group_begin_token
2614   \prg_return_true: \else: \prg_return_false: \fi:
2615 }

```

(End definition for `\token_if_group_begin:N`. These functions are documented on page 54.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.
`\token_if_group_end:N \underline{TF}`

```
2616 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
2617 {
2618   \if_catcode:w \exp_not:N #1 \c_group_end_token
2619   \prg_return_true: \else: \prg_return_false: \fi:
2620 }
```

(End definition for `\token_if_group_end:N`. These functions are documented on page 54.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.
`\token_if_math_toggle:N \underline{TF}`

```
2621 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
2622 {
2623   \if_catcode:w \exp_not:N #1 \c_math_toggle_token
2624   \prg_return_true: \else: \prg_return_false: \fi:
2625 }
```

(End definition for `\token_if_math_toggle:N`. These functions are documented on page 54.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.
`\token_if_alignment:N \underline{TF}`

```
2626 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
2627 {
2628   \if_catcode:w \exp_not:N #1 \c_alignment_token
2629   \prg_return_true: \else: \prg_return_false: \fi:
2630 }
```

(End definition for `\token_if_alignment:N`. These functions are documented on page 54.)

`\token_if_parameter_p:N` Check if token is a parameter token. We use the constant `\c_parameter_token` for this.
`\token_if_parameter:N \underline{TF}` We have to trick \TeX a bit to avoid an error message: within a group we prevent `\c_parameter_token` from behaving like a macro parameter character. The definitions of `\prg_new_conditional:Npnn` are global, so they will remain after the group.

```
2631 \group_begin:
2632 \cs_set_eq:NN \c_parameter_token \scan_stop:
2633 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
2634 {
2635   \if_catcode:w \exp_not:N #1 \c_parameter_token
2636   \prg_return_true: \else: \prg_return_false: \fi:
2637 }
2638 \group_end:
```

(End definition for `\token_if_parameter:N`. These functions are documented on page 55.)

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.
`\token_if_math_superscript:N \underline{TF}`

```
2639 \prg_new_conditional:Npnn \token_if_math_superscript:N #1 { p , T , F , TF }
2640 {
2641   \if_catcode:w \exp_not:N #1 \c_math_superscript_token
2642   \prg_return_true: \else: \prg_return_false: \fi:
2643 }
```

(End definition for `\token_if_math_superscript:N`. These functions are documented on page 55.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_`
`\token_if_math_subscript:N \underline{TF}` token for this.

```

2644 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
2645 {
2646   \if_catcode:w \exp_not:N #1 \c_math_subscript_token
2647   \prg_return_true: \else: \prg_return_false: \fi:
2648 }

```

(End definition for \token_if_math_subscript:N. These functions are documented on page 55.)

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.
`\token_if_space:N \underline{TF}`

```

2649 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
2650 {
2651   \if_catcode:w \exp_not:N #1 \c_space_token
2652   \prg_return_true: \else: \prg_return_false: \fi:
2653 }

```

(End definition for \token_if_space:N. These functions are documented on page 55.)

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.
`\token_if_letter:N \underline{TF}`

```

2654 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
2655 {
2656   \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
2657   \prg_return_true: \else: \prg_return_false: \fi:
2658 }

```

(End definition for \token_if_letter:N. These functions are documented on page 55.)

`\token_if_other_p:N` Check if token is an other char token. We use the constant `\c_catcode_other_token`
`\token_if_other:N \underline{TF}` for this.

```

2659 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
2660 {
2661   \if_catcode:w \exp_not:N #1 \c_catcode_other_token
2662   \prg_return_true: \else: \prg_return_false: \fi:
2663 }

```

(End definition for \token_if_other:N. These functions are documented on page 55.)

`\token_if_active_p:N` Check if token is an active char token. We use the constant `\c_catcode_active_tl` for
`\token_if_active:N \underline{TF}` this. A technical point is that `\c_catcode_active_tl` is in fact a macro expanding to
`\exp_not:N *`, where `*` is active.

```

2664 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
2665 {
2666   \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
2667   \prg_return_true: \else: \prg_return_false: \fi:
2668 }

```

(End definition for \token_if_active:N. These functions are documented on page 55.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.
`\token_if_eq_meaning:NNTF`

```

2669 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
2670 {
2671   \if_meaning:w #1 #2
2672   \prg_return_true: \else: \prg_return_false: \fi:
2673 }

```

(End definition for `\token_if_eq_meaning:NN`. These functions are documented on page 56.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.
`\token_if_eq_catcode:NNTF`

```

2674 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
2675 {
2676   \if_catcode:w \exp_not:N #1 \exp_not:N #2
2677   \prg_return_true: \else: \prg_return_false: \fi:
2678 }

```

(End definition for `\token_if_eq_catcode:NN`. These functions are documented on page 55.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.
`\token_if_eq_charcode:NNTF`

```

2679 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
2680 {
2681   \if_charcode:w \exp_not:N #1 \exp_not:N #2
2682   \prg_return_true: \else: \prg_return_false: \fi:
2683 }

```

(End definition for `\token_if_eq_charcode:NN`. These functions are documented on page 55.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` will always output something like
`\token_if_macro:NTF` `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`.
`_token_if_macro_p:w` However, this can fail the five `\...mark` primitives, whose meaning has the form
`\...mark:<user material>`. The problem is that the `<user material>` can contain `->`.

However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first `:`, and analyse what is left. However, macros can have any combination of `\long`, `\protected` or `\outer` (not used in L^AT_EX3) before the string `macro:.` We thus only select the part of the meaning between the first `ma` and the first following `:`. If this string is `cro`, then we have a macro. If the string is `rk`, then we have a mark. The string can also be `cro parameter character` for a colon with a weird category code (namely the usual category code of `#`). Otherwise, it is empty.

This relies on the fact that `\long`, `\protected`, `\outer` cannot contain `ma`, regardless of the escape character, even if the escape character is `m...`

Both `ma` and `:` must be of category code 12 (other), and we achieve using the standard lowercasing technique.

```

2684 \group_begin:
2685 \char_set_catcode_other:N \M
2686 \char_set_catcode_other:N \A
2687 \char_set_lccode:nn { '\; } { '\: }
2688 \char_set_lccode:nn { '\T } { '\T }
2689 \char_set_lccode:nn { '\F } { '\F }
2690 \tl_to_lowercase:n
2691 {

```

```

2692 \group_end:
2693 \prg_new_conditional:Npnn \token_if_macro:N #1 { p , T , F , TF }
2694 {
2695   \exp_after:wN \__token_if_macro_p:w
2696   \token_to_meaning:N #1 MA; \q_stop
2697 }
2698 \cs_new:Npn \__token_if_macro_p:w #1 MA #2 ; #3 \q_stop
2699 {
2700   \if_int_compare:w \pdfTeX_strcmp:D { #2 } { cro } = \c_zero
2701   \prg_return_true:
2702   \else:
2703     \prg_return_false:
2704   \fi:
2705 }
2706 }

```

(End definition for `\token_if_macro:N`. These functions are documented on page 56.)

`\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.

```

2707 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
2708 {
2709   \if_catcode:w \exp_not:N #1 \scan_stop:
2710   \prg_return_true: \else: \prg_return_false: \fi:
2711 }

```

(End definition for `\token_if_cs:N`. These functions are documented on page 56.)

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that T_EX will temporarily convert `\exp_not:N`
`\token_if_expandable:NTF` `\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).

```

2712 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
2713 {
2714   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
2715   \prg_return_false:
2716   \else:
2717     \if_cs_exist:N #1
2718     \prg_return_true:
2719     \else:
2720       \prg_return_false:
2721     \fi:
2722   \fi:
2723 }

```

(End definition for `\token_if_expandable:N`. These functions are documented on page 56.)

`\token_if_chardef_p:N` Most of these functions have to check the meaning of the token in question so we need to
`\token_if_mathchardef_p:N` do some checkups on which characters are output by `\token_to_meaning:N`. As usual,
`\token_if_dim_register_p:N` these characters have catcode 12 so we must do some serious substitutions in the code
`\token_if_int_register_p:N` 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

```

```

2724 \group_begin:
2725   \char_set_lccode:nn { 'T } { 'T }
2726   \char_set_lccode:nn { 'F } { 'F }
2727   \char_set_lccode:nn { 'X } { 'n }
2728   \char_set_lccode:nn { 'Y } { 't }
2729   \char_set_lccode:nn { 'Z } { 'd }
2730   \tl_map_inline:nn { A C E G H I K L M O P R S U X Y Z R " }
2731     { \char_set_catcode:nn { '#1 } \c_twelve }

```

We convert the token list to lower case and restore the catcode and lowercase code changes.

```

2732 \tl_to_lowercase:n
2733 {
2734   \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).

```

2735   \prg_new_conditional:Npnn \token_if_chardef:N #1 { p , T , F , TF }
2736   {
2737     \__str_if_eq_x_return:nn
2738     {
2739       \exp_after:wN \__token_if_chardef:w
2740       \token_to_meaning:N #1 CHAR" \q_stop
2741     }
2742     { \token_to_str:N \char }
2743   }
2744   \prg_new_conditional:Npnn \token_if_mathchardef:N #1 { p , T , F , TF }
2745   {
2746     \__str_if_eq_x_return:nn
2747     {
2748       \exp_after:wN \__token_if_chardef:w
2749       \token_to_meaning:N #1 CHAR" \q_stop
2750     }
2751     { \token_to_str:N \mathchar }
2752   }
2753   \cs_new:Npn \__token_if_chardef:w #1 CHAR" #2 \q_stop { #1 CHAR }

```

Dim registers are a little more difficult since their `\meaning` has the form `\dimen<number>`, and we must take care of the two primitives `\dimen` and `\dimendef`.

```

2754   \prg_new_conditional:Npnn \token_if_dim_register:N #1 { p , T , F , TF }
2755   {
2756     \if_meaning:w \tex_dimen:D #1
2757     \prg_return_false:
2758   \else:
2759     \if_meaning:w \tex_dimendef:D #1
2760     \prg_return_false:

```

```

2761         \else:
2762             \__str_if_eq_x_return:nn
2763             {
2764                 \exp_after:wN \__token_if_dim_register:w
2765                 \token_to_meaning:N #1 ZIMEX \q_stop
2766             }
2767             { \token_to_str:N \ }
2768         \fi:
2769     \fi:
2770 }
2771 \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`.

```

2772 \prg_new_conditional:Npnn \token_if_int_register:N #1 { p , T , F , TF }
2773 {
2774     % \token_if_chardef:NTF #1 { \prg_return_true: }
2775     % {
2776     %     \token_if_mathchardef:NTF #1 { \prg_return_true: }
2777     %     {
2778     \if_meaning:w \tex_count:D #1
2779     \prg_return_false:
2780     \else:
2781     \if_meaning:w \tex_countdef:D #1
2782     \prg_return_false:
2783     \else:
2784     \__str_if_eq_x_return:nn
2785     {
2786         \exp_after:wN \__token_if_int_register:w
2787         \token_to_meaning:N #1 COUXY \q_stop
2788     }
2789     { \token_to_str:N \ }
2790     \fi:
2791     \fi:
2792     %     }
2793     % }
2794 }
2795 \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.

```

2796 \prg_new_conditional:Npnn \token_if_muskip_register:N #1 { p , T , F , TF }
2797 {
2798     \if_meaning:w \tex_muskip:D #1
2799     \prg_return_false:
2800     \else:
2801     \if_meaning:w \tex_muskipdef:D #1
2802     \prg_return_false:
2803     \else:
2804     \__str_if_eq_x_return:nn
2805     {

```

```

2806         \exp_after:wN \__token_if_muskip_register:w
2807         \token_to_meaning:N #1 MUSKIP \q_stop
2808     }
2809     { \token_to_str:N \ }
2810     \fi:
2811     \fi:
2812 }
2813 \cs_new:Npn \__token_if_muskip_register:w #1 MUSKIP #2 \q_stop { #1 ~ }

```

Skip registers.

```

2814 \prg_new_conditional:Npnn \token_if_skip_register:N #1 { p , T , F , TF }
2815 {
2816     \if_meaning:w \tex_skip:D #1
2817     \prg_return_false:
2818     \else:
2819     \if_meaning:w \tex_skipdef:D #1
2820     \prg_return_false:
2821     \else:
2822     \__str_if_eq_x_return:nn
2823     {
2824         \exp_after:wN \__token_if_skip_register:w
2825         \token_to_meaning:N #1 SKIP \q_stop
2826     }
2827     { \token_to_str:N \ }
2828     \fi:
2829     \fi:
2830 }
2831 \cs_new:Npn \__token_if_skip_register:w #1 SKIP #2 \q_stop { #1 ~ }

```

Toks registers.

```

2832 \prg_new_conditional:Npnn \token_if_toks_register:N #1 { p , T , F , TF }
2833 {
2834     \if_meaning:w \tex_toks:D #1
2835     \prg_return_false:
2836     \else:
2837     \if_meaning:w \tex_toksdef:D #1
2838     \prg_return_false:
2839     \else:
2840     \__str_if_eq_x_return:nn
2841     {
2842         \exp_after:wN \__token_if_toks_register:w
2843         \token_to_meaning:N #1 YOKS \q_stop
2844     }
2845     { \token_to_str:N \ }
2846     \fi:
2847     \fi:
2848 }
2849 \cs_new:Npn \__token_if_toks_register:w #1 YOKS #2 \q_stop { #1 ~ }

```

Protected macros.

```

2850 \prg_new_conditional:Npnn \token_if_protected_macro:N #1

```



```

2851 { p , T , F , TF }
2852 {
2853   \__str_if_eq_x_return:nn
2854   {
2855     \exp_after:wN \__token_if_protected_macro:w
2856     \token_to_meaning:N #1 PROYECY EZ~MACRO \q_stop
2857   }
2858   { \token_to_str:N \ }
2859 }
2860 \cs_new:Npn \__token_if_protected_macro:w
2861 #1 PROYECY EZ~MACRO #2 \q_stop { #1 ~ }

```

Long macros and protected long macros share an auxiliary.

```

2862 \prg_new_conditional:Npnn \token_if_long_macro:N #1 { p , T , F , TF }
2863 {
2864   \__str_if_eq_x_return:nn
2865   {
2866     \exp_after:wN \__token_if_long_macro:w
2867     \token_to_meaning:N #1 LOXG~MACRO \q_stop
2868   }
2869   { \token_to_str:N \ }
2870 }
2871 \prg_new_conditional:Npnn \token_if_protected_long_macro:N #1
2872 { p , T , F , TF }
2873 {
2874   \__str_if_eq_x_return:nn
2875   {
2876     \exp_after:wN \__token_if_long_macro:w
2877     \token_to_meaning:N #1 LOXG~MACRO \q_stop
2878   }
2879   { \token_to_str:N \protected \token_to_str:N \ }
2880 }
2881 \cs_new:Npn \__token_if_long_macro:w #1 LOXG~MACRO #2 \q_stop { #1 ~ }

```

Finally the `\tl_to_lowercase:n` ends!

```

2882 }

```

(End definition for `\token_if_chardef:N` and others. These functions are documented on page 56.)

```

\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 $\langle letters \rangle : \langle user material \rangle$. In other words, the first non-letter is a colon. We remove everything after the first colon.

We are now left with a string, which we must analyze. For primitives, it contains only letters. For non-primitives, it contains either " , or a space, or a digit. Two exceptions remain: `\tex_undefined:D`, which is not a primitive, and `\nullfont`, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for `\nullfont`. Otherwise, we go through characters one by one, and stop at the first character less than 'A (this is not quite a test for "only letters", but is close enough to work in this context). If this first character is : then we have a primitive, or `\tex_undefined:D`, and if it is " or a digit, then the token is not a primitive.

```

2883 \tex_chardef:D \c_token_A_int = 'A ~ %
2884 \group_begin:
2885 \char_set_catcode_other:N \;
2886 \char_set_lccode:nn { '\; } { '\: }
2887 \char_set_lccode:nn { '\T } { '\T }
2888 \char_set_lccode:nn { '\F } { '\F }
2889 \tl_to_lowercase:n {
2890   \group_end:
2891   \prg_new_conditional:Npnn \token_if_primitive:N #1 { p , T , F , TF }
2892   {
2893     \token_if_macro:NTF #1
2894     \prg_return_false:
2895     {
2896       \exp_after:wN \__token_if_primitive:NNw
2897       \token_to_meaning:N #1 ; ; ; \q_stop #1
2898     }
2899   }
2900   \cs_new:Npn \__token_if_primitive:NNw #1#2 #3 ; #4 \q_stop
2901   {
2902     \tl_if_empty:oTF { \__token_if_primitive_space:w #3 ~ }
2903     { \__token_if_primitive_loop:N #3 ; \q_stop }
2904     { \__token_if_primitive_nullfont:N }
2905   }
2906 }
2907 \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
2908 \cs_new:Npn \__token_if_primitive_nullfont:N #1
2909 {
2910   \if_meaning:w \tex_nullfont:D #1
2911   \prg_return_true:
2912   \else:
2913     \prg_return_false:
2914   \fi:
2915 }
2916 \cs_new:Npn \__token_if_primitive_loop:N #1
2917 {
2918   \if_int_compare:w '#1 < \c_token_A_int %

```

```

2919     \exp_after:wN \_token_if_primitive:Nw
2920     \exp_after:wN #1
2921   \else:
2922     \exp_after:wN \_token_if_primitive_loop:N
2923   \fi:
2924 }
2925 \cs_new:Npn \_token_if_primitive:Nw #1 #2 \q_stop
2926 {
2927   \if:w : #1
2928     \exp_after:wN \_token_if_primitive_undefined:N
2929   \else:
2930     \prg_return_false:
2931     \exp_after:wN \use_none:n
2932   \fi:
2933 }
2934 \cs_new:Npn \_token_if_primitive_undefined:N #1
2935 {
2936   \if_cs_exist:N #1
2937     \prg_return_true:
2938   \else:
2939     \prg_return_false:
2940   \fi:
2941 }

```

(End definition for `\token_if_primitive:N`. These functions are documented on page 57.)

7.4 Peeking ahead at the next token

```

2942 <@@=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` 2943 \cs_new_eq:NN \l_peek_token ?

2944 \cs_new_eq:NN \g_peek_token ?

(End definition for `\l_peek_token`. This function is documented on page 58.)

`\l__peek_search_token` The token to search for as an implicit token: cf. `\l__peek_search_tl`.

2945 \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`.

```
2946 \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 2947 \cs_new_nopar:Npn \__peek_true:w { }
\__peek_false:w 2948 \cs_new_nopar:Npn \__peek_true_aux:w { }
\__peek_tmp:w 2949 \cs_new_nopar:Npn \__peek_false:w { }
2950 \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 2951 \cs_new_protected_nopar:Npn \peek_after:Nw
2952 { \tex_futurelet:D \l__peek_token }
2953 \cs_new_protected_nopar:Npn \peek_gafter:Nw
2954 { \tex_global:D \tex_futurelet:D \g__peek_token }
```

(End definition for `\peek_after:Nw`. This function is documented on page 58.)

`__peek_true_remove:w` A function to remove the next token and then regain control.

```
2955 \cs_new_protected:Npn \__peek_true_remove:w
2956 {
2957   \group_align_safe_end:
2958   \tex_afterassignment:D \__peek_true_aux:w
2959   \cs_set_eq:NN \__peek_tmp:w
2960 }
```

(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.

```
2961 \cs_new_protected:Npn \__peek_token_generic:NNTF #1#2#3#4
2962 {
2963   \cs_set_eq:NN \l__peek_search_token #2
2964   \tl_set:Nn \l__peek_search_tl {#2}
2965   \cs_set_nopar:Npx \__peek_true:w
2966   {
2967     \exp_not:N \group_align_safe_end:
2968     \exp_not:n {#3}
2969   }
2970   \cs_set_nopar:Npx \__peek_false:w
2971   {
2972     \exp_not:N \group_align_safe_end:
2973     \exp_not:n {#4}
2974   }
2975   \group_align_safe_begin:
2976   \peek_after:Nw #1
2977 }
2978 \cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
```

```

2979 { \__peek_token_generic:NNTF #1 #2 {#3} { } }
2980 \cs_new_protected:Npn \__peek_token_generic:NNTF #1#2#3
2981 { \__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.

```

2982 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF #1#2#3#4
2983 {
2984   \cs_set_eq:NN \l__peek_search_token #2
2985   \tl_set:Nn \l__peek_search_tl {#2}
2986   \cs_set_eq:NN \__peek_true:w \__peek_true_remove:w
2987   \cs_set_nopar:Npx \__peek_true_aux:w { \exp_not:n {#3} }
2988   \cs_set_nopar:Npx \__peek_false:w
2989   {
2990     \exp_not:N \group_align_safe_end:
2991     \exp_not:n {#4}
2992   }
2993   \group_align_safe_begin:
2994   \peek_after:Nw #1
2995 }
2996 \cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
2997 { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
2998 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF #1#2#3
2999 { \__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.

```

3000 \cs_new_nopar:Npn \__peek_execute_branches_meaning:
3001 {
3002   \if_meaning:w \l_peek_token \l__peek_search_token
3003   \exp_after:wN \__peek_true:w
3004   \else:
3005     \exp_after:wN \__peek_false:w
3006   \fi:
3007 }

```

(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 \TeX 's `\futurelet`, because we can only access the `\meaning` of tokens in that way. In those cases, detected thanks to a comparison with `\scan_stop:`, we grab the following token, and compare it explicitly with the explicit search token stored in `\l_peek_search_tl`. The `\exp_not:N` prevents outer macros (coming from non- \LaTeX 3 code) from blowing up. In the third case, `\l_peek_token` is good enough for the test, and we compare it again with the explicit search token. Just like the peek token, the search token may be of any of the three types above, hence the need to use the explicit token that was given to the peek function.

```

3008 \cs_new_nopar:Npn \__peek_execute_branches_catcode:
3009 { \if_catcode:w \__peek_execute_branches_catcode_aux: }
3010 \cs_new_nopar:Npn \__peek_execute_branches_charcode:
3011 { \if_charcode:w \__peek_execute_branches_catcode_aux: }
3012 \cs_new_nopar:Npn \__peek_execute_branches_catcode_aux:
3013 {
3014     \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
3015     \exp_after:wN \exp_after:wN
3016     \exp_after:wN \__peek_execute_branches_catcode_auxii:N
3017     \exp_after:wN \exp_not:N
3018     \else:
3019     \exp_after:wN \__peek_execute_branches_catcode_auxiii:
3020     \fi:
3021 }
3022 \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
3023 {
3024     \exp_not:N #1
3025     \exp_after:wN \exp_not:N \l_peek_search_tl
3026     \exp_after:wN \__peek_true:w
3027     \else:
3028     \exp_after:wN \__peek_false:w
3029     \fi:
3030     #1
3031 }
3032 \cs_new_nopar:Npn \__peek_execute_branches_catcode_auxiii:
3033 {
3034     \exp_not:N \l_peek_token
3035     \exp_after:wN \exp_not:N \l_peek_search_tl
3036     \exp_after:wN \__peek_true:w
3037     \else:
3038     \exp_after:wN \__peek_false:w
3039     \fi:
3040 }

```

(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- \LaTeX 3 packages). Spaces are removed using a side-effect of `f`-expansion: `\tex_romannumeral:D -'0` removes one space.

```

3041 \cs_new_protected_nopar:Npn \__peek_ignore_spaces_execute_branches:
3042 {
3043   \if_meaning:w \l_peek_token \c_space_token
3044     \exp_after:wN \peek_after:Nw
3045     \exp_after:wN \__peek_ignore_spaces_execute_branches:
3046     \tex_romannumeral:D -'0
3047   \else:
3048     \exp_after:wN \__peek_execute_branches:
3049   \fi:
3050 }

```

(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 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.

```

3051 \group_begin:
3052 \cs_set:Npn \__peek_def:nnnn #1#2#3#4
3053 {
3054   \__peek_def:nnnnn {#1} {#2} {#3} {#4} { TF }
3055   \__peek_def:nnnnn {#1} {#2} {#3} {#4} { T }
3056   \__peek_def:nnnnn {#1} {#2} {#3} {#4} { F }
3057 }
3058 \cs_set:Npn \__peek_def:nnnnn #1#2#3#4#5
3059 {
3060   \cs_new_protected_nopar:cpx { #1 #5 }
3061   {
3062     \tl_if_empty:nF {#2}
3063       { \exp_not:n { \cs_set_eq:NN \__peek_execute_branches: #2 } }
3064     \exp_not:c { #3 #5 }
3065     \exp_not:n {#4}
3066   }
3067 }

```

(End definition for __peek_def:nnnn. This function is documented on page ??.)

\peek_catcode:N~~TF~~ With everything in place the definitions can take place. First for category codes.

\peek_catcode_ignore_spaces:N~~TF~~

\peek_catcode_remove:N~~TF~~

\peek_catcode_remove_ignore_spaces:N~~TF~~

```

3068 \__peek_def:nnnn { peek_catcode:N }
3069 { }
3070 { __peek_token_generic:NN }
3071 { \__peek_execute_branches_catcode: }
3072 \__peek_def:nnnn { peek_catcode_ignore_spaces:N }
3073 { \__peek_execute_branches_catcode: }
3074 { __peek_token_generic:NN }
3075 { \__peek_ignore_spaces_execute_branches: }
3076 \__peek_def:nnnn { peek_catcode_remove:N }
3077 { }
3078 { __peek_token_remove_generic:NN }
3079 { \__peek_execute_branches_catcode: }
3080 \__peek_def:nnnn { peek_catcode_remove_ignore_spaces:N }

```

```

3081 { \__peek_execute_branches_catcode: }
3082 { __peek_token_remove_generic:NN }
3083 { \__peek_ignore_spaces_execute_branches: }

```

(End definition for \peek_catcode:N_{TF} and others. These functions are documented on page 59.)

\peek_charcode:N_{TF}
\peek_charcode_ignore_spaces:N_{TF}
\peek_charcode_remove:N_{TF}
\peek_charcode_remove_ignore_spaces:N_{TF}

Then for character codes.

```

3084 \__peek_def:nnnn { peek_charcode:N }
3085 { }
3086 { __peek_token_generic:NN }
3087 { \__peek_execute_branches_charcode: }
3088 \__peek_def:nnnn { peek_charcode_ignore_spaces:N }
3089 { \__peek_execute_branches_charcode: }
3090 { __peek_token_generic:NN }
3091 { \__peek_ignore_spaces_execute_branches: }
3092 \__peek_def:nnnn { peek_charcode_remove:N }
3093 { }
3094 { __peek_token_remove_generic:NN }
3095 { \__peek_execute_branches_charcode: }
3096 \__peek_def:nnnn { peek_charcode_remove_ignore_spaces:N }
3097 { \__peek_execute_branches_charcode: }
3098 { __peek_token_remove_generic:NN }
3099 { \__peek_ignore_spaces_execute_branches: }

```

(End definition for \peek_charcode:N_{TF} and others. These functions are documented on page 60.)

\peek_meaning:N_{TF}
\peek_meaning_ignore_spaces:N_{TF}
\peek_meaning_remove:N_{TF}
\peek_meaning_remove_ignore_spaces:N_{TF}

Finally for meaning, with the group closed to remove the temporary definition functions.

```

3100 \__peek_def:nnnn { peek_meaning:N }
3101 { }
3102 { __peek_token_generic:NN }
3103 { \__peek_execute_branches_meaning: }
3104 \__peek_def:nnnn { peek_meaning_ignore_spaces:N }
3105 { \__peek_execute_branches_meaning: }
3106 { __peek_token_generic:NN }
3107 { \__peek_ignore_spaces_execute_branches: }
3108 \__peek_def:nnnn { peek_meaning_remove:N }
3109 { }
3110 { __peek_token_remove_generic:NN }
3111 { \__peek_execute_branches_meaning: }
3112 \__peek_def:nnnn { peek_meaning_remove_ignore_spaces:N }
3113 { \__peek_execute_branches_meaning: }
3114 { __peek_token_remove_generic:NN }
3115 { \__peek_ignore_spaces_execute_branches: }
3116 \group_end:

```

(End definition for \peek_meaning:N_{TF} and others. These functions are documented on page 60.)

7.5 Decomposing a macro definition

\token_get_prefix_spec:N
\token_get_arg_spec:N
\token_get_replacement_spec:N
__peek_get_prefix_arg_replacement:wN

We sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the

argument specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token `\scan_stop:` is returned instead.

```

3117 \exp_args:Nno \use:nn
3118 { \cs_new:Npn \__peek_get_prefix_arg_replacement:wN #1 }
3119 { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
3120 { #4 {#1} {#2} {#3} }
3121 \cs_new:Npn \token_get_prefix_spec:N #1
3122 {
3123   \token_if_macro:NTF #1
3124   {
3125     \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3126     \token_to_meaning:N #1 \q_stop \use_i:nnn
3127   }
3128   { \scan_stop: }
3129 }
3130 \cs_new:Npn \token_get_arg_spec:N #1
3131 {
3132   \token_if_macro:NTF #1
3133   {
3134     \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3135     \token_to_meaning:N #1 \q_stop \use_ii:nnn
3136   }
3137   { \scan_stop: }
3138 }
3139 \cs_new:Npn \token_get_replacement_spec:N #1
3140 {
3141   \token_if_macro:NTF #1
3142   {
3143     \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3144     \token_to_meaning:N #1 \q_stop \use_iii:nnn
3145   }
3146   { \scan_stop: }
3147 }

```

(End definition for `\token_get_prefix_spec:N`. This function is documented on page 61.)

```

3148 </initex | package>

```

8 l3int implementation

```

3149 <*initex | package>

```

```

3150 <@@=int>

```

The following test files are used for this code: `m3int001,m3int002,m3int03`.

```

3151 <*package>
3152 \ProvidesExplPackage
3153   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
3154   \__expl_package_check:
3155 </package>

```

`__int_to_roman:w` Done in l3basics.
`\if_int_compare:w` (End definition for `__int_to_roman:w`. This function is documented on page 74.)

`__int_value:w` Here are the remaining primitives for number comparisons and expressions.
`__int_eval:w` 3156 `\cs_new_eq:NN __int_value:w \tex_number:D`
`__int_eval_end:` 3157 `\cs_new_eq:NN __int_eval:w \etex_numexpr:D`
`\if_int_odd:w` 3158 `\cs_new_eq:NN __int_eval_end: \tex_relax:D`
`\if_case:w` 3159 `\cs_new_eq:NN \if_int_odd:w \tex_ifodd:D`
3160 `\cs_new_eq:NN \if_case:w \tex_ifcase:D`
(End definition for `__int_value:w`. This function is documented on page 74.)

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.

```

3161 <*initex>
3162 \cs_set:Npn \int_eval:n #1 { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
3163 </initex>
3164 <*package>
3165 \cs_new:Npn \int_eval:n #1 { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
3166 </package>
(End definition for \int_eval:n. This function is documented on page 62.)

```

`\int_abs:n` Functions for min, max, and absolute value with only one evaluation. The absolute value
`__int_abs:N` is obtained by removing a leading sign if any. All three functions expand in two steps.

```

\int_max:nn 3167 \cs_new:Npn \int_abs:n #1
\int_min:nn 3168 {
\__int_maxmin:wwN 3169 \__int_value:w \exp_after:wN \__int_abs:N
3170 \int_use:N \__int_eval:w #1 \__int_eval_end:
3171 \exp_stop_f:
3172 }
3173 \cs_new:Npn \__int_abs:N #1
3174 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
3175 \cs_set:Npn \int_max:nn #1#2
3176 {
3177 \__int_value:w \exp_after:wN \__int_maxmin:wwN
3178 \int_use:N \__int_eval:w #1 \exp_after:wN ;
3179 \int_use:N \__int_eval:w #2 ;
3180 >
3181 \exp_stop_f:
3182 }
3183 \cs_set:Npn \int_min:nn #1#2
3184 {
3185 \__int_value:w \exp_after:wN \__int_maxmin:wwN
3186 \int_use:N \__int_eval:w #1 \exp_after:wN ;
3187 \int_use:N \__int_eval:w #2 ;
3188 <

```

```

3189     \exp_stop_f:
3190   }
3191   \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
3192   {
3193     \if_int_compare:w #1 #3 #2 ~
3194       #1
3195     \else:
3196       #2
3197     \fi:
3198   }

```

(End definition for `\int_abs:n`. This function is documented on page 63.)

`\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.

```

3199   \cs_new:Npn \int_div_truncate:nn #1#2
3200   {
3201     \int_use:N \__int_eval:w
3202     \exp_after:wN \__int_div_truncate:NwNw
3203     \int_use:N \__int_eval:w #1 \exp_after:wN ;
3204     \int_use:N \__int_eval:w #2 ;
3205     \__int_eval_end:
3206   }
3207   \cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
3208   {
3209     \if_meaning:w 0 #1
3210       \c_zero
3211     \else:
3212       (
3213         #1#2
3214         \if_meaning:w - #1 + \else: - \fi:
3215         ( \if_meaning:w - #3 - \fi: #3#4 - \c_one ) / \c_two
3216       )
3217     \fi:
3218     / #3#4
3219   }

```

For the sake of completeness:

```

3220   \cs_new:Npn \int_div_round:nn #1#2
3221   { \__int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }

```

Finally there's the modulus operation.

```

3222   \cs_new:Npn \int_mod:nn #1#2

```

```

3223 {
3224   \__int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
3225   \__int_value:w \__int_eval:w #1 \exp_after:wN ;
3226   \__int_value:w \__int_eval:w #2 ;
3227   \__int_eval_end:
3228 }
3229 \cs_new:Npn \__int_mod:ww #1; #2;
3230 { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }

```

(End definition for `\int_div_truncate:n`. This function is documented on page 63.)

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.

```

\int_new:c 3231 <*package>
3232 \cs_new_protected:Npn \int_new:N #1
3233 {
3234   \__chk_if_free_cs:N #1
3235   \newcount #1
3236 }
3237 </package>
3238 \cs_generate_variant:Nn \int_new:N { c }

```

(End definition for `\int_new:N` and `\int_new:c`. These functions are documented on page ??.)

`\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.

`\int_const:cn`

```

\__int_constdef:Nw 3239 \cs_new_protected:Npn \int_const:Nn #1#2
\c__max_constdef_int 3240 {
3241   \int_compare:nNnTF {#2} > \c_minus_one
3242   {
3243     \int_compare:nNnTF {#2} > \c__max_constdef_int
3244     {
3245       \int_new:N #1
3246       \int_gset:Nn #1 {#2}
3247     }
3248     {
3249       \__chk_if_free_cs:N #1
3250       \tex_global:D \__int_constdef:Nw #1 =
3251       \__int_eval:w #2 \__int_eval_end:
3252     }
3253   }
3254   {
3255     \int_new:N #1
3256     \int_gset:Nn #1 {#2}
3257   }
3258 }
3259 \cs_generate_variant:Nn \int_const:Nn { c }
3260 \pdfTeX_if_engine:TF
3261 {
3262   \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D

```

```

3263 \tex_mathchardef:D \c__max_constdef_int 32 767 ~
3264 }
3265 {
3266 \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D
3267 \tex_chardef:D \c__max_constdef_int 1 114 111 ~
3268 }

```

(End definition for `\int_const:Nn` and `\int_const:cn`. These functions are documented on page ??.)

`\int_zero:N` Functions that reset an *integer* register to zero.

```

\int_zero:c 3269 \cs_new_protected:Npn \int_zero:N #1 { #1 = \c_zero }
\int_gzero:N 3270 \cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero }
\int_gzero:c 3271 \cs_generate_variant:Nn \int_zero:N { c }
3272 \cs_generate_variant:Nn \int_gzero:N { c }

```

(End definition for `\int_zero:N` and `\int_zero:c`. These functions are documented on page ??.)

`\int_zero_new:N` Create a register if needed, otherwise clear it.

```

\int_zero_new:c 3273 \cs_new_protected:Npn \int_zero_new:N #1
\int_gzero_new:N 3274 { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
\int_gzero_new:c 3275 \cs_new_protected:Npn \int_gzero_new:N #1
3276 { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
3277 \cs_generate_variant:Nn \int_zero_new:N { c }
3278 \cs_generate_variant:Nn \int_gzero_new:N { c }

```

(End definition for `\int_zero_new:N` and others. These functions are documented on page ??.)

`\int_set_eq:NN` Setting equal means using one integer inside the set function of another.

```

\int_set_eq:cN 3279 \cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
\int_set_eq:Nc 3280 \cs_generate_variant:Nn \int_set_eq:NN { c }
\int_set_eq:cc 3281 \cs_generate_variant:Nn \int_set_eq:NN { Nc , cc }
\int_gset_eq:NN 3282 \cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\int_gset_eq:cN 3283 \cs_generate_variant:Nn \int_gset_eq:NN { c }
\int_gset_eq:Nc 3284 \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 ??.)

`\int_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\int_if_exist_p:c 3285 \prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N { TF , T , F , p }
\int_if_exist:NTF 3286 \prg_new_eq_conditional:NNn \int_if_exist:c \cs_if_exist:c { TF , T , F , p }
\int_if_exist:cTF

```

(End definition for `\int_if_exist:N` and `\int_if_exist:c`. These functions are documented on page ??.)

8.3 Setting and incrementing integers

`\int_add:Nn` Adding and subtracting to and from a counter ...

```

\int_add:cn 3287 \cs_new_protected:Npn \int_add:Nn #1#2
\int_gadd:Nn 3288 { \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
\int_gadd:cn 3289 \cs_new_protected:Npn \int_sub:Nn #1#2
\int_sub:Nn 3290 { \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
\int_sub:cn 3291 \cs_new_protected_nopar:Npn \int_gadd:Nn
\int_gsub:Nn 3292 { \tex_global:D \int_add:Nn }
\int_gsub:cn

```

```

3293 \cs_new_protected_nopar:Npn \int_gsub:Nn
3294 { \tex_global:D \int_sub:Nn }
3295 \cs_generate_variant:Nn \int_add:Nn { c }
3296 \cs_generate_variant:Nn \int_gadd:Nn { c }
3297 \cs_generate_variant:Nn \int_sub:Nn { c }
3298 \cs_generate_variant:Nn \int_gsub:Nn { c }

```

(End definition for `\int_add:Nn` and `\int_add:cn`. These functions are documented on page ??.)

`\int_incr:N` Incrementing and decrementing of integer registers is done with the following functions.

```

\int_incr:c 3299 \cs_new_protected:Npn \int_incr:N #1
\int_gincr:N 3300 { \tex_advance:D #1 \c_one }
\int_gincr:c 3301 \cs_new_protected:Npn \int_decr:N #1
\int_decr:N 3302 { \tex_advance:D #1 \c_minus_one }
\int_decr:c 3303 \cs_new_protected_nopar:Npn \int_gincr:N
\int_gdecr:N 3304 { \tex_global:D \int_incr:N }
\int_gdecr:c 3305 \cs_new_protected_nopar:Npn \int_gdecr:N
3306 { \tex_global:D \int_decr:N }
3307 \cs_generate_variant:Nn \int_incr:N { c }
3308 \cs_generate_variant:Nn \int_decr:N { c }
3309 \cs_generate_variant:Nn \int_gincr:N { c }
3310 \cs_generate_variant:Nn \int_gdecr:N { c }

```

(End definition for `\int_incr:N` and `\int_incr:c`. These functions are documented on page ??.)

`\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 3311 \cs_new_protected:Npn \int_set:Nn #1#2
\int_gset:Nn 3312 { #1 ~ \__int_eval:w #2\__int_eval_end: }
\int_gset:cn 3313 \cs_new_protected_nopar:Npn \int_gset:Nn { \tex_global:D \int_set:Nn }
3314 \cs_generate_variant:Nn \int_set:Nn { c }
3315 \cs_generate_variant:Nn \int_gset:Nn { c }

```

(End definition for `\int_set:Nn` and `\int_set:cn`. These functions are documented on page ??.)

8.4 Using integers

`\int_use:N` Here is how counters are accessed:

```

\int_use:c 3316 \cs_new_eq:NN \int_use:N \tex_the:D
3317 \cs_new:Npn \int_use:c #1 { \int_use:N \cs:w #1 \cs_end: }

```

(End definition for `\int_use:N` and `\int_use:c`. These functions are documented on page ??.)

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. 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.

```

3318 \cs_new_protected_nopar:Npn \__prg_compare_error:
3319 {
3320   \if_int_compare:w \c_zero \c_zero \fi:
3321   =
3322   \__prg_compare_error:
3323 }
3324 \cs_new:Npn \__prg_compare_error:Nw
3325   #1#2 \q_stop
3326 {
3327   { }
3328   \c_zero \fi:
3329   \_msg_kernel_expandable_error:nnn
3330   { kernel } { unknown-comparison } {#1}
3331   \prg_return_false:
3332 }

```

(End definition for `__prg_compare_error:` and `__prg_compare_error:NNw`.)

```

\int_compare_p:n
\int_compare:nTF
\_int_compare:w
\_int_compare:Nw
\_int_compare:NNw
\_int_compare:nnN
\_int_compare_end=:NNw
\_int_compare=:NNw
\_int_compare<:NNw
\_int_compare>:NNw
\_int_compare=:NNw
\_int_compare!=:NNw
\_int_compare<=:NNw
\_int_compare>=:NNw

```

Comparison tests using a simple syntax where only one set of braces is required, additional operators such as `!=` and `>=` are supported, and multiple comparisons can be performed at once, for instance `0 < 5 <= 1`. The idea is to loop through the argument, finding one operand at a time, and comparing it to the previous one. The looping auxiliary `__int_compare:Nw` reads one *operand* and one *comparison* symbol, and leaves roughly

```

\operand\prg_return_false:\fi:
\reverse_if:N\if_int_compare:w\operand\comparison
\_int_compare:Nw

```

in the input stream. Each call to this auxiliary provides the second operand of the last call's `\if_int_compare:w`. If one of the *comparisons* is `false`, the `true` branch of the TeX conditional is taken (because of `\reverse_if:N`), immediately returning `false` as the result of the test. There is no TeX conditional waiting the first operand, so we add an `\if_false:` and expand by hand with `__int_value:w`, thus skipping `\prg_return_false:` on the first iteration.

Before starting the loop, the first step is to make sure that there is at least one relation symbol. We first let TeX evaluate this left hand side of the (in)equality using `__int_eval:w`. Since the relation symbols `<`, `>`, `=` and `!` are not allowed in integer expressions, they will terminate it. If the argument contains no relation symbol, `__prg_compare_error:` is expanded, inserting `=` and itself after an error. In all cases, `__int_compare:w` receives as its argument an integer, a relation symbol, and some more tokens. We then setup the loop, which will be ended by the two odd-looking items `e` and `{=nd_}`, with a trailing `\q_stop` used to grab the entire argument when necessary.

```

3333 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
3334 {
3335   \exp_after:wN \__int_compare:w
3336   \int_use:N \__int_eval:w #1 \__prg_compare_error:
3337 }

```

```

3338 \cs_new:Npn \__int_compare:w #1 \__prg_compare_error:
3339 {
3340   \exp_after:wN \if_false: \__int_value:w
3341   \__int_compare:Nw #1 e { = nd_ } \q_stop
3342 }

```

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. 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.

```

3343 \cs_new:Npn \__int_compare:Nw #1#2 \q_stop
3344 {
3345   \exp_after:wN \__int_compare:NNw
3346   \__int_to_roman:w - 0 #2 \q_mark
3347   #1#2 \q_stop
3348 }
3349 \cs_new:Npn \__int_compare:NNw #1#2#3 \q_mark
3350 {
3351   \etex_unexpanded:D
3352   \use:c { __int_compare_ #1 \if_meaning:w = #2 = \fi: :NNw }
3353   \__prg_compare_error:Nw #1
3354 }

```

When the last *operand* 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 *operand*, and `#3` is one of `<`, `=`, or `>`. As announced earlier, we leave the *operand* 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 *operand* `#2` and the comparison `#3`, and call `__int_compare:Nw` to look for additional operands, after evaluating the following expression.

```

3355 \cs_new:cpn { __int_compare_end=:NNw } #1#2#3 e #4 \q_stop
3356 {
3357   {#3} \exp_stop_f:
3358   \prg_return_false: \else: \prg_return_true: \fi:
3359 }
3360 \cs_new:Npn \__int_compare:nnN #1#2#3
3361 {
3362   {#2} \exp_stop_f:
3363   \prg_return_false: \exp_after:wN \use_none_delimit_by_q_stop:w
3364   \fi:
3365   #1 #2 #3 \exp_after:wN \__int_compare:Nw \__int_value:w \__int_eval:w

```


3366 }

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.

```

3367 \cs_new:cpn { __int_compare=:NNw } #1#2#3 =
3368 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
3369 \cs_new:cpn { __int_compare<:NNw } #1#2#3 <
3370 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
3371 \cs_new:cpn { __int_compare>:NNw } #1#2#3 >
3372 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
3373 \cs_new:cpn { __int_compare=:NNw } #1#2#3 ==
3374 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
3375 \cs_new:cpn { __int_compare!=:NNw } #1#2#3 !=
3376 { \__int_compare:nnN { \if_int_compare:w } {#3} = }
3377 \cs_new:cpn { __int_compare<=:NNw } #1#2#3 <=
3378 { \__int_compare:nnN { \if_int_compare:w } {#3} > }
3379 \cs_new:cpn { __int_compare>=:NNw } #1#2#3 >=
3380 { \__int_compare:nnN { \if_int_compare:w } {#3} < }

```

(End definition for `\int_compare:n`. These functions are documented on page 66.)

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

More efficient but less natural in typing.

```

3381 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
3382 {
3383   \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
3384   \prg_return_true:
3385   \else:
3386   \prg_return_false:
3387   \fi:
3388 }

```

(End definition for `\int_compare:nNn`. These functions are documented on page 65.)

`\int_case:nn`
`\int_case:nnTF`
`__int_case:nnTF`
`__int_case:nw`
`__int_case_end:nw`

For integer 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`.

```

3389 \cs_new:Npn \int_case:nnTF #1
3390 {
3391   \tex_romannumeral:D
3392   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
3393 }
3394 \cs_new:Npn \int_case:nnT #1#2#3
3395 {
3396   \tex_romannumeral:D
3397   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
3398 }
3399 \cs_new:Npn \int_case:nnF #1#2
3400 {
3401   \tex_romannumeral:D
3402   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
3403 }

```

```

3404 \cs_new:Npn \int_case:nn #1#2
3405 {
3406   \tex_romannumeral:D
3407   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
3408 }
3409 \cs_new:Npn \__int_case:nnTF #1#2#3#4
3410 { \__int_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
3411 \cs_new:Npn \__int_case:nw #1#2#3
3412 {
3413   \int_compare:nNnTF {#1} = {#2}
3414   { \__int_case_end:nw {#3} }
3415   { \__int_case:nw {#1} }
3416 }
3417 \cs_new_eq:NN \__int_case_end:nw \__prg_case_end:nw

```

(End definition for `\int_case:nn`. This function is documented on page 67.)

`\int_if_odd_p:n` A predicate function.

```

\int_if_odd:nTF
\int_if_even_p:n
\int_if_even:nTF
3418 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
3419 {
3420   \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
3421   \prg_return_true:
3422   \else:
3423   \prg_return_false:
3424   \fi:
3425 }
3426 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
3427 {
3428   \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
3429   \prg_return_false:
3430   \else:
3431   \prg_return_true:
3432   \fi:
3433 }

```

(End definition for `\int_if_odd:n`. These functions are documented on page 67.)

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
3434 \cs_new:Npn \int_while_do:nn #1#2
3435 {
3436   \int_compare:nT {#1}
3437   {
3438     #2
3439     \int_while_do:nn {#1} {#2}
3440   }
3441 }
3442 \cs_new:Npn \int_until_do:nn #1#2
3443 {

```

```

3444     \int_compare:nF {#1}
3445     {
3446         #2
3447         \int_until_do:nn {#1} {#2}
3448     }
3449 }
3450 \cs_new:Npn \int_do_while:nn #1#2
3451 {
3452     #2
3453     \int_compare:nT {#1}
3454     { \int_do_while:nn {#1} {#2} }
3455 }
3456 \cs_new:Npn \int_do_until:nn #1#2
3457 {
3458     #2
3459     \int_compare:nF {#1}
3460     { \int_do_until:nn {#1} {#2} }
3461 }

```

(End definition for `\int_while_do:nn`. This function is documented on page 68.)

`\int_while_do:nNnn`
`\int_until_do:nNnn`
`\int_do_while:nNnn`
`\int_do_until:nNnn`

As above but not using the more natural syntax.

```

3462 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
3463 {
3464     \int_compare:nNnT {#1} #2 {#3}
3465     {
3466         #4
3467         \int_while_do:nNnn {#1} #2 {#3} {#4}
3468     }
3469 }
3470 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
3471 {
3472     \int_compare:nNnF {#1} #2 {#3}
3473     {
3474         #4
3475         \int_until_do:nNnn {#1} #2 {#3} {#4}
3476     }
3477 }
3478 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
3479 {
3480     #4
3481     \int_compare:nNnT {#1} #2 {#3}
3482     { \int_do_while:nNnn {#1} #2 {#3} {#4} }
3483 }
3484 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
3485 {
3486     #4
3487     \int_compare:nNnF {#1} #2 {#3}
3488     { \int_do_until:nNnn {#1} #2 {#3} {#4} }
3489 }

```

(End definition for `\int_while_do:nNnn`. This function is documented on page 67.)

8.7 Integer step functions

```
\int_step_function:nnnN
  \__int_step:NnnnN
```

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).

```
3490 \cs_new:Npn \int_step_function:nnnN #1#2#3#4
3491 {
3492   \int_compare:nNnTF {#2} > \c_zero
3493   { \exp_args:Nnf \__int_step:NnnnN > }
3494   {
3495     \int_compare:nNnTF {#2} = \c_zero
3496     {
3497       \__msg_kernel_expandable_error:nnn { kernel } { zero-step } {#4}
3498       \use_none:nnnn
3499     }
3500     { \exp_args:Nnf \__int_step:NnnnN < }
3501   }
3502   { \int_eval:n {#1} } {#2} {#3} #4
3503 }
3504 \cs_new:Npn \__int_step:NnnnN #1#2#3#4#5
3505 {
3506   \int_compare:nNf {#2} #1 {#4}
3507   {
3508     #5 {#2}
3509     \exp_args:Nnf \__int_step:NnnnN
3510     #1 { \int_eval:n { #2 + #3 } } {#3} {#4} #5
3511   }
3512 }
```

(End definition for `\int_step_function:nnnN`. This function is documented on page 69.)

```
\int_step_inline:nnnn
\int_step_variable:nnnN
  \__int_step:NNnnnn
```

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\int_step_function:nnnN`. We put a `__prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so no breaking function will recognize this break point as its own.

```
3513 \cs_new_protected_nopar:Npn \int_step_inline:nnnn
3514 {
3515   \int_gincr:N \g__prg_map_int
3516   \exp_args:NNc \__int_step:NNnnnn
3517   \cs_gset_nopar:Npn
3518   { __prg_map_ \int_use:N \g__prg_map_int :w }
3519 }
3520 \cs_new_protected:Npn \int_step_variable:nnnN #1#2#3#4#5
3521 {
3522   \int_gincr:N \g__prg_map_int
3523   \exp_args:NNc \__int_step:NNnnnn
3524   \cs_gset_nopar:Npx
```

```

3525     { __prg_map_ \int_use:N \g__prg_map_int :w }
3526     {#1}{#2}{#3}
3527     {
3528         \tl_set:Nn \exp_not:N #4 {##1}
3529         \exp_not:n {#5}
3530     }
3531 }
3532 \cs_new_protected:Npn \__int_step:NNnnn #1#2#3#4#5#6
3533 {
3534     #1 #2 ##1 {#6}
3535     \int_step_function:nnnN {#3} {#4} {#5} #2
3536     \__prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__prg_map_int }
3537 }

```

(End definition for `\int_step_inline:nnnn`. This function is documented on page 69.)

8.8 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```

3538 \cs_new:Npn \int_to_arabic:n #1 { \int_eval:n {#1} }

```

(End definition for `\int_to_arabic:n`. This function is documented on page 69.)

`\int_to_symbols:nnn`
`__int_to_symbols:nnnn`

For conversion of integers to arbitrary symbols the method is in general as follows. The input number (#1) is compared to the total number of symbols available at each place (#2). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an f-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

```

3539 \cs_new:Npn \int_to_symbols:nnn #1#2#3
3540 {
3541     \int_compare:nNnTF {#1} > {#2}
3542     {
3543         \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
3544         {
3545             \int_case:nn
3546             { 1 + \int_mod:nn { #1 - 1 } {#2} }
3547             {#3}
3548         }
3549         {#1} {#2} {#3}
3550     }
3551     { \int_case:nn {#1} {#3} }
3552 }
3553 \cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
3554 {
3555     \exp_args:Nf \int_to_symbols:nnn
3556     { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
3557     #1
3558 }

```

(End definition for `\int_to_symbols:nnn`. This function is documented on page 70.)

`\int_to_alph:n` These both use the above function with input functions that make sense for the alphabet
`\int_to_Alph:n` in English.

```

3559 \cs_new:Npn \int_to_alph:n #1
3560 {
3561   \int_to_symbols:nnn {#1} { 26 }
3562   {
3563     { 1 } { a }
3564     { 2 } { b }
3565     { 3 } { c }
3566     { 4 } { d }
3567     { 5 } { e }
3568     { 6 } { f }
3569     { 7 } { g }
3570     { 8 } { h }
3571     { 9 } { i }
3572     { 10 } { j }
3573     { 11 } { k }
3574     { 12 } { l }
3575     { 13 } { m }
3576     { 14 } { n }
3577     { 15 } { o }
3578     { 16 } { p }
3579     { 17 } { q }
3580     { 18 } { r }
3581     { 19 } { s }
3582     { 20 } { t }
3583     { 21 } { u }
3584     { 22 } { v }
3585     { 23 } { w }
3586     { 24 } { x }
3587     { 25 } { y }
3588     { 26 } { z }
3589   }
3590 }
3591 \cs_new:Npn \int_to_Alph:n #1
3592 {
3593   \int_to_symbols:nnn {#1} { 26 }
3594   {
3595     { 1 } { A }
3596     { 2 } { B }
3597     { 3 } { C }
3598     { 4 } { D }
3599     { 5 } { E }
3600     { 6 } { F }
3601     { 7 } { G }
3602     { 8 } { H }
3603     { 9 } { I }
3604     { 10 } { J }

```

```

3605      { 11 } { K }
3606      { 12 } { L }
3607      { 13 } { M }
3608      { 14 } { N }
3609      { 15 } { O }
3610      { 16 } { P }
3611      { 17 } { Q }
3612      { 18 } { R }
3613      { 19 } { S }
3614      { 20 } { T }
3615      { 21 } { U }
3616      { 22 } { V }
3617      { 23 } { W }
3618      { 24 } { X }
3619      { 25 } { Y }
3620      { 26 } { Z }
3621    }
3622  }

```

(End definition for `\int_to_alph:n` and `\int_to_Alph:n`. These functions are documented on page 70.)

`\int_to_base:nn` 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.

```

3623 \cs_new:Npn \int_to_base:nn #1
3624 { \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } }
3625 \cs_new:Npn \__int_to_base:nn #1#2
3626 {
3627   \int_compare:nNnTF {#1} < \c_zero
3628   { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
3629   { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
3630 }

```

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.

```

3631 \cs_new:Npn \__int_to_base:nnN #1#2#3
3632 {
3633   \int_compare:nNnTF {#1} < {#2}
3634   { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
3635   {
3636     \exp_args:Nf \__int_to_base:nnnN
3637     { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
3638     {#1}
3639     {#2}
3640     #3
3641   }

```

```

3642 }
3643 \cs_new:Npn \__int_to_base:nnnN #1#2#3#4
3644 {
3645   \exp_args:Nf \__int_to_base:nnN
3646     { \int_div_truncate:nn {#2} {#3} }
3647     {#3}
3648     #4
3649     #1
3650 }

```

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:`.

```

3651 \cs_new:Npn \__int_to_letter:n #1
3652 {
3653   \exp_after:wN \exp_after:wN
3654   \if_case:w \__int_eval:w #1 - \c_ten \__int_eval_end:
3655     A
3656     \or: B
3657     \or: C
3658     \or: D
3659     \or: E
3660     \or: F
3661     \or: G
3662     \or: H
3663     \or: I
3664     \or: J
3665     \or: K
3666     \or: L
3667     \or: M
3668     \or: N
3669     \or: O
3670     \or: P
3671     \or: Q
3672     \or: R
3673     \or: S
3674     \or: T
3675     \or: U
3676     \or: V
3677     \or: W
3678     \or: X
3679     \or: Y
3680     \or: Z
3681     \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
3682     \fi:
3683 }

```

(End definition for `\int_to_base:nn`. This function is documented on page 71.)

`\int_to_binary:n`
`\int_to_hexadecimal:n`
`\int_to_octal:n`

Wrappers around the generic function.

```
3684 \cs_new:Npn \int_to_binary:n #1
3685 { \int_to_base:nn {#1} { 2 } }
3686 \cs_new:Npn \int_to_hexadecimal:n #1
3687 { \int_to_base:nn {#1} { 16 } }
3688 \cs_new:Npn \int_to_octal:n #1
3689 { \int_to_base:nn {#1} { 8 } }
```

(End definition for `\int_to_binary:n`, `\int_to_hexadecimal:n`, and `\int_to_octal:n`. These functions are documented on page 71.)

`\int_to_roman:n`

`\int_to_Roman:n`

`__int_to_roman:N`

`__int_to_roman:N`

`__int_to_roman_i:w`

`__int_to_roman_v:w`

`__int_to_roman_x:w`

`__int_to_roman_l:w`

`__int_to_roman_c:w`

`__int_to_roman_d:w`

`__int_to_roman_m:w`

`__int_to_roman_Q:w`

`__int_to_Roman_i:w`

`__int_to_Roman_v:w`

`__int_to_Roman_x:w`

`__int_to_Roman_l:w`

`__int_to_Roman_c:w`

`__int_to_Roman_d:w`

`__int_to_Roman_m:w`

`__int_to_Roman_Q:w`

The `__int_to_roman:w` primitive creates tokens of category code 12 (other). Usually, what is actually wanted is letters. The approach here is to convert the output of the primitive into letters using appropriate control sequence names. That keeps everything expandable. The loop will be terminated by the conversion of the Q.

```
3690 \cs_new:Npn \int_to_roman:n #1
3691 {
3692   \exp_after:wN \__int_to_roman:N
3693   \__int_to_roman:w \int_eval:n {#1} Q
3694 }
3695 \cs_new:Npn \__int_to_roman:N #1
3696 {
3697   \use:c { __int_to_roman_ #1 :w }
3698   \__int_to_roman:N
3699 }
3700 \cs_new:Npn \int_to_Roman:n #1
3701 {
3702   \exp_after:wN \__int_to_Roman_aux:N
3703   \__int_to_roman:w \int_eval:n {#1} Q
3704 }
3705 \cs_new:Npn \__int_to_Roman_aux:N #1
3706 {
3707   \use:c { __int_to_Roman_ #1 :w }
3708   \__int_to_Roman_aux:N
3709 }
3710 \cs_new_nopar:Npn \__int_to_roman_i:w { i }
3711 \cs_new_nopar:Npn \__int_to_roman_v:w { v }
3712 \cs_new_nopar:Npn \__int_to_roman_x:w { x }
3713 \cs_new_nopar:Npn \__int_to_roman_l:w { l }
3714 \cs_new_nopar:Npn \__int_to_roman_c:w { c }
3715 \cs_new_nopar:Npn \__int_to_roman_d:w { d }
3716 \cs_new_nopar:Npn \__int_to_roman_m:w { m }
3717 \cs_new_nopar:Npn \__int_to_roman_Q:w #1 { }
3718 \cs_new_nopar:Npn \__int_to_Roman_i:w { I }
3719 \cs_new_nopar:Npn \__int_to_Roman_v:w { V }
3720 \cs_new_nopar:Npn \__int_to_Roman_x:w { X }
3721 \cs_new_nopar:Npn \__int_to_Roman_l:w { L }
3722 \cs_new_nopar:Npn \__int_to_Roman_c:w { C }
3723 \cs_new_nopar:Npn \__int_to_Roman_d:w { D }
3724 \cs_new_nopar:Npn \__int_to_Roman_m:w { M }
```

```

3725 \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
71.)

```

8.9 Converting from other formats to integers

`__int_get_sign:n` Finding a number and its sign requires dealing with an arbitrary list of + and - symbols.
`__int_get_digits:n` This is done by working through token by token until there is something else at the start
`_int_get_sign_and_digits:nNNN` of the input. The sign of the input is tracked by the first Boolean used by the auxiliary
`_int_get_sign_and_digits:oNNN` function.

```

3726 \cs_new:Npn \__int_get_sign:n #1
3727 {
3728   \_int_get_sign_and_digits:nNNN {#1}
3729   \c_true_bool \c_true_bool \c_false_bool
3730 }
3731 \cs_new:Npn \__int_get_digits:n #1
3732 {
3733   \_int_get_sign_and_digits:nNNN {#1}
3734   \c_true_bool \c_false_bool \c_true_bool
3735 }

```

The auxiliary loops through, finding sign tokens and removing them. The sign itself is carried through as a flag.

```

3736 \cs_new:Npn \__int_get_sign_and_digits:nNNN #1#2#3#4
3737 {
3738   \exp_args:Nf \tl_if_head_eq_charcode:nNTF {#1} -
3739   {
3740     \bool_if:NTF #2
3741     {
3742       \_int_get_sign_and_digits:oNNN
3743       { \use_none:n #1 } \c_false_bool #3#4
3744     }
3745     {
3746       \_int_get_sign_and_digits:oNNN
3747       { \use_none:n #1 } \c_true_bool #3#4
3748     }
3749   }
3750   {
3751     \exp_args:Nf \tl_if_head_eq_charcode:nNTF {#1} +
3752     { \_int_get_sign_and_digits:oNNN { \use_none:n #1 } #2#3#4 }
3753     {
3754       \bool_if:NT #3 { \bool_if:NF #2 - }
3755       \bool_if:NT #4 {#1}
3756     }
3757   }
3758 }
3759 \cs_generate_variant:Nn \__int_get_sign_and_digits:nNNN { o }
(End definition for \__int_get_sign:n. This function is documented on page ??.)

```

```

\int_from_alph:n
\__int_from_alph:n
\__int_from_alph:nN
\__int_from_alph:N

```

The aim here is to iterate through the input, converting one letter at a time to a number. The same approach is also used for base conversion, but this needs a different final auxiliary.

```

3760 \cs_new:Npn \int_from_alph:n #1
3761 {
3762   \int_eval:n
3763   {
3764     \__int_get_sign:n {#1}
3765     \exp_args:Nf \__int_from_alph:n { \__int_get_digits:n {#1} }
3766   }
3767 }
3768 \cs_new:Npn \__int_from_alph:n #1
3769 { \__int_from_alph:nN { 0 } #1 \q_nil }
3770 \cs_new:Npn \__int_from_alph:nN #1#2
3771 {
3772   \quark_if_nil:NTF #2
3773   {#1}
3774   {
3775     \exp_args:Nf \__int_from_alph:nN
3776     { \int_eval:n { #1 * 26 + \__int_from_alph:N #2 } }
3777   }
3778 }
3779 \cs_new:Npn \__int_from_alph:N #1
3780 { \int_eval:n { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } } }

```

(End definition for \int_from_alph:n. This function is documented on page 71.)

```

\int_from_base:nn
\__int_from_base:nn
\__int_from_base:nnN
\__int_from_base:N

```

Conversion to base ten means stripping off the sign then iterating through the input one token at a time. The total number is then added up as the code loops.

```

3781 \cs_new:Npn \int_from_base:nn #1#2
3782 {
3783   \int_eval:n
3784   {
3785     \__int_get_sign:n {#1}
3786     \exp_args:Nf \__int_from_base:nn
3787     { \__int_get_digits:n {#1} } {#2}
3788   }
3789 }
3790 \cs_new:Npn \__int_from_base:nn #1#2
3791 { \__int_from_base:nnN { 0 } { #2 } #1 \q_nil }
3792 \cs_new:Npn \__int_from_base:nnN #1#2#3
3793 {
3794   \quark_if_nil:NTF #3
3795   {#1}
3796   {
3797     \exp_args:Nf \__int_from_base:nnN
3798     { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
3799     {#2}
3800   }
3801 }

```

The conversion here will take lower or upper case letters and turn them into the appropriate number, hence the two-part nature of the function.

```

3802 \cs_new:Npn \__int_from_base:N #1
3803 {
3804   \int_compare:nNnTF { '#1 } < { 58 }
3805     {#1}
3806     {
3807       \int_eval:n
3808         { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
3809     }
3810 }

```

(End definition for `\int_from_base:nn`. This function is documented on page 72.)

```

\int_from_binary:n
\int_from_hexadecimal:n
\int_from_octal:n

```

Wrappers around the generic function.

```

3811 \cs_new:Npn \int_from_binary:n #1
3812 { \int_from_base:nn {#1} \c_two }
3813 \cs_new:Npn \int_from_hexadecimal:n #1
3814 { \int_from_base:nn {#1} \c_sixteen }
3815 \cs_new:Npn \int_from_octal:n #1
3816 { \int_from_base:nn {#1} \c_eight }

```

(End definition for `\int_from_binary:n`, `\int_from_hexadecimal:n`, and `\int_from_octal:n`. These functions are documented on page 72.)

```

\c__int_from_roman_i_int
\c__int_from_roman_v_int
\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
\c__int_from_roman_I_int
\c__int_from_roman_V_int
\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

```

Constants used to convert from Roman numerals to integers.

```

3817 \int_const:cn { c__int_from_roman_i_int } { 1 }
3818 \int_const:cn { c__int_from_roman_v_int } { 5 }
3819 \int_const:cn { c__int_from_roman_x_int } { 10 }
3820 \int_const:cn { c__int_from_roman_l_int } { 50 }
3821 \int_const:cn { c__int_from_roman_c_int } { 100 }
3822 \int_const:cn { c__int_from_roman_d_int } { 500 }
3823 \int_const:cn { c__int_from_roman_m_int } { 1000 }
3824 \int_const:cn { c__int_from_roman_I_int } { 1 }
3825 \int_const:cn { c__int_from_roman_V_int } { 5 }
3826 \int_const:cn { c__int_from_roman_X_int } { 10 }
3827 \int_const:cn { c__int_from_roman_L_int } { 50 }
3828 \int_const:cn { c__int_from_roman_C_int } { 100 }
3829 \int_const:cn { c__int_from_roman_D_int } { 500 }
3830 \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 .

```

\__int_from_roman:NN
\__int_from_roman_end:w
\__int_from_roman_clean_up:w

```

```

3831 \cs_new:Npn \int_from_roman:n #1
3832 {
3833   \tl_if_blank:nF {#1}
3834   {
3835     \exp_after:wN \__int_from_roman_end:w
3836     \__int_value:w \__int_eval:w

```

```

3837         \_int_from_roman:NN #1 Q \q_stop
3838     }
3839 }
3840 \cs_new:Npn \_int_from_roman:NN #1#2
3841 {
3842     \str_if_eq:nnTF {#1} { Q }
3843     {#1#2}
3844     {
3845         \str_if_eq:nnTF {#2} { Q }
3846         {
3847             \int_if_exist:cF { c__int_from_roman_ #1 _int }
3848             { \_int_from_roman_clean_up:w }
3849             +
3850             \use:c { c__int_from_roman_ #1 _int }
3851             #2
3852         }
3853         {
3854             \int_if_exist:cF { c__int_from_roman_ #1 _int }
3855             { \_int_from_roman_clean_up:w }
3856             \int_if_exist:cF { c__int_from_roman_ #2 _int }
3857             { \_int_from_roman_clean_up:w }
3858             \int_compare:nNnTF
3859             { \use:c { c__int_from_roman_ #1 _int } }
3860             <
3861             { \use:c { c__int_from_roman_ #2 _int } }
3862             {
3863                 + \use:c { c__int_from_roman_ #2 _int }
3864                 - \use:c { c__int_from_roman_ #1 _int }
3865                 \_int_from_roman:NN
3866             }
3867             {
3868                 + \use:c { c__int_from_roman_ #1 _int }
3869                 \_int_from_roman:NN #2
3870             }
3871         }
3872     }
3873 }
3874 \cs_new:Npn \_int_from_roman_end:w #1 Q #2 \q_stop
3875 { \tl_if_empty:nTF {#2} {#1} {#2} }
3876 \cs_new:Npn \_int_from_roman_clean_up:w #1 Q { + 0 Q -1 }

```

(End definition for `\int_from_roman:n`. This function is documented on page 72.)

8.10 Viewing integer

`\int_show:N`
`\int_show:c`

```

3877 \cs_new_eq:NN \int_show:N \__kernel_register_show:N
3878 \cs_new_eq:NN \int_show:c \__kernel_register_show:c

```

(End definition for `\int_show:N` and `\int_show:c`. These functions are documented on page ??.)

`\int_show:n` We don't use the TeX primitive `\showthe` to show integer expressions: this gives a more unified output, since the closing brace is read by the integer expression in all cases.

```
3879 \cs_new_protected:Npn \int_show:n #1
3880 { \etex_showtokens:D \exp_after:wN { \int_use:N \__int_eval:w #1 } }
(End definition for \int_show:n. This function is documented on page 72.)
```

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 73.)

`\c_zero` Again, one in `l3basics` for obvious reasons.
(End definition for `\c_zero`. This variable is documented on page 73.)

`\c_six` Once again, in `l3basics`.
`\c_seven` (End definition for `\c_six` and `\c_seven`. These functions are documented on page 73.)

`\c_twelve` Low-number values not previously defined.
`\c_one`
`\c_sixteen`
`\c_two`

```
3881 \int_const:Nn \c_one { 1 }
3882 \int_const:Nn \c_two { 2 }
3883 \int_const:Nn \c_three { 3 }
3884 \int_const:Nn \c_four { 4 }
3885 \int_const:Nn \c_five { 5 }
3886 \int_const:Nn \c_eight { 8 }
3887 \int_const:Nn \c_nine { 9 }
3888 \int_const:Nn \c_ten { 10 }
3889 \int_const:Nn \c_eleven { 11 }
3890 \int_const:Nn \c_thirteen { 13 }
3891 \int_const:Nn \c_fourteen { 14 }
3892 \int_const:Nn \c_fifteen { 15 }
(End definition for \c_one and others. These variables are documented on page 73.)
```

`\c_thirty_two` One middling value.

```
3893 \int_const:Nn \c_thirty_two { 32 }
(End definition for \c_thirty_two. This variable is documented on page 73.)
```

`\c_two_hundred_fifty_five` Two classic mid-range integer constants.

```
3894 \int_const:Nn \c_two_hundred_fifty_five { 255 }
3895 \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 73.)
```

`\c_one_hundred` Simple runs of powers of ten.

```
3896 \int_const:Nn \c_one_hundred { 100 }
3897 \int_const:Nn \c_one_thousand { 1000 }
3898 \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 73.)
```

`\c_max_int` The largest number allowed is $2^{31} - 1$

```
3899 \int_const:Nn \c_max_int { 2 147 483 647 }
```

(End definition for `\c_max_int`. This variable is documented on page 73.)

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` 3900 `\int_new:N \l_tmpa_int`

`\g_tmpa_int` 3901 `\int_new:N \l_tmpb_int`

`\g_tmpb_int` 3902 `\int_new:N \g_tmpa_int`

3903 `\int_new:N \g_tmpb_int`

(End definition for `\l_tmpa_int` and `\l_tmpb_int`. These functions are documented on page 73.)

8.13 Deprecated functions

`\int_case:nnn` Deprecated 2013-07-15.

```
3904 \cs_new_eq:NN \int_case:nnn \int_case:nnF
```

(End definition for `\int_case:nnn`. This function is documented on page ??.)

```
3905 \</initex | package>
```

9 l3skip implementation

```
3906 \<*initex | package>
3907 \<@@=dim>
3908 \<*package>
3909 \ProvidesExplPackage
3910 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
3911 \__expl_package_check:
3912 \</package>
```

9.1 Length primitives renamed

`\if_dim:w` Primitives renamed.

`__dim_eval:w` 3913 `\cs_new_eq:NN \if_dim:w \tex_ifdim:D`

`__dim_eval_end:` 3914 `\cs_new_eq:NN __dim_eval:w \etex_dimexpr:D`

3915 `\cs_new_eq:NN __dim_eval_end: \tex_relax:D`

(End definition for `\if_dim:w`. This function is documented on page 89.)

9.2 Creating and initialising dim variables

\dim_new:N Allocating $\langle dim \rangle$ registers ...

```
\dim_new:c 3916 <*package>
3917 \cs_new_protected:Npn \dim_new:N #1
3918 {
3919     \__chk_if_free_cs:N #1
3920     \newdimen #1
3921 }
3922 </package>
3923 \cs_generate_variant:Nn \dim_new:N { c }
```

(End definition for \dim_new:N and \dim_new:c. These functions are documented on page ??.)

\dim_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\dim_const:cn 3924 \cs_new_protected:Npn \dim_const:Nn #1
3925 {
3926     \dim_new:N #1
3927     \dim_gset:Nn #1
3928 }
3929 \cs_generate_variant:Nn \dim_const:Nn { c }
```

(End definition for \dim_const:Nn and \dim_const:cn. These functions are documented on page ??.)

\dim_zero:N Reset the register to zero.

```
\dim_zero:c 3930 \cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_dim }
\dim_gzero:N 3931 \cs_new_protected:Npn \dim_gzero:N { \tex_global:D \dim_zero:N }
\dim_gzero:c 3932 \cs_generate_variant:Nn \dim_zero:N { c }
3933 \cs_generate_variant:Nn \dim_gzero:N { c }
```

(End definition for \dim_zero:N and \dim_zero:c. These functions are documented on page ??.)

\dim_zero_new:N Create a register if needed, otherwise clear it.

```
\dim_zero_new:c 3934 \cs_new_protected:Npn \dim_zero_new:N #1
\dim_gzero_new:N 3935 { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
\dim_gzero_new:c 3936 \cs_new_protected:Npn \dim_gzero_new:N #1
3937 { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
3938 \cs_generate_variant:Nn \dim_zero_new:N { c }
3939 \cs_generate_variant:Nn \dim_gzero_new:N { c }
```

(End definition for \dim_zero_new:N and others. These functions are documented on page ??.)

\dim_if_exist_p:N Copies of the cs functions defined in l3basics.

```
\dim_if_exist_p:c 3940 \prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N { TF , T , F , p }
\dim_if_exist:N $\underline{TF}$  3941 \prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

\dim_if_exist:c \underline{TF} (End definition for \dim_if_exist:N and \dim_if_exist:c. These functions are documented on page ??.)

9.3 Setting dim variables

\dim_set:Nn Setting dimensions is easy enough.

```
\dim_set:cn      3942 \cs_new_protected:Npn \dim_set:Nn #1#2
\dim_gset:Nn      3943 { #1 ~ \__dim_eval:w #2 \__dim_eval_end: }
\dim_gset:cn      3944 \cs_new_protected:Npn \dim_gset:Nn { \tex_global:D \dim_set:Nn }
                  3945 \cs_generate_variant:Nn \dim_set:Nn { c }
                  3946 \cs_generate_variant:Nn \dim_gset:Nn { c }
```

(End definition for \dim_set:Nn and \dim_set:cn. These functions are documented on page ??.)

\dim_set_eq:NN All straightforward.

```
\dim_set_eq:cN      3947 \cs_new_protected:Npn \dim_set_eq:NN #1#2 { #1 = #2 }
\dim_set_eq:Nc      3948 \cs_generate_variant:Nn \dim_set_eq:NN { c }
\dim_set_eq:cc      3949 \cs_generate_variant:Nn \dim_set_eq:NN { Nc , cc }
\dim_gset_eq:NN      3950 \cs_new_protected:Npn \dim_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\dim_gset_eq:cN      3951 \cs_generate_variant:Nn \dim_gset_eq:NN { c }
\dim_gset_eq:Nc      3952 \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 ??.)
```

\dim_add:Nn Using by here deals with the (incorrect) case \dimen123.

```
\dim_add:cn      3953 \cs_new_protected:Npn \dim_add:Nn #1#2
\dim_gadd:Nn      3954 { \tex_advance:D #1 by \__dim_eval:w #2 \__dim_eval_end: }
\dim_gadd:cn      3955 \cs_new_protected:Npn \dim_gadd:Nn { \tex_global:D \dim_add:Nn }
\dim_sub:Nn      3956 \cs_generate_variant:Nn \dim_add:Nn { c }
\dim_sub:cn      3957 \cs_generate_variant:Nn \dim_gadd:Nn { c }
\dim_gsub:Nn      3958 \cs_new_protected:Npn \dim_sub:Nn #1#2
\dim_gsub:cn      3959 { \tex_advance:D #1 by - \__dim_eval:w #2 \__dim_eval_end: }
                  3960 \cs_new_protected:Npn \dim_gsub:Nn { \tex_global:D \dim_sub:Nn }
                  3961 \cs_generate_variant:Nn \dim_sub:Nn { c }
                  3962 \cs_generate_variant:Nn \dim_gsub:Nn { c }
```

(End definition for \dim_add:Nn and \dim_add:cn. These functions are documented on page ??.)

9.4 Utilities for dimension calculations

\dim_abs:n Functions for min, max, and absolute value with only one evaluation. The absolute value is evaluated by removing a leading - if present.

```
\__dim_abs:N      3963 \cs_new:Npn \dim_abs:n #1
\dim_max:nn      3964 {
\dim_min:nn      3965   \exp_after:wN \__dim_abs:N
\__dim_maxmin:wwN 3966   \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
                  3967 }
                  3968 \cs_new:Npn \__dim_abs:N #1
                  3969 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
                  3970 \cs_set:Npn \dim_max:nn #1#2
                  3971 {
                  3972   \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
                  3973   \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
                  3974   \dim_use:N \__dim_eval:w #2 ;
```

```

3975     >
3976     \__dim_eval_end:
3977   }
3978   \cs_set:Npn \dim_min:nn #1#2
3979   {
3980     \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
3981     \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
3982     \dim_use:N \__dim_eval:w #2 ;
3983     <
3984     \__dim_eval_end:
3985   }
3986   \cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
3987   {
3988     \if_dim:w #1 #3 #2 ~
3989     #1
3990     \else:
3991     #2
3992     \fi:
3993   }

```

(End definition for `\dim_abs:n`. This function is documented on page 77.)

`\dim_ratio:nn` With dimension expressions, something like `10 pt * (5 pt / 10 pt)` will not work. `__dim_ratio:n` Instead, the ratio part needs to be converted to an integer expression. Using `__int_value:w` forces everything into sp, avoiding any decimal parts.

```

3994   \cs_new:Npn \dim_ratio:nn #1#2
3995   { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
3996   \cs_new:Npn \__dim_ratio:n #1
3997   { \__int_value:w \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_ratio:nn`. This function is documented on page 78.)

9.5 Dimension expression conditionals

`\dim_compare_p:nNn` Simple comparison.

```

\dim_compare:nNnTF
3998   \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
3999   {
4000     \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
4001     \prg_return_true: \else: \prg_return_false: \fi:
4002   }

```

(End definition for `\dim_compare:nNn`. These functions are documented on page 78.)

`\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:w
\__dim_compare:wNN
\__dim_compare_=:w
\__dim_compare_!=:w
\__dim_compare_<:w
\__dim_compare_>:w
4003   \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }

```

```

4004 {
4005   \exp_after:wN \__dim_compare:w
4006   \dim_use:N \__dim_eval:w #1 \__prg_compare_error:
4007 }
4008 \cs_new:Npn \__dim_compare:w #1 \__prg_compare_error:
4009 {
4010   \exp_after:wN \if_false: \tex_romannumeral:D -'0
4011   \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \q_stop
4012 }
4013 \exp_args:Nno \use:nn
4014 { \cs_new:Npn \__dim_compare:wNN #1 }
4015 { \tl_to_str:n {pt} }
4016 #2#3
4017 {
4018   \if_meaning:w = #3
4019   \use:c { __dim_compare_#2:w }
4020   \fi:
4021   #1 pt \exp_stop_f:
4022   \prg_return_false:
4023   \exp_after:wN \use_none_delimit_by_q_stop:w
4024   \fi:
4025   \reverse_if:N \if_dim:w #1 pt #2
4026   \exp_after:wN \__dim_compare:wNN
4027   \dim_use:N \__dim_eval:w #3
4028 }
4029 \cs_new:cpn { __dim_compare_ ! :w }
4030 #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
4031 \cs_new:cpn { __dim_compare_ = :w }
4032 #1 \__dim_eval:w = { #1 \__dim_eval:w }
4033 \cs_new:cpn { __dim_compare_ < :w }
4034 #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
4035 \cs_new:cpn { __dim_compare_ > :w }
4036 #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
4037 \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \q_stop
4038 { #1 \prg_return_false: \else: \prg_return_true: \fi: }

```

(End definition for \dim_compare:n. These functions are documented on page 79.)

\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 4039 \cs_new:Npn \dim_case:nnTF #1
\__dim_case:nw 4040 {
\__dim_case_end:nw 4041   \tex_romannumeral:D
4042   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} }
4043 }
4044 \cs_new:Npn \dim_case:nnT #1#2#3
4045 {
4046   \tex_romannumeral:D
4047   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
4048 }

```

```

4049 \cs_new:Npn \dim_case:nnF #1#2
4050 {
4051   \tex_romannumeral:D
4052   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
4053 }
4054 \cs_new:Npn \dim_case:nn #1#2
4055 {
4056   \tex_romannumeral:D
4057   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
4058 }
4059 \cs_new:Npn \__dim_case:nnTF #1#2#3#4
4060 { \__dim_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
4061 \cs_new:Npn \__dim_case:nw #1#2#3
4062 {
4063   \dim_compare:nNnTF {#1} = {#2}
4064   { \__dim_case_end:nw {#3} }
4065   { \__dim_case:nw {#1} }
4066 }
4067 \cs_new_eq:NN \__dim_case_end:nw \__prg_case_end:nw

```

(End definition for `\dim_case:nn`. This function is documented on page 80.)

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
4068 \cs_set:Npn \dim_while_do:nn #1#2
4069 {
4070   \dim_compare:nT {#1}
4071   {
4072     #2
4073     \dim_while_do:nn {#1} {#2}
4074   }
4075 }
4076 \cs_set:Npn \dim_until_do:nn #1#2
4077 {
4078   \dim_compare:nF {#1}
4079   {
4080     #2
4081     \dim_until_do:nn {#1} {#2}
4082   }
4083 }
4084 \cs_set:Npn \dim_do_while:nn #1#2
4085 {
4086   #2
4087   \dim_compare:nT {#1}
4088   { \dim_do_while:nn {#1} {#2} }
4089 }
4090 \cs_set:Npn \dim_do_until:nn #1#2
4091 {

```

```

4092     #2
4093     \dim_compare:nF {#1}
4094     { \dim_do_until:nn {#1} {#2} }
4095 }

```

(End definition for `\dim_while_do:nn`. This function is documented on page 81.)

`\dim_while_do:nNnn` `\dim_until_do:nNnn` `\dim_do_while:nNnn` `\dim_do_until:nNnn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

4096 \cs_set:Npn \dim_while_do:nNnn #1#2#3#4
4097 {
4098     \dim_compare:nNnT {#1} #2 {#3}
4099     {
4100         #4
4101         \dim_while_do:nNnn {#1} #2 {#3} {#4}
4102     }
4103 }
4104 \cs_set:Npn \dim_until_do:nNnn #1#2#3#4
4105 {
4106     \dim_compare:nNnF {#1} #2 {#3}
4107     {
4108         #4
4109         \dim_until_do:nNnn {#1} #2 {#3} {#4}
4110     }
4111 }
4112 \cs_set:Npn \dim_do_while:nNnn #1#2#3#4
4113 {
4114     #4
4115     \dim_compare:nNnT {#1} #2 {#3}
4116     { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
4117 }
4118 \cs_set:Npn \dim_do_until:nNnn #1#2#3#4
4119 {
4120     #4
4121     \dim_compare:nNnF {#1} #2 {#3}
4122     { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
4123 }

```

(End definition for `\dim_while_do:nNnn`. This function is documented on page 80.)

9.7 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

4124 \cs_new:Npn \dim_eval:n #1
4125 { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_eval:n`. This function is documented on page 81.)

`__dim_strip_bp:n`

```

4126 \cs_new:Npn \__dim_strip_bp:n #1
4127 { \__dim_strip_pt:n { 0.996 26 \__dim_eval:w #1 \__dim_eval_end: } }

```

(End definition for `_dim_strip_bp:n`.)

`_dim_strip_pt:n` A function which comes up often enough to deserve a place in the kernel. The idea here is that the input is assumed to be in pt, but can be given in other units, while the output is the value of the dimension in pt but with no units given. This is used a lot by low-level manipulations.

```

4128 \cs_new:Npn \_dim_strip_pt:n #1
4129 {
4130   \exp_after:wN
4131   \_dim_strip_pt:w \dim_use:N \_dim_eval:w #1 \_dim_eval_end: \q_stop
4132 }
4133 \use:x
4134 {
4135   \cs_new:Npn \exp_not:N \_dim_strip_pt:w
4136   ##1 . ##2 \tl_to_str:n { pt } ##3 \exp_not:N \q_stop
4137   {
4138     ##1
4139     \exp_not:N \int_compare:nNtT {##2} > \c_zero
4140     { . ##2 }
4141   }
4142 }

```

(End definition for `_dim_strip_pt:n`. This function is documented on page 89.)

`\dim_use:N` Accessing a $\langle dim \rangle$.

`\dim_use:c`

```

4143 \cs_new_eq:NN \dim_use:N \tex_the:D
4144 \cs_generate_variant:Nn \dim_use:N { c }

```

(End definition for `\dim_use:N` and `\dim_use:c`. These functions are documented on page ??.)

9.8 Viewing dim variables

`\dim_show:N` Diagnostics.

`\dim_show:c`

```

4145 \cs_new_eq:NN \dim_show:N \_kernel_register_show:N
4146 \cs_generate_variant:Nn \dim_show:N { c }

```

(End definition for `\dim_show:N` and `\dim_show:c`. These functions are documented on page ??.)

`\dim_show:n` Diagnostics. We don't use the TeX primitive `\showthe` to show dimension expressions: this gives a more unified output, since the closing brace is read by the dimension expression in all cases.

```

4147 \cs_new_protected:Npn \dim_show:n #1
4148 { \etex_showtokens:D \exp_after:wN { \dim_use:N \_dim_eval:w #1 } }

```

(End definition for `\dim_show:n`. This function is documented on page 82.)

9.9 Constant dimensions

`\c_zero_dim` Constant dimensions: in package mode, a couple of registers can be saved.

`\c_max_dim`

```

4149 \dim_const:Nn \c_zero_dim { 0 pt }
4150 \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 82.)

9.10 Scratch dimensions

`\l_tmpa_dim` We provide two local and two global scratch registers, maybe we need more or less.
`\l_tmppb_dim`
`\g_tmpa_dim`
`\g_tmppb_dim`

```

4151 \dim_new:N \l_tmpa_dim
4152 \dim_new:N \l_tmppb_dim
4153 \dim_new:N \g_tmpa_dim
4154 \dim_new:N \g_tmppb_dim

```

(End definition for `\l_tmpa_dim` and `\l_tmppb_dim`. These functions are documented on page 82.)

9.11 Creating and initialising skip variables

`\skip_new:N` Allocation of a new internal registers.
`\skip_new:c`

```

4155 <*package>
4156 \cs_new_protected:Npn \skip_new:N #1
4157 {
4158   \__chk_if_free_cs:N #1
4159   \newskip #1
4160 }
4161 </package>
4162 \cs_generate_variant:Nn \skip_new:N { c }

```

(End definition for `\skip_new:N` and `\skip_new:c`. These functions are documented on page ??.)

`\skip_const:Nn` Contrarily to integer constants, we cannot avoid using a register, even for constants.
`\skip_const:cn`

```

4163 \cs_new_protected:Npn \skip_const:Nn #1
4164 {
4165   \skip_new:N #1
4166   \skip_gset:Nn #1
4167 }
4168 \cs_generate_variant:Nn \skip_const:Nn { c }

```

(End definition for `\skip_const:Nn` and `\skip_const:cn`. These functions are documented on page ??.)

`\skip_zero:N` Reset the register to zero.
`\skip_zero:c`
`\skip_gzero:N`
`\skip_gzero:c`

```

4169 \cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }
4170 \cs_new_protected:Npn \skip_gzero:N { \tex_global:D \skip_zero:N }
4171 \cs_generate_variant:Nn \skip_zero:N { c }
4172 \cs_generate_variant:Nn \skip_gzero:N { c }

```

(End definition for `\skip_zero:N` and `\skip_zero:c`. These functions are documented on page ??.)

`\skip_zero_new:N` Create a register if needed, otherwise clear it.
`\skip_zero_new:c`
`\skip_gzero_new:N`
`\skip_gzero_new:c`

```

4173 \cs_new_protected:Npn \skip_zero_new:N #1
4174 { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
4175 \cs_new_protected:Npn \skip_gzero_new:N #1
4176 { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
4177 \cs_generate_variant:Nn \skip_zero_new:N { c }
4178 \cs_generate_variant:Nn \skip_gzero_new:N { c }

```

(End definition for `\skip_zero_new:N` and others. These functions are documented on page ??.)

`\skip_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.
`\skip_if_exist_p:c` 4179 `\prg_new_eq_conditional:Nn \skip_if_exist:N \cs_if_exist:N { TF , T , F , p }`
`\skip_if_exist:NTF` 4180 `\prg_new_eq_conditional:Nn \skip_if_exist:c \cs_if_exist:c { TF , T , F , p }`
`\skip_if_exist:cTF` (End definition for `\skip_if_exist:N` and `\skip_if_exist:c`. These functions are documented on page ??.)

9.12 Setting skip variables

`\skip_set:Nn` Much the same as for dimensions.
`\skip_set:cn` 4181 `\cs_new_protected:Npn \skip_set:Nn #1#2`
`\skip_gset:Nn` 4182 `{ #1 ~ \etex_glueexpr:D #2 \scan_stop: }`
`\skip_gset:cn` 4183 `\cs_new_protected:Npn \skip_gset:Nn { \tex_global:D \skip_set:Nn }`
4184 `\cs_generate_variant:Nn \skip_set:Nn { c }`
4185 `\cs_generate_variant:Nn \skip_gset:Nn { c }`
(End definition for `\skip_set:Nn` and `\skip_set:cn`. These functions are documented on page ??.)

`\skip_set_eq:NN` All straightforward.
`\skip_set_eq:cN` 4186 `\cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }`
`\skip_set_eq:Nc` 4187 `\cs_generate_variant:Nn \skip_set_eq:NN { c }`
`\skip_set_eq:cc` 4188 `\cs_generate_variant:Nn \skip_set_eq:NN { Nc , cc }`
`\skip_gset_eq:NN` 4189 `\cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }`
`\skip_gset_eq:cN` 4190 `\cs_generate_variant:Nn \skip_gset_eq:NN { c }`
`\skip_gset_eq:Nc` 4191 `\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 ??.)

`\skip_add:Nn` Using `by` here deals with the (incorrect) case `\skip123`.
`\skip_add:cn` 4192 `\cs_new_protected:Npn \skip_add:Nn #1#2`
`\skip_gadd:Nn` 4193 `{ \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }`
`\skip_gadd:cn` 4194 `\cs_new_protected:Npn \skip_gadd:Nn { \tex_global:D \skip_add:Nn }`
`\skip_sub:Nn` 4195 `\cs_generate_variant:Nn \skip_add:Nn { c }`
`\skip_sub:cn` 4196 `\cs_generate_variant:Nn \skip_gadd:Nn { c }`
`\skip_gsub:Nn` 4197 `\cs_new_protected:Npn \skip_sub:Nn #1#2`
`\skip_gsub:cn` 4198 `{ \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }`
4199 `\cs_new_protected:Npn \skip_gsub:Nn { \tex_global:D \skip_sub:Nn }`
4200 `\cs_generate_variant:Nn \skip_sub:Nn { c }`
4201 `\cs_generate_variant:Nn \skip_gsub:Nn { c }`
(End definition for `\skip_add:Nn` and `\skip_add:cn`. These functions are documented on page ??.)

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.
4202 `\prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }`
4203 `{`
4204 `\if_int_compare:w`
4205 `\pdfTeX_strcmp:D { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }`
4206 `= \c_zero`
4207 `\prg_return_true:`


```

4208     \else:
4209         \prg_return_false:
4210     \fi:
4211 }

```

(End definition for `\skip_if_eq:nn`. These functions are documented on page 84.)

```

\skip_if_finite_p:n
\skip_if_finite:nTF
\__skip_if_finite:wwNw

```

With ε -TeX, we have an easy access to the order of infinities of the stretch and shrink components of a skip. However, to access both, we either need to evaluate the expression twice, or evaluate it, then call an auxiliary to extract both pieces of information from the result. Since we are going to need an auxiliary anyways, it is quicker to make it search for the string `fil` which characterizes infinite glue.

```

4212 \cs_set_protected:Npn \__cs_tmp:w #1
4213 {
4214     \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
4215     {
4216         \exp_after:wN \__skip_if_finite:wwNw
4217         \skip_use:N \etex_glueexpr:D ##1 ; \prg_return_false:
4218         #1 ; \prg_return_true: \q_stop
4219     }
4220     \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \q_stop {##3}
4221 }
4222 \exp_args:No \__cs_tmp:w { \tl_to_str:n { fil } }

```

(End definition for `\skip_if_finite:n`. These functions are documented on page 84.)

9.14 Using skip expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```

4223 \cs_new:Npn \skip_eval:n #1
4224 { \skip_use:N \etex_glueexpr:D #1 \scan_stop: }

```

(End definition for `\skip_eval:n`. This function is documented on page 84.)

`\skip_use:N` Accessing a $\langle skip \rangle$.

```

\skip_use:c
4225 \cs_new_eq:NN \skip_use:N \tex_the:D
4226 \cs_generate_variant:Nn \skip_use:N { c }

```

(End definition for `\skip_use:N` and `\skip_use:c`. These functions are documented on page ??.)

9.15 Inserting skips into the output

`\skip_horizontal:N` Inserting skips.

```

\skip_horizontal:c
\skip_horizontal:n
\skip_vertical:N
\skip_vertical:c
\skip_vertical:n
4227 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
4228 \cs_new:Npn \skip_horizontal:n #1
4229 { \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop: }
4230 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
4231 \cs_new:Npn \skip_vertical:n #1
4232 { \skip_vertical:N \etex_glueexpr:D #1 \scan_stop: }
4233 \cs_generate_variant:Nn \skip_horizontal:N { c }
4234 \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 ??.)

9.16 Viewing skip variables

`\skip_show:N` Diagnostics.

`\skip_show:c` 4235 `\cs_new_eq:NN \skip_show:N __kernel_register_show:N`
4236 `\cs_generate_variant:Nn \skip_show:N { c }`
(End definition for `\skip_show:N` and `\skip_show:c`. These functions are documented on page ??.)

`\skip_show:n` Diagnostics. We don't use the T_EX primitive `\showthe` to show skip expressions: this gives a more unified output, since the closing brace is read by the skip expression in all cases.

4237 `\cs_new_protected:Npn \skip_show:n #1`
4238 `{ \etex_showtokens:D \exp_after:wN { \tex_the:D \etex_glueexpr:D #1 } }`
(End definition for `\skip_show:n`. This function is documented on page 85.)

9.17 Constant skips

`\c_zero_skip` Skips with no rubber component are just dimensions but need to terminate correctly.

`\c_max_skip` 4239 `\skip_const:Nn \c_zero_skip { \c_zero_dim }`
4240 `\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 85.)

9.18 Scratch skips

`\l_tmpa_skip` We provide two local and two global scratch registers, maybe we need more or less.

`\l_tmpb_skip` 4241 `\skip_new:N \l_tmpa_skip`
`\g_tmpa_skip` 4242 `\skip_new:N \l_tmpb_skip`
`\g_tmpb_skip` 4243 `\skip_new:N \g_tmpa_skip`
4244 `\skip_new:N \g_tmpb_skip`
(End definition for `\l_tmpa_skip` and `\l_tmpb_skip`. These functions are documented on page 85.)

9.19 Creating and initialising muskip variables

`\muskip_new:N` And then we add muskips.

`\muskip_new:c` 4245 `\<package>`
4246 `\cs_new_protected:Npn \muskip_new:N #1`
4247 `{`
4248 `__chk_if_free_cs:N #1`
4249 `\newmuskip #1`
4250 `}`
4251 `\</package>`
4252 `\cs_generate_variant:Nn \muskip_new:N { c }`
(End definition for `\muskip_new:N` and `\muskip_new:c`. These functions are documented on page ??.)

\muskip_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\muskip_const:cn
4253 \cs_new_protected:Npn \muskip_const:Nn #1
4254 {
4255     \muskip_new:N #1
4256     \muskip_gset:Nn #1
4257 }
4258 \cs_generate_variant:Nn \muskip_const:Nn { c }
```

(End definition for \muskip_const:Nn and \muskip_const:cn. These functions are documented on page ??.)

\muskip_zero:N Reset the register to zero.

```
\muskip_zero:c
4259 \cs_new_protected:Npn \muskip_zero:N #1
\muskip_gzero:N
4260 { #1 \c_zero_muskip }
\muskip_gzero:c
4261 \cs_new_protected:Npn \muskip_gzero:N { \tex_global:D \muskip_zero:N }
4262 \cs_generate_variant:Nn \muskip_zero:N { c }
4263 \cs_generate_variant:Nn \muskip_gzero:N { c }
```

(End definition for \muskip_zero:N and \muskip_zero:c. These functions are documented on page ??.)

\muskip_zero_new:N Create a register if needed, otherwise clear it.

```
\muskip_zero_new:c
4264 \cs_new_protected:Npn \muskip_zero_new:N #1
\muskip_gzero_new:N
4265 { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
\muskip_gzero_new:c
4266 \cs_new_protected:Npn \muskip_gzero_new:N #1
4267 { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
4268 \cs_generate_variant:Nn \muskip_zero_new:N { c }
4269 \cs_generate_variant:Nn \muskip_gzero_new:N { c }
```

(End definition for \muskip_zero_new:N and others. These functions are documented on page ??.)

\muskip_if_exist_p:N Copies of the cs functions defined in l3basics.

```
\muskip_if_exist_p:c
4270 \prg_new_eq_conditional:NNn \muskip_if_exist:N \cs_if_exist:N { TF , T , F , p }
\muskip_if_exist:NTF
4271 \prg_new_eq_conditional:NNn \muskip_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

\muskip_if_exist:cTF (End definition for \muskip_if_exist:N and \muskip_if_exist:c. These functions are documented on page ??.)

9.20 Setting muskip variables

\muskip_set:Nn This should be pretty familiar.

```
\muskip_set:cn
4272 \cs_new_protected:Npn \muskip_set:Nn #1#2
\muskip_gset:Nn
4273 { #1 ~ \etex_muexpr:D #2 \scan_stop: }
\muskip_gset:cn
4274 \cs_new_protected:Npn \muskip_gset:Nn { \tex_global:D \muskip_set:Nn }
4275 \cs_generate_variant:Nn \muskip_set:Nn { c }
4276 \cs_generate_variant:Nn \muskip_gset:Nn { c }
```

(End definition for \muskip_set:Nn and \muskip_set:cn. These functions are documented on page ??.)

\muskip_set_eq:NN All straightforward.

```
\muskip_set_eq:cn
4277 \cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
\muskip_set_eq:Nc
4278 \cs_generate_variant:Nn \muskip_set_eq:NN { c }
\muskip_set_eq:cc
4279 \cs_generate_variant:Nn \muskip_set_eq:NN { Nc , cc }
\muskip_gset_eq:NN
4280 \cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\muskip_gset_eq:cn
\muskip_gset_eq:Nc
\muskip_gset_eq:cc
```

```

4281 \cs_generate_variant:Nn \muskip_gset_eq:NN { c }
4282 \cs_generate_variant:Nn \muskip_gset_eq:NN { Nc , cc }

```

(End definition for \muskip_set_eq:NN and others. These functions are documented on page ??.)

\muskip_add:Nn Using by here deals with the (incorrect) case \muskip123.

```

\muskip_add:cn
\muskip_gadd:Nn
\muskip_gadd:cn
\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn

```

```

4283 \cs_new_protected:Npn \muskip_add:Nn #1#2
4284 { \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }
4285 \cs_new_protected:Npn \muskip_gadd:Nn { \tex_global:D \muskip_add:Nn }
4286 \cs_generate_variant:Nn \muskip_add:Nn { c }
4287 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
4288 \cs_new_protected:Npn \muskip_sub:Nn #1#2
4289 { \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }
4290 \cs_new_protected:Npn \muskip_gsub:Nn { \tex_global:D \muskip_sub:Nn }
4291 \cs_generate_variant:Nn \muskip_sub:Nn { c }
4292 \cs_generate_variant:Nn \muskip_gsub:Nn { c }

```

(End definition for \muskip_add:Nn and \muskip_add:cn. These functions are documented on page ??.)

9.21 Using muskip expressions and variables

\muskip_eval:n Evaluating a muskip expression expandably.

```

4293 \cs_new:Npn \muskip_eval:n #1
4294 { \muskip_use:N \etex_muexpr:D #1 \scan_stop: }

```

(End definition for \muskip_eval:n. This function is documented on page 87.)

\muskip_use:N Accessing a $\langle muskip \rangle$.

\muskip_use:c

```

4295 \cs_new_eq:NN \muskip_use:N \tex_the:D
4296 \cs_generate_variant:Nn \muskip_use:N { c }

```

(End definition for \muskip_use:N and \muskip_use:c. These functions are documented on page ??.)

9.22 Viewing muskip variables

\muskip_show:N Diagnostics.

\muskip_show:c

```

4297 \cs_new_eq:NN \muskip_show:N \__kernel_register_show:N
4298 \cs_generate_variant:Nn \muskip_show:N { c }

```

(End definition for \muskip_show:N and \muskip_show:c. These functions are documented on page ??.)

\muskip_show:n Diagnostics. We don't use the TeX primitive \showthe to show muskip expressions: this gives a more unified output, since the closing brace is read by the muskip expression in all cases.

```

4299 \cs_new_protected:Npn \muskip_show:n #1
4300 { \etex_showtokens:D \exp_after:wN { \tex_the:D \etex_muexpr:D #1 } }

```

(End definition for \muskip_show:n. This function is documented on page 88.)

9.23 Constant muskips

`\c_zero_muskip` Constant muskips given by their value.
`\c_max_muskip`

```
4301 \muskip_const:Nn \c_zero_muskip { 0 mu }
4302 \muskip_const:Nn \c_max_muskip { 16383.99999 mu }
```

(End definition for `\c_zero_muskip`. This function is documented on page 88.)

9.24 Scratch muskips

We provide two local and two global scratch registers, maybe we need more or less.

```
4303 \muskip_new:N \l_tmpa_muskip
4304 \muskip_new:N \l_tmpb_muskip
4305 \muskip_new:N \g_tmpa_muskip
4306 \muskip_new:N \g_tmpb_muskip
```

(End definition for `\l_tmpa_muskip` and `\l_tmpb_muskip`. These functions are documented on page 88.)

9.25 Deprecated functions

`\dim_case:nnn` Deprecated 2013-07-15.

```
4307 \cs_new_eq:NN \dim_case:nnn \dim_case:nnF
```

(End definition for `\dim_case:nnn`. This function is documented on page ??.)

```
4308 </initex | package>
```

10 l3tl implementation

```
4309 <*initex | package>
4310 <@@=tl>
4311 <*package>
4312 \ProvidesExplPackage
4313   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
4314   \__expl_package_check:
4315 </package>
```

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
`\tl_new:c` the definition.

```
4316 \cs_new_protected:Npn \tl_new:N #1
4317 {
4318   \__chk_if_free_cs:N #1
4319   \cs_gset_eq:NN #1 \c_empty_tl
4320 }
4321 \cs_generate_variant:Nn \tl_new:N { c }
```

(End definition for `\tl_new:N` and `\tl_new:c`. These functions are documented on page ??.)

`\tl_const:Nn` Constants are also easy to generate.

```

\tl_const:Nx 4322 \cs_new_protected:Npn \tl_const:Nn #1#2
\tl_const:cn 4323 {
\tl_const:cx 4324 \__chk_if_free_cs:N #1
4325 \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
4326 }
4327 \cs_new_protected:Npn \tl_const:Nx #1#2
4328 {
4329 \__chk_if_free_cs:N #1
4330 \cs_gset_nopar:Npx #1 {#2}
4331 }
4332 \cs_generate_variant:Nn \tl_const:Nn { c }
4333 \cs_generate_variant:Nn \tl_const:Nx { c }

```

(End definition for `\tl_const:Nn` and others. These functions are documented on page ??.)

`\tl_clear:N` Clearing a token list variable means setting it to an empty value. Error checking will be
`\tl_clear:c` sorted out by the parent function.

`\tl_gclear:N`

```

4334 \cs_new_protected:Npn \tl_clear:N #1
4335 { \tl_set_eq:NN #1 \c_empty_tl }
4336 \cs_new_protected:Npn \tl_gclear:N #1
4337 { \tl_gset_eq:NN #1 \c_empty_tl }
4338 \cs_generate_variant:Nn \tl_clear:N { c }
4339 \cs_generate_variant:Nn \tl_gclear:N { c }

```

(End definition for `\tl_clear:N` and `\tl_clear:c`. These functions are documented on page ??.)

`\tl_clear_new:N` Clearing a token list variable means setting it to an empty value. Error checking will be
`\tl_clear_new:c` sorted out by the parent function.

`\tl_gclear_new:N`

```

4340 \cs_new_protected:Npn \tl_clear_new:N #1
4341 { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
4342 \cs_new_protected:Npn \tl_gclear_new:N #1
4343 { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
4344 \cs_generate_variant:Nn \tl_clear_new:N { c }
4345 \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 ??.)

`\tl_set_eq:NN` For setting token list variables equal to each other.

```

\tl_set_eq:Nc 4346 \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN
\tl_set_eq:cN 4347 \cs_new_eq:NN \tl_set_eq:cN \cs_set_eq:cN
\tl_set_eq:cc 4348 \cs_new_eq:NN \tl_set_eq:Nc \cs_set_eq:Nc
\tl_gset_eq:NN 4349 \cs_new_eq:NN \tl_set_eq:cc \cs_set_eq:cc
\tl_gset_eq:Nc 4350 \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
\tl_gset_eq:cN 4351 \cs_new_eq:NN \tl_gset_eq:cN \cs_gset_eq:cN
\tl_gset_eq:Nc 4352 \cs_new_eq:NN \tl_gset_eq:Nc \cs_gset_eq:Nc
\tl_gset_eq:cc 4353 \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 ??.)

`\tl_concat:NNN` Concatenating token lists is easy.

```

\tl_concat:ccc 4354 \cs_new_protected:Npn \tl_concat:NNN #1#2#3
\tl_gconcat:NNN 4355 { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
\tl_gconcat:ccc 4356 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
4357 { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
4358 \cs_generate_variant:Nn \tl_concat:NNN { ccc }
4359 \cs_generate_variant:Nn \tl_gconcat:NNN { ccc }

```

(End definition for `\tl_concat:NNN` and `\tl_concat:ccc`. These functions are documented on page ??.)

`\tl_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\tl_if_exist_p:c 4360 \prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
\tl_if_exist:NTF 4361 \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:N` and `\tl_if_exist:c`. These functions are documented on page ??.)

10.2 Constant token lists

`\c_empty_tl` Never full. We need to define that constant before using `\tl_new:N`.

```

4362 \tl_const:Nn \c_empty_tl { }

```

(End definition for `\c_empty_tl`. This variable is documented on page 103.)

`\c_job_name_tl` Inherited from the \LaTeX name for the primitive: this needs to actually contain the text of the job name rather than the name of the primitive, of course. \LuaTeX does not quote file names containing spaces, whereas \pdfTeX and \XeTeX do. So there may be a correction to make in the \LuaTeX case.

```

4363 <*initex>
4364 \luatex_if_engine:T
4365 {
4366   \tex_everyjob:D \exp_after:wN
4367   {
4368     \tex_the:D \tex_everyjob:D
4369     \lua_now_x:n
4370     { dofile ( assert ( kpse.find_file ("lualatexquotejobname.lua" ) ) ) }
4371   }
4372 }
4373 \tex_everyjob:D \exp_after:wN
4374 {
4375   \tex_the:D \tex_everyjob:D
4376   \tl_const:Nx \c_job_name_tl { \tex_jobname:D }
4377 }
4378 </initex>
4379 <*package>
4380 \tl_const:Nx \c_job_name_tl { \tex_jobname:D }
4381 </package>

```

(End definition for `\c_job_name_tl`. This variable is documented on page 103.)

`\c_space_tl` A space as a token list (as opposed to as a character).

```

4382 \tl_const:Nn \c_space_tl { ~ }

```

(End definition for `\c_space_tl`. This variable is documented on page 103.)

10.3 Adding to token list variables

By using `\exp_not:n` token list variables can contain `#` tokens, which makes the token list registers provided by `TEX` more or less redundant. The `\tl_set:No` version is done “by hand” as it is used quite a lot.

```

\cs_new_protected:Npn \tl_set:Nn #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
\cs_new_protected:Npn \tl_set:Nv #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
\cs_new_protected:Npn \tl_set:Nx #1#2
\cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
\cs_new_protected:Npn \tl_set:cV #1#2
\cs_set_nopar:Npx #1 {#2} }
\cs_new_protected:Npn \tl_set:cv #1#2
\cs_set_nopar:Npx #1 {#2} }
\cs_new_protected:Npn \tl_set:co #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
\cs_new_protected:Npn \tl_set:cf #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
\cs_new_protected:Npn \tl_set:cx #1#2
\cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
\cs_new_protected:Npn \tl_gset:Nx #1#2
\cs_set_nopar:Npx #1 {#2} }
\cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
\cs_generate_variant:Nn \tl_set:Nx { c }
\cs_generate_variant:Nn \tl_set:Nn { c , co , cV , cv , cf }
\cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
\cs_generate_variant:Nn \tl_gset:Nx { c }
\cs_generate_variant:Nn \tl_gset:Nn { c , co , cV , cv , cf }

```

(End definition for \tl_set:Nn and others. These functions are documented on page ??.)

```

\cs_new_protected:Npn \tl_put_left:Nn #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_put_left:Nv #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_put_left:Nx #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_put_left:cV #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_put_left:cv #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_put_left:co #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_put_left:cx #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:Nn #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:Nv #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:Nx #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:cV #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:cv #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:co #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:cx #1#2
\cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_generate_variant:Nn \tl_put_left:Nn { c }
\cs_generate_variant:Nn \tl_put_left:Nv { c }
\cs_generate_variant:Nn \tl_put_left:Nx { c }
\cs_generate_variant:Nn \tl_gput_left:Nn { c }
\cs_generate_variant:Nn \tl_gput_left:Nv { c }
\cs_generate_variant:Nn \tl_gput_left:Nx { c }

```


The same on the right.

The same on the right.

10.4 Reassigning token list category codes

```

4449 \group_begin:
4450   \tex_lccode:D ‘\A = ‘@ \scan_stop:
4451   \tex_lccode:D ‘\B = ‘@ \scan_stop:
4452   \tex_catcode:D ‘\A = 8 \scan_stop:
4453   \tex_catcode:D ‘\B = 3 \scan_stop:
4454   \tex_lowercase:D
4455   {
4456     \group_end:
4457     \tl_const:Nn \c__tl_rescan_marker_tl { A B }
4458   }

```

```
! File ended while scanning definition of ...
```

341

When expanding a token list this can be handled using `\exp_not:N` but this fails if the token list is not being expanded. So instead a delimited argument is used with an end marker which cannot appear within the token list which is scanned: two `@` symbols with different category codes. The rescanned token list cannot contain the end marker, because all `@` present in the token list are read with the same category code. As every character with charcode `\newlinechar` is replaced by the `\endlinechar`, and an extra `\endlinechar` is added at the end, we need to set both of those to `-1`, “unprintable”.

```

4459 \cs_new_protected_nopar:Npn \tl_set_rescan:Nnn
4460 { \tl_set_rescan:NNnn \tl_set:Nn }
4461 \cs_new_protected_nopar:Npn \tl_gset_rescan:Nnn
4462 { \tl_set_rescan:NNnn \tl_gset:Nn }
4463 \cs_new_protected_nopar:Npn \tl_rescan:nn
4464 { \tl_set_rescan:NNnn \prg_do_nothing: \use:n }
4465 \cs_new_protected:Npn \tl_set_rescan:NNnn #1#2#3#4
4466 {
4467   \group_begin:
4468     \exp_args:No \etex_veryeof:D { \c__tl_rescan_marker_tl \exp_not:N }
4469     \tex_endlinechar:D \c_minus_one
4470     \tex_newlinechar:D \c_minus_one
4471     #3
4472     \use:x
4473     {
4474       \group_end:
4475       #1 \exp_not:N #2
4476       {
4477         \exp_after:wN \tl_rescan:w
4478         \exp_after:wN \prg_do_nothing:
4479         \etex_scantokens:D {#4}
4480       }
4481     }
4482   }
4483   \use:x
4484   {
4485     \cs_new:Npn \exp_not:N \tl_rescan:w ##1
4486       \c__tl_rescan_marker_tl
4487       { \exp_not:N \exp_not:o { ##1 } }
4488   }
4489   \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno , Nnx }
4490   \cs_generate_variant:Nn \tl_set_rescan:Nnn { c , cno , cnx }
4491   \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno , Nnx }
4492   \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 93.)

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`

```

4493 \cs_new_protected:Npn \tl_to_lowercase:n #1
4494 { \tex_lowercase:D {#1} }

```

```

4495 \cs_new_protected:Npn \tl_to_uppercase:n #1
4496 { \tex_uppercase:D {#1} }
(End definition for \tl_to_lowercase:n. This function is documented on page 94.)

```

10.6 Modifying token list variables

```

\tl_replace_all:Nnn All of the replace functions are based on \__tl_replace:NNNnn, whose arguments are:
\tl_replace_all:cnn <function>, \tl_(g)set:Nx, <tl var>, <search tokens>, <replacement tokens>.
\tl_greplace_all:Nnn
\tl_greplace_all:cnn
\tl_replace_once:Nnn
\tl_replace_once:cnn
\tl_greplace_once:Nnn
\tl_greplace_once:cnn
\__tl_replace:NNNnn
\__tl_replace:w
\__tl_replace_all:
\__tl_replace_once:
\__tl_replace_once_end:w

```

```

4497 \cs_new_protected_nopar:Npn \tl_replace_once:Nnn
4498 { \__tl_replace:NNNnn \__tl_replace_once: \tl_set:Nx }
4499 \cs_new_protected_nopar:Npn \tl_greplace_once:Nnn
4500 { \__tl_replace:NNNnn \__tl_replace_once: \tl_gset:Nx }
4501 \cs_new_protected_nopar:Npn \tl_replace_all:Nnn
4502 { \__tl_replace:NNNnn \__tl_replace_all: \tl_set:Nx }
4503 \cs_new_protected_nopar:Npn \tl_greplace_all:Nnn
4504 { \__tl_replace:NNNnn \__tl_replace_all: \tl_gset:Nx }
4505 \cs_generate_variant:Nn \tl_replace_once:Nnn { c }
4506 \cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
4507 \cs_generate_variant:Nn \tl_replace_all:Nnn { c }
4508 \cs_generate_variant:Nn \tl_greplace_all:Nnn { c }

```

The idea is easier to understand by considering the case of `\tl_replace_all:Nnn`. The replacement happens within an `x`-type expansion. We use an auxiliary function `__tl_tmp:w`, which essentially replaces the next `<search tokens>` by `<replacement tokens>`. To avoid runaway arguments, we expand something like `__tl_tmp:w <token list> \q_mark <search tokens> \q_stop`, repeating until the end. How do we detect that we have reached the last occurrence of `<search tokens>`? The last replacement is characterized by the fact that the argument of `__tl_tmp:w` contains `\q_mark`. In the code below, `__tl_replace:w` takes an argument delimited by `\q_mark`, and removes the following token. Before we reach the end, this gobbles `\q_mark \use_none_delimit_by_q_stop:w` which appear in the definition of `__tl_tmp:w`, and leaves the `<replacement tokens>`, passed to `\exp_not:n`, to be included in the `x`-expanding definition. At the end, the first `\q_mark` is within the argument of `__tl_tmp:w`, and `__tl_replace:w` gobbles the second `\q_mark` as well, leaving `\use_none_delimit_by_q_stop:w`, which ends the recursion cleanly.

```

4509 \cs_new_protected:Npn \__tl_replace:NNNnn #1#2#3#4#5
4510 {
4511   \tl_if_empty:nTF {#4}
4512   {
4513     \_msg_kernel_error:nnx { kernel } { empty-search-pattern }
4514     { \tl_to_str:n {#5} }
4515   }
4516   {
4517     \group_align_safe_begin:
4518     \cs_set:Npx \__tl_tmp:w ##1##2 #4
4519     {
4520       ##2
4521       \exp_not:N \q_mark

```

```

4522         \exp_not:N \use_none_delimit_by_q_stop:w
4523         \exp_not:n { \exp_not:n {#5} }
4524         ##1
4525     }
4526     \group_align_safe_end:
4527     #2 #3
4528     {
4529         \exp_after:wN #1
4530         #3 \q_mark #4 \q_stop
4531     }
4532 }
4533 }
4534 \cs_new:Npn \__tl_replace:w #1 \q_mark #2 { \exp_not:o {#1} }

```

The first argument of `__tl_tmp:w` is responsible for repeating the replacement in the case of `replace_all`, and stopping it early for `replace_once`. Note also that we build `__tl_tmp:w` within an x-expansion so that the *<replacement tokens>* can contain `#`. The second `\exp_not:n` ensures that the *<replacement tokens>* are not expanded by `\tl_(g)set:Nx`.

Now on to the difference between “once” and “all”. The `\prg_do_nothing:` and accompanying o-expansion ensure that we don’t lose braces in case the tokens between two occurrences of the *<search tokens>* form a brace group.

```

4535 \cs_new_nopar:Npn \__tl_replace_all:
4536 {
4537     \exp_after:wN \__tl_replace:w
4538     \__tl_tmp:w \__tl_replace_all: \prg_do_nothing:
4539 }
4540 \cs_new_nopar:Npn \__tl_replace_once:
4541 {
4542     \exp_after:wN \__tl_replace:w
4543     \__tl_tmp:w { \__tl_replace_once_end:w \prg_do_nothing: } \prg_do_nothing:
4544 }
4545 \cs_new:Npn \__tl_replace_once_end:w #1 \q_mark #2 \q_stop
4546 { \exp_not:o {#1} }

```

(End definition for `\tl_replace_all:Nnn` and `\tl_replace_all:cnn`. These functions are documented on page ??.)

```

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

```

Removal is just a special case of replacement.

```

4547 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
4548 { \tl_replace_once:Nnn #1 {#2} { } }
4549 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
4550 { \tl_greplace_once:Nnn #1 {#2} { } }
4551 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
4552 \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 ??.)

```

\tl_remove_all:Nn
\tl_remove_all:cn
\tl_gremove_all:Nn
\tl_gremove_all:cn

```

Removal is just a special case of replacement.

```

4553 \cs_new_protected:Npn \tl_remove_all:Nn #1#2

```

```

4554 { \tl_replace_all:Nnn #1 {#2} { } }
4555 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
4556 { \tl_greplace_all:Nnn #1 {#2} { } }
4557 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
4558 \cs_generate_variant:Nn \tl_gremove_all:Nn { c }

```

10.7 Token list conditionals

`\tl_if_blank_p:n` TeX skips spaces when reading a non-delimited arguments. Thus, a *token list* is blank if and only if `\use_none:n` *token list* ? is empty. For performance reasons, we hard-code the emptiness test done in `\tl_if_empty:n(TF)`: convert to harmless characters with `\tl_to_str:n`, and then use `\if_meaning:w \q_nil ... \q_nil`. Note that converting to a string is done after reading the delimited argument for `\use_none:n`. The similar construction `\exp_after:wN \use_none:n \tl_to_str:n {token list} ?` would fail if the token list contains the control sequence `\`, while `\escapechar` is a space or is unprintable.

```

4559 \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
4560 { \tl_if_empty_return:o { \use_none:n #1 ? } }
4561 \cs_generate_variant:Nn \tl_if_blank_p:n { V }
4562 \cs_generate_variant:Nn \tl_if_blank:nT { V }
4563 \cs_generate_variant:Nn \tl_if_blank:nF { V }
4564 \cs_generate_variant:Nn \tl_if_blank:nTF { V }
4565 \cs_generate_variant:Nn \tl_if_blank_p:n { o }
4566 \cs_generate_variant:Nn \tl_if_blank:nT { o }
4567 \cs_generate_variant:Nn \tl_if_blank:nF { o }
4568 \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 ??.)

`\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
4569 \prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
4570 {
4571   \if_meaning:w #1 \c_empty_tl
4572   \prg_return_true:
4573   \else:
4574     \prg_return_false:
4575   \fi:
4576 }
4577 \cs_generate_variant:Nn \tl_if_empty_p:N { c }
4578 \cs_generate_variant:Nn \tl_if_empty:NT { c }
4579 \cs_generate_variant:Nn \tl_if_empty:NF { c }
4580 \cs_generate_variant:Nn \tl_if_empty:NTF { c }

```

(End definition for `\tl_if_empty:N` and `\tl_if_empty:c`. These functions are documented on page ??.)

`\tl_if_empty_p:n` It would be tempting to just use `\if_meaning:w \q_nil #1 \q_nil` as a test since this works really well. However, it fails on a token list starting with `\q_nil` of course but more troubling is the case where argument is a complete conditional such as `\if_true:`

```

\tl_if_empty_p:V
\tl_if_empty:nTF
\tl_if_empty:VTF

```

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... A safer route is to convert the entire token list into harmless characters first and then compare that. This way the test will even accept `\q_nil` as the first token.

```

4581 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
4582 {
4583   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil \tl_to_str:n {#1} \q_nil
4584   \prg_return_true:
4585   \else:
4586     \prg_return_false:
4587   \fi:
4588 }
4589 \cs_generate_variant:Nn \tl_if_empty_p:n { V }
4590 \cs_generate_variant:Nn \tl_if_empty:nTF { V }
4591 \cs_generate_variant:Nn \tl_if_empty:nT { V }
4592 \cs_generate_variant:Nn \tl_if_empty:nF { V }

```

(End definition for `\tl_if_empty:n` and `\tl_if_empty:V`. These functions are documented on page ??.)

`\tl_if_empty_p:o`
`\tl_if_empty:oTF`
`__tl_if_empty_return:o`

The auxiliary function `__tl_if_empty_return:o` is for use in conditionals on token lists, which mostly reduce to testing if a given token list is empty after applying a simple function to it. The test for emptiness is based on `\tl_if_empty:n(TF)`, but the expansion is hard-coded for efficiency, as this auxiliary function is used in many places. Note that this works because `\tl_to_str:n` expands tokens that follow until reading a catcode 1 (begin-group) token.

```

4593 \cs_new:Npn \__tl_if_empty_return:o #1
4594 {
4595   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
4596   \tl_to_str:n \exp_after:wN {#1} \q_nil
4597   \prg_return_true:
4598   \else:
4599     \prg_return_false:
4600   \fi:
4601 }
4602 \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
4603 { \__tl_if_empty_return:o {#1} }

```

(End definition for `\tl_if_empty:o`. These functions are documented on page ??.)

`\tl_if_eq_p:NN`
`\tl_if_eq_p:Nc`
`\tl_if_eq_p:cN`
`\tl_if_eq_p:cc`
`\tl_if_eq:NNTF`
`\tl_if_eq:NcTF`
`\tl_if_eq:cNTF`
`\tl_if_eq:ccTF`

Returns `\c_true_bool` if and only if the two token list variables are equal.

```

4604 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
4605 {
4606   \if_meaning:w #1 #2
4607   \prg_return_true:
4608   \else:
4609     \prg_return_false:
4610   \fi:
4611 }
4612 \cs_generate_variant:Nn \tl_if_eq_p:NN { Nc , c , cc }

```

```

4613 \cs_generate_variant:Nn \tl_if_eq:NNTF { Nc , c , cc }
4614 \cs_generate_variant:Nn \tl_if_eq:NNT { Nc , c , cc }
4615 \cs_generate_variant:Nn \tl_if_eq:NNF { Nc , c , cc }

```

(End definition for \tl_if_eq:NN and others. These functions are documented on page ??.)

\tl_if_eq:nnTF A simple store and compare routine.

```

\l__tl_internal_a_tl 4616 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
\l__tl_internal_b_tl 4617 {
4618   \group_begin:
4619     \tl_set:Nn \l__tl_internal_a_tl {#1}
4620     \tl_set:Nn \l__tl_internal_b_tl {#2}
4621     \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
4622     \group_end:
4623     \prg_return_true:
4624   \else:
4625     \group_end:
4626     \prg_return_false:
4627   \fi:
4628 }
4629 \tl_new:N \l__tl_internal_a_tl
4630 \tl_new:N \l__tl_internal_b_tl

```

(End definition for \tl_if_eq:nn. This function is documented on page ??.)

\tl_if_in:NnTF See \tl_if_in:nn(TF) for further comments. Here we simply expand the token list
\tl_if_in:cnTF variable and pass it to \tl_if_in:nn(TF).

```

4631 \cs_new_protected_nopar:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
4632 \cs_new_protected_nopar:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
4633 \cs_new_protected_nopar:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
4634 \cs_generate_variant:Nn \tl_if_in:NnT { c }
4635 \cs_generate_variant:Nn \tl_if_in:NnF { c }
4636 \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 ??.)

\tl_if_in:nnTF Once more, the test relies on \tl_to_str:n for robustness. The function __tl_tmp:w
\tl_if_in:VnTF removes tokens until the first occurrence of #2. If this does not appear in #1, then the
\tl_if_in:onTF final #2 is removed, leaving an empty token list. Otherwise some tokens remain, and the
\tl_if_in:noTF test is false. See \tl_if_empty:n(TF) for details on the emptiness test.

Special care is needed to treat correctly cases like \tl_if_in:nnTF {a state}{states}, where #1#2 contains #2 before the end. 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.

```

4637 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
4638 {
4639   \cs_set:Npn \__tl_tmp:w ##1 #2 { }
4640   \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
4641   { \prg_return_false: } { \prg_return_true: }
4642 }

```

```

4643 \cs_generate_variant:Nn \tl_if_in:nnT { V , o , no }
4644 \cs_generate_variant:Nn \tl_if_in:nnF { V , o , no }
4645 \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 ??.)

\tl_if_single_p:N Expand the token list and feed it to \tl_if_single:n.

```

\tl_if_single:NTF
4646 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
4647 \cs_new:Npn \tl_if_single:NT { \exp_args:No \tl_if_single:nT }
4648 \cs_new:Npn \tl_if_single:NF { \exp_args:No \tl_if_single:nF }
4649 \cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }

```

(End definition for \tl_if_single:N. These functions are documented on page 95.)

\tl_if_single_p:n A token list has exactly one item if it is a single token or a single brace group, surrounded by optional explicit spaces. The naive version of this test would do \use_none:n #1, and test if the result is empty. However, this will fail when the token list is empty. Furthermore, it does not allow optional trailing spaces.

\tl_if_single:n~~TF~~

```

4650 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
4651 { \__str_if_eq_x_return:nn { \exp_not:o { \use_none:nn #1 ?? } } {?} }

```

(End definition for \tl_if_single:n. These functions are documented on page 95.)

\tl_case:Nn Similar set up to \str_case:nn(TF) as described in l3basics.

```

\tl_case:cn
\tl_case:NnTF
\tl_case:cnTF
\__tl_case:nnTF
\__tl_case:Nw
\__tl_case_end:nw
4652 \cs_new:Npn \tl_case:Nn #1#2
4653 {
4654   \tex_romannumeral:D
4655   \__tl_case:NnTF #1 {#2} { } { }
4656 }
4657 \cs_new:Npn \tl_case:NnT #1#2#3
4658 {
4659   \tex_romannumeral:D
4660   \__tl_case:NnTF #1 {#2} {#3} { }
4661 }
4662 \cs_new:Npn \tl_case:NnF #1#2#3
4663 {
4664   \tex_romannumeral:D
4665   \__tl_case:NnTF #1 {#2} { } {#3}
4666 }
4667 \cs_new:Npn \tl_case:NnTF #1#2
4668 {
4669   \tex_romannumeral:D
4670   \__tl_case:NnTF #1 {#2}
4671 }
4672 \cs_new:Npn \__tl_case:NnTF #1#2#3#4
4673 { \__tl_case:Nw #1 #2 #1 { } \q_mark {#3} \q_mark {#4} \q_stop }
4674 \cs_new:Npn \__tl_case:Nw #1#2#3
4675 {
4676   \tl_if_eq:NNTF #1 #2
4677   { \__tl_case_end:nw {#3} }
4678   { \__tl_case:Nw #1 }
4679 }

```



```

4680 \cs_generate_variant:Nn \tl_case:Nn { c }
4681 \cs_generate_variant:Nn \tl_case:NnT { c }
4682 \cs_generate_variant:Nn \tl_case:NnF { c }
4683 \cs_generate_variant:Nn \tl_case:NnTF { c }
4684 \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`

`\tl_map_function:cN`

`__tl_map_function:Nn`

```

4685 \cs_new:Npn \tl_map_function:nN #1#2
4686 {
4687   \__tl_map_function:Nn #2 #1
4688   \q_recursion_tail
4689   \__prg_break_point:Nn \tl_map_break: { }
4690 }
4691 \cs_new_nopar:Npn \tl_map_function:NN
4692 { \exp_args:No \tl_map_function:nN }
4693 \cs_new:Npn \__tl_map_function:Nn #1#2
4694 {
4695   \__quark_if_recursion_tail_break:nN {#2} \tl_map_break:
4696   #1 {#2} \__tl_map_function:Nn #1
4697 }
4698 \cs_generate_variant:Nn \tl_map_function:NN { c }

```

(End definition for `\tl_map_function:nN`. This function is documented on page ??.)

`\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.

`\tl_map_inline:NN`

`\tl_map_inline:cn`

```

4699 \cs_new_protected:Npn \tl_map_inline:nn #1#2
4700 {
4701   \int_gincr:N \g__prg_map_int
4702   \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
4703   \exp_args:Nc \__tl_map_function:Nn
4704   { __prg_map_ \int_use:N \g__prg_map_int :w }
4705   #1 \q_recursion_tail
4706   \__prg_break_point:Nn \tl_map_break: { \int_gdecr:N \g__prg_map_int }
4707 }
4708 \cs_new_protected:Npn \tl_map_inline:NN
4709 { \exp_args:No \tl_map_inline:nn }
4710 \cs_generate_variant:Nn \tl_map_inline:NN { c }

```

(End definition for `\tl_map_inline:nn`. This function is documented on page ??.)

`\tl_map_variable:nNn` `\tl_map_variable:nNn` $\langle token\ list \rangle$ $\langle temp \rangle$ $\langle action \rangle$ assigns $\langle temp \rangle$ to each element and executes $\langle action \rangle$.

`\tl_map_variable:NNn`

`\tl_map_variable:cNn`

`__tl_map_variable:Nnn`

```

4711 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3

```

```

4712 {
4713   \_tl_map_variable:Nnn #2 {#3} #1
4714   \q_recursion_tail
4715   \_prg_break_point:Nn \tl_map_break: { }
4716 }
4717 \cs_new_protected_nopar:Npn \tl_map_variable:NNn
4718 { \exp_args:No \tl_map_variable:nNn }
4719 \cs_new_protected:Npn \_tl_map_variable:Nnn #1#2#3
4720 {
4721   \tl_set:Nn #1 {#3}
4722   \_quark_if_recursion_tail_break:NN #1 \tl_map_break:
4723   \use:n {#2}
4724   \_tl_map_variable:Nnn #1 {#2}
4725 }
4726 \cs_generate_variant:Nn \tl_map_variable:NNn { c }

```

(End definition for `\tl_map_variable:nNn`. This function is documented on page ??.)

`\tl_map_break:` The break statements use the general `_prg_map_break:Nn`.
`\tl_map_break:n`

```

4727 \cs_new_nopar:Npn \tl_map_break:
4728 { \_prg_map_break:Nn \tl_map_break: { } }
4729 \cs_new_nopar:Npn \tl_map_break:n
4730 { \_prg_map_break:Nn \tl_map_break: }

```

(End definition for `\tl_map_break:.` This function is documented on page 97.)

10.9 Using token lists

`\tl_to_str:n` Another name for a primitive.

```

4731 \cs_new_eq:NN \tl_to_str:n \etex_detokenize:D

```

(End definition for `\tl_to_str:n`. This function is documented on page 98.)

`\tl_to_str:N` These functions return the replacement text of a token list as a string.

`\tl_to_str:c`

```

4732 \cs_new:Npn \tl_to_str:N #1 { \etex_detokenize:D \exp_after:wN {#1} }
4733 \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 ??.)

`\tl_use:N` Token lists which are simply not defined will give a clear T_EX error here. No such luck
`\tl_use:c` for ones equal to `\scan_stop:` so instead a test is made and if there is an issue an error is forced.

```

4734 \cs_new:Npn \tl_use:N #1
4735 {
4736   \tl_if_exist:NTF #1 {#1}
4737   { \_msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#1} }
4738 }
4739 \cs_generate_variant:Nn \tl_use:N { c }

```

(End definition for `\tl_use:N` and `\tl_use:c`. These functions are documented on page ??.)

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 to ensure it works on an empty list.

```

\tl_count:N      4740 \cs_new:Npn \tl_count:n #1
\tl_count:c      4741 {
\tl_count:N      4742   \int_eval:n
\tl_count:c      4743   { 0 \tl_map_function:nN {#1} \__tl_count:n }
\tl_count:N      4744 }
\tl_count:c      4745 \cs_new:Npn \tl_count:N #1
\tl_count:N      4746 {
\tl_count:c      4747   \int_eval:n
\tl_count:N      4748   { 0 \tl_map_function:NN #1 \__tl_count:n }
\tl_count:N      4749 }
\tl_count:c      4750 \cs_new:Npn \__tl_count:n #1 { + \c_one }
\tl_count:c      4751 \cs_generate_variant:Nn \tl_count:n { V , o }
\tl_count:c      4752 \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 ??.)

\tl_reverse_items:n Reversal of a token list is done by taking one item at a time and putting it after \q_stop.

```

\tl_reverse_items:nwNwn 4753 \cs_new:Npn \tl_reverse_items:n #1
\tl_reverse_items:wn     4754 {
\tl_reverse_items:nwNwn 4755   \__tl_reverse_items:nwNwn #1 ?
\tl_reverse_items:wn     4756   \q_mark \__tl_reverse_items:nwNwn
\tl_reverse_items:wn     4757   \q_mark \__tl_reverse_items:wn
\tl_reverse_items:wn     4758   \q_stop { }
\tl_reverse_items:wn     4759 }
\tl_reverse_items:nwNwn 4760 \cs_new:Npn \__tl_reverse_items:nwNwn #1 #2 \q_mark #3 #4 \q_stop #5
\tl_reverse_items:wn     4761 {
\tl_reverse_items:wn     4762   #3 #2
\tl_reverse_items:wn     4763   \q_mark \__tl_reverse_items:nwNwn
\tl_reverse_items:wn     4764   \q_mark \__tl_reverse_items:wn
\tl_reverse_items:wn     4765   \q_stop { {#1} #5 }
\tl_reverse_items:wn     4766 }
\tl_reverse_items:wn     4767 \cs_new:Npn \__tl_reverse_items:wn #1 \q_stop #2
\tl_reverse_items:wn     4768 { \exp_not:o { \use_none:nn #2 } }

```

(End definition for \tl_reverse_items:n. This function is documented on page 99.)

\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.

```

\tl_trim_spaces:n      4769 \cs_new:Npn \tl_trim_spaces:n #1
\tl_trim_spaces:N      4770 { \__tl_trim_spaces:nn { \q_mark #1 } \exp_not:o }
\tl_trim_spaces:c      4771 \cs_new_protected:Npn \tl_trim_spaces:N #1
\tl_gtrim_spaces:N      4772 { \tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }

```

```

4773 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
4774 { \tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
4775 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
4776 \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }

```

(End definition for `\tl_trim_spaces:n`. This function is documented on page ??.)

`__tl_trim_spaces:nn`

```

\__tl_trim_spaces_auxi:w
\__tl_trim_spaces_auxii:w \__tl_trim_spaces_auxiii:w
\__tl_trim_spaces_auxiv:w

```

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)*.

```

4777 \cs_set:Npn \__tl_tmp:w #1
4778 {
4779   \cs_new:Npn \__tl_trim_spaces:nn ##1
4780   {
4781     \__tl_trim_spaces_auxi:w
4782     ##1
4783     \q_nil
4784     \q_mark #1 { }
4785     \q_mark \__tl_trim_spaces_auxii:w
4786     \__tl_trim_spaces_auxiii:w
4787     #1 \q_nil
4788     \__tl_trim_spaces_auxiv:w
4789     \q_stop
4790   }
4791   \cs_new:Npn \__tl_trim_spaces_auxi:w ##1 \q_mark #1 ##2 \q_mark ##3
4792   {
4793     ##3
4794     \__tl_trim_spaces_auxi:w
4795     \q_mark
4796     ##2
4797     \q_mark #1 {##1}
4798   }
4799   \cs_new:Npn \__tl_trim_spaces_auxii:w
4800   \__tl_trim_spaces_auxi:w \q_mark \q_mark ##1
4801   {
4802     \__tl_trim_spaces_auxiii:w
4803     ##1
4804   }
4805   \cs_new:Npn \__tl_trim_spaces_auxiii:w ##1 #1 \q_nil ##2
4806   {
4807     ##2
4808     ##1 \q_nil
4809     \__tl_trim_spaces_auxiii:w

```

```

4810     }
4811     \cs_new:Npn \__tl_trim_spaces_auxiv:w ##1 \q_nil ##2 \q_stop ##3
4812     { ##3 { \use_none:n ##1 } }
4813   }
4814   \__tl_tmp:w { ~ }

```

(End definition for `__tl_trim_spaces:nm`. This function is documented on page 103.)

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 proceeded by `\romannumeral` and ends by producing `\c_zero` 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`.

```

4815 \cs_new:Npn \__tl_act:NNNnn #1#2#3#4#5
4816 {
4817   \group_align_safe_begin:
4818   \__tl_act_loop:w #5 \q___tl_act_mark \q___tl_act_stop
4819   {#4} #1 #2 #3
4820   \__tl_act_result:n { }
4821 }

```

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 `\c_zero` and the result in the input stream, to terminate the expansion of `\romannumeral`. 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:wnnNNN` gobble the space.

```

4822 \cs_new:Npn \__tl_act_loop:w #1 \q___tl_act_stop
4823 {
4824   \tl_if_head_is_N_type:nTF {#1}
4825   { \__tl_act_normal:NwnNNN }
4826   {
4827     \tl_if_head_is_group:nTF {#1}
4828     { \__tl_act_group:nwnNNN }
4829     { \__tl_act_space:wnnNNN }
4830   }
4831   #1 \q___tl_act_stop
4832 }
4833 \cs_new:Npn \__tl_act_normal:NwnNNN #1 #2 \q___tl_act_stop #3#4
4834 {

```

```

4835 \if_meaning:w \q___tl_act_mark #1
4836 \exp_after:wN \__tl_act_end:wn
4837 \fi:
4838 #4 {#3} #1
4839 \__tl_act_loop:w #2 \q___tl_act_stop
4840 {#3} #4
4841 }
4842 \cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
4843 { \group_align_safe_end: \c_zero #2 }
4844 \cs_new:Npn \__tl_act_group:nwnNNN #1 #2 \q___tl_act_stop #3#4#5
4845 {
4846   #5 {#3} {#1}
4847   \__tl_act_loop:w #2 \q___tl_act_stop
4848   {#3} #4 #5
4849 }
4850 \exp_last_unbraced:NN
4851 \cs_new:Npn \__tl_act_space:wNnnN \c_space_tl #1 \q___tl_act_stop #2#3#4#5
4852 {
4853   #5 {#2}
4854   \__tl_act_loop:w #1 \q___tl_act_stop
4855   {#2} #3 #4 #5
4856 }

```

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.

```

4857 \cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
4858 { #2 \__tl_act_result:n { #3 #1 } }
4859 \cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
4860 { #2 \__tl_act_result:n { #1 #3 } }

```

(End definition for `__tl_act:NNNnn`. This function is documented on page ??.)

<pre> __tl_reverse_normal:nN __tl_reverse_group_preserve:nn __tl_reverse_space:n </pre>	<p>The goal here is to reverse without losing spaces nor braces. This is done using the general internal function <code>__tl_act:NNNnn</code>. Spaces and “normal” tokens are output on the left of the current output. Grouped tokens are output to the left but without any reversal within the group. All of the internal functions here drop one argument: this is needed by <code>__tl_act:NNNnn</code> when changing case (to record which direction the change is in), but not when reversing the tokens.</p>
--	--

```

4861 \cs_new:Npn \tl_reverse:n #1
4862 {
4863   \etex_unexpanded:D \exp_after:wN
4864   {
4865     \tex_romannumeral:D
4866     \__tl_act:NNNnn
4867     \__tl_reverse_normal:nN
4868     \__tl_reverse_group_preserve:nn
4869     \__tl_reverse_space:n
4870     { }
4871     {#1}
4872   }

```

```

4873 }
4874 \cs_generate_variant:Nn \tl_reverse:n { o , V }
4875 \cs_new:Npn \__tl_reverse_normal:nN #1#2
4876 { \__tl_act_reverse_output:n {#2} }
4877 \cs_new:Npn \__tl_reverse_group_preserve:nn #1#2
4878 { \__tl_act_reverse_output:n { {#2} } }
4879 \cs_new:Npn \__tl_reverse_space:n #1
4880 { \__tl_act_reverse_output:n { ~ } }

```

(End definition for `\tl_reverse:n`, `\tl_reverse:o`, and `\tl_reverse:V`. These functions are documented on page ??.)

`\tl_reverse:N` This reverses the list, leaving `\exp_stop_f:` in front, which stops the f-expansion.

```

\tl_reverse:c 4881 \cs_new_protected:Npn \tl_reverse:N #1
\tl_greverse:N 4882 { \tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
\tl_greverse:c 4883 \cs_new_protected:Npn \tl_greverse:N #1
4884 { \tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
4885 \cs_generate_variant:Nn \tl_reverse:N { c }
4886 \cs_generate_variant:Nn \tl_greverse:N { c }

```

(End definition for `\tl_reverse:N` and others. These functions are documented on page ??.)

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 this is consistent with for example mapping to a list. The empty brace groups in `\tl_head:n` ensure that a blank argument gives an empty result. The result is returned within the `\unexpanded` primitive. The approach here is to use `\if_false:` to allow us to use `}` as the closing delimiter: this is the only safe choice, as any other token would not be able to parse it's own code. Using a marker, we can see if what we are grabbing is exactly the marker, or there is anything else to deal with. If there is, there is a loop. If not, tidy up and leave the item in the output stream. More detail in <http://tex.stackexchange.com/a/70168>.

```

\tl_head:n 4887 \cs_new:Npn \tl_head:n #1
\tl_head:V 4888 {
\tl_head:v 4889 \etex_unexpanded:D
\tl_head:f 4890 \if_false: { \fi: \__tl_head_auxi:nw #1 { } \q_stop }
4891 }
4892 \cs_new:Npn \__tl_head_auxi:nw #1#2 \q_stop
4893 { \exp_after:wN \__tl_head_auxii:nw \exp_after:wN { \if_false: } \fi: {#1} }
4894 \cs_new:Npn \__tl_head_auxii:nw #1
4895 {
4896 \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
4897 \tl_to_str:n \exp_after:wN { \use_none:n #1 } \q_nil
4898 \exp_after:wN \use_i:nn
4899 \else:
4900 \exp_after:wN \use_ii:nn
4901 \fi:
4902 {#1}
4903 { \if_false: { \fi: \__tl_head_auxi:nw #1 } }

```

```

4904 }
4905 \cs_generate_variant:Nn \tl_head:n { V , v , f }
4906 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
4907 \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 `\etex_unexpanded:D`. While we could optimise the test here, this would leave some tokens “banned” in the input, which we do not have with this definition.

```

4908 \cs_new:Npn \tl_tail:n #1
4909 {
4910   \etex_unexpanded:D
4911   \tl_if_blank:nTF {#1}
4912   { { } }
4913   { \exp_after:wN { \use_none:n #1 } }
4914 }
4915 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
4916 \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 ??.)

\str_head:n After `\tl_to_str:n`, we have a list of character tokens, all with category code 12, except the space, which has category code 10. Directly using `\tl_head:w` would thus lose leading spaces. Instead, we take an argument delimited by an explicit space, and then only use `\tl_head:w`. If the string started with a space, then the argument of `__str_head:w` is empty, and the function correctly returns a space character. Otherwise, it returns the first token of `#1`, which is the first token of the string. If the string is empty, we return an empty result.

\str_tail:n
`__str_head:w`
`__str_tail:w`

To remove the first character of `\tl_to_str:n {#1}`, we test it using `\if_charcode:w \scan_stop:`, always `false` for characters. If the argument was non-empty, then `__str_tail:w` returns everything until the first X (with category code letter, no risk of confusing with the user input). If the argument was empty, the first X is taken by `\if_charcode:w`, and nothing is returned. We use X as a *marker*, rather than a quark because the test `\if_charcode:w \scan_stop: <marker>` has to be `false`.

```

4917 \cs_new:Npn \str_head:n #1
4918 {
4919   \exp_after:wN \__str_head:w
4920   \tl_to_str:n {#1}
4921   { { } } ~ \q_stop
4922 }
4923 \cs_new:Npn \__str_head:w #1 ~ %
4924 { \tl_head:w #1 { ~ } }

```



```

4925 \cs_new:Npn \str_tail:n #1
4926 {
4927   \exp_after:wN \__str_tail:w
4928   \reverse_if:N \if_charcode:w
4929   \scan_stop: \tl_to_str:n {#1} X X \q_stop
4930 }
4931 \cs_new:Npn \__str_tail:w #1 X #2 \q_stop { \fi: #1 }

```

(End definition for `\str_head:n` and `\str_tail:n`. These functions are documented on page 101.)

`\tl_if_head_eq_meaning_p:nN` Accessing the first token of a token list is tricky in three cases: when it has category code 1 (begin-group token), when it is an explicit space, with category code 10 and character code 32, or when the token list is empty (obviously).

`\tl_if_head_eq_meaning:nNTF` Forgetting temporarily about this issue we would use the following test in `\tl_if_head_eq_charcode_p:nN`. Here, `\tl_head:w` yields the first token of the token list, then `\tl_if_head_eq_charcode_p:nN` passed to `\exp_not:N`.

```

\if_charcode:w
\exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop
\exp_not:N #2

```

The two first special cases are detected by testing if the token list starts with an N-type token (the extra ? sends empty token lists to the true branch of this test). In those cases, the first token is a character, and since we only care about its character code, we can use `\str_head:n` to access it (this works even if it is a space character). An empty argument will result in `\tl_head:w` leaving two tokens: ? which is taken in the `\if_charcode:w` test, and `\use_none:nn`, which ensures that `\prg_return_false:` is returned regardless of whether the charcode test was true or false.

```

4932 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
4933 {
4934   \if_charcode:w
4935   \exp_not:N #2
4936   \tl_if_head_is_N_type:nTF { #1 ? }
4937   {
4938     \exp_after:wN \exp_not:N
4939     \tl_head:w #1 { ? \use_none:nn } \q_stop
4940   }
4941   { \str_head:n {#1} }
4942   \prg_return_true:
4943   \else:
4944     \prg_return_false:
4945   \fi:
4946 }
4947 \cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN { f }
4948 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF { f }
4949 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT { f }
4950 \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.

```

4951 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
4952 {
4953   \if_catcode:w
4954     \exp_not:N #2
4955     \tl_if_head_is_N_type:nTF { #1 ? }
4956     {
4957       \exp_after:wN \exp_not:N
4958       \tl_head:w #1 { ? \use_none:nn } \q_stop
4959     }
4960     {
4961       \tl_if_head_is_group:nTF {#1}
4962       { \c_group_begin_token }
4963       { \c_space_token }
4964     }
4965     \prg_return_true:
4966   \else:
4967     \prg_return_false:
4968   \fi:
4969 }

```

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.

```

4970 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
4971 {
4972   \tl_if_head_is_N_type:nTF { #1 ? }
4973   { \_tl_if_head_eq_meaning_normal:nN }
4974   { \_tl_if_head_eq_meaning_special:nN }
4975   {#1} #2
4976 }
4977 \cs_new:Npn \_tl_if_head_eq_meaning_normal:nN #1 #2
4978 {
4979   \exp_after:wN \if_meaning:w
4980   \tl_head:w #1 { ?? \use_none:nnn } \q_stop #2
4981   \prg_return_true:
4982   \else:
4983     \prg_return_false:
4984   \fi:
4985 }
4986 \cs_new:Npn \_tl_if_head_eq_meaning_special:nN #1 #2
4987 {

```

```

4988 \if_charcode:w \str_head:n {#1} \exp_not:N #2
4989 \exp_after:wN \use:n
4990 \else:
4991 \prg_return_false:
4992 \exp_after:wN \use_none:n
4993 \fi:
4994 {
4995 \if_catcode:w \exp_not:N #2
4996 \tl_if_head_is_group:nTF {#1}
4997 { \c_group_begin_token }
4998 { \c_space_token }
4999 \prg_return_true:
5000 \else:
5001 \prg_return_false:
5002 \fi:
5003 }
5004 }

```

(End definition for `\tl_if_head_eq_meaning:nN`. These functions are documented on page 101.)

`\tl_if_head_is_N_type_p:n`
`\tl_if_head_is_N_type:nTF`

The first token of a token list can be either an N-type argument, a begin-group token (catcode 1), or an explicit space token (catcode 10 and charcode 32). The latter two cases are characterized by the fact that `\use:n` removes some tokens from `#1`, hence changing its string representation (no token can have an empty string representation). The extra brace group covers the case of an empty argument, whose head is not “normal”.

```

5005 \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
5006 {
5007 \__str_if_eq_x_return:nn
5008 { \exp_not:o { \use:n #1 { } } }
5009 { \exp_not:n { #1 { } } }
5010 }

```

(End definition for `\tl_if_head_is_N_type:n`. These functions are documented on page 102.)

`\tl_if_head_is_group_p:n`
`\tl_if_head_is_group:nTF`

Pass the first token of `#1` through `\token_to_str:N`, then check for the brace balance. The extra ? caters for an empty argument.⁵

```

5011 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
5012 {
5013 \if_catcode:w *
5014 \exp_after:wN \use_none:n
5015 \exp_after:wN {
5016 \exp_after:wN {
5017 \token_to_str:N #1 ?
5018 }
5019 }
5020 *
5021 \prg_return_false:

```

⁵Bruno: 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.

```

5022     \else:
5023       \prg_return_true:
5024     \fi:
5025   }

```

(End definition for `\tl_if_head_is_group:n`. These functions are documented on page 102.)

```

\tl_if_head_is_space_p:n
\tl_if_head_is_space:nTF
\__tl_if_head_is_space:w

```

If the first token of the token list is an explicit space, i.e., a character token with character code 32 and category code 10, then this test will be `true`. It is `false` if the token list is empty, if the first token is an implicit space token, such as `\c_space_token`, or any token other than an explicit space. The slightly convoluted approach with `\romannumeral` ensures that each expansion step gives a balanced token list.

```

5026 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
5027 {
5028   \tex_romannumeral:D \if_false: { \fi:
5029     \__tl_if_head_is_space:w ? #1 ? ~ }
5030 }
5031 \cs_new:Npn \__tl_if_head_is_space:w #1 ~
5032 {
5033   \tl_if_empty:oTF { \use_none:n #1 }
5034   { \exp_after:wN \c_zero \exp_after:wN \prg_return_true: }
5035   { \exp_after:wN \c_zero \exp_after:wN \prg_return_false: }
5036   \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
5037 }

```

(End definition for `\tl_if_head_is_space:n`. These functions are documented on page 102.)

10.13 Viewing token lists

`\tl_show:N` Showing token list variables is done after checking that the variable is defined (see `__kernel_register_show:N`).

```

5038 \cs_new_protected:Npn \tl_show:N #1
5039 {
5040   \tl_if_exist:NTF #1
5041   { \cs_show:N #1 }
5042   {
5043     \__msg_kernel_error:nmx { kernel } { variable-not-defined }
5044     { \token_to_str:N #1 }
5045   }
5046 }
5047 \cs_generate_variant:Nn \tl_show:N { c }

```

(End definition for `\tl_show:N` and `\tl_show:c`. These functions are documented on page ??.)

`\tl_show:n` The `__msg_show_variable:n` internal function performs line-wrapping, removes a leading `>`, then 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.

```

5048 \cs_new_protected:Npn \tl_show:n #1
5049 { \__msg_show_variable:n { > ~ \tl_to_str:n {#1} } }

```

(End definition for `\tl_show:n`. This function is documented on page 102.)

10.14 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.

```
5050 \tl_new:N \g_tmpa_tl
5051 \tl_new:N \g_tmpb_tl
```

(End definition for `\g_tmpa_tl` and `\g_tmpb_tl`. These variables are documented on page 103.)

`\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.

```
5052 \tl_new:N \l_tmpa_tl
5053 \tl_new:N \l_tmpb_tl
```

(End definition for `\l_tmpa_tl` and `\l_tmpb_tl`. These variables are documented on page 103.)

10.15 Deprecated functions

`\tl_case:Nnn` Deprecated 2013-07-15.

`\tl_case:cnn`

```
5054 \cs_new_eq:NN \tl_case:Nnn \tl_case:NnF
5055 \cs_new_eq:NN \tl_case:cnn \tl_case:cnF
```

(End definition for `\tl_case:Nnn` and `\tl_case:cnn`. These functions are documented on page ??.)

```
5056 \</initex | package>
```

11 l3seq implementation

The following test files are used for this code: `m3seq002`, `m3seq003`.

```
5057 <*initex | package>
5058 <@@=seq>
5059 <*package>
5060 \ProvidesExplPackage
5061   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
5062   \__expl_package_check:
5063 </package>
```

A sequence is a control sequence whose top-level expansion is of the form “`\s__seq __seq_item:n {<item1>} ... __seq_item:n {<itemn>} \s_obj_end`”, with a leading scan mark, *n* items of the same form, and a trailing scan mark. 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 112.)

__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.

```

5064 \cs_new:Npn \__seq_item:n
5065 {
5066   \_msg_kernel_expandable_error:nn { kernel } { misused-sequence }
5067   \use_none:n
5068 }

```

(End definition for __seq_item:n.)

\l__seq_internal_a_tl Scratch space for various internal uses.

```

\l__seq_internal_b_tl
5069 \tl_new:N \l__seq_internal_a_tl
5070 \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 ??.)

\c_empty_seq A sequence with no item, following the structure mentioned above.

```

5071 \tl_const:Nn \c_empty_seq { \s__seq \s_obj_end }

```

(End definition for \c_empty_seq. This variable is documented on page 112.)

11.1 Allocation and initialisation

\seq_new:N Sequences are initialized to \c_empty_seq.

```

\seq_new:c
5072 \cs_new_protected:Npn \seq_new:N #1
5073 {
5074   \__chk_if_free_cs:N #1
5075   \cs_gset_eq:NN #1 \c_empty_seq
5076 }
5077 \cs_generate_variant:Nn \seq_new:N { c }

```

(End definition for \seq_new:N and \seq_new:c. These functions are documented on page ??.)

\seq_clear:N Clearing a sequence is similar to setting it equal to the empty one.

```

\seq_clear:c
5078 \cs_new_protected:Npn \seq_clear:N #1
\seq_gclear:N
5079 { \seq_set_eq:NN #1 \c_empty_seq }
\seq_gclear:c
5080 \cs_generate_variant:Nn \seq_clear:N { c }
5081 \cs_new_protected:Npn \seq_gclear:N #1
5082 { \seq_gset_eq:NN #1 \c_empty_seq }
5083 \cs_generate_variant:Nn \seq_gclear:N { c }

```

(End definition for \seq_clear:N and \seq_clear:c. These functions are documented on page ??.)

\seq_clear_new:N Once again we copy code from the token list functions.

```

\seq_clear_new:c
5084 \cs_new_protected:Npn \seq_clear_new:N #1
\seq_gclear_new:N
5085 { \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }
\seq_gclear_new:c
5086 \cs_generate_variant:Nn \seq_clear_new:N { c }
5087 \cs_new_protected:Npn \seq_gclear_new:N #1
5088 { \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }
5089 \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 ??.)

\seq_set_eq:NN Copying a sequence is the same as copying the underlying token list.

```

\seq_set_eq:cN 5090 \cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN
\seq_set_eq:Nc 5091 \cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc
\seq_set_eq:cc 5092 \cs_new_eq:NN \seq_set_eq:cN \tl_set_eq:cN
\seq_gset_eq:NN 5093 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
\seq_gset_eq:cN 5094 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
\seq_gset_eq:Nc 5095 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
\seq_gset_eq:cN 5096 \cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
\seq_gset_eq:cc 5097 \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 ??.)

\seq_set_split:Nnn When the separator is empty, everything is very simple, just map `__seq_wrap_item:n` through the items of the last argument. For non-trivial separators, the goal is to split a given token list at the marker, strip spaces from each item, and remove one set of outer braces if after removing leading and trailing spaces the item is enclosed within braces. After `\tl_replace_all:Nnn`, the token list `\l__seq_internal_a_tl` is a repetition of the pattern `__seq_set_split_auxi:w \prg_do_nothing: <item with spaces>` `__seq_set_split_end:.` Then, x-expansion causes `__seq_set_split_auxi:w` to trim spaces, and leaves its result as `__seq_set_split_auxii:w <trimmed item>` `__seq_set_split_end:.` This is then converted to the `l3seq` internal structure by another x-expansion. In the first step, we insert `\prg_do_nothing:` to avoid losing braces too early: that would cause space trimming to act within those lost braces. The second step is solely there to strip braces which are outermost after space trimming.

```

5098 \cs_new_protected_nopar:Npn \seq_set_split:Nnn
5099 { \__seq_set_split:Nnnn \tl_set:Nx }
5100 \cs_new_protected_nopar:Npn \seq_gset_split:Nnn
5101 { \__seq_set_split:Nnnn \tl_gset:Nx }
5102 \cs_new_protected:Npn \__seq_set_split:Nnnn #1#2#3#4
5103 {
5104   \tl_if_empty:nTF {#3}
5105   {
5106     \tl_set:Nn \l__seq_internal_a_tl
5107     { \tl_map_function:nN {#4} \__seq_wrap_item:n }
5108   }
5109   {
5110     \tl_set:Nn \l__seq_internal_a_tl
5111     {
5112       \__seq_set_split_auxi:w \prg_do_nothing:
5113       #4
5114       \__seq_set_split_end:
5115     }
5116     \tl_replace_all:Nnn \l__seq_internal_a_tl { #3 }
5117     {
5118       \__seq_set_split_end:
5119       \__seq_set_split_auxi:w \prg_do_nothing:
5120     }
5121     \tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
5122   }

```

```

5123     #1 #2 { \s__seq \l__seq_internal_a_tl \s_obj_end }
5124   }
5125   \cs_new:Npn \__seq_set_split_auxi:w #1 \__seq_set_split_end:
5126   {
5127     \exp_not:N \__seq_set_split_auxii:w
5128     \exp_args:No \tl_trim_spaces:n {#1}
5129     \exp_not:N \__seq_set_split_end:
5130   }
5131   \cs_new:Npn \__seq_set_split_auxii:w #1 \__seq_set_split_end:
5132   { \__seq_wrap_item:n {#1} }
5133   \cs_generate_variant:Nn \seq_set_split:Nnn { NnV }
5134   \cs_generate_variant:Nn \seq_gset_split:Nnn { NnV }

```

(End definition for \seq_set_split:Nnn and others. These functions are documented on page ??.)

\seq_concat:NNN When concatenating sequences, one must take care of the trailing \s_obj_end and leading \s__seq. The second argument of __seq_concat:NN is \s__seq, and __seq_concat:w removes \s_obj_end by a delimited approach. The result starts with \s__seq (of the first sequence), which stops f-expansion.

```

\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc
\__seq_concat:NN
\__seq_concat:w
5135   \cs_new_protected:Npn \seq_concat:NNN #1#2#3
5136   { \tl_set:Nf #1 { \exp_last_unbraced:NNo \__seq_concat:NN #2 #3 } }
5137   \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
5138   { \tl_gset:Nf #1 { \exp_last_unbraced:NNo \__seq_concat:NN #2 #3 } }
5139   \cs_new:Npn \__seq_concat:NN #1#2 { \exp_after:wN \__seq_concat:w #1 }
5140   \cs_new:Npn \__seq_concat:w #1 \s_obj_end {#1}
5141   \cs_generate_variant:Nn \seq_concat:NNN { ccc }
5142   \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End definition for \seq_concat:NNN and \seq_concat:ccc. These functions are documented on page ??.)

\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
5143   \prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N { TF , T , F , p }
5144   \prg_new_eq_conditional:NNn \seq_if_exist:c \cs_if_exist:c { TF , T , F , p }

```

(End definition for \seq_if_exist:N and \seq_if_exist:c. These functions are documented on page ??.)

11.2 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
5145   \cs_new_protected:Npn \seq_put_left:Nn #1#2
5146   {
5147     \tl_set:Nx #1
5148     {
5149       \exp_not:n { \s__seq \__seq_item:n {#2} }
5150       \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
5151     }
5152   }
5153   \cs_new_protected:Npn \seq_gput_left:Nn #1#2

```

```

\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

```



```

5154 {
5155   \tl_gset:Nx #1
5156   {
5157     \exp_not:n { \s__seq \__seq_item:n {#2} }
5158     \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
5159   }
5160 }
5161 \cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
5162 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
5163 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
5164 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
5165 \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 ??.)

\seq_put_right:Nn When adding to the right of a sequence, remove \s_obj_end. This is done by __seq_put_right_aux:w, which takes its result through \exp_not:n.

```

5166 \cs_new_protected:Npn \seq_put_right:Nn #1#2
5167 {
5168   \tl_set:Nx #1
5169   {
5170     \exp_after:wN \__seq_put_right_aux:w #1
5171     \exp_not:n { \__seq_item:n {#2} \s_obj_end }
5172   }
5173 }
5174 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
5175 {
5176   \tl_gset:Nx #1
5177   {
5178     \exp_after:wN \__seq_put_right_aux:w #1
5179     \exp_not:n { \__seq_item:n {#2} \s_obj_end }
5180   }
5181 }
5182 \cs_new:Npn \__seq_put_right_aux:w #1 \s_obj_end { \exp_not:n {#1} }
5183 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
5184 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
5185 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
5186 \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 ??.)

11.3 Modifying sequences

__seq_wrap_item:n This function converts its argument to a proper sequence item in an x-expansion context.

```

5187 \cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }

```

(End definition for __seq_wrap_item:n.)

\l__seq_remove_seq An internal sequence for the removal routines.

```

5188 \seq_new:N \l__seq_remove_seq

```

(End definition for \l__seq_remove_seq. This variable is documented on page ??.)

```

\seq_remove_duplicates:N
\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
__seq_remove_duplicates:NN

```

Removing duplicates means making a new list then copying it.

```

5189 \cs_new_protected:Npn \seq_remove_duplicates:N
5190 { __seq_remove_duplicates:NN \seq_set_eq:NN }
5191 \cs_new_protected:Npn \seq_gremove_duplicates:N
5192 { __seq_remove_duplicates:NN \seq_gset_eq:NN }
5193 \cs_new_protected:Npn __seq_remove_duplicates:NN #1#2
5194 {
5195   \seq_clear:N \l__seq_remove_seq
5196   \seq_map_inline:Nn #2
5197   {
5198     \seq_if_in:NnF \l__seq_remove_seq {##1}
5199     { \seq_put_right:Nn \l__seq_remove_seq {##1} }
5200   }
5201   #1 #2 \l__seq_remove_seq
5202 }
5203 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
5204 \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 ??.)

```

\seq_remove_all:Nn
\seq_remove_all:cn
\seq_gremove_all:Nn
\seq_gremove_all:cn
__seq_remove_all_aux:NNn

```

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_aux: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.

```

5205 \cs_new_protected:Npn \seq_remove_all:Nn
5206 { __seq_remove_all_aux:NNn \tl_set:Nx }
5207 \cs_new_protected:Npn \seq_gremove_all:Nn
5208 { __seq_remove_all_aux:NNn \tl_gset:Nx }
5209 \cs_new_protected:Npn __seq_remove_all_aux:NNn #1#2#3
5210 {
5211   __seq_push_item_def:n
5212   {
5213     \str_if_eq:nnT {##1} {#3}
5214     {
5215       \if_false: { \fi: }
5216       \tl_set:Nn \l__seq_internal_b_tl {##1}
5217       #1 #2
5218       { \if_false: } \fi:
5219       \exp_not:o {#2}
5220       \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
5221       { \use_none:nn }
5222     }
5223     __seq_wrap_item:n {##1}

```

```

5224     }
5225     \tl_set:Nn \l__seq_internal_a_tl {#3}
5226     #1 #2 {#2}
5227     \__seq_pop_item_def:
5228   }
5229   \cs_generate_variant:Nn \seq_remove_all:Nn { c }
5230   \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 ??.)

11.4 Sequence conditionals

`\seq_if_empty_p:N` Similar to token lists, we compare with the empty sequence.

```

\seq_if_empty_p:c 5231 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
\seq_if_empty:NTF 5232 {
\seq_if_empty:cTF 5233   \if_meaning:w #1 \c_empty_seq
5234   \prg_return_true:
5235   \else:
5236   \prg_return_false:
5237   \fi:
5238 }
5239 \cs_generate_variant:Nn \seq_if_empty_p:N { c }
5240 \cs_generate_variant:Nn \seq_if_empty:NT { c }
5241 \cs_generate_variant:Nn \seq_if_empty:NF { c }
5242 \cs_generate_variant:Nn \seq_if_empty:NTF { c }

```

(End definition for `\seq_if_empty:N` and `\seq_if_empty:c`. These functions are documented on page ??.)

`\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.

```

\seq_if_in:NvTF 5243 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
\seq_if_in:NoTF 5244 { T , F , TF }
\seq_if_in:NxTF 5245 {
\seq_if_in:cnTF 5246   \group_begin:
\seq_if_in:cVTF 5247   \tl_set:Nn \l__seq_internal_a_tl {#2}
\seq_if_in:cvTF 5248   \cs_set_protected:Npn \__seq_item:n ##1
\seq_if_in:coTF 5249   {
\seq_if_in:cxTF 5250     \tl_set:Nn \l__seq_internal_b_tl {##1}
5251     \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
5252     \exp_after:wN \__seq_if_in:
5253     \fi:
5254   }
5255   #1
5256   \group_end:
5257   \prg_return_false:
5258   \__prg_break_point:

```

```

5259 }
5260 \cs_new_nopar:Npn \__seq_if_in:
5261 { \__prg_break:n { \group_end: \prg_return_true: } }
5262 \cs_generate_variant:Nn \seq_if_in:NnT { NV , Nv , No , Nx }
5263 \cs_generate_variant:Nn \seq_if_in:NnT { c , cV , cv , co , cx }
5264 \cs_generate_variant:Nn \seq_if_in:NnF { NV , Nv , No , Nx }
5265 \cs_generate_variant:Nn \seq_if_in:NnF { c , cV , cv , co , cx }
5266 \cs_generate_variant:Nn \seq_if_in:NnTF { NV , Nv , No , Nx }
5267 \cs_generate_variant:Nn \seq_if_in:NnTF { c , cV , cv , co , cx }

```

(End definition for \seq_if_in:Nn and others. These functions are documented on page ??.)

11.5 Recovering data from sequences

__seq_pop:NNNN The two pop functions share their emptiness tests. We also use a common emptiness test
 __seq_pop_TF:NNNN for all branching get and pop functions.

```

5268 \cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
5269 {
5270   \if_meaning:w #3 \c_empty_seq
5271   \tl_set:Nn #4 { \q_no_value }
5272   \else:
5273     #1#2#3#4
5274   \fi:
5275 }
5276 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
5277 {
5278   \if_meaning:w #3 \c_empty_seq
5279   % \tl_set:Nn #4 { \q_no_value }
5280   \prg_return_false:
5281   \else:
5282     #1#2#3#4
5283   \prg_return_true:
5284   \fi:
5285 }

```

(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

```

5286 \cs_new_protected:Npn \seq_get_left:NN #1#2
5287 {
5288   \tl_set:Nx #2
5289   {
5290     \exp_after:wN \__seq_get_left:wnw
5291     #1 \__seq_item:n { \q_no_value } \q_stop
5292   }
5293 }
5294 \cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \q_stop
5295 { \exp_not:n {#2} }
5296 \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 ??.)

`\seq_pop_left:NN`
`\seq_pop_left:cN`
`\seq_gpop_left:NN`
`\seq_gpop_left:cN`
`__seq_pop_left:NNN`
`__seq_pop_left:wnwNNN`

The approach to popping an item is pretty similar to that to get an item, with the only difference being that the sequence itself has to be redefined. This makes it more sensible to use an auxiliary function for the local and global cases.

```

5297 \cs_new_protected_nopar:Npn \seq_pop_left:NN
5298 { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_set:Nn }
5299 \cs_new_protected_nopar:Npn \seq_gpop_left:NN
5300 { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_gset:Nn }
5301 \cs_new_protected:Npn \__seq_pop_left:NNN #1#2#3
5302 { \exp_after:wN \__seq_pop_left:wnwNNN #2 \q_stop #1#2#3 }
5303 \cs_new_protected:Npn \__seq_pop_left:wnwNNN
5304 #1 \__seq_item:n #2#3 \q_stop #4#5#6
5305 {
5306   #4 #5 { #1 #3 }
5307   \tl_set:Nn #6 {#2}
5308 }
5309 \cs_generate_variant:Nn \seq_pop_left:NN { c }
5310 \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 ??.)

`\seq_get_right:NN`
`\seq_get_right:cN`
`__seq_get_right_setup:wN`
`__seq_get_right_loop:nn`

First prepend `\q_no_value`, then take two arguments at a time. Apart from the right-hand end of the sequence, this be a brace group followed by `__seq_item:n`. The `\use_none:nn` removes both of those. 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. The `\afterassignment` primitive places `\use_none:n` to get rid of a trailing `__seq_get_right_loop:nn`.

```

5311 \cs_new_protected:Npn \seq_get_right:NN #1#2
5312 { \exp_after:wN \__seq_get_right_setup:wN #1 #2 }
5313 \cs_new_protected:Npn \__seq_get_right_setup:wN \s__seq #1 \s_obj_end #2
5314 {
5315   \__seq_get_right_loop:nn
5316   \q_no_value
5317   #1
5318   {
5319     ??
5320     \tex_afterassignment:D \use_none:n
5321     \tl_set:Nn #2
5322   }
5323 }
5324 \cs_new_protected:Npn \__seq_get_right_loop:nn #1#2
5325 {
5326   \use_none:nn #2 {#1}
5327   \__seq_get_right_loop:nn
5328 }
5329 \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 ??.)

`\seq_pop_right:NN` The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a “flexible length” way to set a token list variable. This is supplied by the `{ \if_false: } \fi: ... \if_false: { \fi: }` construct. Using an x-type expansion and a “non-expanding” definition for `__seq_item:n`, the left-most $n - 1$ entries in a sequence of n items will be stored back in the sequence. That needs a loop of unknown length, hence using the strange `\if_false:` way of including brackets. When the last item of the sequence is reached, the closing bracket for the assignment is inserted, and `\tl_set:Nn #3` is inserted in front of the final entry. This therefore does the pop assignment. The trailing looping macro is removed by placing a `\use_none:n` using the `\afterassignment` primitive.

```

5330 \cs_new_protected_nopar:Npn \seq_pop_right:NN
5331 { \__seq_pop:NNNN \__seq_pop_right_aux:NNN \tl_set:Nx }
5332 \cs_new_protected_nopar:Npn \seq_gpop_right:NN
5333 { \__seq_pop:NNNN \__seq_pop_right_aux:NNN \tl_gset:Nx }
5334 \cs_new_protected:Npn \__seq_pop_right_aux:NNN #1#2#3
5335 {
5336   \cs_set_eq:NN \seq_tmp:w \__seq_item:n
5337   \cs_set_eq:NN \__seq_item:n \scan_stop:
5338   #1 #2
5339   { \if_false: } \fi: \s__seq
5340   \exp_after:wN \__seq_pop_right_setup:w
5341   #2
5342   {
5343     \s_obj_end \if_false: { \fi: }
5344     \tex_afterassignment:D \use_none:n
5345     \tl_set:Nx #3
5346   }
5347   \cs_set_eq:NN \__seq_item:n \seq_tmp:w
5348 }
5349 \cs_new:Npn \__seq_pop_right_setup:w \s__seq #1 \s_obj_end
5350 { \exp_after:wN \__seq_pop_right_loop:nn \use_none:n #1 }
5351 \cs_new:Npn \__seq_pop_right_loop:nn #1#2
5352 {
5353   #2 { \exp_not:n {#1} }
5354   \__seq_pop_right_loop:nn
5355 }
5356 \cs_generate_variant:Nn \seq_pop_right:NN { c }
5357 \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 ??.)

`\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`

```

5358 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
5359 { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
5360 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }

```

```

5361 { \_seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
5362 \cs_generate_variant:Nn \seq_get_left:NNT { c }
5363 \cs_generate_variant:Nn \seq_get_left:NNF { c }
5364 \cs_generate_variant:Nn \seq_get_left:NNTF { c }
5365 \cs_generate_variant:Nn \seq_get_right:NNT { c }
5366 \cs_generate_variant:Nn \seq_get_right:NNF { c }
5367 \cs_generate_variant:Nn \seq_get_right:NNTF { c }

```

(End definition for \seq_get_left:NN and \seq_get_left:cN. These functions are documented on page ??.)

```

\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
5368 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 { T , F , TF }
5369 { \_seq_pop_TF:NNNN \_seq_pop_left:NNN \tl_set:Nn #1 #2 }
5370 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 { T , F , TF }
5371 { \_seq_pop_TF:NNNN \_seq_pop_left:NNN \tl_gset:Nn #1 #2 }
5372 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2 { T , F , TF }
5373 { \_seq_pop_TF:NNNN \_seq_pop_right_aux:NNN \tl_set:Nx #1 #2 }
5374 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2 { T , F , TF }
5375 { \_seq_pop_TF:NNNN \_seq_pop_right_aux:NNN \tl_gset:Nx #1 #2 }
5376 \cs_generate_variant:Nn \seq_pop_left:NNT { c }
5377 \cs_generate_variant:Nn \seq_pop_left:NNF { c }
5378 \cs_generate_variant:Nn \seq_pop_left:NNTF { c }
5379 \cs_generate_variant:Nn \seq_gpop_left:NNT { c }
5380 \cs_generate_variant:Nn \seq_gpop_left:NNF { c }
5381 \cs_generate_variant:Nn \seq_gpop_left:NNTF { c }
5382 \cs_generate_variant:Nn \seq_pop_right:NNT { c }
5383 \cs_generate_variant:Nn \seq_pop_right:NNF { c }
5384 \cs_generate_variant:Nn \seq_pop_right:NNTF { c }
5385 \cs_generate_variant:Nn \seq_gpop_right:NNT { c }
5386 \cs_generate_variant:Nn \seq_gpop_right:NNF { c }
5387 \cs_generate_variant:Nn \seq_gpop_right:NNTF { c }

```

(End definition for \seq_pop_left:NN and \seq_pop_left:cN. These functions are documented on page ??.)

11.6 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.

```

5388 \cs_new_nopar:Npn \seq_map_break:
5389 { \_prg_map_break:Nn \seq_map_break: { } }
5390 \cs_new_nopar:Npn \seq_map_break:n
5391 { \_prg_map_break:Nn \seq_map_break: }

```

(End definition for \seq_map_break:. This function is documented on page 109.)

\seq_map_function:NN The idea here is to apply the code of #2 to each item in the sequence without altering
\seq_map_function:cN the definition of `_seq_item:n`. This is done as by noting that every odd token in the
_seq_map_function:wN sequence must be `_seq_item:n`, which can be gobbled by `\use_none:n`. At the end
_seq_map_function:NNN

of the loop, #2 is instead ? \seq_map_break:, which therefore breaks the loop without needing to do a (relatively-expensive) quark test.

```

5392 \cs_new:Npn \seq_map_function:NN #1#2
5393 { \exp_after:wN \__seq_map_function:wN #1 #2 }
5394 \cs_new:Npn \__seq_map_function:wN \s__seq #1 \s_obj_end #2
5395 {
5396   \__seq_map_function:NNn #2 #1 { ? \seq_map_break: } { }
5397   \__prg_break_point:Nn \seq_map_break: { }
5398 }
5399 \cs_new:Npn \__seq_map_function:NNn #1#2#3
5400 {
5401   \use_none:n #2
5402   #1 {#3}
5403   \__seq_map_function:NNn #1
5404 }
5405 \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 ??.)

```

\__seq_push_item_def:n
\__seq_push_item_def:x
\__seq_push_item_def:
\__seq_pop_item_def:

```

The definition of __seq_item:n needs to be saved and restored at various points within the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

```

5406 \cs_new_protected:Npn \__seq_push_item_def:n
5407 {
5408   \__seq_push_item_def:
5409   \cs_gset:Npn \__seq_item:n ##1
5410 }
5411 \cs_new_protected:Npn \__seq_push_item_def:x
5412 {
5413   \__seq_push_item_def:
5414   \cs_gset:Npx \__seq_item:n ##1
5415 }
5416 \cs_new_protected:Npn \__seq_push_item_def:
5417 {
5418   \int_gincr:N \g__prg_map_int
5419   \cs_gset_eq:cN { __prg_map_ \int_use:N \g__prg_map_int :w }
5420   \__seq_item:n
5421 }
5422 \cs_new_protected_nopar:Npn \__seq_pop_item_def:
5423 {
5424   \cs_gset_eq:Nc \__seq_item:n
5425   { __prg_map_ \int_use:N \g__prg_map_int :w }
5426   \int_gdecr:N \g__prg_map_int
5427 }

```

(End definition for __seq_push_item_def:n and __seq_push_item_def:x. These functions are documented on page 112.)

```

\seq_map_inline:Nn
\seq_map_inline:cn

```

The idea here is that __seq_item:n is already “applied” to each item in a sequence, and so an in-line mapping is just a case of redefining __seq_item:n.


```

5428 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
5429 {
5430   \__seq_push_item_def:n {#2}
5431   #1
5432   \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
5433 }
5434 \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 ??.)

\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

```

```

5435 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
5436 {
5437   \__seq_push_item_def:x
5438   {
5439     \tl_set:Nn \exp_not:N #2 {##1}
5440     \exp_not:n {#3}
5441   }
5442   #1
5443   \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
5444 }
5445 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
5446 \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 ??.)

\seq_count:N Counting the items in a sequence is done using the same approach as for other count functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics.

```

\seq_count:c
\__seq_count:n

```

```

5447 \cs_new:Npn \seq_count:N #1
5448 {
5449   \int_eval:n
5450   {
5451     0
5452     \seq_map_function:NN #1 \__seq_count:n
5453   }
5454 }
5455 \cs_new:Npn \__seq_count:n #1 { + \c_one }
5456 \cs_generate_variant:Nn \seq_count:N { c }

```

(End definition for \seq_count:N and \seq_count:c. These functions are documented on page ??.)

11.7 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 and \s_obj_end.

```

\seq_use:cn
\__seq_use:wnwn
\__seq_use_setup:w
\__seq_use:nwwwnwn
\__seq_use:nwwn

```

```

5457 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
5458 {

```

```

5459 \seq_if_exist:NTF #1
5460 {
5461   \int_case:nnF { \seq_count:N #1 }
5462   {
5463     { 0 } { }
5464     { 1 }
5465     {
5466       \exp_after:wN \__seq_use:wnwnn #1
5467       \__seq_item:n { } \s_obj_end { }
5468     }
5469     { 2 } { \exp_after:wN \__seq_use:wnwnn #1 {#2} }
5470   }
5471   {
5472     \exp_after:wN \__seq_use_setup:w #1 \__seq_item:n
5473     \q_mark { \__seq_use:nwwwnwn {#3} }
5474     \q_mark { \__seq_use:nwn {#4} }
5475     \q_stop { }
5476   }
5477 }
5478 { \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#1} }
5479 }
5480 \cs_generate_variant:Nn \seq_use:Nnnn { c }
5481 \cs_new:Npn \__seq_use:wnwnn
5482   #1 \__seq_item:n #2#3 \__seq_item:n #4#5 \s_obj_end #6
5483   { \exp_not:n { #2 #6 #4 } }
5484 \cs_new:Npn \__seq_use_setup:w \s__seq #1 \s_obj_end
5485   { \__seq_use:nwwwnwn { } #1 }
5486 \cs_new:Npn \__seq_use:nwwwnwn
5487   #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
5488   \q_mark #6#7 \q_stop #8
5489   {
5490     #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
5491     \q_mark {#6} #7 \q_stop { #8 #1 #2 }
5492   }
5493 \cs_new:Npn \__seq_use:nwn #1 \__seq_item:n #2 #3 \q_stop #4
5494   { \exp_not:n { #4 #1 #2 } }
5495 \cs_new:Npn \seq_use:Nn #1#2
5496   { \seq_use:Nnnn #1 {#2} {#2} {#2} }
5497 \cs_generate_variant:Nn \seq_use:Nn { c }

```

(End definition for \seq_use:Nnnn and \seq_use:cnnn. These functions are documented on page ??.)

11.8 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 5498 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
\seq_push:Nv 5499 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
\seq_push:No 5500 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
\seq_push:Nx
\seq_push:cn
\seq_push:cV
\seq_push:cV
\seq_push:co
\seq_push:cx
\seq_gpush:Nn
\seq_gpush:NV
\seq_gpush:Nv
\seq_gpush:No

```

```

5501 \cs_new_eq:NN \seq_push:No \seq_put_left:No
5502 \cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx
5503 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
5504 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
5505 \cs_new_eq:NN \seq_push:cv \seq_put_left:cv
5506 \cs_new_eq:NN \seq_push:co \seq_put_left:co
5507 \cs_new_eq:NN \seq_push:cx \seq_put_left:cx
5508 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
5509 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
5510 \cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
5511 \cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
5512 \cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
5513 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
5514 \cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv
5515 \cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
5516 \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 ??.)

`\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
\seq_pop:NN
\seq_pop:cN
\seq_gpop:NN
\seq_gpop:cN
5518 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
5519 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
5520 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
5521 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
5522 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
5523 \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 ??.)

`\seq_get:NNTF` More copies.

```

\seq_get:cNTF
\seq_pop:NTF
\seq_pop:cNTF
\seq_gpop:NTF
\seq_gpop:cNTF
5524 \prg_new_eq_conditional:NNn \seq_get:NN \seq_get_left:NN { T , F , TF }
5525 \prg_new_eq_conditional:NNn \seq_get:cN \seq_get_left:cN { T , F , TF }
5526 \prg_new_eq_conditional:NNn \seq_pop:NN \seq_pop_left:NN { T , F , TF }
5527 \prg_new_eq_conditional:NNn \seq_pop:cN \seq_pop_left:cN { T , F , TF }
5528 \prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
5529 \prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }

```

(End definition for `\seq_get:NN` and `\seq_get:cN`. These functions are documented on page ??.)

11.9 Viewing sequences

`\seq_show:N` Apply the general `_msg_show_variable:Nnn`.

```

\seq_show:c
5530 \cs_new_protected:Npn \seq_show:N #1
5531 {
5532   \_msg_show_variable:Nnn #1 { seq }
5533   { \seq_map_function:NN #1 \_msg_show_item:n }
5534 }
5535 \cs_generate_variant:Nn \seq_show:N { c }

```

(End definition for `\seq_show:N` and `\seq_show:c`. These functions are documented on page ??.)

11.10 Scratch sequences

`\l_tmpa_seq` Temporary comma list variables.
`\l_tmpb_seq` 5536 `\seq_new:N \l_tmpa_seq`
`\g_tmpa_seq` 5537 `\seq_new:N \l_tmpb_seq`
`\g_tmpb_seq` 5538 `\seq_new:N \g_tmpa_seq`
5539 `\seq_new:N \g_tmpb_seq`
(End definition for `\l_tmpa_seq` and others. These variables are documented on page 112.)
5540 `\initex | package`

12 l3clist implementation

The following test files are used for this code: `m3clist002`.

5541 `*initex | package`
5542 `\@@=clist`
5543 `*package`
5544 `\ProvidesExplPackage`
5545 `{\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}`
5546 `__expl_package_check:`
5547 `\package`

`\c_empty_clist` An empty comma list is simply an empty token list.

5548 `\cs_new_eq:NN \c_empty_clist \c_empty_tl`
(End definition for `\c_empty_clist`. This variable is documented on page 121.)

`\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`

5549 `\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.

5550 `\cs_new_protected:Npn __clist_tmp:w { }`
(End definition for `__clist_tmp:w`.)

12.1 Allocation and initialisation

`\clist_new:N` Internally, comma lists are just token lists.

`\clist_new:c` 5551 `\cs_new_eq:NN \clist_new:N \tl_new:N`
5552 `\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 ??.)

`\clist_clear:N` Clearing comma lists is just the same as clearing token lists.

`\clist_clear:c` 5553 `\cs_new_eq:NN \clist_clear:N \tl_clear:N`
`\clist_gclear:N` 5554 `\cs_new_eq:NN \clist_clear:c \tl_clear:c`
`\clist_gclear:c` 5555 `\cs_new_eq:NN \clist_gclear:N \tl_gclear:N`
5556 `\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 ??.)

```
\clist_clear_new:N Once again a copy from the token list functions.
\clist_clear_new:c 5557 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
\clist_gclear_new:N 5558 \cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c
\clist_gclear_new:c 5559 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
5560 \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 ??.)

```
\clist_set_eq:NN Once again, these are simple copies from the token list functions.
\clist_set_eq:cN 5561 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
\clist_set_eq:Nc 5562 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
\clist_set_eq:cc 5563 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
\clist_gset_eq:NN 5564 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
\clist_gset_eq:cN 5565 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\clist_gset_eq:Nc 5566 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\clist_gset_eq:cN 5567 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
5568 \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 ??.)

```
\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 5569 \cs_new_protected_nopar:Npn \clist_concat:NNN
\clist_gconcat:ccc 5570 { \__clist_concat:NNNN \tl_set:Nx }
\__clist_concat:NNNN 5571 \cs_new_protected_nopar:Npn \clist_gconcat:NNN
5572 { \__clist_concat:NNNN \tl_gset:Nx }
5573 \cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
5574 {
5575   #1 #2
5576   {
5577     \exp_not:o #3
5578     \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
5579     \exp_not:o #4
5580   }
5581 }
5582 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
5583 \cs_generate_variant:Nn \clist_gconcat:NNN { ccc }
```

(End definition for \clist_concat:NNN and \clist_concat:ccc. These functions are documented on page ??.)

```
\clist_if_exist_p:N Copies of the cs functions defined in l3basics.
\clist_if_exist_p:c 5584 \prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N { TF , T , F , p }
\clist_if_exist:NTF 5585 \prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c { TF , T , F , p }
\clist_if_exist:cTF (End definition for \clist_if_exist:N and \clist_if_exist:c. These functions are documented on
page ??.)
```

12.2 Removing spaces around items

`_clist_trim_spaces_generic:nw`
`_clist_trim_spaces_generic:nn`

This expands to the $\langle code \rangle$, followed by a brace group containing the $\langle item \rangle$, with leading and trailing spaces removed. The calling function is responsible for inserting `\q_mark` in front of the $\langle item \rangle$, 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 $\langle code \rangle$ in front of the $\langle trimmed item \rangle$.

```
5586 \cs_new:Npn \_clist_trim_spaces_generic:nw #1#2 ,
5587 {
5588   \_tl_trim_spaces:nn {#2}
5589   { \exp_args:No \_clist_trim_spaces_generic:nn } {#1}
5590 }
5591 \cs_new:Npn \_clist_trim_spaces_generic:nn #1#2 { #2 {#1} }
```

(End definition for `_clist_trim_spaces_generic:nw`. This function is documented on page ??.)

`_clist_trim_spaces:n`
`_clist_trim_spaces:nn`

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.

```
5592 \cs_new:Npn \_clist_trim_spaces:n #1
5593 {
5594   \_clist_trim_spaces_generic:nw
5595   { \_clist_trim_spaces:nn { } }
5596   \q_mark #1 ,
5597   \q_recursion_tail, \q_recursion_stop
5598 }
5599 \cs_new:Npn \_clist_trim_spaces:nn #1 #2
5600 {
5601   \quark_if_recursion_tail_stop:n {#2}
5602   \tl_if_empty:nTF {#2}
5603   {
5604     \_clist_trim_spaces_generic:nw
5605     { \_clist_trim_spaces:nn {#1} } \q_mark
5606   }
5607   {
5608     #1 \exp_not:n {#2}
5609     \_clist_trim_spaces_generic:nw
5610     { \_clist_trim_spaces:nn { , } } \q_mark
5611   }
5612 }
```

(End definition for `_clist_trim_spaces:n`. This function is documented on page ??.)

12.3 Adding data to comma lists

`\clist_set:Nn`

`\clist_set:NV`

`\clist_set:No`

`\clist_set:Nx`

`\clist_set:cn`

`\clist_set:cV`

`\clist_set:co`

`\clist_set:cx`

`\clist_gset:Nn`

`\clist_gset:NV`

`\clist_gset:No`

`\clist_gset:Nx`

`\clist_gset:cn`

```
5613 \cs_new_protected:Npn \clist_set:Nn #1#2
5614 { \tl_set:Nx #1 { \_clist_trim_spaces:n {#2} } }
```

```

5615 \cs_new_protected:Npn \clist_gset:Nn #1#2
5616 { \tl_gset:Nx #1 { \_clist_trim_spaces:n {#2} } }
5617 \cs_generate_variant:Nn \clist_set:Nn { NV , No , Nx , c , cV , co , cx }
5618 \cs_generate_variant:Nn \clist_gset:Nn { NV , No , Nx , c , cV , co , cx }
(End definition for \clist_set:Nn and others. These functions are documented on page ??.)

```

Comma lists cannot hold empty values: there are therefore a couple of sanity checks to avoid accumulating commas.

```

\clist_put_left:Nn
\clist_put_left:NV
\clist_put_left:No
\clist_put_left:Nx
\clist_put_left:cn
\clist_put_left:cV
\clist_put_left:co
\clist_put_left:cx
\clist_gput_left:Nn
\clist_gput_left:NV
\clist_gput_left:No
\clist_gput_left:Nx
\clist_gput_left:cn
\clist_gput_left:cV
\clist_gput_left:co
\clist_gput_left:cx
\__\clistpput_right:Nn
\clist_put_right:NV
\clist_put_right:No
\clist_put_right:Nx
\clist_put_right:cn
\clist_put_right:cV
\clist_put_right:co
\clist_put_right:cx
\clist_gput_right:Nn
\clist_gput_right:NV
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cn
\clist_gput_right:cV
\clist_gput_right:co
\clist_gput_right:cx
\__clist_put_right:NNNn
\clist_get:Nn
\clist_get:cn
\__clist_get:wN
5619 \cs_new_protected_nopar:Npn \clist_put_left:Nn
5620 { \_clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
5621 \cs_new_protected_nopar:Npn \clist_gput_left:Nn
5622 { \_clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
5623 \cs_new_protected:Npn \_clist_put_left:NNNn #1#2#3#4
5624 {
5625   #2 \l__clist_internal_clist {#4}
5626   #1 #3 \l__clist_internal_clist #3
5627 }
5628 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
5629 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
5630 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
5631 \cs_generate_variant:Nn \clist_gput_left:Nn { c , cV , co , cx }
(End definition for \clist_put_left:Nn and others. These functions are documented on page ??.)

```

```

5632 \cs_new_protected_nopar:Npn \clist_put_right:Nn
5633 { \_clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
5634 \cs_new_protected_nopar:Npn \clist_gput_right:Nn
5635 { \_clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
5636 \cs_new_protected:Npn \_clist_put_right:NNNn #1#2#3#4
5637 {
5638   #2 \l__clist_internal_clist {#4}
5639   #1 #3 #3 \l__clist_internal_clist
5640 }
5641 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
5642 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
5643 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
5644 \cs_generate_variant:Nn \clist_gput_right:Nn { c , cV , co , cx }
(End definition for \clist_put_right:Nn and others. These functions are documented on page ??.)

```

12.4 Comma lists as stacks

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma.

```

5645 \cs_new_protected:Npn \clist_get:NN #1#2
5646 {
5647   \if_meaning:w #1 \c_empty_clist
5648     \tl_set:Nn #2 { \q_no_value }
5649   \else:

```

```

5650         \exp_after:wN \_clist_get:wN #1 , \q_stop #2
5651     \fi:
5652 }
5653 \cs_new_protected:Npn \_clist_get:wN #1 , #2 \q_stop #3
5654 { \tl_set:Nn #3 {#1} }
5655 \cs_generate_variant:Nn \clist_get:NN { c }

```

(End definition for \clist_get:NN and \clist_get:cN. These functions are documented on page ??.)

\clist_pop:NN An empty clist leads to \q_no_value, otherwise grab until the first comma and assign to the variable. The second argument of _clist_pop:wwNNN is a comma list ending in a comma and \q_mark, unless the original clist contained exactly one item: then the argument is just \q_mark. The next auxiliary picks either \exp_not:n or \use_none:n as #2, ensuring that the result can safely be an empty comma list.

```

\_clist_pop:NNN
\_clist_pop:wwNNN
\_clist_pop:wN
5656 \cs_new_protected_nopar:Npn \clist_pop:NN
5657 { \_clist_pop:NNN \tl_set:Nx }
5658 \cs_new_protected_nopar:Npn \clist_gpop:NN
5659 { \_clist_pop:NNN \tl_gset:Nx }
5660 \cs_new_protected:Npn \_clist_pop:NNN #1#2#3
5661 {
5662     \if_meaning:w #2 \c_empty_clist
5663     \tl_set:Nn #3 { \q_no_value }
5664 \else:
5665     \exp_after:wN \_clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
5666 \fi:
5667 }
5668 \cs_new_protected:Npn \_clist_pop:wwNNN #1 , #2 \q_stop #3#4#5
5669 {
5670     \tl_set:Nn #5 {#1}
5671     #3 #4
5672     {
5673         \_clist_pop:wN \prg_do_nothing:
5674         #2 \exp_not:o
5675         , \q_mark \use_none:n
5676     }
5677 }
5678 }
5679 \cs_new:Npn \_clist_pop:wN #1 , \q_mark #2 #3 \q_stop { #2 {#1} }
5680 \cs_generate_variant:Nn \clist_pop:NN { c }
5681 \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End definition for \clist_pop:NN and \clist_pop:cN. These functions are documented on page ??.)

\clist_get:NTF The same, as branching code: very similar to the above.

```

\_clist_get:cNTF
5682 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
5683 {
5684     \if_meaning:w #1 \c_empty_clist
5685     \prg_return_false:
5686 \else:
5687     \exp_after:wN \_clist_get:wN #1 , \q_stop #2
5688     \prg_return_true:

```

\clist_pop:NTF

\clist_gpop:NTF

\clist_pop:cNTF

_clist_pop_TF:NNN


```

5689     \fi:
5690   }
5691   \cs_generate_variant:Nn \clist_get:NNT { c }
5692   \cs_generate_variant:Nn \clist_get:NNF { c }
5693   \cs_generate_variant:Nn \clist_get:NNTF { c }
5694   \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
5695   { \__clist_pop_TF:NNN \tl_set:Nx #1 #2 }
5696   \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
5697   { \__clist_pop_TF:NNN \tl_gset:Nx #1 #2 }
5698   \cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
5699   {
5700     \if_meaning:w #2 \c_empty_clist
5701     \prg_return_false:
5702   \else:
5703     \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
5704     \prg_return_true:
5705   \fi:
5706 }
5707 \cs_generate_variant:Nn \clist_pop:NNT { c }
5708 \cs_generate_variant:Nn \clist_pop:NNF { c }
5709 \cs_generate_variant:Nn \clist_pop:NNTF { c }
5710 \cs_generate_variant:Nn \clist_gpop:NNT { c }
5711 \cs_generate_variant:Nn \clist_gpop:NNF { c }
5712 \cs_generate_variant:Nn \clist_gpop:NNTF { c }

```

(End definition for \clist_get:NN and \clist_get:cn. These functions are documented on page ??.)

```

\clist_push:Nn Pushing to a comma list is the same as adding on the left.
\clist_push:NV
\clist_push:No
\clist_push:Nx
\clist_push:cn
\clist_push:cV
\clist_push:co
\clist_push:cx
\clist_gpush:Nn
\clist_gpush:NV
\clist_gpush:No
\clist_gpush:Nx
\clist_gpush:cn
\clist_gpush:cV
\clist_gpush:co
\clist_gpush:cx

```

(End definition for \clist_push:Nn and others. These functions are documented on page ??.)

12.5 Modifying comma lists

\l__clist_internal_remove_clist An internal comma list for the removal routines.

```

5729 \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
5730 \cs_new_protected:Npn \clist_remove_duplicates:N
5731 { \__clist_remove_duplicates:NN \clist_set_eq:NN }
5732 \cs_new_protected:Npn \clist_gremove_duplicates:N
5733 { \__clist_remove_duplicates:NN \clist_gset_eq:NN }
5734 \cs_new_protected:Npn \__clist_remove_duplicates:NN #1#2
5735 {
5736   \clist_clear:N \l__clist_internal_remove_clist
5737   \clist_map_inline:Nn #2
5738   {
5739     \clist_if_in:NnF \l__clist_internal_remove_clist {##1}
5740     { \clist_put_right:Nn \l__clist_internal_remove_clist {##1} }
5741   }
5742   #1 #2 \l__clist_internal_remove_clist
5743 }
5744 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }
5745 \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 ??.)

```

\clist_remove_all:Nn The method used here is very similar to \tl_replace_all:Nnn. Build a function de-
\clist_remove_all:cn limited by the <item> that should be removed, surrounded with commas, and call that
\clist_gremove_all:Nn function followed by the expanded comma list, and another copy of the <item>. The loop
\clist_gremove_all:cn is controlled by the argument grabbed by \__clist_remove_all:w: when the item was
\__clist_remove_all:NNn found, the \q_mark delimiter used is the one inserted by \__clist_tmp:w, and \use_-
\__clist_remove_all:w none_delimit_by_q_stop:w is deleted. At the end, the final <item> is grabbed, and
\__clist_remove_all: 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.

```

5746 \cs_new_protected:Npn \clist_remove_all:Nn
5747 { \__clist_remove_all:NNn \tl_set:Nx }
5748 \cs_new_protected:Npn \clist_gremove_all:Nn
5749 { \__clist_remove_all:NNn \tl_gset:Nx }
5750 \cs_new_protected:Npn \__clist_remove_all:NNn #1#2#3
5751 {
5752   \cs_set:Npn \__clist_tmp:w ##1 , #3 ,
5753   {
5754     ##1
5755     , \q_mark , \use_none_delimit_by_q_stop:w ,
5756     \__clist_remove_all:
5757   }

```

```

5758     #1 #2
5759     {
5760         \exp_after:wN \__clist_remove_all:
5761         #2 , \q_mark , #3 , \q_stop
5762     }
5763     \clist_if_empty:NF #2
5764     {
5765         #1 #2
5766         {
5767             \exp_args:No \exp_not:o
5768             { \exp_after:wN \use_none:n #2 }
5769         }
5770     }
5771 }
5772 \cs_new:Npn \__clist_remove_all:
5773 { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
5774 \cs_new:Npn \__clist_remove_all:w #1 , \q_mark , #2 , { \exp_not:n {#1} }
5775 \cs_generate_variant:Nn \clist_remove_all:Nn { c }
5776 \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 ??.)

12.6 Comma list conditionals

\clist_if_empty_p:N Simple copies from the token list variable material.

```

5777 \prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N { p , T , F , TF }
5778 \prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c { p , T , F , TF }

```

\clist_if_empty:N **TF**

\clist_if_empty:c **TF** (End definition for \clist_if_empty:N and \clist_if_empty:c. These functions are documented on page ??.)

\clist_if_in:Nn **TF** See description of the \tl_if_in:Nn function for details. We simply surround the comma list, and the item, with commas.

\clist_if_in:NVT **TF**

\clist_if_in:No **TF**

```

5779 \prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }
5780 {
5781     \exp_args:No \__clist_if_in_return:nn #1 {#2}
5782 }
5783 \prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T , F , TF }
5784 {
5785     \clist_set:Nn \l__clist_internal_clist {#1}
5786     \exp_args:No \__clist_if_in_return:nn \l__clist_internal_clist {#2}
5787 }
5788 \cs_new_protected:Npn \__clist_if_in_return:nn #1#2
5789 {
5790     \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
5791     \tl_if_empty:oTF
5792     { \__clist_tmp:w ,#1, {} {} } ,#2, {
5793     { \prg_return_false: } { \prg_return_true: }
5794 }
5795 \cs_generate_variant:Nn \clist_if_in:NnT { NV , No }

```

\clist_if_in:cn **TF**

\clist_if_in:c **TF**

\clist_if_in:co **TF**

\clist_if_in:nn **TF**

\clist_if_in:nVT **TF**

\clist_if_in:no **TF**

__clist_if_in_return:nn

```

5796 \cs_generate_variant:Nn \clist_if_in:NnT { c , cV , co }
5797 \cs_generate_variant:Nn \clist_if_in:NnF {      NV , No }
5798 \cs_generate_variant:Nn \clist_if_in:NnF { c , cV , co }
5799 \cs_generate_variant:Nn \clist_if_in:NnTF {      NV , No }
5800 \cs_generate_variant:Nn \clist_if_in:NnTF { c , cV , co }
5801 \cs_generate_variant:Nn \clist_if_in:nnT {      nV , no }
5802 \cs_generate_variant:Nn \clist_if_in:nnF {      nV , no }
5803 \cs_generate_variant:Nn \clist_if_in:nnTF {      nV , no }

```

(End definition for \clist_if_in:Nn and others. These functions are documented on page ??.)

12.7 Mapping to comma lists

\clist_map_function:NN If the variable is empty, the mapping is skipped (otherwise, that comma-list would be seen as consisting of one empty item). Then loop over the comma-list, grabbing one comma-delimited item at a time. The end is marked by \q_recursion_tail. The auxiliary function __clist_map_function:Nw is used directly in \clist_map_inline:Nn. Change with care.

```

5804 \cs_new:Npn \clist_map_function:NN #1#2
5805 {
5806   \clist_if_empty:NF #1
5807   {
5808     \exp_last_unbraced:NNo \__clist_map_function:Nw #2 #1
5809     , \q_recursion_tail ,
5810     \__prg_break_point:Nn \clist_map_break: { }
5811   }
5812 }
5813 \cs_new:Npn \__clist_map_function:Nw #1#2 ,
5814 {
5815   \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
5816   #1 {#2}
5817   \__clist_map_function:Nw #1
5818 }
5819 \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 ??.)

\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. The auxiliary __clist_map_function_n:Nn receives as arguments the function, and the result of removing leading and trailing spaces from the item which lies until the next comma. Empty items are ignored, then one level of braces is removed by __clist_map_unbrace:Nw.

```

5820 \cs_new:Npn \clist_map_function:nN #1#2
5821 {
5822   \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #2 }
5823   \q_mark #1, \q_recursion_tail,
5824   \__prg_break_point:Nn \clist_map_break: { }
5825 }

```

```

5826 \cs_new:Npn \__clist_map_function_n:Nn #1 #2
5827 {
5828   \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
5829   \tl_if_empty:nF {#2} { \__clist_map_unbrace:Nw #1 #2, }
5830   \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #1 }
5831   \q_mark
5832 }
5833 \cs_new:Npn \__clist_map_unbrace:Nw #1 #2, { #1 {#2} }

```

(End definition for \clist_map_function:nN. This function is documented on page ??.)

\clist_map_inline:Nn Inline mapping is done by creating a suitable function “on the fly”: this is done globally
\clist_map_inline:cn to avoid any issues with T_EX’s groups. We use a different function for each level of
\clist_map_inline:nn 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.

```

5834 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
5835 {
5836   \clist_if_empty:NF #1
5837   {
5838     \int_gincr:N \g__prg_map_int
5839     \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
5840     \exp_last_unbraced:Nco \__clist_map_function:Nw
5841     { __prg_map_ \int_use:N \g__prg_map_int :w }
5842     #1 , \q_recursion_tail ,
5843     \__prg_break_point:Nn \clist_map_break:
5844     { \int_gdecr:N \g__prg_map_int }
5845   }
5846 }
5847 \cs_new_protected:Npn \clist_map_inline:nn #1
5848 {
5849   \clist_set:Nn \l__clist_internal_clist {#1}
5850   \clist_map_inline:Nn \l__clist_internal_clist
5851 }
5852 \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 ??.)

\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.

```

5853 \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
5854 {
5855   \clist_if_empty:NF #1
5856   {
5857     \exp_args:Nno \use:nn
5858     { \__clist_map_variable:Nnw #2 {#3} }
5859     #1

```

```

5860         , \q_recursion_tail , \q_recursion_stop
5861         \__prg_break_point:Nn \clist_map_break: { }
5862     }
5863 }
5864 \cs_new_protected:Npn \clist_map_variable:nNn #1
5865 {
5866     \clist_set:Nn \l__clist_internal_clist {#1}
5867     \clist_map_variable:NNn \l__clist_internal_clist
5868 }
5869 \cs_new_protected:Npn \__clist_map_variable:Nnw #1#2#3,
5870 {
5871     \tl_set:Nn #1 {#3}
5872     \quark_if_recursion_tail_stop:N #1
5873     \use:n {#2}
5874     \__clist_map_variable:Nnw #1 {#2}
5875 }
5876 \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 ??.)

\clist_map_break: The break statements use the general __prg_map_break:Nn mechanism.
\clist_map_break:n

```

5877 \cs_new_nopar:Npn \clist_map_break:
5878 { \__prg_map_break:Nn \clist_map_break: { } }
5879 \cs_new_nopar:Npn \clist_map_break:n
5880 { \__prg_map_break:Nn \clist_map_break: }

```

(End definition for \clist_map_break: and \clist_map_break:n. These functions are documented on page 118.)

\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).

```

5881 \cs_new:Npn \clist_count:N #1
5882 {
5883     \int_eval:n
5884     {
5885         0
5886         \clist_map_function:NN #1 \__clist_count:n
5887     }
5888 }
5889 \cs_generate_variant:Nn \clist_count:N { c }
5890 \cs_new:Npx \clist_count:n #1
5891 {
5892     \exp_not:N \int_eval:n
5893     {
5894         0
5895         \exp_not:N \__clist_count:w \c_space_tl
5896         #1 \exp_not:n { , \q_recursion_tail , \q_recursion_stop }

```

```

5897     }
5898   }
5899   \cs_new:Npn \__clist_count:n #1 { + \c_one }
5900   \cs_new:Npx \__clist_count:w #1 ,
5901   {
5902     \exp_not:n { \exp_args:Nf \quark_if_recursion_tail_stop:n } {#1}
5903     \exp_not:N \tl_if_blank:nF {#1} { + \c_one }
5904     \exp_not:N \__clist_count:w \c_space_tl
5905   }

```

(End definition for `\clist_count:N`, `\clist_count:c`, and `\clist_count:n`. These functions are documented on page ??.)

12.8 Using comma lists

```

\clist_use:Nnnn
\clist_use:cnmn
  \__clist_use:wwn
\__clist_use:nwwwnwn
  \__clist_use:nwwn
\clist_use:Nn
\clist_use:cn

```

First check that the variable exists. Then count the items in the comma list. If it has none, output nothing. If it has one item, output that item, brace stripped (note that space-trimming has already been done when the comma list was assigned). If it has two, place the *separator between two* in the middle.

Otherwise, `__clist_use:nwwwnwn` takes the following arguments; 1: a *separator*, 2, 3, 4: three items from the comma list (or quarks), 5: the rest of the comma list, 6: a *continuation* function (`use_ii` or `use_iii` with its *separator* argument), 7: junk, and 8: the temporary result, which is built in a brace group following `\q_stop`. The *separator* and the first of the three items are placed in the result, then we use the *continuation*, placing the remaining two items after it. When we begin this loop, the three items really belong to the comma list, the first `\q_mark` is taken as a delimiter to the `use_ii` function, and the continuation is `use_ii` itself. When we reach the last two items of the original token list, `\q_mark` is taken as a third item, and now the second `\q_mark` serves as a delimiter to `use_iii`, switching to the other *continuation*, `use_iii`, which uses the *separator between final two*.

```

5906   \cs_new:Npn \clist_use:Nnnn #1#2#3#4
5907   {
5908     \clist_if_exist:NTF #1
5909     {
5910       \int_case:nnF { \clist_count:N #1 }
5911       {
5912         { 0 } { }
5913         { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
5914         { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
5915       }
5916       {
5917         \exp_after:wN \__clist_use:nwwwnwn
5918         \exp_after:wN { \exp_after:wN } #1 ,
5919         \q_mark , { \__clist_use:nwwwnwn {#3} }
5920         \q_mark , { \__clist_use:nwwn {#4} }
5921         \q_stop { }
5922       }
5923     }
5924     { \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#1} }

```

```

5925 }
5926 \cs_generate_variant:Nn \clist_use:Nnnn { c }
5927 \cs_new:Npn \__clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
5928 \cs_new:Npn \__clist_use:nwwwnwn
5929   #1#2 , #3 , #4 , #5 \q_mark , #6#7 \q_stop #8
5930   { #6 {#3} , {#4} , #5 \q_mark , {#6} #7 \q_stop { #8 #1 #2 } }
5931 \cs_new:Npn \__clist_use:nwn #1#2 , #3 \q_stop #4
5932   { \exp_not:n { #4 #1 #2 } }
5933 \cs_new:Npn \clist_use:Nn #1#2
5934   { \clist_use:Nnnn #1 {#2} {#2} {#2} }
5935 \cs_generate_variant:Nn \clist_use:Nn { c }

```

(End definition for \clist_use:Nnnn and \clist_use:cnnn. These functions are documented on page ??.)

12.9 Viewing comma lists

\clist_show:N Apply the general __msg_show_variable:Nnn. In the case of an n-type comma-list, first store it in a scratch variable, then show that variable: The message takes care of omitting its name.

```

5936 \cs_new_protected:Npn \clist_show:N #1
5937   {
5938     \__msg_show_variable:Nnn #1 { clist }
5939     { \clist_map_function:NN #1 \__msg_show_item:n }
5940   }
5941 \cs_new_protected:Npn \clist_show:n #1
5942   {
5943     \clist_set:Nn \l__clist_internal_clist {#1}
5944     \clist_show:N \l__clist_internal_clist
5945   }
5946 \cs_generate_variant:Nn \clist_show:N { c }

```

(End definition for \clist_show:N and \clist_show:c. These functions are documented on page 121.)

12.10 Scratch comma lists

\l_tmpa_clist Temporary comma list variables.

```

5947 \clist_new:N \l_tmpa_clist
5948 \clist_new:N \l_tmpb_clist
5949 \clist_new:N \g_tmpa_clist
5950 \clist_new:N \g_tmpb_clist

```

(End definition for \l_tmpa_clist and \l_tmpb_clist. These functions are documented on page 121.)

```

5951 \</initex | package)

```

13 l3prop implementation

The following test files are used for this code: m3prop001, m3prop002, m3prop003, m3prop004, m3show001.

```

5952 \< *initex | package)

```



```

5953 <@@=prop>
5954 <*package>
5955 \ProvidesExplPackage
5956   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
5957 \__expl_package_check:
5958 </package>

```

A property list is a macro whose top-level expansion is for the form

```

\s__prop \__prop_pair:wn <key1> \s__prop {<value1>}
...
\__prop_pair:wn <keyn> \s__prop {<valuen>}
\s_obj_end

```

where the separators `\s__prop` and `\s_obj_end` are scan marks (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.

```

5959 \__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.

```

5960 \cs_new:Npn \__prop_pair:wn #1 \s__prop #2
5961   { \_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.

```

5962 \tl_new:N \l__prop_internal_tl
(End definition for \l__prop_internal_tl. This variable is documented on page 127.)

```

`\c_empty_prop` An empty prop.

```

5963 \tl_const:Nn \c_empty_prop { \s__prop \s_obj_end }
(End definition for \c_empty_prop. This variable is documented on page 127.)

```

13.1 Allocation and initialisation

`\prop_new:N` Property lists are initialized with the value `\c_empty_prop`.

```

\prop_new:c
5964 \cs_new_protected:Npn \prop_new:N #1
5965   {
5966     \__chk_if_free_cs:N #1
5967     \cs_gset_eq:NN #1 \c_empty_prop
5968   }
5969 \cs_generate_variant:Nn \prop_new:N { c }
(End definition for \prop_new:N and \prop_new:c. These functions are documented on page ??.)

```

`\prop_clear:N` The same idea for clearing.

```

\prop_clear:c      5970 \cs_new_protected:Npn \prop_clear:N #1
\prop_gclear:N     5971 { \prop_set_eq:NN #1 \c_empty_prop }
\prop_gclear:c     5972 \cs_generate_variant:Nn \prop_clear:N { c }
                   5973 \cs_new_protected:Npn \prop_gclear:N #1
                   5974 { \prop_gset_eq:NN #1 \c_empty_prop }
                   5975 \cs_generate_variant:Nn \prop_gclear:N { c }

```

(End definition for `\prop_clear:N` and `\prop_clear:c`. These functions are documented on page ??.)

`\prop_clear_new:N` Once again a simple variation of the token list functions.

```

\prop_clear_new:c  5976 \cs_new_protected:Npn \prop_clear_new:N #1
\prop_gclear_new:N 5977 { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\prop_gclear_new:c 5978 \cs_generate_variant:Nn \prop_clear_new:N { c }
                   5979 \cs_new_protected:Npn \prop_gclear_new:N #1
                   5980 { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
                   5981 \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 ??.)

`\prop_set_eq:NN` These are simply copies from the token list functions.

```

\prop_set_eq:cN    5982 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
\prop_set_eq:Nc    5983 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
\prop_set_eq:cc    5984 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
\prop_gset_eq:NN   5985 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
\prop_gset_eq:cN   5986 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
\prop_gset_eq:Nc   5987 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
\prop_gset_eq:cN   5988 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
\prop_gset_eq:cc   5989 \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 ??.)

`\l_tmpa_prop` We can now initialize the scratch variables.

```

\l_tmpb_prop      5990 \prop_new:N \l_tmpa_prop
\g_tmpa_prop      5991 \prop_new:N \l_tmpb_prop
\g_tmpb_prop      5992 \prop_new:N \g_tmpa_prop
                   5993 \prop_new:N \g_tmpb_prop

```

(End definition for `\l_tmpa_prop` and `\l_tmpb_prop`. These functions are documented on page 127.)

13.2 Accessing data in property lists

`__prop_strip_end:w` Strip the `\s_obj_end` marker from an expanded property list variable.

```

5994 \cs_new:Npn \__prop_strip_end:w #1 \s_obj_end { \exp_not:n {#1} }

```

(End definition for `__prop_strip_end:w`. This function is documented on page ??.)

`__prop_split:NnTF` This function is used by most of the module, and hence must be fast. It receives a
`__prop_split_aux:NnTF` $\langle \textit{property list} \rangle$, a $\langle \textit{key} \rangle$, a $\langle \textit{true code} \rangle$ and a $\langle \textit{false code} \rangle$. The aim is to split the $\langle \textit{property}$
`__prop_split_aux:w` $\textit{list} \rangle$ at the given $\langle \textit{key} \rangle$ into the $\langle \textit{extract}_1 \rangle$ before the key–value pair, the $\langle \textit{value} \rangle$ associated
with the $\langle \textit{key} \rangle$ and the $\langle \textit{extract}_2 \rangle$ after the key–value pair. This is done using a delimited
function, whose definition is as follows, where the $\langle \textit{key} \rangle$ is turned into a string.

```

\cs_set:Npn \__prop_split_aux:w #1
\__prop_pair:wn <key> \s__prop #2
#3 \q_mark #4 #5 \q_stop
{ #4 {<true code>} {<false code>}}

```

If the $\langle key \rangle$ is present in the property list, $\backslash__prop_split_aux:w$'s #1 is the part before the $\langle key \rangle$, #2 is the $\langle value \rangle$, #3 is the part after the $\langle key \rangle$, #4 is $\backslash use_i:nn$, and #5 is additional tokens that we do not care about. The $\langle true\ code \rangle$ is left in the input stream, and can use the parameters #1, #2, #3 for the three parts of the property list as desired. Namely, the original property list is in this case #1 $\backslash__prop_pair:wn\ \langle key \rangle\ \backslash s_prop\ \{ \#2 \}\ \#3$.

If the $\langle key \rangle$ is not there, then the $\langle function \rangle$ is $\backslash use_ii:nn$, which keeps the $\langle false\ code \rangle$.

```

5995 \cs_new_protected:Npn \__prop_split:NnTF #1#2
5996 { \exp_args:NNo \__prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
5997 \cs_new_protected:Npn \__prop_split_aux:NnTF #1#2#3#4
5998 {
5999   \cs_set:Npn \__prop_split_aux:w ##1
6000     \__prop_pair:wn #2 \s__prop ##2 ##3 \q_mark ##4 ##5 \q_stop
6001     { ##4 {#3} {#4} }
6002   \exp_after:wN \__prop_split_aux:w #1 \q_mark \use_i:nn
6003   \__prop_pair:wn #2 \s__prop { } \q_mark \use_ii:nn \q_stop
6004 }
6005 \cs_new:Npn \__prop_split_aux:w { }

```

(End definition for $\backslash__prop_split:NnTF$. This function is documented on page 127.)

$\backslash prop_remove:Nn$ Deleting from a property starts by splitting the list. If the key is present in the property list, the returned value is ignored. If the key is missing, nothing happens.

```

\prop_remove:NV
\prop_remove:cn
\prop_remove:cV
\prop_gremove:Nn
\prop_gremove:NV
\prop_gremove:cn
\prop_gremove:cV
6006 \cs_new_protected:Npn \prop_remove:Nn #1#2
6007 {
6008   \__prop_split:NnTF #1 {#2}
6009   { \tl_set:Nn #1 { ##1 ##3 } }
6010   { }
6011 }
6012 \cs_new_protected:Npn \prop_gremove:Nn #1#2
6013 {
6014   \__prop_split:NnTF #1 {#2}
6015   { \tl_gset:Nn #1 { ##1 ##3 } }
6016   { }
6017 }
6018 \cs_generate_variant:Nn \prop_remove:Nn { NV }
6019 \cs_generate_variant:Nn \prop_remove:Nn { c , cV }
6020 \cs_generate_variant:Nn \prop_gremove:Nn { NV }
6021 \cs_generate_variant:Nn \prop_gremove:Nn { c , cV }

```

(End definition for $\backslash prop_remove:Nn$ and others. These functions are documented on page ??.)

$\backslash prop_get:NnN$ 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 $\backslash q_no_value$.

```

\prop_get:NVN
\prop_get:NoN
\prop_get:cnN
\prop_get:cVN
\prop_get:coN

```

```

6022 \cs_new_protected:Npn \prop_get:NnN #1#2#3
6023 {
6024   \__prop_split:NnTF #1 {#2}
6025   { \tl_set:Nn #3 {##2} }
6026   { \tl_set:Nn #3 { \q_no_value } }
6027 }
6028 \cs_generate_variant:Nn \prop_get:NnN { NV , No }
6029 \cs_generate_variant:Nn \prop_get:NnN { c , cV , co }

```

(End definition for \prop_get:NnN and others. These functions are documented on page ??.)

\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
\prop_gpop:NnN
\prop_gpop:NoN
\prop_gpop:cnN
\prop_gpop:coN
6030 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
6031 {
6032   \__prop_split:NnTF #1 {#2}
6033   {
6034     \tl_set:Nn #3 {##2}
6035     \tl_set:Nn #1 { ##1 ##3 }
6036   }
6037   { \tl_set:Nn #3 { \q_no_value } }
6038 }
6039 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
6040 {
6041   \__prop_split:NnTF #1 {#2}
6042   {
6043     \tl_set:Nn #3 {##2}
6044     \tl_gset:Nn #1 { ##1 ##3 }
6045   }
6046   { \tl_set:Nn #3 { \q_no_value } }
6047 }
6048 \cs_generate_variant:Nn \prop_pop:NnN { No }
6049 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
6050 \cs_generate_variant:Nn \prop_gpop:NnN { No }
6051 \cs_generate_variant:Nn \prop_gpop:NnN { c , co }

```

(End definition for \prop_pop:NnN and others. These functions are documented on page ??.)

\prop_pop:NnNTF Popping an item from a property list, keeping track of whether the key was present or not, is implemented as a conditional. If the key was missing, neither the property list, nor the token list are altered. Otherwise, **\prg_return_true:** is used after the assignments.

```

\prop_gpop:NnNTF
\prop_gpop:cnNTF
6052 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
6053 {
6054   \__prop_split:NnTF #1 {#2}
6055   {
6056     \tl_set:Nn #3 {##2}
6057     \tl_set:Nn #1 { ##1 ##3 }
6058     \prg_return_true:
6059   }
6060   { \prg_return_false: }

```

```

6061 }
6062 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
6063 {
6064   \__prop_split:NnTF #1 {#2}
6065   {
6066     \tl_set:Nn #3 {##2}
6067     \tl_gset:Nn #1 { ##1 ##3 }
6068     \prg_return_true:
6069   }
6070   { \prg_return_false: }
6071 }
6072 \cs_generate_variant:Nn \prop_pop:NnNT { c }
6073 \cs_generate_variant:Nn \prop_pop:NnNF { c }
6074 \cs_generate_variant:Nn \prop_pop:NnNTF { c }
6075 \cs_generate_variant:Nn \prop_gpop:NnNT { c }
6076 \cs_generate_variant:Nn \prop_gpop:NnNF { c }
6077 \cs_generate_variant:Nn \prop_gpop:NnNTF { c }

```

(End definition for \prop_pop:NnNTF and others. These functions are documented on page ??.)

\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 (__prop_strip_end:w moves the \s_obj_end marker). Otherwise concatenate the extracts ##1 and ##3 with the new key–value pair \l__prop_internal_tl. The updated entry is placed at the same spot as the original *<key>* in the property list, preserving the order of entries.

```

\prop_put:cnV 6078 \cs_new_protected_nopar:Npn \prop_put:Nnn { \__prop_put:NNnn \tl_set:Nx }
\prop_put:cno 6079 \cs_new_protected_nopar:Npn \prop_gput:Nnn { \__prop_put:NNnn \tl_gset:Nx }
\prop_put:cnx 6080 \cs_new_protected:Npn \__prop_put:NNnn #1#2#3#4
\prop_put:cVn 6081 {
\prop_put:cVV 6082   \tl_set:Nn \l__prop_internal_tl
\prop_put:con 6083   {
\prop_put:coo 6084     \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
6085     \s__prop { \exp_not:n {#4} }
6086   }
\prop_gput:NnV 6087   \__prop_split:NnTF #2 {#3}
\prop_gput:Nno 6088   { #1 #2 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } }
\prop_gput:Nnx 6089   {
\prop_gput:NVn 6090     #1 #2
\prop_gput:NVV 6091     {
\prop_gput:Non 6092       \exp_after:wN \__prop_strip_end:w #2
\prop_gput:Noo 6093       \l__prop_internal_tl \s_obj_end
\prop_gput:cn 6094     }
\prop_gput:cnV 6095   }
\prop_gput:cno 6096 }
\prop_gput:cnx 6097 \cs_generate_variant:Nn \prop_put:Nnn
\prop_gput:cVn 6098 { NnV , Nno , Nnx , NV , NVV , No , Noo }
\prop_gput:cVV
\prop_gput:con
\prop_gput:coo
\__prop_put:NNnn

```

```

6099 \cs_generate_variant:Nn \prop_put:Nnn
6100 { c , cnV , cno , cnx , cV , cVV , co , coo }
6101 \cs_generate_variant:Nn \prop_gput:Nnn
6102 { NnV , Nno , Nnx , NV , NVV , No , Noo }
6103 \cs_generate_variant:Nn \prop_gput:Nnn
6104 { c , cnV , cno , cnx , cV , cVV , co , coo }

```

(End definition for `\prop_put:Nnn` and others. These functions are documented on page ??.)

```

\prop_put_if_new:Nnn
\prop_put_if_new:cnn
\prop_gput_if_new:Nnn
\prop_gput_if_new:cnn
\__prop_put_if_new:NNnn

```

Adding conditionally also splits. If the key is already present, the three brace groups given by `__prop_split:NnTF` are removed. If the key is new, then the value is added, being careful to convert the key to a string using `\tl_to_str:n`.

```

6105 \cs_new_protected_nopar:Npn \prop_put_if_new:Nnn
6106 { \__prop_put_if_new:NNnn \tl_set:Nx }
6107 \cs_new_protected_nopar:Npn \prop_gput_if_new:Nnn
6108 { \__prop_put_if_new:NNnn \tl_gset:Nx }
6109 \cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4
6110 {
6111   \tl_set:Nn \l__prop_internal_tl
6112   {
6113     \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
6114     \s__prop \exp_not:n { {#4} }
6115   }
6116   \__prop_split:NnTF #2 {#3}
6117   { }
6118   {
6119     #1 #2
6120     {
6121       \exp_after:wN \__prop_strip_end:w #2
6122       \l__prop_internal_tl \s_obj_end
6123     }
6124   }
6125 }
6126 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }
6127 \cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }

```

(End definition for `\prop_put_if_new:Nnn` and `\prop_put_if_new:cnn`. These functions are documented on page ??.)

13.3 Property list conditionals

```

\prop_if_exist_p:N
\prop_if_exist_p:c
\prop_if_exist:NTF
\prop_if_exist:cTF

```

Copies of the `cs` functions defined in `l3basics`.

```

6128 \prg_new_eq_conditional:NNn \prop_if_exist:N \cs_if_exist:N { TF , T , F , p }
6129 \prg_new_eq_conditional:NNn \prop_if_exist:c \cs_if_exist:c { TF , T , F , p }

```

(End definition for `\prop_if_exist:N` and `\prop_if_exist:c`. These functions are documented on page ??.)

```

\prop_if_empty_p:N
\prop_if_empty_p:c
\prop_if_empty:NTF
\prop_if_empty:cTF

```

Same test as for token lists.

```

6130 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
6131 {
6132   \tl_if_eq:NNTF #1 \c_empty_prop

```

```

6133     \prg_return_true: \prg_return_false:
6134 }
6135 \cs_generate_variant:Nn \prop_if_empty_p:N { c }
6136 \cs_generate_variant:Nn \prop_if_empty:NT { c }
6137 \cs_generate_variant:Nn \prop_if_empty:NF { c }
6138 \cs_generate_variant:Nn \prop_if_empty:NTF { c }

```

(End definition for `\prop_if_empty:N` and `\prop_if_empty:c`. These functions are documented on page ??.)

```

\prop_if_in_p:Nn
\prop_if_in_p:NV
\prop_if_in_p:No
\prop_if_in_p:cn
\prop_if_in_p:cV
\prop_if_in_p:co
\prop_if_in:NnTF
\prop_if_in:NVTF
\prop_if_in:NoTF
\prop_if_in:cnTF
\prop_if_in:cVTF
\prop_if_in:coTF
__prop_if_in:nwn
__prop_if_in:N

```

Testing expandably if a key is in a property list requires to go through the key-value pairs one by one. This is rather slow, and a faster test would be

```

\prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
{
  \@@_split:NnTF #1 {#2}
  { \prg_return_true: }
  { \prg_return_false: }
}

```

but `__prop_split:NnTF` is non-expandable.

Instead, the key is compared to each key in turn using `\str_if_eq_x:nn`, which is expandable. To terminate the mapping, we append to the property list the key that is searched for. This second `\tl_to_str:n` is not expanded at the start, but only when included in the `\str_if_eq_x:nn`. It cannot make the breaking mechanism choke, because the arbitrary token list material is enclosed in braces. The second argument of `__prop_if_in:nwn` is most often empty. When the *<key>* is found in the list, `__prop_if_in:N` receives `\s__prop` or `\s_obj_end`, 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.

```

6139 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
6140 {
6141   \exp_last_unbraced:Noo \__prop_if_in:nwn { \tl_to_str:n {#2} } #1
6142   \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
6143   \q_recursion_tail
6144   \__prg_break_point:
6145 }
6146 \cs_new:Npn \__prop_if_in:nwn #1#2 \__prop_pair:wn #3 \s__prop #4
6147 {
6148   \str_if_eq_x:nnTF {#1} {#3}
6149   { \__prop_if_in:N }
6150   { \__prop_if_in:nwn {#1} }
6151 }
6152 \cs_new:Npn \__prop_if_in:N #1
6153 {
6154   \if_meaning:w \q_recursion_tail #1
6155   \prg_return_false:
6156   \else:
6157     \prg_return_true:

```

```

6158     \fi:
6159     \__prg_break:
6160 }
6161 \cs_generate_variant:Nn \prop_if_in_p:Nn { NV , No }
6162 \cs_generate_variant:Nn \prop_if_in_p:Nn { c , cV , co }
6163 \cs_generate_variant:Nn \prop_if_in:NnT { NV , No }
6164 \cs_generate_variant:Nn \prop_if_in:NnT { c , cV , co }
6165 \cs_generate_variant:Nn \prop_if_in:NnF { NV , No }
6166 \cs_generate_variant:Nn \prop_if_in:NnF { c , cV , co }
6167 \cs_generate_variant:Nn \prop_if_in:NnTF { NV , No }
6168 \cs_generate_variant:Nn \prop_if_in:NnTF { c , cV , co }

```

(End definition for `\prop_if_in:Nn` and others. These functions are documented on page ??.)

13.4 Recovering values from property lists with branching

`\prop_get:NnNTF` Getting the value corresponding to a key, keeping track of whether the key was present or not, is implemented as a conditional (with side effects). If the key was absent, the token list is not altered.

```

\prop_get:NnNTF 6169 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
\prop_get:NVNTF 6170 {
\prop_get:NoNTF 6171     \__prop_split:NnTF #1 {#2}
\prop_get:cVNTF 6172     {
\prop_get:coNTF 6173         \tl_set:Nn #3 {##2}
6174         \prg_return_true:
6175     }
6176     { \prg_return_false: }
6177 }
6178 \cs_generate_variant:Nn \prop_get:NnNT { NV , No }
6179 \cs_generate_variant:Nn \prop_get:NnNF { NV , No }
6180 \cs_generate_variant:Nn \prop_get:NnNTF { NV , No }
6181 \cs_generate_variant:Nn \prop_get:NnNT { c , cV , co }
6182 \cs_generate_variant:Nn \prop_get:NnNF { c , cV , co }
6183 \cs_generate_variant:Nn \prop_get:NnNTF { c , cV , co }

```

(End definition for `\prop_get:NnNTF` and others. These functions are documented on page ??.)

13.5 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 can only consist of `\s__prop` and/or `\s_obj_end`.

```

\__prop_map_function:Nwwn 6184 \cs_new:Npn \prop_map_function:NN #1#2
6185 {
6186     \exp_last_unbraced:NNo \__prop_map_function:Nwwn #2 #1
6187     \__prop_pair:wn \q_recursion_tail \s__prop { }
6188     \__prg_break_point:Nn \prop_map_break: { }
6189 }
6190 \cs_new:Npn \__prop_map_function:Nwwn #1#2 \__prop_pair:wn #3 \s__prop #4

```



```

6191 {
6192   \if_meaning:w \q_recursion_tail #3
6193   \exp_after:wN \prop_map_break:
6194   \fi:
6195   #1 {#3} {#4}
6196   \__prop_map_function:Nwwn #1
6197 }
6198 \cs_generate_variant:Nn \prop_map_function:NN { Nc }
6199 \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 ??.)

\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.

```

6200 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
6201 {
6202   \cs_gset_eq:cN
6203   { __prg_map_ \int_use:N \g__prg_map_int :wn } \__prop_pair:wn
6204   \int_gincr:N \g__prg_map_int
6205   \cs_gset:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
6206   #1
6207   \__prg_break_point:Nn \prop_map_break:
6208   {
6209     \int_gdecr:N \g__prg_map_int
6210     \cs_gset_eq:Nc \__prop_pair:wn
6211     { __prg_map_ \int_use:N \g__prg_map_int :wn }
6212   }
6213 }
6214 \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 ??.)

\prop_map_break: The break statements are based on the general __prg_map_break:Nn.
\prop_map_break:n

```

6215 \cs_new_nopar:Npn \prop_map_break:
6216 { \__prg_map_break:Nn \prop_map_break: { } }
6217 \cs_new_nopar:Npn \prop_map_break:n
6218 { \__prg_map_break:Nn \prop_map_break: }

```

(End definition for \prop_map_break:. This function is documented on page 126.)

13.6 Viewing property lists

\prop_show:N Apply the general __msg_show_variable:Nnn. Contrarily to sequences and comma lists, we use __msg_show_item:nn to format both the key and the value for each pair.

```

6219 \cs_new_protected:Npn \prop_show:N #1
6220 {
6221   \__msg_show_variable:Nnn #1 { prop }

```

```

6222     { \prop_map_function:NN #1 \__msg_show_item:nn }
6223   }
6224   \cs_generate_variant:Nn \prop_show:N { c }
(End definition for \prop_show:N and \prop_show:c. These functions are documented on page ??.)
6225 \</initex | package>

```

14 l3box implementation

```

6226 \*initex | package>
6227 \@@=box>
6228 \*package>
6229 \ProvidesExplPackage
6230   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
6231   \__expl_package_check:
6232 \</package>

```

The code in this module is very straight forward so I'm not going to comment it very extensively.

14.1 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
6233 \*package>
6234 \cs_new_protected:Npn \box_new:N #1
6235   {
6236     \__chk_if_free_cs:N #1
6237     \newbox #1
6238   }
6239 \</package>
6240 \cs_generate_variant:Nn \box_new:N { c }

\box_clear:N Clear a \box register.
\box_clear:c
\box_gclear:N
\box_gclear:c
6241 \cs_new_protected:Npn \box_clear:N #1
6242   { \box_set_eq:NN #1 \c_empty_box }
6243 \cs_new_protected:Npn \box_gclear:N #1
6244   { \box_gset_eq:NN #1 \c_empty_box }
6245 \cs_generate_variant:Nn \box_clear:N { c }
6246 \cs_generate_variant:Nn \box_gclear:N { c }

\box_clear_new:N Clear or new.
\box_clear_new:c
\box_gclear_new:N
\box_gclear_new:c
6247 \cs_new_protected:Npn \box_clear_new:N #1
6248   { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
6249 \cs_new_protected:Npn \box_gclear_new:N #1
6250   { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
6251 \cs_generate_variant:Nn \box_clear_new:N { c }
6252 \cs_generate_variant:Nn \box_gclear_new:N { c }

```

<code>\box_set_eq:NN</code>	Assigning the contents of a box to be another box.
<code>\box_set_eq:cN</code>	6253 <code>\cs_new_protected:Npn \box_set_eq:NN #1#2</code>
<code>\box_set_eq:Nc</code>	6254 <code>{ \tex_setbox:D #1 \tex_copy:D #2 }</code>
<code>\box_set_eq:cc</code>	6255 <code>\cs_new_protected:Npn \box_gset_eq:NN</code>
<code>\box_gset_eq:NN</code>	6256 <code>{ \tex_global:D \box_set_eq:NN }</code>
<code>\box_gset_eq:cN</code>	6257 <code>\cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }</code>
<code>\box_gset_eq:Nc</code>	6258 <code>\cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }</code>
<code>\box_gset_eq:cc</code>	
<code>\box_set_eq_clear:NN</code>	Assigning the contents of a box to be another box. This clears the second box globally
<code>\box_set_eq_clear:cN</code>	(that's how TeX does it).
<code>\box_set_eq_clear:Nc</code>	6259 <code>\cs_new_protected:Npn \box_set_eq_clear:NN #1#2</code>
<code>\box_set_eq_clear:cc</code>	6260 <code>{ \tex_setbox:D #1 \tex_box:D #2 }</code>
<code>\box_gset_eq_clear:NN</code>	6261 <code>\cs_new_protected:Npn \box_gset_eq_clear:NN</code>
<code>\box_gset_eq_clear:cN</code>	6262 <code>{ \tex_global:D \box_set_eq_clear:NN }</code>
<code>\box_gset_eq_clear:Nc</code>	6263 <code>\cs_generate_variant:Nn \box_set_eq_clear:NN { c , Nc , cc }</code>
<code>\box_gset_eq_clear:cc</code>	6264 <code>\cs_generate_variant:Nn \box_gset_eq_clear:NN { c , Nc , cc }</code>
<code>\box_if_exist_p:N</code>	Copies of the cs functions defined in l3basics.
<code>\box_if_exist_p:c</code>	6265 <code>\prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N { TF , T , F , p }</code>
<code>\box_if_exist:NTF</code>	6266 <code>\prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c { TF , T , F , p }</code>
<code>\box_if_exist:cTF</code>	

14.2 Measuring and setting box dimensions

<code>\box_ht:N</code>	Accessing the height, depth, and width of a $\langle box \rangle$ register.
<code>\box_ht:c</code>	6267 <code>\cs_new_eq:NN \box_ht:N \tex_ht:D</code>
<code>\box_dp:N</code>	6268 <code>\cs_new_eq:NN \box_dp:N \tex_dp:D</code>
<code>\box_dp:c</code>	6269 <code>\cs_new_eq:NN \box_wd:N \tex_wd:D</code>
<code>\box_wd:N</code>	6270 <code>\cs_generate_variant:Nn \box_ht:N { c }</code>
<code>\box_wd:c</code>	6271 <code>\cs_generate_variant:Nn \box_dp:N { c }</code>
	6272 <code>\cs_generate_variant:Nn \box_wd:N { c }</code>
<code>\box_set_ht:Nn</code>	Measuring is easy: all primitive work. These primitives are not expandable, so the derived
<code>\box_set_ht:cn</code>	functions are not either.
<code>\box_set_dp:Nn</code>	6273 <code>\cs_new_protected:Npn \box_set_dp:Nn #1#2</code>
<code>\box_set_dp:cn</code>	6274 <code>{ \box_dp:N #1 __dim_eval:w #2 __dim_eval_end: }</code>
<code>\box_set_wd:Nn</code>	6275 <code>\cs_new_protected:Npn \box_set_ht:Nn #1#2</code>
<code>\box_set_wd:cn</code>	6276 <code>{ \box_ht:N #1 __dim_eval:w #2 __dim_eval_end: }</code>
	6277 <code>\cs_new_protected:Npn \box_set_wd:Nn #1#2</code>
	6278 <code>{ \box_wd:N #1 __dim_eval:w #2 __dim_eval_end: }</code>
	6279 <code>\cs_generate_variant:Nn \box_set_ht:Nn { c }</code>
	6280 <code>\cs_generate_variant:Nn \box_set_dp:Nn { c }</code>
	6281 <code>\cs_generate_variant:Nn \box_set_wd:Nn { c }</code>

14.3 Using boxes

`\box_use_clear:N` Using a $\langle box \rangle$. These are just TeX primitives with meaningful names.

```
\box_use_clear:c 6282 \cs_new_eq:NN \box_use_clear:N \tex_box:D
\box_use:N       6283 \cs_new_eq:NN \box_use:N \tex_copy:D
\box_use:c       6284 \cs_generate_variant:Nn \box_use_clear:N { c }
                  6285 \cs_generate_variant:Nn \box_use:N { c }
```

`\box_move_left:nn` Move box material in different directions.

```
\box_move_right:nn 6286 \cs_new_protected:Npn \box_move_left:nn #1#2
\box_move_up:nn    6287 { \tex_moveleft:D \__dim_eval:w #1 \__dim_eval_end: #2 }
\box_move_down:nn  6288 \cs_new_protected:Npn \box_move_right:nn #1#2
                  6289 { \tex_moveright:D \__dim_eval:w #1 \__dim_eval_end: #2 }
                  6290 \cs_new_protected:Npn \box_move_up:nn #1#2
                  6291 { \tex_raise:D \__dim_eval:w #1 \__dim_eval_end: #2 }
                  6292 \cs_new_protected:Npn \box_move_down:nn #1#2
                  6293 { \tex_lower:D \__dim_eval:w #1 \__dim_eval_end: #2 }
```

14.4 Box conditionals

`\if_hbox:N` The primitives for testing if a $\langle box \rangle$ is empty/void or which type of box it is.

```
\if_vbox:N 6294 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
\if_box_empty:N 6295 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
                  6296 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D
```

```
\box_if_horizontal_p:N 6297 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
\box_if_horizontal_p:c 6298 { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_horizontal:N $\underline{TF}$  6299 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
\box_if_vertical_p:N 6300 { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_vertical_p:c 6301 \cs_generate_variant:Nn \box_if_horizontal_p:N { c }
\box_if_vertical:N $\underline{TF}$  6302 \cs_generate_variant:Nn \box_if_horizontal:NT { c }
\box_if_vertical:c $\underline{TF}$  6303 \cs_generate_variant:Nn \box_if_horizontal:NF { c }
6304 \cs_generate_variant:Nn \box_if_horizontal:NTF { c }
6305 \cs_generate_variant:Nn \box_if_vertical_p:N { c }
6306 \cs_generate_variant:Nn \box_if_vertical:NT { c }
6307 \cs_generate_variant:Nn \box_if_vertical:NF { c }
6308 \cs_generate_variant:Nn \box_if_vertical:NTF { c }
```

`\box_if_empty_p:N` Testing if a $\langle box \rangle$ is empty/void.

```
\box_if_empty_p:c 6309 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
\box_if_empty:N $\underline{TF}$  6310 { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_empty:c $\underline{TF}$  6311 \cs_generate_variant:Nn \box_if_empty_p:N { c }
6312 \cs_generate_variant:Nn \box_if_empty:NT { c }
6313 \cs_generate_variant:Nn \box_if_empty:NF { c }
6314 \cs_generate_variant:Nn \box_if_empty:NTF { c }
```

(End definition for `\box_new:N` and `\box_new:c`. These functions are documented on page ??.)

14.5 The last box inserted

`\box_set_to_last:N` Set a box to the previous box.

```

\box_set_to_last:c 6315 \cs_new_protected:Npn \box_set_to_last:N #1
\box_gset_to_last:N 6316 { \tex_setbox:D #1 \tex_lastbox:D }
\box_gset_to_last:c 6317 \cs_new_protected:Npn \box_gset_to_last:N
6318 { \tex_global:D \box_set_to_last:N }
6319 \cs_generate_variant:Nn \box_set_to_last:N { c }
6320 \cs_generate_variant:Nn \box_gset_to_last:N { c }

```

(End definition for `\box_set_to_last:N` and `\box_gset_to_last:c`. These functions are documented on page ??.)

14.6 Constant boxes

`\c_empty_box` A box we never use.

```

6321 \box_new:N \c_empty_box

```

(End definition for `\c_empty_box`. This variable is documented on page 131.)

14.7 Scratch boxes

`\l_tmpa_box` Scratch boxes.

```

\l_tmpb_box 6322 \box_new:N \l_tmpa_box
\g_tmpa_box 6323 \box_new:N \l_tmpb_box
\g_tmpb_box 6324 \box_new:N \g_tmpa_box
6325 \box_new:N \g_tmpb_box

```

(End definition for `\l_tmpa_box` and others. These variables are documented on page 131.)

14.8 Viewing box contents

TeX's `\tex_showbox:D` 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 6326 \cs_new_protected:Npn \box_show:N #1
\box_show:Nnn 6327 { \box_show:Nnn #1 \c_max_int \c_max_int }
\box_show:cnn 6328 \cs_generate_variant:Nn \box_show:N { c }
6329 \cs_new_protected_nopar:Npn \box_show:Nnn
6330 { \__box_show:Nnn \c_one }
6331 \cs_generate_variant:Nn \box_show:Nnn { c }

```

(End definition for `\box_show:N` and `\box_show:c`. These functions are documented on page ??.)

`\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 6332 \cs_new_protected:Npn \box_log:N #1
\box_log:Nnn 6333 { \box_log:Nnn #1 \c_max_int \c_max_int }
\box_log:cnn 6334 \cs_generate_variant:Nn \box_log:N { c }
6335 \cs_new_protected:Npn \box_log:Nnn #1#2#3

```

```

6336 {
6337   \use:x
6338   {
6339     \etex_interactionmode:D \c_zero
6340     \_box_show:NNnn \c_zero \exp_not:N #1
6341     { \int_eval:n {#2} } { \int_eval:n {#3} }
6342     \etex_interactionmode:D
6343     = \tex_the:D \etex_interactionmode:D \scan_stop:
6344   }
6345 }
6346 \cs_generate_variant:Nn \box_log:Nnn { c }

```

(End definition for `\box_log:N` and `\box_log:c`. These functions are documented on page ??.)

`_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 `\tex_tracingonline:D` is used to control what appears in the terminal.

```

6347 \cs_new_protected:Npn \_box_show:NNnn #1#2#3#4
6348 {
6349   \group_begin:
6350   \int_set:Nn \tex_showboxbreadth:D {#3}
6351   \int_set:Nn \tex_showboxdepth:D {#4}
6352   \int_set_eq:NN \tex_tracingonline:D #1
6353   \box_if_exist:NTF #2
6354   { \tex_showbox:D \use:n {#2} }
6355   {
6356     \__msg_kernel_error:nxx { kernel } { variable-not-defined }
6357     { \token_to_str:N #2 }
6358   }
6359   \group_end:
6360 }

```

(End definition for `_box_show:NNnn`.)

14.9 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.

```

6361 \cs_new_protected:Npn \hbox:n { \tex_hbox:D \scan_stop: }

```

(End definition for `\hbox:n`. This function is documented on page 132.)

`\hbox_set:Nn`

`\hbox_set:cn`

`\hbox_gset:Nn`

`\hbox_gset:cn`

```

6362 \cs_new_protected:Npn \hbox_set:Nn #1#2 { \tex_setbox:D #1 \tex_hbox:D {#2} }
6363 \cs_new_protected:Npn \hbox_gset:Nn { \tex_global:D \hbox_set:Nn }
6364 \cs_generate_variant:Nn \hbox_set:Nn { c }
6365 \cs_generate_variant:Nn \hbox_gset:Nn { c }

```

(End definition for `\hbox_set:Nn` and `\hbox_set:cn`. These functions are documented on page ??.)

`\hbox_set_to_wd:Nnn` Storing material in a horizontal box with a specified width.

```

\hbox_set_to_wd:cnn 6366 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
\hbox_gset_to_wd:Nnn 6367 { \tex_setbox:D #1 \tex_hbox:D to \_dim_eval:w #2 \_dim_eval_end: {#3} }
\hbox_gset_to_wd:cnn 6368 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn
6369 { \tex_global:D \hbox_set_to_wd:Nnn }
6370 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
6371 \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 ??.)

`\hbox_set:Nw` Storing material in a horizontal box. This type is useful in environment definitions.

```

\hbox_set:cw 6372 \cs_new_protected:Npn \hbox_set:Nw #1
\hbox_gset:Nw 6373 { \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token }
\hbox_gset:cw 6374 \cs_new_protected:Npn \hbox_gset:Nw
\hbox_set_end: 6375 { \tex_global:D \hbox_set:Nw }
\hbox_gset_end: 6376 \cs_generate_variant:Nn \hbox_set:Nw { c }
6377 \cs_generate_variant:Nn \hbox_gset:Nw { c }
6378 \cs_new_eq:NN \hbox_set_end: \c_group_end_token
6379 \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 133.)

`\hbox_set_inline_begin:N` Renamed September 2011.

```

\hbox_set_inline_begin:c 6380 \cs_new_eq:NN \hbox_set_inline_begin:N \hbox_set:Nw
\hbox_gset_inline_begin:N 6381 \cs_new_eq:NN \hbox_set_inline_begin:c \hbox_set:cw
\hbox_gset_inline_begin:c 6382 \cs_new_eq:NN \hbox_set_inline_end: \hbox_set_end:
\hbox_set_inline_end: 6383 \cs_new_eq:NN \hbox_gset_inline_begin:N \hbox_gset:Nw
\hbox_gset_inline_end: 6384 \cs_new_eq:NN \hbox_gset_inline_begin:c \hbox_gset:cw
6385 \cs_new_eq:NN \hbox_gset_inline_end: \hbox_gset_end:

```

(End definition for `\hbox_set_inline_begin:N` and `\hbox_set_inline_begin:c`. These functions are documented on page ??.)

`\hbox_to_wd:nn` Put a horizontal box directly into the input stream.

```

\hbox_to_zero:n 6386 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
6387 { \tex_hbox:D to \_dim_eval:w #1 \_dim_eval_end: {#2} }
6388 \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 132.)

`\hbox_overlap_left:n` Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

```

\hbox_overlap_right:n 6389 \cs_new_protected:Npn \hbox_overlap_left:n #1
6390 { \hbox_to_zero:n { \tex_hss:D #1 } }
6391 \cs_new_protected:Npn \hbox_overlap_right:n #1
6392 { \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 132.)

\hbox_unpack:N Unpacking a box and if requested also clear it.

\hbox_unpack:c

\hbox_unpack_clear:N

\hbox_unpack_clear:c

```

6393 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
6394 \cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
6395 \cs_generate_variant:Nn \hbox_unpack:N { c }
6396 \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 ??.)

14.10 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*.

\vbox_top:n The following test files are used for this code: *m3box003.lvt*.

Put a vertical box directly into the input stream.

```

6397 \cs_new_protected:Npn \vbox:n #1 { \tex_vbox:D { #1 \par } }
6398 \cs_new_protected:Npn \vbox_top:n #1 { \tex_vtop:D { #1 \par } }

```

(End definition for \vbox:n. This function is documented on page 133.)

\vbox_to_ht:nn Put a vertical box directly into the input stream.

```

6399 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
6400 { \tex_vbox:D to \__dim_eval:w #1 \__dim_eval_end: { #2 \par } }
6401 \cs_new_protected:Npn \vbox_to_zero:n #1
6402 { \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 134.)

\vbox_set:Nn Storing material in a vertical box with a natural height.

```

6403 \cs_new_protected:Npn \vbox_set:Nn #1#2
6404 { \tex_setbox:D #1 \tex_vbox:D { #2 \par } }
6405 \cs_new_protected:Npn \vbox_gset:Nn { \tex_global:D \vbox_set:Nn }
6406 \cs_generate_variant:Nn \vbox_set:Nn { c }
6407 \cs_generate_variant:Nn \vbox_gset:Nn { c }

```

(End definition for \vbox_set:Nn and \vbox_set:cn. These functions are documented on page ??.)

\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.

```

6408 \cs_new_protected:Npn \vbox_set_top:Nn #1#2
6409 { \tex_setbox:D #1 \tex_vtop:D { #2 \par } }
6410 \cs_new_protected:Npn \vbox_gset_top:Nn
6411 { \tex_global:D \vbox_set_top:Nn }
6412 \cs_generate_variant:Nn \vbox_set_top:Nn { c }
6413 \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 ??.)

`\vbox_set_to_ht:Nnn` Storing material in a vertical box with a specified height.

`\vbox_set_to_ht:cnn` 6414 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3

`\vbox_gset_to_ht:Nnn` 6415 { \tex_setbox:D #1 \tex_vbox:D to _dim_eval:w #2 _dim_eval_end: { #3 \par } }

`\vbox_gset_to_ht:cnn` 6416 \cs_new_protected:Npn \vbox_gset_to_ht:Nnn

6417 { \tex_global:D \vbox_set_to_ht:Nnn }

6418 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }

6419 \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 ??.)

`\vbox_set:Nw` Storing material in a vertical box. This type is useful in environment definitions.

`\vbox_set:cw` 6420 \cs_new_protected:Npn \vbox_set:Nw #1

`\vbox_gset:Nw` 6421 { \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }

`\vbox_gset:cw` 6422 \cs_new_protected:Npn \vbox_gset:Nw

`\vbox_set_end:` 6423 { \tex_global:D \vbox_set:Nw }

`\vbox_gset_end:` 6424 \cs_generate_variant:Nn \vbox_set:Nw { c }

6425 \cs_generate_variant:Nn \vbox_gset:Nw { c }

6426 \cs_new_protected:Npn \vbox_set_end:

6427 {

6428 \par

6429 \c_group_end_token

6430 }

6431 \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 134.)

`\vbox_set_inline_begin:N` Renamed September 2011.

`\vbox_set_inline_begin:c` 6432 \cs_new_eq:NN \vbox_set_inline_begin:N \vbox_set:Nw

`\vbox_gset_inline_begin:N` 6433 \cs_new_eq:NN \vbox_set_inline_begin:c \vbox_set:cw

`\vbox_gset_inline_begin:c` 6434 \cs_new_eq:NN \vbox_set_inline_end: \vbox_set_end:

`\vbox_set_inline_end:` 6435 \cs_new_eq:NN \vbox_gset_inline_begin:N \vbox_gset:Nw

`\vbox_gset_inline_end:` 6436 \cs_new_eq:NN \vbox_gset_inline_begin:c \vbox_gset:cw

6437 \cs_new_eq:NN \vbox_gset_inline_end: \vbox_gset_end:

(End definition for \vbox_set_inline_begin:N and \vbox_set_inline_begin:c. These functions are documented on page ??.)

`\vbox_unpack:N` Unpacking a box and if requested also clear it.

`\vbox_unpack:c` 6438 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D

`\vbox_unpack_clear:N` 6439 \cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D

`\vbox_unpack_clear:c` 6440 \cs_generate_variant:Nn \vbox_unpack:N { c }

6441 \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 ??.)

`\vbox_set_split_to_ht:NNn` Splitting a vertical box in two.

6442 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3

6443 { \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 134.)

6444 </initex | package>

15 l3coffins Implementation

```

6445 <*initex | package>
6446 <@@=coffin>
6447 <*package>
6448 \ProvidesExplPackage
6449   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
6450   \__expl_package_check:
6451 </package>

```

15.1 Coffins: data structures and general variables

`\l__coffin_internal_box` Scratch variables.

```

\l__coffin_internal_dim 6452 \box_new:N \l__coffin_internal_box
\l__coffin_internal_tl 6453 \dim_new:N \l__coffin_internal_dim
6454 \tl_new:N \l__coffin_internal_tl
(End definition for \l__coffin_internal_box. This function 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.

```

6455 \prop_new:N \c__coffin_corners_prop
6456 \prop_put:Nnn \c__coffin_corners_prop { tl } { { 0 pt } { 0 pt } }
6457 \prop_put:Nnn \c__coffin_corners_prop { tr } { { 0 pt } { 0 pt } }
6458 \prop_put:Nnn \c__coffin_corners_prop { bl } { { 0 pt } { 0 pt } }
6459 \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.

```

6460 \prop_new:N \c__coffin_poles_prop
6461 \tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 0 pt } { 1000 pt } }
6462 \prop_put:Nno \c__coffin_poles_prop { l } { \l__coffin_internal_tl }
6463 \prop_put:Nno \c__coffin_poles_prop { hc } { \l__coffin_internal_tl }
6464 \prop_put:Nno \c__coffin_poles_prop { r } { \l__coffin_internal_tl }
6465 \tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 1000 pt } { 0 pt } }
6466 \prop_put:Nno \c__coffin_poles_prop { b } { \l__coffin_internal_tl }
6467 \prop_put:Nno \c__coffin_poles_prop { vc } { \l__coffin_internal_tl }
6468 \prop_put:Nno \c__coffin_poles_prop { t } { \l__coffin_internal_tl }
6469 \prop_put:Nno \c__coffin_poles_prop { B } { \l__coffin_internal_tl }
6470 \prop_put:Nno \c__coffin_poles_prop { H } { \l__coffin_internal_tl }
6471 \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 6472 \fp_new:N \l__coffin_slope_x_fp
6473 \fp_new:N \l__coffin_slope_y_fp
(End definition for \l__coffin_slope_x_fp. This function is documented on page ??.)

```

`\l__coffin_error_bool` For propagating errors so that parts of the code can work around them.

```

6474 \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 from those requested in an alignment for the positions of the handles.

```
6475 \dim_new:N \l__coffin_offset_x_dim
6476 \dim_new:N \l__coffin_offset_y_dim
```

(End definition for \l__coffin_offset_x_dim. This function is documented on page ??.)

\l__coffin_pole_a_tl Needed for finding the intersection of two poles.

```
\l__coffin_pole_b_tl
6477 \tl_new:N \l__coffin_pole_a_tl
6478 \tl_new:N \l__coffin_pole_b_tl
```

(End definition for \l__coffin_pole_a_tl. This function is documented on page ??.)

\l__coffin_x_dim For calculating intersections and so forth.

```
\l__coffin_y_dim
6479 \dim_new:N \l__coffin_x_dim
6480 \dim_new:N \l__coffin_y_dim
\l__coffin_x_prime_dim
6481 \dim_new:N \l__coffin_x_prime_dim
\l__coffin_y_prime_dim
6482 \dim_new:N \l__coffin_y_prime_dim
```

(End definition for \l__coffin_x_dim. This function is documented on page ??.)

15.2 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
6483 \prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
6484 {
6485   \cs_if_exist:NTF #1
6486   {
6487     \cs_if_exist:cTF { l__coffin_poles_ \__int_value:w #1 _prop }
6488     { \prg_return_true: }
6489     { \prg_return_false: }
6490   }
6491   { \prg_return_false: }
6492 }
6493 \cs_generate_variant:Nn \coffin_if_exist_p:N { c }
6494 \cs_generate_variant:Nn \coffin_if_exist:NT { c }
6495 \cs_generate_variant:Nn \coffin_if_exist:NF { c }
6496 \cs_generate_variant:Nn \coffin_if_exist:NTF { c }
```

(End definition for \coffin_if_exist:N and \coffin_if_exist:c. These functions are documented on page ??.)

`__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.

```

6497 \cs_new_protected:Npn \__coffin_if_exist:NT #1#2
6498 {
6499   \coffin_if_exist:NTF #1
6500     { #2 }
6501     {
6502       \__msg_kernel_error:nxx { kernel } { unknown-coffin }
6503       { \token_to_str:N #1 }
6504     }
6505 }

```

(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
6506 \cs_new_protected:Npn \coffin_clear:N #1
6507 {
6508   \__coffin_if_exist:NT #1
6509   {
6510     \box_clear:N #1
6511     \__coffin_reset_structure:N #1
6512   }
6513 }
6514 \cs_generate_variant:Nn \coffin_clear:N { c }

```

(End definition for \coffin_clear:N and \coffin_clear:c. These functions are documented on page ??.)

`\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.

```

6515 \cs_new_protected:Npn \coffin_new:N #1
6516 {
6517   \box_new:N #1
6518   \prop_clear_new:c { l__coffin_corners_ } \__int_value:w #1 _prop }
6519   \prop_clear_new:c { l__coffin_poles_ } \__int_value:w #1 _prop }
6520   \prop_gset_eq:cN { l__coffin_corners_ } \__int_value:w #1 _prop }
6521   \c__coffin_corners_prop
6522   \prop_gset_eq:cN { l__coffin_poles_ } \__int_value:w #1 _prop }
6523   \c__coffin_poles_prop
6524 }
6525 \cs_generate_variant:Nn \coffin_new:N { c }

```

(End definition for \coffin_new:N and \coffin_new:c. These functions are documented on page ??.)

`\hcoffin_set:Nn` Horizontal coffins are relatively easy: set the appropriate box, reset the structures then
`\hcoffin_set:cn` update the handle positions.

```

6526 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
6527 {

```

```

6528   \_coffin_if_exist:NT #1
6529   {
6530     \hbox_set:Nn #1
6531     {
6532       \color_group_begin:
6533       \color_ensure_current:
6534       #2
6535       \color_group_end:
6536     }
6537     \_coffin_reset_structure:N #1
6538     \_coffin_update_poles:N #1
6539     \_coffin_update_corners:N #1
6540   }
6541 }
6542 \cs_generate_variant:Nn \hcoffin_set:Nn { c }

```

(End definition for `\hcoffin_set:Nn` and `\hcoffin_set:cn`. These functions are documented on page ??.)

`\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 *T_EX by Topic* for discussion of the `\vtop` primitive, used to do the measuring).

```

6543 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
6544 {
6545   \_coffin_if_exist:NT #1
6546   {
6547     \vbox_set:Nn #1
6548     {
6549       \dim_set:Nn \tex_hsize:D {#2}
6550     }
6551     (*package)
6552     \dim_set_eq:NN \linewidth \tex_hsize:D
6553     \dim_set_eq:NN \columnwidth \tex_hsize:D
6554     (/package)
6555     \color_group_begin:
6556     #3
6557     \color_group_end:
6558   }
6559   \_coffin_reset_structure:N #1
6560   \_coffin_update_poles:N #1
6561   \_coffin_update_corners:N #1
6562   \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
6563   \_coffin_set_pole:Nnx #1 { T }
6564   {
6565     { 0 pt }
6566     { \dim_eval:n { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box } }
6567     { 1000 pt }
6568     { 0 pt }
6569   }

```

```
6569         \box_clear:N \l__coffin_internal_box
```

```
6570     }
```

```
6571 }
```

```
6572 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }
```

(End definition for \vcoffin_set:Nnn and \vcoffin_set:cnn. These functions are documented on page ??.)

\hcoffin_set:Nw These are the “begin”/“end” versions of the above: watch the grouping!

\hcoffin_set:cw

\hcoffin_set_end:

```
6573 \cs_new_protected:Npn \hcoffin_set:Nw #1
6574 {
6575     \__coffin_if_exist:NT #1
6576     {
6577         \hbox_set:Nw #1 \color_group_begin: \color_ensure_current:
6578         \cs_set_protected_nopar:Npn \hcoffin_set_end:
6579         {
6580             \color_group_end:
6581             \hbox_set_end:
6582             \__coffin_reset_structure:N #1
6583             \__coffin_update_poles:N #1
6584             \__coffin_update_corners:N #1
6585         }
6586     }
6587 }
6588 \cs_new_protected_nopar:Npn \hcoffin_set_end: { }
6589 \cs_generate_variant:Nn \hcoffin_set:Nw { c }
```

(End definition for \hcoffin_set:Nw and \hcoffin_set:cw. These functions are documented on page 137.)

\vcoffin_set:Nnw

\vcoffin_set:cnw

\vcoffin_set_end:

The same for vertical coffins.

```
6590 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
6591 {
6592     \__coffin_if_exist:NT #1
6593     {
6594         \vbox_set:Nw #1
6595         \dim_set:Nn \tex_hsize:D {#2}
6596         <*package>
6597             \dim_set_eq:NN \linewidth \tex_hsize:D
6598             \dim_set_eq:NN \columnwidth \tex_hsize:D
6599         </package>
6600         \color_group_begin: \color_ensure_current:
6601         \cs_set_protected:Npn \vcoffin_set_end:
6602         {
6603             \color_group_end:
6604             \vbox_set_end:
6605             \__coffin_reset_structure:N #1
6606             \__coffin_update_poles:N #1
6607             \__coffin_update_corners:N #1
6608             \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
6609             \__coffin_set_pole:Nnx #1 { T }
```

```

6610         {
6611             { 0 pt }
6612             {
6613                 \dim_eval:n { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
6614             }
6615             { 1000 pt }
6616             { 0 pt }
6617         }
6618         \box_clear:N \l__coffin_internal_box
6619     }
6620 }
6621 }
6622 \cs_new_protected_nopar:Npn \vcoffin_set_end: { }
6623 \cs_generate_variant:Nn \vcoffin_set:Nnw { c }

```

(End definition for \vcoffin_set:Nnw and \vcoffin_set:cnw. These functions are documented on page 137.)

\coffin_set_eq:NN Setting two coffins equal is just a wrapper around other functions.

```

\coffin_set_eq:Nc 6624 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
\coffin_set_eq:cN 6625 {
\coffin_set_eq:cc 6626     \__coffin_if_exist:NT #1
6627     {
6628         \box_set_eq:NN #1 #2
6629         \__coffin_set_eq_structure:NN #1 #2
6630     }
6631 }
6632 \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 ??.)

\c_empty_coffin

\l__coffin_aligned_coffin

\l__coffin_aligned_internal_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.

```

6633 \coffin_new:N \c_empty_coffin
6634 \hbox_set:Nn \c_empty_coffin { }
6635 \coffin_new:N \l__coffin_aligned_coffin
6636 \coffin_new:N \l__coffin_aligned_internal_coffin

```

(End definition for \c_empty_coffin. This function is documented on page ??.)

\l_tmpa_coffin

\l_tmpb_coffin

The usual scratch space.

```

6637 \coffin_new:N \l_tmpa_coffin
6638 \coffin_new:N \l_tmpb_coffin

```

(End definition for \l_tmpa_coffin and \l_tmpb_coffin. These variables are documented on page 139.)

15.3 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

\coffin_ht:N 6639 \cs_new_eq:NN \coffin_dp:N \box_dp:N

\coffin_ht:c 6640 \cs_new_eq:NN \coffin_dp:c \box_dp:c

\coffin_wd:N 6641 \cs_new_eq:NN \coffin_ht:N \box_ht:N

\coffin_wd:c 6642 \cs_new_eq:NN \coffin_ht:c \box_ht:c

6643 \cs_new_eq:NN \coffin_wd:N \box_wd:N

6644 \cs_new_eq:NN \coffin_wd:c \box_wd:c

(End definition for \coffin_dp:N and others. These functions are documented on page ??.)

15.4 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.

6645 \cs_new_protected:Npn __coffin_get_pole:NnN #1#2#3

6646 {

6647 \prop_get:cnNF

6648 { l__coffin_poles_ __int_value:w #1 _prop } {#2} #3

6649 {

6650 __msg_kernel_error:nxxx { kernel } { unknown-coffin-pole }

6651 {#2} { \token_to_str:N #1 }

6652 \tl_set:Nn #3 { { 0 pt } { 0 pt } { 0 pt } { 0 pt } }

6653 }

6654 }

(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.

6655 \cs_new_protected:Npn __coffin_reset_structure:N #1

6656 {

6657 \prop_set_eq:cN { l__coffin_corners_ __int_value:w #1 _prop }

6658 \c__coffin_corners_prop

6659 \prop_set_eq:cN { l__coffin_poles_ __int_value:w #1 _prop }

6660 \c__coffin_poles_prop

6661 }

(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 6662 \cs_new_protected:Npn __coffin_set_eq_structure:NN #1#2

6663 {

6664 \prop_set_eq:cc { l__coffin_corners_ __int_value:w #1 _prop }

6665 { l__coffin_corners_ __int_value:w #2 _prop }

6666 \prop_set_eq:cc { l__coffin_poles_ __int_value:w #1 _prop }

6667 { l__coffin_poles_ __int_value:w #2 _prop }

6668 }

6669 \cs_new_protected:Npn __coffin_gset_eq_structure:NN #1#2

6670 {


```

6671 \prop_gset_eq:cc { l__coffin_corners_ \__int_value:w #1 _prop }
6672 { l__coffin_corners_ \__int_value:w #2 _prop }
6673 \prop_gset_eq:cc { l__coffin_poles_ \__int_value:w #1 _prop }
6674 { l__coffin_poles_ \__int_value:w #2 _prop }
6675 }

```

(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
\coffin_set_horizontal_pole:cnn
\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn
\__coffin_set_pole:Nnn
\__coffin_set_pole:Nnx

```

Setting the pole of a coffin at the user/designer level requires a bit more care. The idea here is to provide a reasonable interface to the system, then to do the setting with full expansion. The three-argument version is used internally to do a direct setting.

```

6676 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
6677 {
6678   \__coffin_if_exist:NT #1
6679   {
6680     \__coffin_set_pole:Nnx #1 {#2}
6681     {
6682       { 0 pt } { \dim_eval:n {#3} }
6683       { 1000 pt } { 0 pt }
6684     }
6685   }
6686 }
6687 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
6688 {
6689   \__coffin_if_exist:NT #1
6690   {
6691     \__coffin_set_pole:Nnx #1 {#2}
6692     {
6693       { \dim_eval:n {#3} } { 0 pt }
6694       { 0 pt } { 1000 pt }
6695     }
6696   }
6697 }
6698 \cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
6699 { \prop_put:cnn { l__coffin_poles_ \__int_value:w #1 _prop } {#2} {#3} }
6700 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
6701 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
6702 \cs_generate_variant:Nn \__coffin_set_pole:Nnn { Nnx }

```

(End definition for \coffin_set_horizontal_pole:Nnn and \coffin_set_horizontal_pole:cnn. These functions are documented on page ??.)

```
\__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 $\text{T}_{\text{E}}\text{X}$ box.

```

6703 \cs_new_protected:Npn \__coffin_update_corners:N #1
6704 {
6705   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tl }
6706   { { 0 pt } { \dim_use:N \box_ht:N #1 } }
6707   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tr }
6708   { { \dim_use:N \box_wd:N #1 } { \dim_use:N \box_ht:N #1 } }

```

```

6709 \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { bl }
6710 { { 0 pt } { \dim_eval:n { - \box_dp:N #1 } } }
6711 \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { br }
6712 { { \dim_use:N \box_wd:N #1 } { \dim_eval:n { - \box_dp:N #1 } } }
6713 }

```

(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.

```

6714 \cs_new_protected:Npn \__coffin_update_poles:N #1
6715 {
6716   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { hc }
6717   {
6718     { \dim_eval:n { 0.5 \box_wd:N #1 } }
6719     { 0 pt } { 0 pt } { 1000 pt }
6720   }
6721   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { r }
6722   {
6723     { \dim_use:N \box_wd:N #1 }
6724     { 0 pt } { 0 pt } { 1000 pt }
6725   }
6726   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { vc }
6727   {
6728     { 0 pt }
6729     { \dim_eval:n { ( \box_ht:N #1 - \box_dp:N #1 ) / 2 } }
6730     { 1000 pt }
6731     { 0 pt }
6732   }
6733   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { t }
6734   {
6735     { 0 pt }
6736     { \dim_use:N \box_ht:N #1 }
6737     { 1000 pt }
6738     { 0 pt }
6739   }
6740   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { b }
6741   {
6742     { 0 pt }
6743     { \dim_eval:n { - \box_dp:N #1 } }
6744     { 1000 pt }
6745     { 0 pt }
6746   }
6747 }

```

(End definition for `__coffin_update_poles:N`. This function is documented on page ??.)

15.5 Coffins: calculation of pole intersections

```

__coffin_calculate_intersection:Nnn
__coffin_calculate_intersection:nnnnnnnn
__coffin_calculate_intersection_aux:nnnnnN

```

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.

```

6748 \cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
6749 {
6750   \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
6751   \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
6752   \bool_set_false:N \l__coffin_error_bool
6753   \exp_last_two_unbraced:Noo
6754   \__coffin_calculate_intersection:nnnnnnnn
6755   \l__coffin_pole_a_tl \l__coffin_pole_b_tl
6756   \bool_if:NT \l__coffin_error_bool
6757   {
6758     \__msg_kernel_error:nn { kernel } { no-pole-intersection }
6759     \dim_zero:N \l__coffin_x_dim
6760     \dim_zero:N \l__coffin_y_dim
6761   }
6762 }

```

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.

```

6763 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
6764   #1#2#3#4#5#6#7#8
6765 {
6766   \dim_compare:nNnTF {#3} = { \c_zero_dim }

```

The case where the first pole is vertical. So the x -component of the intersection will be at a . There is then a test on the second pole: if it is also vertical then there is an error.

```

6767   {
6768     \dim_set:Nn \l__coffin_x_dim {#1}
6769     \dim_compare:nNnTF {#7} = { \c_zero_dim
6770       { \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.

```

6771   {
6772     \dim_compare:nNnTF {#8} = { \c_zero_dim
6773       { \dim_set:Nn \l__coffin_y_dim {#6} }
6774       {
6775         \__coffin_calculate_intersection_aux:nnnnnN

```

```

6776         {#1} {#5} {#6} {#7} {#8} \l__coffin_y_dim
6777     }
6778 }
6779 }

```

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.

```

6780 {
6781     \dim_compare:nNnTF {#4} = \c_zero_dim
6782     {
6783         \dim_set:Nn \l__coffin_y_dim {#2}
6784         \dim_compare:nNnTF {#8} = { \c_zero_dim }
6785         { \bool_set_true:N \l__coffin_error_bool }
6786     }
6787     \dim_compare:nNnTF {#7} = \c_zero_dim
6788     { \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.

```

6789 {
6790     \__coffin_calculate_intersection_aux:nnnnnN
6791     {#2} {#6} {#5} {#8} {#7} \l__coffin_x_dim
6792 }
6793 }
6794 }

```

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.

```

6795 {
6796     \dim_compare:nNnTF {#7} = \c_zero_dim
6797     {
6798         \dim_set:Nn \l__coffin_x_dim {#5}
6799         \__coffin_calculate_intersection_aux:nnnnnN
6800         {#5} {#1} {#2} {#3} {#4} \l__coffin_y_dim
6801     }
6802     {
6803         \dim_compare:nNnTF {#8} = \c_zero_dim
6804         {
6805             \dim_set:Nn \l__coffin_x_dim {#6}
6806             \__coffin_calculate_intersection_aux:nnnnnN
6807             {#6} {#2} {#1} {#4} {#3} \l__coffin_x_dim
6808         }

```

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.

```

6809 {

```

```

6810      \fp_set:Nn \l__coffin_slope_x_fp
6811      { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
6812      \fp_set:Nn \l__coffin_slope_y_fp
6813      { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
6814      \fp_compare:nNnTF
6815      \l__coffin_slope_x_fp = \l__coffin_slope_y_fp
6816      { \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.

```

6817      {
6818      \dim_set:Nn \l__coffin_x_dim
6819      {
6820      \fp_to_dim:n
6821      {
6822      (
6823      \dim_to_fp:n {#1} * \l__coffin_slope_x_fp
6824      - ( \dim_to_fp:n {#5} * \l__coffin_slope_y_fp )
6825      - \dim_to_fp:n {#2}
6826      + \dim_to_fp:n {#6}
6827      )
6828      /
6829      ( \l__coffin_slope_x_fp - \l__coffin_slope_y_fp )
6830      }
6831      }
6832      \__coffin_calculate_intersection_aux:nnnnnN
6833      { \l__coffin_x_dim }
6834      {#5} {#6} {#8} {#7} \l__coffin_y_dim
6835      }
6836      }
6837      }
6838      }
6839      }
6840      }

```

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.

```

6841      \cs_new_protected:Npn \__coffin_calculate_intersection_aux:nnnnnN #1#2#3#4#5#6
6842      {
6843      \dim_set:Nn #6

```

```

6844 {
6845   \fp_to_dim:n
6846   {
6847     \dim_to_fp:n {#4} *
6848     ( \dim_to_fp:n {#1} - \dim_to_fp:n {#2} ) /
6849     \dim_to_fp:n {#5}
6850     + \dim_to_fp:n {#3}
6851   }
6852 }
6853 }

```

(End definition for `_coffin_calculate_intersection:Nnn`. This function is documented on page ??.)

15.6 Aligning and typesetting of coffins

`\coffin_join:NnnNnnnn`
`\coffin_join:cnnNnnnn`
`\coffin_join:Nnnncnnnn`
`\coffin_join:cnncnnnn`

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which will have all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```

6854 \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
6855 {
6856   \_coffin_align:NnnNnnnnN
6857   #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.

```

6858   \hbox_set:Nn \l__coffin_aligned_coffin
6859   {
6860     \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
6861     { \tex_kern:D -\l__coffin_offset_x_dim }
6862     \hbox_unpack:N \l__coffin_aligned_coffin
6863     \dim_set:Nn \l__coffin_internal_dim
6864     { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
6865     \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
6866     { \tex_kern:D -\l__coffin_internal_dim }
6867   }

```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```

6868   \_coffin_reset_structure:N \l__coffin_aligned_coffin
6869   \prop_clear:c
6870   { l__coffin_corners_ \_int_value:w \l__coffin_aligned_coffin _ prop }
6871   \_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.

```

6872   \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim

```

```

6873 {
6874     \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
6875     \__coffin_offset_poles:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
6876     \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
6877     \__coffin_offset_corners:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
6878 }
6879 {
6880     \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
6881     \__coffin_offset_poles:Nnn #4
6882         { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
6883     \__coffin_offset_corners:Nnn #1 { 0 pt } { 0 pt }
6884     \__coffin_offset_corners:Nnn #4
6885         { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
6886 }
6887 \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
6888 \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
6889 }
6890 \cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cncn }

```

(End definition for \coffin_join:NnnNnnnn and others. These functions are documented on page ??.)

\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. The function used when marking a position is hear also as it is similar but without the structure updates.

\coffin_attach:cnnNnnnn

\coffin_attach:Nnncnnnn

\coffin_attach:cncnnnn

\coffin_attach_mark:NnnNnnnn

```

6891 \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
6892 {
6893     \__coffin_align:NnnNnnnnN
6894     #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
6895     \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
6896     \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
6897     \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
6898     \__coffin_reset_structure:N \l__coffin_aligned_coffin
6899     \prop_set_eq:cc
6900     { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
6901     { l__coffin_corners_ \__int_value:w #1 _prop }
6902     \__coffin_update_poles:N \l__coffin_aligned_coffin
6903     \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
6904     \__coffin_offset_poles:Nnn #4
6905         { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
6906     \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
6907     \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
6908 }
6909 \cs_new_protected:Npn \coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
6910 {
6911     \__coffin_align:NnnNnnnnN
6912     #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
6913     \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
6914     \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
6915     \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }

```

```

6916     \box_set_eq:NN #1 \l__coffin_aligned_coffin
6917   }
6918   \cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }

```

(End definition for \coffin_attach:NnnNnnnn and others. These functions are documented on page ??.)

__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.

```

6919   \cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
6920   {
6921     \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
6922     \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
6923     \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
6924     \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
6925     \dim_set:Nn \l__coffin_offset_x_dim
6926     { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
6927     \dim_set:Nn \l__coffin_offset_y_dim
6928     { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
6929     \hbox_set:Nn \l__coffin_aligned_internal_coffin
6930     {
6931       \box_use:N #1
6932       \tex_kern:D -\box_wd:N #1
6933       \tex_kern:D \l__coffin_offset_x_dim
6934       \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
6935     }
6936     \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
6937   }

```

(End definition for __coffin_align:NnnNnnnnN. This function is documented on page ??.)

__coffin_offset_poles:Nnn 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.

```

6938   \cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
6939   {
6940     \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
6941     { \__coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
6942   }
6943   \cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
6944   {
6945     \dim_set:Nn \l__coffin_x_dim { #3 + #7 }

```



```

6946 \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
6947 \tl_if_in:nnTF {#2} { - }
6948 { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
6949 { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
6950 \exp_last_unbraced:NNo \__coffin_set_pole:Nnx \l__coffin_aligned_coffin
6951 { \l__coffin_internal_tl }
6952 {
6953   { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
6954   {#5} {#6}
6955 }
6956 }

```

(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`

```

6957 \cs_new_protected:Npn \__coffin_offset_corners:Nnn #1#2#3
6958 {
6959   \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
6960   { \__coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
6961 }
6962 \cs_new_protected:Npn \__coffin_offset_corner:Nnnnn #1#2#3#4#5#6
6963 {
6964   \prop_put:cnx
6965   { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
6966   { #1 - #2 }
6967   {
6968     { \dim_eval:n { #3 + #5 } }
6969     { \dim_eval:n { #4 + #6 } }
6970   }
6971 }

```

(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`

```

6972 \cs_new_protected:Npn \__coffin_update_vertical_poles:NNN #1#2#3
6973 {
6974   \__coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
6975   \__coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
6976   \exp_last_two_unbraced:Noo \__coffin_update_T:nnnnnnnnN
6977   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
6978   \__coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
6979   \__coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
6980   \exp_last_two_unbraced:Noo \__coffin_update_B:nnnnnnnnN
6981   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
6982 }
6983 \cs_new_protected:Npn \__coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
6984 {
6985   \dim_compare:nNnTF {#2} < {#6}

```

```

6986     {
6987         \__coffin_set_pole:Nnx #9 { T }
6988         { { 0 pt } {#6} { 1000 pt } { 0 pt } }
6989     }
6990     {
6991         \__coffin_set_pole:Nnx #9 { T }
6992         { { 0 pt } {#2} { 1000 pt } { 0 pt } }
6993     }
6994 }
6995 \cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
6996 {
6997     \dim_compare:nNnTF {#2} < {#6}
6998     {
6999         \__coffin_set_pole:Nnx #9 { B }
7000         { { 0 pt } {#2} { 1000 pt } { 0 pt } }
7001     }
7002     {
7003         \__coffin_set_pole:Nnx #9 { B }
7004         { { 0 pt } {#6} { 1000 pt } { 0 pt } }
7005     }
7006 }

```

(End definition for __coffin_update_vertical_poles:NNN. This function is documented on page ??.)

\coffin_typeset:Nnnnn
\coffin_typeset:cnnnn

Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

```

7007 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
7008 {
7009     \hbox_unpack:N \c_empty_box
7010     \__coffin_align:NnnNnnnnN \c_empty_coffin { H } { 1 }
7011     #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
7012     \box_use:N \l__coffin_aligned_coffin
7013 }
7014 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }

```

(End definition for \coffin_typeset:Nnnnn and \coffin_typeset:cnnnn. These functions are documented on page ??.)

15.7 Coffin diagnostics

\l__coffin_display_coffin

Used for printing coffins with data structures attached.

```

\l__coffin_display_coord_coffin 7015 \coffin_new:N \l__coffin_display_coffin
\l__coffin_display_pole_coffin 7016 \coffin_new:N \l__coffin_display_coord_coffin
7017 \coffin_new:N \l__coffin_display_pole_coffin

```

(End definition for \l__coffin_display_coffin. This function 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.

```

7018 \prop_new:N \l__coffin_display_handles_prop

```

```

7019 \prop_put:Nnn \l__coffin_display_handles_prop { tl }
7020 { { b } { r } { -1 } { 1 } }
7021 \prop_put:Nnn \l__coffin_display_handles_prop { thc }
7022 { { b } { hc } { 0 } { 1 } }
7023 \prop_put:Nnn \l__coffin_display_handles_prop { tr }
7024 { { b } { l } { 1 } { 1 } }
7025 \prop_put:Nnn \l__coffin_display_handles_prop { vcl }
7026 { { vc } { r } { -1 } { 0 } }
7027 \prop_put:Nnn \l__coffin_display_handles_prop { vhc }
7028 { { vc } { hc } { 0 } { 0 } }
7029 \prop_put:Nnn \l__coffin_display_handles_prop { vcr }
7030 { { vc } { l } { 1 } { 0 } }
7031 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
7032 { { t } { r } { -1 } { -1 } }
7033 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
7034 { { t } { hc } { 0 } { -1 } }
7035 \prop_put:Nnn \l__coffin_display_handles_prop { br }
7036 { { t } { l } { 1 } { -1 } }
7037 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }
7038 { { t } { r } { -1 } { -1 } }
7039 \prop_put:Nnn \l__coffin_display_handles_prop { Thc }
7040 { { t } { hc } { 0 } { -1 } }
7041 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
7042 { { t } { l } { 1 } { -1 } }
7043 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
7044 { { vc } { r } { -1 } { 1 } }
7045 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
7046 { { vc } { hc } { 0 } { 1 } }
7047 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
7048 { { vc } { l } { 1 } { 1 } }
7049 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
7050 { { b } { r } { -1 } { -1 } }
7051 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
7052 { { b } { hc } { 0 } { -1 } }
7053 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
7054 { { 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.

```

7055 \dim_new:N \l__coffin_display_offset_dim
7056 \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` As the intersections of poles have to be calculated to find which ones to print, there is
`\l__coffin_display_y_dim` a need to avoid repetition. This is done by saving the intersection into two dedicated
values.

```

7057 \dim_new:N \l__coffin_display_x_dim
7058 \dim_new:N \l__coffin_display_y_dim

```

(End definition for \l__coffin_display_x_dim. This function is documented on page ??.)

<code>\l__coffin_display_poles_prop</code>	<p>A property list for printing poles: various things need to be deleted from this to get a “nice” output.</p> <pre> 7059 \prop_new:N \l__coffin_display_poles_prop (End definition for \l__coffin_display_poles_prop. This variable is documented on page ??.) </pre>
<code>\l__coffin_display_font_tl</code>	<p>Stores the settings used to print coffin data: this keeps things flexible.</p> <pre> 7060 \tl_new:N \l__coffin_display_font_tl 7061 <*initex> 7062 \tl_set:Nn \l__coffin_display_font_tl { } % TODO 7063 </initex> 7064 <*package> 7065 \tl_set:Nn \l__coffin_display_font_tl { \sfamily \tiny } 7066 </package> (End definition for \l__coffin_display_font_tl. This variable is documented on page ??.) </pre>
<code>\coffin_mark_handle:Nnnn</code> <code>\coffin_mark_handle:cnnn</code> <code>__coffin_mark_handle_aux:nnnnNnn</code>	<p>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.</p> <pre> 7067 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4 7068 { 7069 \hcoffin_set:Nn \l__coffin_display_pole_coffin 7070 { 7071 <*initex> 7072 \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO 7073 </initex> 7074 <*package> 7075 \color {#4} 7076 \rule { 1 pt } { 1 pt } 7077 </package> 7078 } 7079 \coffin_attach_mark:NnnNnnnn #1 {#2} {#3} 7080 \l__coffin_display_pole_coffin { hc } { vc } { 0 pt } { 0 pt } 7081 \hcoffin_set:Nn \l__coffin_display_coord_coffin 7082 { 7083 <*initex> 7084 % TODO 7085 </initex> 7086 <*package> 7087 \color {#4} 7088 </package> 7089 \l__coffin_display_font_tl 7090 (\tl_to_str:n { #2 , #3 }) 7091 } 7092 \prop_get:NnN \l__coffin_display_handles_prop 7093 { #2 #3 } \l__coffin_internal_tl 7094 \quark_if_no_value:NTF \l__coffin_internal_tl 7095 { 7096 \prop_get:NnN \l__coffin_display_handles_prop </pre>

```

7097         { #3 #2 } \l__coffin_internal_tl
7098     \quark_if_no_value:NTF \l__coffin_internal_tl
7099     {
7100         \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
7101         \l__coffin_display_coord_coffin { 1 } { vc }
7102         { 1 pt } { 0 pt }
7103     }
7104     {
7105         \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
7106         \l__coffin_internal_tl #1 {#2} {#3}
7107     }
7108 }
7109 {
7110     \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
7111     \l__coffin_internal_tl #1 {#2} {#3}
7112 }
7113 }
7114 \cs_new_protected:Npn \__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
7115 {
7116     \coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
7117     \l__coffin_display_coord_coffin {#1} {#2}
7118     { #3 \l__coffin_display_offset_dim }
7119     { #4 \l__coffin_display_offset_dim }
7120 }
7121 \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 ??.)

\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.

```

7122 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
7123 {
7124     \hcoffin_set:Nn \l__coffin_display_pole_coffin
7125     {
7126     <*initex>
7127         \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
7128     </initex>
7129     <*package>
7130         \color {#2}
7131         \rule { 1 pt } { 1 pt }
7132     </package>
7133     }
7134     \prop_set_eq:Nc \l__coffin_display_poles_prop
7135     { \l__coffin_poles_ \__int_value:w #1 _prop }
7136     \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
7137     \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
7138     \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl

```

```

7139     { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
7140     \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
7141     \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
7142     { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
7143     \coffin_set_eq:NN \l__coffin_display_coffin #1
7144     \prop_map_inline:Nn \l__coffin_display_poles_prop
7145     {
7146         \prop_remove:Nn \l__coffin_display_poles_prop {##1}
7147         \__coffin_display_handles_aux:nnnnnn {##1} ##2 {##2}
7148     }
7149     \box_use:N \l__coffin_display_coffin
7150 }

```

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.

```

7151 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
7152 {
7153     \prop_map_inline:Nn \l__coffin_display_poles_prop
7154     {
7155         \bool_set_false:N \l__coffin_error_bool
7156         \__coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2
7157         \bool_if:NF \l__coffin_error_bool
7158         {
7159             \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
7160             \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
7161             \__coffin_display_attach:Nnnnn
7162             \l__coffin_display_pole_coffin { hc } { vc }
7163             { 0 pt } { 0 pt }
7164             \hcoffin_set:Nn \l__coffin_display_coord_coffin
7165             {
7166                 <*initex>
7167                     % TODO
7168                 </initex>
7169                 <*package>
7170                     \color {#6}
7171                 </package>
7172                 \l__coffin_display_font_tl
7173                 ( \tl_to_str:n { #1 , ##1 } )
7174             }
7175             \prop_get:NnN \l__coffin_display_handles_prop
7176             { #1 ##1 } \l__coffin_internal_tl
7177             \quark_if_no_value:NTF \l__coffin_internal_tl
7178             {
7179                 \prop_get:NnN \l__coffin_display_handles_prop
7180                 { ##1 #1 } \l__coffin_internal_tl
7181                 \quark_if_no_value:NTF \l__coffin_internal_tl
7182                 {
7183                     \__coffin_display_attach:Nnnnn
7184                     \l__coffin_display_coord_coffin { l } { vc }

```

```

7185         { 1 pt } { 0 pt }
7186     }
7187     {
7188         \exp_last_unbraced:No
7189         \__coffin_display_handles_aux:nnnn
7190         \l__coffin_internal_tl
7191     }
7192 }
7193 {
7194     \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
7195     \l__coffin_internal_tl
7196 }
7197 }
7198 }
7199 }
7200 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
7201 {
7202     \__coffin_display_attach:Nnnnn
7203     \l__coffin_display_coord_coffin {#1} {#2}
7204     { #3 \l__coffin_display_offset_dim }
7205     { #4 \l__coffin_display_offset_dim }
7206 }
7207 \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.

```

7208 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
7209 {
7210     \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
7211     \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
7212     \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
7213     \dim_set:Nn \l__coffin_offset_x_dim
7214     { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
7215     \dim_set:Nn \l__coffin_offset_y_dim
7216     { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
7217     \hbox_set:Nn \l__coffin_aligned_coffin
7218     {
7219         \box_use:N \l__coffin_display_coffin
7220         \tex_kern:D -\box_wd:N \l__coffin_display_coffin
7221         \tex_kern:D \l__coffin_offset_x_dim
7222         \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
7223     }
7224     \box_set_ht:Nn \l__coffin_aligned_coffin
7225     { \box_ht:N \l__coffin_display_coffin }
7226     \box_set_dp:Nn \l__coffin_aligned_coffin
7227     { \box_dp:N \l__coffin_display_coffin }
7228     \box_set_wd:Nn \l__coffin_aligned_coffin
7229     { \box_wd:N \l__coffin_display_coffin }
7230     \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin

```

```
7231 }
```

(End definition for `\coffin_display_handles:Nn` and `\coffin_display_handles:cn`. These functions are documented on page ??.)

`\coffin_show_structure:N`
`\coffin_show_structure:c`

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.

```
7232 \cs_new_protected:Npn \coffin_show_structure:N #1
7233 {
7234   \__coffin_if_exist:NT #1
7235   {
7236     \__msg_show_variable:Nnn #1 { coffins }
7237     {
7238       \prop_map_function:cn
7239       { l__coffin_poles_ \__int_value:w #1 _prop }
7240       \__msg_show_item_unbraced:nn
7241     }
7242   }
7243 }
7244 \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 ??.)

15.8 Messages

```
7245 \__msg_kernel_new:nnnn { kernel } { no-pole-intersection }
7246 { No~intersection~between~coffin~poles. }
7247 {
7248   \c_msg_coding_error_text_tl
7249   LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
7250   but~they~do~not~have~a~unique~meeting~point:~
7251   the~value~(0~pt,~0~pt)~will~be~used.
7252 }
7253 \__msg_kernel_new:nnnn { kernel } { unknown-coffin }
7254 { Unknown~coffin~'#1'. }
7255 { The~coffin~'#1'~was~never~defined. }
7256 \__msg_kernel_new:nnnn { kernel } { unknown-coffin-pole }
7257 { Pole~'#1'~unknown~for~coffin~'#2'. }
7258 {
7259   \c_msg_coding_error_text_tl
7260   LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
7261   but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
7262 }
7263 \__msg_kernel_new:nnn { kernel } { show-coffins }
7264 {
7265   Size-of~coffin~\token_to_str:N #1 : \\
7266   > ~ ht==~\dim_use:N \box_ht:N #1 \\
7267   > ~ dp==~\dim_use:N \box_dp:N #1 \\
7268   > ~ wd==~\dim_use:N \box_wd:N #1 \\
7269   Poles~of~coffin~\token_to_str:N #1 :
```



```

7270 }
7271 </initex | package>

```

16 l3color Implementation

```

7272 <*initex | package>
7273 <*package>
7274 \ProvidesExplPackage
7275   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
7276   \_expl_package_check:
7277 </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.

```

7278 \cs_new_eq:NN \color_group_begin: \group_begin:
7279 \cs_new_protected_nopar:Npn \color_group_end:
7280   {
7281     \tex_par:D
7282     \group_end:
7283   }

```

(End definition for `\color_group_begin:` and `\color_group_end:`. These functions are documented on page 140.)

\color_ensure_current: A driver-independent wrapper for setting the foreground color to the current color “now”.

```

7284 <*initex>
7285 \cs_new_protected_nopar:Npn \color_ensure_current:
7286   { \_driver_color_ensure_current: }
7287 </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.

```

7288 <*package>
7289 \cs_new_protected_nopar:Npn \color_ensure_current: { }
7290 \AtBeginDocument
7291   {
7292     \cs_if_exist:NTF \_driver_color_ensure_current:
7293       {
7294         \cs_set_protected_nopar:Npn \color_ensure_current:
7295           { \_driver_color_ensure_current: }
7296       }
7297       {
7298         \cs_if_exist:NT \set@color
7299         { \cs_set_protected_nopar:Npn \color_ensure_current: { \set@color } }
7300       }
7301   }
7302 </package>

```

(End definition for `\color_ensure_current:`. This function is documented on page 140.)

```

7303 </initex | package>

```

17 l3msg implementation

```

7304 <*initex | package>
7305 <@@=msg>
7306 <*package>
7307 \ProvidesExplPackage
7308   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
7309   \_expl_package_check:
7310 </package>

```

`\l__msg_internal_tl` A general scratch for the module.

```

7311 \tl_new:N \l__msg_internal_tl
(End definition for \l__msg_internal_tl. This variable is documented on page ??.)

```

17.1 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`

```

7312 \tl_const:Nn \c__msg_text_prefix_tl { msg~text~>~ }
7313 \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`

```

7314 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
7315 {
7316   \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
7317   { \prg_return_true: } { \prg_return_false: }
7318 }

```

(End definition for `\msg_if_exist:nn`. These functions are documented on page 142.)

`__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.

```

7319 \cs_new_protected:Npn \__chk_if_free_msg:nn #1#2
7320 {
7321   \msg_if_exist:nnT {#1} {#2}
7322   {
7323     \_msg_kernel_error:nnxx { kernel } { message-already-defined }
7324     {#1} {#2}
7325   }
7326 }
7327 <*package>
7328 \tex_ifodd:D \l@expl@log@functions@bool

```

```

7329 \cs_gset_protected:Npn \__chk_if_free_msg:nn #1#2
7330 {
7331   \msg_if_exist:nnT {#1} {#2}
7332   {
7333     \__msg_kernel_error:nnxx { kernel } { message-already-defined }
7334     {#1} {#2}
7335   }
7336   \iow_log:x { Defining-message~ #1 / #2 ~\msg_line_context: }
7337 }
7338 \fi:
7339 </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
7340 \cs_new_protected:Npn \msg_new:nnnn #1#2
7341 {
7342   \__chk_if_free_msg:nn {#1} {#2}
7343   \msg_gset:nnnn {#1} {#2}
7344 }
7345 \cs_new_protected:Npn \msg_new:nnn #1#2#3
7346 { \msg_new:nnnn {#1} {#2} {#3} { } }
7347 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
7348 {
7349   \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
7350   ##1##2##3##4 {#3}
7351   \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
7352   ##1##2##3##4 {#4}
7353 }
7354 \cs_new_protected:Npn \msg_set:nnn #1#2#3
7355 { \msg_set:nnnn {#1} {#2} {#3} { } }
7356 \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
7357 {
7358   \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
7359   ##1##2##3##4 {#3}
7360   \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
7361   ##1##2##3##4 {#4}
7362 }
7363 \cs_new_protected:Npn \msg_gset:nnn #1#2#3
7364 { \msg_gset:nnnn {#1} {#2} {#3} { } }
(End definition for \msg_new:nnnn and \msg_new:nnn. These functions are documented on page ??.)

```

17.2 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
7365 \tl_const:Nn \c_msg_coding_error_text_tl
7366 {
7367   This-is-a-coding-error.
7368   \\ \\

```

```

7369 }
7370 \tl_const:Nn \c_msg_continue_text_tl
7371 { Type~<return>~to~continue }
7372 \tl_const:Nn \c_msg_critical_text_tl
7373 { Reading~the~current~file~will~stop }
7374 \tl_const:Nn \c_msg_fatal_text_tl
7375 { This~is~a~fatal~error:~LaTeX~will~abort }
7376 \tl_const:Nn \c_msg_help_text_tl
7377 { For~immediate~help~type~H~<return> }
7378 \tl_const:Nn \c_msg_no_info_text_tl
7379 {
7380   LaTeX~does~not~know~anything~more~about~this~error,~sorry.
7381   \c_msg_return_text_tl
7382 }
7383 \tl_const:Nn \c_msg_on_line_text_tl { on-line }
7384 \tl_const:Nn \c_msg_return_text_tl
7385 {
7386   \\ \\
7387   Try~typing~<return>~to~proceed.
7388   \\
7389   If~that~doesn't~work,~type~X~<return>~to~quit.
7390 }
7391 \tl_const:Nn \c_msg_trouble_text_tl
7392 {
7393   \\ \\
7394   More~errors~will~almost~certainly~follow: \\
7395   the~LaTeX~run~should~be~aborted.
7396 }

```

(End definition for `\c_msg_coding_error_text_tl` and others. These variables are documented on page ??.)

`\msg_line_number:` For writing the line number nicely. `\msg_line_context:` was set up earlier, so this is not new.

```

7397 \cs_new_nopar:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
7398 \cs_gset_nopar:Npn \msg_line_context:
7399 {
7400   \c_msg_on_line_text_tl
7401   \c_space_tl
7402   \msg_line_number:
7403 }

```

(End definition for `\msg_line_number:` and `\msg_line_context:`. These functions are documented on page 142.)

17.3 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`.

```

7404 \cs_new_protected:Npn \msg_interrupt:nnn #1#2#3
7405 {
7406   \tl_if_empty:nTF {#3}
7407   {
7408     \__msg_interrupt_wrap:nn { \ \c_msg_no_info_text_tl }
7409     {#1 \\\ \c_msg_continue_text_tl }
7410   }
7411   {
7412     \__msg_interrupt_wrap:nn { \ \c_msg_help_text_tl }
7413     {#1 \\\ \c_msg_help_text_tl }
7414   }
7415 }

```

(End definition for `\msg_interrupt:nnn`. This function is documented on page 146.)

```

\__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.

```

7416 \cs_new_protected:Npn \__msg_interrupt_wrap:nn #1#2
7417 {
7418   \iow_wrap:nnnN {#1} { | ~ } { } \__msg_interrupt_more_text:n
7419   \iow_wrap:nnnN {#2} { ! ~ } { } \__msg_interrupt_text:n
7420 }
7421 \cs_new_protected:Npn \__msg_interrupt_more_text:n #1
7422 {
7423   \exp_args:Nx \tex_errhelp:D
7424   {
7425     |,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
7426     #1 \iow_newline:
7427     |.....
7428   }
7429 }

```

(End definition for `__msg_interrupt_wrap:nn`. This function is documented on page 146.)

```

\__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.

```

7430 \group_begin:
7431 \char_set_lccode:nn {'\} {'\ }
7432 \char_set_lccode:nn {'\} {'\ }
7433 \char_set_lccode:nn {'&} {'!\}

```

```

7434 \char_set_catcode_active:N \&
7435 \tl_to_lowercase:n
7436 {
7437   \group_end:
7438   \cs_new_protected:Npn \__msg_interrupt_text:n #1
7439   {
7440     \iow_term:x
7441     {
7442       \iow_newline:
7443       !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
7444       \iow_newline:
7445       !
7446     }
7447     \group_begin:
7448     \cs_set_protected_nopar:Npn &
7449     {
7450       \tex_errmessage:D
7451       {
7452         #1
7453         \use_none:n
7454         { ..... }
7455       }
7456     }
7457     \exp_after:wN
7458     \group_end:
7459     &
7460   }
7461 }

```

(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.

```

7462 \cs_new_protected:Npn \msg_log:n #1
7463 {
7464   \iow_log:n { ..... }
7465   \iow_wrap:nnnN { . ~ #1 } { . ~ } { } \iow_log:n
7466   \iow_log:n { ..... }
7467 }
7468 \cs_new_protected:Npn \msg_term:n #1
7469 {
7470   \iow_term:n { ***** }
7471   \iow_wrap:nnnN { * ~ #1 } { * ~ } { } \iow_term:n
7472   \iow_term:n { ***** }
7473 }

```

(End definition for \msg_log:n. This function is documented on page 147.)

17.4 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.

```

7474 <*initex>
7475 \int_gset_eq:NN \tex_errorcontextlines:D \c_minus_one
7476 </initex>

```

\msg_fatal_text:n A function for issuing messages: both the text and order could in principle vary.

\msg_critical_text:n

\msg_error_text:n

\msg_warning_text:n

\msg_info_text:n

```

7477 \cs_new:Npn \msg_fatal_text:n #1 { Fatal~#1~error }
7478 \cs_new:Npn \msg_critical_text:n #1 { Critical~#1~error }
7479 \cs_new:Npn \msg_error_text:n #1 { #1~error }
7480 \cs_new:Npn \msg_warning_text:n #1 { #1~warning }
7481 \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 143.)

\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”.

```

7482 \cs_new:Npn \msg_see_documentation_text:n #1
7483 {
7484   \\\ See-the~
7485   \str_if_eq:nnTF {#1} { LaTeX } { LaTeX3 } {#1} ~
7486   documentation~for~further~information.
7487 }

```

(End definition for \msg_see_documentation_text:n. This function is documented on page 143.)

__msg_class_new:nn

```

7488 \group_begin:
7489 \cs_set_protected:Npn \__msg_class_new:nn #1#2
7490 {
7491   \prop_new:c { l__msg_redirect_ #1 _prop }
7492   \cs_new_protected:cpn { __msg_ #1 _code:nnnnnn } ##1##2##3##4##5##6 {#2}
7493   \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
7494   {
7495     \use:x
7496     {
7497       \exp_not:n { \__msg_use:nnnnnnn {#1} {##1} {##2} }
7498       { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
7499       { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
7500     }
7501   }
7502   \cs_new_protected:cpx { msg_ #1 :nnnnn } ##1##2##3##4##5
7503   { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
7504   \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4
7505   { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
7506   \cs_new_protected:cpx { msg_ #1 :nnn } ##1##2##3
7507   { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
7508   \cs_new_protected:cpx { msg_ #1 :nn } ##1##2

```

```

7509      { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
7510 \cs_new_protected:cpx { msg_ #1 :nnxxxx } ##1##2##3##4##5##6
7511 {
7512     \use:x
7513     {
7514         \exp_not:N \exp_not:n
7515         { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
7516         {##3} {##4} {##5} {##6}
7517     }
7518 }
7519 \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
7520 { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
7521 \cs_new_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
7522 { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
7523 \cs_new_protected:cpx { msg_ #1 :nnx } ##1##2##3
7524 { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
7525 }

```

(End definition for `_msg_class_new:nn`. This function is documented on page ??.)

`\msg_fatal:nnnnnn` For fatal errors, after the error message T_EX bails out.

```

\msg_fatal:nnnnnn 7526 \_msg_class_new:nn { fatal }
\msg_fatal:nnnn 7527 {
\msg_fatal:nnn 7528     \msg_interrupt:nnn
\msg_fatal:nn 7529     { \msg_fatal_text:n {#1} : ~ "#2" }
\msg_fatal:nnxxxx 7530     {
\msg_fatal:nnxxx 7531         \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
\msg_fatal:nnxx 7532         \msg_see_documentation_text:n {#1}
\msg_fatal:nnx 7533     }
7534     { \c_msg_fatal_text_tl }
7535     \tex_end:D
7536 }

```

(End definition for `\msg_fatal:nnnnnn` and others. These functions are documented on page ??.)

`\msg_critical:nnnnnn` Not quite so bad: just end the current file.

```

\msg_critical:nnnnnn 7537 \_msg_class_new:nn { critical }
\msg_critical:nnnn 7538 {
\msg_critical:nnn 7539     \msg_interrupt:nnn
\msg_critical:nn 7540     { \msg_critical_text:n {#1} : ~ "#2" }
\msg_critical:nnxxxx 7541     {
\msg_critical:nnxxx 7542         \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
\msg_critical:nnxx 7543         \msg_see_documentation_text:n {#1}
\msg_critical:nnx 7544     }
7545     { \c_msg_critical_text_tl }
7546     \tex_endinput:D
7547 }

```

(End definition for `\msg_critical:nnnnnn` and others. These functions are documented on page ??.)

`\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

```



```

7548 \_msg_class_new:nn { error }
7549 {
7550   \_msg_error:cnnnnn
7551   { \c__msg_more_text_prefix_tl #1 / #2 }
7552   {#3} {#4} {#5} {#6}
7553   {
7554     \msg_interrupt:nnn
7555     { \msg_error_text:n {#1} : ~ "#2" }
7556     {
7557       \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
7558       \msg_see_documentation_text:n {#1}
7559     }
7560   }
7561 }
7562 \cs_new_protected:Npn \_msg_error:cnnnnn #1#2#3#4#5#6
7563 {
7564   \cs_if_eq:cNTF {#1} \_msg_no_more_text:nnnn
7565   { #6 { } }
7566   { #6 { \use:c {#1} {#2} {#3} {#4} {#5} } }
7567 }
7568 \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 ??.)

\msg_warning:nnnnnn

Warnings are printed to the terminal.

```

\msg_warning:nnnnnn 7569 \_msg_class_new:nn { warning }
\msg_warning:nnnn 7570 {
\msg_warning:nnn 7571   \msg_term:n
\msg_warning:nn 7572   {
\msg_warning:nnxxxx 7573     \msg_warning_text:n {#1} : ~ "#2" \\ \\
\msg_warning:nnxxx 7574     \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
\msg_warning:nnxx 7575   }
\msg_warning:nnx 7576 }

```

(End definition for \msg_warning:nnnnnn and others. These functions are documented on page ??.)

\msg_info:nnnnnn

Information only goes into the log.

```

\msg_info:nnnnnn 7577 \_msg_class_new:nn { info }
\msg_info:nnnn 7578 {
\msg_info:nnn 7579   \msg_log:n
\msg_info:nn 7580   {
\msg_info:nnxxxx 7581     \msg_info_text:n {#1} : ~ "#2" \\ \\
\msg_info:nnxxx 7582     \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
\msg_info:nnxx 7583   }
\msg_info:nnx 7584 }

```

(End definition for \msg_info:nnnnnn and others. These functions are documented on page ??.)

\msg_log:nnnnnn

“Log” data is very similar to information, but with no extras added.

```

\msg_log:nnnnnn 7585 \_msg_class_new:nn { log }
\msg_log:nnnn 7586 {
\msg_log:nnn 7587   \iow_wrap:nnnN
\msg_log:nn
\msg_log:nnxxxx
\msg_log:nnxxx
\msg_log:nnxx
\msg_log:nnx

```

```

7588         { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
7589         { } { } \iow_log:n
7590     }

```

(End definition for \msg_log:nnnnnn and others. These functions are documented on page ??.)

\msg_none:nnnnnn

The none message type is needed so that input can be gobbled.

\msg_none:nnnnn

```

7591     \__msg_class_new:nn { none } { }

```

\msg_none:nnnn

(End definition for \msg_none:nnnnnn and others. These functions are documented on page ??.)

\msg_none:nnn

End the group to eliminate __msg_class_new:nn.

\msg_none:nn

```

7592 \group_end:

```

\msg_none:nnxxxx

__msg_class_chk_exist:nnxxx

\msg_none:nnxx

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.

\msg_none:nnx

```

7593 \cs_new:Npn \__msg_class_chk_exist:nT #1
7594 {
7595     \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
7596     { \__msg_kernel_error:nnx { kernel } { message-class-unknown } {#1} }
7597 }

```

(End definition for __msg_class_chk_exist:nT.)

\l__msg_class_tl

Support variables needed for the redirection system.

\l__msg_current_class_tl

```

7598 \tl_new:N \l__msg_class_tl
7599 \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

```

7600 \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 {}.

```

7601 \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.

```

7602 \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.

__msg_use_redirect_name:n

__msg_use_hierarchy:nwN

__msg_use_redirect_module:n

__msg_use_code:

```

7603 \cs_new_protected:Npn \__msg_use:nnnnnnn #1#2#3#4#5#6#7
7604 {

```

```

7605 \msg_if_exist:nnTF {#2} {#3}
7606 {
7607     \__msg_class_chk_exist:nT {#1}
7608     {
7609         \tl_set:Nn \l__msg_current_class_tl {#1}
7610         \cs_set_protected_nopar:Npx \__msg_use_code:
7611         {
7612             \exp_not:n
7613             {
7614                 \use:c { __msg_ \l__msg_class_tl _code:nnnnnn }
7615                 {#2} {#3} {#4} {#5} {#6} {#7}
7616             }
7617         }
7618         \__msg_use_redirect_name:n { #2 / #3 }
7619     }
7620 }
7621 { \__msg_kernel_error:nnxx { kernel } { message-unknown } {#2} {#3} }
7622 }
7623 \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.

```

7624 \cs_new_protected:Npn \__msg_use_redirect_name:n #1
7625 {
7626     \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
7627     { \__msg_use_code: }
7628     {
7629         \seq_clear:N \l__msg_hierarchy_seq
7630         \__msg_use_hierarchy:nwN { }
7631         #1 \q_mark \__msg_use_hierarchy:nwN
7632         / \q_mark \use_none_delimit_by_q_stop:w
7633         \q_stop
7634         \__msg_use_redirect_module:n { }
7635     }
7636 }
7637 \cs_new_protected:Npn \__msg_use_hierarchy:nwN #1#2 / #3 \q_mark #4
7638 {
7639     \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
7640     #4 { #1 / #2 } #3 \q_mark #4
7641 }

```

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**.

```

7642 \cs_new_protected:Npn \__msg_use_redirect_module:n #1
7643 {
7644   \seq_map_inline:Nn \l__msg_hierarchy_seq
7645   {
7646     \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
7647     {##1} \l__msg_class_tl
7648     {
7649       \seq_map_break:n
7650       {
7651         \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
7652         { \__msg_use_code: }
7653         {
7654           \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
7655           \__msg_use_redirect_module:n {##1}
7656         }
7657       }
7658     }
7659     {
7660       \str_if_eq:nnT {##1} {#1}
7661       {
7662         \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
7663         \seq_map_break:n { \__msg_use_code: }
7664       }
7665     }
7666   }
7667 }

```

(End definition for `__msg_use:nnnnnnn`. This function is documented on page ??.)

\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.

```

7668 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
7669 {
7670   \tl_if_empty:nTF {#3}
7671   { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
7672   {
7673     \__msg_class_chk_exist:nT {#3}
7674     { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
7675   }
7676 }

```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page 146.)

\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:nnn
\__msg_redirect_loop_chk:nnn
\__msg_redirect_loop_list:n

```

```

7677 \cs_new_protected_nopar:Npn \msg_redirect_class:nn
7678 { \__msg_redirect:nnn { } }
7679 \cs_new_protected:Npn \msg_redirect_module:nnn #1
7680 { \__msg_redirect:nnn { / #1 } }
7681 \cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
7682 {
7683   \__msg_class_chk_exist:nT {#2}
7684   {
7685     \tl_if_empty:nTF {#3}
7686     { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
7687     {
7688       \__msg_class_chk_exist:nT {#3}
7689       {
7690         \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
7691         \tl_set:Nn \l__msg_current_class_tl {#2}
7692         \seq_clear:N \l__msg_class_loop_seq
7693         \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
7694       }
7695     }
7696   }
7697 }

```

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.

```

7698 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
7699 {
7700   \seq_put_right:Nn \l__msg_class_loop_seq {#1}
7701   \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
7702   {
7703     \str_if_eq:x:nnF { \l__msg_class_tl } {#1}
7704     {
7705       \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
7706       {
7707         \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
7708         \__msg_kernel_warning:nnxxx { kernel } { message-redirect-loop }
7709         { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
7710         { \seq_item:Nn \l__msg_class_loop_seq { \c_two } }
7711         {#3}
7712       }
7713       \seq_map_function:NN \l__msg_class_loop_seq

```

```

7714         \_msg_redirect_loop_list:n
7715         { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
7716     }
7717 }
7718 { \_msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
7719 }
7720 }
7721 }

```

```

7722 \cs_generate_variant:Nn \_msg_redirect_loop_chk:nnn { o }
7723 \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 146.)

17.5 Kernel-specific functions

`_msg_kernel_new:nnnn` 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.

```

\_msg_kernel_new:nnn
\_msg_kernel_set:nnnn
\_msg_kernel_set:nnn
7724 \cs_new_protected:Npn \_msg_kernel_new:nnnn #1#2
7725 { \msg_new:nnnn { LaTeX } { #1 / #2 } }
7726 \cs_new_protected:Npn \_msg_kernel_new:nnn #1#2
7727 { \msg_new:nnn { LaTeX } { #1 / #2 } }
7728 \cs_new_protected:Npn \_msg_kernel_set:nnnn #1#2
7729 { \msg_set:nnnn { LaTeX } { #1 / #2 } }
7730 \cs_new_protected:Npn \_msg_kernel_set:nnn #1#2
7731 { \msg_set:nnn { LaTeX } { #1 / #2 } }

```

(End definition for `_msg_kernel_new:nnnn` and `_msg_kernel_new:nnn`. These functions are documented on page ??.)

`_msg_kernel_class_new:nN` All the functions for kernel messages come in variants ranging from 0 to 4 arguments. Those with less than 4 arguments are defined in terms of the 4-argument variant, in a way very similar to `_msg_class_new:nn`. This auxiliary is destroyed at the end of the group.

`_msg_kernel_class_new_aux:nN`

```

7732 \group_begin:
7733 \cs_set_protected:Npn \_msg_kernel_class_new:nN #1
7734 { \_msg_kernel_class_new_aux:nN { kernel_ #1 } }
7735 \cs_set_protected:Npn \_msg_kernel_class_new_aux:nN #1#2
7736 {
7737     \cs_new_protected:cpn { __msg_ #1 :nnnnnn } ##1##2##3##4##5##6
7738     {
7739         \use:x
7740         {
7741             \exp_not:n { #2 { LaTeX } { ##1 / ##2 } }
7742             { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
7743             { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
7744         }
7745     }
7746     \cs_new_protected:cpn { __msg_ #1 :nnnnnn } ##1##2##3##4##5
7747     { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }

```

```

7748 \cs_new_protected:cpx { __msg_ #1 :nnnn } ##1##2##3##4
7749 { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
7750 \cs_new_protected:cpx { __msg_ #1 :nnn } ##1##2##3
7751 { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
7752 \cs_new_protected:cpx { __msg_ #1 :nn } ##1##2
7753 { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
7754 \cs_new_protected:cpx { __msg_ #1 :nnxxxx } ##1##2##3##4##5##6
7755 {
7756   \use:x
7757   {
7758     \exp_not:N \exp_not:n
7759     { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} }
7760     {##3} {##4} {##5} {##6}
7761   }
7762 }
7763 \cs_new_protected:cpx { __msg_ #1 :nnxxx } ##1##2##3##4##5
7764 { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
7765 \cs_new_protected:cpx { __msg_ #1 :nnxx } ##1##2##3##4
7766 { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
7767 \cs_new_protected:cpx { __msg_ #1 :nnx } ##1##2##3
7768 { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
7769 }

```

(End definition for `__msg_kernel_class_new:nN`. This function is documented on page ??.)

```

\__msg_kernel_fatal:nnnnnn
\__msg_kernel_fatal:nnnnn
\__msg_kernel_fatal:nnnn
\__msg_kernel_fatal:nnn
\__msg_kernel_fatal:nn
\__msg_kernel_fatal:nnxxxx
\__msg_kernel_fatal:nnxxx
\__msg_kernel_fatal:nnxx
\__msg_kernel_fatal:nnx
\__msg_kernel_fatal:nn
\__msg_kernel_error:nnnnnn

```

Neither fatal kernel errors nor kernel errors can be redirected. We directly use the code for (non-kernel) fatal errors and errors, adding the “`LATEX`” module name. Three functions are already defined by `l3basics`; we need to undefine them to avoid errors.

```

7770 \__msg_kernel_class_new:nN { fatal } \__msg_fatal_code:nnnnnn
7771 \cs_undefine:N \__msg_kernel_error:nnxx
7772 \cs_undefine:N \__msg_kernel_error:nnx
7773 \cs_undefine:N \__msg_kernel_error:nn
7774 \__msg_kernel_class_new:nN { error } \__msg_error_code:nnnnnn

```

(End definition for `__msg_kernel_fatal:nnnnnn` and others. These functions are documented on page ??.)

```

\__msg_kernel_warning:nnnnnn
\__msg_kernel_warning:nnnnn
\__msg_kernel_warning:nnnn
\__msg_kernel_warning:nnn
\__msg_kernel_warning:nn
\__msg_kernel_warning:nnxxxx
\__msg_kernel_warning:nnxxx
\__msg_kernel_warning:nnxx
\__msg_kernel_warning:nnx
\__msg_kernel_warning:nn
\__msg_kernel_warning:nnx
\__msg_kernel_info:nnnnnn
\__msg_kernel_info:nnnnn
\__msg_kernel_info:nnnn
\__msg_kernel_info:nnn
\__msg_kernel_info:nn
\__msg_kernel_info:nnxxxx
\__msg_kernel_info:nnxxx
\__msg_kernel_info:nnxx
\__msg_kernel_info:nnx

```

Kernel messages which can be redirected simply use the machinery for normal messages, with the module name “`LATEX`”.

```

7775 \__msg_kernel_class_new:nN { warning } \msg_warning:nnxxxx
7776 \__msg_kernel_class_new:nN { info } \msg_info:nnxxxx

```

(End definition for `__msg_kernel_warning:nnnnnn` and others. These functions are documented on page ??.)

End the group to eliminate `__msg_kernel_class_new:nN`.

```

7777 \group_end:

```

Error messages needed to actually implement the message system itself.

```

7778 \__msg_kernel_new:nnnn { kernel } { message-already-defined }
7779 { Message~'#2'~for-module~'#1'~already-defined. }
7780 {
7781   \c_msg_coding_error_text_tl

```

```

7782 LaTeX~was~asked~to~define~a~new~message~called~'#2'\
7783 by~the~module~'#1':~this~message~already~exists.
7784 \c_msg_return_text_tl
7785 }
7786 \_msg_kernel_new:nnnn { kernel } { message-unknown }
7787 { Unknown~message~'#2'~for~module~'#1'. }
7788 {
7789 \c_msg_coding_error_text_tl
7790 LaTeX~was~asked~to~display~a~message~called~'#2'\
7791 by~the~module~'#1':~this~message~does~not~exist.
7792 \c_msg_return_text_tl
7793 }
7794 \_msg_kernel_new:nnnn { kernel } { message-class-unknown }
7795 { Unknown~message~class~'#1'. }
7796 {
7797 LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\
7798 this~was~never~defined.
7799 \c_msg_return_text_tl
7800 }
7801 \_msg_kernel_new:nnnn { kernel } { message-redirect-loop }
7802 {
7803 Message~redirection~loop~caused~by~ {#1} ~>~ {#2}
7804 \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
7805 }
7806 {
7807 Adding~the~message~redirection~ {#1} ~>~ {#2}
7808 \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
7809 created~an~infinite~loop\\
7810 \iow_indent:n { #4 \\ }
7811 }

```

Messages for earlier kernel modules.

```

7812 \_msg_kernel_new:nnnn { kernel } { bad-number-of-arguments }
7813 { Function~'#1'~cannot~be~defined~with~#2~arguments. }
7814 {
7815 \c_msg_coding_error_text_tl
7816 LaTeX~has~been~asked~to~define~a~function~'#1'~with~
7817 #2~arguments.~
7818 TeX~allows~between~0~and~9~arguments~for~a~single~function.
7819 }
7820 \_msg_kernel_new:nnnn { kernel } { command-already-defined }
7821 { Control~sequence~#1~already~defined. }
7822 {
7823 \c_msg_coding_error_text_tl
7824 LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
7825 but~this~name~has~already~been~used~elsewhere. \\ \\
7826 The~current~meaning~is:\\
7827 \ \ #2
7828 }
7829 \_msg_kernel_new:nnnn { kernel } { command-not-defined }

```



```

7830 { Control~sequence~#1~undefined. }
7831 {
7832   \c_msg_coding_error_text_tl
7833   LaTeX~has~been~asked~to~use~a~command~#1,~but~this~has~not~
7834   been~defined~yet.
7835 }
7836 \_msg_kernel_new:nnnn { kernel } { empty-search-pattern }
7837 { Empty~search~pattern. }
7838 {
7839   \c_msg_coding_error_text_tl
7840   LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'#1':~that~
7841   would~lead~to~an~infinite~loop!
7842 }
7843 \_msg_kernel_new:nnnn { kernel } { out-of-registers }
7844 { No~room~for~a~new~#1. }
7845 {
7846   TeX~only~supports~\int\_use:N \c\_max\_register\_int \
7847   of~each~type.~All~the~#1~registers~have~been~used.~
7848   This~run~will~be~aborted~now.
7849 }
7850 \_msg_kernel_new:nnnn { kernel } { missing-colon }
7851 { Function~'#1'~contains~no~':'~. }
7852 {
7853   \c_msg_coding_error_text_tl
7854   Code~level~functions~must~contain~':'~to~separate~the~
7855   argument~specification~from~the~function~name.~This~is~
7856   needed~when~defining~conditionals~or~variants,~or~when~building~a~
7857   parameter~text~from~the~number~of~arguments~of~the~function.
7858 }
7859 \_msg_kernel_new:nnnn { kernel } { protected-predicate }
7860 { Predicate~'#1'~must~be~expandable. }
7861 {
7862   \c_msg_coding_error_text_tl
7863   LaTeX~has~been~asked~to~define~'#1'~as~a~protected~predicate.~
7864   Only~expandable~tests~can~have~a~predicate~version.
7865 }
7866 \_msg_kernel_new:nnnn { kernel } { conditional-form-unknown }
7867 { Conditional~form~'#1'~for~function~'#2'~unknown. }
7868 {
7869   \c_msg_coding_error_text_tl
7870   LaTeX~has~been~asked~to~define~the~conditional~form~'#1'~of~
7871   the~function~'#2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.
7872 }
7873 \_msg_kernel_new:nnnn { kernel } { scanmark-already-defined }
7874 { Scan~mark~#1~already~defined. }
7875 {
7876   \c_msg_coding_error_text_tl
7877   LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~
7878   but~this~name~has~already~been~used~for~a~scan~mark.
7879 }

```

```

7880 \_msg_kernel_new:nnnn { kernel } { variable-not-defined }
7881 { Variable~#1~undefined. }
7882 {
7883   \c_msg_coding_error_text_tl
7884   LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
7885   been~defined~yet.
7886 }
7887 \_msg_kernel_new:nnnn { kernel } { variant-too-long }
7888 { Variant~form~'~#1'~longer~than~base~signature~of~'~#2'. }
7889 {
7890   \c_msg_coding_error_text_tl
7891   LaTeX~has~been~asked~to~create~a~variant~of~the~function~'~#2'~
7892   with~a~signature~starting~with~'~#1',~but~that~is~longer~than~
7893   the~signature~(part~after~the~colon)~of~'~#2'.
7894 }
7895 \_msg_kernel_new:nnnn { kernel } { invalid-variant }
7896 { Variant~form~'~#1'~invalid~for~base~form~'~#2'. }
7897 {
7898   \c_msg_coding_error_text_tl
7899   LaTeX~has~been~asked~to~create~a~variant~of~the~function~'~#2'~
7900   with~a~signature~starting~with~'~#1',~but~cannot~change~an~argument~
7901   from~type~'~#3'~to~type~'~#4'.
7902 }

```

Some errors only appear in expandable settings, hence don't need a “more-text” argument.

```

7903 \_msg_kernel_new:nnn { kernel } { bad-variable }
7904 { Erroneous~variable~#1~used! }
7905 \_msg_kernel_new:nnn { kernel } { misused-sequence }
7906 { A~sequence~was~misused. }
7907 \_msg_kernel_new:nnn { kernel } { misused-prop }
7908 { A~property~list~was~misused. }
7909 \_msg_kernel_new:nnn { kernel } { negative-replication }
7910 { Negative~argument~for~\prg_replicate:nn. }
7911 \_msg_kernel_new:nnn { kernel } { unknown-comparison }
7912 { Relation~symbol~'~#1'~unknown:~use~,~<,~>,~==,~!=,~<=,~>=. }
7913 \_msg_kernel_new:nnn { kernel } { zero-step }
7914 { Zero~step~size~for~step~function~#1. }

```

Messages used by the “show” functions.

```

7915 \_msg_kernel_new:nnn { kernel } { show-clist }
7916 {
7917   The~comma~list~
7918   \str_if_eq:nnF {#1} { \l__clist_internal_clist } { \token_to_str:N #1~}
7919   \clist_if_empty:NTF #1
7920   { is~empty }
7921   { contains~the~items~(without~outer~braces): }
7922 }
7923 \_msg_kernel_new:nnn { kernel } { show-prop }
7924 {

```

```

7925 The~property~list~\token_to_str:N #1~
7926 \prop_if_empty:NTF #1
7927 { is~empty }
7928 { contains~the~pairs~(without~outer~braces): }
7929 }
7930 \__msg_kernel_new:nnn { kernel } { show-seq }
7931 {
7932 The~sequence~\token_to_str:N #1~
7933 \seq_if_empty:NTF #1
7934 { is~empty }
7935 { contains~the~items~(without~outer~braces): }
7936 }
7937 \__msg_kernel_new:nnn { kernel } { show-no-stream }
7938 { No~ #1 ~streams~are~open }
7939 \__msg_kernel_new:nnn { kernel } { show-open-streams }
7940 { The~following~ #1 ~streams~are~in~use: }

```

17.6 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 `\c_zero` prevents losing braces around the user-inserted text if any, and stops the expansion of `\romannumeral`.

```

7941 \group_begin:
7942 \char_set_catcode_math_superscript:N \^
7943 \char_set_lccode:nn { ^ } { \ }
7944 \char_set_lccode:nn { L } { 'L }
7945 \char_set_lccode:nn { T } { 'T }
7946 \char_set_lccode:nn { X } { 'X }
7947 \tl_to_lowercase:n
7948 {
7949 \cs_new:Npx \__msg_expandable_error:n #1
7950 {
7951 \exp_not:n
7952 {
7953 \tex_romannumeral:D
7954 \exp_after:wN \exp_after:wN
7955 \exp_after:wN \__msg_expandable_error:w
7956 \exp_after:wN \exp_after:wN
7957 \exp_after:wN \c_zero

```

```

7958     }
7959     \exp_not:N \use:n { \exp_not:c { LaTeX3-error: } ^ #1 } ^
7960   }
7961   \cs_new:Npn \__msg_expandable_error:w #1 ^ #2 ^ { #1 }
7962 }
7963 \group_end:

```

(End definition for `__msg_expandable_error:n`. This function is documented on page 149.)

`__msg_kernel_expandable_error:nnnnnn` 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`.

```

7964 \cs_new:Npn \__msg_kernel_expandable_error:nnnnnn #1#2#3#4#5#6
7965 {
7966   \exp_args:Nf \__msg_expandable_error:n
7967   {
7968     \exp_args:NNc \exp_after:wN \exp_stop_f:
7969     { \c_@@_text_prefix_tl LaTeX / #1 / #2 }
7970     {#3} {#4} {#5} {#6}
7971   }
7972 }
7973 \cs_new:Npn \__msg_kernel_expandable_error:nnnnn #1#2#3#4#5
7974 {
7975   \__msg_kernel_expandable_error:nnnnnn
7976   {#1} {#2} {#3} {#4} {#5} { }
7977 }
7978 \cs_new:Npn \__msg_kernel_expandable_error:nnnn #1#2#3#4
7979 {
7980   \__msg_kernel_expandable_error:nnnnnn
7981   {#1} {#2} {#3} {#4} { } { }
7982 }
7983 \cs_new:Npn \__msg_kernel_expandable_error:nnn #1#2#3
7984 {
7985   \__msg_kernel_expandable_error:nnnnnn
7986   {#1} {#2} {#3} { } { } { }
7987 }
7988 \cs_new:Npn \__msg_kernel_expandable_error:nn #1#2
7989 {
7990   \__msg_kernel_expandable_error:nnnnnn
7991   {#1} {#2} { } { } { } { }
7992 }

```

(End definition for `__msg_kernel_expandable_error:nnnnnn` and others. These functions are documented on page ??.)

17.7 Showing variables

Functions defined in this section are used for diagnostic functions in `l3clist`, `l3file`, `l3prop`, `l3seq`, `xtemplate`

`__msg_term:nnnnnn` Print the text of a message to the terminal without formatting: short cuts around `\iow_`
`__msg_term:nnnnnV` wrap:nnnN.
`__msg_term:nnnnn`
`__msg_term:nnn`
`__msg_term:nn`

```

7993 \cs_new_protected:Npn \__msg_term:nnnnnn #1#2#3#4#5#6
7994 {
7995   \iow_wrap:nnnN
7996     { \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
7997     { } { } \iow_term:n
7998 }
7999 \cs_generate_variant:Nn \__msg_term:nnnnnn { nnnnnV }
8000 \cs_new_protected:Npn \__msg_term:nnnnnn #1#2#3#4#5
8001 { \__msg_term:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
8002 \cs_new_protected:Npn \__msg_term:nnnn #1#2#3
8003 { \__msg_term:nnnnnn {#1} {#2} {#3} { } { } { } }
8004 \cs_new_protected:Npn \__msg_term:nn #1#2
8005 { \__msg_term:nnnnnn {#1} {#2} { } { } { } { } }

```

(End definition for __msg_term:nnnnnn and __msg_term:nnnnnV. These functions are documented on page ??.)

```

\__msg_show_variable:Nnn
\__msg_show_variable:n
\__msg_show_variable_aux:n
\__msg_show_variable_aux:w

```

The arguments of __msg_show_variable:Nnn are

- The $\langle variable \rangle$ to be shown.
- The type of the variable.
- A mapping of the form \seq_map_function:NN $\langle variable \rangle$ __msg_show_item:n, which produces the formatted string.

As for __kernel_register_show:N, check that the variable is defined. If it is, output the introductory message, then show the contents #3 using __msg_show_variable:n. This wraps the contents (with leading $\>_$) to a fixed number of characters per line. The expansion of #3 may either be empty or start with $\>_$. A leading $\>$, if present, is removed using a w-type auxiliary, as well as a space following it (via f-expansion). Note that we cannot remove the space as a delimiter for the w-type auxiliary, because a line-break may be taken there, and the space would then disappear. Finally, the resulting token list \l__msg_internal_tl is displayed to the terminal, with an odd \exp_after:wN which expands the closing brace to improve the output slightly.

```

8006 \cs_new_protected:Npn \__msg_show_variable:Nnn #1#2#3
8007 {
8008   \cs_if_exist:NTF #1
8009   {
8010     \__msg_term:nnn { LaTeX / kernel } { show- #2 } {#1}
8011     \__msg_show_variable:n {#3}
8012   }
8013   {
8014     \__msg_kernel_error:nnx { kernel } { variable-not-defined }
8015     { \token_to_str:N #1 }
8016   }
8017 }
8018 \cs_new_protected:Npn \__msg_show_variable:n #1
8019 { \iow_wrap:nnnN {#1} { } { } \__msg_show_variable_aux:n }
8020 \cs_new_protected:Npn \__msg_show_variable_aux:n #1
8021 {

```

```

8022 \tl_if_empty:nTF {#1}
8023 { \tl_clear:N \l__msg_internal_tl }
8024 { \tl_set:Nf \l__msg_internal_tl { \__msg_show_variable_aux:w #1 } }
8025 \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
8026 { \exp_after:wN \l__msg_internal_tl }
8027 }
8028 \cs_new:Npn \__msg_show_variable_aux:w #1 > { }

```

(End definition for __msg_show_variable:Nnn. This function is documented on page 149.)

```

\__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.

```

8029 \cs_new:Npn \__msg_show_item:n #1
8030 {
8031   \> \ \ \{ \tl_to_str:n {#1} \}
8032 }
8033 \cs_new:Npn \__msg_show_item:nn #1#2
8034 {
8035   \> \ \ \{ \tl_to_str:n {#1} \}
8036   \ \ => \ \ \{ \tl_to_str:n {#2} \}
8037 }
8038 \cs_new:Npn \__msg_show_item_unbraced:nn #1#2
8039 {
8040   \> \ \ \{ \tl_to_str:n {#1}
8041   \ \ => \ \ \{ \tl_to_str:n {#2}
8042   }

```

(End definition for __msg_show_item:n. This function is documented on page 149.)

```

8043 </initex | package>

```

18 l3keys Implementation

```

8044 <*initex | package>
8045 <*package>
8046 \ProvidesExplPackage
8047   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
8048 \__expl_package_check:
8049 </package>

```

18.1 Low-level interface

```

8050 <@@=keyval>

```

For historical reasons this code uses the ‘keyval’ module prefix.

\g__keyval_level_int To allow nesting of \keyval_parse:NNn, an integer is needed for the current level.

```

8051 \int_new:N \g__keyval_level_int

```

(End definition for \g__keyval_level_int. This variable is documented on page ??.)

```

\l__keyval_key_tl
\l__keyval_value_tl

```

The current key name and value.

```

8052 \tl_new:N \l__keyval_key_tl
8053 \tl_new:N \l__keyval_value_tl

```

(End definition for \l__keyval_key_tl and \l__keyval_value_tl. These variables are documented on page ??.)

\l__keyval_sanitise_tl Token list variables for dealing with awkward category codes in the input.

```
\l__keyval_parse_tl
8054 \tl_new:N \l__keyval_sanitise_tl
8055 \tl_new:N \l__keyval_parse_tl
```

(End definition for \l__keyval_sanitise_tl. This function is documented on page ??.)

__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.

```
8056 \group_begin:
8057 \char_set_catcode_active:n { '=' }
8058 \char_set_catcode_active:n { '\', }
8059 \char_set_lccode:nn { '\8 } { '=' }
8060 \char_set_lccode:nn { '\9 } { '\', }
8061 \tl_to_lowercase:n
8062 {
8063 \group_end:
8064 \cs_new_protected:Npn \__keyval_parse:n #1
8065 {
8066 \group_begin:
8067 \tl_clear:N \l__keyval_sanitise_tl
8068 \tl_set:Nn \l__keyval_sanitise_tl {#1}
8069 \tl_replace_all:Nnn \l__keyval_sanitise_tl { = } { 8 }
8070 \tl_replace_all:Nnn \l__keyval_sanitise_tl { , } { 9 }
8071 \tl_clear:N \l__keyval_parse_tl
8072 \exp_after:wN \__keyval_parse_elt:w \exp_after:wN
8073 \q_nil \l__keyval_sanitise_tl 9 \q_recursion_tail 9 \q_recursion_stop
8074 \exp_after:wN \group_end:
8075 \l__keyval_parse_tl
8076 }
8077 }
```

(End definition for __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. If this is the case, the system loops round. If there is something to parse, there is a check for the end of the input before handing off.

```
8078 \cs_new_protected:Npn \__keyval_parse_elt:w #1 ,
8079 {
8080 \tl_if_blank:oTF { \use_none:n #1 }
8081 { \__keyval_parse_elt:w \q_nil }
8082 {
8083 \quark_if_recursion_tail_stop:o { \use_ii:nn #1 }
8084 \__keyval_split_key_value:w #1 = = \q_stop
8085 \__keyval_parse_elt:w \q_nil
8086 }
8087 }
```

(End definition for __keyval_parse_elt:w. This function is documented on page ??.)

`__keyval_split_key_value:w` The key and value are handled separately. First the key is grabbed and saved as `\l__keyval_key_tl`. Then a check is need to see if there is a value at all: if not then the key name is simply added to the output. If there is a value then there is a check to ensure that there was only one `=` in the input (remembering some extra ones are around at the moment to prevent errors). All being well, there is an hand-off to find the value: the `\q_nil` is there to prevent loss of braces.

```

8088 \cs_new_protected:Npn \__keyval_split_key_value:w #1 = #2 \q_stop
8089 {
8090   \__keyval_split_key:n {#1}
8091   \str_if_eq:nnTF {#2} { = }
8092   {
8093     \tl_put_right:Nx \l__keyval_parse_tl
8094     {
8095       \exp_not:c
8096       { __keyval_key_no_value_elt_ \int_use:N \g__keyval_level_int :n }
8097       { \exp_not:o \l__keyval_key_tl }
8098     }
8099   }
8100   {
8101     \__keyval_split_key_value:wTF #2 \q_no_value \q_stop
8102     { \__keyval_split_value:w \q_nil #2 }
8103     { \__msg_kernel_error:nn { kernel } { misplaced-equals-sign } }
8104   }
8105 }
8106 \cs_new:Npn \__keyval_split_key_value:wTF #1 = #2#3 \q_stop
8107 { \tl_if_head_eq_meaning:nNTF {#3} \q_no_value }

```

(End definition for __keyval_split_key_value:w. This function is documented on page ??.)

`__keyval_split_key:n` 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. (This code does not have to cover the case with no key, as that's already taken out above.)

```

8108 \cs_new_protected:Npn \__keyval_split_key:n #1
8109 {
8110   \quark_if_nil:oTF { \use_none:nnn #1 \q_nil \q_nil }
8111   { \tl_set:Nx \l__keyval_key_tl { \exp_not:o { \use_ii:nnn #1 \q_nil } } }
8112   { \__keyval_split_key:w #1 \q_stop }
8113 }
8114 \cs_new_protected:Npn \__keyval_split_key:w \q_nil #1 \q_stop
8115 { \tl_set:Nx \l__keyval_key_tl { \tl_trim_spaces:n {#1} } }

```

(End definition for __keyval_split_key:n. This function is documented on page ??.)

`__keyval_split_value:w` Here the value has to be separated from the equals signs and the leading `\q_nil` added in to keep the brace levels. Fist the processing function can be added to the output list. If there is no value, setting `\l__keyval_value_tl` with three groups removed will leave nothing at all, and so an empty group can be added to the parsed list. On the other

hand, if the value is entirely contained within a set of braces then `\l__keyval_value_tl` will contain `\q_nil` only. In that case, strip off the leading quark using `\use_ii:nnn`, which also deals with any spaces.

```

8116 \cs_new_protected:Npn \__keyval_split_value:w #1 = =
8117 {
8118   \tl_put_right:Nx \l__keyval_parse_tl
8119   {
8120     \exp_not:c
8121     { __keyval_key_value_elt_ \int_use:N \g__keyval_level_int :nn }
8122     { \exp_not:o \l__keyval_key_tl }
8123   }
8124   \tl_set:Nx \l__keyval_value_tl
8125   { \exp_not:o { \use_none:nnn #1 \q_nil \q_nil } }
8126   \tl_if_empty:NTF \l__keyval_value_tl
8127   { \tl_put_right:Nn \l__keyval_parse_tl { { } } }
8128   {
8129     \quark_if_nil:NTF \l__keyval_value_tl
8130     {
8131       \tl_put_right:Nx \l__keyval_parse_tl
8132       { { \exp_not:o { \use_ii:nnn #1 \q_nil } } }
8133     }
8134     { \__keyval_split_value_aux:w #1 \q_stop }
8135   }
8136 }

```

A similar idea to the key code: remove the spaces from each end and deal with one set of braces.

```

8137 \cs_new_protected:Npn \__keyval_split_value_aux:w \q_nil #1 \q_stop
8138 {
8139   \tl_set:Nx \l__keyval_value_tl { \tl_trim_spaces:n {#1} }
8140   \tl_put_right:Nx \l__keyval_parse_tl
8141   { { \exp_not:o \l__keyval_value_tl } }
8142 }

```

(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.

```

8143 \cs_new_protected:Npn \keyval_parse:NNn #1#2#3
8144 {
8145   \int_gincr:N \g__keyval_level_int
8146   \cs_gset_eq:cN
8147   { __keyval_key_no_value_elt_ \int_use:N \g__keyval_level_int :n } #1
8148   \cs_gset_eq:cN
8149   { __keyval_key_value_elt_ \int_use:N \g__keyval_level_int :nn } #2
8150   \__keyval_parse:n {#3}
8151   \int_gdecr:N \g__keyval_level_int
8152 }

```

(End definition for `\keyval_parse:NNn`. This function is documented on page 161.)

One message for the low level parsing system.

```

8153 \__msg_kernel_new:nnnn { kernel } { misplaced-equals-sign }

```

```

8154 { Misplaced~equals~sign~in~key~value~input~\msg_line_number: }
8155 {
8156   LaTeX~is~attempting~to~parse~some~key~value~input~but~found~
8157   two~equals~signs~not~separated~by~a~comma.
8158 }

```

18.2 Constants and variables

```

8159 <@@=keys>

```

`\c__keys_code_root_tl` The prefixes for the code and variables of the keys themselves.
`\c__keys_info_root_tl`

```

8160 \tl_const:Nn \c__keys_code_root_tl { key~code~>~ }
8161 \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.

```

8162 \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 set.
`\l_keys_choice_tl`

```

8163 \int_new:N \l_keys_choice_int
8164 \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 156.)

`\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.

```

8165 \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.

```

8166 \tl_new:N \l_keys_key_tl

```

(End definition for `\l_keys_key_tl`. This variable is documented on page 158.)

`\l__keys_module_tl` The module for an entire set of keys.

```

8167 \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.

```

8168 \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.

```

8169 \bool_new:N \l_keys_only_known_bool

```

(End definition for `\l_keys_only_known_bool`. This variable is documented on page ??.)

<code>\l_keys_path_tl</code>	The “path” of the current key is stored here: this is available to the programmer and so is public. <code>8170 \tl_new:N \l_keys_path_tl</code> <i>(End definition for \l_keys_path_tl. This variable is documented on page 158.)</i>
<code>\l__keys_property_tl</code>	The “property” begin set for a key at definition time is stored here. <code>8171 \tl_new:N \l__keys_property_tl</code> <i>(End definition for \l__keys_property_tl. This variable is documented on page ??.)</i>
<code>\l_keys_selective_bool</code> <code>\l_keys_filtered_bool</code>	Two flags for using key groups: one to indicate that “selective” setting is active, a second to specify which type (“opt-in” or “opt-out”). <code>8172 \bool_new:N \l_keys_selective_bool</code> <code>8173 \bool_new:N \l_keys_filtered_bool</code> <i>(End definition for \l_keys_selective_bool and \l_keys_filtered_bool. These variables are documented on page ??.)</i>
<code>\l_keys_selective_seq</code>	The list of key groups being filtered in or out during selective setting. <code>8174 \seq_new:N \l__keys_selective_seq</code> <i>(End definition for \l_keys_selective_seq. This variable is documented on page ??.)</i>
<code>\l__keys_unused_clist</code>	Used when setting only some keys to store those left over. <code>8175 \tl_new:N \l__keys_unused_clist</code> <i>(End definition for \l__keys_unused_clist. This variable is documented on page ??.)</i>
<code>\l_keys_value_tl</code>	The value given for a key: may be empty if no value was given. <code>8176 \tl_new:N \l_keys_value_tl</code> <i>(End definition for \l_keys_value_tl. This variable is documented on page 158.)</i>
<code>\l__keys_tmp_bool</code>	Scratch space. <code>8177 \bool_new:N \l__keys_tmp_bool</code> <i>(End definition for \l__keys_tmp_bool. This variable is documented on page ??.)</i>

18.3 The key defining mechanism

<code>\keys_define:nn</code> <code>__keys_define:nnn</code> <code>__keys_define:onn</code>	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). <code>8178 \cs_new_protected:Npn \keys_define:nn</code> <code>8179 { __keys_define:onn \l__keys_module_tl }</code> <code>8180 \cs_new_protected:Npn __keys_define:nnn #1#2#3</code> <code>8181 {</code> <code>8182 \tl_set:Nx \l__keys_module_tl { \tl_to_str:n {#2} }</code> <code>8183 \keyval_parse:NNn __keys_define_elt:n __keys_define_elt:nn {#3}</code> <code>8184 \tl_set:Nn \l__keys_module_tl {#1}</code> <code>8185 }</code> <code>8186 \cs_generate_variant:Nn __keys_define:nnn { o }</code> <i>(End definition for \keys_define:nn. This function is documented on page 151.)</i>
--	--

`_keys_define_elt:n` The outer functions here record whether a value was given and then converge on a
`_keys_define_elt:nn` common internal mechanism. There is first a search for a property in the current key
`_keys_define_elt_aux:nn` name, then a check to make sure it is known before the code hands off to the next step.

```

8187 \\cs_new_protected:Npn \\_keys_define_elt:n #1
8188 {
8189   \\bool_set_true:N \\l__keys_no_value_bool
8190   \\_keys_define_elt_aux:nn {#1} { }
8191 }
8192 \\cs_new_protected:Npn \\_keys_define_elt:nn #1#2
8193 {
8194   \\bool_set_false:N \\l__keys_no_value_bool
8195   \\_keys_define_elt_aux:nn {#1} {#2}
8196 }
8197 \\cs_new_protected:Npn \\_keys_define_elt_aux:nn #1#2
8198 {
8199   \\_keys_property_find:n {#1}
8200   \\cs_if_exist:cTF { \\c__keys_props_root_tl \\l__keys_property_tl }
8201   { \\_keys_define_key:n {#2} }
8202   {
8203     \\str_if_eq_x:nnF { \\l__keys_property_tl } { .abort: }
8204     {
8205       \\_msg_kernel_error:nnxx { kernel } { property-unknown }
8206       { \\l__keys_property_tl } { \\l_keys_path_tl }
8207     }
8208   }
8209 }

```

(End definition for `_keys_define_elt:n`. This function is documented on page ??.)

`_keys_property_find:n` Searching for a property means finding the last . in the input, and storing the text before
`_keys_property_find:w` and after it. Everything is turned into strings, so there is no problem using an x-type
 expansion.

```

8210 \\cs_new_protected:Npn \\_keys_property_find:n #1
8211 {
8212   \\tl_set:Nx \\l_keys_path_tl { \\l__keys_module_tl / }
8213   \\tl_if_in:nnTF {#1} { . }
8214   { \\_keys_property_find:w #1 \\q_stop }
8215   {
8216     \\_msg_kernel_error:nnx { kernel } { key-no-property } {#1}
8217     \\tl_set:Nn \\l__keys_property_tl { .abort: }
8218   }
8219 }
8220 \\cs_new_protected:Npn \\_keys_property_find:w #1 . #2 \\q_stop
8221 {
8222   \\tl_set:Nx \\l_keys_path_tl { \\l_keys_path_tl \\tl_to_str:n {#1} }
8223   \\tl_if_in:nnTF {#2} { . }
8224   {
8225     \\tl_set:Nx \\l_keys_path_tl { \\l_keys_path_tl . }
8226     \\_keys_property_find:w #2 \\q_stop
8227   }

```

```

8228     { \tl_set:Nn \l__keys_property_tl { . #2 } }
8229   }

```

(End definition for `__keys_property_find:n`. This function is documented on page ??.)

`__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.

```

8230 \cs_new_protected:Npn \__keys_define_key:n #1
8231 {
8232   \bool_if:NTF \l__keys_no_value_bool
8233   {
8234     \exp_after:wN \__keys_define_key:w
8235     \l__keys_property_tl \q_stop
8236     { \use:c { \c__keys_props_root_tl \l__keys_property_tl } }
8237     {
8238       \__msg_kernel_error:nnxx { kernel }
8239       { property-requires-value } { \l__keys_property_tl }
8240       { \l_keys_path_tl }
8241     }
8242   }
8243   { \use:c { \c__keys_props_root_tl \l__keys_property_tl } {#1} }
8244 }
8245 \cs_new_protected:Npn \__keys_define_key:w #1 : #2 \q_stop
8246 { \tl_if_empty:NTF {#2} }

```

(End definition for `__keys_define_key:n`. This function is documented on page ??.)

18.4 Turning properties into actions

`__keys_bool_set:Nn`
`__keys_bool_set:cn`

Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```

8247 \cs_new_protected:Npn \__keys_bool_set:Nn #1#2
8248 {
8249   \bool_if_exist:NF #1 { \bool_new:N #1 }
8250   \__keys_choice_make:
8251   \__keys_cmd_set:nx { \l_keys_path_tl / true }
8252   { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
8253   \__keys_cmd_set:nx { \l_keys_path_tl / false }
8254   { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
8255   \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
8256   {
8257     \__msg_kernel_error:nnx { kernel } { boolean-values-only }
8258     { \l_keys_key_tl }
8259   }
8260   \__keys_default_set:n { true }
8261 }
8262 \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.

`__keys_bool_set_inverse:cn`

```

8263 \cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
8264 {
8265   \bool_if_exist:NF #1 { \bool_new:N #1 }
8266   \__keys_choice_make:
8267   \__keys_cmd_set:nx { \l_keys_path_tl / true }
8268   { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
8269   \__keys_cmd_set:nx { \l_keys_path_tl / false }
8270   { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
8271   \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
8272   {
8273     \__msg_kernel_error:nxx { kernel } { boolean-values-only }
8274     { \l_keys_key_tl }
8275   }
8276   \__keys_default_set:n { true }
8277 }
8278 \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.

```

8279 \cs_new_protected_nopar:Npn \__keys_choice_make:
8280 {
8281   \__keys_cmd_set:nn { \l_keys_path_tl }
8282   { \__keys_choice_find:n {##1} }
8283   \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
8284   {
8285     \__msg_kernel_error:nxxx { kernel } { key-choice-unknown }
8286     { \l_keys_path_tl } {##1}
8287   }
8288 }
(End definition for \__keys_choice_make:.)

```

`__keys_choices_make:nn` Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

```

8289 \cs_new_protected:Npn \__keys_choices_make:nn #1#2
8290 {
8291   \__keys_choice_make:
8292   \int_zero:N \l_keys_choice_int
8293   \clist_map_inline:nn {#1}
8294   {
8295     \int_incr:N \l_keys_choice_int
8296     \__keys_cmd_set:nx { \l_keys_path_tl / ##1 }
8297     {
8298       \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
8299       \int_set:Nn \exp_not:N \l_keys_choice_int
8300       { \int_use:N \l_keys_choice_int }
8301       \exp_not:n {#2}
8302     }
8303   }

```

```
8304 }
(End definition for \__keys_choices_make:nn.)
```

__keys_cmd_set:nn Creating a new command means tidying up the properties and then making the internal
 __keys_cmd_set:nx function which actually does the work.

```
\__keys_cmd_set:Vn 8305 \cs_new_protected:Npn \__keys_cmd_set:nn #1#2
\__keys_cmd_set:Vo 8306 {
8307   \prop_clear_new:c { \c__keys_info_root_tl #1 }
8308   \cs_set:cpn { \c__keys_code_root_tl #1 } ##1 {#2}
8309 }
8310 \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.

```
8311 \cs_new_protected:Npn \__keys_default_set:n #1
8312 {
8313   \prop_if_exist:cT { \c__keys_info_root_tl \l_keys_path_tl }
8314   { \prop_put:cn { \c__keys_info_root_tl \l_keys_path_tl } { default } {#1} }
8315 }
(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.

```
8316 \cs_new_protected:Npn \__keys_groups_set:n #1
8317 {
8318   \prop_if_exist:cT { \c__keys_info_root_tl \l_keys_path_tl }
8319   {
8320     \clist_set:Nn \l__keys_groups_clist {#1}
8321     \prop_put:cnV { \c__keys_info_root_tl \l_keys_path_tl }
8322     { groups } \l__keys_groups_clist
8323   }
8324 }
(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
 __keys_initialise:wn into a module and a key name. At this stage, \l_keys_path_tl will contain / so a split
 is easy to do.

```
8325 \cs_new_protected:Npn \__keys_initialise:n #1
8326 { \exp_after:wN \__keys_initialise:wn \l_keys_path_tl \q_stop {#1} }
8327 \cs_new_protected:Npn \__keys_initialise:wn #1 / #2 \q_stop #3
8328 { \keys_set:nn {#1} { #2 = {#3} } }
(End definition for \__keys_initialise:n. This function is documented on page ??.)
```

__keys_meta_make:n To create a meta-key, simply set up to pass data through.

```
\__keys_meta_make:nn 8329 \cs_new_protected:Npn \__keys_meta_make:n #1
8330 {
8331   \__keys_cmd_set:Vo \l_keys_path_tl
8332   { \exp_after:wN \keys_set:nn \exp_after:wN { \l_keys_module_tl } {#1} }
```

```

8333 }
8334 \cs_new_protected:Npn \__keys_meta_make:nn #1#2
8335 { \__keys_cmd_set:Vn \l_keys_path_tl { \keys_set:nn {#1} {#2} } }
(End definition for \__keys_meta_make:n. This function is documented on page ??.)

```

`__keys_multichoice_find:n` Choices where several values can be selected are very similar to normal exclusive choices.
`__keys_multichoice_make:` There is just a slight change in implementation to map across a comma-separated list.
`__keys_multichoices_make:nn` This then requires that the appropriate set up takes place elsewhere.

```

8336 \cs_new:Npn \__keys_multichoice_find:n #1
8337 { \clist_map_function:nN {#1} \__keys_choice_find:n }
8338 \cs_new_protected_nopar:Npn \__keys_multichoice_make:
8339 {
8340   \__keys_cmd_set:nn { \l_keys_path_tl }
8341   { \__keys_multichoice_find:n {##1} }
8342   \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
8343   {
8344     \__msg_kernel_error:nnxx { kernel } { key-choice-unknown }
8345     { \l_keys_path_tl } {##1}
8346   }
8347 }
8348 \cs_new_protected:Npn \__keys_multichoices_make:nn #1#2
8349 {
8350   \__keys_multichoice_make:
8351   \int_zero:N \l_keys_choice_int
8352   \clist_map_inline:nn {#1}
8353   {
8354     \int_incr:N \l_keys_choice_int
8355     \__keys_cmd_set:nx { \l_keys_path_tl / ##1 }
8356     {
8357       \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
8358       \int_set:Nn \exp_not:N \l_keys_choice_int
8359       { \int_use:N \l_keys_choice_int }
8360       \exp_not:n {#2}
8361     }
8362   }
8363 }

```

(End definition for __keys_multichoice_find:n. This function is documented on page ??.)

`__keys_value_requirement:n` Values can be required or forbidden by having the appropriate marker set. First, both the required and forbidden ones are clear, just in case!

```

8364 \cs_new_protected:Npn \__keys_value_requirement:n #1
8365 {
8366   \prop_if_exist:cT { \c__keys_info_root_tl \l_keys_path_tl }
8367   {
8368     \prop_remove:cn { \c__keys_info_root_tl \l_keys_path_tl } { required }
8369     \prop_remove:cn { \c__keys_info_root_tl \l_keys_path_tl } { forbidden }
8370     \prop_put:cnn { \c__keys_info_root_tl \l_keys_path_tl } {#1} { true }
8371   }
8372 }

```


(End definition for `_keys_value_requirement:n`.)

`_keys_variable_set:NnnN` Setting a variable takes the type and scope separately so that it is easy to make a new
`_keys_variable_set:cnnN` variable if needed.

```
8373 \cs_new_protected:Npn \_keys_variable_set:NnnN #1#2#3#4
8374 {
8375   \use:c { #2_if_exist:Nf } #1 { \use:c { #2_new:N } #1 }
8376   \_keys_cmd_set:nx { \_keys_path_tl }
8377   { \exp_not:c { #2 _ #3 set:N #4 } \exp_not:N #1 \exp_not:n { {##1} } }
8378 }
8379 \cs_generate_variant:Nn \_keys_variable_set:NnnN { c }
```

(End definition for `_keys_variable_set:NnnN` and `_keys_variable_set:cnnN`.)

18.5 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
8380 \cs_new_protected:cpn { \_keys_props_root_tl .bool_set:N } #1
8381 { \_keys_bool_set:Nn #1 { } }
8382 \cs_new_protected:cpn { \_keys_props_root_tl .bool_set:c } #1
8383 { \_keys_bool_set:cn {#1} { } }
8384 \cs_new_protected:cpn { \_keys_props_root_tl .bool_gset:N } #1
8385 { \_keys_bool_set:Nn #1 { g } }
8386 \cs_new_protected:cpn { \_keys_props_root_tl .bool_gset:c } #1
8387 { \_keys_bool_set:cn {#1} { g } }
```

(End definition for `.bool_set:N` and `.bool_set:c`. These functions are documented on page 152.)

`.bool_set_inverse:N` One function for this.

```
.bool_set_inverse:c
8388 \cs_new_protected:cpn { \_keys_props_root_tl .bool_set_inverse:N } #1
8389 { \_keys_bool_set_inverse:Nn #1 { } }
8390 \cs_new_protected:cpn { \_keys_props_root_tl .bool_set_inverse:c } #1
8391 { \_keys_bool_set_inverse:cn {#1} { } }
8392 \cs_new_protected:cpn { \_keys_props_root_tl .bool_gset_inverse:N } #1
8393 { \_keys_bool_set_inverse:Nn #1 { g } }
8394 \cs_new_protected:cpn { \_keys_props_root_tl .bool_gset_inverse:c } #1
8395 { \_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 152.)

`.choice:` Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```
8396 \cs_new_protected_nopar:cpn { \_keys_props_root_tl .choice: }
8397 { \_keys_choice_make: }
```

(End definition for `.choice:`. This function is documented on page 152.)

`.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
.choices:on
.choices:xn
8398 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:nn } #1
8399 { \__keys_choices_make:nn #1 }
8400 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:Vn } #1
8401 { \exp_args:NV \__keys_choices_make:nn #1 }
8402 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:on } #1
8403 { \exp_args:No \__keys_choices_make:nn #1 }
8404 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:xn } #1
8405 { \exp_args:Nx \__keys_choices_make:nn #1 }
```

(End definition for `.choices:nn` and others. These functions are documented on page 152.)

`.code:n` Creating code is simply a case of passing through to the underlying set function.

```
8406 \cs_new_protected:cpn { \c__keys_props_root_tl .code:n } #1
8407 { \__keys_cmd_set:nn { \l_keys_path_tl } {#1} }
```

(End definition for `.code:n`. This function is documented on page 152.)

`.clist_set:N`

`.clist_set:c`

`.clist_gset:N`

`.clist_gset:c`

```
8408 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:N } #1
8409 { \__keys_variable_set:NnnN #1 { clist } { } n }
8410 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:c } #1
8411 { \__keys_variable_set:cnnN {#1} { clist } { } n }
8412 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:N } #1
8413 { \__keys_variable_set:NnnN #1 { clist } { g } n }
8414 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:c } #1
8415 { \__keys_variable_set:cnnN {#1} { clist } { g } n }
```

(End definition for `.clist_set:N` and `.clist_set:c`. These functions are documented on page 152.)

`.default:n` Expansion is left to the internal functions.

`.default:V`

`.default:o`

`.default:x`

```
8416 \cs_new_protected:cpn { \c__keys_props_root_tl .default:n } #1
8417 { \__keys_default_set:n {#1} }
8418 \cs_new_protected:cpn { \c__keys_props_root_tl .default:V } #1
8419 { \exp_args:NV \__keys_default_set:n #1 }
8420 \cs_new_protected:cpn { \c__keys_props_root_tl .default:o } #1
8421 { \exp_args:No \__keys_default_set:n {#1} }
8422 \cs_new_protected:cpn { \c__keys_props_root_tl .default:x } #1
8423 { \exp_args:Nx \__keys_default_set:n {#1} }
```

(End definition for `.default:n` and others. These functions are documented on page 153.)

`.dim_set:N` Setting a variable is very easy: just pass the data along.

`.dim_set:c`

`.dim_gset:N`

`.dim_gset:c`

```
8424 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:N } #1
8425 { \__keys_variable_set:NnnN #1 { dim } { } n }
8426 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:c } #1
8427 { \__keys_variable_set:cnnN {#1} { dim } { } n }
8428 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:N } #1
8429 { \__keys_variable_set:NnnN #1 { dim } { g } n }
8430 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
8431 { \__keys_variable_set:cnnN {#1} { dim } { g } n }
```

(End definition for `.dim_set:N` and `.dim_set:c`. These functions are documented on page 153.)

`.fp_set:N` Setting a variable is very easy: just pass the data along.

```
8432 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:N } #1
8433 { \__keys_variable_set:NnnN #1 { fp } { } n }
8434 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:c } #1
8435 { \__keys_variable_set:cnnN {#1} { fp } { } n }
8436 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:N } #1
8437 { \__keys_variable_set:NnnN #1 { fp } { g } n }
8438 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
8439 { \__keys_variable_set:cnnN {#1} { fp } { g } n }
```

(End definition for `.fp_set:N` and `.fp_set:c`. These functions are documented on page 153.)

`.groups:n` A single property to create groups of keys.

```
8440 \cs_new_protected:cpn { \c__keys_props_root_tl .groups:n } #1
8441 { \__keys_groups_set:n {#1} }
```

(End definition for `.groups:n`. This function is documented on page 153.)

`.initial:n` The standard hand-off approach.

```
8442 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:n } #1
8443 { \__keys_initialise:n {#1} }
8444 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:V } #1
8445 { \exp_args:NV \__keys_initialise:n #1 }
8446 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:o } #1
8447 { \exp_args:No \__keys_initialise:n {#1} }
8448 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
8449 { \exp_args:Nx \__keys_initialise:n {#1} }
```

(End definition for `.initial:n` and others. These functions are documented on page 153.)

`.int_set:N` Setting a variable is very easy: just pass the data along.

```
8450 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:N } #1
8451 { \__keys_variable_set:NnnN #1 { int } { } n }
8452 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:c } #1
8453 { \__keys_variable_set:cnnN {#1} { int } { } n }
8454 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:N } #1
8455 { \__keys_variable_set:NnnN #1 { int } { g } n }
8456 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
8457 { \__keys_variable_set:cnnN {#1} { int } { g } n }
```

(End definition for `.int_set:N` and `.int_set:c`. These functions are documented on page 153.)

`.meta:n` Making a meta is handled internally.

```
8458 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:n } #1
8459 { \__keys_meta_make:n {#1} }
```

(End definition for `.meta:n`. This function is documented on page 154.)

`.meta:nn` Meta with path: potentially lots of variants, but for the moment no so many defined.

```
8460 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:nn } #1
8461 { \__keys_meta_make:nn #1 }
```

(End definition for `.meta:nn`. This function is documented on page 154.)

`.multichoice:` The same idea as `.choice:` and `.choices:nn`, but where more than one choice is allowed.

```

8462 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .multichoice: }
8463 { \__keys_multichoice_make: }
8464 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:nn } #1
8465 { \__keys_multichoices_make:nn #1 }
8466 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:Vn } #1
8467 { \exp_args:NV \__keys_multichoices_make:nn #1 }
8468 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:on } #1
8469 { \exp_args:No \__keys_multichoices_make:nn #1 }
8470 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:xn } #1
8471 { \exp_args:Nx \__keys_multichoices_make:nn #1 }

```

(End definition for `.multichoice:`. This function is documented on page 154.)

`.skip_set:N` Setting a variable is very easy: just pass the data along.

```

8472 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:N } #1
8473 { \__keys_variable_set:NnnN #1 { skip } { } n }
8474 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:c } #1
8475 { \__keys_variable_set:cnnN {#1} { skip } { } n }
8476 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:N } #1
8477 { \__keys_variable_set:NnnN #1 { skip } { g } n }
8478 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:c } #1
8479 { \__keys_variable_set:cnnN {#1} { skip } { g } n }

```

(End definition for `.skip_set:N` and `.skip_set:c`. These functions are documented on page 154.)

`.tl_set:N` Setting a variable is very easy: just pass the data along.

```

8480 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:N } #1
8481 { \__keys_variable_set:NnnN #1 { tl } { } n }
8482 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:c } #1
8483 { \__keys_variable_set:cnnN {#1} { tl } { } n }
8484 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:N } #1
8485 { \__keys_variable_set:NnnN #1 { tl } { } x }
8486 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:c } #1
8487 { \__keys_variable_set:cnnN {#1} { tl } { } x }
8488 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:N } #1
8489 { \__keys_variable_set:NnnN #1 { tl } { g } n }
8490 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:c } #1
8491 { \__keys_variable_set:cnnN {#1} { tl } { g } n }
8492 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:N } #1
8493 { \__keys_variable_set:NnnN #1 { tl } { g } x }
8494 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:c } #1
8495 { \__keys_variable_set:cnnN {#1} { tl } { g } x }

```

(End definition for `.tl_set:N` and `.tl_set:c`. These functions are documented on page 154.)

`.value_forbidden:` These are very similar, so both call the same function.

```

8496 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .value_forbidden: }
8497 { \__keys_value_requirement:n { forbidden } }
8498 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .value_required: }
8499 { \__keys_value_requirement:n { required } }

```

(End definition for `.value_forbidden:`. This function is documented on page 154.)

18.6 Setting keys

```
\keys_set:nn
\keys_set:nV
\keys_set:nv
\keys_set:no
\__keys_set:nnn
\__keys_set:onn
8500 \cs_new_protected_nopar:Npn \keys_set:nn
8501 { \__keys_set:onn { \l__keys_module_tl } }
8502 \cs_new_protected:Npn \__keys_set:nnn #1#2#3
8503 {
8504   \tl_set:Nx \l__keys_module_tl { \tl_to_str:n {#2} }
8505   \keyval_parse:Nn \__keys_set_elt:n \__keys_set_elt:nn {#3}
8506   \tl_set:Nn \l__keys_module_tl {#1}
8507 }
8508 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
8509 \cs_generate_variant:Nn \__keys_set:nnn { o }
```

(End definition for `\keys_set:nn` and others. These functions are documented on page ??.)

`\keys_set_known:nnN` Setting known keys simply means setting the appropriate flag, then running the standard code.

```
\keys_set_known:nVN
\keys_set_known:nvN
\keys_set_known:noN
\keys_set_known:nn
\keys_set_known:nV
\keys_set_known:nv
\keys_set_known:nnN
8510 \cs_new_protected:Npn \keys_set_known:nnN #1#2#3
8511 {
8512   \clist_clear:N \l__keys_unused_clist
8513   \keys_set_known:nn {#1} {#2}
8514   \tl_set:Nx #3 { \exp_not:o { \l__keys_unused_clist } }
8515 }
8516 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
8517 \cs_new_protected:Npn \keys_set_known:nn #1#2
8518 {
8519   \bool_set_true:N \l__keys_only_known_bool
8520   \keys_set:nn {#1} {#2}
8521   \bool_set_false:N \l__keys_only_known_bool
8522 }
8523 \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 ??.)

`\keys_set_filter:nnnN` The idea of setting keys in a selective manner again uses flags wrapped around the basic code.

```
\keys_set_filter:nnVN
\keys_set_filter:nnn
\keys_set_filter:nnV
\keys_set_filter:nnv
\keys_set_filter:nnv\keys_set_filter:nno
\keys_set_groups:nnn
\keys_set_groups:nnV
\keys_set_groups:nnv\keys_set_groups:nno
8524 \cs_new_protected:Npn \keys_set_filter:nnnN #1#2#3#4
8525 {
8526   \clist_clear:N \l__keys_unused_clist
8527   \keys_set_filter:nnn {#1} {#2} {#3}
8528   \tl_set:Nx #4 { \exp_not:o { \l__keys_unused_clist } }
8529 }
8530 \cs_generate_variant:Nn \keys_set_filter:nnnN { nnV , nnv , nno }
8531 \cs_new_protected:Npn \keys_set_filter:nnn #1#2#3
8532 {
8533   \bool_set_true:N \l__keys_selective_bool
8534   \bool_set_true:N \l__keys_filtered_bool
```

```

8535     \seq_set_from_clist:Nn \l__keys_selective_seq {#2}
8536     \keys_set:nn {#1} {#3}
8537     \bool_set_false:N \l__keys_selective_bool
8538   }
8539   \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
8540   \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
8541   {
8542     \bool_set_true:N \l__keys_selective_bool
8543     \bool_set_false:N \l__keys_filtered_bool
8544     \seq_set_from_clist:Nn \l__keys_selective_seq {#2}
8545     \keys_set:nn {#1} {#3}
8546     \bool_set_false:N \l__keys_selective_bool
8547   }
8548   \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:nnoN.
These functions are documented on page ??.)

`__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:nn` move on to execute the code.
`__keys_set_elt_aux:`
`__keys_set_elt_selective:`

```

8549   \cs_new_protected:Npn \__keys_set_elt:n #1
8550   {
8551     \bool_set_true:N \l__keys_no_value_bool
8552     \__keys_set_elt_aux:nn {#1} { }
8553   }
8554   \cs_new_protected:Npn \__keys_set_elt:nn #1#2
8555   {
8556     \bool_set_false:N \l__keys_no_value_bool
8557     \__keys_set_elt_aux:nn {#1} {#2}
8558   }
8559   \cs_new_protected:Npn \__keys_set_elt_aux:nn #1#2
8560   {
8561     \tl_set:Nx \l_keys_key_tl { \tl_to_str:n {#1} }
8562     \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / \l_keys_key_tl }
8563     \__keys_value_or_default:n {#2}
8564     \bool_if:NTF \l__keys_selective_bool
8565     { \__keys_set_elt_selective: }
8566     { \__keys_set_elt_aux: }
8567   }
8568   \cs_new_protected_nopar:Npn \__keys_set_elt_aux:
8569   {
8570     \bool_if:nTF
8571     {
8572       \__keys_if_value_p:n { required } &&
8573       \l__keys_no_value_bool
8574     }
8575     {
8576       \__msg_kernel_error:nmx { kernel } { value-required }
8577       { \l_keys_path_tl }

```

```

8578     }
8579     {
8580         \bool_if:nTF
8581         {
8582             \__keys_if_value_p:n { forbidden } &&
8583             ! \l__keys_no_value_bool
8584         }
8585         {
8586             \__msg_kernel_error:nxxx { kernel } { value-forbidden }
8587             { \l_keys_path_tl } { \l_keys_value_tl }
8588         }
8589         { \__keys_execute: }
8590     }
8591 }

```

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.

```

8592 \cs_new_protected_nopar:Npn \__keys_set_elt_selective:
8593 {
8594     \prop_if_exist:cTF { \c__keys_info_root_tl \l_keys_path_tl }
8595     {
8596         \prop_get:cnNTF { \c__keys_info_root_tl \l_keys_path_tl }
8597         { groups } \l__keys_groups_clist
8598         { \__keys_check_groups: }
8599         {
8600             \bool_if:NTF \l__keys_filtered_bool
8601             { \__keys_set_elt_aux: }
8602             { \__keys_store_unused: }
8603         }
8604     }
8605     {
8606         \bool_if:NTF \l__keys_filtered_bool
8607         { \__keys_set_elt_aux: }
8608         { \__keys_store_unused: }
8609     }
8610 }

```

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.

```

8611 \cs_new_protected_nopar:Npn \__keys_check_groups:
8612 {
8613     \bool_set_false:N \l__keys_tmp_bool
8614     \seq_map_inline:Nn \l__keys_selective_seq
8615     {
8616         \clist_map_inline:Nn \l__keys_groups_clist
8617         {
8618             \str_if_eq:nnT {##1} {####1}
8619             {

```

```

8620         \bool_set_true:N \l__keys_tmp_bool
8621         \clist_map_break:n { \seq_map_break: }
8622     }
8623 }
8624 }
8625 \bool_if:NTF \l__keys_tmp_bool
8626 {
8627     \bool_if:NTF \l__keys_filtered_bool
8628     { \__keys_store_unused: }
8629     { \__keys_set_elt_aux: }
8630 }
8631 {
8632     \bool_if:NTF \l__keys_filtered_bool
8633     { \__keys_set_elt_aux: }
8634     { \__keys_store_unused: }
8635 }
8636 }

```

(End definition for __keys_set_elt:n and __keys_set_elt:nn. These functions are documented on page ??.)

__keys_value_or_default:n If a value is given, return it as #1, otherwise send a default if available.

```

8637 \cs_new_protected:Npn \__keys_value_or_default:n #1
8638 {
8639     \bool_if:NTF \l__keys_no_value_bool
8640     {
8641         \prop_get:cnNF { \c__keys_info_root_tl \l_keys_path_tl }
8642         { default } \l_keys_value_tl
8643         { \tl_clear:N \l_keys_value_tl }
8644     }
8645     { \tl_set:Nn \l_keys_value_tl {#1} }
8646 }

```

(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.

```

8647 \prg_new_conditional:Npnn \__keys_if_value:n #1 { p }
8648 {
8649     \prop_if_exist:cTF { \c__keys_info_root_tl \l_keys_path_tl }
8650     {
8651         \prop_if_in:cnTF { \c__keys_info_root_tl \l_keys_path_tl } {#1}
8652         { \prg_return_true: }
8653         { \prg_return_false: }
8654     }
8655     { \prg_return_false: }
8656 }

```

(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

__keys_execute_unknown:
 __keys_execute:nn
 __keys_store_unused:

happens if a key is unknown depends on whether unknown keys are being skipped or if an error should be raised.

```

8657 \cs_new_protected_nopar:Npn \__keys_execute:
8658 { \__keys_execute:nn { \l_keys_path_tl } { \__keys_execute_unknown: } }
8659 \cs_new_protected_nopar:Npn \__keys_execute_unknown:
8660 {
8661   \bool_if:NTF \l_keys_only_known_bool
8662   { \__keys_store_unused: }
8663   {
8664     \__keys_execute:nn { \l_keys_module_tl / unknown }
8665     {
8666       \__msg_kernel_error:nxxx { kernel } { key-unknown }
8667       { \l_keys_path_tl } { \l_keys_module_tl }
8668     }
8669   }
8670 }
8671 \cs_new:Npn \__keys_execute:nn #1#2
8672 {
8673   \cs_if_exist:cTF { \c_keys_code_root_tl #1 }
8674   {
8675     \exp_args:Nc \exp_args:No { \c_keys_code_root_tl #1 }
8676     \l_keys_value_tl
8677   }
8678   {#2}
8679 }
8680 \cs_new_protected_nopar:Npn \__keys_store_unused:
8681 {
8682   \clist_put_right:Nx \l_keys_unused_clist
8683   {
8684     \exp_not:o \l_keys_key_tl
8685     \bool_if:NF \l_keys_no_value_bool
8686     { = { \exp_not:o \l_keys_value_tl } }
8687   }
8688 }

```

(End definition for `__keys_execute:..` This function is documented on page ??.)

`__keys_choice_find:n` Executing a choice has two parts. First, try the choice given, then if that fails call the unknown key. That will exist, as it is created when a choice is first made. So there is no need for any escape code.

```

8689 \cs_new:Npn \__keys_choice_find:n #1
8690 {
8691   \__keys_execute:nn { \l_keys_path_tl / \tl_to_str:n {#1} }
8692   { \__keys_execute:nn { \l_keys_path_tl / unknown } { } }
8693 }

```

(End definition for `__keys_choice_find:n`.)

18.7 Utilities

`\keys_if_exist_p:nn` A utility for others to see if a key exists.

`\keys_if_exist:nn`[TF](#)

```

8694 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
8695 {
8696   \cs_if_exist:cTF { \c__keys_code_root_tl #1 / #2 }
8697   { \prg_return_true: }
8698   { \prg_return_false: }
8699 }

```

(End definition for \keys_if_exist:nn. These functions are documented on page 160.)

\keys_if_choice_exist_p:nnn

Just an alternative view on \keys_if_exist:nn(TF).

\keys_if_choice_exist:nnnTF

```

8700 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3 { p , T , F , TF }
8701 {
8702   \cs_if_exist:cTF { \c__keys_code_root_tl #1 / #2 / #3 }
8703   { \prg_return_true: }
8704   { \prg_return_false: }
8705 }

```

(End definition for \keys_if_choice_exist:nnn. These functions are documented on page 160.)

\keys_show:nn

Showing a key is just a question of using the correct name.

```

8706 \cs_new_protected:Npn \keys_show:nn #1#2
8707 { \cs_show:c { \c__keys_code_root_tl #1 / \tl_to_str:n {#2} } }

```

(End definition for \keys_show:nn. This function is documented on page 160.)

18.8 Messages

For when there is a need to complain.

```

8708 \__msg_kernel_new:nnnn { kernel } { boolean-values-only }
8709 { Key~'#1'~accepts~boolean-values-only. }
8710 { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
8711 \__msg_kernel_new:nnnn { kernel } { choice-unknown }
8712 { Choice~'#2'~unknown~for~key~'#1'. }
8713 {
8714   The~key~'#1'~takes~a~limited~number~of~values.\\
8715   The~input~given,~'#2',~is~not~on~the~list~accepted.
8716 }
8717 \__msg_kernel_new:nnnn { kernel } { key-choice-unknown }
8718 { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
8719 {
8720   The~key~'#1'~only~accepts~predefined~values,~and~'#2'~is~not~one~of~these.
8721 }
8722 \__msg_kernel_new:nnnn { kernel } { key-no-property }
8723 { No~property~given~in~definition~of~key~'#1'. }
8724 {
8725   \c_msg_coding_error_text_tl
8726   Inside~\keys_define:nn  each~key~name~
8727   needs~a~property:  \\ \\
8728   \iow_indent:n { #1 .<property> } \\ \\
8729   LaTeX~did~not~find~a~'. ' ~to~indicate~the~start~of~a~property.
8730 }
8731 \__msg_kernel_new:nnnn { kernel } { key-unknown }

```

```

8732 { The~key~'#1'~is~unknown~and~is~being~ignored. }
8733 {
8734   The~module~'#2'~does~not~have~a~key~called~'#1'.\\
8735   Check~that~you~have~spelled~the~key~name~correctly.
8736 }
8737 \_msg_kernel_new:nnnn { kernel } { property-requires-value }
8738 { The~property~'#1'~requires~a~value. }
8739 {
8740   \c_msg_coding_error_text_tl
8741   LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'.\\
8742   No~value~was~given~for~the~property,~and~one~is~required.
8743 }
8744 \_msg_kernel_new:nnnn { kernel } { property-unknown }
8745 { The~key~property~'#1'~is~unknown. }
8746 {
8747   \c_msg_coding_error_text_tl
8748   LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
8749   this~property~is~not~defined.
8750 }
8751 \_msg_kernel_new:nnnn { kernel } { value-forbidden }
8752 { The~key~'#1'~does~not~taken~a~value. }
8753 {
8754   The~key~'#1'~should~be~given~without~a~value.\\
8755   LaTeX~will~ignore~the~given~value~'#2'.
8756 }
8757 \_msg_kernel_new:nnnn { kernel } { value-required }
8758 { The~key~'#1'~requires~a~value. }
8759 {
8760   The~key~'#1'~must~have~a~value.\\
8761   No~value~was~present:~the~key~will~be~ignored.
8762 }

```

18.9 Deprecated functions

_keys_choice_code_store:n Deprecated on 2013-07-09.

```

\_keys_choice_code_store:x      8763 \cs_new_protected:Npn \_keys_choice_code_store:n #1
    .choice_code:n             8764 {
    .choice_code:x              8765   \cs_if_exist:cF
\_keys_choices_generate:n      8766   { \c__keys_info_root_tl \l_keys_path_tl .choice-code }
    \_keys_choices_generate_aux:n 8767   {
    .generate_choices:n         8768     \tl_new:c
                                8769     { \c__keys_info_root_tl \l_keys_path_tl .choice-code }
                                8770   }
                                8771   \tl_set:cn { \c__keys_info_root_tl \l_keys_path_tl .choice-code }
                                8772   {#1}
                                8773 }
                                8774 \cs_generate_variant:Nn \_keys_choice_code_store:n { x }
                                8775 \cs_new_protected:cpn { \c__keys_props_root_tl .choice_code:n } #1
                                8776 { \_keys_choice_code_store:n {#1} }

```

```

8777 \cs_new_protected:cpn { \c__keys_props_root_tl .choice_code:x } #1
8778 { \__keys_choice_code_store:x {#1} }
8779 \cs_new_protected:Npn \__keys_choices_generate:n #1
8780 {
8781   \cs_if_exist:cTF
8782   { \c__keys_info_root_tl \l_keys_path_tl .choice~code }
8783   {
8784     \__keys_choice_make:
8785     \int_zero:N \l_keys_choice_int
8786     \clist_map_function:nN {#1} \__keys_choices_generate_aux:n
8787   }
8788   {
8789     \__msg_kernel_error:nxx { kernel }
8790     { generate-choices-before-code } { \l_keys_path_tl }
8791   }
8792 }
8793 \cs_new_protected:Npn \__keys_choices_generate_aux:n #1
8794 {
8795   \int_incr:N \l_keys_choice_int
8796   \__keys_cmd_set:nx { \l_keys_path_tl / #1 }
8797   {
8798     \tl_set:Nn \exp_not:N \l_keys_choice_tl {#1}
8799     \int_set:Nn \exp_not:N \l_keys_choice_int
8800     { \int_use:N \l_keys_choice_int }
8801     \exp_not:v
8802     { \c__keys_info_root_tl \l_keys_path_tl .choice~code }
8803   }
8804 }
8805 \__msg_kernel_new:nnnn { kernel } { generate-choices-before-code }
8806 { No~code~available~to~generate~choices~for~key~'#1'. }
8807 {
8808   \c_msg_coding_error_text_tl
8809   Before~using~.generate_choices:n~the~code~should~be~defined~
8810   with~'.choice_code:n'~or~'.choice_code:x'.
8811 }
8812 \cs_new_protected:cpn { \c__keys_props_root_tl .generate_choices:n } #1
8813 { \__keys_choices_generate:n {#1} }

```

(End definition for `__keys_choice_code_store:n` and `__keys_choice_code_store:x`. These functions are documented on page ??.)

```

8814 </initex | package>

```

19 l3file implementation

The following test files are used for this code: `m3file001`.

```

8815 <*initex | package>
8816 <@@=file>

```

```

8817 <*package>
8818 \ProvidesExplPackage
8819   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
8820   \__expl_package_check:
8821 </package>

```

19.1 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 package mode the current file name is collected from L^AT_EX 2_ε.

```

8822 \tl_new:N \g_file_current_name_tl
8823 <*initex>
8824 \tex_everyjob:D \exp_after:wN
8825   {
8826     \tex_the:D \tex_everyjob:D
8827     \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
8828   }
8829 </initex>
8830 <*package>
8831 \tl_gset_eq:NN \g_file_current_name_tl \@currname
8832 </package>

```

(End definition for \g_file_current_name_tl. This variable is documented on page 162.)

\g__file_stack_seq The input list of files is stored as a sequence stack.

```

8833 \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.

```

8834 \seq_new:N \g__file_record_seq
8835 <*initex>
8836 \tex_everyjob:D \exp_after:wN
8837   {
8838     \tex_the:D \tex_everyjob:D
8839     \seq_gput_right:NV \g__file_record_seq \g_file_current_name_tl
8840   }
8841 </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.

```

8842 \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.

```

8843 \tl_new:N \l__file_internal_name_tl
(End definition for \l__file_internal_name_tl. This variable is documented on page 168.)

```

`\l__file_search_path_seq` The current search path.

```

8844 \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.

```

8845 <*package>
8846 \seq_new:N \l_file_saved_search_path_seq
8847 </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.

```

8848 <*package>
8849 \seq_new:N \l__file_internal_seq
8850 </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.

```

8851 \cs_new_protected:Npn \__file_name_sanitize:nn #1#2
8852 {
8853   \group_begin:
8854     \seq_map_inline:Nn \l_char_active_seq
8855       { \cs_set_nopar:Npx ##1 { \token_to_str:N ##1 } }
8856     \tl_set:Nx \l__file_internal_name_tl {#1}
8857     \tl_set:Nx \l__file_internal_name_tl
8858       { \tl_to_str:N \l__file_internal_name_tl }
8859     \tl_if_in:NnT \l__file_internal_name_tl { ~ }
8860     {
8861       \__msg_kernel_error:nnx { kernel } { space-in-file-name }
8862       { \l__file_internal_name_tl }
8863       \tl_remove_all:Nn \l__file_internal_name_tl { ~ }
8864     }
8865     \use:x
8866     {
8867       \group_end:
8868       \exp_not:n {#2} { \l__file_internal_name_tl }
8869     }
8870   }
(End definition for \__file_name_sanitize:nn.)

```

`\file_add_path:nN`
`__file_add_path:nN`
`__file_add_path_search:nN`

The way to test if a file exists is to try to open it: if it does not exist then TeX will report end-of-file. For files which are in the current directory, this is straight-forward. For other locations, a search has to be made looking at each potential path in turn. The first location is of course treated as the correct one. If nothing is found, #2 is returned empty.

```

8871 \cs_new_protected:Npn \file_add_path:nN #1
8872 { \__file_name_sanitiz:n {#1} { \__file_add_path:nN } }
8873 \cs_new_protected:Npn \__file_add_path:nN #1#2
8874 {
8875   \__ior_open:Nn \g__file_internal_ior {#1}
8876   \ior_if_eof:NTF \g__file_internal_ior
8877     { \__file_add_path_search:nN {#1} #2 }
8878     { \tl_set:Nn #2 {#1} }
8879   \ior_close:N \g__file_internal_ior
8880 }
8881 \cs_new_protected:Npn \__file_add_path_search:nN #1#2
8882 {
8883   \tl_set:Nn #2 { \q_no_value }
8884   <*package>
8885   \cs_if_exist:NT \input@path
8886   {
8887     \seq_set_eq:NN \l__file_saved_search_path_seq \l__file_search_path_seq
8888     \seq_set_split:NnV \l__file_internal_seq { , } \input@path
8889     \seq_concat:NNN \l__file_search_path_seq
8890       \l__file_search_path_seq \l__file_internal_seq
8891   }
8892   </package>
8893   \seq_map_inline:Nn \l__file_search_path_seq
8894   {
8895     \__ior_open:Nn \g__file_internal_ior { ##1 #1 }
8896     \ior_if_eof:NF \g__file_internal_ior
8897     {
8898       \tl_set:Nx #2 { ##1 #1 }
8899       \seq_map_break:
8900     }
8901   }
8902   <*package>
8903   \cs_if_exist:NT \input@path
8904   { \seq_set_eq:NN \l__file_search_path_seq \l__file_saved_search_path_seq }
8905   </package>
8906 }

```

(End definition for `\file_add_path:nN`. This function is documented on page 162.)

`\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`.

```

8907 \prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
8908 {
8909   \file_add_path:nN {#1} \l__file_internal_name_tl

```

```

8910 \quark_if_no_value:NTF \l__file_internal_name_tl
8911 { \prg_return_false: }
8912 { \prg_return_true: }
8913 }

```

(End definition for \file_if_exist:nTF. This function is documented on page 162.)

\file_input:n

Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the \g__file_stack_seq, and add it to the file list, either \g__file_record_seq, or \@filelist in package mode.

```

\__file_input:n\__file_input:V
\__file_input_aux:n
\__file_input_aux:o

```

```

8914 \cs_new_protected:Npn \file_input:n #1
8915 {
8916   \file_add_path:nN {#1} \l__file_internal_name_tl
8917   \quark_if_no_value:NTF \l__file_internal_name_tl
8918   {
8919     \__file_name_sanitiz:nn {#1}
8920     { \__msg_kernel_error:nxx { kernel } { file-not-found } }
8921   }
8922   { \__file_input:V \l__file_internal_name_tl }
8923 }
8924 \cs_new_protected:Npn \__file_input:n #1
8925 {
8926   \tl_if_in:nnTF {#1} { . }
8927   { \__file_input_aux:n {#1} }
8928   { \__file_input_aux:o { \tl_to_str:n { #1 . tex } } }
8929 }
8930 \cs_generate_variant:Nn \__file_input:n { V }
8931 \cs_new_protected:Npn \__file_input_aux:n #1
8932 {
8933   <*initex>
8934   \seq_gput_right:Nn \g__file_record_seq {#1}
8935   </initex>
8936   <*package>
8937   \clist_if_exist:NTF \@filelist
8938   { \@addtofilelist {#1} }
8939   { \seq_gput_right:Nn \g__file_record_seq {#1} }
8940   </package>
8941   \seq_gpush:No \g__file_stack_seq \g_file_current_name_tl
8942   \tl_gset:Nn \g_file_current_name_tl {#1}
8943   \tex_input:D #1 \c_space_tl
8944   \seq_gpop:NN \g__file_stack_seq \l__file_internal_tl
8945   \tl_gset_eq:NN \g_file_current_name_tl \l__file_internal_tl
8946 }
8947 \cs_generate_variant:Nn \__file_input_aux:n { o }

```

(End definition for \file_input:n. This function is documented on page 162.)

\file_path_include:n

Wrapper functions to manage the search path.

\file_path_remove:n

```

\__file_path_include:n

```

```

8948 \cs_new_protected:Npn \file_path_include:n #1
8949 { \__file_name_sanitiz:nn {#1} { \__file_path_include:n } }
8950 \cs_new_protected:Npn \__file_path_include:n #1

```



```

8951 {
8952   \seq_if_in:NnF \l__file_search_path_seq {#1}
8953   { \seq_put_right:Nn \l__file_search_path_seq {#1} }
8954 }
8955 \cs_new_protected:Npn \file_path_remove:n #1
8956 {
8957   \__file_name_sanitiz:nn {#1}
8958   { \seq_remove_all:Nn \l__file_search_path_seq }
8959 }

```

(End definition for \file_path_include:n. This function is documented on page 163.)

\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 it into account (we capture it \AtBeginDocument into \g__file_record_seq), turning each file name into a string.

```

8960 \cs_new_protected_nopar:Npn \file_list:
8961 {
8962   \seq_set_eq:NN \l__file_internal_seq \g__file_record_seq
8963   <*package>
8964   \clist_if_exist:NT \@filelist
8965   {
8966     \clist_map_inline:Nn \@filelist
8967     {
8968       \seq_put_right:No \l__file_internal_seq
8969       { \tl_to_str:n {##1} }
8970     }
8971   }
8972   </package>
8973   \seq_remove_duplicates:N \l__file_internal_seq
8974   \iow_log:n { *~File~List~* }
8975   \seq_map_inline:Nn \l__file_internal_seq { \iow_log:n {##1} }
8976   \iow_log:n { ***** }
8977 }

```

(End definition for \file_list:. This function is documented on page 163.)

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.

```

8978 <*package>
8979 \AtBeginDocument
8980 {
8981   \clist_map_inline:Nn \@filelist
8982   { \seq_gput_right:No \g__file_record_seq { \tl_to_str:n {#1} } }
8983 }
8984 </package>

```

19.2 Input operations

```

8985 <@@=ior>

```

19.2.1 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.

```
8986 \cs_new_eq:NN \c_term_ior \c_sixteen
(End definition for \c_term_ior. This variable is documented on page 168.)
```

\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!

```
8987 \seq_new:N \g__ior_streams_seq
8988 \*initex
8989 \seq_gset_split:Nnn \g__ior_streams_seq { , }
8990 { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 }
8991 \*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.

```
8992 \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.

```
8993 \prop_new:N \g__ior_streams_prop
8994 \*package
8995 \prop_gput:Nnn \g__ior_streams_prop { 0 } { LaTeX2e-reserved }
8996 \*package
(End definition for \g__ior_streams_prop. This variable is documented on page ??.)
```

19.2.2 Stream management

\ior_new:N Reserving a new stream is done by defining the name as equal to using the terminal.

```
\ior_new:c 8997 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c_term_ior }
8998 \cs_generate_variant:Nn \ior_new:N { c }
(End definition for \ior_new:N and \ior_new:c. These functions are documented on page ??.)
```

\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
8999 \cs_new_protected:Npn \ior_open:Nn #1#2
9000 { \__file_name_sanitize:nn {#2} { \__ior_open_aux:Nn #1 } }
9001 \cs_generate_variant:Nn \ior_open:Nn { c }
9002 \cs_new_protected:Npn \__ior_open_aux:Nn #1#2
9003 {
9004   \file_add_path:nN {#2} \l__file_internal_name_tl
9005   \quark_if_no_value:NTF \l__file_internal_name_tl
9006   { \__msg_kernel_error:nmx { kernel } { file-not-found } {#2} }
9007   { \__ior_open:No #1 \l__file_internal_name_tl }
9008 }
```

(End definition for `\ior_open:Nn` and `\ior_open:cn`. These functions are documented on page ??.)

`\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
9009 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
9010 { \__file_name_sanitize:nn {#2} { \__ior_open_aux:NnTF #1 } }
9011 \cs_generate_variant:Nn \ior_open:NnT { c }
9012 \cs_generate_variant:Nn \ior_open:NnF { c }
9013 \cs_generate_variant:Nn \ior_open:NnTF { c }
9014 \cs_new_protected:Npn \__ior_open_aux:NnTF #1#2
9015 {
9016   \file_add_path:nN {#2} \l__file_internal_name_tl
9017   \quark_if_no_value:NTF \l__file_internal_name_tl
9018   { \prg_return_false: }
9019   {
9020     \__ior_open:No #1 \l__file_internal_name_tl
9021     \prg_return_true:
9022   }
9023 }
```

(End definition for `\ior_open:Nn` and `\ior_open:cn`. These functions are documented on page ??.)

`__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 $\text{\LaTeX} 2_{\epsilon}$ for a new stream and
use that number (after a bit of conversion).

```
9024 \cs_new_protected:Npn \__ior_open:Nn #1#2
9025 {
9026   \ior_close:N #1
9027   \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
9028   { \__ior_open_stream:Nn #1 {#2} }
9029   <*initex>
9030   { \__msg_kernel_fatal:nn { kernel } { input-streams-exhausted } }
9031   </initex>
9032   <*package>
9033   {
9034     \newread #1
9035     \tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
9036     \__ior_open_stream:Nn #1 {#2}
9037   }
9038   </package>
9039   }
9040   \cs_generate_variant:Nn \__ior_open:Nn { No }
9041   \cs_new_protected:Npn \__ior_open_stream:Nn #1#2
9042   {
9043     \tex_global:D \tex_chardef:D #1 = \l__ior_stream_tl \scan_stop:
9044     \prop_gput:NVn \g__ior_streams_prop #1 {#2}
9045     \tex_openin:D #1 #2 \scan_stop:
```

```
9046 }
(End definition for \_ior\_open:Nn and \_ior\_open:No. These functions are documented on page ??.)
```

\ior_close:N Closing a stream means getting rid of it at the T_EX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range [0,15]), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.

\ior_close:c

```
9047 \cs\_new\_protected:Npn \ior\_close:N #1
9048 {
9049   \int\_compare:nT { \c\_minus\_one < #1 < \c\_sixteen }
9050   {
9051     \tex\_closein:D #1
9052     \prop\_gremove:NV \g\_ior\_streams\_prop #1
9053     \seq\_if\_in:NVF \g\_ior\_streams\_seq #1
9054     { \seq\_gpush:NV \g\_ior\_streams\_seq #1 }
9055     \cs\_gset\_eq:NN #1 \c\_term\_ior
9056   }
9057 }
9058 \cs\_generate\_variant:Nn \ior\_close:N { c }
```

(End definition for \ior_close:N and \ior_close:c. These functions are documented on page ??.)

\ior_list_streams: Show the property lists, but with some “pretty printing”. See the l3msg module. If there are no open read streams, issue the message `show-no-stream`, and show an empty token list. If there are open read streams, format them with `_msg_show_item_unbraced:nn`, and with the message `show-open-streams`.

_ior_list_streams:Nn

```
9059 \cs\_new\_protected\_nopar:Npn \ior\_list\_streams:
9060 { \_ior\_list\_streams:Nn \g\_ior\_streams\_prop { input } }
9061 \cs\_new\_protected:Npn \_ior\_list\_streams:Nn #1#2
9062 {
9063   \_msg\_term:nnn { LaTeX / kernel }
9064   { \prop\_if\_empty:NTF #1 { show-no-stream } { show-open-streams } }
9065   {#2}
9066   \_msg\_show\_variable:n
9067   { \prop\_map\_function:NN #1 \_msg\_show\_item\_unbraced:nn }
9068 }
```

(End definition for \ior_list_streams:. This function is documented on page 164.)

19.2.3 Reading input

\if_eof:w The primitive conditional

```
9069 \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:N \overline{TF}

```
9070 \prg\_new\_conditional:Nnn \ior\_if\_eof:N { p , T , F , TF }
9071 {
9072   \cs\_if\_exist:NTF #1
9073   {
```

```

9074         \if_int_compare:w #1 = \c_sixteen
9075         \prg_return_true:
9076     \else:
9077         \if_eof:w #1
9078         \prg_return_true:
9079     \else:
9080         \prg_return_false:
9081     \fi:
9082 \fi:
9083 }
9084 { \prg_return_true: }
9085 }

```

(End definition for `\ior_if_eof:N`. These functions are documented on page 165.)

`\ior_get:NN` And here we read from files.

```

9086 \cs_new_protected:Npn \ior_get:NN #1#2
9087 { \tex_read:D #1 to #2 }

```

(End definition for `\ior_get:NN`. This function is documented on page 164.)

`\ior_get_str:NN` Reading as strings is a more complicated wrapper, as we wish to remove the endline character.

```

9088 \cs_new_protected:Npn \ior_get_str:NN #1#2
9089 {
9090     \use:x
9091     {
9092         \int_set_eq:NN \tex_endlinechar:D \c_minus_one
9093         \exp_not:n { \etex_readline:D #1 to #2 }
9094         \int_set:Nn \tex_endlinechar:D { \int_use:N \tex_endlinechar:D }
9095     }
9096 }

```

(End definition for `\ior_get_str:NN`. This function is documented on page 165.)

`\g__file_internal_ior` Needed by the higher-level code, but cannot be created until here.

```

9097 \ior_new:N \g__file_internal_ior

```

(End definition for `\g__file_internal_ior`. This variable is documented on page ??.)

19.3 Output operations

```

9098 <@@=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.

19.3.1 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`).

```

9099 \cs_new_eq:NN \c_log_iow \c_minus_one
9100 \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 168.)

`\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!

```

9101 \seq_new:N \g__iow_streams_seq
9102 <*initex>
9103 \seq_gset_eq:NN \g__iow_streams_seq \g__ior_streams_seq
9104 </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.

```

9105 \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, but with more reserved as L^AT_EX 2_ε takes up a few here.

```

9106 \prop_new:N \g__iow_streams_prop
9107 <*package>
9108 \prop_put:Nnn \g__iow_streams_prop { 0 } { LaTeX2e-reserved }
9109 \prop_put:Nnn \g__iow_streams_prop { 1 } { LaTeX2e-reserved }
9110 \prop_put:Nnn \g__iow_streams_prop { 2 } { LaTeX2e-reserved }
9111 </package>

```

(End definition for `\g__iow_streams_prop`. This variable is documented on page ??.)

19.4 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.

```

9112 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
9113 \cs_generate_variant:Nn \iow_new:N { c }

```

(End definition for `\iow_new:N` and `\iow_new:c`. These functions are documented on page ??.)

`\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.

```

\__iow_open:Nn
\__iow_open_stream:Nn
9114 \cs_new_protected:Npn \iow_open:Nn #1#2
9115 { \__file_name_sanitize:nn {#2} { \__iow_open:Nn #1 } }
9116 \cs_generate_variant:Nn \iow_open:Nn { c }
9117 \cs_new_protected:Npn \__iow_open:Nn #1#2
9118 {
9119   \iow_close:N #1
9120   \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
9121   { \__iow_open_stream:Nn #1 {#2} }
9122 <*initex>
9123   { \__msg_kernel_fatal:nn { kernel } { output-streams-exhausted } }
9124 </initex>
9125 <*package>
9126 {
9127   \newwrite #1
9128   \tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
9129   \__iow_open_stream:Nn #1 {#2}
9130 }

```

```

9131 </package>
9132 }
9133 \cs_generate_variant:Nn \__iow_open:Nn { No }
9134 \cs_new_protected:Npn \__iow_open_stream:Nn #1#2
9135 {
9136   \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
9137   \prop_gput:NVn \g__iow_streams_prop #1 {#2}
9138   \tex_immediate:D \tex_openout:D #1 #2 \scan_stop:
9139 }

```

(End definition for \iow_open:Nn and \iow_open:cn. These functions are documented on page ??.)

\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.

```

9140 \cs_new_protected:Npn \iow_close:N #1
9141 {
9142   \int_compare:nT { \c_minus_one < #1 < \c_sixteen }
9143   {
9144     \tex_immediate:D \tex_closeout:D #1
9145     \prop_gremove:NV \g__iow_streams_prop #1
9146     \seq_if_in:NVF \g__iow_streams_seq #1
9147     { \seq_gpush:NV \g__iow_streams_seq #1 }
9148     \cs_gset_eq:NN #1 \c_term_ior
9149   }
9150 }
9151 \cs_generate_variant:Nn \iow_close:N { c }

```

(End definition for \iow_close:N and \iow_close:c. These functions are documented on page ??.)

\iow_list_streams: Done as for input, but with a copy of the auxiliary so the name is correct.

```

\__iow_list_streams:Nn
9152 \cs_new_protected_nopar:Npn \iow_list_streams:
9153 { \__iow_list_streams:Nn \g__iow_streams_prop { output } }
9154 \cs_new_eq:NN \__iow_list_streams:Nn \__ior_list_streams:Nn

```

(End definition for \iow_list_streams:. This function is documented on page ??.)

19.4.1 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
9155 \cs_new_protected:Npn \iow_shipout_x:Nn #1#2
9156 { \tex_write:D #1 {#2} }
9157 \cs_generate_variant:Nn \iow_shipout_x:Nn { Nx }

```

(End definition for \iow_shipout_x:Nn and \iow_shipout_x:Nx. These functions are documented on page ??.)

\iow_shipout:Nn With ϵ -TeX available deferred writing without expansion is easy.

```

\iow_shipout:Nx
9158 \cs_new_protected:Npn \iow_shipout:Nn #1#2
9159 { \tex_write:D #1 { \exp_not:n {#2} } }
9160 \cs_generate_variant:Nn \iow_shipout:Nn { Nx }

```

(End definition for \iow_shipout:Nn and \iow_shipout:Nx. These functions are documented on page ??.)

19.4.2 Immediate writing

\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 no output stream at all, we get an internal error. We don't use the expansion done by **\write** to get the Nx variant, because it differs in subtle ways from x-expansion, namely, macro parameter characters would not need to be doubled.

```
9161 \cs_new_protected:Npn \iow_now:Nn #1#2
9162 { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
9163 \cs_generate_variant:Nn \iow_now:Nn { Nx }
```

(End definition for \iow_now:Nn and \iow_now:Nx. These functions are documented on page ??.)

\iow_log:n Writing to the log and the terminal directly are relatively easy.

```
\iow_log:x 9164 \cs_set_protected_nopar:Npn \iow_log:x { \iow_now:Nx \c_log_iow }
\iow_term:n 9165 \cs_new_protected_nopar:Npn \iow_log:n { \iow_now:Nn \c_log_iow }
\iow_term:x 9166 \cs_set_protected_nopar:Npn \iow_term:x { \iow_now:Nx \c_term_iow }
9167 \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 ??.)

19.4.3 Special characters for writing

\iow_newline: Global variable holding the character that forces a new line when something is written to an output stream

```
9168 \cs_new_nopar:Npn \iow_newline: { ^^J }
```

(End definition for \iow_newline:. This function is documented on page 166.)

\iow_char:N Function to write any escaped char to an output stream.

```
9169 \cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(End definition for \iow_char:N. This function is documented on page 166.)

19.4.4 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.

```
9170 \int_new:N \l_iow_line_count_int
9171 \int_set:Nn \l_iow_line_count_int { 78 }
```

(End definition for \l_iow_line_count_int. This variable is documented on page 167.)

\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.

```
9172 \int_new:N \l__iow_target_count_int
```

(End definition for \l__iow_target_count_int.)

<code>\l__iow_current_line_int</code> <code>\l__iow_current_word_int</code> <code>\l__iow_current_indentation_int</code>	<p>These store the number of characters in the line and word currently being constructed, and the current indentation, respectively.</p> <pre> 9173 \int_new:N \l__iow_current_line_int 9174 \int_new:N \l__iow_current_word_int 9175 \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.) </pre>
<code>\l__iow_current_line_tl</code> <code>\l__iow_current_word_tl</code> <code>\l__iow_current_indentation_tl</code>	<p>These hold the current line of text and current word, and a number of spaces for indentation, respectively.</p> <pre> 9176 \tl_new:N \l__iow_current_line_tl 9177 \tl_new:N \l__iow_current_word_tl 9178 \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.) </pre>
<code>\l__iow_wrap_tl</code>	<p>Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.</p> <pre> 9179 \tl_new:N \l__iow_wrap_tl (End definition for \l__iow_wrap_tl.) </pre>
<code>\l__iow_newline_tl</code>	<p>The token list inserted to produce the new line, with the <i><run-on text></i>.</p> <pre> 9180 \tl_new:N \l__iow_newline_tl (End definition for \l__iow_newline_tl.) </pre>
<code>\l__iow_line_start_bool</code>	<p>Boolean to avoid adding a space at the beginning of forced newlines, and to know when to add the indentation.</p> <pre> 9181 \bool_new:N \l__iow_line_start_bool (End definition for \l__iow_line_start_bool.) </pre>
<code>\c_catcode_other_space_tl</code>	<p>Lowercase a character with category code 12 to produce an “other” space. We can do everything within the group, because <code>\tl_const:Nn</code> defines its argument globally.</p> <pre> 9182 \group_begin: 9183 \char_set_catcode_other:N * 9184 \char_set_lccode:nn {'*} {'\ } 9185 \tl_to_lowercase:n { \tl_const:Nn \c_catcode_other_space_tl { * } } 9186 \group_end: (End definition for \c_catcode_other_space_tl. This function is documented on page 168.) </pre>
<code>\c__iow_wrap_marker_tl</code> <code>\c__iow_wrap_end_marker_tl</code> <code>\c__iow_wrap_newline_marker_tl</code> <code>\c__iow_wrap_indent_marker_tl</code> <code>\c__iow_wrap_unindent_marker_tl</code>	<p>Every special action of the wrapping code is preceded by the same recognizable string, <code>\c__iow_wrap_marker_tl</code>. Upon seeing that “word”, the wrapping code reads one space-delimited argument to know what operation to perform. The setting of <code>\escapechar</code> here is not very important, but makes <code>\c__iow_wrap_marker_tl</code> look nicer.</p> <pre> 9187 \group_begin: 9188 \int_set_eq:NN \tex_escapechar:D \c_minus_one 9189 \tl_const:Nx \c__iow_wrap_marker_tl 9190 { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } } 9191 \group_end: 9192 \tl_map_inline:nn </pre>

```

9193 { { end } { newline } { indent } { unindent } }
9194 {
9195   \tl_const:cx { c__iow_wrap_ #1 _marker_tl }
9196   {
9197     \c_catcode_other_space_tl
9198     \c__iow_wrap_marker_tl
9199     \c_catcode_other_space_tl
9200     #1
9201     \c_catcode_other_space_tl
9202   }
9203 }

```

(End definition for `\c__iow_wrap_marker_tl`. This function is documented on page 168.)

`\iow_indent:n` We give a dummy (protected) definition to `\iow_indent:n` when outside messages.
`__iow_indent:n` 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.

```

9204 \cs_new_protected:Npn \iow_indent:n #1 { }
9205 \cs_new:Npx \__iow_indent:n #1
9206 {
9207   \c__iow_wrap_indent_marker_tl
9208   #1
9209   \c__iow_wrap_unindent_marker_tl
9210 }

```

(End definition for `\iow_indent:n`. This function is documented on page 167.)

`\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.

```

9211 \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
9212 {
9213   \group_begin:
9214   \int_set_eq:NN \tex_escapechar:D \c_minus_one
9215   \cs_set_nopar:Npx \{ { \token_to_str:N \{ }
9216   \cs_set_nopar:Npx \# { \token_to_str:N \# }
9217   \cs_set_nopar:Npx \} { \token_to_str:N \} }
9218   \cs_set_nopar:Npx \% { \token_to_str:N \% }
9219   \cs_set_nopar:Npx \~ { \token_to_str:N \~ }
9220   \int_set:Nn \tex_escapechar:D { 92 }
9221   \cs_set_eq:NN \ \ \c__iow_wrap_newline_marker_tl

```

```

9222 \cs_set_eq:NN \ \c_catcode_other_space_tl
9223 \cs_set_eq:NN \iow_indent:n \__iow_indent:n
9224 #3
9225 <*initex>
9226 \tl_set:Nx \l__iow_wrap_tl {#1}
9227 </initex>
9228 <*package>
9229 \protected@edef \l__iow_wrap_tl {#1}
9230 </package>

```

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.

```

9231 \tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
9232 \tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
9233 \tl_replace_all:Nnn \l__iow_newline_tl { ~ } { \c_space_tl }
9234 \int_set:Nn \l__iow_target_count_int
9235 { \l__iow_line_count_int - \tl_count:N \l__iow_newline_tl + \c_one }
9236 \int_zero:N \l__iow_current_indentation_int
9237 \tl_clear:N \l__iow_current_indentation_tl
9238 \int_zero:N \l__iow_current_line_int
9239 \tl_clear:N \l__iow_current_line_tl
9240 \bool_set_true:N \l__iow_line_start_bool
9241 \use:x
9242 {
9243 \exp_not:n { \tl_clear:N \l__iow_wrap_tl }
9244 \__iow_wrap_loop:w
9245 \tl_to_str:N \l__iow_wrap_tl
9246 \tl_to_str:N \c__iow_wrap_end_marker_tl
9247 \c_space_tl \c_space_tl
9248 \exp_not:N \q_stop
9249 }
9250 \exp_args:NNo \group_end:
9251 #4 \l__iow_wrap_tl
9252 }

```

(End definition for `\iow_wrap:nnnN`. This function is documented on page 167.)

`__iow_wrap_loop:w` The loop grabs one word in the input, and checks whether it is the special marker, or a normal word.

```

9253 \cs_new_protected:Npn \__iow_wrap_loop:w #1 ~ %
9254 {
9255 \tl_set:Nn \l__iow_current_word_tl {#1}
9256 \tl_if_eq:NNTF \l__iow_current_word_tl \c__iow_wrap_marker_tl
9257 { \__iow_wrap_special:w }
9258 { \__iow_wrap_word: }
9259 }

```

(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, `__iow_wrap_word_fits:`
`__iow_wrap_word_newline:`

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.

```

9260 \cs_new_protected_nopar:Npn \__iow_wrap_word:
9261 {
9262   \int_set:Nn \l__iow_current_word_int
9263     { \__str_count_ignore_spaces:N \l__iow_current_word_tl }
9264   \int_add:Nn \l__iow_current_line_int { \l__iow_current_word_int }
9265   \int_compare:nNnTF \l__iow_current_line_int < \l__iow_target_count_int
9266     { \__iow_wrap_word_fits: }
9267     { \__iow_wrap_word_newline: }
9268   \__iow_wrap_loop:w
9269 }
9270 \cs_new_protected_nopar:Npn \__iow_wrap_word_fits:
9271 {
9272   \bool_if:NTF \l__iow_line_start_bool
9273     {
9274       \bool_set_false:N \l__iow_line_start_bool
9275       \tl_put_right:Nx \l__iow_current_line_tl
9276         { \l__iow_current_indentation_tl \l__iow_current_word_tl }
9277       \int_add:Nn \l__iow_current_line_int
9278         { \l__iow_current_indentation_int }
9279     }
9280     {
9281       \tl_put_right:Nx \l__iow_current_line_tl
9282         { ~ \l__iow_current_word_tl }
9283       \int_incr:N \l__iow_current_line_int
9284     }
9285 }
9286 \cs_new_protected_nopar:Npn \__iow_wrap_word_newline:
9287 {
9288   \tl_put_right:Nx \l__iow_wrap_tl
9289     { \l__iow_current_line_tl \l__iow_newline_tl }
9290   \int_set:Nn \l__iow_current_line_int
9291     {
9292       \l__iow_current_word_int
9293       + \l__iow_current_indentation_int
9294     }
9295   \tl_set:Nx \l__iow_current_line_tl
9296     { \l__iow_current_indentation_tl \l__iow_current_word_tl }
9297 }

```

(End definition for __iow_wrap_word:. This function is documented on page 167.)

<pre> __iow_wrap_special:w __iow_wrap_newline:w __iow_wrap_indent:w __iow_wrap_unindent:w __iow_wrap_end:w </pre>	<p>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.</p>
--	---

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.

```

9298 \cs_new_protected:Npn \__iow_wrap_special:w #1 ~ #2 ~ #3 ~ %
9299 {
9300   \use:c { __iow_wrap_#1: }
9301   \str_if_eq_x:nnTF { #2~#3 } { ~ \c__iow_wrap_marker_tl }
9302   { \__iow_wrap_special:w }
9303   { \__iow_wrap_loop:w #2 ~ #3 ~ }
9304 }
9305 \cs_new_protected_nopar:Npn \__iow_wrap_newline:
9306 {
9307   \tl_put_right:Nx \l__iow_wrap_tl
9308   { \l__iow_current_line_tl \l__iow_newline_tl }
9309   \int_zero:N \l__iow_current_line_int
9310   \tl_clear:N \l__iow_current_line_tl
9311   \bool_set_true:N \l__iow_line_start_bool
9312 }
9313 \cs_new_protected_nopar:Npx \__iow_wrap_indent:
9314 {
9315   \int_add:Nn \l__iow_current_indentation_int \c_four
9316   \tl_put_right:Nx \exp_not:N \l__iow_current_indentation_tl
9317   { \c_space_tl \c_space_tl \c_space_tl \c_space_tl }
9318 }
9319 \cs_new_protected_nopar:Npn \__iow_wrap_unindent:
9320 {
9321   \int_sub:Nn \l__iow_current_indentation_int \c_four
9322   \tl_set:Nx \l__iow_current_indentation_tl
9323   { \prg_replicate:nn \l__iow_current_indentation_int { ~ } }
9324 }
9325 \cs_new_protected_nopar:Npn \__iow_wrap_end:
9326 {
9327   \tl_put_right:Nx \l__iow_wrap_tl
9328   { \l__iow_current_line_tl }
9329   \use_none_delimit_by_q_stop:w
9330 }

```

(End definition for `__iow_wrap_special:w`. This function is documented on page 167.)

```

\__str_count_ignore_spaces:N
\__str_count_ignore_spaces:n
\__str_count_loop:NNNNNNNN

```

The wrapping code requires to measure the number of character in each word. This could be done with `\tl_count:n`, but it is ten times faster (literally) to use the code below.

```

9331 \cs_new_nopar:Npn \__str_count_ignore_spaces:N
9332 { \exp_args:No \__str_count_ignore_spaces:n }
9333 \cs_new:Npn \__str_count_ignore_spaces:n #1
9334 {
9335   \__int_value:w \__int_eval:w
9336   \exp_after:wN \__str_count_loop:NNNNNNNN \tl_to_str:n {#1}
9337   { X8 } { X7 } { X6 } { X5 } { X4 } { X3 } { X2 } { X1 } { X0 } \q_stop
9338   \__int_eval_end:
9339 }
9340 \cs_new:Npn \__str_count_loop:NNNNNNNN #1#2#3#4#5#6#7#8#9

```

```

9341 {
9342   \if_catcode:w X #9
9343   \exp_after:wN \use_none_delimit_by_q_stop:w
9344   \else:
9345     9 +
9346     \exp_after:wN \__str_count_loop:NNNNNNNNN
9347   \fi:
9348 }

```

(End definition for `__str_count_ignore_spaces:N`. This function is documented on page 167.)

19.5 Messages

```

9349 \__msg_kernel_new:nnnn { kernel } { file-not-found }
9350 { File~'#1'~not-found. }
9351 {
9352   The~requested~file~could~not~be~found~in~the~current~directory,~
9353   in~the~TeX~search~path~or~in~the~LaTeX~search~path.
9354 }
9355 \__msg_kernel_new:nnnn { kernel } { input-streams-exhausted }
9356 { Input~streams~exhausted }
9357 {
9358   TeX~can~only~open~up~to~16~input~streams~at~one~time.\\
9359   All~16~are~currently~in~use,~and~something~wanted~to~open~
9360   another~one.
9361 }
9362 \__msg_kernel_new:nnnn { kernel } { output-streams-exhausted }
9363 { Output~streams~exhausted }
9364 {
9365   TeX~can~only~open~up~to~16~output~streams~at~one~time.\\
9366   All~16~are~currently~in~use,~and~something~wanted~to~open~
9367   another~one.
9368 }
9369 \__msg_kernel_new:nnnn { kernel } { space-in-file-name }
9370 { Space~in~file~name~'#1'. }
9371 {
9372   Spaces~are~not~permitted~in~files~loaded~by~LaTeX: \\
9373   Further~errors~may~follow!
9374 }
9375 </initex | package>

```

20 l3fp implementation

```

9376 <*package>
9377 \ProvidesExplPackage
9378   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
9379   \__expl_package_check:
9380 </package>

```

21 l3fp-aux implementation

```

9381 <*initex | package>
9382 <@@=fp>

```

22 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:⁶

- 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

```
\s__fp \__fp_chk:w 1 <sign> {<exponent>} {<X1>} {<X2>} {<X3>} {<X4>} ;
```

⁶Bruno: 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 {\langle exponent \rangle} {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} ;	Positive floating point.
1 2 {\langle exponent \rangle} {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} ;	Negative floating point.
2 0 \s_fp_... ;	Positive infinity.
2 2 \s_fp_... ;	Negative infinity.
3 1 \s_fp_... ;	Quiet nan.
3 1 \s_fp_... ;	Signalling nan.

Here, the $\langle exponent \rangle$ 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 $\langle exponent \rangle$ is minimal. This implies $1000 \leq \langle X_1 \rangle \leq 9999$.

23 Internal storage of floating points numbers

A floating point number $\langle X \rangle$ is stored as

`\s_fp _fp_chk:w \langle case \rangle \langle sign \rangle \langle body \rangle ;`

Here, $\langle case \rangle$ is 0 for ± 0 , 1 for normal numbers, 2 for $\pm \infty$, and 3 for `nan`, and $\langle sign \rangle$ is 0 for positive numbers, 1 for `nans`, and 2 for negative numbers. The $\langle body \rangle$ of normal numbers is $\{\langle exponent \rangle\} \{\langle X_1 \rangle\} \{\langle X_2 \rangle\} \{\langle X_3 \rangle\} \{\langle X_4 \rangle\}$, 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 $\langle exponent \rangle$ 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.

23.1 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.

9383 `\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`

```

9384 \cs_new:Npn \__fp_use_s:n #1 { #1; }
9385 \cs_new:Npn \__fp_use_braced_s:n #1 { {#1} ; }
9386 \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`

```

9387 \cs_new:Npn \__fp_use_none_until_s:w #1; { }
9388 \cs_new:Npn \__fp_use_i_until_s:nw #1#2; {#1}
9389 \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.

```

9390 \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.

```

9391 \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 such argument.

```

9392 \cs_new:Npn \__fp_use_i:ww #1; #2; { #1; }

```

(End definition for `__fp_use_i:ww`.)

23.2 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.

```

9393 \__scan_new:N \s__fp
9394 \cs_new_protected:Npn \__fp_chk:w #1 ;
9395 {
9396   \__msg_kernel_error:nnx { kernel } { misused-fp }
9397   { \fp_to_tl:n { \s__fp \__fp_chk:w #1 ; } }
9398 }

```

(End definition for `\s__fp` and `__fp_chk:w`.)

`\s__fp_mark` Aliases of `\tex_relax:D`, used to terminate expressions.

`\s__fp_stop`

```

9399 \__scan_new:N \s__fp_mark
9400 \__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` 9401 `__scan_new:N \s__fp_invalid`

`\s__fp_overflow` 9402 `__scan_new:N \s__fp_underflow`

`\s__fp_division` 9403 `__scan_new:N \s__fp_overflow`

`\s__fp_exact` 9404 `__scan_new:N \s__fp_division`

9405 `__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`

`\c_nan_fp`

where the dots in `\s__fp...` are one of `invalid`, `underflow`, `overflow`, `division`, `exact`, describing how the floating point was created. We define the floating points here as “exact”.

9406 `\tl_const:Nn \c_zero_fp { \s__fp __fp_chk:w 0 0 \s__fp_exact ; }`

9407 `\tl_const:Nn \c_minus_zero_fp { \s__fp __fp_chk:w 0 2 \s__fp_exact ; }`

9408 `\tl_const:Nn \c_inf_fp { \s__fp __fp_chk:w 2 0 \s__fp_exact ; }`

9409 `\tl_const:Nn \c_minus_inf_fp { \s__fp __fp_chk:w 2 2 \s__fp_exact ; }`

9410 `\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 ??.)

`__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.

9411 `\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` 9412 `\cs_new:Npn __fp_zero_fp:N #1 { \s__fp __fp_chk:w 0 #1 \s__fp_underflow ; }`

9413 `\cs_new:Npn __fp_inf_fp:N #1 { \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` In some cases, we need to output the smallest or biggest positive or negative finite numbers.

`__fp_min_fp:N`

9414 `\cs_new:Npn __fp_min_fp:N #1`

9415 `{`

9416 `\s__fp __fp_chk:w 1 #1`

9417 `{ \int_eval:n { - \c__fp_max_exponent_int } }`

9418 `{1000} {0000} {0000} {0000} ;`

9419 `}`

9420 `\cs_new:Npn __fp_max_fp:N #1`

9421 `{`

```

9422     \s__fp \__fp_chk:w 1 #1
9423     { \int_use:N \c__fp_max_exponent_int }
9424     {9999} {9999} {9999} {9999} ;
9425 }
(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.

```

9426 \cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
9427 {
9428     \if_meaning:w 1 #1
9429     \exp_after:wN \__fp_use_ii_until_s:nw
9430 \else:
9431     \exp_after:wN \__fp_use_i_until_s:nw
9432     \exp_after:wN 0
9433 \fi:
9434 }
(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.

```

9435 \cs_new:Npn \__fp_neg_sign:N #1
9436 { \__int_eval:w \c_two - #1 \__int_eval_end: }
(End definition for \__fp_neg_sign:N.)

```

23.3 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`.

```

9437 \cs_new:Npn \__fp_sanitize:Nw #1 #2;
9438 {
9439     \if_case:w \if_int_compare:w #2 > \c__fp_max_exponent_int \c_one \else:
9440         \if_int_compare:w #2 < - \c__fp_max_exponent_int \c_two \else:
9441         \if_meaning:w 1 #1 \c_three \else: \c_zero \fi: \fi: \fi:
9442     \or: \exp_after:wN \__fp_overflow:w
9443     \or: \exp_after:wN \__fp_underflow:w
9444     \or: \exp_after:wN \__fp_sanitize_zero:w
9445     \fi:
9446     \s__fp \__fp_chk:w 1 #1 {#2}
9447 }
9448 \cs_new:Npn \__fp_sanitize:wN #1; #2 { \__fp_sanitize:Nw #2 #1; }
9449 \cs_new:Npn \__fp_sanitize_zero:w \s__fp \__fp_chk:w #1 #2 #3; { \c_zero_fp }
(End definition for \__fp_sanitize:Nw and \__fp_sanitize:wN. These functions are documented on
page ??.)

```

23.4 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.

```

9450 \cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
9451 {
9452   \if_meaning:w 1 #1
9453     \exp_after:wN \__fp_exp_after_normal:nNNw
9454   \else:
9455     \exp_after:wN \__fp_exp_after_special:nNNw
9456   \fi:
9457   { }
9458   #1
9459 }
9460 \cs_new:Npn \__fp_exp_after_o:nw #1 \s__fp \__fp_chk:w #2
9461 {
9462   \if_meaning:w 1 #2
9463     \exp_after:wN \__fp_exp_after_normal:nNNw
9464   \else:
9465     \exp_after:wN \__fp_exp_after_special:nNNw
9466   \fi:
9467   { #1 }
9468   #2
9469 }
9470 \cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
9471 {
9472   \if_meaning:w 1 #2
9473     \exp_after:wN \__fp_exp_after_normal:nNNw
9474   \else:
9475     \exp_after:wN \__fp_exp_after_special:nNNw
9476   \fi:
9477   { \tex_romannumeral:D -'0 #1 }
9478   #2
9479 }
```

(End definition for `__fp_exp_after_o:w`. This function is documented on page ??.)

`__fp_exp_after_special:nNNw` Special floating point numbers are easy to jump over since they contain few tokens.

```

9480 \cs_new:Npn \__fp_exp_after_special:nNNw #1#2#3#4;
9481 {
9482   \exp_after:wN \s__fp
9483   \exp_after:wN \__fp_chk:w
9484   \exp_after:wN #2
9485   \exp_after:wN #3
9486   \exp_after:wN #4
9487   \exp_after:wN ;
9488   #1
```

```

9489 }
(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.

```

9490 \cs_new:Npn \__fp_exp_after_normal:nNNw #1 1 #2 #3 #4#5#6#7;
9491 {
9492   \exp_after:wN \__fp_exp_after_normal:Nwwwww
9493   \exp_after:wN #2
9494   \__int_value:w #3   \exp_after:wN ;
9495   \__int_value:w 1 #4 \exp_after:wN ;
9496   \__int_value:w 1 #5 \exp_after:wN ;
9497   \__int_value:w 1 #6 \exp_after:wN ;
9498   \__int_value:w 1 #7 \exp_after:wN ; #1
9499 }
9500 \cs_new:Npn \__fp_exp_after_normal:Nwwwww
9501   #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
9502   { \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`

```

9503 \cs_new:Npn \__fp_exp_after_array_f:w #1
9504 {
9505   \cs:w \__fp_exp_after \__fp_type_from_scan:N #1 _f:nw \cs_end:
9506   { \__fp_exp_after_array_f:w }
9507   #1
9508 }
9509 \cs_new_eq:NN \__fp_exp_after_stop_f:nw \use_none:nn
(End definition for \__fp_exp_after_array_f:w. This function is documented on page ??.)

```

23.5 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 $\text{T}_{\text{E}}\text{X}$'s integers). This method is very heavily relied upon in `l3fp-basics`.

More specifically, the auxiliary inserts `+ #1#2#3#4#5 ; {#6}`, which allows us to compute several blocks of 4 digits in a nested manner, performing carries on the fly. Say we want to compute $1\,2345 \times 6677\,8899$. With simplified names, we would do

```

\exp_after:wN \post_processing:w
\int_use:N \__int_eval:w - 5 0000
\exp_after:wN \pack:NNNNNw
\int_use:N \__int_eval:w 4 9995 0000
+ 12345 * 6677
\exp_after:wN \pack:NNNNNw
\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:NNNNNw
\__fp_pack:NNNNNwn
\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. The `__fp_pack:NNNNNwn` function brings a braced *<continuation>* up through the levels of expansion.

```

9510 \int_const:Nn \c__fp_leading_shift_int { - 5 0000 }
9511 \int_const:Nn \c__fp_middle_shift_int { 5 0000 * 9999 }
9512 \int_const:Nn \c__fp_trailing_shift_int { 5 0000 * 10000 }
9513 \cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }
9514 \cs_new:Npn \__fp_pack:NNNNNwn #1 #2#3#4#5 #6; #7
9515 { + #1#2#3#4#5 ; {#7} {#6} }

```

(End definition for `__fp_pack:NNNNNw` and `__fp_pack:NNNNNwn`. These functions are documented on page ??.)

```

\__fp_pack_big:NNNNNNw
\__fp_pack_big:NNNNNNwn
\c__fp_big_trailing_shift_int
\c__fp_big_middle_shift_int
\c__fp_big_leading_shift_int

```

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.

```

9516 \int_const:Nn \c__fp_big_leading_shift_int { - 15 2374 }
9517 \int_const:Nn \c__fp_big_middle_shift_int { 15 2374 * 9999 }

```

```

9518 \int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 }
9519 \cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
9520 { + #1#2#3#4#5#6 ; {#7} }
9521 \cs_new:Npn \__fp_pack_big:NNNNNNwn #1#2 #3#4#5#6 #7; #8
9522 { + #1#2#3#4#5#6 ; {#8} {#7} }

```

(End definition for `__fp_pack_big:NNNNNNw` and `__fp_pack_big:NNNNNNwn`. These functions are documented on page ??.)

```

\__fp_pack_Bigg:NNNNNNw
\c__fp_Bigg_trailing_shift_int
\c__fp_Bigg_middle_shift_int
\c__fp_Bigg_leading_shift_int

```

This set of shifts allows for computations involving 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.

```

9523 \int_const:Nn \c__fp_Bigg_leading_shift_int { - 20 0000 }
9524 \int_const:Nn \c__fp_Bigg_middle_shift_int { 20 0000 * 9999 }
9525 \int_const:Nn \c__fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
9526 \cs_new:Npn \__fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
9527 { + #1#2#3#4#5#6 ; {#7} }

```

(End definition for `__fp_pack_Bigg:NNNNNNw`. This function is documented on page ??.)

```
\_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.

```

9528 \cs_new:Npn \__fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
9529 { #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.

```

9530 \cs_new:Npn \__fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
9531 { #1 {#2#3#4#5#6#7#8#9} ; }

```

(End definition for `__fp_pack_eight:wNNNNNNNN`.)

23.6 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 $\langle \textit{rounding} \rangle$ 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 $\langle \textit{shift} \rangle$.

```

9532 \cs_new:Npn \__fp_decimate:nNnnnn #1
9533 {
9534   \cs:w
9535     __fp_decimate_
9536     \if_int_compare:w \__int_eval:w #1 > \c_sixteen
9537       tiny
9538     \else:
9539       \tex_romannumeral:D \__int_eval:w #1
9540     \fi:
9541     :Nnnnn
9542   \cs_end:
9543 }
```

Each of the auxiliaries see the function $\langle f_1 \rangle$, followed by 4 blocks of 4 digits.
(End definition for $\backslash_fp_decimate:nNnnnn$.)

```

\__fp_decimate_:Nnnnn
\__fp_decimate_tiny:Nnnnn
```

If the $\langle \textit{shift} \rangle$ is zero, or too big, life is very easy.

```

9544 \cs_new:Npn \__fp_decimate_:Nnnnn #1 #2#3#4#5
9545   { #1 0 {#2#3} {#4#5} ; }
9546 \cs_new:Npn \__fp_decimate_tiny:Nnnnn #1 #2#3#4#5
9547   { #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 $\langle \textit{rounding} \rangle$ digit, and repack digits into two blocks of 8. The sixteen functions are very similar, and defined through $\backslash_fp_tmp:w$. 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 $\backslash_fp_round_digit:Nw$ to the $\langle \textit{rounding} \rangle$ digit. This triggers the f-expansion of $\backslash_fp_decimate_pack:nnnnnnnnnw$,⁷ responsible for building two blocks of 8 digits, and removing the rest. For this to work, #3 alternates between braced and unbraced blocks of 4 digits, in such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.

```

9548 \cs_new:Npn \__fp_tmp:w #1 #2 #3
9549 {
9550   \cs_new:cpn { __fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
9551   {
9552     \exp_after:wN ##1
9553     \__int_value:w
9554     \exp_after:wN \__fp_round_digit:Nw #2 ;
9555     \__fp_decimate_pack:nnnnnnnnnw #3 ;
9556   }
9557 }
9558 \__fp_tmp:w {i} {\use_none:nnn #50} { 0{#2}#3{#4}#5 }
9559 \__fp_tmp:w {iii} {\use_none:nn #5 } { 00{#2}#3{#4}#5 }
9560 \__fp_tmp:w {iiii} {\use_none:n #5 } { 000{#2}#3{#4}#5 }
```

⁷No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.


```

9561 \_fp_tmp:w {iv} { #5 } { {0000}#2{#3}#4 #5 }
9562 \_fp_tmp:w {v} {\use\_none:nnn #4#5 } { 0{0000}#2{#3}#4 #5 }
9563 \_fp_tmp:w {vi} {\use\_none:nn #4#5 } { 00{0000}#2{#3}#4 #5 }
9564 \_fp_tmp:w {vii} {\use\_none:n #4#5 } { 000{0000}#2{#3}#4 #5 }
9565 \_fp_tmp:w {viii}{ #4#5 } { {0000}0000{#2}#3 #4 #5 }
9566 \_fp_tmp:w {ix} {\use\_none:nnn #3#4+#5 } { 0{0000}0000{#2}#3 #4 #5 }
9567 \_fp_tmp:w {x} {\use\_none:nn #3#4+#5 } { 00{0000}0000{#2}#3 #4 #5 }
9568 \_fp_tmp:w {xi} {\use\_none:n #3#4+#5 } { 000{0000}0000{#2}#3 #4 #5 }
9569 \_fp_tmp:w {xii} { #3#4+#5 } { {0000}0000{0000}#2 #3 #4 #5 }
9570 \_fp_tmp:w {xiii}{\use\_none:nnn#2#3+#4#5} { 0{0000}0000{0000}#2 #3 #4 #5 }
9571 \_fp_tmp:w {xiv} {\use\_none:nn #2#3+#4#5} { 00{0000}0000{0000}#2 #3 #4 #5 }
9572 \_fp_tmp:w {xv} {\use\_none:n #2#3+#4#5} { 000{0000}0000{0000}#2 #3 #4 #5 }
9573 \_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` 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.

```

9574 \cs_new:Npn \_fp_decimate_pack:nnnnnnnnnw #1#2#3#4#5
9575 { \_fp_decimate_pack:nnnnnnnw { #1#2#3#4#5 } }
9576 \cs_new:Npn \_fp_decimate_pack:nnnnnnnw #1 #2#3#4#5#6
9577 { {#1} {#2#3#4#5#6} }
(End definition for \_fp_round_digit:Nw and \_fp_decimate_pack:nnnnnnnnnw.)

```

23.7 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 *fp var*, expanding once after that floating point. Case 1 will do *some computation* using the *floating point* (presumably compute the operation requested by the user in that non-trivial case). Case 2 will return the *floating point* without modifying it, removing the *junk* and expanding once after. Case 3 will close the conditional, remove the *junk*

and the *floating point*, and expand *something* next. In other cases, the “*junk*” is expanded, performing some other operation on the *floating point*. We provide similar functions with two trailing *floating points*.

`__fp_case_use:nw` This function ends a TeX conditional, removes junk until the next floating point, and places its first argument before that floating point, to perform some operation on the floating point.

```
9578 \cs_new:Npn \__fp_case_use:nw #1#2 \fi: #3 \s__fp { \fi: #1 \s__fp }
(End definition for \__fp_case_use:nw.)
```

`__fp_case_return:nw` This function ends a TeX conditional, removes junk and a floating point, and places its first argument in the input stream. A quirk is that we don’t define this function requiring a floating point to follow, simply anything ending in a semicolon. This, in turn, means that the *junk* may not contain semicolons.

```
9579 \cs_new:Npn \__fp_case_return:nw #1#2 \fi: #3 ; { \fi: #1 }
(End definition for \__fp_case_return:nw.)
```

`__fp_case_return_o:Nw` This function ends a TeX conditional, removes junk and a floating point, and returns its first argument (an *fp var*) then expands once after it.

```
9580 \cs_new:Npn \__fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 ;
9581 { \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.

```
9582 \cs_new:Npn \__fp_case_return_same_o:w #1 \fi: #2 \s__fp
9583 { \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.

```
9584 \cs_new:Npn \__fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 ; #5 ;
9585 { \fi: \exp_after:wN #1 }
(End definition for \__fp_case_return_o:Nww.)
```

`__fp_case_return_i_o:ww` Similar to `__fp_case_return_same_o:w`, but this returns the first or second of two trailing floating point numbers, expanding once after the result.

`__fp_case_return_ii_o:ww`

```
9586 \cs_new:Npn \__fp_case_return_i_o:ww #1 \fi: #2 \s__fp #3 ; \s__fp #4 ;
9587 { \fi: \__fp_exp_after_o:w \s__fp #3 ; }
9588 \cs_new:Npn \__fp_case_return_ii_o:ww #1 \fi: #2 \s__fp #3 ;
9589 { \fi: \__fp_exp_after_o:w }
(End definition for \__fp_case_return_i_o:ww and \__fp_case_return_ii_o:ww.)
```

23.8 Small integer floating points

```

    \_fp_small_int:wTF
    \_fp_small_int_true:wTF
    \_fp_small_int_normal:NnwTF
    \_fp_small_int_test:NnnwNTF

```

This function tests if its floating point argument is an integer in the range $[-99999999, 99999999]$. If it is, the result of the conversion is fed as a braced argument to the *⟨true code⟩*. Otherwise, the *⟨false code⟩* is performed. First filter special cases: neither `nan` nor infinities are integers. 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.

```

9590 \cs_new:Npn \_fp_small_int:wTF \s_fp \_fp_chk:w #1
9591 {
9592   \if_case:w #1 \exp_stop_f:
9593     \_fp_case_return:nw { \_fp_small_int_true:wTF 0 ; }
9594   \or:   \exp_after:wN \_fp_small_int_normal:NnwTF
9595   \else: \_fp_case_return:nw \use_ii:nn
9596   \fi:
9597 }
9598 \cs_new:Npn \_fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
9599 \cs_new:Npn \_fp_small_int_normal:NnwTF #1#2#3;
9600 {
9601   \if_int_compare:w #2 > \c_zero
9602     \if_int_compare:w #2 > \c_eight
9603       \exp_after:wN \exp_after:wN
9604       \exp_after:wN \use_iii:nnn
9605     \else:
9606       \_fp_decimate:nNnnnn { \c_sixteen - #2 }
9607       \_fp_small_int_test:NnnwNTF
9608       #3 #1
9609     \fi:
9610   \else:
9611     \exp_after:wN \use_iii:nnn
9612   \fi:
9613   ;
9614 }
9615 \cs_new:Npn \_fp_small_int_test:NnnwNTF #1#2#3#4; #5
9616 {
9617   \if_meaning:w 0 #1
9618     \exp_after:wN \_fp_small_int_true:wTF
9619     \_int_value:w \if_meaning:w 2 #5 - \fi: #3
9620   \else:
9621     \exp_after:wN \use_i:nn
9622   \fi:
9623 }

```

(End definition for `_fp_small_int:wTF`. This function is documented on page ??.)

23.9 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.

```

9624 \cs_new:Npn \__fp_array_count:n #1
9625 {
9626   \int_use:N \__int_eval:w \c_zero
9627   \__fp_array_count_loop:Nw #1 { ? \__prg_break: } ;
9628   \__prg_break_point:
9629   \__int_eval_end:
9630 }
9631 \cs_new:Npn \__fp_array_count_loop:Nw #1#2;
9632 { \use_none:n #1 + \c_one \__fp_array_count_loop:Nw }

```

(End definition for `__fp_array_count:n`. This function is documented on page ??.)

23.10 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.

```

9633 \cs_new:Npn \__fp_expand:n #1
9634 {
9635   \__fp_expand_loop:nwnN { }
9636   #1 \prg_do_nothing:
9637   \s__fp_mark { } \__fp_expand_loop:nwnN
9638   \s__fp_mark { } \__fp_use_i_until_s:nw ;
9639 }
9640 \cs_new:Npn \__fp_expand_loop:nwnN #1#2 \s__fp_mark #3 #4
9641 {
9642   \exp_after:wN #4 \tex_romannumeral:D -'0
9643   #2
9644   \s__fp_mark { #3 #1 } #4
9645 }

```

(End definition for `__fp_expand:n`. This function is documented on page ??.)

23.11 Messages

Using a floating point directly is an error.

```

9646 \__msg_kernel_new:nnnn { kernel } { misused-fp }
9647 { A-floating-point-with-value-'#1'-was-misused. }
9648 {
9649   To-obtain-the-value-of-a-floating-point-variable,~use~
9650   '\token_to_str:N \fp_to_decimal:N',~

```

```

9651     '\token_to_str:N \fp_to_scientific:N',~or~other~
9652     conversion~functions.
9653   }
9654 </initex | package>

```

24 l3fp-traps Implementation

```

9655 <*initex | package>
9656 <@@=fp>

```

Exceptions should be accessed by an `n`-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

24.1 Flags

`\fp_flag_off:n` Function to turn a flag off. Simply undefine it.

```

9657 \cs_new_protected:Npn \fp_flag_off:n #1
9658 { \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 177.)

`\fp_flag_on:n` Function to turn a flag on expandably: use TeX's automatic assignment to `\scan_stop:`.

```

9659 \cs_new:Npn \fp_flag_on:n #1
9660 { \exp_args:Nc \use_none:n { l__fp_ #1 _flag_token } }

```

(End definition for `\fp_flag_on:n`. This function is documented on page 177.)

`\fp_if_flag_on_p:n` Returns true if the flag is on, false otherwise.

```

\fp_if_flag_on:nTF
9661 \prg_new_conditional:Npnn \fp_if_flag_on:n #1 { p , T , F , TF }
9662 {
9663   \if_cs_exist:w l__fp_ #1 _flag_token \cs_end:
9664     \prg_return_true:
9665   \else:
9666     \prg_return_false:
9667   \fi:
9668 }

```

(End definition for `\fp_if_flag_on:n`. These functions are documented on page 177.)

```

\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.

```

9669 \cs_new_eq:NN \l__fp_invalid_operation_flag_token \tex_undefined:D
9670 \cs_new_eq:NN \l__fp_division_by_zero_flag_token \tex_undefined:D
9671 \cs_new_eq:NN \l__fp_overflow_flag_token \tex_undefined:D
9672 \cs_new_eq:NN \l__fp_underflow_flag_token \tex_undefined:D
(End definition for \l__fp_invalid_operation_flag_token and others.)

```

24.2 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:Nnw`,
- `__fp_invalid_operation_tl_o:nf`,
- `__fp_division_by_zero_o:Nnw`,
- `__fp_division_by_zero_o:NNnw`,
- `__fp_overflow:w`,
- `__fp_underflow:w`.

Rather than changing them directly, we provide a user interface as `\fp_trap:nn {<exception>} {<way of trapping>}`, where the *<way of trapping>* is one of `error`, `flag`, or `none`.

We also provide `__fp_invalid_operation_o:nw`, defined in terms of `__fp_invalid_operation:nnw`.

`\fp_trap:nn`

```

9673 \cs_new_protected:Npn \fp_trap:nn #1#2
9674 {
9675   \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
9676   {
9677     \clist_if_in:nnTF
9678     { invalid_operation , division_by_zero , overflow , underflow }
9679     {#1}
9680     {

```

```

9681         \_msg_kernel_error:nxxx { kernel }
9682         { unknown-fpu-trap-type } {#1} {#2}
9683     }
9684     { \_msg_kernel_error:nxx { kernel } { unknown-fpu-exception } {#1} }
9685 }
9686 }

```

(End definition for `\fp_trap:nn`. This function is documented on page 177.)

`_fp_trap_invalid_operation_set_error:` We provide three types of trapping for invalid operations: either produce an error and
`_fp_trap_invalid_operation_set_flag:` raise the relevant flag; or only raise the flag; or don't even raise the flag. In most cases,
`_fp_trap_invalid_operation_set_none:` the function produces as a result its first argument, possibly with post-expansion.
`_fp_trap_invalid_operation_set:N`

```

9687 \cs_new_protected_nopar:Npn \_fp_trap_invalid_operation_set_error:
9688 { \_fp_trap_invalid_operation_set:N \prg_do_nothing: }
9689 \cs_new_protected_nopar:Npn \_fp_trap_invalid_operation_set_flag:
9690 { \_fp_trap_invalid_operation_set:N \use_none:nnnnn }
9691 \cs_new_protected_nopar:Npn \_fp_trap_invalid_operation_set_none:
9692 { \_fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
9693 \cs_new_protected:Npn \_fp_trap_invalid_operation_set:N #1
9694 {
9695     \exp_args:Nno \use:n
9696     { \cs_set:Npn \_fp_invalid_operation:nnw ##1##2##3; }
9697     {
9698         #1
9699         \_fp_error:nnfn { invalid } {##2} { \fp_to_tl:n { ##3; } } { }
9700         \fp_flag_on:n { invalid_operation }
9701         ##1
9702     }
9703     \exp_args:Nno \use:n
9704     { \cs_set:Npn \_fp_invalid_operation_o:Nww ##1##2; ##3; }
9705     {
9706         #1
9707         \_fp_error:nffn { invalid-ii }
9708         { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
9709         \fp_flag_on:n { invalid_operation }
9710         \exp_after:wN \c_nan_fp
9711     }
9712     \exp_args:Nno \use:n
9713     { \cs_set:Npn \_fp_invalid_operation_tl_o:nf ##1##2 }
9714     {
9715         #1
9716         \_fp_error:nnfn { invalid } {##1} {##2} { }
9717         \fp_flag_on:n { invalid_operation }
9718         \exp_after:wN \c_nan_fp
9719     }
9720 }

```

(End definition for `_fp_trap_invalid_operation_set_error:` and others.)

`_fp_trap_division_by_zero_set_error:` We provide three types of trapping for invalid operations and division by zero: either
`_fp_trap_division_by_zero_set_flag:` produce an error and raise the relevant flag; or only raise the flag; or don't even raise the
`_fp_trap_division_by_zero_set_none:`
`_fp_trap_division_by_zero_set:N`

flag. In all cases, the function must produce a result, namely its first argument, $\pm\infty$ or `nan`.

```

9721 \cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_error:
9722 { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
9723 \cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_flag:
9724 { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
9725 \cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_none:
9726 { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
9727 \cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
9728 {
9729   \exp_args:Nno \use:n
9730   { \cs_set:Npn \__fp_division_by_zero_o:Nnw ##1##2##3; }
9731   {
9732     #1
9733     \__fp_error:nnfn { zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
9734     \fp_flag_on:n { division_by_zero }
9735     \exp_after:wN ##1
9736   }
9737   \exp_args:Nno \use:n
9738   { \cs_set:Npn \__fp_division_by_zero_o:NNww ##1##2##3; ##4; }
9739   {
9740     #1
9741     \__fp_error:nffn { zero-div-ii }
9742     { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
9743     \fp_flag_on:n { division_by_zero }
9744     \exp_after:wN ##1
9745   }
9746 }

```

(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 \TeX , and `__fp_overflow:w` receives $\pm\infty$ (`__fp_underflow:w` would receive ± 0); then we cannot do better than simply say an overflow or underflow occurred.

```

9747 \cs_new_protected_nopar:Npn \__fp_trap_overflow_set_error:
9748 { \__fp_trap_overflow_set:N \prg_do_nothing: }
9749 \cs_new_protected_nopar:Npn \__fp_trap_overflow_set_flag:
9750 { \__fp_trap_overflow_set:N \use_none:nnnnn }
9751 \cs_new_protected_nopar:Npn \__fp_trap_overflow_set_none:
9752 { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
9753 \cs_new_protected:Npn \__fp_trap_overflow_set:N #1
9754 { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
9755 \cs_new_protected_nopar:Npn \__fp_trap_underflow_set_error:

```



```

9756 { \__fp_trap_underflow_set:N \prg_do_nothing: }
9757 \cs_new_protected_nopar:Npn \__fp_trap_underflow_set_flag:
9758 { \__fp_trap_underflow_set:N \use_none:nnnnn }
9759 \cs_new_protected_nopar:Npn \__fp_trap_underflow_set_none:
9760 { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
9761 \cs_new_protected:Npn \__fp_trap_underflow_set:N #1
9762 { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }
9763 \cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4
9764 {
9765   \exp_args:Nno \use:n
9766   { \cs_set:cpn { __fp_ #2 :w } \s__fp \__fp_chk:w ##1##2##3; }
9767   {
9768     #1
9769     \__fp_error:nffn
9770     { flow \if_meaning:w 1 ##1 -to \fi: }
9771     { \fp_to_tl:n { \s__fp \__fp_chk:w ##1##2##3; } }
9772     { \token_if_eq_meaning:NnF 0 ##2 { - } #4 }
9773     {#2}
9774     \fp_flag_on:n {#2}
9775     #3 ##2
9776   }
9777 }

```

(End definition for __fp_trap_overflow_set_error: and others. These functions are documented on page 177.)

__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_o:nf
\__fp_division_by_zero_o:Nnw
\__fp_division_by_zero_o:NNww
\__fp_overflow:w
\__fp_underflow:w
9778 \cs_new:Npn \__fp_invalid_operation:nnw #1#2#3; { }
9779 \cs_new:Npn \__fp_invalid_operation_o:Nnw #1#2; #3; { }
9780 \cs_new:Npn \__fp_invalid_operation_tl_o:nf #1 #2 { }
9781 \cs_new:Npn \__fp_division_by_zero_o:Nnw #1#2#3; { }
9782 \cs_new:Npn \__fp_division_by_zero_o:NNww #1#2#3; #4; { }
9783 \cs_new:Npn \__fp_overflow:w { }
9784 \cs_new:Npn \__fp_underflow:w { }
9785 \fp_trap:nn { invalid_operation } { error }
9786 \fp_trap:nn { division_by_zero } { flag }
9787 \fp_trap:nn { overflow } { flag }
9788 \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.

```

9789 \cs_new_nopar:Npn \__fp_invalid_operation_o:nw
9790 { \__fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } }
9791 \cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }

```

(End definition for __fp_invalid_operation_o:nw and __fp_invalid_operation_o:fw.)

24.3 Errors

```

\__fp_error:nnnn
\__fp_error:nnfn 9792 \cs_new:Npn \__fp_error:nnnn #1
\__fp_error:nffn 9793 { \__msg_kernel_expandable_error:nnnn { kernel } { fp - #1 } }
9794 \cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff }
(End definition for \__fp_error:nnnn, \__fp_error:nnfn, and \__fp_error:nffn.)

```

24.4 Messages

Some messages.

```

9795 \__msg_kernel_new:nnnn { kernel } { unknown-fpu-exception }
9796 { The~FPU~exception~'#1'~is~not~known:~that~trap~will~never~be~triggered. }
9797 {
9798   The~only~exceptions~to~which~traps~can~be~attached~are \\
9799   \iow_indent:n
9800   {
9801     * ~ invalid_operation \\
9802     * ~ division_by_zero \\
9803     * ~ overflow \\
9804     * ~ underflow
9805   }
9806 }
9807 \__msg_kernel_new:nnnn { kernel } { unknown-fpu-trap-type }
9808 { The~FPU~trap~type~'#2'~is~not~known. }
9809 {
9810   The~trap~type~must~be~one~of \\
9811   \iow_indent:n
9812   {
9813     * ~ error \\
9814     * ~ flag \\
9815     * ~ none
9816   }
9817 }
9818 \__msg_kernel_new:nnn { kernel } { fp-flow }
9819 { An ~ #3 ~ occurred. }
9820 \__msg_kernel_new:nnn { kernel } { fp-flow-to }
9821 { #1 ~ #3 ed ~ to ~ #2 . }
9822 \__msg_kernel_new:nnn { kernel } { fp-zero-div }
9823 { Division~by~zero~in~ #1 (#2) }
9824 \__msg_kernel_new:nnn { kernel } { fp-zero-div-ii }
9825 { Division~by~zero~in~ (#1) #3 (#2) }
9826 \__msg_kernel_new:nnn { kernel } { fp-invalid }
9827 { Invalid~operation~ #1 (#2) }
9828 \__msg_kernel_new:nnn { kernel } { fp-invalid-ii }
9829 { Invalid~operation~ (#1) #3 (#2) }
9830 </initex | package>

```

25 l3fp-round implementation

9831 $\langle *initex \mid package \rangle$

9832 $\langle @@=fp \rangle$

25.1 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 $\langle sign \rangle \langle digit_1 \rangle \langle digit_2 \rangle$` can expand to `\c_zero` or `\c_one`.
- `__fp_round_s:NNNw $\langle sign \rangle \langle digit_1 \rangle \langle digit_2 \rangle \langle more\ digits \rangle$` ; can expand to `\c_zero` ; or `\c_one` ;.
- `__fp_round_neg:NNN $\langle sign \rangle \langle digit_1 \rangle \langle digit_2 \rangle$` 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_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.

```

9833 \cs_new:Npn \__fp_round_return_one:
9834 { \exp_after:wN \c_one \tex_romannumeral:D }
9835 \cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
9836 {
9837   \if_meaning:w 2 #1
9838     \if_int_compare:w #3 > \c_zero
9839       \__fp_round_return_one:
9840     \fi:
9841   \fi:
9842   \c_zero
9843 }
9844 \cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { \c_zero }
9845 \cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
9846 {
9847   \if_meaning:w 0 #1
9848     \if_int_compare:w #3 > \c_zero
9849       \__fp_round_return_one:
9850     \fi:
9851   \fi:
9852   \c_zero
9853 }
9854 \cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
9855 {
9856   \if_int_compare:w #3 > \c_five
9857     \__fp_round_return_one:
9858   \else:
9859     \if_meaning:w 5 #3
9860       \if_int_odd:w #2 \exp_stop_f:
9861       \__fp_round_return_one:
9862     \fi:
9863   \fi:
9864   \fi:
9865   \c_zero
9866 }
9867 \cs_new_eq:NN \__fp_round:NNN \__fp_round_to_nearest:NNN

```

(End definition for `__fp_round:NNN`. This function is documented on page ??.)

`__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.

```

9868 \cs_new:Npn \__fp_round_s:NNNw #1 #2 #3 #4;
9869 {
9870   \exp_after:wN \__fp_round:NNN
9871   \exp_after:wN #1
9872   \exp_after:wN #2
9873   \int_use:N \__int_eval:w
9874   \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
9875   \if_meaning:w 5 #3 1 \fi:
9876   \exp_stop_f:
9877   \if_int_compare:w \__int_eval:w #4 > \c_zero
9878   1 +
9879   \fi:
9880   \fi:
9881   #3
9882 ;
9883 }

```

(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.

```

9884 \cs_new:Npn \__fp_round_digit:Nw #1 #2;
9885 {
9886   \if_int_odd:w \if_meaning:w 0 #1 \c_one \else:
9887   \if_meaning:w 5 #1 \c_one \else:
9888   \c_zero \fi: \fi:
9889   \if_int_compare:w \__int_eval:w #2 > \c_zero
9890   \__int_eval:w \c_one +
9891   \fi:
9892   \fi:
9893   #1
9894 }

```

(End definition for `__fp_round_digit:Nw`.)

`__fp_round_neg:NNN` This expands to `\c_zero` or `\c_one`. Consider a number of the form $\langle final\ sign \rangle . X \dots X \langle digit_1 \rangle$
`__fp_round_to_nearest_neg:NNN` with exactly 15 (non-all-zero) digits before $\langle digit_1 \rangle$, and subtract from it $\langle final\ sign \rangle . 0 \dots 0 \langle digit_2 \rangle$,
`__fp_round_to_ninf_neg:NNN` where there are 16 zeros. If in the current rounding mode the result should be rounded
`__fp_round_to_zero_neg:NNN` down, then this function returns `\c_one`. Otherwise, *i.e.*, if the result is rounded back to
`__fp_round_to_pinf_neg:NNN` 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.

```

9895 \cs_new:Npn \__fp_round_to_ninf_neg:NNN #1 #2 #3
9896 {
9897   \if_meaning:w 0 #1
9898   \if_int_compare:w #3 > \c_zero

```

```

9899         \__fp_round_return_one:
9900         \fi:
9901         \fi:
9902         \c_zero
9903     }
9904 \cs_new:Npn \__fp_round_to_zero_neg:NNN #1 #2 #3
9905 {
9906     \if_int_compare:w #3 > \c_zero
9907         \__fp_round_return_one:
9908     \fi:
9909     \c_zero
9910 }
9911 \cs_new:Npn \__fp_round_to_pinf_neg:NNN #1 #2 #3
9912 {
9913     \if_meaning:w 2 #1
9914         \if_int_compare:w #3 > \c_zero
9915             \__fp_round_return_one:
9916         \fi:
9917     \fi:
9918     \c_zero
9919 }
9920 \cs_new_eq:NN \__fp_round_to_nearest_neg:NNN \__fp_round_to_nearest:NNN
9921 \cs_new_eq:NN \__fp_round_neg:NNN \__fp_round_to_nearest_neg:NNN

```

(End definition for __fp_round_neg:NNN. This function is documented on page ??.)

25.2 The round function

```

\__fp_round:Nww
\__fp_round:Nwn
\__fp_round_normal:NwNNnw
\__fp_round_normal:NnnwNNnn
\__fp_round_pack:Nw
\__fp_round_normal:NNwNnn
\__fp_round_normal_end:wwNnn
\__fp_round_special:NwwNnn
\__fp_round_special_aux:Nw

```

```

9922 \cs_new:Npn \__fp_round:Nww #1#2 ; #3 ;
9923 {
9924     \__fp_small_int:wTF #3; { \__fp_round:Nwn #1#2; }
9925     {
9926         \__fp_invalid_operation_tl_o:nf
9927         { round } { \__fp_array_to_clist:n { #2; #3; } }
9928     }
9929 }
9930 \cs_new:Npn \__fp_round:Nwn #1 \s__fp \__fp_chk:w #2#3#4; #5
9931 {
9932     \if_meaning:w 1 #2
9933         \exp_after:wN \__fp_round_normal:NwNNnw
9934         \exp_after:wN #1
9935         \__int_value:w #5
9936     \else:
9937         \exp_after:wN \__fp_exp_after_o:w
9938     \fi:
9939     \s__fp \__fp_chk:w #2#3#4;
9940 }
9941 \cs_new:Npn \__fp_round_normal:NwNNnw #1#2 \s__fp \__fp_chk:w 1#3#4#5;
9942 {

```

```

9943     \_fp_decimate:nNnnnn { \c_sixteen - #4 - #2 }
9944     \_fp_round_normal:NnnwNnn #5 #1 #3 {#4} {#2}
9945   }
9946   \cs_new:Npn \_fp_round_normal:NnnwNnn #1#2#3#4; #5#6
9947   {
9948     \exp_after:wN \_fp_round_normal:NNwNnn
9949     \int_use:N \_int_eval:w
9950     \if_int_compare:w #2 > \c_zero
9951       1 \_int_value:w #2
9952       \exp_after:wN \_fp_round_pack:Nw
9953       \int_use:N \_int_eval:w 1#3 +
9954     \else:
9955       \if_int_compare:w #3 > \c_zero
9956       1 \_int_value:w #3 +
9957       \fi:
9958     \fi:
9959     \exp_after:wN #5
9960     \exp_after:wN #6
9961     \use_none:nnnnnn #3
9962     #1
9963     \_int_eval_end:
9964     0000 0000 0000 0000 ; #6
9965   }
9966   \cs_new:Npn \_fp_round_pack:Nw #1
9967   { \if_meaning:w 2 #1 + \c_one \fi: \_int_eval_end: }
9968   \cs_new:Npn \_fp_round_normal:NNwNnn #1 #2
9969   {
9970     \if_meaning:w 0 #2
9971       \exp_after:wN \_fp_round_special:NwwNnn
9972       \exp_after:wN #1
9973     \fi:
9974     \_fp_pack_twice_four:wNNNNNNNN
9975     \_fp_pack_twice_four:wNNNNNNNN
9976     \_fp_round_normal_end:wwNnn
9977     ; #2
9978   }
9979   \cs_new:Npn \_fp_round_normal_end:wwNnn #1;#2;#3#4#5
9980   {
9981     \exp_after:wN \_fp_exp_after_o:w \tex_romannumeral:D -‘0
9982     \_fp_sanitize:Nw #3 #4 ; #1 ;
9983   }
9984   \cs_new:Npn \_fp_round_special:NwwNnn #1#2;#3;#4#5#6
9985   {
9986     \if_meaning:w 0 #1
9987       \_fp_case_return:nw
9988       { \exp_after:wN \_fp_zero_fp:N \exp_after:wN #4 }
9989     \else:
9990       \exp_after:wN \_fp_round_special_aux:Nw
9991       \exp_after:wN #4
9992       \int_use:N \_int_eval:w \c_one

```

```

9993         \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
9994     \fi:
9995     ;
9996 }
9997 \cs_new:Npn \__fp_round_special_aux:Nw #1#2;
9998 {
9999     \exp_after:wN \__fp_exp_after_o:w \tex_romannumeral:D -‘0
10000     \__fp_sanitize:Nw #1#2; {1000}{0000}{0000}{0000};
10001 }

```

(End definition for `__fp_round:Nww` and `__fp_round:Nwn`. These functions are documented on page ??.)

```

10002 </initex | package>

```

26 l3fp-parse implementation

```

10003 <*initex | package>
10004 <@@=fp>

```

26.1 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.

- 32 Juxtaposition for implicit multiplication.
- 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 `%`.

- 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.

26.1.1 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 ;
\tex_romannumeral:D \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 `\tex_romannumeral:D`, 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 ;
\tex_romannumeral:D \c_zero 333444 ;
```

where in fact `\exp_after:wN` has already been expanded, `__int_value:w` has already seen 12345, and `\tex_romannumeral:D` 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.

26.1.2 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 `^`.
- Compare the precedences of `-` and `^`. Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to `\operand:Nw ^`.
- Clean up 3 and find `*`.

- Compare the precedences of \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_⟨operator⟩:N` when it is encountered for the first time. Instead of passing both precedences to a test function to do the comparison steps above, we pass the *⟨precedence⟩* (of the earlier operator) to the `infix` auxiliary for the following *⟨operator⟩*, to know whether to perform the computation of the *⟨operator⟩*. If it should not be performed, the `infix` auxiliary expands to

```
@ \use_none:n \_fp\_parse\_infix\_⟨operator⟩:N
```

and otherwise it calls `_fp_parse_operand:Nw` with the precedence of the *⟨operator⟩* to find its second operand *⟨number₂⟩* and the next *⟨operator₂⟩*, and expands to

```
@ \_fp\_parse\_apply\_binary:NwNwN
  ⟨operator⟩ ⟨number2⟩
@ \_fp\_parse\_infix\_⟨operator2⟩:N
```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand *⟨number⟩* is before the `infix` function in the input stream. This is why we stop the expansion here and give control to another function to close the loop.

A definition of `_fp_parse_operand:Nw ⟨precedence⟩` with some of the expansion control removed is

```
\exp_after:wN \_fp\_parse\_continue:NwN
\exp_after:wN ⟨precedence⟩
\tex_romannumeral:D -‘0
\_fp\_parse\_one:Nw ⟨precedence⟩
```

This expands `_fp_parse_one:Nw ⟨precedence⟩` completely, which finds a number, wraps the next *⟨operator⟩* into an `infix` function, feeds this function the *⟨precedence⟩*, and expands it, yielding either

```
\_fp\_parse\_continue:NwN ⟨precedence⟩
  ⟨number⟩ @
\use_none:n \_fp\_parse\_infix\_⟨operator⟩:N
```

or

```
\_fp\_parse\_continue:NwN ⟨precedence⟩
  ⟨number⟩ @
\_fp\_parse\_apply\_binary:NwNwN
  ⟨operator⟩ ⟨number2⟩
@ \_fp\_parse\_infix\_⟨operator2⟩:N
```

The definition of `_fp_parse_continue:NwN` is then very simple:

```
\cs_new:Npn \_fp\_parse\_continue:NwN #1#2@#3 { #3 #1 #2 @ }
```

In the first case, #3 is `\use_none:n`, yielding

```
\use_none:n <precedence> <number> @
\__fp_parse_infix_<operator>:N
```

then `<number> @ __fp_parse_infix_<operator>:N`. In the second case, #3 is `__fp_parse_apply_binary:NwNwN`, whose role is to compute `<number> <operator> <number2>` and to prepare for the next comparison of precedences: first we get

```
\__fp_parse_apply_binary:NwNwN
<precedence> <number> @
<operator> <number2>
@ \__fp_parse_infix_<operator2>:N
```

then

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN <precedence>
\tex_romannumeral:D -‘0
\__fp_<operator>_o:ww <number> <number2>
\tex_romannumeral:D -‘0
\__fp_parse_infix_<operator2>:N <precedence>
```

where `__fp_<operator>_o:ww` computes `<number> <operator> <number2>` and expands after the result, thus triggers the comparison of the precedence of the `<operator2>` and the `<precedence>`, continuing the loop.

We have introduced the most important functions here, and the next few paragraphs will describe various subtleties.

26.1.3 Prefix operators, parentheses, and functions

Prefix operators (unary `-`, `+`, `!`) and parentheses are taken care of by the same mechanism, and functions (`sin`, `exp`, etc.) as well. Finding the argument of the unary `-`, for instance, is very similar to grabbing the second operand of a binary infix operator, with a subtle precedence explained below. Once that operand is found, the operator can be applied to it (for the unary `-`, this simply flips the sign). A left parenthesis is just a prefix operator with a very low precedence equal to that of the closing parenthesis (which is treated as an infix operator, since it normally appears just after numbers), so that all computations are performed until the closing parenthesis. The prefix operator associated to the left parenthesis does not alter its argument, but it removes the closing parenthesis (with some checks).

Prefix operators are the reason why we only summarily described the function `__fp_parse_one:Nw` earlier. This function is responsible for reading in the input stream the first possible `<number>` and the next infix `<operator>`. If what follows `__fp_parse_one:Nw <precedence>` is a prefix operator, then we must find the operand of this prefix operator through a nested call to `__fp_parse_operand:Nw` with the appropriate precedence, then apply the operator to the operand found to yield the result of `__fp_parse_one:Nw`. So far, all is simple.

The unary operators `+`, `-`, `!` complicate things a little bit: `-3**2` should be $-(3^2) = -9$, and not $(-3)^2 = 9$. This would easily be done by giving `-` a lower precedence, equal to that of the infix `+` and `-`. Unfortunately, this fails in cases such as `3**-2*4`, yielding $3^{-2 \times 4}$ instead of the correct $3^{-2} \times 4$. A second attempt would be to call `_fp_parse_operand:Nw` with the $\langle precedence \rangle$ 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, except for juxtaposition.

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.

26.1.4 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 $\langle significand \rangle$ of a number: we look for an exponent.
- If the next token is a digit, we remove any leading zeros, then read a significand larger than 1 if the next character is a digit, read a significand smaller than 1 if the next character is a period, or we have found a significand equal to 0 otherwise, and look for an exponent.

- If the next token is a letter, we collect more letters until the first non-letter: the resulting word may denote a function such as `asin`, a constant such as `pi` or be unknown. In the first case, we call `__fp_parse_operand:Nw` to find the argument of the function, then apply the function, before declaring that we are done. Otherwise, we are done, either with the value of the constant, or with the value `nan` for unknown words.
- If the next token is anything else, we check whether it is a known prefix operator, in which case `__fp_parse_operand:Nw` finds its operand. If it is not known, then either a number is missing (if the token is a known infix operator) or the token is simply invalid in floating point expressions.

Once a number is found, `__fp_parse_one:Nw` also finds an infix operator. This goes as follows.

-
- If the next token is a control sequence, it could be the end-marker `\s__fp_mark`, which has the lowest precedence, 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`.

26.2 Main auxiliary functions

<code>__fp_parse_operand:Nw</code>	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 <code>end</code> , and the "..." start just after the $\langle operation \rangle$. (End definition for <code>__fp_parse_operand:Nw</code> .)
<code>__fp_parse_infix_+:N</code>	If <code>+</code> 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 <code>__fp_parse_infix_+:N</code> .)
<code>__fp_parse_one:Nw</code>	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 <code>__fp_parse_one:Nw</code> .)

26.3 Helpers

<code>__fp_parse_expand:w</code>	This function must always come within a <code>\romannumeral</code> 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. 10005 <code>\cs_new:Npn __fp_parse_expand:w #1 { -'0 #1 }</code>
-----------------------------------	---

(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.

```
10006 \cs_new:Npn \__fp_parse_return_semicolon:w
10007     #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
10008 \group_begin:
10009 \char_set_catcode_other:N \S
10010 \char_set_catcode_other:N \F
10011 \char_set_catcode_other:N \P
10012 \char_set_lccode:nn { '\- } { '\_ }
10013 \tl_to_lowercase:n
10014 {
10015     \group_end:
10016     \cs_new:Npn \__fp_type_from_scan:N #1
10017     {
10018         \exp_after:wN \__fp_type_from_scan:w
10019         \token_to_str:N #1 \q_mark S--FP-? \q_mark \q_stop
10020     }
10021     \cs_new:Npn \__fp_type_from_scan:w #1 S--FP #2 \q_mark #3 \q_stop {#2}
10022 }
```

(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.
`__fp_parse_digits_vi:N` The functions read tokens one by one, and output digits into the input stream, until
`__fp_parse_digits_v:N` meeting a non-digit, or up to a number of digits equal to their index. The full expansion
`__fp_parse_digits_iv:N` is
`__fp_parse_digits_iii:N`
`__fp_parse_digits_ii:N`
`__fp_parse_digits_i: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.

```
10023 \cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
10024 {
10025     \cs_new:cpn { __fp_parse_digits_ #1 :N } ##1
10026     {
10027         \if_int_compare:w \c_nine < 1 \token_to_str:N ##1 \exp_stop_f:
10028         \token_to_str:N ##1 \exp_after:wN #2 \tex_romannumeral:D
10029         \else:
10030             \__fp_parse_return_semicolon:w #3 ##1
10031         \fi:
```

```

10032         \__fp_parse_expand:w
10033     }
10034 }
10035 \__fp_tmp:w {vii} \__fp_parse_digits_vi:N { 0000000 ; 7 }
10036 \__fp_tmp:w {vi} \__fp_parse_digits_v:N { 000000 ; 6 }
10037 \__fp_tmp:w {v} \__fp_parse_digits_iv:N { 00000 ; 5 }
10038 \__fp_tmp:w {iv} \__fp_parse_digits_iii:N { 0000 ; 4 }
10039 \__fp_tmp:w {iii} \__fp_parse_digits_ii:N { 000 ; 3 }
10040 \__fp_tmp:w {ii} \__fp_parse_digits_i:N { 00 ; 2 }
10041 \__fp_tmp:w {i} \__fp_parse_digits_:N { 0 ; 1 }
10042 \cs_new_nopar:Npn \__fp_parse_digits_:N { ; ; 0 }
(End definition for \__fp_parse_digits_vii:N and others.)

```

26.4 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 *precedence*, and #2 the first token of the operand. We distinguish four cases: #2 is equal to `\scan_stop:` in meaning, #2 is a different control sequence, #2 is a digit, and #2 is something else (this last case will be split further. Despite the earlier `f`-expansion, #2 may still be expandable if it was protected by `\exp_not:N`, as happens with the \LaTeX 2_ϵ command `\protect`. Testing if #2 is a control sequence thus includes `\exp_not:N`.

```

10043 \cs_new:Npn \__fp_parse_one:Nw #1 #2
10044 {
10045     \if_catcode:w \scan_stop: \exp_not:N #2
10046     \if_meaning:w \scan_stop: #2
10047         \exp_after:wN \exp_after:wN
10048         \exp_after:wN \__fp_parse_one_fp:NN
10049     \else:
10050         \exp_after:wN \exp_after:wN
10051         \exp_after:wN \__fp_parse_one_register:NN
10052     \fi:
10053 \else:
10054     \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
10055         \exp_after:wN \exp_after:wN
10056         \exp_after:wN \__fp_parse_one_digit:NN
10057     \else:
10058         \exp_after:wN \exp_after:wN
10059         \exp_after:wN \__fp_parse_one_other:NN
10060     \fi:
10061 \fi:
10062     #1 #2
10063 }
(End definition for \__fp_parse_one:Nw.)

```

`__fp_parse_one_fp:NN` This function receives a *precedence* and a control sequence equal to `\scan_stop:` in meaning. There are three cases, dispatched using `__fp_type_from_scan:N`.
`__fp_exp_after_mark_f:nw`
`__fp_exp_after_?_f:nw`

- `\s__fp` starts a floating point number, and we call `__fp_exp_after_f:nw`, which `f`-expands after the floating point.
- `\s__fp_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.

```

10064 \cs_new:Npn \__fp_parse_one_fp:NN #1#2
10065 {
10066   \cs:w __fp_exp_after \__fp_type_from_scan:N #2 _f:nw \cs_end:
10067   {
10068     \exp_after:wN \__fp_parse_infix:NN
10069     \exp_after:wN #1 \tex_romannumeral:D \__fp_parse_expand:w
10070   }
10071   #2
10072 }
10073 \cs_new:Npn \__fp_exp_after_mark_f:nw #1
10074 {
10075   \_msg_kernel_expandable_error:nn { kernel } { fp-early-end }
10076   \exp_after:wN \c_nan_fp \tex_romannumeral:D -'0 #1
10077 }
10078 \cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
10079 {
10080   \_msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
10081   \exp_after:wN \c_nan_fp \tex_romannumeral:D -'0 #1
10082 }
(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 `TEX` does not accurately represent the binary value that it manipulates, so we extract this binary value as a number of scaled points with `__int_value:w __dim_eval:w` $\langle decimal value \rangle$ `pt`, and use an auxiliary of `\dim_to_fp:n`, which performs the multiplication by 2^{-16} , correctly rounded.

```

10083 \cs_new:Npn \__fp_parse_one_register:NN #1#2
10084 {
10085   \exp_after:wN \__fp_parse_infix_after_operand:NwN

```

```

10086 \exp_after:wN #1
10087 \tex_romannumeral:D -‘0
10088 \exp_after:wN \_fp_parse_one_register_aux:Nw
10089 \exp_after:wN #2
10090 \_int_value:w
10091 \exp_after:wN \_fp_parse_exponent:N
10092 \tex_romannumeral:D \_fp_parse_expand:w
10093 }
10094 \group_begin:
10095 \char_set_catcode_other:N \P
10096 \char_set_catcode_other:N \T
10097 \char_set_catcode_other:N \M
10098 \char_set_catcode_other:N \U
10099 \tl_to_lowercase:n
10100 {
10101 \group_end:
10102 \cs_new:Npn \_fp_parse_one_register_aux:Nw #1
10103 {
10104 \exp_after:wN \use:nn
10105 \exp_after:wN \_fp_parse_one_register_auxii:wwwNw
10106 \exp_after:wN { \tex_the:D \exp_not:N #1 }
10107 ; \_fp_parse_one_register_dim:ww
10108 PT ; \_fp_parse_one_register_mu:www
10109 . PT ; \_fp_parse_one_register_int:www
10110 \q_stop
10111 }
10112 \cs_new:Npn \_fp_parse_one_register_auxii:wwwNw
10113 #1 . #2 PT #3 ; #4#5 \q_stop { #4 #1.#2; }
10114 \cs_new:Npn \_fp_parse_one_register_mu:www #1 MU; #2;
10115 { \_fp_parse_one_register_dim:ww #1; }
10116 }
10117 \cs_new:Npn \_fp_parse_one_register_int:www #1; #2.; #3;
10118 { \_fp_parse:n { #1 e #3 } }
10119 \cs_new:Npn \_fp_parse_one_register_dim:ww #1; #2;
10120 {
10121 \exp_after:wN \_fp_from_dim_test:ww
10122 \_int_value:w #2 \exp_after:wN ,
10123 \_int_value:w \_dim_eval:w #1 pt ;
10124 }

```

(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.

```

10125 \cs_new:Npn \_fp_parse_one_digit:NN #1
10126 {
10127 \exp_after:wN \_fp_parse_infix_after_operand:NwN

```

```

10128 \exp_after:wN #1
10129 \tex_romannumeral:D -‘0
10130 \exp_after:wN \__fp_sanitize:wN
10131 \int_use:N \__int_eval:w \c_zero \__fp_parse_trim_zeros:N
10132 }

```

(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.

```

10133 \cs_new:Npn \__fp_parse_one_other:NN #1 #2
10134 {
10135   \if_int_compare:w
10136     \__int_eval:w
10137     ( ‘#2 \if_int_compare:w ‘#2 > ‘Z - \c_thirty_two \fi: ) / 26
10138     = \c_three
10139     \exp_after:wN \__fp_parse_word:Nw
10140     \exp_after:wN #1
10141     \exp_after:wN #2
10142     \tex_romannumeral:D \exp_after:wN \__fp_parse_letters:N
10143     \tex_romannumeral:D
10144   \else:
10145     \exp_after:wN \__fp_parse_prefix:NNN
10146     \exp_after:wN #1
10147     \exp_after:wN #2
10148     \cs:w \__fp_parse_prefix_#2:Nw \exp_after:wN \cs_end:
10149     \tex_romannumeral:D
10150   \fi:
10151   \__fp_parse_expand:w
10152 }

```

(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.

```

10153 \cs_new:Npn \__fp_parse_word:Nw #1#2;
10154 {
10155   \cs_if_exist_use:cF { \__fp_parse_word_#2:N }
10156   {
10157     \__msg_kernel_expandable_error:nnn
10158     { kernel } { unknown-fp-word } {#2}
10159     \exp_after:wN \c_nan_fp \tex_romannumeral:D -‘0
10160     \__fp_parse_infix:NN
10161   }

```

```

10162     #1
10163   }
10164   \cs_new:Npn \__fp_parse_letters:N #1
10165   {
10166     -'0
10167     \if_int_compare:w
10168       \if_catcode:w \scan_stop: \exp_not:N #1
10169       \c_zero
10170     \else:
10171       \__int_eval:w
10172         ( '#1 \if_int_compare:w '#1 > 'Z - \c_thirty_two \fi: )
10173         / 26
10174       \fi:
10175     = \c_three
10176     \exp_after:wN #1
10177     \tex_romannumeral:D \exp_after:wN \__fp_parse_letters:N
10178     \tex_romannumeral:D
10179   \else:
10180     \__fp_parse_return_semicolon:w #1
10181   \fi:
10182   \__fp_parse_expand:w
10183 }

```

(End definition for __fp_parse_word:Nw. This function is documented on page ??.)

__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.

```

10184 \cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
10185 {
10186   \if_meaning:w \scan_stop: #3
10187     \exp_after:wN \__fp_parse_prefix_unknown:NNN
10188     \exp_after:wN #2
10189   \fi:
10190   #3 #1
10191 }
10192 \cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
10193 {
10194   \cs_if_exist:cTF { \__fp_parse_infix_#1:N }
10195   {
10196     \__msg_kernel_expandable_error:nnn
10197       { kernel } { fp-missing-number } {#1}
10198     \exp_after:wN \c_nan_fp \tex_romannumeral:D -'0
10199     \__fp_parse_infix:NN #3 #1
10200   }
10201   {

```

```

10202         \_msg_kernel_expandable_error:nnn
10203         { kernel } { fp-unknown-symbol } {#1}
10204         \_fp_parse_one:Nw #3
10205     }
10206 }

```

(End definition for `_fp_parse_prefix:NNN` and `_fp_parse_prefix_unknown:NNN`.)

26.4.1 Numbers: trimming leading zeros

Numbers will be parsed as follows: first we trim leading zeros, then if the next character is a digit, start reading a significand ≥ 1 with the set of functions `_fp_parse_large...`; if it is a period, the significand is < 1 ; and otherwise it is zero. In the second case, trim additional zeros after the period, counting them for an exponent shift $\langle exp_1 \rangle < 0$, then read the significand with the set of functions `_fp_parse_small...`. Once the significand is read, read the exponent if `e` is present.

```

\_fp_parse_trim_zeros:N
\_fp_parse_trim_end:w

```

This function expects an already expanded token. It removes any leading zero, then distinguishes three cases: if the first non-zero token is a digit, then call `_fp_parse_large:N` (the significand is ≥ 1); if it is `.`, then continue trimming zeros with `_fp_parse_strim_zeros:N`; otherwise, our number is exactly zero, and we call `_fp_parse_zero:` to take care of that case.

```

10207 \cs_new:Npn \_fp_parse_trim_zeros:N #1
10208 {
10209     \if:w 0 \exp_not:N #1
10210         \exp_after:wN \_fp_parse_trim_zeros:N
10211         \tex_romannumeral:D
10212     \else:
10213         \if:w . \exp_not:N #1
10214             \exp_after:wN \_fp_parse_strim_zeros:N
10215             \tex_romannumeral:D
10216         \else:
10217             \_fp_parse_trim_end:w #1
10218         \fi:
10219     \fi:
10220     \_fp_parse_expand:w
10221 }
10222 \cs_new:Npn \_fp_parse_trim_end:w #1 \fi: \fi: \_fp_parse_expand:w
10223 {
10224     \fi:
10225     \fi:
10226     \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10227         \exp_after:wN \_fp_parse_large:N
10228     \else:
10229         \exp_after:wN \_fp_parse_zero:
10230     \fi:
10231     #1
10232 }

```

(End definition for `_fp_parse_trim_zeros:N` and `_fp_parse_trim_end:w`.)

`__fp_parse_strim_zeros:N` 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”.

```

10233 \cs_new:Npn \__fp_parse_strim_zeros:N #1
10234 {
10235   \if:w 0 \exp_not:N #1
10236     - \c_one
10237     \exp_after:wN \__fp_parse_strim_zeros:N \tex_romannumeral:D
10238   \else:
10239     \__fp_parse_strim_end:w #1
10240   \fi:
10241   \__fp_parse_expand:w
10242 }
10243 \cs_new:Npn \__fp_parse_strim_end:w #1 \fi: \__fp_parse_expand:w
10244 {
10245   \fi:
10246   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10247     \exp_after:wN \__fp_parse_small:N
10248   \else:
10249     \exp_after:wN \__fp_parse_zero:
10250   \fi:
10251   #1
10252 }

```

(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).

```

10253 \cs_new:Npn \__fp_parse_zero:
10254 {
10255   \exp_after:wN ; \exp_after:wN 1
10256   \__int_value:w \__fp_parse_exponent:N
10257 }

```

(End definition for __fp_parse_zero:.)

26.4.2 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.

```

10258 \cs_new:Npn \__fp_parse_small:N #1

```



```

10259 {
10260   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
10261   \int_use:N \__int_eval:w 1 \token_to_str:N #1
10262   \exp_after:wN \__fp_parse_small_leading:wwNN
10263   \__int_value:w 1
10264   \exp_after:wN \__fp_parse_digits_vii:N
10265   \tex_romannumeral:D \__fp_parse_expand:w
10266 }

```

(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.

```

10267 \cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
10268 {
10269   #1 #2
10270   \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
10271   \exp_after:wN \c_zero
10272   \int_use:N \__int_eval:w 1
10273   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
10274     \token_to_str:N #4
10275     \exp_after:wN \__fp_parse_small_trailing:wwNN
10276     \__int_value:w 1
10277     \exp_after:wN \__fp_parse_digits_vi:N
10278     \tex_romannumeral:D
10279   \else:
10280     0000 0000 \__fp_parse_exponent:Nw #4
10281   \fi:
10282   \__fp_parse_expand:w
10283 }

```

(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.

```

10284 \cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
10285 {
10286   #1 #2
10287   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
10288     \token_to_str:N #4
10289     \exp_after:wN \__fp_parse_small_round:NN
10290     \exp_after:wN #4
10291     \tex_romannumeral:D

```

```

10292     \else:
10293         0 \_fp_parse_exponent:Nw #4
10294     \fi:
10295     \_fp_parse_expand:w
10296 }
(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.

```

10297 \cs_new:Npn \_fp_parse_pack_trailing:NNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
10298 {
10299     \if_meaning:w 2 #2 + \c_one \fi:
10300     ; #8 + #1 ; {#3#4#5#6} {#7};
10301 }
10302 \cs_new:Npn \_fp_parse_pack_leading:NNNNNNww #1 #2#3#4#5 #6; #7;
10303 {
10304     + #7
10305     \if_meaning:w 2 #1 \_fp_parse_pack_carry:w \fi:
10306     ; 0 {#2#3#4#5} {#6}
10307 }
10308 \cs_new:Npn \_fp_parse_pack_carry:w \fi: ; 0 #1
10309 { \fi: + \c_one ; 0 {1000} }
(End definition for \_fp_parse_pack_trailing:NNNNNNww, \_fp_parse_pack_leading:NNNNNNww, and
\_fp_parse_pack_carry:w.)

```

26.4.3 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.

```

10310 \cs_new:Npn \_fp_parse_large:N #1
10311 {
10312     \exp_after:wN \_fp_parse_large_leading:wwNN
10313     \_int_value:w 1 \token_to_str:N #1
10314     \exp_after:wN \_fp_parse_digits_vii:N

```

```

10315 \tex_romannumeral:D \__fp_parse_expand:w
10316 }

```

(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.

```

10317 \cs_new:Npn \__fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
10318 {
10319   + \c_eight - #3
10320   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
10321   \int_use:N \__int_eval:w 1 #1
10322   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
10323     \exp_after:wN \__fp_parse_large_trailing:wwNN
10324     \__int_value:w 1 \token_to_str:N #4
10325     \exp_after:wN \__fp_parse_digits_vi:N
10326     \tex_romannumeral:D
10327   \else:
10328     \if:w . \exp_not:N #4
10329       \exp_after:wN \__fp_parse_small_leading:wwNN
10330       \__int_value:w 1
10331       \cs:w
10332         __fp_parse_digits_
10333         \tex_romannumeral:D #3
10334         :N \exp_after:wN
10335       \cs_end:
10336       \tex_romannumeral:D
10337     \else:
10338       #2
10339       \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
10340       \exp_after:wN \c_zero
10341       \__int_value:w 1 0000 0000
10342       \__fp_parse_exponent:Nw #4
10343     \fi:
10344   \fi:
10345   \__fp_parse_expand:w
10346 }

```

(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.

```

10347 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
10348 {
10349   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
10350   \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
10351   \exp_after:wN \c_eight
10352   \int_use:N \__int_eval:w 1 #1 \token_to_str:N #4
10353   \exp_after:wN \__fp_parse_large_round:NN
10354   \exp_after:wN #4
10355   \tex_romannumeral:D
10356 \else:
10357   \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
10358   \int_use:N \__int_eval:w \c_seven - #3 \exp_stop_f:
10359   \int_use:N \__int_eval:w 1 #1
10360   \if:w . \exp_not:N #4
10361   \exp_after:wN \__fp_parse_small_trailing:wwNN
10362   \__int_value:w 1
10363   \cs:w
10364     __fp_parse_digits_
10365     \tex_romannumeral:D #3
10366     :N \exp_after:wN
10367     \cs_end:
10368     \tex_romannumeral:D
10369   \else:
10370     #2 0 \__fp_parse_exponent:Nw #4
10371   \fi:
10372 \fi:
10373 \__fp_parse_expand:w
10374 }
```

(End definition for `__fp_parse_large_trailing:wwNN`.)

26.4.4 Number: beyond 16 digits, rounding

`__fp_parse_round_loop:N`
`__fp_parse_round_up:N`

This loop is called when rounding a number (whether the mantissa is small or large). It should appear in an integer expression. This function reads digits one by one, until reaching a non-digit, and adds 1 to the integer expression for each digit. If all digits found are 0, the function ends the expression by `;\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.

```

10375 \cs_new:Npn \__fp_parse_round_loop:N #1
10376 {
10377   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10378   + \c_one
10379   \if:w 0 \token_to_str:N #1
10380   \exp_after:wN \__fp_parse_round_loop:N
10381   \tex_romannumeral:D
```

```

10382     \else:
10383         \exp_after:wN \__fp_parse_round_up:N
10384         \tex_romannumeral:D
10385     \fi:
10386 \else:
10387     \__fp_parse_return_semicolon:w \c_zero #1
10388 \fi:
10389 \__fp_parse_expand:w
10390 }
10391 \cs_new:Npn \__fp_parse_round_up:N #1
10392 {
10393     \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10394         + \c_one
10395         \exp_after:wN \__fp_parse_round_up:N
10396         \tex_romannumeral:D
10397     \else:
10398         \__fp_parse_return_semicolon:w \c_one #1
10399     \fi:
10400     \__fp_parse_expand:w
10401 }

```

(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.

```

10402 \cs_new:Npn \__fp_parse_round_after:wN #1; #2
10403 {
10404     + #2 \exp_after:wN ;
10405     \int_use:N \__int_eval:w #1 + \__fp_parse_exponent:N
10406 }

```

(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 `;\langle exponent \rangle` only. `__fp_parse_round_after:wN` Otherwise, we will expand to `+\c_zero` or `+\c_one`, then `;\langle exponent \rangle`. 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`.

```

10407 \cs_new:Npn \__fp_parse_small_round:NN #1#2
10408 {
10409     \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
10410         +
10411         \exp_after:wN \__fp_round_s:NNNw
10412         \exp_after:wN 0

```

```

10413 \exp_after:wN #1
10414 \exp_after:wN #2
10415 \int_use:N \__int_eval:w
10416 \exp_after:wN \__fp_parse_round_after:wN
10417 \int_use:N \__int_eval:w \c_zero * \__int_eval:w \c_zero
10418 \exp_after:wN \__fp_parse_round_loop:N
10419 \tex_romannumeral:D
10420 \else:
10421 \__fp_parse_exponent:Nw #2
10422 \fi:
10423 \__fp_parse_expand:w
10424 }

```

(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.

```

10425 \cs_new:Npn \__fp_parse_large_round:NN #1#2
10426 {
10427   \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
10428   +
10429   \exp_after:wN \__fp_round_s:NNNw
10430   \exp_after:wN 0
10431   \exp_after:wN #1
10432   \exp_after:wN #2
10433   \int_use:N \__int_eval:w
10434   \exp_after:wN \__fp_parse_large_round_aux:wNN
10435   \int_use:N \__int_eval:w \c_one
10436   \exp_after:wN \__fp_parse_round_loop:N
10437   \else: %^^A could be dot, or e, or other
10438   \exp_after:wN \__fp_parse_large_round_test:NN
10439   \exp_after:wN #1
10440   \exp_after:wN #2
10441   \fi:
10442 }
10443 \cs_new:Npn \__fp_parse_large_round_test:NN #1#2
10444 {
10445   \if:w . \exp_not:N #2
10446   \exp_after:wN \__fp_parse_small_round:NN
10447   \exp_after:wN #1
10448   \tex_romannumeral:D
10449   \else:
10450   \__fp_parse_exponent:Nw #2

```

```

10451 \fi:
10452 \__fp_parse_expand:w
10453 }
10454 \cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
10455 {
10456 + #2
10457 \exp_after:wN \__fp_parse_round_after:wN
10458 \int_use:N \__int_eval:w #1
10459 \if:w . \exp_not:N #3
10460 + \c_zero * \__int_eval:w \c_zero
10461 \exp_after:wN \__fp_parse_round_loop:N
10462 \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
10463 \else:
10464 \exp_after:wN ;
10465 \exp_after:wN \c_zero
10466 \exp_after:wN #3
10467 \fi:
10468 }

```

(End definition for `__fp_parse_large_round:NN`, `__fp_parse_large_round_test:NN`, and `__fp_parse_large_round_aux:w`.)

26.4.5 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.

```

10469 \cs_new:Npn \__fp_parse_exponent:Nw #1 #2 \__fp_parse_expand:w

```

```

10470 {
10471   \exp_after:wN ;
10472   \__int_value:w #2 \__fp_parse_exponent:N #1
10473 }

```

(End definition for __fp_parse_exponent:Nw.)

__fp_parse_exponent:N
__fp_parse_exponent_aux:N

This function should be called within an __int_value:w expansion (or within an integer expression. It leaves digits of the exponent behind it in the input stream, and terminates the expansion with a semicolon. If there is no e, leave an exponent of 0. If there is an e, expand the next token to run some tests on it. The first rough test is that if the character code of #1 is greater than that of 9 (largest code valid for an exponent, less than any code valid for an identifier), there was in fact no exponent; otherwise, we search for the sign of the exponent.

```

10474 \cs_new:Npn \__fp_parse_exponent:N #1
10475 {
10476   \if:w e \exp_not:N #1
10477     \exp_after:wN \__fp_parse_exponent_aux:N
10478     \tex_romannumeral:D
10479   \else:
10480     0 \__fp_parse_return_semicolon:w #1
10481   \fi:
10482   \__fp_parse_expand:w
10483 }
10484 \cs_new:Npn \__fp_parse_exponent_aux:N #1
10485 {
10486   \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #1
10487     \c_zero \else: ' #1 \fi: > '9 \exp_stop_f:
10488     0 \exp_after:wN ; \exp_after:wN e
10489   \else:
10490     \exp_after:wN \__fp_parse_exponent_sign:N
10491   \fi:
10492   #1
10493 }

```

(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).

```

10494 \cs_new:Npn \__fp_parse_exponent_sign:N #1
10495 {
10496   \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
10497     \exp_after:wN \__fp_parse_exponent_sign:N
10498     \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
10499   \else:
10500     \exp_after:wN \__fp_parse_exponent_body:N
10501     \exp_after:wN #1
10502   \fi:
10503 }

```

(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).

```

10504 \cs_new:Npn \__fp_parse_exponent_body:N #1
10505 {
10506   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10507     \token_to_str:N #1
10508     \exp_after:wN \__fp_parse_exponent_digits:N
10509     \tex_romannumeral:D
10510   \else:
10511     \__fp_parse_exponent_keep:NTF #1
10512     { \__fp_parse_return_semicolon:w #1 }
10513     {
10514       \exp_after:wN ;
10515       \tex_romannumeral:D
10516     }
10517   \fi:
10518   \__fp_parse_expand:w
10519 }

```

(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.

```

10520 \cs_new:Npn \__fp_parse_exponent_digits:N #1
10521 {
10522   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10523     \token_to_str:N #1
10524     \exp_after:wN \__fp_parse_exponent_digits:N
10525     \tex_romannumeral:D
10526   \else:
10527     \__fp_parse_return_semicolon:w #1
10528   \fi:
10529   \__fp_parse_expand:w
10530 }

```

(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.

```

10531 \prg_new_conditional:Npnn \__fp_parse_exponent_keep:N #1 { TF }
10532 {
10533   \if_catcode:w \scan_stop: \exp_not:N #1
10534   \if_meaning:w \scan_stop: #1
10535   \if_int_compare:w
10536     \pdfTeX_strcmp:D { \s__fp } { \exp_not:N #1 } = \c_zero
10537     0
10538     \__msg_kernel_expandable_error:nnn
10539     { kernel } { fp-after-e } { floating~point~ }
10540     \prg_return_true:
10541   \else:
10542     0
10543     \__msg_kernel_expandable_error:nnn
10544     { kernel } { bad-variable } { #1 }
10545     \prg_return_false:
10546   \fi:
10547 \else:
10548   \if_int_compare:w
10549     \pdfTeX_strcmp:D { \__int_value:w #1 } { \tex_the:D #1 }
10550     = \c_zero
10551     \__int_value:w #1
10552   \else:
10553     0
10554     \__msg_kernel_expandable_error:nnn
10555     { kernel } { fp-after-e } { dimension~#1 }
10556   \fi:
10557   \prg_return_false:
10558 \fi:
10559 \else:
10560   0
10561   \__msg_kernel_expandable_error:nnn
10562   { kernel } { fp-missing } { exponent }
10563   \prg_return_true:
10564 \fi:
10565 }

```

(End definition for __fp_parse_exponent_keep:NNTF.)

26.5 Constants, functions and prefix operators

26.5.1 Prefix operators

__fp_parse_prefix+:Nw A unary + does nothing: we should continue looking for a number.

```

10566 \cs_new_eq:cN { \__fp_parse_prefix+:Nw } \__fp_parse_one:Nw

```

(End definition for __fp_parse_prefix+:Nw.)

__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 \tex_romannumeral:D thus the infix function #5.

```

10567 \cs_new:Npn \__fp_parse_apply_unary:NNNwN #1#2#3#4#5
10568 {
10569     #3 #2 #4 @
10570     \tex_romannumeral:D -'0 #5 #1
10571 }

```

(End definition for __fp_parse_apply_unary:NNNwN.)

__fp_parse_prefix_-:Nw
__fp_parse_prefix_!:Nw

The unary - and boolean not are harder: we parse the operand using a precedence equal to the maximum of the previous precedence ##1 and the precedence \c_twelve of the unary operator, then call the appropriate __fp_⟨operation⟩_o:w function, where the ⟨operation⟩ is set_sign or not.

```

10572 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
10573 {
10574     \cs_new:cpn { __fp_parse_prefix_ #1 :Nw } ##1
10575     {
10576         \exp_after:wN \__fp_parse_apply_unary:NNNwN
10577         \exp_after:wN ##1
10578         \exp_after:wN #4
10579         \cs:w __fp_#3_o:w \exp_after:wN \cs_end:
10580         \tex_romannumeral:D
10581         \if_int_compare:w #2 < ##1
10582             \__fp_parse_operand:Nw ##1
10583         \else:
10584             \__fp_parse_operand:Nw #2
10585         \fi:
10586         \__fp_parse_expand:w
10587     }
10588 }
10589 \__fp_tmp:w - \c_twelve { set_sign } 2
10590 \__fp_tmp:w ! \c_twelve { not } ?

```

(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.

```

10591 \cs_new:cpn { __fp_parse_prefix_:Nw } #1
10592 {
10593     \exp_after:wN \__fp_parse_infix_after_operand:NwN
10594     \exp_after:wN #1
10595     \tex_romannumeral:D -'0
10596     \exp_after:wN \__fp_sanitize:wN
10597     \int_use:N \__int_eval:w \c_zero \__fp_parse_strim_zeros:N
10598 }

```

(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.

```

10599 \group_begin:
10600 \char_set_catcode_letter:N (
10601 \char_set_catcode_letter:N )
10602 \cs_new:Npn \__fp_parse_prefix_(:Nw #1
10603 {
10604   \exp_after:wN \__fp_parse_lparen_after:NwN
10605   \exp_after:wN #1
10606   \tex_romannumeral:D
10607   \if_int_compare:w #1 = \c_sixteen
10608     \__fp_parse_operand:Nw \c_one
10609   \else:
10610     \__fp_parse_operand:Nw \c_zero
10611   \fi:
10612   \__fp_parse_expand:w
10613 }
10614 \cs_new:Npn \__fp_parse_lparen_after:NwN #1#2 @ #3
10615 {
10616   \token_if_eq_meaning:NNTF #3 \__fp_parse_infix_):N
10617   {
10618     \__fp_exp_after_array_f:w #2 \s__fp_stop
10619     \exp_after:wN \__fp_parse_infix:NN
10620     \exp_after:wN #1
10621     \tex_romannumeral:D \__fp_parse_expand:w
10622   }
10623   {
10624     \__msg_kernel_expandable_error:nnn
10625     { kernel } { fp-missing } { { } }
10626     #2 @ \use_none:n #3
10627   }
10628 }
10629 \group_end:
(End definition for \__fp_parse_prefix_(:Nw and \__fp_parse_lparen_after:NwN.)

```

26.5.2 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`.

```

10630 \cs_set_protected:Npn \__fp_tmp:w #1 #2
10631 {
10632   \cs_new_nopar:cpn { __fp_parse_word_#1:N }
10633   { \exp_after:wN #2 \tex_romannumeral:D -'0 \__fp_parse_infix:NN }

```

```

10634 }
10635 \__fp_tmp:w { inf } \c_inf_fp
10636 \__fp_tmp:w { nan } \c_nan_fp
10637 \__fp_tmp:w { pi } \c_pi_fp
10638 \__fp_tmp:w { deg } \c_one_degree_fp
10639 \__fp_tmp:w { true } \c_one_fp
10640 \__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
10641 \cs_set_protected:Npn \__fp_tmp:w #1 #2
10642 {
10643   \cs_new_nopar:cpn { __fp_parse_word_#1:N }
10644   {
10645     \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
10646     \s__fp \__fp_chk:w 10 #2 ;
10647   }
10648 }
10649 \__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
10650 \__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
10651 \__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
10652 \__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
10653 \__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
10654 \__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
10655 \__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
10656 \__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
10657 \__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
10658 \__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
10659 \__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
10660 \tl_map_inline:nn { {em} {ex} }
10661 {
10662   \cs_new_nopar:cpn { __fp_parse_word_#1:N }
10663   {
10664     \exp_after:wN \__fp_from_dim_test:ww
10665     \exp_after:wN 0 \exp_after:wN ,
10666     \__int_value:w \__dim_eval:w 1 #1 \exp_after:wN ;
10667     \tex_romannumeral:D -'0 \__fp_parse_infix:NN
10668   }
10669 }
(End definition for \__fp_parse_word_em:N and \__fp_parse_word_ex:N.)

```

26.5.3 Functions

```

\__fp_parse_unary_function:nNN
\__fp_parse_function:NNN
10670 \cs_new:Npn \__fp_parse_unary_function:nNN #1#2#3
10671 {
10672   \exp_after:wN \__fp_parse_apply_unary:NNNwN
10673   \exp_after:wN #3
10674   \exp_after:wN #2
10675   \cs:w \__fp_#1_o:w \exp_after:wN \cs_end:
10676   \tex_romannumeral:D
10677   \__fp_parse_operand:Nw \c_fifteen \__fp_parse_expand:w
10678 }
10679 \cs_new:Npn \__fp_parse_function:NNN #1#2#3
10680 {
10681   \exp_after:wN \__fp_parse_apply_unary:NNNwN
10682   \exp_after:wN #3
10683   \exp_after:wN #2
10684   \exp_after:wN #1
10685   \tex_romannumeral:D
10686   \__fp_parse_operand:Nw \c_sixteen \__fp_parse_expand:w
10687 }
(End definition for \__fp_parse_unary_function:nNN and \__fp_parse_function:NNN.)

```

Those functions are also unary (not binary), but may receive a variable number of arguments.

```

\__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
10688 \cs_new_nopar:Npn \__fp_parse_word_acot:N
10689 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_i:nn }
10690 \cs_new_nopar:Npn \__fp_parse_word_acotd:N
10691 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_ii:nn }
10692 \cs_new_nopar:Npn \__fp_parse_word_atan:N
10693 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_i:nn }
10694 \cs_new_nopar:Npn \__fp_parse_word_atand:N
10695 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_ii:nn }
10696 \cs_new_nopar:Npn \__fp_parse_word_max:N
10697 { \__fp_parse_function:NNN \__fp_minmax_o:Nw 2 }
10698 \cs_new_nopar:Npn \__fp_parse_word_min:N
10699 { \__fp_parse_function:NNN \__fp_minmax_o:Nw 0 }
(End definition for \__fp_parse_word_acot:N and others.)

```

Unary functions.

```

\__fp_parse_word_abs:N
\__fp_parse_word_exp:N
\__fp_parse_word_ln:N
10700 \cs_new:Npn \__fp_parse_word_abs:N
10701 { \__fp_parse_unary_function:nNN { set_sign } 0 }
10702 \cs_new_nopar:Npn \__fp_parse_word_exp:N
10703 { \__fp_parse_unary_function:nNN {exp} ? }
10704 \cs_new_nopar:Npn \__fp_parse_word_ln:N
10705 { \__fp_parse_unary_function:nNN {ln} ? }
(End definition for \__fp_parse_word_abs:N, \__fp_parse_word_exp:N, and \__fp_parse_word_ln:N.)

```

```

\__fp_parse_word_acos:N
\__fp_parse_word_acosd:N
\__fp_parse_word_acsc:N
\__fp_parse_word_acscd:N
\__fp_parse_word_asec:N
\__fp_parse_word_asecd:N
\__fp_parse_word_asin:N
\__fp_parse_word_asind:N
\__fp_parse_word_cos:N
\__fp_parse_word_cosd:N
\__fp_parse_word_cot:N
\__fp_parse_word_cotd:N
\__fp_parse_word_csc:N
\__fp_parse_word_cscd:N
\__fp_parse_word_sec:N
\__fp_parse_word_secd:N
\__fp_parse_word_sin:N
\__fp_parse_word_sind:N
\__fp_parse_word_tan:N
\__fp_parse_word_tand:N

```

Unary functions.

```

10706 \tl_map_inline:nn
10707 {
10708   {acos} {acsc} {asec} {asin}
10709   {cos} {cot} {csc} {sec} {sin} {tan}
10710 }
10711 {
10712   \cs_new_nopar:cpn { __fp_parse_word_#1:N }
10713     { \__fp_parse_unary_function:nnn {#1} \use_i:nn }
10714   \cs_new_nopar:cpn { __fp_parse_word_#1d:N }
10715     { \__fp_parse_unary_function:nnn {#1} \use_ii:nn }
10716 }

```

(End definition for __fp_parse_word_acos:N and others.)

This function expects one or two arguments.

```

10717 \cs_new:Npn \__fp_parse_word_round:N #1#2
10718 {
10719   \if_meaning:w + #2
10720     \__fp_parse_round:Nw \__fp_round_to_pinf:NNN
10721   \else:
10722     \if_meaning:w 0 #2
10723       \__fp_parse_round:Nw \__fp_round_to_zero:NNN
10724     \else:
10725       \if_meaning:w - #2
10726         \__fp_parse_round:Nw \__fp_round_to_ninf:NNN
10727       \fi:
10728     \fi:
10729   \fi:
10730   \exp_after:wN \__fp_parse_apply_round:NNwN
10731   \exp_after:wN #1
10732   \exp_after:wN \__fp_round_to_nearest:NNN
10733   \tex_romannumeral:D
10734   \__fp_parse_operand:Nw \c_sixteen \__fp_parse_expand:w #2
10735 }
10736 \cs_new:Npn \__fp_parse_round:Nw
10737   #1 #2 \__fp_round_to_nearest:NNN #3 \__fp_parse_expand:w #4
10738   { #2 #1 #3 \__fp_parse_expand:w }
10739 \cs_new:Npn \__fp_parse_apply_round:NNwN #1#2#3@#4
10740 {
10741   \if_case:w \__int_eval:w \__fp_array_count:n {#3} - \c_one \__int_eval_end:
10742     \__fp_round:Nwn #2 #3 {0} \tex_romannumeral:D
10743   \or: \__fp_round:Nww #2 #3 \tex_romannumeral:D
10744   \else:
10745     \__msg_kernel_expandable_error:nnnnn
10746       { kernel } { fp-num-args } { round() } { 1 } { 2 }
10747     \exp_after:wN \c_nan_fp \tex_romannumeral:D
10748     \fi:
10749     -'0 #4 #1
10750 }

```

(End definition for __fp_parse_word_round:N.)

26.6 Main functions

`__fp_parse:n` Start a `\romannumeral` 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. Then stop the initial expansion with `\c_zero`.

```

10751 \cs_new:Npn \__fp_parse:n #1
10752 {
10753   \tex_romannumeral:D
10754   \exp_after:wN \__fp_parse_after:ww
10755   \tex_romannumeral:D
10756   \__fp_parse_operand:Nw \c_minus_one
10757   \__fp_parse_expand:w #1 \s__fp_mark
10758   \s__fp_stop
10759 }
10760 \cs_new:Npn \__fp_parse_after:ww
10761   #1@ \__fp_parse_infix_end:N \s__fp_mark \s__fp_stop
10762 { \c_zero #1 }

```

(End definition for `__fp_parse:n`. This function is documented on page ??.)

`__fp_parse_operand:Nw` The `__fp_parse_operand` This is just a shorthand which sets up both `__fp_parse_continue` and `__fp_parse_one` with the same precedence. Note the trailing `\tex_romannumeral:D`. This function should be used with much care.

```

10763 \cs_new:Npn \__fp_parse_operand:Nw #1
10764 {
10765   -'0
10766   \exp_after:wN \__fp_parse_continue:NwN
10767   \exp_after:wN #1
10768   \tex_romannumeral:D -'0
10769   \exp_after:wN \__fp_parse_one:Nw
10770   \exp_after:wN #1
10771   \tex_romannumeral:D
10772 }
10773 \cs_new:Npn \__fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }

```

(End definition for `__fp_parse_operand:Nw`. This function is documented on page ??.)

`__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.

```

10774 \cs_new:Npn \__fp_parse_apply_binary:NwNwN #1 #2@ #3 #4@ #5
10775 {
10776   \exp_after:wN \__fp_parse_continue:NwN
10777   \exp_after:wN #1
10778   \tex_romannumeral:D -'0 \cs:w __fp_#3_o:ww \cs_end: #2 #4
10779   \tex_romannumeral:D -'0 #5 #1
10780 }

```

(End definition for `__fp_parse_apply_binary:NwNwN`.)

26.7 Infix operators

_fp_parse_infix_after_operand:NwN

```

10781 \cs_new:Npn \_fp_parse_infix_after_operand:NwN #1 #2;
10782 {
10783   \_fp_exp_after_f:nw { \_fp_parse_infix:NN #1 }
10784   #2;
10785 }
10786 \group_begin:
10787 \char_set_catcode_letter:N \*
10788 \cs_new:Npn \_fp_parse_infix:NN #1 #2
10789 {
10790   \if_catcode:w \scan_stop: \exp_not:N #2
10791   \if_int_compare:w
10792     \pdfTeX_strcmp:D { \s_fp_mark } { \exp_not:N #2 }
10793     = \c_zero
10794     \exp_after:wN \exp_after:wN
10795     \exp_after:wN \_fp_parse_infix_end:N
10796   \else:
10797     \exp_after:wN \exp_after:wN
10798     \exp_after:wN \_fp_parse_infix_juxtapose:N
10799   \fi:
10800   \else:
10801     \if_int_compare:w
10802       \_int_eval:w
10803       ( '#2 \if_int_compare:w '#2 > 'Z - \c_thirty_two \fi: )
10804       / 26
10805       = \c_three
10806       \exp_after:wN \exp_after:wN
10807       \exp_after:wN \_fp_parse_infix_juxtapose:N
10808     \else:
10809       \exp_after:wN \_fp_parse_infix_check:NNN
10810       \cs:w
10811       \_fp_parse_infix_#2:N
10812       \exp_after:wN \exp_after:wN \exp_after:wN
10813     \cs_end:
10814   \fi:
10815   \fi:
10816   #1
10817   #2
10818 }
10819 \cs_new:Npn \_fp_parse_infix_check:NNN #1#2#3
10820 {
10821   \if_meaning:w \scan_stop: #1
10822   \_msg_kernel_expandable_error:nnn
10823   { kernel } { fp-missing } { * }
10824   \exp_after:wN \_fp_parse_infix_*:N
10825   \exp_after:wN #2
10826   \exp_after:wN #3
10827   \else:

```

```

10828         \exp_after:wN #1
10829         \exp_after:wN #2
10830         \tex_romannumeral:D \exp_after:wN \_fp_parse_expand:w
10831     \fi:
10832 }
10833 \group_end:
(End definition for \_fp_parse_infix_after_operand:NwN.)

```

26.7.1 Closing parentheses and commas

`_fp_parse_infix_end:N` This one is a little bit odd: force every previous operator to end, regardless of the precedence.

```

10834 \cs_new:Npn \_fp_parse_infix_end:N #1
10835 { @ \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.

```

10836 \group_begin:
10837 \char_set_catcode_letter:N \)
10838 \cs_new:Npn \_fp_parse_infix_):N #1
10839 {
10840     \if_int_compare:w #1 < \c_zero
10841     \_msg_kernel_expandable_error:nnn { kernel } { fp-extra } { ) }
10842     \exp_after:wN \_fp_parse_infix:NN
10843     \exp_after:wN #1
10844     \tex_romannumeral:D \exp_after:wN \_fp_parse_expand:w
10845     \else:
10846     \exp_after:wN @
10847     \exp_after:wN \use_none:n
10848     \exp_after:wN \_fp_parse_infix_):N
10849     \fi:
10850 }
10851 \group_end:
(End definition for \_fp_parse_infix_):N.)

```

```

\_fp_parse_infix_
:N
10852 \group_begin:
10853 \char_set_catcode_letter:N \,
10854 \cs_new:Npn \_fp_parse_infix_,:N #1
10855 {
10856     \if_int_compare:w #1 > \c_one
10857     \exp_after:wN @
10858     \exp_after:wN \use_none:n
10859     \exp_after:wN \_fp_parse_infix_,:N
10860     \else:
10861     \if_int_compare:w #1 = \c_one
10862     \exp_after:wN \_fp_parse_infix_comma:w
10863     \tex_romannumeral:D

```

```

10864         \else:
10865             \exp_after:wN \__fp_parse_infix_comma_gobble:w
10866             \tex_romannumeral:D
10867         \fi:
10868         \__fp_parse_operand:Nw \c_one
10869         \exp_after:wN \__fp_parse_expand:w
10870     \fi:
10871 }
10872 \cs_new:Npn \__fp_parse_infix_comma:w #1 @
10873 { #1 @ \use_none:n }
10874 \cs_new:Npn \__fp_parse_infix_comma_gobble:w #1 @
10875 {
10876     \__msg_kernel_expandable_error:nn { kernel } { fp-extra-comma }
10877     @ \use_none:n
10878 }
10879 \group_end:

```

(End definition for __fp_parse_infix_ and :N.)

26.7.2 Usual infix operators

As described in the “work plan”, each infix operator has an associated `\infix` function, a computing function, and precedence, given as arguments to `__fp_tmp:w`. Using the general mechanism for arithmetic operations. The power operation must be associative in the opposite order from all others. For this, we use two distinct precedences.

```

\__fp_parse_infix_+:N
\__fp_parse_infix_-:N
\__fp_parse_infix_/:N
\__fp_parse_infix_mul:N
\__fp_parse_infix_and:N
\__fp_parse_infix_or:N
\__fp_parse_infix^:N
10880 \group_begin:
10881     \char_set_catcode_other:N \&
10882     \char_set_catcode_letter:N \^
10883     \char_set_catcode_letter:N \ /
10884     \char_set_catcode_letter:N \-
10885     \char_set_catcode_letter:N \+
10886     \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
10887     {
10888         \cs_new:Npn #1 ##1
10889         {
10890             \if_int_compare:w ##1 < #3
10891                 \exp_after:wN @
10892                 \exp_after:wN \__fp_parse_apply_binary:NwNwN
10893                 \exp_after:wN #2
10894                 \tex_romannumeral:D
10895                 \__fp_parse_operand:Nw #4
10896                 \exp_after:wN \__fp_parse_expand:w
10897             \else:
10898                 \exp_after:wN @
10899                 \exp_after:wN \use_none:n
10900                 \exp_after:wN #1
10901             \fi:
10902         }
10903     }
10904     \__fp_tmp:w \__fp_parse_infix^:N ^ \c_fifteen \c_fourteen

```

```

10905 \__fp_tmp:w \__fp_parse_infix_/:N / \c_ten \c_ten
10906 \__fp_tmp:w \__fp_parse_infix_mul:N * \c_ten \c_ten
10907 \__fp_tmp:w \__fp_parse_infix -:N - \c_nine \c_nine
10908 \__fp_tmp:w \__fp_parse_infix +:N + \c_nine \c_nine
10909 \__fp_tmp:w \__fp_parse_infix_and:N & \c_five \c_five
10910 \__fp_tmp:w \__fp_parse_infix_or:N | \c_four \c_four
10911 \group_end:

```

(End definition for `__fp_parse_infix_+:N` and others. These functions are documented on page ??.)

26.7.3 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`.

```

10912 \cs_new:cpn { \__fp_parse_infix_(:N } #1
10913 { \__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:NwwN` rather than directly calling `__fp_parse_apply_binary:NwNwN`. This lets us catch errors such as `max(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.

```

10914 \cs_new:Npn \__fp_parse_infix_juxtapose:N #1
10915 {
10916   \if_int_compare:w #1 < \c_thirty_two
10917     \exp_after:wN @
10918     \exp_after:wN \__fp_parse_apply_juxtapose:NwwN
10919     \tex_romannumeral:D
10920     \__fp_parse_operand:Nw \c_thirty_two
10921     \exp_after:wN \__fp_parse_expand:w
10922   \else:
10923     \exp_after:wN @
10924     \exp_after:wN \use_none:n
10925     \exp_after:wN \__fp_parse_infix_juxtapose:N
10926   \fi:
10927 }
10928 \cs_new:Npn \__fp_parse_apply_juxtapose:NwwN #1 #2;#3@ #4;#5@
10929 {
10930   \if_catcode:w ^ \tl_to_str:n { #3 #5 } ^
10931   \else:
10932     \__fp_error:nffn { invalid-ii }
10933     { \__fp_array_to_clist:n { #2; #3 } }
10934     { \__fp_array_to_clist:n { #4; #5 } }
10935     { }
10936   \fi:
10937   \__fp_parse_apply_binary:NwNwN #1 #2;@ * #4;@
10938 }

```

(End definition for `__fp_parse_infix_juxtapose:N` and `__fp_parse_apply_juxtapose:NwwN`.)

26.7.4 Multi-character cases

`__fp_parse_infix_*:N`

```

10939 \group_begin:
10940   \char_set_catcode_letter:N ^
10941   \cs_new:cpn { __fp_parse_infix_*:N } #1#2
10942   {
10943     \if:w * \exp_not:N #2
10944       \exp_after:wN \__fp_parse_infix_^:N
10945       \exp_after:wN #1
10946     \else:
10947       \exp_after:wN \__fp_parse_infix_mul:N
10948       \exp_after:wN #1
10949       \exp_after:wN #2
10950     \fi:
10951   }
10952 \group_end:

```

(End definition for `__fp_parse_infix_*:N`.)

`__fp_parse_infix_|:Nw`

`__fp_parse_infix_&:Nw`

```

10953 \group_begin:
10954   \char_set_catcode_letter:N \
10955   \char_set_catcode_letter:N &
10956   \cs_new:Npn \__fp_parse_infix_|:N #1#2
10957   {
10958     \if:w | \exp_not:N #2
10959       \exp_after:wN \__fp_parse_infix_|:N
10960       \exp_after:wN #1
10961       \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
10962     \else:
10963       \exp_after:wN \__fp_parse_infix_or:N
10964       \exp_after:wN #1
10965       \exp_after:wN #2
10966     \fi:
10967   }
10968   \cs_new:Npn \__fp_parse_infix_&:N #1#2
10969   {
10970     \if:w & \exp_not:N #2
10971       \exp_after:wN \__fp_parse_infix_&:N
10972       \exp_after:wN #1
10973       \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
10974     \else:
10975       \exp_after:wN \__fp_parse_infix_and:N
10976       \exp_after:wN #1
10977       \exp_after:wN #2
10978     \fi:
10979   }
10980 \group_end:

```

(End definition for `__fp_parse_infix_|:Nw`. This function is documented on page ??.)

26.7.5 Ternary operator

```

\__fp_parse_infix_?:N
\__fp_parse_infix_::N
10981 \group_begin:
10982 \char_set_catcode_letter:N \?
10983 \cs_new:Npn \__fp_parse_infix_?:N #1
10984 {
10985   \if_int_compare:w #1 < \c_three
10986     \exp_after:wN @
10987     \exp_after:wN \__fp_ternary:NwwN
10988     \tex_romannumeral:D
10989     \__fp_parse_operand:Nw \c_three
10990     \exp_after:wN \__fp_parse_expand:w
10991   \else:
10992     \exp_after:wN @
10993     \exp_after:wN \use_none:n
10994     \exp_after:wN \__fp_parse_infix_?:N
10995   \fi:
10996 }
10997 \cs_new:Npn \__fp_parse_infix_::N #1
10998 {
10999   \if_int_compare:w #1 < \c_three
11000     \__msg_kernel_expandable_error:nnnn
11001     { kernel } { fp-missing } { ? } { ~for~?: }
11002     \exp_after:wN @
11003     \exp_after:wN \__fp_ternary_auxii:NwwN
11004     \tex_romannumeral:D
11005     \__fp_parse_operand:Nw \c_two
11006     \exp_after:wN \__fp_parse_expand:w
11007   \else:
11008     \exp_after:wN @
11009     \exp_after:wN \use_none:n
11010     \exp_after:wN \__fp_parse_infix_::N
11011   \fi:
11012 }
11013 \group_end:
(End definition for \__fp_parse_infix_?:N and \__fp_parse_infix_::N.)

```

26.7.6 Comparisons

```

\__fp_parse_infix_<:N
\__fp_parse_infix_=:N
\__fp_parse_infix_>:N
\__fp_parse_infix_!:N
\__fp_parse_infix_excl_aux:NN
\__fp_parse_infix_excl_error:
\__fp_infix_compare:N
\__fp_parse_compare:NNNNNNw
\__fp_parse_compare_expand:NNNNw
\__fp_parse_compare_end:NNNN
\__fp_compare:wNNNNw
11014 \cs_new:cpn { __fp_parse_infix_<:N } #1
11015 {
11016   \__fp_infix_compare:N #1 \c_one_fp
11017   \c_zero_fp \c_zero_fp \c_zero_fp \c_zero_fp <
11018 }
11019 \cs_new:cpn { __fp_parse_infix_=:N } #1
11020 {
11021   \__fp_infix_compare:N #1 \c_one_fp

```

```

11022     \c_zero_fp \c_zero_fp \c_zero_fp \c_zero_fp =
11023   }
11024   \cs_new:cpn { __fp_parse_infix_>:N } #1
11025   {
11026     \__fp_infix_compare:N #1 \c_one_fp
11027     \c_zero_fp \c_zero_fp \c_zero_fp \c_zero_fp >
11028   }
11029   \cs_new:cpn { __fp_parse_infix_!:N } #1
11030   {
11031     \exp_after:wN \__fp_parse_infix_excl_aux:NN
11032     \exp_after:wN #1 \tex_romannumeral:D \__fp_parse_expand:w
11033   }
11034   \cs_new:Npn \__fp_parse_infix_excl_aux:NN #1#2
11035   {
11036     \__fp_infix_compare:N #1 \c_zero_fp
11037     \c_one_fp \c_one_fp \c_one_fp \c_one_fp #2
11038   }
11039   \cs_new:Npn \__fp_parse_infix_excl_error:
11040   {
11041     \_msg_kernel_expandable_error:nnnn
11042     { kernel } { fp-missing } { = } { ~after~!. }
11043   }
11044   \cs_new:Npn \__fp_infix_compare:N #1
11045   {
11046     \if_int_compare:w #1 < \c_seven
11047       \exp_after:wN \__fp_parse_compare:NNNNNNw
11048       \exp_after:wN \__fp_parse_infix_excl_error:
11049     \else:
11050       \exp_after:wN @
11051       \exp_after:wN \use_none:n
11052       \exp_after:wN \__fp_infix_compare:N
11053     \fi:
11054   }
11055   \cs_new:Npn \__fp_parse_compare:NNNNNNw #1#2#3#4#5#6#7
11056   {
11057     \if_case:w
11058       \if_catcode:w \scan_stop: #7
11059       \c_minus_one
11060     \else:
11061       \__int_eval:w '#7 - '< \__int_eval_end:
11062     \fi:
11063     \__fp_parse_compare_expand:NNNNNw #2#2#4#5#6
11064     \or: \__fp_parse_compare_expand:NNNNNw #2#3#2#5#6
11065     \or: \__fp_parse_compare_expand:NNNNNw #2#3#4#2#6
11066     \or: \__fp_parse_compare_expand:NNNNNw #2#3#4#5#2
11067     \else: #1 \__fp_parse_compare_end:NNNN #3#4#5#6#7
11068     \fi:
11069   }
11070   \cs_new:Npn \__fp_parse_compare_expand:NNNNNw #1#2#3#4#5
11071   {

```

```

11072 \exp_after:wN \__fp_parse_compare:NNNNNNw
11073 \exp_after:wN \prg_do_nothing:
11074 \exp_after:wN #1
11075 \exp_after:wN #2
11076 \exp_after:wN #3
11077 \exp_after:wN #4
11078 \exp_after:wN #5
11079 \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
11080 }
11081 \cs_new:Npn \__fp_parse_compare_end:NNNN #1#2#3#4#5 \fi:
11082 {
11083   \fi:
11084   \exp_after:wN @
11085   \exp_after:wN \__fp_parse_apply_compare:NwNNNNwN
11086   \exp_after:wN #1
11087   \exp_after:wN #2
11088   \exp_after:wN #3
11089   \exp_after:wN #4
11090   \tex_romannumeral:D
11091   \__fp_parse_operand:Nw \c_seven \__fp_parse_expand:w #5
11092 }
11093 \cs_new:Npn \__fp_parse_apply_compare:NwNNNNwN #1 #2@ #3#4#5#6 #7@ #8
11094 {
11095   \exp_after:wN \__fp_parse_continue:NwN
11096   \exp_after:wN #1
11097   \tex_romannumeral:D -'0
11098   \exp_after:wN \exp_after:wN
11099   \exp_after:wN \exp_after:wN
11100   \exp_after:wN \exp_after:wN
11101   \if_case:w \__fp_compare_back:ww #7 #2 \exp_stop_f:
11102     #4
11103     \or: #5
11104     \or: #6
11105     \else: #3
11106     \fi:
11107   \tex_romannumeral:D -'0 #8 #1
11108 }

```

(End definition for __fp_parse_infix_<:N and others. These functions are documented on page ??.)

26.8 Messages

```

11109 \__msg_kernel_new:nnn { kernel } { unknown-fp-word }
11110 { Unknown~fp~word~#1. }
11111 \__msg_kernel_new:nnn { kernel } { fp-missing }
11112 { Missing~#1~inserted #2. }
11113 \__msg_kernel_new:nnn { kernel } { fp-extra }
11114 { Extra~#1~ignored. }
11115 \__msg_kernel_new:nnn { kernel } { fp-early-end }
11116 { Premature~end~in~fp~expression. }

```



```

11117 \_msg_kernel_new:nnn { kernel } { fp-after-e }
11118 { Cannot~use~#1 after~'e'. }
11119 \_msg_kernel_new:nnn { kernel } { fp-missing-number }
11120 { Missing~number~before~'#1'. }
11121 \_msg_kernel_new:nnn { kernel } { fp-unknown-symbol }
11122 { Unknown~symbol~#1~ignored. }
11123 \_msg_kernel_new:nnn { kernel } { fp-extra-comma }
11124 { Unexpected~comma:~extra~arguments~ignored. }
11125 \_msg_kernel_new:nnn { kernel } { fp-num-args }
11126 { #1~expects~between~#2~and~#3~arguments. }
11127 </initex | package>

```

27 l3fp-logic Implementation

```

11128 <*initex | package>
11129 <@@=fp>

```

27.1 Syntax of internal functions

- `_fp_compare_npos:nwnw` { $\langle exp_1 \rangle$ } $\langle body_1 \rangle$; { $\langle exp_2 \rangle$ } $\langle body_2 \rangle$;
- `_fp_minmax_o:Nw` $\langle sign \rangle$ $\langle floating\ point\ array \rangle$
- `_fp_not_o:w ?` $\langle floating\ point\ array \rangle$ (with one floating point number only)
- `_fp_&_o:ww` $\langle floating\ point \rangle$ $\langle floating\ point \rangle$
- `_fp_|_o:ww` $\langle floating\ point \rangle$ $\langle floating\ point \rangle$
- `_fp_ternary:NwwN`, `_fp_ternary_auxi:NwwN`, `_fp_ternary_auxii:NwwN`
have to be understood.

27.2 Existence test

`\fp_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.
`\fp_if_exist_p:c` 11130 `\prg_new_eq_conditional:NNn \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }`
`\fp_if_exist:N \underline{TF}` 11131 `\prg_new_eq_conditional:NNn \fp_if_exist:c \cs_if_exist:c { TF , T , F , p }`
`\fp_if_exist:c \underline{TF}` (End definition for `\fp_if_exist:N` and `\fp_if_exist:c`. These functions are documented on page ??.)

27.3 Comparison

`\fp_compare_p:n` Within floating point expressions, comparison operators are treated as operations, so we
`\fp_compare:n \underline{TF}` evaluate #1, then compare with 0.
`_fp_compare_return:w` 11132 `\prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }`
11133 {
11134 `\exp_after:wN _fp_compare_return:w`
11135 `\tex_romannumeral:D -'0 _fp_parse:n {#1}`
11136 }
11137 `\cs_new:Npn _fp_compare_return:w \s__fp _fp_chk:w #1#2;`

```

11138 {
11139   \if_meaning:w 0 #1
11140   \prg_return_false:
11141   \else:
11142     \prg_return_true:
11143   \fi:
11144 }

```

(End definition for `\fp_compare:n`. These functions are documented on page ??.)

`\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 $?$.
`__fp_compare_aux:wn`

```

11145 \prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
11146 {
11147   \if_int_compare:w
11148     \exp_after:wN \__fp_compare_aux:wn
11149     \tex_romannumeral:D -‘0 \__fp_parse:n {#1} {#3}
11150     = \__int_eval:w ‘#2 - ‘= \__int_eval_end:
11151   \prg_return_true:
11152   \else:
11153     \prg_return_false:
11154   \fi:
11155 }
11156 \cs_new:Npn \__fp_compare_aux:wn #1; #2
11157 {
11158   \exp_after:wN \__fp_compare_back:ww
11159   \tex_romannumeral:D -‘0 \__fp_parse:n {#2} #1;
11160 }

```

(End definition for `\fp_compare:nNn`. These functions are documented on page 174.)

`__fp_compare_back:ww`
`__fp_compare_nan:w`

`__fp_compare_back:ww` $\langle y \rangle$; $\langle x \rangle$;
 Expands (in the same way as `\int_eval:n`) to -1 if $x < y$, 0 if $x = y$, 1 if $x > y$, and 2 otherwise (denoted as $x?y$). If either operand is `nan`, stop the comparison with `__fp_compare_nan:w` returning 2 . If x is negative, swap the outputs 1 and -1 (i.e., $>$ and $<$); we can henceforth assume that $x \geq 0$. If $y \geq 0$, and they have the same type, either they are normal and we compare them with `__fp_compare_npos:nwnw`, or they are equal. If $y \geq 0$, but of a different type, the highest type is a larger number. Finally, if $y \leq 0$, then $x > y$, unless both are zero.

```

11161 \cs_new:Npn \__fp_compare_back:ww
11162   \s__fp \__fp_chk:w #1 #2 #3;
11163   \s__fp \__fp_chk:w #4 #5 #6;
11164 {
11165   \__int_value:w
11166   \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
11167   \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
11168   \if_meaning:w 2 #5 - \fi:
11169   \if_meaning:w #2 #5
11170   \if_meaning:w #1 #4

```

```

11171         \if_meaning:w 1 #1
11172         \__fp_compare_npos:nwnw #6; #3;
11173         \else:
11174             0
11175         \fi:
11176     \else:
11177         \if_int_compare:w #4 < #1 - \fi: 1
11178     \fi:
11179 \else:
11180     \if_int_compare:w #1#4 = \c_zero
11181     0
11182 \else:
11183     1
11184 \fi:
11185 \fi:
11186 \exp_stop_f:
11187 }
11188 \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 {<expo1>} <body1>} {<expo2>} <body2>} ;`
`__fp_compare_significand:nnnnnnnn` Within an `__int_value:w ... \exp_stop_f:` construction, this expands to 0 if

the two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```

11189 \cs_new:Npn \__fp_compare_npos:nwnw #1#2; #3#4;
11190 {
11191     \if_int_compare:w #1 = #3 \exp_stop_f:
11192     \__fp_compare_significand:nnnnnnnn #2 #4
11193 \else:
11194     \if_int_compare:w #1 < #3 - \fi: 1
11195 \fi:
11196 }
11197 \cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
11198 {
11199     \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
11200     \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
11201     0
11202 \else:
11203     \if_int_compare:w #3#4 < #7#8 - \fi: 1
11204 \fi:
11205 \else:
11206     \if_int_compare:w #1#2 < #5#6 - \fi: 1
11207 \fi:
11208 }

```

(End definition for `__fp_compare_npos:nwnw`. This function is documented on page ??.)

27.4 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.

```

11209 \cs_new:Npn \fp_do_until:nn #1#2
11210 {
11211     #2
11212     \fp_compare:nF {#1}
11213     { \fp_do_until:nn {#1} {#2} }
11214 }
11215 \cs_new:Npn \fp_do_while:nn #1#2
11216 {
11217     #2
11218     \fp_compare:nT {#1}
11219     { \fp_do_while:nn {#1} {#2} }
11220 }
11221 \cs_new:Npn \fp_until_do:nn #1#2
11222 {
11223     \fp_compare:nF {#1}
11224     {
11225         #2
11226         \fp_until_do:nn {#1} {#2}
11227     }
11228 }
11229 \cs_new:Npn \fp_while_do:nn #1#2
11230 {
11231     \fp_compare:nT {#1}
11232     {
11233         #2
11234         \fp_while_do:nn {#1} {#2}
11235     }
11236 }

```

(End definition for `\fp_do_until:nn` and others. These functions are documented on page 175.)

`\fp_do_until:nNnn` As above but not using the `nNn` syntax.

```

11237 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
11238 {
11239     #4
11240     \fp_compare:nNnF {#1} #2 {#3}
11241     { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
11242 }
11243 \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
11244 {
11245     #4
11246     \fp_compare:nNnT {#1} #2 {#3}
11247     { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
11248 }
11249 \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
11250 {

```

```

11251     \fp_compare:nNnF {#1} #2 {#3}
11252     {
11253         #4
11254         \fp_until_do:nNnn {#1} #2 {#3} {#4}
11255     }
11256 }
11257 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
11258 {
11259     \fp_compare:nNnT {#1} #2 {#3}
11260     {
11261         #4
11262         \fp_while_do:nNnn {#1} #2 {#3} {#4}
11263     }
11264 }

```

(End definition for `\fp_do_until:nNnn` and others. These functions are documented on page 175.)

27.5 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`.

```

11265 \cs_new:Npn \__fp_minmax_o:Nw #1#2 @
11266 {
11267     \if_meaning:w 0 #1
11268     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN \c_one
11269     \else:
11270     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN \c_minus_one
11271     \fi:
11272     #2
11273     \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
11274     \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
11275 }

```

(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.

```

11276 \cs_new:Npn \__fp_minmax_loop:Nww
11277     #1 \s__fp \__fp_chk:w #2#3; \s__fp \__fp_chk:w #4#5;
11278 {

```

```

11279 \if_meaning:w 3 #4
11280 \if_meaning:w 3 #2
11281 \__fp_minmax_auxi:ww
11282 \else:
11283 \__fp_minmax_auxii:ww
11284 \fi:
11285 \else:
11286 \if_int_compare:w
11287 \__fp_compare_back:ww
11288 \s__fp \__fp_chk:w #4#5;
11289 \s__fp \__fp_chk:w #2#3;
11290 = #1
11291 \__fp_minmax_auxii:ww
11292 \else:
11293 \__fp_minmax_auxi:ww
11294 \fi:
11295 \fi:
11296 \__fp_minmax_loop:Nww #1
11297 \s__fp \__fp_chk:w #2#3;
11298 \s__fp \__fp_chk:w #4#5;
11299 }
(End definition for \__fp_minmax_loop:Nww.)

```

__fp_minmax_auxi:ww
__fp_minmax_auxii:ww

Keep the first/second number, and remove the other.

```

11300 \cs_new:Npn \__fp_minmax_auxi:ww #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
11301 { \fi: \fi: #2 \s__fp #3 ; }
11302 \cs_new:Npn \__fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
11303 { \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.

```

11304 \cs_new:Npn \__fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3; #4;
11305 { \fi: \__fp_exp_after_o:w \s__fp #3; }
(End definition for \__fp_minmax_break_o:w.)

```

27.6 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.

```

11306 \cs_new:cpn { __fp_not_o:w } #1 \s__fp \__fp_chk:w #2#3; @
11307 {
11308 \if_meaning:w 0 #2
11309 \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
11310 \else:
11311 \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
11312 \fi:
11313 }

```

(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.

```

11314 \group_begin:
11315 \char_set_catcode_letter:N &
11316 \char_set_catcode_letter:N |
11317 \cs_new:Npn \_fp_&o:ww #1 \s_fp \_fp_chk:w #2#3;
11318 {
11319     \if_meaning:w 0 #2 #1
11320     \_fp_and_return:wNw \s_fp \_fp_chk:w #2#3;
11321     \fi:
11322     \_fp_exp_after_o:w
11323 }
11324 \cs_new_nopar:Npn \_fp_|o:ww { \_fp_&o:ww \else: }
11325 \group_end:
11326 \cs_new:Npn \_fp_and_return:wNw #1; \fi: #2#3; { \fi: #2 #1; }

```

(End definition for _fp_&o:ww. This function is documented on page ??.)

27.7 Ternary operator

_fp_ternary:NwwN The first function receives the test and the true branch of the ?: ternary operator. It
_fp_ternary_auxi:NwwN returns the true branch, unless the test branch is zero. In that case, the function returns
_fp_ternary_auxii:NwwN a very specific nan. The second function receives the output of the first function, and the
_fp_ternary_loop_break:w false branch. It returns the previous input, unless that is the special nan, in which case
_fp_ternary_loop:Nw we return the false branch.

```

11327 \cs_new:Npn \_fp_ternary:NwwN #1 #2@ #3@ #4
11328 {
11329     \if_meaning:w \_fp_parse_infix_:N #4
11330     \_fp_ternary_loop:Nw
11331     #2
11332     \s_fp \_fp_chk:w { \_fp_ternary_loop_break:w } ;
11333     \_fp_ternary_break_point:n { \exp_after:wN \_fp_ternary_auxi:NwwN }
11334     \exp_after:wN #1
11335     \tex_romannumeral:D -‘0
11336     \_fp_exp_after_array_f:w #3 \s_fp_stop
11337     \exp_after:wN @
11338     \tex_romannumeral:D
11339     \_fp_parse_operand:Nw \c_two
11340     \_fp_parse_expand:w
11341     \else:
11342     \_msg_kernel_expandable_error:nnnn
11343     { kernel } { fp-missing } { : } { ~for~?: }
11344     \exp_after:wN \_fp_parse_continue:NwN
11345     \exp_after:wN #1

```

```

11346         \tex_romannumeral:D -‘0
11347         \__fp_exp_after_array_f:w #3 \s__fp_stop
11348         \exp_after:wN #4
11349         \exp_after:wN #1
11350     \fi:
11351 }
11352 \cs_new:Npn \__fp_ternary_loop_break:w #1 \fi: #2 \__fp_ternary_break_point:n #3
11353 {
11354     \c_zero = \c_zero \fi:
11355     \exp_after:wN \__fp_ternary_auxii:NwwN
11356 }
11357 \cs_new:Npn \__fp_ternary_loop:Nw \s__fp \__fp_chk:w #1#2;
11358 {
11359     \if_int_compare:w #1 > \c_zero
11360         \exp_after:wN \__fp_ternary_map_break:
11361         \fi:
11362     \__fp_ternary_loop:Nw
11363 }
11364 \cs_new:Npn \__fp_ternary_map_break: #1 \__fp_ternary_break_point:n #2 {#2}
11365 \cs_new:Npn \__fp_ternary_auxi:NwwN #1#2@#3@#4
11366 {
11367     \exp_after:wN \__fp_parse_continue:NwN
11368     \exp_after:wN #1
11369     \tex_romannumeral:D -‘0
11370     \__fp_exp_after_array_f:w #2 \s__fp_stop
11371     #4 #1
11372 }
11373 \cs_new:Npn \__fp_ternary_auxii:NwwN #1#2@#3@#4
11374 {
11375     \exp_after:wN \__fp_parse_continue:NwN
11376     \exp_after:wN #1
11377     \tex_romannumeral:D -‘0
11378     \__fp_exp_after_array_f:w #3 \s__fp_stop
11379     #4 #1
11380 }

```

(End definition for __fp_ternary:NwwN, __fp_ternary_auxi:NwwN, and __fp_ternary_auxii:NwwN.
These functions are documented on page ??.)

```

11381 </initex | package>

```

28 l3fp-basics Implementation

```

11382 <*initex | package>
11383 <@@=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>.

28.1 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}`.

```

11384 \cs_new:Npn \__fp_basics_pack_low:NNNNNw #1 #2#3#4#5 #6;
11385 {
11386   \if_meaning:w 2 #1
11387     + \c_one
11388   \fi:
11389   ; {#2#3#4#5} {#6} ;
11390 }
11391 \cs_new:Npn \__fp_basics_pack_high:NNNNNw #1 #2#3#4#5 #6;
11392 {
11393   \if_meaning:w 2 #1
11394     \__fp_basics_pack_high_carry:w
11395   \fi:
11396   ; {#2#3#4#5} {#6}
11397 }
11398 \cs_new:Npn \__fp_basics_pack_high_carry:w \fi: ; #1
11399 { \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.

```

11400 \cs_new:Npn \__fp_basics_pack_weird_low:NNNNw #1 #2#3#4 #5;
11401 {
11402   \if_meaning:w 2 #1
11403     + \c_one
11404   \fi:
11405   \__int_eval_end:
11406   #2#3#4; {#5} ;
11407 }
11408 \cs_new:Npn \__fp_basics_pack_weird_high:NNNNNNNNw
11409   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`.)

28.2 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.

28.2.1 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.

```

11410 \cs_new_nopar:cpx { __fp_-_o:ww } \s__fp
11411 {
11412   \exp_not:c { __fp+_o:ww }
11413   \exp_not:n { \s__fp \__fp_neg_sign:N }
11414 }

```

(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.

```

11415 \cs_new:cpn { __fp+_o:ww }
11416   \s__fp #1 \__fp_chk:w #2 #3 ; \s__fp \__fp_chk:w #4 #5
11417   {
11418     \if_case:w
11419       \if_meaning:w #2 #4
11420       #2 \exp_stop_f:
11421     \else:
11422       \if_int_compare:w #2 > #4 \exp_stop_f:
11423       \c_three
11424     \else:
11425       \c_minus_one
11426     \fi:
11427   \fi:
11428     \exp_after:wN \__fp_add_zeros_o:Nww \__int_value:w
11429   \or:   \exp_after:wN \__fp_add_normal_o:Nww \__int_value:w
11430   \or:   \exp_after:wN \__fp_add_inf_o:Nww \__int_value:w
11431   \or:   \__fp_case_return_i_o:ww
11432   \else: \exp_after:wN \__fp_add_return_ii_o:Nww \__int_value:w
11433   \fi:
11434   #1 #5
11435   \s__fp \__fp_chk:w #2 #3 ;
11436   \s__fp \__fp_chk:w #4 #5
11437 }

```

(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.

```

11438 \cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
11439 { \__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.

```

11440 \cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
11441 {
11442   \if_int_compare:w #2 #1 = 20 \exp_stop_f:
11443   \exp_after:wN \__fp_add_return_ii_o:Nww
11444 \else:
11445   \__fp_case_return_i_o:ww
11446 \fi:
11447 #1
11448 \s__fp \__fp_chk:w 0 #2
11449 }

```

(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.

```

11450 \cs_new:Npn \__fp_add_inf_o:Nww
11451   #1 \s__fp \__fp_chk:w 2 #2 #3; \s__fp \__fp_chk:w 2 #4

```

```

11452 {
11453   \if_meaning:w #1 #2
11454   \__fp_case_return_i_o:ww
11455   \else:
11456   \__fp_case_use:nw
11457   {
11458     \if_meaning:w #1 #4
11459     \exp_after:wN \__fp_invalid_operation_o:Nww
11460     \exp_after:wN +
11461     \else:
11462     \exp_after:wN \__fp_invalid_operation_o:Nww
11463     \exp_after:wN -
11464     \fi:
11465   }
11466   \fi:
11467   \s__fp \__fp_chk:w 2 #2 #3;
11468   \s__fp \__fp_chk:w 2 #4
11469 }
(End definition for \__fp_add_inf_o:Nww.)

```

```

\__fp_add_normal_o:Nww \__fp_add_normal_o:Nww <sign2> \s__fp \__fp_chk:w 1 <sign1> <exp1>
<body1> ; \s__fp \__fp_chk:w 1 <initial sign2> <exp2> <body2> ;

```

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```

11470 \cs_new:Npn \__fp_add_normal_o:Nww #1 \s__fp \__fp_chk:w 1 #2
11471 {
11472   \if_meaning:w #1#2
11473   \exp_after:wN \__fp_add_npos_o:NnwNnw
11474   \else:
11475   \exp_after:wN \__fp_sub_npos_o:NnwNnw
11476   \fi:
11477   #2
11478 }
(End definition for \__fp_add_normal_o:Nww.)

```

28.2.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

```

\__fp_add_npos_o:NnwNnw \__fp_add_npos_o:NnwNnw <sign1> <exp1> <body1> ; \s__fp \__fp_chk:w 1
<initial sign2> <exp2> <body2> ;

```

Since we are doing an addition, the final sign is $\langle sign_1 \rangle$. Start an $__int_eval:w$, responsible for computing the exponent: the result, and the $\langle final\ sign \rangle$ are then given to $__fp_sanitize:Nw$ which checks for overflow. The exponent is computed as the largest exponent #2 or #5, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by $__fp_add_big_i:wNww$ or $__fp_add_big_ii:wNww$. We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```

11479 \cs_new:Npn \__fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp \__fp_chk:w 1 #4 #5
11480 {
11481   \exp_after:wN \__fp_sanitizew:Nw
11482   \exp_after:wN #1
11483   \int_use:N \__int_eval:w
11484   \if_int_compare:w #2 > #5 \exp_stop_f:
11485     #2
11486     \exp_after:wN \__fp_add_big_i_o:wNww \__int_value:w -
11487   \else:
11488     #5
11489     \exp_after:wN \__fp_add_big_ii_o:wNww \__int_value:w
11490   \fi:
11491   \__int_eval:w #5 - #2 ; #1 #3;
11492 }

```

(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.

```

11493 \cs_new:Npn \__fp_add_big_i_o:wNww #1; #2 #3; #4;
11494 {
11495   \__fp_decimate:nNnnnn {#1}
11496   \__fp_add_significand_o:NnnwnnnnN
11497   #4
11498   #3
11499   #2
11500 }
11501 \cs_new:Npn \__fp_add_big_ii_o:wNww #1; #2 #3; #4;
11502 {
11503   \__fp_decimate:nNnnnn {#1}
11504   \__fp_add_significand_o:NnnwnnnnN
11505   #3
11506   #4
11507   #2
11508 }

```

(End definition for __fp_add_big_i_o:wNww. This function is documented on page ??.)

__fp_add_significand_o:NnnwnnnnN __fp_add_significand_o:NnnwnnnnN $\langle rounding\ digit \rangle$ $\{ \langle Y'_1 \rangle \}$ $\{ \langle Y'_2 \rangle \}$ $\langle extra-digits \rangle$
 __fp_add_significand_pack:NNNNNNN ; $\{ \langle X_1 \rangle \}$ $\{ \langle X_2 \rangle \}$ $\{ \langle X_3 \rangle \}$ $\{ \langle X_4 \rangle \}$ $\langle final\ sign \rangle$
 __fp_add_significand_test_o:N To round properly, we must know at which digit the rounding should occur. This

requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as $0.99 \cdots 95 \rightarrow 1.00 \cdots 0$, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```

11509 \cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
11510 {
11511   \exp_after:wN \__fp_add_significand_test_o:N
11512   \int_use:N \__int_eval:w 1#5#6 + #2

```

```

11513     \exp_after:wN \__fp_add_significand_pack:NNNNNNN
11514     \int_use:N \__int_eval:w 1#7#8 + #3 ; #1
11515 }
11516 \cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
11517 {
11518     \if_meaning:w 2 #1
11519     + \c_one
11520     \fi:
11521     ; #2 #3 #4 #5 #6 #7 ;
11522 }
11523 \cs_new:Npn \__fp_add_significand_test_o:N #1
11524 {
11525     \if_meaning:w 2 #1
11526     \exp_after:wN \__fp_add_significand_carry_o:wwwNN
11527     \else:
11528     \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
11529     \fi:
11530 }

```

(End definition for `__fp_add_significand_o:NnnwnnnN`. This function is documented on page ??.)

`__fp_add_significand_no_carry_o:wwwNN` `__fp_add_significand_no_carry_o:wwwNN` $\langle 8d \rangle$; $\langle 6d \rangle$; $\langle 2d \rangle$; $\langle \text{rounding digit} \rangle$ $\langle \text{sign} \rangle$

If there's no carry, grab all the digits again and round. The packing function `__fp_basics_pack_high:NNNNNw` takes care of the case where rounding brings a carry.

```

11531 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
11532 #1; #2; #3#4 ; #5#6
11533 {
11534     \exp_after:wN \__fp_basics_pack_high:NNNNNw
11535     \int_use:N \__int_eval:w 1 #1
11536     \exp_after:wN \__fp_basics_pack_low:NNNNNw
11537     \int_use:N \__int_eval:w 1 #2 #3#4
11538     + \__fp_round:NNN #6 #4 #5
11539     \exp_after:wN ;
11540 }

```

(End definition for `__fp_add_significand_no_carry_o:wwwNN`.)

`__fp_add_significand_carry_o:wwwNN` `__fp_add_significand_carry_o:wwwNN` $\langle 8d \rangle$; $\langle 6d \rangle$; $\langle 2d \rangle$; $\langle \text{rounding digit} \rangle$ $\langle \text{sign} \rangle$

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.

```

11541 \cs_new:Npn \__fp_add_significand_carry_o:wwwNN
11542 #1; #2; #3#4; #5#6
11543 {
11544     + \c_one
11545     \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
11546     \int_use:N \__int_eval:w 1 1 #1
11547     \exp_after:wN \__fp_basics_pack_weird_low:NNNNNw
11548     \int_use:N \__int_eval:w 1 #2#3 +
11549     \exp_after:wN \__fp_round:NNN

```

```

11550         \exp_after:wN #6
11551         \exp_after:wN #3
11552         \__int_value:w \__fp_round_digit:Nw #4 #5 ;
11553         \exp_after:wN ;
11554     }
(End definition for \__fp_add_significand_carry_o:wwwNN.)

```

28.2.3 Absolute subtraction

```

\__fp_sub_npos_o:NnwNnw \__fp_sub_npos_o:NnwNnw <sign1> <exp1> <body1> ; \s__fp \__fp_chk:w 1
\__fp_sub_eq_o:Nnwnw <initial sign2> <exp2> <body2> ;
\__fp_sub_npos_ii_o:Nnwnw

```

Rounding properly in some modes requires to know what the sign of the result will be. Thus, we start by comparing the exponents and significands. If the numbers coincide, return zero. If the second number is larger, swap the numbers and call `__fp_sub_npos_i_o:Nnwnw` with the opposite of $\langle sign_1 \rangle$.

```

11555 \cs_new:Npn \__fp_sub_npos_o:NnwNnw #1#2#3; \s__fp \__fp_chk:w 1 #4#5#6;
11556 {
11557     \if_case:w \__fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
11558     \exp_after:wN \__fp_sub_eq_o:Nnwnw
11559     \or:
11560     \exp_after:wN \__fp_sub_npos_i_o:Nnwnw
11561     \else:
11562     \exp_after:wN \__fp_sub_npos_ii_o:Nnwnw
11563     \fi:
11564     #1 {#2} #3; {#5} #6;
11565 }
11566 \cs_new:Npn \__fp_sub_eq_o:Nnwnw #1#2; #3; { \exp_after:wN \c_zero_fp }
11567 \cs_new:Npn \__fp_sub_npos_ii_o:Nnwnw #1 #2; #3;
11568 {
11569     \exp_after:wN \__fp_sub_npos_i_o:Nnwnw
11570     \int_use:N \__int_eval:w \c_two - #1 \__int_eval_end:
11571     #3; #2;
11572 }
(End definition for \__fp_sub_npos_o:NnwNnw. This function is documented on page ??.)

```

`__fp_sub_npos_i_o:Nnwnw` After the computation is done, `__fp_sanitize:Nw` checks for overflow/underflow. It expects the $\langle final\ sign \rangle$ and the $\langle exponent \rangle$ (delimited by `;`). Start an integer expression for the exponent, which starts with the exponent of the largest number, and may be decreased if the two numbers are very close. If the two numbers have the same exponent, call the `near` auxiliary. Otherwise, decimate y , then call the `far` auxiliary to evaluate the difference between the two significands. Note that we decimate by 1 less than one could expect.

```

11573 \cs_new:Npn \__fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
11574 {
11575     \exp_after:wN \__fp_sanitize:Nw
11576     \exp_after:wN #1
11577     \int_use:N \__int_eval:w
11578     #2

```

```

11579 \if_int_compare:w #2 = #4 \exp_stop_f:
11580 \exp_after:wN \__fp_sub_back_near_o:nnnnnnnnN
11581 \else:
11582 \exp_after:wN \__fp_decimate:nNnnnn \exp_after:wN
11583 { \int_use:N \__int_eval:w #2 - #4 - \c_one \exp_after:wN }
11584 \exp_after:wN \__fp_sub_back_far_o:NnnwnnnnnN
11585 \fi:
11586 #5
11587 #3
11588 #1
11589 }

```

(End definition for __fp_sub_npos_i_o:Nnnwnw.)

```

\__fp_sub_back_near_o:nnnnnnnnN \__fp_sub_back_near_o:nnnnnnnnN {⟨Y1⟩} {⟨Y2⟩} {⟨Y3⟩} {⟨Y4⟩} {⟨X1⟩}
\__fp_sub_back_near_pack:NNNNNNw {⟨X2⟩} {⟨X3⟩} {⟨X4⟩} ⟨final sign⟩
\__fp_sub_back_near_after:wNNNNw

```

In this case, the subtraction is exact, so we discard the *⟨final sign⟩* #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.

```

11590 \cs_new:Npn \__fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
11591 {
11592 \exp_after:wN \__fp_sub_back_near_after:wNNNNw
11593 \int_use:N \__int_eval:w 10#5#6 - #1#2 - \c_eleven
11594 \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
11595 \int_use:N \__int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
11596 }
11597 \cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
11598 { + #1#2 ; {#3#4#5#6} {#7} ; }
11599 \cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
11600 {
11601 \if_meaning:w 0 #1
11602 \exp_after:wN \__fp_sub_back_shift:wnnnn
11603 \fi:
11604 ; {#1#2#3#4} {#5}
11605 }

```

(End definition for __fp_sub_back_near_o:nnnnnnnnN. This function is documented on page ??.)

```

\__fp_sub_back_shift:wnnnn
\__fp_sub_back_shift_ii:ww
\__fp_sub_back_shift_iii:NNNNNNNw
\__fp_sub_back_shift_iv:nnnnw

```

__fp_sub_back_shift:wnnnn ; {⟨Z₁⟩} {⟨Z₂⟩} {⟨Z₃⟩} {⟨Z₄⟩} ;
This function is called with $\langle Z_1 \rangle \leq 999$. Act with \number to trim leading zeros from $\langle Z_1 \rangle$ $\langle Z_2 \rangle$ (we don't do all four blocks at once, since non-zero blocks would then overflow TeX's integers). If the first two blocks are zero, the auxiliary receives an empty #1 and trims #2#30 from leading zeros, yielding a total shift between 7 and 16 to the exponent. Otherwise we get the shift from #1 alone, yielding a result between 1 and 6. Once the exponent is taken care of, trim leading zeros from #1#2#3 (when #1 is empty, the space before #2#3 is ignored), get four blocks of 4 digits and finally clean up. Trailing zeros are added so that digits can be grabbed safely.

```

11606 \cs_new:Npn \__fp_sub_back_shift:wnnnn ; #1#2
11607 {

```



```

11608 \exp_after:wN \_fp_sub_back_shift_ii:ww
11609 \_int_value:w #1 #2 0 ;
11610 }
11611 \cs_new:Npn \_fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
11612 {
11613   \if_meaning:w @ #1 @
11614     - \c_seven
11615     - \exp_after:wN \use_i:nnn
11616     \exp_after:wN \_fp_sub_back_shift_iii:NNNNNNNNw
11617     \_int_value:w #2#3 0 ~ 123456789;
11618   \else:
11619     - \_fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
11620   \fi:
11621   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
11622   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
11623   \exp_after:wN \_fp_sub_back_shift_iv:nnnnw
11624   \exp_after:wN ;
11625   \_int_value:w
11626   #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
11627 }
11628 \cs_new:Npn \_fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
11629 \cs_new:Npn \_fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }
(End definition for \_fp_sub_back_shift:wnnnn. This function is documented on page ??.)

```

$_fp_sub_back_far_o:NnnwnnnnN \langle rounding \rangle \{ \langle Y'_1 \rangle \} \{ \langle Y'_2 \rangle \} \langle extra-digits \rangle$
 $; \{ \langle X_1 \rangle \} \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \} \langle final\ sign \rangle$

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`).

```

11630 \cs_new:Npn \_fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
11631 {
11632   \if_case:w
11633     \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
11634     \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
11635     \c_zero
11636   \else:
11637     \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: \c_one
11638   \fi:
11639   \else:
11640     \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: \c_one
11641   \fi:
11642     \exp_after:wN \_fp_sub_back_quite_far_o:wwNN
11643   \or:   \exp_after:wN \_fp_sub_back_very_far_o:wwwNNN
11644   \else: \exp_after:wN \_fp_sub_back_not_far_o:wwwNNN
11645   \fi:
11646   #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1

```

```

11647 }
(End definition for \_fp_sub_back_far_o:NnnwnnnN.)

```

_fp_sub_back_quite_far_o:wwNN
_fp_sub_back_quite_far_ii:NN

The easiest case is when $x - y$ is extremely close to a power of 10, namely the first digit of x is 1, and all others vanish when subtracting y . Then the *rounding* #3 and the *final sign* #4 control whether we get 1 or 0.9999999999999999. In the usual round-to-nearest mode, we 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).

```

11648 \cs_new:Npn \_fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
11649 {
11650   \exp_after:wN \_fp_sub_back_quite_far_ii:NN
11651   \exp_after:wN #3
11652   \exp_after:wN #4
11653 }
11654 \cs_new:Npn \_fp_sub_back_quite_far_ii:NN #1#2
11655 {
11656   \if_case:w \_fp_round_neg:NNN #2 0 #1
11657   \exp_after:wN \use_i:nn
11658   \else:
11659   \exp_after:wN \use_ii:nn
11660   \fi:
11661   { ; {1000} {0000} {0000} {0000} ; }
11662   { - \c_one ; {9999} {9999} {9999} {9999} ; }
11663 }

```

(End definition for _fp_sub_back_quite_far_o:wwNN. This function is documented on page ??.)

_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.

```

11664 \cs_new:Npn \_fp_sub_back_not_far_o:wwwNN #1 ~ #2; #3 ~ #4; #5#6
11665 {
11666   - \c_one
11667   \exp_after:wN \_fp_sub_back_near_after:wNNNNw
11668   \int_use:N \_int_eval:w 1#30 - #1 - \c_eleven
11669   \exp_after:wN \_fp_sub_back_near_pack:NNNNNNw
11670   \int_use:N \_int_eval:w 11 0000 0000 + #40 - #2
11671   - \exp_after:wN \_fp_round_neg:NNN
11672   \exp_after:wN #6
11673   \use_none:nnnnnn #2 #5
11674   \exp_after:wN ;
11675 }

```

(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.

```

11676 \cs_new:Npn \_fp_sub_back_very_far_o:wwwNN #1#2#3#4#5#6#7
11677 {
11678   \_fp_pack_eight:wNNNNNNNN
11679   \_fp_sub_back_very_far_ii_o:nnNwwNN
11680   { 0 #1#2#3 #4#5#6#7 }
11681   ;
11682 }
11683 \cs_new:Npn \_fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
11684 {
11685   \exp_after:wN \_fp_basics_pack_high:NNNNw
11686   \int_use:N \_int_eval:w 1#4 - #1 - \c_one
11687   \exp_after:wN \_fp_basics_pack_low:NNNNw
11688   \int_use:N \_int_eval:w 2#5 - #2
11689   - \exp_after:wN \_fp_round_neg:NNN
11690   \exp_after:wN #7
11691   \_int_value:w
11692   \if_int_odd:w \_int_eval:w #5 - #2 \_int_eval_end:
11693     1 \else: 2 \fi:
11694   \_int_value:w \_fp_round_digit:Nw #3 #6 ;
11695   \exp_after:wN ;
11696 }

```

(End definition for `_fp_sub_back_very_far_o:wwwNN`. This function is documented on page ??.)

28.3 Multiplication

28.3.1 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`.

```

11697 \cs_new_nopar:cpn { \_fp*_o:ww }
11698 {
11699   \_fp_mul_cases_o:NnNww
11700   *
11701   { - \c_two + }
11702   \_fp_mul_npos_o:Nww
11703   { }
11704 }

```

(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.

```

11705 \cs_new:Npn \__fp_mul_cases_o:NnNnnww
11706   #1#2#3#4 \s__fp \__fp_chk:w #5#6#7; \s__fp \__fp_chk:w #8#9
11707   {
11708     \if_case:w \__int_eval:w
11709       \if_int_compare:w #5 #8 = \c_eleven
11710         \c_one
11711       \else:
11712         \if_meaning:w 3 #8
11713         \c_three
11714       \else:
11715         \if_meaning:w 3 #5
11716         \c_two
11717       \else:
11718         \if_int_compare:w #5 #8 = \c_ten
11719         \c_nine #2 - \c_two
11720       \else:
11721         (#5 #2 #8) / \c_two * \c_two + \c_seven
11722       \fi:
11723     \fi:
11724     \fi:
11725     \fi:
11726     \if_meaning:w #6 #9 - \c_one \fi:
11727     \__int_eval_end:
11728     \__fp_case_use:nw { #3 0 }
11729   \or: \__fp_case_use:nw { #3 2 }
11730   \or: \__fp_case_return_i_o:ww
11731   \or: \__fp_case_return_ii_o:ww
11732   \or: \__fp_case_return_o:Nww \c_zero_fp
11733   \or: \__fp_case_return_o:Nww \c_minus_zero_fp
11734   \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
11735   \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
11736   \or: \__fp_case_return_o:Nww \c_inf_fp
11737   \or: \__fp_case_return_o:Nww \c_minus_inf_fp
11738   #4
11739   \fi:
11740   \s__fp \__fp_chk:w #5 #6 #7;
11741   \s__fp \__fp_chk:w #8 #9
11742 }

```

(End definition for `_fp_mul_cases_o:nNnnww`.)

28.3.2 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`.

```
11743 \cs_new:Npn \_fp_mul_npos_o:Nww
11744   #1 \s\_fp \_fp_chk:w #2 #3 #4 #5 ; \s\_fp \_fp_chk:w #6 #7 #8 #9 ;
11745   {
11746     \exp_after:wN \_fp_sanitize:Nw
11747     \exp_after:wN #1
11748     \int_use:N \_int_eval:w
11749       #4 + #8
11750     \_fp_mul_significand_o:nnnnNnnnn #5 #1 #9
11751   }
```

(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:NNNNWw {<Y1>} {<Y2>} {<Y3>} {<Y4>}
\_fp_mul_significand_keep:NNNNWw
```

Note the three semicolons at the end of the definition. One is for the last `_fp_mul_significand_drop:NNNNWw`; 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:NNNNWw`.

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`.

```
11752 \cs_new:Npn \_fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
11753   {
11754     \exp_after:wN \_fp_mul_significand_test_f:NNN
11755     \exp_after:wN #5
11756     \int_use:N \_int_eval:w 99990000 + #1*#6 +
11757     \exp_after:wN \_fp_mul_significand_keep:NNNNWw
11758     \int_use:N \_int_eval:w 99990000 + #1*#7 + #2*#6 +
11759     \exp_after:wN \_fp_mul_significand_keep:NNNNWw
11760     \int_use:N \_int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
11761     \exp_after:wN \_fp_mul_significand_drop:NNNNWw
11762     \int_use:N \_int_eval:w 99990000 + #1*#9 + #2*#8 + #3*#7 + #4*#6 +
11763     \exp_after:wN \_fp_mul_significand_drop:NNNNWw
```

```

11764         \int_use:N \__int_eval:w 99990000 + #2*#9 + #3*#8 + #4*#7 +
11765         \exp_after:wN \__fp_mul_significand_drop:NNNNNw
11766         \int_use:N \__int_eval:w 99990000 + #3*#9 + #4*#8 +
11767         \exp_after:wN \__fp_mul_significand_drop:NNNNNw
11768         \int_use:N \__int_eval:w 100000000 + #4*#9 ;
11769     ; \exp_after:wN ;
11770 }
11771 \cs_new:Npn \__fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
11772 { #1#2#3#4#5 ; + #6 }
11773 \cs_new:Npn \__fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
11774 { #1#2#3#4#5 ; #6 ; }

```

(End definition for `__fp_mul_significand_o:nnnnNnnnn`. This function is documented on page ??.)

```

\__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.

```

11775 \cs_new:Npn \__fp_mul_significand_test_f:NNN #1 #2 #3
11776 {
11777     \if_meaning:w 0 #3
11778     \exp_after:wN \__fp_mul_significand_small_f:NNwwwN
11779     \else:
11780     \exp_after:wN \__fp_mul_significand_large_f:NwwNNNN
11781     \fi:
11782     #1 #3
11783 }

```

(End definition for `__fp_mul_significand_test_f:NNN`.)

`__fp_mul_significand_large_f:NwwNNNN` In this branch, *<digit 1>* is non-zero. The result is thus *<digits 1–16>*, plus some rounding which depends on the digits 16, 17, and whether all subsequent digits are zero or not. Here, `__fp_round_digit:Nw` takes digits 17 and further (as an integer expression), and replaces it by a *<rounding digit>*, suitable for `__fp_round:NNN`.

```

11784 \cs_new:Npn \__fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
11785 {
11786     \exp_after:wN \__fp_basics_pack_high:NNNNNw
11787     \int_use:N \__int_eval:w 1#2
11788     \exp_after:wN \__fp_basics_pack_low:NNNNNw
11789     \int_use:N \__int_eval:w 1#3#4#5#6#7
11790     + \exp_after:wN \__fp_round:NNN
11791     \exp_after:wN #1
11792     \exp_after:wN #7
11793     \__int_value:w \__fp_round_digit:Nw
11794 }

```

(End definition for `__fp_mul_significand_large_f:NwwNNNN`.)

`_fp_mul_significand_small_f:NNwwN`

In this branch, $\langle digit\ 1 \rangle$ is zero. Our result 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.

```

11795 \cs_new:Npn \_fp_mul_significand_small_f:NNwwN #1 #2#3; #4#5; #6; + #7
11796 {
11797   - \c_one
11798   \exp_after:wN \_fp_basics_pack_high:NNNNw
11799   \int_use:N \_int_eval:w 1#3#4
11800   \exp_after:wN \_fp_basics_pack_low:NNNNw
11801   \int_use:N \_int_eval:w 1#5#6#7
11802   + \exp_after:wN \_fp_round:NNN
11803   \exp_after:wN #1
11804   \exp_after:wN #7
11805   \_int_value:w \_fp_round_digit:Nw
11806 }
(End definition for \_fp_mul_significand_small_f:NNwwN.)

```

28.4 Division

28.4.1 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:NnNww` are provided as the fourth argument here.

```

11807 \cs_new_nopar:cpn { \_fp/_o:ww }
11808 {
11809   \_fp_mul_cases_o:NnNww
11810   /
11811   { - }
11812   \_fp_div_npos_o:Nww
11813   {
11814     \or:
11815     \_fp_case_use:nw
11816     { \_fp_division_by_zero_o:NNww \c_inf_fp / }
11817     \or:
11818     \_fp_case_use:nw
11819     { \_fp_division_by_zero_o:NNww \c_minus_inf_fp / }
11820   }
11821 }
(End definition for \_fp/_o:ww.)

```

```

\__fp_div_npos_o:Nww \__fp_div_npos_o:Nww  $\langle final\ sign \rangle$  \s__fp \__fp_chk:w 1  $\langle sign_A \rangle$   $\{\langle exp\ A \rangle\}$ 
 $\{\langle A_1 \rangle\}$   $\{\langle A_2 \rangle\}$   $\{\langle A_3 \rangle\}$   $\{\langle A_4 \rangle\}$  ; \s__fp \__fp_chk:w 1  $\langle sign_Z \rangle$   $\{\langle exp\ Z \rangle\}$ 
 $\{\langle Z_1 \rangle\}$   $\{\langle Z_2 \rangle\}$   $\{\langle Z_3 \rangle\}$   $\{\langle Z_4 \rangle\}$  ;

```

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.

```

11822 \cs_new:Npn \__fp_div_npos_o:Nww
11823   #1 \s__fp \__fp_chk:w 1 #2 #3 #4 ; \s__fp \__fp_chk:w 1 #5 #6 #7#8#9;
11824   {
11825     \exp_after:wN \__fp_sanitize:Nw
11826     \exp_after:wN #1
11827     \int_use:N \__int_eval:w
11828       #3 - #6
11829     \exp_after:wN \__fp_div_significand_i_o:wnnw
11830     \int_use:N \__int_eval:w #7 \use_i:nnnn #8 + \c_one ;
11831     #4
11832     {#7}{#8}#9 ;
11833     #1
11834   }
(End definition for \__fp_div_npos_o:Nww.)

```

28.4.2 Work plan

In this subsection, we explain how to avoid overflowing \TeX '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 $\varepsilon\text{-TeX}$'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 TeX can only handle integers less than roughly $2 \cdot 10^9$.

A better formula is to take

$$Q_A = \backslash\text{int_eval:n}\left\{\frac{10 \cdot A_1 A_2}{\lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1} - 1\right\}.$$

This is always less than $10^9 A/(10^5 Z)$, as we wanted. In words, we take the 5 first digits of Z into account, and the 8 first digits of A , using 0 as a 9-th digit rather than the true digit for efficiency reasons. We shall prove that using this formula to define all the Q_i avoids any overflow. For convenience, let us denote

$$y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1,$$

so that, taking into account the fact that $\varepsilon\text{-TeX}$ rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1 A_2 0}{y} - \frac{3}{2}. \end{aligned}$$

Note that $10^4 < y \leq 10^5$, and $999 \leq Q_A \leq 99989$. Also note that this formula does not cause an overflow as long as $A < (2^{31} - 1)/10^9 \simeq 2.147 \cdots$, 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 ε -TeX round

$$P = \backslash\mathrm{int_eval:n}\left\{\frac{2 \cdot E_1 E_2}{Z_1 Z_2}\right\},$$

which differs from $2E/Z$ by at most

$$\frac{1}{2} + 2 \left| \frac{E}{Z} - \frac{E}{10^{-8} Z_1 Z_2} \right| + 2 \left| \frac{10^8 E - E_1 E_2}{Z_1 Z_2} \right| < 1,$$

($1/2$ comes from ε -TeX's rounding) because each absolute value is less than 10^{-7} . Thus P is either the correct integer part, or is off by 1; furthermore, if $2E/Z$ is an integer, $P = 2E/Z$. We will check the sign of $2E - PZ$. If it is negative, then $E/Z \in ((P-1)/2, P/2)$. If it is zero, then $E/Z = P/2$. If it is positive, then $E/Z \in (P/2, (P+1)/2)$. In each case, we know how to round to an integer, depending on the parity of P , and the rounding mode.

28.4.3 Implementing the significand division

`_fp_div_significand_i_o:wnnw`

`_fp_div_significand_i_o:wnnw` $\langle y \rangle$; $\{\langle A_1 \rangle\}$ $\{\langle A_2 \rangle\}$ $\{\langle A_3 \rangle\}$ $\{\langle A_4 \rangle\}$
 $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$; $\langle sign \rangle$

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace $\langle A_1 \rangle$ and $\langle A_2 \rangle$, and prepare the $\langle continuation \rangle$ arguments for 4 consecutive calls to `_fp_div_significand_calc:wnnnnnnnn`. Each of these calls 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.

```

11835 \cs_new:Npn \_fp\_div\_significand\_i\_o:wnnw #1 ; #2#3 #4 ;
11836 {
11837   \exp_after:wN \_fp\_div\_significand\_test\_o:w
11838   \int_use:N \_int\_eval:w
11839   \exp_after:wN \_fp\_div\_significand\_calc:wnnnnnnnn
11840   \int_use:N \_int\_eval:w 999999 + #2 #3 0 / #1 ;
11841   #2 #3 ;
11842   #4
11843   { \exp_after:wN \_fp\_div\_significand\_ii:wnn \_int\_value:w #1 }
11844   { \exp_after:wN \_fp\_div\_significand\_ii:wnn \_int\_value:w #1 }
11845   { \exp_after:wN \_fp\_div\_significand\_ii:wnn \_int\_value:w #1 }
11846   { \exp_after:wN \_fp\_div\_significand\_iii:wnnnnnn \_int\_value:w #1 }
11847 }

```

(End definition for `_fp_div_significand_i_o:wnnw`.)

`_fp_div_significand_calc:wnnnnnnnn`
`_fp_div_significand_calc_i:wnnnnnnnn`
`_fp_div_significand_calc_ii:wnnnnnnnn`

`_fp_div_significand_calc:wnnnnnnnn` $\langle 10^6 + Q_A \rangle$; $\langle A_1 \rangle$ $\langle A_2 \rangle$; $\{\langle A_3 \rangle\}$
 $\{\langle A_4 \rangle\}$ $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$ $\{\langle continuation \rangle\}$
 expands to

$$\langle 10^6 + Q_A \rangle \langle continuation \rangle ; \langle B_1 \rangle \langle B_2 \rangle ; \{ \langle B_3 \rangle \} \{ \langle B_4 \rangle \} \{ \langle Z_1 \rangle \} \{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \} \{ \langle Z_4 \rangle \}$$

where $B = 10^4 A - Q_A \cdot Z$. This function is also used to compute C , D , E (with the input shifted accordingly), and is used in `l3fp-expo`.

We know that $0 < Q_A < 1.8 \cdot 10^5$, so the product of Q_A with each Z_i is within $\text{T}_{\text{E}}\text{X}$'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):

$$10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i \rangle \cdot \#5\#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i \rangle \cdot \#7) + 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i \rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8),$$

where $\langle i \rangle$ stands for the 10^5 digit of Q_A , which is 0 or 1, and $\#1$, $\#2$, *etc.* are the parameters of either auxiliary. The factors of 10 come from the fact that $Q_A = 10 \cdot 10^4 \cdot \langle i \rangle + \#1$. As usual, to combine all the terms, we need to choose some shifts which must ensure that the number of digits of the second, third, and fourth terms are each fixed. Here, the positive contributions are at most 10^8 and the negative contributions can go up to 10^9 . Indeed, for the auxiliary with $\langle i \rangle = 1$, $\#1$ is at most 80000, leading to contributions of at worse $-8 \cdot 10^8 4$, while the other negative term is very small $< 10^6$ (except in the first expression, where we don't care about the number of digits); for the auxiliary with $\langle i \rangle = 0$, $\#1$ can go up to 99999, but there is no other negative term. Hence, a good choice is $2 \cdot 10^9$, which produces totals in the range $[10^9, 2.1 \cdot 10^9]$. We are flirting with $\text{T}_{\text{E}}\text{X}$'s limits once more.

```

11848 \cs_new:Npn \__fp_div_significand_calc:wwnnnnnnn 1#1
11849 {
11850   \if_meaning:w 1 #1
11851     \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnn
11852   \else:
11853     \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnn
11854   \fi:
11855 }
11856 \cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
11857 {
11858   1 1 #1
11859   #9 \exp_after:wN ;
11860   \int_use:N \__int_eval:w \c__fp_Bigg_leading_shift_int
11861   + #2 - #1 * #5 - #5#60
11862   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
11863   \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
11864   + #3 - #1 * #6 - #70
11865   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
11866   \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int

```

```

11867         + #4 - #1 * #7 - #80
11868         \exp_after:wN \__fp_pack_Bigg:NNNNNNw
11869         \int_use:N \__int_eval:w \c__fp_Bigg_trailing_shift_int
11870         - #1 * #8 ;
11871     {#5}{#6}{#7}{#8}
11872 }
11873 \cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
11874 {
11875     1 0 #1
11876     #9 \exp_after:wN ;
11877     \int_use:N \__int_eval:w \c__fp_Bigg_leading_shift_int
11878     + #2 - #1 * #5
11879     \exp_after:wN \__fp_pack_Bigg:NNNNNNw
11880     \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
11881     + #3 - #1 * #6
11882     \exp_after:wN \__fp_pack_Bigg:NNNNNNw
11883     \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
11884     + #4 - #1 * #7
11885     \exp_after:wN \__fp_pack_Bigg:NNNNNNw
11886     \int_use:N \__int_eval:w \c__fp_Bigg_trailing_shift_int
11887     - #1 * #8 ;
11888     {#5}{#6}{#7}{#8}
11889 }

```

(End definition for __fp_div_significand_calc:wwnnnnnnn. This function is documented on page ??.)

__fp_div_significand_ii:wwn

__fp_div_significand_ii:wwn $\langle y \rangle$; $\langle B_1 \rangle$; $\{\langle B_2 \rangle\} \{\langle B_3 \rangle\} \{\langle B_4 \rangle\} \{\langle Z_1 \rangle\}$
 $\{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\} \{\langle Z_4 \rangle\} \langle \text{continuations} \rangle \langle \text{sign} \rangle$

Compute Q_B by evaluating $\langle B_1 \rangle \langle B_2 \rangle 0 / y - 1$. The result will be output to the left, in an $\backslash__int_eval:w$ 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 .

```

11890 \cs_new:Npn \__fp_div_significand_ii:wwn #1; #2;#3
11891 {
11892     \exp_after:wN \__fp_div_significand_pack:NNN
11893     \int_use:N \__int_eval:w
11894     \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
11895     \int_use:N \__int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
11896 }

```

(End definition for __fp_div_significand_ii:wwn.)

__fp_div_significand_iii:wwnnnnn

__fp_div_significand_iii:wwnnnnn $\langle y \rangle$; $\langle E_1 \rangle$; $\{\langle E_2 \rangle\} \{\langle E_3 \rangle\} \{\langle E_4 \rangle\}$
 $\{\langle Z_1 \rangle\} \{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\} \{\langle Z_4 \rangle\} \langle \text{sign} \rangle$

We compute $P \simeq 2E/Z$ by rounding $2E_1E_2/Z_1Z_2$. 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 .

```

11897 \cs_new:Npn \__fp_div_significand_iii:wwnnnnn #1; #2;#3#4#5 #6#7
11898 {

```

```

11899      0
11900      \exp_after:wN \_fp_div_significand_iv:wwnnnnnnnn
11901      \int_use:N \_int_eval:w (\c_two * #2 #3) / #6 #7 ; % <- P
11902      #2 ; {#3} {#4} {#5}
11903      {#6} {#7}
11904    }

```

(End definition for _fp_div_significand_iii:wwnnnnnn.)

```

\_fp_div_significand_iv:wwnnnnnnnn
\_fp_div_significand_v:NNw
\_fp_div_significand_vi:Nw

```

_fp_div_significand_iv:wwnnnnnnnn $\langle P \rangle$; $\langle E_1 \rangle$; $\{\langle E_2 \rangle\}$ $\{\langle E_3 \rangle\}$ $\{\langle E_4 \rangle\}$
 $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$ $\langle sign \rangle$

This adds to the current expression $(10^7 + 10 \cdot Q_D)$ a contribution of $5 \cdot P + \text{sign}(T)$ with $T = 2E - PZ$. This amounts to adding $P/2$ to Q_D , with an extra $\langle \text{rounding} \rangle$ digit. This $\langle \text{rounding} \rangle$ digit is 0 or 5 if T does not contribute, *i.e.*, if $0 = T = 2E - PZ$, in other words if $10^{16}A/Z$ is an integer or half-integer. Otherwise it is in the appropriate range, $[1, 4]$ or $[6, 9]$. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute T exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation $\#1 \cdot \#6\#7$ below does not cause an overflow: naively, P can be up to 35, and $\#6\#7$ up to 10^8 , but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For $P < 10$, the product obeys $P \cdot \#6\#7 < 10^8 \cdot P < 10^9$.
- For large $P \geq 3$, the rounding error on P , which is at most 1, is less than a factor of 2, hence $P \leq 4E/Z$. Also, $\#6\#7 \leq 10^8 \cdot Z$, hence $P \cdot \#6\#7 \leq 4E \cdot 10^8 < 10^9$.

Both inequalities could be made tighter if needed.

Note however that $P \cdot \#8\#9$ may overflow, since the two factors are now independent, and the result may reach $3.5 \cdot 10^9$. Thus we compute the two lower levels separately. The rest is standard, except that we use $+$ as a separator (ending integer expressions explicitly). T is negative if the first character is $-$, it is positive if the first character is neither 0 nor $-$. It is also positive if the first character is 0 and second argument of $_fp_div_significand_vi:Nw$, a sum of several terms, is also zero. Otherwise, there was an exact agreement: $T = 0$.

```

11905 \cs_new:Npn \_fp_div_significand_iv:wwnnnnnnnn #1; #2;#3#4#5 #6#7#8#9
11906 {
11907   + \c_five * #1
11908   \exp_after:wN \_fp_div_significand_vi:Nw
11909   \int_use:N \_int_eval:w -20 + 2*#2#3 - #1*#6#7 +
11910   \exp_after:wN \_fp_div_significand_v:NN
11911   \int_use:N \_int_eval:w 199980 + 2*#4 - #1*#8 +
11912   \exp_after:wN \_fp_div_significand_v:NN
11913   \int_use:N \_int_eval:w 200000 + 2*#5 - #1*#9 ;
11914 }
11915 \cs_new:Npn \_fp_div_significand_v:NN #1#2 { #1#2 \_int_eval_end: + }
11916 \cs_new:Npn \_fp_div_significand_vi:Nw #1#2;
11917 {

```

```

11918 \if_meaning:w 0 #1
11919 \if_int_compare:w \__int_eval:w #2 > \c_zero + \c_one \fi:
11920 \else:
11921 \if_meaning:w - #1 - \else: + \fi: \c_one
11922 \fi:
11923 ;
11924 }

```

(End definition for `__fp_div_significand_iv:wwnnnnnnnn`, `__fp_div_significand_v:NNw`, and `__fp_div_significand_v:`

`__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 10^6 + Q_A \__fp_div_significand_-
pack:NNN 10^6 + Q_B \__fp_div_significand_pack:NNN 10^6 + Q_C \__fp_-
div_significand_pack:NNN 10^7 + 10 \cdot Q_D + 5 \cdot P + \varepsilon ; \langle sign \rangle

```

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).

```

11925 \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 \langle 5d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 5d \rangle ; \langle sign \rangle`

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.

```

11926 \cs_new:Npn \__fp_div_significand_test_o:w 10 #1
11927 {
11928 \if_meaning:w 0 #1
11929 \exp_after:wN \__fp_div_significand_small_o:wwwNNNNwN
11930 \else:
11931 \exp_after:wN \__fp_div_significand_large_o:wwwNNNNwN
11932 \fi:
11933 #1
11934 }

```

(End definition for `__fp_div_significand_test_o:w`.)

`__fp_div_significand_small_o:wwwNNNNwN` `__fp_div_significand_small_o:wwwNNNNwN 0 \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 5d \rangle ; \langle final sign \rangle`

Standard use of `__fp_basics_pack_low:NNNNw` and `__fp_basics_pack_high:NNNNw`. We finally get to use the `\langle final sign \rangle` which has been sitting there for a while.

```

11935 \cs_new:Npn \__fp_div_significand_small_o:wwwNNNNwN
11936 0 #1; #2; #3; #4#5#6#7#8; #9
11937 {
11938 \exp_after:wN \__fp_basics_pack_high:NNNNw
11939 \int_use:N \__int_eval:w 1 #1#2

```

```

11940 \exp_after:wN \_fp_basics_pack_low:NNNNw
11941 \int_use:N \_int_eval:w 1 #3#4#5#6#7
11942 + \_fp_round:NN #9 #7 #8
11943 \exp_after:wN ;
11944 }
(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.

```

11945 \cs_new:Npn \_fp_div_significand_large_o:wwwNNNNwN
11946 #1; #2; #3; #4#5#6#7#8; #9
11947 {
11948 + \c_one
11949 \exp_after:wN \_fp_basics_pack_weird_high:NNNNNNNNw
11950 \int_use:N \_int_eval:w 1 #1 #2
11951 \exp_after:wN \_fp_basics_pack_weird_low:NNNNw
11952 \int_use:N \_int_eval:w 1 #3 #4 #5 #6 +
11953 \exp_after:wN \_fp_round:NNN
11954 \exp_after:wN #9
11955 \exp_after:wN #6
11956 \_int_value:w \_fp_round_digit:Nw #7 #8 ;
11957 \exp_after:wN ;
11958 }
(End definition for \_fp_div_significand_large_o:wwwNNNNwN.)

```

28.5 Unary operations

`_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`.

```

11959 \cs_new:Npn \_fp_set_sign_o:w #1 \s__fp \_fp_chk:w #2#3#4; @
11960 {
11961 \exp_after:wN \_fp_exp_after_o:w
11962 \exp_after:wN \s__fp
11963 \exp_after:wN \_fp_chk:w
11964 \exp_after:wN #2
11965 \_int_value:w
11966 \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
11967 #4;
11968 }
(End definition for \_fp_set_sign_o:w.)
11969 </initex | package>

```


29 l3fp-extended implementation

11970 $\langle *initex | package \rangle$

11971 $\langle @@=fp \rangle$

29.1 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:wnn  $\langle X_1 \rangle$  ;  $\langle X_2 \rangle$  ;
\__fp_fixed_mul:wnn  $\langle X_3 \rangle$  ;
\__fp_fixed_add:wnn  $\langle X_4 \rangle$  ;
```

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 $\backslash_fp_fixed_to_float:wn$. 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.

29.2 Helpers for numbers with extended precision

$\backslash c_fp_one_fixed_tl$ The fixed-point number 1, used in l3fp-expo.

11972 $\backslash tl_const:Nn \backslash c_fp_one_fixed_tl$

11973 $\{ \{10000\} \{0000\} \{0000\} \{0000\} \{0000\} \{0000\} \}$

(End definition for $\backslash c_fp_one_fixed_tl$.)

$\backslash_fp_fixed_continue:wn$ This function does nothing. Of course, there is no bound on a_1 (except T_EX’s own $2^{31} - 1$).

11974 $\backslash cs_new:Npn \backslash_fp_fixed_continue:wn \#1; \#2 \{ \#2 \#1; \}$

(End definition for `_fp_fixed_continue:wn`.)

`_fp_fixed_add_one:wn` This function adds 1 to the fixed point $\langle a \rangle$, by changing a_1 to $10000 + a_1$, then calls the $\langle continuation \rangle$. This requires $a_1 \leq 2^{31} - 10001$.

```

11975 \cs_new:Npn \_fp_fixed_add_one:wn #1#2; #3
11976 {
11977   \exp_after:wn #3 \exp_after:wn
11978   { \int_use:N \_int_eval:w \c_ten_thousand + #1 } #2 ;
11979 }

```

(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 $[0, 5 \cdot 10^8 - 1]$.

```

11980 \cs_new:Npn \_fp_fixed_div_myriad:wn #1#2#3#4#5#6; #7
11981 {
11982   \exp_after:wn \_fp_fixed_mul_after:wn
11983   \int_use:N \_int_eval:w \c\_fp_leading_shift_int
11984   \exp_after:wn \_fp_pack:NNNNNwn
11985   \int_use:N \_int_eval:w \c\_fp_trailing_shift_int
11986   + #1 ; {#7} {#2}{#3}{#4}{#5};
11987 }

```

(End definition for `_fp_fixed_div_myriad:wn`.)

`_fp_fixed_mul_after:wn` The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the $\langle continuation \rangle$ #2 in front. The $\langle continuation \rangle$ was brought up through the expansions by the packing functions.

```

11988 \cs_new:Npn \_fp_fixed_mul_after:wn #1; #2 { #2 {#1} }

```

(End definition for `_fp_fixed_mul_after:wn`.)

29.3 Multiplying a fixed point number by a short one

`_fp_fixed_mul_short:wn` 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 $\text{T}_{\text{E}}\text{X}$ 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:wn` would).

```

11989 \cs_new:Npn \_fp_fixed_mul_short:wn #1#2#3#4#5#6; #7#8#9;
11990 {
11991   \exp_after:wn \_fp_fixed_mul_after:wn
11992   \int_use:N \_int_eval:w \c\_fp_leading_shift_int
11993   + #1*#7
11994   \exp_after:wn \_fp_pack:NNNNNwn
11995   \int_use:N \_int_eval:w \c\_fp_middle_shift_int
11996   + #1*#8 + #2*#7

```

```

11997 \exp_after:wN \__fp_pack:NNNNNwn
11998 \int_use:N \__int_eval:w \c__fp_middle_shift_int
11999 + #1*#9 + #2*#8 + #3*#7
12000 \exp_after:wN \__fp_pack:NNNNNwn
12001 \int_use:N \__int_eval:w \c__fp_middle_shift_int
12002 + #2*#9 + #3*#8 + #4*#7
12003 \exp_after:wN \__fp_pack:NNNNNwn
12004 \int_use:N \__int_eval:w \c__fp_middle_shift_int
12005 + #3*#9 + #4*#8 + #5*#7
12006 \exp_after:wN \__fp_pack:NNNNNwn
12007 \int_use:N \__int_eval:w \c__fp_trailing_shift_int
12008 + #4*#9 + #5*#8 + #6*#7
12009 + ( #5*#9 + #6*#8 + #6*#9 / \c_ten_thousand )
12010 / \c_ten_thousand
12011 \exp_after:wN ; \__fp_use_braced_s:n
12012 }
(End definition for \__fp_fixed_mul_short:wwn.)

```

29.4 Dividing a fixed point number by a small integer

```

\__fp_fixed_div_int:wwN
\__fp_fixed_div_int:wnN
\__fp_fixed_div_int_auxi:wnn
\__fp_fixed_div_int_auxii:wnn
\__fp_fixed_div_int_pack:Nw
\__fp_fixed_div_int_after:Nw

```

Divides the fixed point number $\langle a \rangle$ by the (small) integer $0 < \langle n \rangle < 10^4$ and feeds the result to the $\langle continuation \rangle$. There is no bound on a_1 .

The arguments of the **i** auxiliary are 1: one of the a_i , 2: n , 3: the **ii** or the **iii** auxiliary. It computes a (somewhat tight) lower bound Q_i for the ratio a_i/n .

The **ii** auxiliary receives Q_i , n , and a_i as arguments. It adds Q_i to a surrounding integer expression, and starts a new one with the initial value 9999, which ensures that the result of this expression 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 + Q_2
\__fp_fixed_div_int_pack:Nw 9999 + Q_3
\__fp_fixed_div_int_pack:Nw 9999 + Q_4
\__fp_fixed_div_int_pack:Nw 9999 + Q_5
\__fp_fixed_div_int_pack:Nw 9999
\__fp_fixed_div_int_auxii:wnn Q_6 ; \{ \langle n \rangle \} \{ \langle a_6 \rangle \}

```

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.

```

12013 \cs_new:Npn \__fp_fixed_div_int:wwN #1#2#3#4#5#6 ; #7 ; #8
12014 {

```

```

12015 \exp_after:wN \_fp_fixed_div_int_after:Nw
12016 \exp_after:wN #8
12017 \int_use:N \_int_eval:w \c_minus_one
12018 \_fp_fixed_div_int:wnN
12019 #1; {#7} \_fp_fixed_div_int_auxi:wnn
12020 #2; {#7} \_fp_fixed_div_int_auxi:wnn
12021 #3; {#7} \_fp_fixed_div_int_auxi:wnn
12022 #4; {#7} \_fp_fixed_div_int_auxi:wnn
12023 #5; {#7} \_fp_fixed_div_int_auxi:wnn
12024 #6; {#7} \_fp_fixed_div_int_auxii:wnn ;
12025 }
12026 \cs_new:Npn \_fp_fixed_div_int:wnN #1; #2 #3
12027 {
12028 \exp_after:wN #3
12029 \int_use:N \_int_eval:w #1 / #2 - \c_one ;
12030 {#2}
12031 {#1}
12032 }
12033 \cs_new:Npn \_fp_fixed_div_int_auxi:wnn #1; #2 #3
12034 {
12035 + #1
12036 \exp_after:wN \_fp_fixed_div_int_pack:Nw
12037 \int_use:N \_int_eval:w 9999
12038 \exp_after:wN \_fp_fixed_div_int:wnN
12039 \int_use:N \_int_eval:w #3 - #1*#2 \_int_eval_end:
12040 }
12041 \cs_new:Npn \_fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + \c_two ; }
12042 \cs_new:Npn \_fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
12043 \cs_new:Npn \_fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }
(End definition for \_fp_fixed_div_int:wnN. This function is documented on page ??.)

```

29.5 Adding and subtracting fixed points

`_fp_fixed_add:wnn` 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.

```

12044 \cs_new_nopar:Npn \_fp_fixed_add:wnn { \_fp_fixed_add:Nnnnnwnn + }
12045 \cs_new_nopar:Npn \_fp_fixed_sub:wnn { \_fp_fixed_add:Nnnnnwnn - }
12046 \cs_new:Npn \_fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
12047 {
12048 \exp_after:wN \_fp_fixed_add_after:NNNNNwnn
12049 \int_use:N \_int_eval:w 9 9999 9998 + #2#3 #1 #7#8

```

```

12050      \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
12051      \int_use:N \__int_eval:w 1 9999 9998 + #4#5
12052      \__fp_fixed_add:nnNnnnwn #6 #1
12053    }
12054 \cs_new:Npn \__fp_fixed_add:nnNnnnwn #1#2 #3 #4#5 #6#7 ; #8
12055 {
12056   #3 #4#5
12057   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
12058   \int_use:N \__int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
12059 }
12060 \cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
12061 { + #1 ; {#7} {#2#3#4#5} {#6} }
12062 \cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
12063 { #7 {#1#2#3#4#5} {#6} }

```

(End definition for __fp_fixed_add:wnn and __fp_fixed_sub:wnn. These functions are documented on page ??.)

29.6 Multiplying fixed points

```

\__fp_fixed_mul:wnn
\__fp_fixed_mul:nnnnnnnwn

```

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 $\text{T}_{\text{E}}\text{X}$ 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

$$\begin{aligned}
a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} + a_1 \cdot b_5 + a_5 \cdot b_1 \right)
\end{aligned}$$

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 packing auxiliaries bring the $\langle continuation \rangle$ up through the expansion chain, as `#7`, and it is finally placed in front of the 6 brace groups by `__fp_fixed_mul_after:wn`.

```

12064 \cs_new:Npn \__fp_fixed_mul:wnn #1#2#3#4 #5; #6#7#8#9
12065 {
12066   \exp_after:wN \__fp_fixed_mul_after:wn
12067   \int_use:N \__int_eval:w \c__fp_leading_shift_int
12068   \exp_after:wN \__fp_pack:NNNNNwn

```

```

12069 \int_use:N \__int_eval:w \c__fp_middle_shift_int
12070 + #1*#6
12071 \exp_after:wN \__fp_pack:NNNNNwn
12072 \int_use:N \__int_eval:w \c__fp_middle_shift_int
12073 + #1*#7 + #2*#6
12074 \exp_after:wN \__fp_pack:NNNNNwn
12075 \int_use:N \__int_eval:w \c__fp_middle_shift_int
12076 + #1*#8 + #2*#7 + #3*#6
12077 \exp_after:wN \__fp_pack:NNNNNwn
12078 \int_use:N \__int_eval:w \c__fp_middle_shift_int
12079 + #1*#9 + #2*#8 + #3*#7 + #4*#6
12080 \exp_after:wN \__fp_pack:NNNNNwn
12081 \int_use:N \__int_eval:w \c__fp_trailing_shift_int
12082 + #2*#9 + #3*#8 + #4*#7
12083 + ( #3*#9 + #4*#8
12084 + \__fp_fixed_mul:nnnnnnwn #5 {#6}{#7} {#1}{#2}
12085 }
12086 \cs_new:Npn \__fp_fixed_mul:nnnnnnwn #1#2 #3#4 #5#6 #7#8 ; #9
12087 {
12088 #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c_ten_thousand
12089 + #1*#3 + #5*#7 ;
12090 {#9} ;
12091 }

```

(End definition for __fp_fixed_mul:wnn. This function is documented on page ??.)

29.7 Combining product and sum of fixed points

```

\__fp_fixed_mul_add:wwwn
\__fp_fixed_mul_sub_back:wwwn
\__fp_fixed_mul_one_minus_mul:wwn

```

Compute $a \times b + c$, $c - a \times b$, and $1 - a \times b$ and feed the result to the $\langle continuation \rangle$. Those functions require $0 \leq a_1, b_1, c_1 \leq 10000$. Since those functions are at the heart of the computation of Taylor expansions, we over-optimize them a bit, and in particular we do not factor out the common parts of the three functions.

For definiteness, consider the task of computing $a \times b + c$. We will perform carries in

$$\begin{aligned}
a \times b + c = & (a_1 \cdot b_1 + c_1 c_2) \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} + a_1 \cdot b_5 + a_5 \right)
\end{aligned}$$

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:wwn`, will be taken in the integer expression for the 10^{-24} level. The $\langle continuation \rangle$ is placed correctly to be taken upstream by packing auxiliaries.

```

12092 \cs_new:Npn \_fp_fixed_mul_add:wwwn #1; #2; #3#4#5#6#7#8; #9
12093 {
12094   \exp_after:wN \_fp_fixed_mul_after:wn
12095   \int_use:N \_int_eval:w \c\_fp_big_leading_shift_int
12096   \exp_after:wN \_fp_pack_big:NNNNNNwn
12097   \int_use:N \_int_eval:w \c\_fp_big_middle_shift_int + #3 #4
12098   \_fp_fixed_mul_add:Nwnnnwnnn +
12099     + #5 #6 ; #2 ; #1 ; #2 ; +
12100     + #7 #8 ; {#9} ;
12101 }
12102 \cs_new:Npn \_fp_fixed_mul_sub_back:wwwn #1; #2; #3#4#5#6#7#8; #9
12103 {
12104   \exp_after:wN \_fp_fixed_mul_after:wn
12105   \int_use:N \_int_eval:w \c\_fp_big_leading_shift_int
12106   \exp_after:wN \_fp_pack_big:NNNNNNwn
12107   \int_use:N \_int_eval:w \c\_fp_big_middle_shift_int + #3 #4
12108   \_fp_fixed_mul_add:Nwnnnwnnn -
12109     + #5 #6 ; #2 ; #1 ; #2 ; -
12110     + #7 #8 ; {#9} ;
12111 }
12112 \cs_new:Npn \_fp_fixed_one_minus_mul:wwn #1; #2; #3
12113 {
12114   \exp_after:wN \_fp_fixed_mul_after:wn
12115   \int_use:N \_int_eval:w \c\_fp_big_leading_shift_int
12116   \exp_after:wN \_fp_pack_big:NNNNNNwn
12117   \int_use:N \_int_eval:w \c\_fp_big_middle_shift_int + 1 0000 0000
12118   \_fp_fixed_mul_add:Nwnnnwnnn -
12119     ; #2 ; #1 ; #2 ; -
12120     ; {#3} ;
12121 }

```

(End definition for `_fp_fixed_mul_add:wwwn`, `_fp_fixed_mul_sub_back:wwwn`, and `_fp_fixed_mul_one_minus_mul:wwn`)

`_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:wwn`. We call the *ii* auxiliary for levels 10^{-20} and 10^{-24} , keeping the pieces of $\langle a \rangle$ we've read, but not $\langle b \rangle$, since there is another copy later in the input stream.

```

12122 \cs_new:Npn \_fp_fixed_mul_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
12123 {
12124   #1 #7*#3
12125   \exp_after:wN \_fp_pack_big:NNNNNNwn
12126   \int_use:N \_int_eval:w \c\_fp_big_middle_shift_int
12127   #1 #7*#4 #1 #8*#3
12128   \exp_after:wN \_fp_pack_big:NNNNNNwn
12129   \int_use:N \_int_eval:w \c\_fp_big_middle_shift_int
12130   #1 #7*#5 #1 #8*#4 #1 #9*#3 #2

```

```

12131      \exp_after:wN \__fp_pack_big:NNNNNNwn
12132      \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
12133      #1 \__fp_fixed_mul_add:nnnnwnnnn {#7}{#8}{#9}
12134    }
(End definition for \__fp_fixed_mul_add:Nwnnnwnnnn.)

```

__fp_fixed_mul_add:nnnnwnnnn Level 10^{-20} is $(a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1)$, multiplied by the sign, which was inserted by the *i* auxiliary. Then we prepare level 10^{-24} . We don't have access to all parts of $\langle a \rangle$ and $\langle b \rangle$ needed to make all products. Instead, we prepare the partial expressions

$$b_1 + a_4 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1$$

$$b_2 + a_4 \cdot b_3 + a_3 \cdot b_4 + a_2.$$

Obviously, those expressions make no mathematical sense: we 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 $\langle b \rangle$.

```

12135 \cs_new:Npn \__fp_fixed_mul_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
12136 {
12137   ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
12138   \exp_after:wN \__fp_pack_big:NNNNNNwn
12139   \int_use:N \__int_eval:w \c__fp_big_trailing_shift_int
12140   \__fp_fixed_mul_add:nnnnwnnnwN
12141   { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
12142   { #7 + #4*#8 + #3*#9 + #2 }
12143   {#1} #5;
12144   {#6}
12145 }
(End definition for \__fp_fixed_mul_add:nnnnwnnnn.)

```

__fp_fixed_mul_add:nnnnwnnnw 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.

```

12146 \cs_new:Npn \__fp_fixed_mul_add:nnnnwnnnwN #1#2 #3#4#5; #6#7#8; #9
12147 {
12148   #9 (#4* #1 *#7)
12149   #9 (#5*#6+#4* #2 *#7+#3*#8) / \c_ten_thousand
12150 }
(End definition for \__fp_fixed_mul_add:nnnnwnnnwN.)

```

29.8 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` 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.

```
\__fp_ep_to_fixed_auxi:www
\__fp_ep_to_fixed_auxii:nnnnnnnnwn
12151 \cs_new:Npn \__fp_ep_to_fixed:wwn #1,#2
12152 {
12153   \exp_after:wN \__fp_ep_to_fixed_auxi:www
12154   \int_use:N \__int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
12155   \tex_romannumeral:D -'0
12156   \prg_replicate:nn { \c_four - \int_max:nn {#1} { -32 } } { 0 } ;
12157 }
12158 \cs_new:Npn \__fp_ep_to_fixed_auxi:www #1; #2; #3#4#5#6#7;
12159 {
12160   \__fp_pack_eight:wNNNNNNNN
12161   \__fp_pack_twice_four:wNNNNNNNN
12162   \__fp_pack_twice_four:wNNNNNNNN
12163   \__fp_pack_twice_four:wNNNNNNNN
12164   \__fp_ep_to_fixed_auxii:nnnnnnnnwn ;
12165   #2 #1#3#4#5#6#7 0000 !
12166 }
12167 \cs_new:Npn \__fp_ep_to_fixed_auxii:nnnnnnnnwn #1#2#3#4#5#6#7; #8! #9
12168 { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }
```

(End definition for `__fp_ep_to_fixed:wwn`. This function is documented on page ??.)

`__fp_ep_to_ep:wwN` Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation `#8` is placed before the resulting exponent–mantissa pair. The input exponent may in fact be given as an integer expression. The `loop` auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the `end` auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by some 0, then packs them up neatly in $3 \times 2 = 6$ blocks of four. At the end of the day, remove with `__fp_use_i:ww` any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).

```
12169 \cs_new:Npn \__fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
12170 {
12171   \exp_after:wN #8
12172   \int_use:N \__int_eval:w #1 + \c_four
12173   \exp_after:wN \use_i:nn
12174   \exp_after:wN \__fp_ep_to_ep_loop:N
12175   \int_use:N \__int_eval:w 1 0000 0000 + #2 \__int_eval_end:
12176   #3#4#5#6#7 ; ; !
12177 }
12178 \cs_new:Npn \__fp_ep_to_ep_loop:N #1
12179 {
```

```

12180     \if_meaning:w 0 #1
12181     - \c_one
12182     \else:
12183     \__fp_ep_to_ep_end:www #1
12184     \fi:
12185     \__fp_ep_to_ep_loop:N
12186   }
12187   \cs_new:Npn \__fp_ep_to_ep_end:www
12188     #1 \fi: \__fp_ep_to_ep_loop:N #2; #3!
12189   {
12190     \fi:
12191     \if_meaning:w ; #1
12192     - \c_two * \c__fp_max_exponent_int
12193     \__fp_ep_to_ep_zero:ww
12194     \fi:
12195     \__fp_pack_twice_four:wNNNNNNNN
12196     \__fp_pack_twice_four:wNNNNNNNN
12197     \__fp_pack_twice_four:wNNNNNNNN
12198     \__fp_use_i:ww , ;
12199     #1 #2 0000 0000 0000 0000 0000 0000 ;
12200   }
12201   \cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
12202   { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }

```

(End definition for __fp_ep_to_ep:wwN. This function is documented on page ??.)

__fp_ep_compare:www In l3fp-trig we need to compare two extended-precision numbers. This is based on the same function for positive floating point numbers, with an extra test if comparing only 16 decimals is not enough to distinguish the numbers. Note that this function only works if the numbers are normalized so that their first block is in [1000,9999].

```

12203   \cs_new:Npn \__fp_ep_compare:www #1,#2#3#4#5#6#7;
12204   { \__fp_ep_compare_aux:www {#1}{#2}{#3}{#4}{#5}; #6#7; }
12205   \cs_new:Npn \__fp_ep_compare_aux:www #1;#2;#3;#4#5#6#7#8#9;
12206   {
12207     \if_case:w
12208     \__fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
12209     \if_int_compare:w #2 = #8#9 \exp_stop_f:
12210     0
12211     \else:
12212     \if_int_compare:w #2 < #8#9 - \fi: 1
12213     \fi:
12214     \or: 1
12215     \else: -1
12216     \fi:
12217   }

```

(End definition for __fp_ep_compare:www. This function is documented on page ??.)

__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].

```

12218 \cs_new:Npn \__fp_ep_mul:wwwN #1,#2; #3,#4;
12219 {
12220   \__fp_ep_to_ep:wwN #3,#4;
12221   \__fp_fixed_continue:wn
12222   {
12223     \__fp_ep_to_ep:wwN #1,#2;
12224     \__fp_ep_mul_raw:wwwN
12225   }
12226   \__fp_fixed_continue:wn
12227 }
12228 \cs_new:Npn \__fp_ep_mul_raw:wwwN #1,#2; #3,#4; #5
12229 {
12230   \__fp_fixed_mul:wn #2; #4;
12231   { \exp_after:wn #5 \int_use:N \__int_eval:w #1 + #3 , }
12232 }

```

(End definition for `__fp_ep_mul:wwwN` and `__fp_ep_mul_raw:wwwN`.)

29.9 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

$$\begin{aligned}
\alpha &= \left\lceil \frac{10^9}{\langle d_1 \rangle + 1} \right\rceil \\
\beta &= \left\lceil \frac{10^9}{\langle d_1 \rangle} \right\rceil \\
a &= 10^3 \alpha + (\beta - \alpha) \cdot \left(10^3 - \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil \right) - 1250,
\end{aligned}$$

where $\lceil \cdot \rceil$ 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 $\left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil$ 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,

$$\begin{aligned} 10^7 \langle d \rangle a &< \left(10^3 \langle d_1 \rangle + \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil + \frac{1}{2} \right) \left(\left(10^3 - \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil \right) \beta + \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil \alpha - 1250 \right) \\ &< \left(10^3 \langle d_1 \rangle + \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil + \frac{1}{2} \right) \left(\left(10^3 - \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil \right) \left(\frac{10^9}{\langle d_1 \rangle} + \frac{1}{2} \right) + \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil \left(\frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2} \right) - 1250 \right) \end{aligned} \quad (1)$$

$$< \left(10^3 \langle d_1 \rangle + \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil + \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 750 \right) \quad (3)$$

We recognize a quadratic polynomial in $[\langle d_2 \rangle / 10]$ with a negative leading coefficient: this polynomial is bounded above, according to $([\langle d_2 \rangle / 10] + a)(b - c[\langle d_2 \rangle / 10]) \leq (b + ca)^2 / (4c)$. Hence,

$$10^7 \langle d \rangle a < \frac{10^{15}}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} \left(\langle d_1 \rangle + \frac{1}{2} + \frac{1}{4} 10^{-3} - \frac{3}{8} \cdot 10^{-9} \langle d_1 \rangle (\langle d_1 \rangle + 1) \right)^2$$

Since $\langle d_1 \rangle$ takes integer values within $[1000, 9999]$, it is a simple programming exercise to check that the squared expression is always less than $\langle d_1 \rangle (\langle d_1 \rangle + 1)$, hence $10^7 \langle d \rangle a < 10^{15}$. The upper bound is proven. We also find that $\frac{3}{8}$ can be replaced by slightly smaller numbers, but nothing less than $0.374563\dots$, and going back through the derivation of the upper bound, we find that 1250 is as small a shift as we can obtain without breaking the bound.

Now, the lower bound. The same computation as for the upper bound implies

$$10^7 \langle d \rangle a > \left(10^3 \langle d_1 \rangle + \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil - \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 1750 \right)$$

This time, we want to find the minimum of this quadratic polynomial. Since the leading coefficient is still negative, the minimum is reached for one of the extreme values $[y/10] = 0$ or $[y/10] = 100$, and we easily check the bound for those values.

We have proven that the algorithm will give us a precise enough answer. Incidentally, the upper bound that we derived tells us that $a < 10^8 / \langle d \rangle \leq 10^9$, hence we can compute a safely as a TeX integer, and even add 10^9 to it to ease grabbing of all the digits. The lower bound implies $10^8 - 1755 < a$, which we do not care about.

`_fp_ep_div:wwwn` Compute the ratio of two extended-precision numbers. The result is an extended-precision number whose first block lies in the range $[100, 9999]$, and is placed after the *<continuation>* once we are done. First normalize the inputs so that both first block lie in

[1000, 9999], then call `_fp_ep_div_esti:wwwn` $\langle denominator \rangle$ $\langle numerator \rangle$, responsible for estimating the inverse of the denominator.

```

12233 \cs_new:Npn \_fp_ep_div:wwwn #1,#2; #3,#4;
12234 {
12235   \_fp_ep_to_ep:wwN #1,#2;
12236   \_fp_fixed_continue:wn
12237   {
12238     \_fp_ep_to_ep:wwN #3,#4;
12239     \_fp_ep_div_esti:wwwn
12240   }
12241 }

```

(End definition for `_fp_ep_div:wwwn`.)

```

\_fp_ep_div_esti:wwwn
\_fp_ep_div_estii:wwnnwwn
\_fp_ep_div_estiii:NNNNNwwwn

```

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`.

```

12242 \cs_new:Npn \_fp_ep_div_esti:wwwn #1,#2#3; #4,
12243 {
12244   \exp_after:wN \_fp_ep_div_estii:wwnnwwn
12245   \int_use:N \_int_eval:w 10 0000 0000 / ( #2 + \c_one )
12246   \exp_after:wN ;
12247   \int_use:N \_int_eval:w #4 - #1 + \c_one ,
12248   {#2} #3;
12249 }
12250 \cs_new:Npn \_fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
12251 {
12252   \exp_after:wN \_fp_ep_div_estiii:NNNNNwwwn
12253   \int_use:N \_int_eval:w 10 0000 0000 - 1750
12254   + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
12255   {#3}{#4}#5; #6; { #7 #2, }
12256 }
12257 \cs_new:Npn \_fp_ep_div_estiii:NNNNNwwwn 1#1#2#3#4#5#6; #7;
12258 {
12259   \_fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
12260   \_fp_ep_div_epsilon:wnNNNNn {#1#2#3#4}#5#6
12261   \_fp_fixed_mul:wwn
12262 }

```

(End definition for `_fp_ep_div_esti:wwwn`, `_fp_ep_div_estii:wwnnwwn`, and `_fp_ep_div_estiii:NNNNNwwwn`.)

```

\_fp_ep_div_epsilon:wnNNNNNn
\_fp_ep_div_epsilon_pack:NNNNNw
\_fp_ep_div_epsilonii:wwnNNNNNn

```

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`.

```

12263 \cs_new:Npn \__fp_ep_div_epsilon:wnNNNNNn #1#2#3#4#5#6;
12264 {
12265   \exp_after:wN \__fp_ep_div_epsilon:wnNNNNNn
12266   \int_use:N \__int_eval:w 1 9998 - #2
12267   \exp_after:wN \__fp_ep_div_epsilon_pack:NNNNNw
12268   \int_use:N \__int_eval:w 1 9999 9998 - #3#4
12269   \exp_after:wN \__fp_ep_div_epsilon_pack:NNNNNw
12270   \int_use:N \__int_eval:w 2 0000 0000 - #5#6 ; ;
12271 }
12272 \cs_new:Npn \__fp_ep_div_epsilon_pack:NNNNNw #1#2#3#4#5#6;
12273 { + #1 ; {#2#3#4#5} {#6} }
12274 \cs_new:Npn \__fp_ep_div_epsilon:wnNNNNNn 1#1; #2; #3#4#5#6#7#8
12275 {
12276   \__fp_fixed_mul:wnn {0000}{#1}#2; {0000}{#1}#2;
12277   \__fp_fixed_add_one:wN
12278   \__fp_fixed_mul:wnn {10000} {#1} #2 ;
12279   {
12280     \__fp_fixed_mul_short:wnn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
12281     \__fp_fixed_div_myriad:wn
12282     \__fp_fixed_mul:wnn
12283   }
12284   \__fp_fixed_add:wnn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
12285 }

```

(End definition for `__fp_ep_div_epsilon:wnNNNNNn`, `__fp_ep_div_epsilon_pack:NNNNNw`, and `__fp_ep_div_epsilon:wnNNNNNn`.)

29.10 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 $r \mapsto (r + 10^8/(x_1r))/2$, starting from a guess that optimizes the number of steps before convergence. In fact, just as there is a slight shift when computing divisions to ensure that some inequalities hold, we will replace 10^8 by a slightly larger number which will ensure that $r^2x \geq 10^4$. This also causes $r \in [101, 1003]$. Another correction to the above is that the input is actually normalized to $[0.1, 1)$, and we use either 10^8 or 10^9 in the Newton method, depending on the parity of the exponent. Skipping those technical hurdles, once we have the approximation r , we set $y = 10^{-4}r^2x$ (or rather, the correct power of 10 to get $y \simeq 1$) and compute $y^{-1/2}$ through another application of Newton's method. This time, the starting value is $z = 1$, each step maps $z \mapsto z(1.5 - 0.5yz^2)$, and we perform a fixed number of steps. Our final result combines r with $y^{-1/2}$ as $x^{-1/2} = 10^{-2}ry^{-1/2}$.

```

\__fp_ep_isqrt:wnn
\__fp_ep_isqrt_aux:wnn
\__fp_ep_isqrt_auxii:wnnnwn

```

First normalize the input, then check the parity of the exponent #1. If it is even, the result's exponent will be $-\#1/2$, otherwise it will be $(\#1 - 1)/2$ (except in the case

where the input was an exact power of 100). The `auxii` function receives as #1 the result's exponent just computed, as #2 the starting value for the iteration giving r (the values 168 and 535 lead to the least number of iterations before convergence, on average), as #3 and #4 one empty argument and one 0, depending on the parity of the original exponent, as #5 and #6 the normalized mantissa ($\#5 \in [1000, 9999]$), and as #7 the continuation. It sets up the iteration giving r : the `esti` function thus receives the initial two guesses #2 and 0, an approximation #5 of 10^4x (its first block of digits), and the empty/zero arguments #3 and #4, followed by the mantissa and an altered continuation where we have stored the result's exponent.

```

12286 \cs_new:Npn \__fp_ep_isqrt:wnn #1,#2;
12287 {
12288   \__fp_ep_to_ep:wwN #1,#2;
12289   \__fp_ep_isqrt_auxi:wnn
12290 }
12291 \cs_new:Npn \__fp_ep_isqrt_auxi:wnn #1,
12292 {
12293   \exp_after:wN \__fp_ep_isqrt_auxii:wwnnwn
12294   \int_use:N \__int_eval:w
12295   \int_if_odd:nTF {#1}
12296     { (\c_one - #1) / \c_two , 535 , { 0 } { } }
12297     { \c_one - #1 / \c_two , 168 , { } { 0 } }
12298 }
12299 \cs_new:Npn \__fp_ep_isqrt_auxii:wwnnwn #1, #2, #3#4 #5#6; #7
12300 {
12301   \__fp_ep_isqrt_esti:wwnnwn #2, 0, #5, {#3} {#4}
12302   {#5} #6 ; { #7 #1 , }
12303 }

```

(End definition for `__fp_ep_isqrt:wnn`. This function is documented on page ??.)

```

\__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 .

```

12304 \cs_new:Npn \__fp_ep_isqrt_esti:wwnnwn #1, #2, #3, #4
12305 {
12306   \if_int_compare:w #1 = #2 \exp_stop_f:
12307   \exp_after:wN \__fp_ep_isqrt_estii:wwnnwn
12308   \fi:
12309   \exp_after:wN \__fp_ep_isqrt_esti:wwnnwn
12310   \int_use:N \__int_eval:w
12311   (#1 + 1 0050 0000 #4 / (#1 * #3)) / \c_two ,

```

```

12312     #1, #3, {#4}
12313   }
12314 \cs_new:Npn \__fp_ep_isqrt_estii:wwnnwn #1, #2, #3, #4#5
12315   {
12316     \exp_after:wN \__fp_ep_isqrt_estiii:NNNNNwwnn
12317     \int_use:N \__int_eval:w 1000 0000 + #2 * #2 #5 * \c_five
12318     \exp_after:wN , \int_use:N \__int_eval:w 10000 + #2 ;
12319   }
12320 \cs_new:Npn \__fp_ep_isqrt_estiii:NNNNNwwnn 1#1#2#3#4#5#6, 1#7#8; #9;
12321   {
12322     \__fp_fixed_mul_short:wwn #9; {#1} {#2#3#4#5} {#600} ;
12323     \__fp_ep_isqrt_epsi:wN
12324     \__fp_fixed_mul_short:wwn {#7} {#80} {0000} ;
12325   }

```

(End definition for `__fp_ep_isqrt_esti:wwnnwn`, `__fp_ep_isqrt_estii:wwnnwn`, and `__fp_ep_isqrt_estiii:NNNNNwwnn`)

`__fp_ep_isqrt_epsi:wN`
`__fp_ep_isqrt_epsi: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`.

```

12326 \cs_new:Npn \__fp_ep_isqrt_epsi:wN #1;
12327   {
12328     \__fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}{0000}; #1;
12329     \__fp_ep_isqrt_epsi:wwN #1;
12330     \__fp_ep_isqrt_epsi:wwN #1;
12331     \__fp_ep_isqrt_epsi:wwN #1;
12332   }
12333 \cs_new:Npn \__fp_ep_isqrt_epsi:wwN #1; #2;
12334   {
12335     \__fp_fixed_mul:wwn #1; #1;
12336     \__fp_fixed_mul_sub_back:wwnn #2;
12337     {15000}{0000}{0000}{0000}{0000}{0000};
12338     \__fp_fixed_mul:wwn #1;
12339   }

```

(End definition for `__fp_ep_isqrt_epsi:wN` and `__fp_ep_isqrt_epsi:wwN`.)

29.11 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.

```

12340 \cs_new:Npn \__fp_ep_to_float:wwN #1,
12341   { + \__int_eval:w #1 \__fp_fixed_to_float:wN }
12342 \cs_new:Npn \__fp_ep_inv_to_float:wwN #1,#2;

```



```

12343 {
12344   \__fp_ep_div:wwwn 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1,#2;
12345   \__fp_ep_to_float:wwN
12346 }
(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.
12347 \cs_new:Npn \__fp_fixed_inv_to_float:wN
12348 { \__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.
12349 \cs_new:Npn \__fp_fixed_to_float_rad:wN #1;
12350 {
12351   \__fp_fixed_mul:wwN #1; {5729}{5779}{5130}{8232}{0876}{7981};
12352   { \__fp_ep_to_float:wwN 2, }
12353 }
(End definition for \__fp_fixed_to_float_rad:wN.)

\__fp_fixed_to_float:wN yields
\__fp_fixed_to_float:Nw
<exponent'> ; {<a'1>} {<a'2>} {<a'3>} {<a'4>} ;

```

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 .⁸

```

12354 \cs_new:Npn \__fp_fixed_to_float:Nw #1#2; { \__fp_fixed_to_float:wN #2; #1 }
12355 \cs_new:Npn \__fp_fixed_to_float:wN #1#2#3#4#5#6; #7
12356 {
12357   + \__int_eval:w \c_four % for the 8-digit-at-the-start thing.
12358   \exp_after:wN \exp_after:wN
12359   \exp_after:wN \__fp_fixed_to_loop:N
12360   \exp_after:wN \use_none:n
12361   \int_use:N \__int_eval:w
12362   1 0000 0000 + #1 \exp_after:wN \__fp_use_none_stop_f:n
12363   \__int_value:w 1#2 \exp_after:wN \__fp_use_none_stop_f:n
12364   \__int_value:w 1#3#4 \exp_after:wN \__fp_use_none_stop_f:n
12365   \__int_value:w 1#5#6
12366   \exp_after:wN ;
12367   \exp_after:wN ;
12368 }
12369 \cs_new:Npn \__fp_fixed_to_loop:N #1
12370 {
12371   \if_meaning:w 0 #1
12372   - \c_one
12373   \exp_after:wN \__fp_fixed_to_loop:N

```

⁸Bruno: I must double check this assumption.

```

12374 \else:
12375 \exp_after:wN \__fp_fixed_to_loop_end:w
12376 \exp_after:wN #1
12377 \fi:
12378 }
12379 \cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
12380 {
12381 \if_meaning:w ; #1
12382 \exp_after:wN \__fp_fixed_to_float_zero:w
12383 \else:
12384 \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
12385 \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
12386 \exp_after:wN \__fp_fixed_to_float_pack:ww
12387 \exp_after:wN ;
12388 \fi:
12389 #1 #2 0000 0000 0000 0000 ;
12390 }
12391 \cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
12392 {
12393 - \c_two * \c__fp_max_exponent_int ;
12394 {0000} {0000} {0000} {0000} ;
12395 }
12396 \cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
12397 {
12398 \if_int_compare:w #2 > \c_four
12399 \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
12400 \fi:
12401 ; #1 ;
12402 }
12403 \cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
12404 {
12405 \exp_after:wN \__fp_basics_pack_high:NNNNNw
12406 \int_use:N \__int_eval:w 1 #1#2
12407 \exp_after:wN \__fp_basics_pack_low:NNNNNw
12408 \int_use:N \__int_eval:w 1 #3#4 + \c_one ;
12409 }
(End definition for \__fp_fixed_to_float:wN and \__fp_fixed_to_float:Nw.)
12410 </initex | package>

```

30 l3fp-expo implementation

```

12411 <*initex | package>
12412 <@@=fp>

```

30.1 Logarithm

30.1.1 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?

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.

30.1.2 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>	12413	<code>\tl_const:Nn \c_fp_ln_i_fixed_t1</code>	{ {0000}{0000}{0000}{0000}{0000}{0000} }
<code>\c_fp_ln_ii_fixed_t1</code>	12414	<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>	12415	<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>	12416	<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>	12417	<code>\tl_const:Nn \c_fp_ln_vii_fixed_t1</code>	{ {17917}{5946}{9228}{0550}{0081}{2477} }
<code>\c_fp_ln_viii_fixed_t1</code>	12418	<code>\tl_const:Nn \c_fp_ln_viii_fixed_t1</code>	{ {19459}{1014}{9055}{3133}{0510}{5353} }
<code>\c_fp_ln_ix_fixed_t1</code>	12419	<code>\tl_const:Nn \c_fp_ln_ix_fixed_t1</code>	{ {20794}{4154}{1679}{8359}{2825}{1696} }
<code>\c_fp_ln_x_fixed_t1</code>	12420	<code>\tl_const:Nn \c_fp_ln_x_fixed_t1</code>	{ {21972}{2457}{7336}{2193}{8279}{0490} }
	12421	<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.)

30.1.3 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`.

```

12422 \cs_new:Npn \__fp_ln_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
12423 {
12424   \if_meaning:w 2 #3
12425     \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
12426   \fi:
12427   \if_case:w #2 \exp_stop_f:
12428     \__fp_case_use:nw
12429     { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
12430   \or:
12431   \else:
12432     \__fp_case_return_same_o:w
12433   \fi:
12434   \__fp_ln_npos_o:w \s__fp \__fp_chk:w #2#3#4;
12435 }

```

(End definition for __fp_ln_o:w.)

30.1.4 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.

```

12436 \cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3;
12437 { %^A todo: ln(1) should be "exact zero", not "underflow"
12438   \exp_after:wN \__fp_sanitizew:Nw
12439   \__int_value:w % for the overall sign
12440   \if_int_compare:w #1 < \c_one
12441     2
12442   \else:
12443     0
12444   \fi:
12445   \exp_after:wN \exp_stop_f:
12446   \int_use:N \__int_eval:w % for the exponent
12447   \__fp_ln_significand:NNNNnnnnN #2#3
12448   \__fp_ln_exponent:wn {#1}
12449 }

```

(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.

```

12450 \cs_new:Npn \__fp_ln_significand:NNNNnnnnN #1#2#3#4
12451 {
12452   \exp_after:wN \__fp_ln_x_ii:wnnnnn
12453   \__int_value:w
12454   \if_case:w #1 \exp_stop_f:
12455   \or:
12456     \if_int_compare:w #2 < \c_four

```

```

12457         \__int_eval:w \c_ten - #2
12458     \else:
12459         6
12460     \fi:
12461     \or: 4
12462     \or: 3
12463     \or: 2
12464     \or: 2
12465     \or: 2
12466     \else: 1
12467     \fi:
12468     ; { #1 #2 #3 #4 }
12469 }

```

(End definition for __fp_ln_significand:NNNNnnnN.)

__fp_ln_x_ii:wnnnn We have thus found c . It is chosen such that $0.7 \leq ac < 1.4$ in all cases. Compute $1 + x = 1 + ac \in [1.7, 2.4)$.

```

12470 \cs_new:Npn \__fp_ln_x_ii:wnnnn #1; #2#3#4#5
12471 {
12472     \exp_after:wN \__fp_ln_div_after:Nw
12473     \cs:w c__fp_ln_ \tex_romannumeral:D #1 _fixed_tl \exp_after:wN \cs_end:
12474     \__int_value:w
12475     \exp_after:wN \__fp_ln_x_iv:wnnnnnnnn
12476     \int_use:N \__int_eval:w
12477     \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
12478     \int_use:N \__int_eval:w 9999 9999 + #1*#2#3 +
12479     \exp_after:wN \__fp_ln_x_iii:NNNNNw
12480     \int_use:N \__int_eval:w 1 0000 0000 + #1*#4#5 ;
12481     {20000} {0000} {0000} {0000}
12482 } %^A todo: reoptimize (a generalization attempt failed).
12483 \cs_new:Npn \__fp_ln_x_iii:NNNNNw #1 #2#3#4#5 #6; { #1; {#2#3#4#5} {#6} }
12484 \cs_new:Npn \__fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
12485 {
12486     #1#2#3#4#5 + \c_one ;
12487     {#1#2#3#4#5} {#6}
12488 }

```

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

$$10^4 B \leq A_1 A_2 \cdot A_3 A_4 - \left(\frac{A_1 A_2}{y} - \frac{3}{2} \right) 10^4 Z \leq A_1 A_2 \left(1 - \frac{10^4 Z}{y} \right) + 1 + \frac{3}{2} y \leq 10^8 \frac{A}{y} + 1 + \frac{3}{2} y$$

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).⁹

`__fp_ln_x_iv:wnnnnnnnn <1 or 2> <8d> ; {\<4d>} {\<4d>} <fixed-tl>`

The number is x . Compute y by adding 1 to the five first digits.

```
12489 \cs_new:Npn \__fp_ln_x_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
12490 {
12491   \exp_after:wN \__fp_div_significand_pack:NNN
12492   \int_use:N \__int_eval:w
12493   \__fp_ln_div_i:w #1 ;
12494   #6 #7 ; {#8} {#9}
12495   {#2} {#3} {#4} {#5}
12496   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
12497   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
12498   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
12499   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
```

⁹Bruno: to be completed.

```

12500     { \exp_after:wN \__fp_ln_div_vi:wnn \__int_value:w #1 }
12501   }
12502   \cs_new:Npn \__fp_ln_div_i:w #1;
12503   {
12504     \exp_after:wN \__fp_div_significand_calc:wnnnnnnnn
12505     \int_use:N \__int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
12506   }
12507   \cs_new:Npn \__fp_ln_div_ii:wnn #1; #2;#3 % y; B1;B2 <- for k=1
12508   {
12509     \exp_after:wN \__fp_div_significand_pack:NNN
12510     \int_use:N \__int_eval:w
12511     \exp_after:wN \__fp_div_significand_calc:wnnnnnnnn
12512     \int_use:N \__int_eval:w 999999 + #2 #3 / #1 ; % Q2
12513     #2 #3 ;
12514   }
12515   \cs_new:Npn \__fp_ln_div_vi:wnn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
12516   {
12517     \exp_after:wN \__fp_div_significand_pack:NNN
12518     \int_use:N \__int_eval:w 1000000 + #2 #3 / #1 ; % Q6
12519   }

```

We now have essentially¹⁰

$$\begin{aligned} & _ _ \text{fp_ln_div_after:Nw} \langle \text{fixed tl} \rangle _ _ \text{fp_div_significand_pack:NNN} 10^6 + \\ & Q_1 _ _ \text{fp_div_significand_pack:NNN} 10^6 + Q_2 _ _ \text{fp_div_significand_} \\ & \text{pack:NNN} 10^6 + Q_3 _ _ \text{fp_div_significand_pack:NNN} 10^6 + Q_4 _ _ \text{fp_} \\ & \text{div_significand_pack:NNN} 10^6 + Q_5 _ _ \text{fp_div_significand_pack:NNN} \\ & 10^6 + Q_6 ; \langle \text{exponent} \rangle ; \langle \text{continuation} \rangle \end{aligned}$$

where $\langle \text{fixed tl} \rangle$ holds the logarithm of a number in $[1, 10]$, and $\langle \text{exponent} \rangle$ is the exponent. Also, the expansion is done backwards. Then $_ _ \text{fp_div_significand_pack:NNN}$ 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

$$\begin{aligned} & _ _ \text{fp_ln_div_after:Nw} \langle \text{fixed-tl} \rangle \langle 1d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \\ & \langle 4d \rangle ; \langle \text{exponent} \rangle ; \end{aligned}$$

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.

```

12520   \cs_new:Npn \__fp_ln_div_after:Nw #1#2;
12521   {
12522     \if_meaning:w 0 #2
12523     \exp_after:wN \__fp_ln_t_small:Nw
12524     \else:
12525     \exp_after:wN \__fp_ln_t_large:NNw
12526     \exp_after:wN -
12527     \fi:

```

¹⁰Bruno: 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.

```

12528     #1
12529   }
12530   \cs_new:Npn \__fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
12531   {
12532     \exp_after:wN \__fp_ln_t_large:NNw
12533     \exp_after:wN + % <sign>
12534     \exp_after:wN #1
12535     \int_use:N \__int_eval:w 9999 - #2 \exp_after:wN ;
12536     \int_use:N \__int_eval:w 9999 - #3 \exp_after:wN ;
12537     \int_use:N \__int_eval:w 9999 - #4 \exp_after:wN ;
12538     \int_use:N \__int_eval:w 9999 - #5 \exp_after:wN ;
12539     \int_use:N \__int_eval:w 9999 - #6 \exp_after:wN ;
12540     \int_use:N \__int_eval:w 1 0000 - #7 ;
12541   }

  \__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.



```

12542 \cs_new:Npn __fp_ln_t_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
12543 {
12544 \exp_after:wN __fp_ln_square_t_after:w
12545 \int_use:N __int_eval:w 9999 0000 + #3*#3
12546 \exp_after:wN __fp_ln_square_t_pack:NNNNNw
12547 \int_use:N __int_eval:w 9999 0000 + 2*#3*#4
12548 \exp_after:wN __fp_ln_square_t_pack:NNNNNw
12549 \int_use:N __int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
12550 \exp_after:wN __fp_ln_square_t_pack:NNNNNw
12551 \int_use:N __int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
12552 \exp_after:wN __fp_ln_square_t_pack:NNNNNw
12553 \int_use:N __int_eval:w 1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
12554 + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
12555 % ; ; ;
12556 \exp_after:wN __fp_ln_twice_t_after:w
12557 \int_use:N __int_eval:w -1 + 2*#3
12558 \exp_after:wN __fp_ln_twice_t_pack:Nw
12559 \int_use:N __int_eval:w 9999 + 2*#4
12560 \exp_after:wN __fp_ln_twice_t_pack:Nw
12561 \int_use:N __int_eval:w 9999 + 2*#5
12562 \exp_after:wN __fp_ln_twice_t_pack:Nw
12563 \int_use:N __int_eval:w 9999 + 2*#6
12564 \exp_after:wN __fp_ln_twice_t_pack:Nw
12565 \int_use:N __int_eval:w 9999 + 2*#7
12566 \exp_after:wN __fp_ln_twice_t_pack:Nw
12567 \int_use:N __int_eval:w 10000 + 2*#8 ; ;
12568 { __fp_ln_c:NwNw #1 }
12569 #2

```


```



```

12570 }
12571 \cs_new:Npn \__fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
12572 \cs_new:Npn \__fp_ln_twice_t_after:w #1; { ;;; {#1} }
12573 \cs_new:Npn \__fp_ln_square_t_pack:NNNNw #1 #2#3#4#5 #6;
12574 { + #1#2#3#4#5 ; {#6} }
12575 \cs_new:Npn \__fp_ln_square_t_after:w 1 0 #1#2#3 #4;
12576 { \__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;

```

¹¹

This uses the routine for dividing a number by a small integer ($< 10^4$).

```

12577 \cs_new:Npn \__fp_ln_Taylor:wwNw
12578 { \__fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000}{0000} ; }
12579 \cs_new:Npn \__fp_ln_Taylor_loop:www #1; #2; #3;
12580 {
12581   \if_int_compare:w #1 = \c_one
12582     \__fp_ln_Taylor_break:w
12583   \fi:
12584   \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl ; #1;
12585   \__fp_fixed_add:wnn #2;
12586   \__fp_fixed_mul:wnn #3;
12587   {
12588     \exp_after:wN \__fp_ln_Taylor_loop:www
12589     \int_use:N \__int_eval:w #1 - \c_two ;
12590   }
12591   #3;
12592 }
12593 \cs_new:Npn \__fp_ln_Taylor_break:w \fi: #1 \__fp_fixed_add:wnn #2#3; #4 ;;
12594 {

```

¹¹Bruno: add explanations.

```

12595 \fi:
12596 \exp_after:wN \_fp_fixed_mul:wwn
12597 \exp_after:wN { \int_use:N \_int_eval:w 10000 + #2 } #3;
12598 }

```

(End definition for `_fp_ln_Taylor:wwNw`. This function is documented on page ??.)

```

\_fp_ln_c:NwNw \_fp_ln_c:NwNw <sign> {<r1>} {<r2>} {<r3>} {<r4>} {<r5>} {<r6>} ; <fixed tl>
<exponent> ; <continuation>

```

We are now reduced to finding $\ln(c)$ and $\langle exponent \rangle \ln(10)$ in a table, and adding it to the mixture. The first step is to get $\ln(c) - \ln(x) = -\ln(a)$, then we get $\ln(10)$ 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 $\ln(10/7) \simeq 0.35$.¹²

```

12599 \cs_new:Npn \_fp_ln_c:NwNw #1 #2; #3
12600 {
12601   \if_meaning:w + #1
12602     \exp_after:wN \exp_after:wN \exp_after:wN \_fp_fixed_sub:wwn
12603   \else:
12604     \exp_after:wN \exp_after:wN \exp_after:wN \_fp_fixed_add:wwn
12605   \fi:
12606   #3 ; #2 ;
12607 }

```

13

(End definition for `_fp_ln_c:NwNw`. This function is documented on page ??.)

```

\_fp_ln_exponent:wn \_fp_ln_exponent:wn {<s1>} {<s2>} {<s3>} {<s4>} {<s5>} {<s6>} ; {<exponent>}

```

Compute $\langle exponent \rangle$ times $\ln(10)$. Apart from the cases where $\langle exponent \rangle$ is 0 or 1, the result will necessarily be at least $\ln(10) \simeq 2.3$ in magnitude. We can thus drop the least significant 4 digits. In the case of a very large (positive or negative) exponent, we can (and we need to) drop 4 additional digits, since the result is of order 10^4 . Naively, one would think that in both cases we can drop 4 more digits than we do, but that would be slightly too tight for rounding to happen correctly. Besides, we already have addition and subtraction for 24 digits fixed point numbers.

```

12608 \cs_new:Npn \_fp_ln_exponent:wn #1; #2
12609 {
12610   \if_case:w #2 \exp_stop_f:
12611     \c_zero \_fp_case_return:nw { \_fp_fixed_to_float:Nw 2 }
12612   \or:
12613     \exp_after:wN \_fp_ln_exponent_one:ww \_int_value:w
12614   \else:
12615     \if_int_compare:w #2 > \c_zero
12616       \exp_after:wN \_fp_ln_exponent_small:NNww
12617       \exp_after:wN 0
12618     \exp_after:wN \_fp_fixed_sub:wwn \_int_value:w
12619   \else:

```

¹²Bruno: that was wrong at some point, I must check.

¹³Bruno: this *must* be updated with correct values!

```

12620         \exp_after:wN \__fp_ln_exponent_small:NNww
12621         \exp_after:wN 2
12622         \exp_after:wN \__fp_fixed_add:wwn \__int_value:w -
12623     \fi:
12624     \fi:
12625     #2; #1;
12626 }

```

Now we painfully write all the cases.¹⁴ No overflow nor underflow can happen, except when computing $\ln(1)$.

```

12627 \cs_new:Npn \__fp_ln_exponent_one:ww #1; #1;
12628 {
12629     \c_zero
12630     \exp_after:wN \__fp_fixed_sub:wwn \c__fp_ln_x_fixed_t1 ; #1;
12631     \__fp_fixed_to_float:wN 0
12632 }

```

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.

```

12633 \cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
12634 {
12635     \c_four
12636     \exp_after:wN \__fp_fixed_mul:wwn
12637     \c__fp_ln_x_fixed_t1 ;
12638     {#3}{0000}{0000}{0000}{0000}{0000} ;
12639     #2
12640     {0000}{#4}{#5}{#6}{#7}{#8};
12641     \__fp_fixed_to_float:wN #1
12642 }

```

(End definition for `__fp_ln_exponent:wn`. This function is documented on page ??.)

30.2 Exponential

30.2.1 Sign, exponent, and special numbers

`__fp_exp_o:w`

```

12643 \cs_new:Npn \__fp_exp_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
12644 {
12645     \if_case:w #2 \exp_stop_f:
12646     \__fp_case_return_o:Nw \c_one_fp
12647     \or:
12648     \exp_after:wN \__fp_exp_normal:w
12649     \or:
12650     \if_meaning:w 0 #3
12651     \exp_after:wN \__fp_case_return_o:Nw
12652     \exp_after:wN \c_inf_fp
12653     \else:

```

¹⁴Bruno: do rounding.

```

12654         \exp_after:wN \__fp_case_return_o:Nw
12655         \exp_after:wN \c_zero_fp
12656     \fi:
12657 \or:
12658     \__fp_case_return_same_o:w
12659 \fi:
12660 \s__fp \__fp_chk:w #2#3#4;
12661 }

```

(End definition for __fp_exp_o:w.)

__fp_exp_normal:w
__fp_exp_pos:Nnwnw

```

12662 \cs_new:Npn \__fp_exp_normal:w \s__fp \__fp_chk:w 1#1
12663 {
12664     \if_meaning:w 0 #1
12665         \__fp_exp_pos:Nnwnw + \__fp_fixed_to_float:wN
12666     \else:
12667         \__fp_exp_pos:Nnwnw - \__fp_fixed_inv_to_float:wN
12668 \fi:
12669 }
12670 \cs_new:Npn \__fp_exp_pos:Nnwnw #1#2#3 \fi: #4#5;
12671 {
12672     \fi:
12673     \exp_after:wN \__fp_sanitize:Nw
12674     \exp_after:wN 0
12675     \__int_value:w #1 \__int_eval:w
12676     \if_int_compare:w #4 < - \c_eight
12677         \c_one
12678         \exp_after:wN \__fp_add_big_i_o:wNww
12679         \int_use:N \__int_eval:w \c_one - #4 ;
12680         0 {1000}{0000}{0000}{0000} ; #5;
12681         \tex_romannumeral:D
12682     \else:
12683         \if_int_compare:w #4 > \c_five % cf \c__fp_max_exponent_int
12684             \exp_after:wN \__fp_exp_overflow:
12685             \tex_romannumeral:D
12686         \else:
12687             \if_int_compare:w #4 < \c_zero
12688                 \exp_after:wN \use_i:nn
12689             \else:
12690                 \exp_after:wN \use_ii:nn
12691 \fi:
12692 {
12693     \c_zero
12694     \__fp_decimate:nNnnnn { - #4 }
12695     \__fp_exp_Taylor:Nnwn
12696 }
12697 {
12698     \__fp_decimate:nNnnnn { \c_sixteen - #4 }
12699     \__fp_exp_pos_large:NnnNwn
12700 }

```

```

12701         #5
12702         {#4}
12703         #1 #2 0
12704         \tex_romannumeral:D
12705         \fi:
12706         \fi:
12707         \exp_after:wN \c_zero
12708     }
12709     \cs_new:Npn \__fp_exp_overflow:
12710     { + \c_two * \c__fp_max_exponent_int ; {1000} {0000} {0000} {0000} ; }
(End definition for \__fp_exp_normal:w and \__fp_exp_pos:Nnnwn.)

```

```

\__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.

```

12711 \cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
12712 {
12713     #6
12714     \__fp_pack_twice_four:wNNNNNNNN
12715     \__fp_pack_twice_four:wNNNNNNNN
12716     \__fp_pack_twice_four:wNNNNNNNN
12717     \__fp_exp_Taylor_ii:ww
12718     ; #2#3#4 0000 0000 ;
12719 }
12720 \cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;
12721 { \__fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s__stop }
12722 \cs_new:Npn \__fp_exp_Taylor_loop:www #1; #2; #3;
12723 {
12724     \if_int_compare:w #1 = \c_one
12725     \exp_after:wN \__fp_exp_Taylor_break:Nww
12726     \fi:
12727     \__fp_fixed_div_int:wwN #3 ; #1 ;
12728     \__fp_fixed_add_one:wN
12729     \__fp_fixed_mul:wwN #2 ;
12730     {
12731         \exp_after:wN \__fp_exp_Taylor_loop:www
12732         \int_use:N \__int_eval:w #1 - 1 ;
12733         #2 ;
12734     }
12735 }
12736 \cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2; #3 \s__stop
12737 { \__fp_fixed_add_one:wN #2 ; }

```

(End definition for __fp_exp_Taylor:Nnnwn. This function is documented on page ??.)

```

\__fp_exp_pos_large:NnnNwn
\__fp_exp_large_after:wwN
    \__fp_exp_large:w
    \__fp_exp_large_v:wN
    \__fp_exp_large_iv:wN
    \__fp_exp_large_iii:wN
    \__fp_exp_large_ii:wN
    \__fp_exp_large_i:wN
    \__fp_exp_large_:wN

```

The first two arguments are irrelevant (a rounding digit, and a brace group with 8 zeros). The third argument is the integer part of our number, then we have the decimal part delimited by a semicolon, and finally the exponent, in the range $[0, 5]$. Remove leading zeros from the integer part: putting #4 in there too ensures that an integer part of 0 is

also removed. Then read digits one by one, looking up $\exp(\langle digit \rangle \cdot 10^{\langle exponent \rangle})$ in a table, and multiplying that to the current total. The loop is done by having the auxiliary for one exponent call the auxiliary for the next exponent. The current total is expressed by leaving the exponent behind in the input stream (we are currently within an `__int_eval:w`), and keeping track of a fixed point number, #1 for the numbered auxiliaries. Our usage of `\if_case:w` is somewhat dirty for optimization: \TeX jumps to the appropriate case, but we then close the `\if_case:w` “by hand”, using `\or:` and `\fi:` as delimiters.

```

12738 \cs_new:Npn \__fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
12739 {
12740   \exp_after:wN \exp_after:wN
12741   \cs:w \__fp_exp_large\tex_romannumeral:D #6:wN \exp_after:wN \cs_end:
12742   \exp_after:wN \c__fp_one_fixed_tl
12743   \exp_after:wN ;
12744   \__int_value:w #3 #4 \exp_stop_f:
12745   #5 00000 ;
12746 }
12747 \cs_new:Npn \__fp_exp_large:w #1 \or: #2 \fi:
12748 { \fi: \__fp_fixed_mul:wwn #1; }
12749 \cs_new:Npn \__fp_exp_large_v:wN #1; #2
12750 {
12751   \if_case:w #2 ~ \exp_after:wN \__fp_fixed_continue:wn \or:
12752   + 4343 \__fp_exp_large:w {8806}{8182}{2566}{2921}{5872}{6150} \or:
12753   + 8686 \__fp_exp_large:w {7756}{0047}{2598}{6861}{0458}{3204} \or:
12754   + 13029 \__fp_exp_large:w {6830}{5723}{7791}{4884}{1932}{7351} \or:
12755   + 17372 \__fp_exp_large:w {6015}{5609}{3095}{3052}{3494}{7574} \or:
12756   + 21715 \__fp_exp_large:w {5297}{7951}{6443}{0315}{3251}{3576} \or:
12757   + 26058 \__fp_exp_large:w {4665}{6719}{0099}{3379}{5527}{2929} \or:
12758   + 30401 \__fp_exp_large:w {4108}{9724}{3326}{3186}{5271}{5665} \or:
12759   + 34744 \__fp_exp_large:w {3618}{6973}{3140}{0875}{3856}{4102} \or:
12760   + 39087 \__fp_exp_large:w {3186}{9209}{6113}{3900}{6705}{9685} \or:
12761   \fi:
12762   #1;
12763   \__fp_exp_large_iv:wN
12764 }
12765 \cs_new:Npn \__fp_exp_large_iv:wN #1; #2
12766 {
12767   \if_case:w #2 ~ \exp_after:wN \__fp_fixed_continue:wn \or:
12768   + 435 \__fp_exp_large:w {1970}{0711}{1401}{7046}{9938}{8888} \or:
12769   + 869 \__fp_exp_large:w {3881}{1801}{9428}{4368}{5764}{8232} \or:
12770   + 1303 \__fp_exp_large:w {7646}{2009}{8905}{4704}{8893}{1073} \or:
12771   + 1738 \__fp_exp_large:w {1506}{3559}{7005}{0524}{9009}{7592} \or:
12772   + 2172 \__fp_exp_large:w {2967}{6283}{8402}{3667}{0689}{6630} \or:
12773   + 2606 \__fp_exp_large:w {5846}{4389}{5650}{2114}{7278}{5046} \or:
12774   + 3041 \__fp_exp_large:w {1151}{7900}{5080}{6878}{2914}{4154} \or:
12775   + 3475 \__fp_exp_large:w {2269}{1083}{0850}{6857}{8724}{4002} \or:
12776   + 3909 \__fp_exp_large:w {4470}{3047}{3316}{5442}{6408}{6591} \or:
12777   \fi:
12778   #1;
12779   \__fp_exp_large_iii:wN

```

```

12780 }
12781 \cs_new:Npn \__fp_exp_large_iii:wN #1; #2
12782 {
12783   \if_case:w #2 ~ \exp_after:wN \__fp_fixed_continue:wn \or:
12784     + 44 \__fp_exp_large:w {2688}{1171}{4181}{6135}{4484}{1263} \or:
12785     + 87 \__fp_exp_large:w {7225}{9737}{6812}{5749}{2581}{7748} \or:
12786     + 131 \__fp_exp_large:w {1942}{4263}{9524}{1255}{9365}{8421} \or:
12787     + 174 \__fp_exp_large:w {5221}{4696}{8976}{4143}{9505}{8876} \or:
12788     + 218 \__fp_exp_large:w {1403}{5922}{1785}{2837}{4107}{3977} \or:
12789     + 261 \__fp_exp_large:w {3773}{0203}{0092}{9939}{8234}{0143} \or:
12790     + 305 \__fp_exp_large:w {1014}{2320}{5473}{5004}{5094}{5533} \or:
12791     + 348 \__fp_exp_large:w {2726}{3745}{7211}{2566}{5673}{6478} \or:
12792     + 391 \__fp_exp_large:w {7328}{8142}{2230}{7421}{7051}{8866} \or:
12793   \fi:
12794   #1;
12795   \__fp_exp_large_ii:wN
12796 }
12797 \cs_new:Npn \__fp_exp_large_ii:wN #1; #2
12798 {
12799   \if_case:w #2 ~ \exp_after:wN \__fp_fixed_continue:wn \or:
12800     + 5 \__fp_exp_large:w {2202}{6465}{7948}{0671}{6516}{9579} \or:
12801     + 9 \__fp_exp_large:w {4851}{6519}{5409}{7902}{7796}{9107} \or:
12802     + 14 \__fp_exp_large:w {1068}{6474}{5815}{2446}{2146}{9905} \or:
12803     + 18 \__fp_exp_large:w {2353}{8526}{6837}{0199}{8540}{7900} \or:
12804     + 22 \__fp_exp_large:w {5184}{7055}{2858}{7072}{4640}{8745} \or:
12805     + 27 \__fp_exp_large:w {1142}{0073}{8981}{5684}{2836}{6296} \or:
12806     + 31 \__fp_exp_large:w {2515}{4386}{7091}{9167}{0062}{6578} \or:
12807     + 35 \__fp_exp_large:w {5540}{6223}{8439}{3510}{0525}{7117} \or:
12808     + 40 \__fp_exp_large:w {1220}{4032}{9431}{7840}{8020}{0271} \or:
12809   \fi:
12810   #1;
12811   \__fp_exp_large_i:wN
12812 }
12813 \cs_new:Npn \__fp_exp_large_i:wN #1; #2
12814 {
12815   \if_case:w #2 ~ \exp_after:wN \__fp_fixed_continue:wn \or:
12816     + 1 \__fp_exp_large:w {2718}{2818}{2845}{9045}{2353}{6029} \or:
12817     + 1 \__fp_exp_large:w {7389}{0560}{9893}{0650}{2272}{3043} \or:
12818     + 2 \__fp_exp_large:w {2008}{5536}{9231}{8766}{7740}{9285} \or:
12819     + 2 \__fp_exp_large:w {5459}{8150}{0331}{4423}{9078}{1103} \or:
12820     + 3 \__fp_exp_large:w {1484}{1315}{9102}{5766}{0342}{1116} \or:
12821     + 3 \__fp_exp_large:w {4034}{2879}{3492}{7351}{2260}{8387} \or:
12822     + 4 \__fp_exp_large:w {1096}{6331}{5842}{8458}{5992}{6372} \or:
12823     + 4 \__fp_exp_large:w {2980}{9579}{8704}{1728}{2747}{4359} \or:
12824     + 4 \__fp_exp_large:w {8103}{0839}{2757}{5384}{0077}{1000} \or:
12825   \fi:
12826   #1;
12827   \__fp_exp_large_:wN
12828 }
12829 \cs_new:Npn \__fp_exp_large_:wN #1; #2

```

```

12830 {
12831     \if_case:w #2 ~          \exp_after:wN \__fp_fixed_continue:wn \or:
12832     + 1 \__fp_exp_large:w {1105}{1709}{1807}{5647}{6248}{1171} \or:
12833     + 1 \__fp_exp_large:w {1221}{4027}{5816}{0169}{8339}{2107} \or:
12834     + 1 \__fp_exp_large:w {1349}{8588}{0757}{6003}{1039}{8374} \or:
12835     + 1 \__fp_exp_large:w {1491}{8246}{9764}{1270}{3178}{2485} \or:
12836     + 1 \__fp_exp_large:w {1648}{7212}{7070}{0128}{1468}{4865} \or:
12837     + 1 \__fp_exp_large:w {1822}{1188}{0039}{0508}{9748}{7537} \or:
12838     + 1 \__fp_exp_large:w {2013}{7527}{0747}{0476}{5216}{2455} \or:
12839     + 1 \__fp_exp_large:w {2225}{5409}{2849}{2467}{6045}{7954} \or:
12840     + 1 \__fp_exp_large:w {2459}{6031}{1115}{6949}{6638}{0013} \or:
12841     \fi:
12842     #1;
12843     \__fp_exp_large_after:wwn
12844 }
12845 \cs_new:Npn \__fp_exp_large_after:wwn #1; #2; #3
12846 {
12847     \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
12848     \__fp_fixed_mul:wwn #1;
12849 }

```

(End definition for __fp_exp_pos_large:NnnNwn and others.)

30.3 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.

```

12850 \cs_new:cpn { __fp_ \iow_char:N \^_o:ww }
12851 \s__fp __fp_chk:w #1#2#3; \s__fp __fp_chk:w #4#5#6;
12852 {
12853   \if_meaning:w 0 #4
12854     \__fp_case_return_o:Nw \c_one_fp
12855   \fi:
12856   \if_case:w #2 \exp_stop_f:
12857     \exp_after:wN \use_i:nn
12858   \or:
12859     \__fp_case_return_o:Nw \c_nan_fp
12860   \else:
12861     \exp_after:wN \__fp_pow_neg:www
12862     \tex_romannumeral:D -'0 \exp_after:wN \use:nn
12863   \fi:
12864   {
12865     \if_meaning:w 1 #1
12866       \exp_after:wN \__fp_pow_normal:ww
12867     \else:
12868       \exp_after:wN \__fp_pow_zero_or_inf:ww
12869     \fi:
12870     \s__fp __fp_chk:w #1#2#3;
12871   }
12872   { \s__fp __fp_chk:w #4#5#6; \s__fp __fp_chk:w #1#2#3; }
12873   \s__fp __fp_chk:w #4#5#6;
12874 }

```

(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.

```

12875 \cs_new:Npn \__fp_pow_zero_or_inf:ww \s__fp __fp_chk:w #1#2; \s__fp __fp_chk:w #3#4
12876 {
12877   \if_meaning:w 1 #4
12878     \__fp_case_return_same_o:w
12879   \fi:
12880   \if_meaning:w #1 #4
12881     \__fp_case_return_o:Nw \c_zero_fp
12882   \fi:
12883   \if_meaning:w 0 #1

```

```

12884     \__fp_case_use:nw
12885     {
12886         \__fp_division_by_zero_o:NNww \c_inf_fp ^
12887         \s__fp \__fp_chk:w #1 #2 ;
12888     }
12889     \else:
12890         \__fp_case_return_o:Nw \c_inf_fp
12891     \fi:
12892     \s__fp \__fp_chk:w #3#4
12893 }
(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 .

```

12894 \cs_new:Npn \__fp_pow_normal:ww \s__fp \__fp_chk:w 1 #1#2#3; \s__fp \__fp_chk:w #4#5
12895 {
12896     \if_int_compare:w \pdfTeX_strcmp:D { #2 #3 }
12897         { 1 {1000} {0000} {0000} {0000} } = \c_zero
12898     \if_int_compare:w #4 #1 = 32 \exp_stop_f:
12899         \exp_after:wN \__fp_case_return_ii_o:ww
12900     \fi:
12901     \__fp_case_return_o:Nww \c_one_fp
12902 \fi:
12903 \if_case:w #4 \exp_stop_f:
12904 \or:
12905     \exp_after:wN \__fp_pow_npos:Nww
12906     \exp_after:wN #5
12907 \or:
12908     \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
12909     \if_int_compare:w #2 > \c_zero
12910         \exp_after:wN \__fp_case_return_o:Nww
12911         \exp_after:wN \c_inf_fp
12912     \else:
12913         \exp_after:wN \__fp_case_return_o:Nww
12914         \exp_after:wN \c_zero_fp
12915     \fi:
12916 \or:
12917     \__fp_case_return_ii_o:ww

```

```

12918 \fi:
12919 \s__fp \__fp_chk:w 1 #1 {#2} #3 ;
12920 \s__fp \__fp_chk:w #4 #5
12921 }

```

(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.

```

12922 \cs_new:Npn \__fp_pow_npos:Nww #1 \s__fp \__fp_chk:w 1#2#3
12923 {
12924   \exp_after:wN \__fp_sanitize:Nw
12925   \exp_after:wN 0
12926   \__int_value:w
12927   \if:w #1 \if_int_compare:w #3 > \c_zero 0 \else: 2 \fi:
12928     \exp_after:wN \__fp_pow_npos_aux:NNnww
12929     \exp_after:wN +
12930     \exp_after:wN \__fp_fixed_to_float:wN
12931   \else:
12932     \exp_after:wN \__fp_pow_npos_aux:NNnww
12933     \exp_after:wN -
12934     \exp_after:wN \__fp_fixed_inv_to_float:wN
12935   \fi:
12936   {#3}
12937 }

```

(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 $-\ln(x)$.

```

12938 \cs_new:Npn \__fp_pow_npos_aux:NNnww #1#2#3#4#5; \s__fp \__fp_chk:w 1#6#7#8;
12939 {
12940   #1
12941   \__int_eval:w
12942   \__fp_ln_significand:NNNNnnnN #4#5
12943   \__fp_pow_exponent:wnN {#3}
12944   \__fp_fixed_mul:wwN #8 {0000}{0000} ;
12945   \__fp_pow_B:wwN #7;
12946   #1 #2 0 % fixed_to_float:wN
12947 }
12948 \cs_new:Npn \__fp_pow_exponent:wnN #1; #2
12949 {
12950   \if_int_compare:w #2 > \c_zero
12951     \exp_after:wN \__fp_pow_exponent:Nwnnnnnwn % n\ln(10) - (-\ln(x))
12952     \exp_after:wN +

```

```

12953     \else:
12954         \exp_after:wN \__fp_pow_exponent:Nwnnnnnwn % -( |n|\ln(10) + (-\ln(x)) )
12955         \exp_after:wN -
12956     \fi:
12957     #2; #1;
12958 }
12959 \cs_new:Npn \__fp_pow_exponent:Nwnnnnnwn #1#2; #3#4#5#6#7#8; #9
12960 { %^A todo: use that in ln.
12961     \exp_after:wN \__fp_fixed_mul_after:wn
12962     \int_use:N \__int_eval:w \c__fp_leading_shift_int
12963     \exp_after:wN \__fp_pack:NNNNNwn
12964     \int_use:N \__int_eval:w \c__fp_middle_shift_int
12965     #1#2*23025 - #1 #3
12966     \exp_after:wN \__fp_pack:NNNNNwn
12967     \int_use:N \__int_eval:w \c__fp_middle_shift_int
12968     #1 #2*8509 - #1 #4
12969     \exp_after:wN \__fp_pack:NNNNNwn
12970     \int_use:N \__int_eval:w \c__fp_middle_shift_int
12971     #1 #2*2994 - #1 #5
12972     \exp_after:wN \__fp_pack:NNNNNwn
12973     \int_use:N \__int_eval:w \c__fp_middle_shift_int
12974     #1 #2*0456 - #1 #6
12975     \exp_after:wN \__fp_pack:NNNNNwn
12976     \int_use:N \__int_eval:w \c__fp_trailing_shift_int
12977     #1 #2*8401 - #1 #7
12978     #1 ( #2*7991 - #8 ) / 1 0000 ; {#9} ;
12979 }
12980 \cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
12981 {
12982     \if_int_compare:w #7 < \c_zero
12983         \exp_after:wN \__fp_pow_C_neg:w \__int_value:w -
12984     \else:
12985         \if_int_compare:w #7 < 22 \exp_stop_f:
12986             \exp_after:wN \__fp_pow_C_pos:w \__int_value:w
12987         \else:
12988             \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
12989         \fi:
12990     \fi:
12991     #7 \exp_after:wN ;
12992     \int_use:N \__int_eval:w 10 0000 + #1 \__int_eval_end:
12993     #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^A todo: how many 0?
12994 }
12995 \cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
12996 {
12997     + \c_two * \c__fp_max_exponent_int
12998     \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl ;
12999 }
13000 \cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
13001 {
13002     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w

```

```

13003     \prg_replicate:nn {#1} {0}
13004   }
13005   \cs_new:Npn \__fp_pow_C_pos:w #1; 1
13006     { \__fp_pow_C_pos_loop:wN #1; }
13007   \cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
13008     {
13009       \if_meaning:w 0 #1
13010         \exp_after:wN \__fp_pow_C_pack:w
13011         \exp_after:wN #2
13012       \else:
13013         \if_meaning:w 0 #2
13014           \exp_after:wN \__fp_pow_C_pos_loop:wN \__int_value:w
13015         \else:
13016           \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
13017         \fi:
13018         \__int_eval:w #1 - \c_one \exp_after:wN ;
13019       \fi:
13020     }
13021   \cs_new:Npn \__fp_pow_C_pack:w
13022     { \exp_after:wN \__fp_exp_large_v:wN \c__fp_one_fixed_t1 ; }
(End definition for \__fp_pow_npos_aux:NNnw.)

```

$\backslash_fp_pow_neg:www$
 $\backslash_fp_pow_neg_aux:wNN$

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 $\backslash_fp_pow_neg_aux:wNN$. 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 $\backslash_fp_pow_neg:www$ is called, $(-0.1)**(12345.6)$ will give $+0$ rather than complaining that the sign is not defined.

```

13023   \cs_new:Npn \__fp_pow_neg:www \s__fp \__fp_chk:w #1#2; #3; #4;
13024   {
13025     \if_case:w \__fp_pow_neg_case:w #4 ;
13026       \exp_after:wN \__fp_pow_neg_aux:wNN
13027     \or:
13028       \if_int_compare:w \__int_eval:w #1 / \c_two = \c_one
13029         \__fp_invalid_operation_o:Nww ^ #3; #4;
13030         \tex_romannumeral:D -'0
13031         \exp_after:wN \exp_after:wN
13032         \exp_after:wN \__fp_use_none_until_s:w
13033       \fi:
13034     \fi:
13035     \__fp_exp_after_o:w
13036     \s__fp \__fp_chk:w #1#2;
13037   }
13038   \cs_new:Npn \__fp_pow_neg_aux:wNN #1 \s__fp \__fp_chk:w #2#3
13039   {
13040     \exp_after:wN \__fp_exp_after_o:w
13041     \exp_after:wN \s__fp
13042     \exp_after:wN \__fp_chk:w
13043     \exp_after:wN #2

```

```

13044 \int_use:N \__int_eval:w \c_two - #3 \__int_eval_end:
13045 }

```

(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`.

```

13046 \cs_new:Npn \__fp_pow_neg_case:w \s__fp \__fp_chk:w #1#2#3;
13047 {
13048   \if_case:w #1 \exp_stop_f:
13049     \c_minus_one
13050   \or:   \__fp_pow_neg_case_aux:nnnnn #3
13051   \else: \c_one
13052   \fi:
13053 }
13054 \cs_new:Npn \__fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
13055 {
13056   \if_int_compare:w #1 > \c_eight
13057     \if_int_compare:w #1 > \c_sixteen
13058       \c_minus_one
13059     \else:
13060       \exp_after:wN \exp_after:wN
13061       \exp_after:wN \__fp_pow_neg_case_aux:NNNNNNNNw
13062       \prg_replicate:nn { \c_sixteen - #1 } { 0 } #4#5 ;
13063     \fi:
13064   \else:
13065     \if_int_compare:w #1 > \c_zero
13066       \if_int_compare:w #4#5 = \c_zero
13067         \exp_after:wN \exp_after:wN
13068         \exp_after:wN \__fp_pow_neg_case_aux:NNNNNNNNw
13069         \prg_replicate:nn { \c_eight - #1 } { 0 } #2#3 ;
13070       \else:
13071         \c_one
13072       \fi:
13073     \else:
13074       \c_one
13075     \fi:
13076   \fi:
13077 }
13078 \cs_new:Npn \__fp_pow_neg_case_aux:NNNNNNNNw #1#2#3#4#5#6#7#8#9;
13079 {
13080   \if_int_compare:w 0 #9 = \c_zero

```

```

13081     \if_int_odd:w #8 \exp_stop_f:
13082         \c_zero
13083     \else:
13084         \c_minus_one
13085     \fi:
13086 \else:
13087     \c_one
13088 \fi:
13089 }
(End definition for \__fp_pow_neg_case:w, \__fp_pow_neg_case_aux:nnnnn, and \__fp_pow_neg_case_aux:NNNNNNNNw.)
13090 </initex | package>

```

31 l3fp-trig Implementation

```

13091 <*initex | package>
13092 <@@=fp>

```

31.1 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.

31.1.1 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 $\sin(x) = \#3 \sin|x|$.

```

13093 \cs_new:Npn \__fp_sin_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
13094 {
13095   \if_case:w #2 \exp_stop_f:
13096     \__fp_case_return_same_o:w
13097   \or: \__fp_case_use:nw
13098     {
13099       \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
13100       \__fp_ep_to_float:wwN #3 \c_zero
13101     }
13102   \or: \__fp_case_use:nw
13103     { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
13104   \else: \__fp_case_return_same_o:w
13105   \fi:
13106   \s__fp \__fp_chk:w #2 #3 #4;
13107 }
(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 $\cos(x) = +\sin(\pi/2 + |x|)$.

```

13108 \cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
13109 {
13110   \if_case:w #2 \exp_stop_f:
13111     \__fp_case_return_o:Nw \c_one_fp
13112   \or: \__fp_case_use:nw
13113     {
13114       \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
13115       \__fp_ep_to_float:wwN 0 \c_two
13116     }
13117   \or: \__fp_case_use:nw
13118     { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
13119   \else: \__fp_case_return_same_o:w
13120   \fi:
13121   \s__fp \__fp_chk:w #2 #3;
13122 }
(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 $\csc(x) = \#3(\sin|x|)^{-1}$.


```

13123 \cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
13124 {
13125     \if_case:w #2 \exp_stop_f:
13126         \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
13127     \or: \__fp_case_use:nw
13128         {
13129             \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
13130             \__fp_ep_inv_to_float:wwN #3 \c_zero
13131         }
13132     \or: \__fp_case_use:nw
13133         { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
13134     \else: \__fp_case_return_same_o:w
13135     \fi:
13136     \s__fp \__fp_chk:w #2 #3 #4;
13137 }

```

(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 $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```

13138 \cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3; @
13139 {
13140     \if_case:w #2 \exp_stop_f:
13141         \__fp_case_return_o:Nw \c_one_fp
13142     \or: \__fp_case_use:nw
13143         {
13144             \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
13145             \__fp_ep_inv_to_float:wwN 0 \c_two
13146         }
13147     \or: \__fp_case_use:nw
13148         { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } }
13149     \else: \__fp_case_return_same_o:w
13150     \fi:
13151     \s__fp \__fp_chk:w #2 #3;
13152 }

```

(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.

```

13153 \cs_new:Npn \__fp_tan_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
13154 {
13155     \if_case:w #2 \exp_stop_f:
13156         \__fp_case_return_same_o:w
13157     \or: \__fp_case_use:nw
13158         {

```

```

13159         \__fp_trig:NNNNNwn #1
13160         \__fp_tan_series_o:NNwww 0 #3 \c_one
13161     }
13162 \or: \__fp_case_use:nw
13163     { \__fp_invalid_operation_o:fw { #1 { tan } { tand } } }
13164 \else: \__fp_case_return_same_o:w
13165 \fi:
13166 \s__fp \__fp_chk:w #2 #3 #4;
13167 }

```

(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.

```

13168 \cs_new:Npn \__fp_cot_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
13169 {
13170     \if_case:w #2 \exp_stop_f:
13171         \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
13172 \or: \__fp_case_use:nw
13173     {
13174         \__fp_trig:NNNNNwn #1
13175         \__fp_tan_series_o:NNwww 2 #3 \c_three
13176     }
13177 \or: \__fp_case_use:nw
13178     { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
13179 \else: \__fp_case_return_same_o:w
13180 \fi:
13181 \s__fp \__fp_chk:w #2 #3 #4;
13182 }
13183 \cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:
13184 {
13185     \fi:
13186     \token_if_eq_meaning:NNTF 0 #1
13187     { \exp_args:Nnf \__fp_division_by_zero_o:Nnw \c_inf_fp }
13188     { \exp_args:Nnf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
13189     {#2}
13190 }

```

(End definition for __fp_cot_o:w. This function is documented on page ??.)

31.1.2 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.

```

13191 \cs_new:Npn \__fp_trig:NNNNwn #1#2#3#4#5 \s_fp \__fp_chk:w 1#6#7#8;
13192 {
13193   \exp_after:wN #2
13194   \exp_after:wN #3
13195   \exp_after:wN #4
13196   \int_use:N \__int_eval:w #5
13197   \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
13198   \if_int_compare:w #7 > #1 \c_zero \c_one
13199   #1 \__fp_trig_large:ww \__fp_trigd_large:ww
13200   \else:
13201   #1 \__fp_trig_small:ww \__fp_trigd_small:ww
13202   \fi:
13203   #7,#8{0000}{0000};
13204 }
(End definition for \__fp_trig:NNNNwn.)

```

31.1.3 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.

```

13205 \cs_new:Npn \__fp_trig_small:ww #1,#2;
13206 { \__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.

```

13207 \cs_new:Npn \__fp_trigd_small:ww #1,#2;
13208 {
13209   \__fp_ep_mul_raw:wwwN
13210   -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;

```

```

13211     \_fp_trig_small:ww
13212 }
(End definition for \_fp_trigd_small:ww.)

```

31.1.4 Argument reduction in degrees

```

\_fp_trigd_large:ww
\_fp_trigd_large_auxi:nnnwNNNN
\_fp_trigd_large_auxii:wNw
\_fp_trigd_large_auxiii:www

```

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).

```

13213 \cs_new:Npn \_fp_trigd_large:ww #1, #2#3#4#5#6#7;
13214 {
13215     \exp_after:wN \_fp_pack_eight:wNNNNNNNN
13216     \exp_after:wN \_fp_pack_eight:wNNNNNNNN
13217     \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
13218     \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
13219     \exp_after:wN \_fp_trigd_large_auxi:nnnwNNNN
13220     \exp_after:wN ;
13221     \tex_romannumeral:D -'0
13222     \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
13223     #2#3#4#5#6#7 0000 0000 0000 !
13224 }
13225 \cs_new:Npn \_fp_trigd_large_auxi:nnnwNNNN #1#2#3#4#5; #6#7#8#9
13226 {
13227     \exp_after:wN \_fp_trigd_large_auxii:wNw
13228     \int_use:N \_int_eval:w #1 + #2
13229     - (#1 + #2 - \c_four) / \c_nine * \c_nine \_int_eval_end:
13230     #3;
13231     #4; #5{#6#7#8#9};
13232 }
13233 \cs_new:Npn \_fp_trigd_large_auxii:wNw #1; #2#3;
13234 {
13235     + (#1#2 - \c_four) / \c_nine * \c_two
13236     \exp_after:wN \_fp_trigd_large_auxiii:www
13237     \int_use:N \_int_eval:w #1#2

```


13257 4564487476673440588967976342265350901138027662530859560728427267 ~
13258 5795803689291184611457865287796741073169983922923996693740907757 ~
13259 3077746396925307688717392896217397661693362390241723629011832380 ~
13260 1142226997557159404618900869026739561204894109369378440855287230 ~
13261 9994644340024867234773945961089832309678307490616698646280469944 ~
13262 8652187881574786566964241038995874139348609983868099199962442875 ~
13263 5851711788584311175187671605465475369880097394603647593337680593 ~
13264 0249449663530532715677550322032477781639716602294674811959816584 ~
13265 0606016803035998133911987498832786654435279755070016240677564388 ~
13266 8495713108801221993761476813777647378906330680464579784817613124 ~
13267 2731406996077502450029775985708905690279678513152521001631774602 ~
13268 0924811606240561456203146484089248459191435211575407556200871526 ~
13269 6068022171591407574745827225977462853998751553293908139817724093 ~
13270 5825479707332871904069997590765770784934703935898280871734256403 ~
13271 6689511662545705943327631268650026122717971153211259950438667945 ~
13272 0376255608363171169525975812822494162333431451061235368785631136 ~
13273 3669216714206974696012925057833605311960859450983955671870995474 ~
13274 6510431623815517580839442979970999505254387566129445883306846050 ~
13275 7852915151410404892988506388160776196993073410389995786918905980 ~
13276 9373777206187543222718930136625526123878038753888110681406765434 ~
13277 0828278526933426799556070790386060352738996245125995749276297023 ~
13278 5940955843011648296411855777124057544494570217897697924094903272 ~
13279 9477021664960356531815354400384068987471769158876319096650696440 ~
13280 4776970687683656778104779795450353395758301881838687937766124814 ~
13281 9530599655802190835987510351271290432315804987196868777594656634 ~
13282 6221034204440855497850379273869429353661937782928735937843470323 ~
13283 0237145837923557118636341929460183182291964165008783079331353497 ~
13284 7909974586492902674506098936890945883050337030538054731232158094 ~
13285 3197676032283131418980974982243833517435698984750103950068388003 ~
13286 9786723599608024002739010874954854787923568261139948903268997427 ~
13287 0834961149208289037767847430355045684560836714793084567233270354 ~
13288 8539255620208683932409956221175331839402097079357077496549880868 ~
13289 6066360968661967037474542102831219251846224834991161149566556037 ~
13290 9696761399312829960776082779901007830360023382729879085402387615 ~
13291 5744543092601191005433799838904654921248295160707285300522721023 ~
13292 6017523313173179759311050328155109373913639645305792607180083617 ~
13293 9548767246459804739772924481092009371257869183328958862839904358 ~
13294 6866663975673445140950363732719174311388066383072592302759734506 ~
13295 0548212778037065337783032170987734966568490800326988506741791464 ~
13296 6835082816168533143361607309951498531198197337584442098416559541 ~
13297 5225064339431286444038388356150879771645017064706751877456059160 ~
13298 8716857857939226234756331711132998655941596890719850688744230057 ~
13299 5191977056900382183925622033874235362568083541565172971088117217 ~
13300 9593683256488518749974870855311659830610139214454460161488452770 ~
13301 2511411070248521739745103866736403872860099674893173561812071174 ~
13302 0478899368886556923078485023057057144063638632023685201074100574 ~
13303 8592281115721968003978247595300166958522123034641877365043546764 ~
13304 6456565971901123084767099309708591283646669191776938791433315566 ~
13305 5066981321641521008957117286238426070678451760111345080069947684 ~
13306 2235698962488051577598095339708085475059753626564903439445420581 ~

13307 7886435683042000315095594743439252544850674914290864751442303321 ~
13308 3324569511634945677539394240360905438335528292434220349484366151 ~
13309 4663228602477666660495314065734357553014090827988091478669343492 ~
13310 2737602634997829957018161964321233140475762897484082891174097478 ~
13311 2637899181699939487497715198981872666294601830539583275209236350 ~
13312 6853889228468247259972528300766856937583659722919824429747406163 ~
13313 8183113958306744348516928597383237392662402434501997809940402189 ~
13314 6134834273613676449913827154166063424829363741850612261086132119 ~
13315 9863346284709941839942742955915628333990480382117501161211667205 ~
13316 1912579303552929241134403116134112495318385926958490443846807849 ~
13317 0973982808855297045153053991400988698840883654836652224668624087 ~
13318 2540140400911787421220452307533473972538149403884190586842311594 ~
13319 6322744339066125162393106283195323883392131534556381511752035108 ~
13320 7459558201123754359768155340187407394340363397803881721004531691 ~
13321 8295194879591767395417787924352761740724605939160273228287946819 ~
13322 3649128949714953432552723591659298072479985806126900733218844526 ~
13323 7943350455801952492566306204876616134365339920287545208555344144 ~
13324 0990512982727454659118132223284051166615650709837557433729548631 ~
13325 2041121716380915606161165732000083306114606181280326258695951602 ~
13326 4632166138576614804719932707771316441201594960110632830520759583 ~
13327 4850305079095584982982186740289838551383239570208076397550429225 ~
13328 9847647071016426974384504309165864528360324933604354657237557916 ~
13329 1366324120457809969715663402215880545794313282780055246132088901 ~
13330 8742121092448910410052154968097113720754005710963406643135745439 ~
13331 9159769435788920793425617783022237011486424925239248728713132021 ~
13332 7667360756645598272609574156602343787436291321097485897150713073 ~
13333 9104072643541417970572226547980381512759579124002534468048220261 ~
13334 7342299001020483062463033796474678190501811830375153802879523433 ~
13335 4195502135689770912905614317878792086205744999257897569018492103 ~
13336 2420647138519113881475640209760554895793785141404145305151583964 ~
13337 2823265406020603311891586570272086250269916393751527887360608114 ~
13338 5569484210322407772727421651364234366992716340309405307480652685 ~
13339 0930165892136921414312937134106157153714062039784761842650297807 ~
13340 8606266969960809184223476335047746719017450451446166382846208240 ~
13341 8673595102371302904443779408535034454426334130626307459513830310 ~
13342 2293146934466832851766328241515210179422644395718121717021756492 ~
13343 1964449396532222187658488244511909401340504432139858628621083179 ~
13344 3939608443898019147873897723310286310131486955212620518278063494 ~
13345 5711866277825659883100535155231665984394090221806314454521212978 ~
13346 9734471488741258268223860236027109981191520568823472398358013366 ~
13347 0683786328867928619732367253606685216856320119489780733958419190 ~
13348 6659583867852941241871821727987506103946064819585745620060892122 ~
13349 8416394373846549589932028481236433466119707324309545859073361878 ~
13350 6290631850165106267576851216357588696307451999220010776676830946 ~
13351 9814975622682434793671310841210219520899481912444048751171059184 ~
13352 4139907889455775184621619041530934543802808938628073237578615267 ~
13353 7971143323241969857805637630180884386640607175368321362629671224 ~
13354 2609428540110963218262765120117022552929289655594608204938409069 ~
13355 0760692003954646191640021567336017909631872891998634341086903200 ~
13356 5796637103128612356988817640364252540837098108148351903121318624 ~

13357 7228181050845123690190646632235938872454630737272808789830041018 ~
13358 9485913673742589418124056729191238003306344998219631580386381054 ~
13359 2457893450084553280313511884341007373060595654437362488771292628 ~
13360 9807423539074061786905784443105274262641767830058221486462289361 ~
13361 9296692992033046693328438158053564864073184440599549689353773183 ~
13362 6726613130108623588021288043289344562140479789454233736058506327 ~
13363 0439981932635916687341943656783901281912202816229500333012236091 ~
13364 8587559201959081224153679499095448881099758919890811581163538891 ~
13365 6339402923722049848375224236209100834097566791710084167957022331 ~
13366 7897107102928884897013099533995424415335060625843921452433864640 ~
13367 3432440657317477553405404481006177612569084746461432976543900008 ~
13368 3826521145210162366431119798731902751191441213616962045693602633 ~
13369 6102355962140467029012156796418735746835873172331004745963339773 ~
13370 2477044918885134415363760091537564267438450166221393719306748706 ~
13371 2881595464819775192207710236743289062690709117919412776212245117 ~
13372 2354677115640433357720616661564674474627305622913332030953340551 ~
13373 3841718194605321501426328000879551813296754972846701883657425342 ~
13374 5016994231069156343106626043412205213831587971115075454063290657 ~
13375 0248488648697402872037259869281149360627403842332874942332178578 ~
13376 7750735571857043787379693402336902911446961448649769719434527467 ~
13377 4429603089437192540526658890710662062575509930379976658367936112 ~
13378 8137451104971506153783743579555867972129358764463093757203221320 ~
13379 2460565661129971310275869112846043251843432691552928458573495971 ~
13380 5042565399302112184947232132380516549802909919676815118022483192 ~
13381 5127372199792134331067642187484426215985121676396779352982985195 ~
13382 8545392106957880586853123277545433229161989053189053725391582222 ~
13383 9232597278133427818256064882333760719681014481453198336237910767 ~
13384 1255017528826351836492103572587410356573894694875444694018175923 ~
13385 0609370828146501857425324969212764624247832210765473750568198834 ~
13386 5641035458027261252285503154325039591848918982630498759115406321 ~
13387 0354263890012837426155187877318375862355175378506956599570028011 ~
13388 5841258870150030170259167463020842412449128392380525772514737141 ~
13389 2310230172563968305553583262840383638157686828464330456805994018 ~
13390 7001071952092970177990583216417579868116586547147748964716547948 ~
13391 8312140431836079844314055731179349677763739898930227765607058530 ~
13392 4083747752640947435070395214524701683884070908706147194437225650 ~
13393 2823145872995869738316897126851939042297110721350756978037262545 ~
13394 8141095038270388987364516284820180468288205829135339013835649144 ~
13395 3004015706509887926715417450706686888783438055583501196745862340 ~
13396 8059532724727843829259395771584036885940989939255241688378793572 ~
13397 7967951654076673927031256418760962190243046993485989199060012977 ~
13398 7469214532970421677817261517850653008552559997940209969455431545 ~
13399 2745856704403686680428648404512881182309793496962721836492935516 ~
13400 2029872469583299481932978335803459023227052612542114437084359584 ~
13401 9443383638388317751841160881711251279233374577219339820819005406 ~
13402 3292937775306906607415304997682647124407768817248673421685881509 ~
13403 9133422075930947173855159340808957124410634720893194912880783576 ~
13404 3115829400549708918023366596077070927599010527028150868897828549 ~
13405 4340372642729262103487013992868853550062061514343078665396085995 ~
13406 0058714939141652065302070085265624074703660736605333805263766757 ~


```

13407 2018839497277047222153633851135483463624619855425993871933367482 ~
13408 0422097449956672702505446423243957506869591330193746919142980999 ~
13409 3424230550172665212092414559625960554427590951996824313084279693 ~
13410 7113207021049823238195747175985519501864630940297594363194450091 ~
13411 9150616049228764323192129703446093584259267276386814363309856853 ~
13412 2786024332141052330760658841495858718197071242995959226781172796 ~
13413 4438853796763139274314227953114500064922126500133268623021550837
13414 \cs_end:
13415 }

```

(End definition for `__fp_trig_inverse_two_pi:`.)

`__fp_trig_large:ww` 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.

```

13416 \cs_new:Npn \__fp_trig_large:ww #1, #2#3#4#5#6;
13417 {
13418   \exp_after:wN \__fp_trig_large_auxi:wwwww
13419   \int_use:N \__int_eval:w (#1 - 32) / 64 \exp_after:wN ,
13420   \int_use:N \__int_eval:w (#1 - 4) / 8 \exp_after:wN ,
13421   \__int_value:w #1 \__fp_trig_inverse_two_pi: ;
13422   {#2}{#3}{#4}{#5} ;
13423 }
13424 \cs_new:Npn \__fp_trig_large_auxi:wwwww #1, #2, #3, #4!
13425 {
13426   \prg_replicate:nn {#1} { \__fp_trig_large_auxii:ww }
13427   \prg_replicate:nn { #2 - #1 * \c_eight }
13428   { \__fp_trig_large_auxiii:wNNNNNNNN }
13429   \prg_replicate:nn { #3 - #2 * \c_eight }
13430   { \__fp_trig_large_auxiv:wN }
13431   \prg_replicate:nn { \c_eight } { \__fp_pack_twice_four:wNNNNNNNN }
13432   \__fp_trig_large_auxv:www
13433   ;
13434 }
13435 \cs_new:Npn \__fp_trig_large_auxii:ww #1; #2 ~ { #1; }
13436 \cs_new:Npn \__fp_trig_large_auxiii:wNNNNNNNN
13437   #1; #2#3#4#5#6#7#8#9 { #1; }
13438 \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` 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.

```

13439 \cs_new:Npn \__fp_trig_large_auxv:www #1; #2; #3;
13440 {
13441   \exp_after:wN \__fp_use_i_until_s:nw
13442   \exp_after:wN \__fp_trig_large_auxvii:w
13443   \int_use:N \__int_eval:w \c__fp_leading_shift_int
13444   \prg_replicate:nn { \c_thirteen }
13445   { \__fp_trig_large_auxvi:wnnnnnnnnn }
13446   + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
13447   \__fp_use_i_until_s:nw
13448   ; #3 #1 ; ;
13449 }
13450 \cs_new:Npn \__fp_trig_large_auxvi:wnnnnnnnnn #1; #2#3#4#5#6#7#8#9
13451 {
13452   \exp_after:wN \__fp_trig_large_pack:NNNNNw
13453   \int_use:N \__int_eval:w \c__fp_middle_shift_int
13454   + #2*#9 + #3*#8 + #4*#7 + #5*#6
13455   #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
13456 }
13457 \cs_new:Npn \__fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
13458 { + #1#2#3#4#5 ; #6 }

```

(End definition for `__fp_trig_large_auxv:www`, `__fp_trig_large_auxvi:wnnnnnnnnn`, 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.

```

13459 \cs_new:Npn \__fp_trig_large_auxvii:w #1#2#3
13460 {
13461   \exp_after:wN \__fp_trig_large_auxviii:ww
13462   \int_use:N \__int_eval:w (#1#2#3 - 62) / 125 ;
13463   #1#2#3
13464 }
13465 \cs_new:Npn \__fp_trig_large_auxviii:ww #1;
13466 {
13467   + #1
13468   \if_int_odd:w #1 \exp_stop_f:

```

```

13469     \exp_after:wN \__fp_trig_large_auxix:Nw
13470     \exp_after:wN -
13471   \else:
13472     \exp_after:wN \__fp_trig_large_auxix:Nw
13473     \exp_after:wN +
13474   \fi:
13475 }
13476 \cs_new_nopar:Npn \__fp_trig_large_auxix:Nw
13477 {
13478   \exp_after:wN \__fp_use_i_until_s:nw
13479   \exp_after:wN \__fp_trig_large_auxxi:w
13480   \int_use:N \__int_eval:w \c__fp_leading_shift_int
13481   \prg_replicate:nn { \c_thirteen }
13482   { \__fp_trig_large_auxx:wNNNNN }
13483   + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
13484   ;
13485 }
13486 \cs_new:Npn \__fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
13487 {
13488   \exp_after:wN \__fp_trig_large_pack:NNNNNw
13489   \int_use:N \__int_eval:w \c__fp_middle_shift_int
13490   #2 \c_eight * #3#4#5#6
13491   #1; #2
13492 }
13493 \cs_new:Npn \__fp_trig_large_auxxi:w #1;
13494 {
13495   \exp_after:wN \__fp_ep_mul_raw:wwwN
13496   \int_use:N \__int_eval:w \c_zero \__fp_ep_to_ep_loop:N #1 ; ; !
13497   0,{7853}{9816}{3397}{4483}{0961}{5661};
13498   \__fp_trig_small:ww
13499 }

```

(End definition for __fp_trig_large_auxvii:w and __fp_trig_large_auxviii:w. These functions are documented on page ??.)

31.1.6 Computing the power series

__fp_sin_series_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 $\{1, 2, 5, 6, \dots\}$, we are near an extremum of the function and we use the series

$$\cos(x) = 1 - x^2 \left(\frac{1}{2!} - x^2 \left(\frac{1}{4!} - x^2 \left(\dots \right) \right) \right).$$

Otherwise, the series

$$\sin(x) = x \left(1 - x^2 \left(\frac{1}{3!} - x^2 \left(\frac{1}{5!} - x^2 \left(\dots \right) \right) \right) \right)$$

is used. Finally, the extended-precision number is converted to a floating point number with the given sign, and `_fp_sanitize:Nw` checks for overflow and underflow.

```

13500 \cs_new:Npn \_fp_sin_series_o:NNwww #1#2#3. #4;
13501 {
13502   \_fp_fixed_mul:wn #4; #4;
13503   {
13504     \exp_after:wN \_fp_sin_series_aux_o:NNwww
13505     \exp_after:wN #1
13506     \_int_value:w
13507     \if_int_odd:w \_int_eval:w ( #3 + \c_two ) / \c_four \_int_eval_end:
13508       #2
13509     \else:
13510       \if_meaning:w #2 0 2 \else: 0 \fi:
13511     \fi:
13512     {#3}
13513   }
13514 }
13515 \cs_new:Npn \_fp_sin_series_aux_o:NNwww #1#2#3 #4; #5,#6;
13516 {
13517   \if_int_odd:w \_int_eval:w #3 / \c_two \_int_eval_end:
13518   \exp_after:wN \use_i:nn
13519   \else:
13520   \exp_after:wN \use_ii:nn
13521   \fi:
13522   { % 1/18!
13523     \_fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0001}{5619}{2070};
13524     #4; {0000}{0000}{0000}{0477}{9477}{3324};
13525     \_fp_fixed_mul_sub_back:wwwn #4; {0000}{0000}{0011}{4707}{4559}{7730};
13526     \_fp_fixed_mul_sub_back:wwwn #4; {0000}{0000}{2087}{6756}{9878}{6810};
13527     \_fp_fixed_mul_sub_back:wwwn #4; {0000}{0027}{5573}{1922}{3985}{8907};
13528     \_fp_fixed_mul_sub_back:wwwn #4; {0000}{2480}{1587}{3015}{8730}{1587};
13529     \_fp_fixed_mul_sub_back:wwwn #4; {0013}{8888}{8888}{8888}{8888}{8889};
13530     \_fp_fixed_mul_sub_back:wwwn #4; {0416}{6666}{6666}{6666}{6666}{6667};
13531     \_fp_fixed_mul_sub_back:wwwn #4; {5000}{0000}{0000}{0000}{0000}{0000};
13532     \_fp_fixed_mul_sub_back:wwwn #4; {10000}{0000}{0000}{0000}{0000}{0000};
13533     { \_fp_fixed_continue:wn 0, }
13534   }
13535   { % 1/17!
13536     \_fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0028}{1145}{7254};
13537     #4; {0000}{0000}{0000}{7647}{1637}{3182};

```

```

13538     \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0000}{0160}{5904}{3836}{8216};
13539     \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0002}{5052}{1083}{8544}{1719};
13540     \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0275}{5731}{9223}{9858}{9065};
13541     \__fp_fixed_mul_sub_back:wwwn #4; {0001}{9841}{2698}{4126}{9841}{2698};
13542     \__fp_fixed_mul_sub_back:wwwn #4; {0083}{3333}{3333}{3333}{3333}{3333};
13543     \__fp_fixed_mul_sub_back:wwwn #4; {1666}{6666}{6666}{6666}{6666}{6667};
13544     \__fp_fixed_mul_sub_back:wwwn #4; {10000}{0000}{0000}{0000}{0000}{0000};
13545     { \__fp_ep_mul:wwwn 0, } #5,#6;
13546 }
13547 {
13548     \exp_after:wN \__fp_sanitizew
13549     \exp_after:wN #2
13550     \int_use:N \__int_eval:w #1
13551 }
13552 #2
13553 }

```

(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.

```

13554 \cs_new:Npn \__fp_tan_series_o:NNwww #1#2#3. #4;
13555 {
13556     \__fp_fixed_mul:wwn #4; #4;
13557     {
13558         \exp_after:wN \__fp_tan_series_aux_o:Nnwww
13559         \__int_value:w
13560         \if_int_odd:w \__int_eval:w #3 / \c_two \__int_eval_end:
13561         \exp_after:wN \reverse_if:N
13562         \fi:
13563         \if_meaning:w #1#2 2 \else: 0 \fi:
13564     }#3}
13565 }

```

```

13566 }
13567 \cs_new:Npn \__fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
13568 {
13569   \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{1527}{3493}{0856}{7059};
13570   #3; {0000}{0159}{6080}{0274}{5257}{6472};
13571   \__fp_fixed_mul_sub_back:wwwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
13572   \__fp_fixed_mul_sub_back:wwwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
13573   \__fp_fixed_mul_sub_back:wwwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
13574   \__fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
13575   { \__fp_ep_mul:wwwn 0, } #4,#5;
13576   {
13577     \__fp_fixed_mul_sub_back:wwwn {0000}{0007}{0258}{0681}{9408}{4706};
13578     #3; {0000}{2343}{7175}{1399}{6151}{7670};
13579     \__fp_fixed_mul_sub_back:wwwn #3; {0019}{2638}{4588}{9232}{8861}{3691};
13580     \__fp_fixed_mul_sub_back:wwwn #3; {0536}{6357}{0691}{4344}{6852}{4252};
13581     \__fp_fixed_mul_sub_back:wwwn #3; {5263}{1578}{9473}{6842}{1052}{6315};
13582     \__fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
13583     {
13584       \reverse_if:N \if_int_odd:w
13585       \__int_eval:w (#2 - \c_one) / \c_two \__int_eval_end:
13586       \exp_after:wN \__fp_reverse_args:Nww
13587       \fi:
13588       \__fp_ep_div:wwwn 0,
13589     }
13590   }
13591   {
13592     \exp_after:wN \__fp_sanitize:Nw
13593     \exp_after:wN #1
13594     \int_use:N \__int_eval:w \__fp_ep_to_float:wwN
13595   }
13596   #1
13597 }

```

(End definition for __fp_tan_series_o:NNwww and __fp_tan_series_aux_o:Nnwww.)

31.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arc-cosecant, and arcsecant) are based on a function often denoted `atan2`. This 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

$$\operatorname{acos} x = \operatorname{atan}(\sqrt{1-x^2}, x) \quad (4)$$

$$\operatorname{asin} x = \operatorname{atan}(x, \sqrt{1-x^2}) \quad (5)$$

$$\operatorname{asec} x = \operatorname{atan}(\sqrt{x^2-1}, 1) \quad (6)$$

$$\operatorname{acsc} x = \operatorname{atan}(1, \sqrt{x^2-1}) \quad (7)$$

$$\operatorname{atan} x = \operatorname{atan}(x, 1) \quad (8)$$

$$\operatorname{acot} x = \operatorname{atan}(1, x). \quad (9)$$

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 $\operatorname{atan}(y, x) = \operatorname{atan}(y/x)$ is the angular coordinate of the point (x, y) .

As for direct trigonometric functions, the first step in computing $\operatorname{atan}(y, x)$ 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.

31.2.1 Arctangent and arccotangent

`__fp_atan_o:Nw` The parsing step manipulates **atan** and **acot** like **min** and **max**, reading in an array of
`__fp_acot_o:Nw` operands, but also leaves `\use_i:nn` or `\use_ii:nn` depending on whether the result
`__fp_atan_dispatch_o:NNnNw` should be given in radians or in degrees. Here, we dispatch according to the number of

arguments. The one-argument versions of arctangent and arccotangent are special cases of the two-argument ones: $\text{atan}(y) = \text{atan}(y, 1) = \text{acot}(1, y)$ and $\text{acot}(x) = \text{atan}(1, x) = \text{acot}(x, 1)$.

```

13598 \cs_new_nopar:Npn \__fp_atan_o:Nw
13599 {
13600   \__fp_atan_dispatch_o:NNnNw
13601   \__fp_acotii_o:Nww \__fp_atanii_o:Nww { atan }
13602 }
13603 \cs_new_nopar:Npn \__fp_acot_o:Nw
13604 {
13605   \__fp_atan_dispatch_o:NNnNw
13606   \__fp_atanii_o:Nww \__fp_acotii_o:Nww { acot }
13607 }
13608 \cs_new:Npn \__fp_atan_dispatch_o:NNnNw #1#2#3#4#5@
13609 {
13610   \if_case:w
13611     \__int_eval:w \__fp_array_count:n {#5} - \c_one \__int_eval_end:
13612     \exp_after:wN #1 \exp_after:wN #4 \c_one_fp #5
13613     \tex_romannumeral:D
13614   \or: #2 #4 #5 \tex_romannumeral:D
13615   \else:
13616     \__msg_kernel_expandable_error:nnnnn
13617     { kernel } { fp-num-args } { #3() } { 1 } { 2 }
13618     \exp_after:wN \c_nan_fp \tex_romannumeral:D
13619   \fi:
13620   \exp_after:wN \c_zero
13621 }

```

(End definition for `__fp_atan_o:Nw` and `__fp_acot_o:Nw`. These functions are documented on page ??.)

`__fp_atanii_o:Nww`
`__fp_acotii_o:Nww`

If either operand is `nan`, we return it. If both are normal, we call `__fp_atan_normal_o:NNnwNnw`. If both are zero or both infinity, we call `__fp_atan_inf_o:NNNw` with argument 2, leading to a result among $\{\pm\pi/4, \pm3\pi/4\}$ (in degrees, $\{\pm45, \pm135\}$). Otherwise, one is much bigger than the other, and we call `__fp_atan_inf_o:NNNw` with either an argument of 4, leading to the values $\pm\pi/2$ (in degrees, ±90), or 0, leading to $\{\pm0, \pm\pi\}$ (in degrees, $\{\pm0, \pm180\}$). Since $\text{acot}(x, y) = \text{atan}(y, x)$, `__fp_acotii_o:ww` simply reverses its two arguments.

```

13622 \cs_new:Npn \__fp_atanii_o:Nww
13623   #1 \s__fp \__fp_chk:w #2#3#4; \s__fp \__fp_chk:w #5
13624 {
13625   \if_meaning:w 3 #2 \__fp_case_return_i_o:ww \fi:
13626   \if_meaning:w 3 #5 \__fp_case_return_ii_o:ww \fi:
13627   \if_case:w
13628     \if_meaning:w #2 #5
13629     \if_meaning:w 1 #2 \c_ten \else: \c_zero \fi:
13630   \else:
13631     \if_int_compare:w #2 > #5 \c_one \else: \c_two \fi:
13632   \fi:
13633   \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 \c_two }

```



```

13634 \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 \c_four }
13635 \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 \c_zero }
13636 \fi:
13637 \__fp_atan_normal_o:NNnwNnw #1
13638 \s__fp \__fp_chk:w #2#3#4;
13639 \s__fp \__fp_chk:w #5
13640 }
13641 \cs_new:Npn \__fp_acotii_o:Nww #1#2; #3;
13642 { \__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:NwwwwwN`, 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.

```

13643 \cs_new:Npn \__fp_atan_inf_o:NNNw #1#2#3 \s__fp \__fp_chk:w #4#5#6;
13644 {
13645   \exp_after:wN \__fp_atan_combine_o:NwwwwwN
13646   \exp_after:wN #2
13647   \int_use:N \__int_eval:w
13648   \if_meaning:w 2 #5 \c_seven - \fi: #3 \exp_after:wN ;
13649   \c__fp_one_fixed_t1 ;
13650   {0000}{0000}{0000}{0000}{0000}{0000};
13651   0,{0000}{0000}{0000}{0000}{0000}{0000}; #1
13652 }
(End definition for \__fp_atan_inf_o:NNNw.)

```

`__fp_atan_normal_o:NNnwNnw` Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a semi-colon. This extended precision is required by other inverse trigonometric functions, to compute things like $\text{atan}(x, \sqrt{1-x^2})$ without intermediate rounding errors.

```

13653 \cs_new_protected:Npn \__fp_atan_normal_o:NNnwNnw
13654   #1 \s__fp \__fp_chk:w 1#2#3#4; \s__fp \__fp_chk:w 1#5#6#7;
13655 {
13656   \__fp_atan_test_o:NwwNwwN
13657   #2 #3, #4{0000}{0000};
13658   #5 #6, #7{0000}{0000}; #1
13659 }
(End definition for \__fp_atan_normal_o:NNnwNnw.)

```

`__fp_atan_test_o:NwwNwwN` This receives: the sign `#1` of y , its exponent `#2`, its 24 digits `#3` in groups of 4, and similarly for x . We prepare to call `__fp_atan_combine_o: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: $\langle octant \rangle \rightarrow 7 - \langle octant \rangle$ 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.

```

13660 \cs_new:Npn \__fp_atan_test_o:NwwNwwN #1#2,#3; #4#5,#6;
13661 {
13662   \exp_after:wN \__fp_atan_combine_o:NwwwwwN
13663   \exp_after:wN #1
13664   \int_use:N \__int_eval:w
13665   \if_meaning:w 2 #4
13666     \c_seven - \__int_eval:w
13667   \fi:
13668   \if_int_compare:w
13669     \__fp_ep_compare:www #2,#3; #5,#6; > \c_zero
13670     \c_three -
13671     \exp_after:wN \__fp_reverse_args:Nww
13672   \fi:
13673   \__fp_atan_div:wnwnw #2,#3; #5,#6;
13674 }

```

(End definition for `__fp_atan_test_o:NwwNwwN`.)

```

\__fp_atan_div:wnwnw
\__fp_atan_near:wwn
\__fp_atan_near_aux:wwn

```

This receives two positive numbers a and b (equal to $|x|$ and $|y|$ in some order), each as an exponent and 6 blocks of 4 digits, such that $0 < a < b$. If $0.41421b < a$, the two numbers are “near”, hence the point (y, x) that we started with is closer to the diagonals $\{|y| = |x|\}$ than to the axes $\{xy = 0\}$. In that case, the octant is 1 (possibly combined with the 7- and 3- inserted earlier) and we wish to compute $\operatorname{atan} \frac{b-a}{a+b}$. 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.

```

13675 \cs_new:Npn \__fp_atan_div:wnwnw #1,#2#3; #4,#5#6;
13676 {
13677   \if_int_compare:w
13678     \__int_eval:w 41421 * #5 < #2 000
13679     \if_case:w \__int_eval:w #4 - #1 \__int_eval_end: 00 \or: 0 \fi:
13680     \exp_stop_f:
13681     \exp_after:wN \__fp_atan_near:wwn
13682   \fi:
13683   \c_zero
13684   \__fp_ep_div:wwwn #1,{#2}#3; #4,{#5}#6;
13685   \__fp_atan_auxi:ww
13686 }
13687 \cs_new:Npn \__fp_atan_near:wwn
13688   \c_zero \__fp_ep_div:wwwn #1,#2; #3,
13689   {
13690     \c_one
13691     \__fp_ep_to_fixed:wwn #1 - #3, #2;

```

```

13692     \__fp_atan_near_aux:wwn
13693   }
13694   \cs_new:Npn \__fp_atan_near_aux:wwn #1; #2;
13695   {
13696     \__fp_fixed_add:wwn #1; #2;
13697     { \__fp_fixed_sub:wwn #2; #1; { \__fp_ep_div:wwwn 0, } 0, }
13698   }

```

(End definition for __fp_atan_div:wwwnw and __fp_atan_near:wwwn. These functions are documented on page ??.)

__fp_atan_auxi:ww 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.

__fp_atan_auxii:w

```

13699   \cs_new:Npn \__fp_atan_auxi:ww #1,#2;
13700   { \__fp_ep_to_fixed:wwn #1,#2; \__fp_atan_auxii:w #1,#2; }
13701   \cs_new:Npn \__fp_atan_auxii:w #1;
13702   {
13703     \__fp_fixed_mul:wwn #1; #1;
13704     {
13705       \__fp_atan_Taylor_loop:www 39 ;
13706       {0000}{0000}{0000}{0000}{0000}{0000} ;
13707     }
13708     ! #1;
13709   }

```

(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.

```

13710   \cs_new:Npn \__fp_atan_Taylor_loop:www #1; #2; #3;
13711   {
13712     \if_int_compare:w #1 = \c_minus_one
13713     \__fp_atan_Taylor_break:w
13714     \fi:
13715     \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl ; #1;
13716     \__fp_rrot:www \__fp_fixed_mul_sub_back:wwwn #2; #3;
13717     {
13718       \exp_after:wN \__fp_atan_Taylor_loop:www
13719       \int_use:N \__int_eval:w #1 - \c_two ;
13720     }
13721     #3;
13722   }
13723   \cs_new:Npn \__fp_atan_Taylor_break:w
13724   \fi: #1 \__fp_fixed_mul_sub_back:wwwn #2; #3 !
13725   { \fi: ; #2 ; }

```

(End definition for __fp_atan_Taylor_loop:www and __fp_atan_Taylor_break:w.)

_fp_atan_combine_o:NwwwwN
_fp_atan_combine_aux:ww

This receives a $\langle sign \rangle$, an $\langle octant \rangle$, a fixed point value of $(\text{atan } z)/z$, a fixed point number z , and another representation of z , as an $\langle exponent \rangle$ and the fixed point number $10^{-\langle exponent \rangle} z$, followed by either `\use_i:nn` (when working in radians) or `\use_ii:nn` (when working in degrees). The function computes the floating point result

$$\langle sign \rangle \left(\left\lceil \frac{\langle octant \rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle octant \rangle} \frac{\text{atan } z}{z} \cdot z \right), \quad (10)$$

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 = $\frac{\text{atan } z}{z}$ with #6, the adjusted z . Otherwise, multiply #3 = $\frac{\text{atan } z}{z}$ with #4 = z , then compute the appropriate multiple of $\frac{\pi}{4}$ and add or subtract the product #3 · #4. In both cases, convert to a floating point with `_fp_fixed_to_float:wN`.

```

13726 \cs_new:Npn \_fp_atan_combine_o:NwwwwN #1 #2; #3; #4; #5,#6; #7
13727 {
13728   \exp_after:wN \_fp_sanitize:Nw
13729   \exp_after:wN #1
13730   \int_use:N \_int_eval:w
13731   \if_meaning:w 0 #2
13732     \exp_after:wN \use_i:nn
13733   \else:
13734     \exp_after:wN \use_ii:nn
13735   \fi:
13736   { #5 \_fp_fixed_mul:wwn #3; #6; }
13737   {
13738     \_fp_fixed_mul:wwn #3; #4;
13739     {
13740       \exp_after:wN \_fp_atan_combine_aux:ww
13741       \int_use:N \_int_eval:w #2 / \c_two ; #2;
13742     }
13743   }
13744   { #7 \_fp_fixed_to_float:wN \_fp_fixed_to_float_rad:wN }
13745   #1
13746 }
13747 \cs_new:Npn \_fp_atan_combine_aux:ww #1; #2;
13748 {
13749   \_fp_fixed_mul_short:wwn
13750   {7853}{9816}{3397}{4483}{0961}{5661};
13751   {#1}{0000}{0000};
13752   {
13753     \if_int_odd:w #2 \exp_stop_f:
13754     \exp_after:wN \_fp_fixed_sub:wwn
13755   \else:
13756     \exp_after:wN \_fp_fixed_add:wwn
13757   \fi:
13758 }
13759 }
```

(End definition for `_fp_atan_combine_o:NwwwwwN` and `_fp_atan_combine_aux:ww`.)

31.2.2 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).

```

13760 \cs_new:Npn \_fp_asin_o:w #1 \s__fp \_fp_chk:w #2#3; @
13761 {
13762   \if_case:w #2 \exp_stop_f:
13763     \_fp_case_return_same_o:w
13764   \or:
13765     \_fp_case_use:nw
13766     { \_fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
13767   \or:
13768     \_fp_case_use:nw
13769     { \_fp_invalid_operation_o:fw { #1 { asin } { asind } } }
13770   \else:
13771     \_fp_case_return_same_o:w
13772   \fi:
13773   \s__fp \_fp_chk:w #2 #3;
13774 }

```

(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.

```

13775 \cs_new:Npn \_fp_acos_o:w #1 \s__fp \_fp_chk:w #2#3; @
13776 {
13777   \if_case:w #2 \exp_stop_f:
13778     \_fp_case_use:nw { \_fp_atan_inf_o:NNNw #1 0 \c_four }
13779   \or:
13780     \_fp_case_use:nw
13781     {
13782       \_fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
13783       \_fp_reverse_args:Nww
13784     }
13785   \or:
13786     \_fp_case_use:nw
13787     { \_fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
13788   \else:
13789     \_fp_case_return_same_o:w
13790   \fi:
13791   \s__fp \_fp_chk:w #2 #3;
13792 }

```

(End definition for `_fp_acos_o:w`.)

`_fp_asin_normal_o:NfwNnnnw` 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` ≥ 1 , `#6#7` ≥ 10000000 , `#8#9` ≥ 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.

```

13793 \cs_new:Npn \_fp_asin_normal_o:NfwNnnnw
13794   #1#2#3 \s_fp \_fp_chk:w 1#4#5#6#7#8#9;
13795   {
13796     \if_int_compare:w #5 < \c_one
13797       \exp_after:wN \_fp_use_none_until_s:w
13798     \fi:
13799     \if_int_compare:w \_int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
13800       \exp_after:wN \_fp_use_none_until_s:w
13801     \fi:
13802     \_fp_use_i:ww
13803     \_fp_invalid_operation_o:fw {#2}
13804     \s_fp \_fp_chk:w 1#4#{#5}{#6}{#7}{#8}{#9};
13805     \_fp_asin_auxi_o:NnNww
13806     #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
13807   }

```

(End definition for `_fp_asin_normal_o:NfwNnnnw`.)

`_fp_asin_auxi_o:NnNww` We compute $x/\sqrt{1-x^2}$. This function is used by `asin` and `acos`, but also by `acsc` and `asec` after inverting the operand, thus it must manipulate extended-precision numbers. First evaluate $1-x^2$ as $(1+x)(1-x)$: this behaves better near $x = 1$. We do the addition/subtraction with fixed point numbers (they are not implemented for extended-precision floats), but go back to extended-precision floats to multiply and compute the inverse square root $1/\sqrt{1-x^2}$. Finally, multiply by the (positive) extended-precision float $|x|$, and feed the (signed) result, and the number $+1$, as arguments to the arctangent function. When computing the arccosine, the arguments $x/\sqrt{1-x^2}$ and $+1$ are swapped by `#2` (`_fp_reverse_args:Nww` in that case) before `_fp_atan_test_o:NwwNwwN` is evaluated. Note that the arctangent function requires normalized arguments, hence the need for `ep_to_ep` and continue after `ep_mul`.

```

13808 \cs_new:Npn \_fp_asin_auxi_o:NnNww #1#2#3#4,#5;
13809   {
13810     \_fp_ep_to_fixed:wwn #4,#5;
13811     \_fp_asin_isqrt:wn
13812     \_fp_ep_mul:wwwn #4,#5;
13813     \_fp_ep_to_ep:wwN
13814     \_fp_fixed_continue:wn
13815     { #2 \_fp_atan_test_o:NwwNwwN #3 }
13816     0 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1
13817   }
13818 \cs_new:Npn \_fp_asin_isqrt:wn #1;
13819   {

```

```

13820 \exp_after:wN \_fp_fixed_sub:wn \c\_fp_one_fixed_t1 ; #1;
13821 {
13822     \_fp_fixed_add_one:wN #1;
13823     \_fp_fixed_continue:wn { \_fp_ep_mul:wwwn 0, } 0,
13824 }
13825 \_fp_ep_isqrt:wn
13826 }

```

(End definition for `_fp_asin_auxi_o:NnNww` and `_fp_asin_isqrt:wn`.)

31.2.3 Arccosecant and arcsecant

`_fp_acsc_o:w` Cases are mostly labelled by #2, except when #2 is 2: then we use #3#2, which is 02 = 2 when the number is $+\infty$ and 22 when the number is $-\infty$. The arccosecant of ± 0 raises an invalid operation exception. The arccosecant of $\pm\infty$ is ± 0 with the same sign. The arcsecant 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.

```

13827 \cs_new:Npn \_fp_acsc_o:w #1 \s\_fp \_fp_chk:w #2#3#4; @
13828 {
13829     \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
13830         \_fp_case_use:nw
13831         { \_fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
13832     \or: \_fp_case_use:nw
13833         { \_fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
13834     \or: \_fp_case_return_o:Nw \c_zero_fp
13835     \or: \_fp_case_return_same_o:w
13836     \else: \_fp_case_return_o:Nw \c_minus_zero_fp
13837     \fi:
13838     \s\_fp \_fp_chk:w #2 #3 #4;
13839 }

```

(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 arccosecant 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`.

```

13840 \cs_new:Npn \_fp_asec_o:w #1 \s\_fp \_fp_chk:w #2#3; @
13841 {
13842     \if_case:w #2 \exp_stop_f:
13843         \_fp_case_use:nw
13844         { \_fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
13845     \or:
13846         \_fp_case_use:nw
13847         {
13848             \_fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
13849             \_fp_reverse_args:Nww
13850         }
13851     \or: \_fp_case_use:nw { \_fp_atan_inf_o:NNNw #1 0 \c_four }

```

```

13852     \else: \__fp_case_return_same_o:w
13853     \fi:
13854     \s__fp \__fp_chk:w #2 #3;
13855 }
(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 $\text{acsc}(x) = \text{asin}(1/x)$ and $\text{asec}(x) = \text{acos}(1/x)$.

```

13856 \cs_new:Npn \__fp_acsc_normal_o:NfwNnw #1#2#3 \s__fp \__fp_chk:w 1#4#5#6;
13857 {
13858     \int_compare:nNnTF {#5} < \c_one
13859     {
13860         \__fp_invalid_operation_o:fw {#2}
13861         \s__fp \__fp_chk:w 1#4{#5}#6;
13862     }
13863     {
13864         \__fp_ep_div:wwwwn
13865         1,{1000}{0000}{0000}{0000}{0000}{0000};
13866         #5,#6{0000}{0000};
13867         { \__fp_asin_auxi_o:nNww #1 {#3} #4 }
13868     }
13869 }
(End definition for \__fp_acsc_normal_o:NfwNnw.)
13870 </initex | package>

```

32 13fp-convert implementation

```

13871 <*initex | package>
13872 <@@=fp>

```

32.1 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.

```

13873 \cs_new:Npn \__fp_trim_zeros:w #1 ;
13874 {
13875     \__fp_trim_zeros_loop:w #1
13876     ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s__stop
13877 }
13878 \cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
13879 \cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
13880 \cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__stop { #1 }
(End definition for \__fp_trim_zeros:w. This function is documented on page ??.)

```


32.2 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`

```

13881 \cs_new:Npn \fp_to_scientific:N #1
13882 { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
13883 \cs_generate_variant:Nn \fp_to_scientific:N { c }
13884 \cs_new_nopar:Npn \fp_to_scientific:n
13885 {
13886   \exp_after:wN \__fp_to_scientific_dispatch:w
13887   \tex_romannumeral:D -'0 \__fp_parse:n
13888 }

```

(End definition for `\fp_to_scientific:N`, `\fp_to_scientific:c`, and `\fp_to_scientific:n`. These functions are documented on page ??.)

`__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”.

```

13889 \group_begin:
13890 \char_set_catcode_other:N E
13891 \tl_to_lowercase:n
13892 {
13893   \group_end:
13894   \cs_new:Npn \__fp_to_scientific_dispatch:w \s_fp \__fp_chk:w #1#2
13895   {
13896     \if_meaning:w 2 #2 \exp_after:wN - \tex_romannumeral:D -'0 \fi:
13897     \if_case:w #1 \exp_stop_f:
13898       \__fp_case_return:nw { 0 }
13899     \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
13900     \or:
13901       \__fp_case_use:nw
13902       {
13903         \__fp_invalid_operation:nnw
13904         {
13905           \exp_after:wN 1
13906           \exp_after:wN E
13907           \int_use:N \c__fp_max_exponent_int
13908         }
13909         { fp_to_scientific }
13910       }
13911     \or:
13912       \__fp_case_use:nw

```

```

13913         {
13914             \__fp_invalid_operation:nnw
13915             { 0 }
13916             { fp_to_scientific }
13917         }
13918     \fi:
13919     \s__fp \__fp_chk:w #1 #2
13920 }
13921 \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
13922 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
13923 {
13924     \if_int_compare:w #2 = \c_one
13925         \exp_after:wN \__fp_to_scientific_normal:wNw
13926     \else:
13927         \exp_after:wN \__fp_to_scientific_normal:wNw
13928         \exp_after:wN E
13929         \int_use:N \__int_eval:w #2 - \c_one
13930     \fi:
13931     ; #3 #4 #5 #6 ;
13932 }
13933 }
13934 \cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
13935 { \__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)

32.3 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

```

13936 \cs_new:Npn \fp_to_decimal:N #1
13937 { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
13938 \cs_generate_variant:Nn \fp_to_decimal:N { c }
13939 \cs_new_nopar:Npn \fp_to_decimal:n
13940 {
13941     \exp_after:wN \__fp_to_decimal_dispatch:w
13942     \tex_romannumeral:D -'0 \__fp_parse:n
13943 }

```

(End definition for \fp_to_decimal:N, \fp_to_decimal:c, and \fp_to_decimal:n. These functions are documented on page ??.)

__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.<zeros><digits>, trimmed.

```

13944 \cs_new:Npn \__fp_to_decimal_dispatch:w \s__fp \__fp_chk:w #1#2
13945 {
13946   \if_meaning:w 2 #2 \exp_after:wN - \tex_romannumeral:D -'0 \fi:
13947   \if_case:w #1 \exp_stop_f:
13948     \__fp_case_return:nw { 0 }
13949   \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
13950   \or:
13951     \__fp_case_use:nw
13952     {
13953       \__fp_invalid_operation:nnw
13954       {
13955         \exp_after:wN \exp_after:wN \exp_after:wN 1
13956         \prg_replicate:nn \c__fp_max_exponent_int 0
13957       }
13958       { fp_to_decimal }
13959     }
13960   \or:
13961     \__fp_case_use:nw
13962     {
13963       \__fp_invalid_operation:nnw
13964       { 0 }
13965       { fp_to_decimal }
13966     }
13967   \fi:
13968   \s__fp \__fp_chk:w #1 #2
13969 }
13970 \cs_new:Npn \__fp_to_decimal_normal:wnnnnn
13971 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
13972 {
13973   \int_compare:nNnTF {#2} > \c_zero
13974   {
13975     \int_compare:nNnTF {#2} < \c_sixteen
13976     {
13977       \__fp_decimate:nNnnnn { \c_sixteen - #2 }
13978       \__fp_to_decimal_large:Nnnw
13979     }
13980     {
13981       \exp_after:wN \exp_after:wN
13982       \exp_after:wN \__fp_to_decimal_huge:wnnnn
13983       \prg_replicate:nn { #2 - \c_sixteen } { 0 } ;
13984     }
13985     {#3} {#4} {#5} {#6}
13986   }
13987   {
13988     \exp_after:wN \__fp_trim_zeros:w
13989     \exp_after:wN 0
13990     \exp_after:wN .
13991     \tex_romannumeral:D -'0 \prg_replicate:nn { - #2 } { 0 }
13992     #3#4#5#6 ;
13993   }

```

```

13994 }
13995 \cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
13996 {
13997   \exp_after:wN \__fp_trim_zeros:w \__int_value:w
13998   \if_int_compare:w #2 > \c_zero
13999     #2
14000   \fi:
14001   \exp_stop_f:
14002   #3.#4 ;
14003 }
14004 \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.)

```

32.4 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

```

14005 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
14006 \cs_generate_variant:Nn \fp_to_tl:N { c }
14007 \cs_new_nopar:Npn \fp_to_tl:n
14008 {
14009   \exp_after:wN \__fp_to_tl_dispatch:w
14010   \tex_romannumeral:D -'0 \__fp_parse:n
14011 }

```

(End definition for `\fp_to_tl:N`, `\fp_to_tl:c`, and `\fp_to_tl:n`. These functions are documented on page ??.)

`__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.

```

14012 \cs_new:Npn \__fp_to_tl_dispatch:w \s__fp \__fp_chk:w #1#2
14013 {
14014   \if_meaning:w 2 #2 \exp_after:wN - \tex_romannumeral:D -'0 \fi:
14015   \if_case:w #1 \exp_stop_f:
14016     \__fp_case_return:nw { 0 }
14017   \or:   \exp_after:wN \__fp_to_tl_normal:nnnnn
14018   \or:   \__fp_case_return:nw { \tl_to_str:n {inf} }
14019   \else: \__fp_case_return:nw { \tl_to_str:n {nan} }
14020   \fi:
14021 }
14022 \cs_new:Npn \__fp_to_tl_normal:nnnnn #1
14023 {
14024   \if_int_compare:w #1 > \c_sixteen
14025     \exp_after:wN \__fp_to_scientific_normal:wnnnnn
14026   \else:
14027     \if_int_compare:w #1 < - \c_two
14028       \exp_after:wN \exp_after:wN
14029       \exp_after:wN \__fp_to_scientific_normal:wnnnnn

```

```

14030     \else:
14031         \exp_after:wN \exp_after:wN
14032         \exp_after:wN \_fp_to_decimal_normal:wnnnnn
14033     \fi:
14034     \fi:
14035     \s__fp \_fp_chk:w 1 0 {#1}
14036 }

```

(End definition for `_fp_to_tl_dispatch:w` and `_fp_to_tl_normal:nnnnn`.)

32.5 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.

32.6 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
14037 \cs_new:Npx \fp_to_dim:N #1
14038 { \exp_not:N \fp_to_decimal:N #1 \tl_to_str:n {pt} }
14039 \cs_generate_variant:Nn \fp_to_dim:N { c }
14040 \cs_new:Npx \fp_to_dim:n #1
14041 { \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 ??.)

`\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
14042 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \_fp_to_int_dispatch:w #1 }
14043 \cs_generate_variant:Nn \fp_to_int:N { c }
14044 \cs_new_nopar:Npn \fp_to_int:n
14045 {
14046     \exp_after:wN \_fp_to_int_dispatch:w
14047     \tex_romannumeral:D -'0 \_fp_parse:n
14048 }

```

(End definition for `\fp_to_int:N`, `\fp_to_int:c`, and `\fp_to_int:n`. These functions are documented on page ??.)

`_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.

```

14049 \cs_new:Npn \_fp_to_int_dispatch:w #1;
14050 {
14051     \exp_after:wN \_fp_to_decimal_dispatch:w \tex_romannumeral:D -'0
14052     \_fp_round:Nwn \_fp_round_to_nearest:NNN #1; { 0 }
14053 }

```

(End definition for `_fp_to_int_dispatch:w`.)

32.7 Convert from a dimension

```

\dim_to_fp:n
  \__fp_from_dim_test:ww
    \__fp_from_dim:wNw
  \__fp_from_dim:wNNnnnnnn
  \__fp_from_dim:wnnnnwNw

```

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`.

```

14054 \cs_new:Npn \dim_to_fp:n #1
14055 {
14056   \exp_after:wN \__fp_from_dim_test:ww
14057   \exp_after:wN 0
14058   \exp_after:wN ,
14059   \__int_value:w \etex_glueexpr:D #1 ;
14060 }
14061 \cs_new:Npn \__fp_from_dim_test:ww #1, #2
14062 {
14063   \if_meaning:w 0 #2
14064     \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
14065   \else:
14066     \exp_after:wN \__fp_from_dim:wNw
14067     \int_use:N \__int_eval:w #1 - \c_four
14068     \if_meaning:w - #2
14069       \exp_after:wN , \exp_after:wN 2 \__int_value:w
14070     \else:
14071       \exp_after:wN , \exp_after:wN 0 \__int_value:w #2
14072     \fi:
14073   \fi:
14074 }
14075 \cs_new:Npn \__fp_from_dim:wNw #1,#2#3;
14076 {
14077   \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
14078   #3 000 0000 00 {10}987654321; #2 {#1}
14079 }
14080 \cs_new:Npn \__fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
14081 { \__fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
14082 \cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
14083 {
14084   \__fp_mul_npos_o:Nww #7
14085   \s__fp \__fp_chk:w 1 #7 {#5} #1 ;
14086   \s__fp \__fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
14087   \prg_do_nothing:
14088 }

```

(End definition for `\dim_to_fp:n`. This function is documented on page 185.)

32.8 Use and eval

\fp_use:N Those public functions are simple copies of the decimal conversions.
\fp_use:c 14089 \cs_new_eq:NN \fp_use:N \fp_to_decimal:N
\fp_eval:n 14090 \cs_generate_variant:Nn \fp_use:N { c }
14091 \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 172.)

\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.
14092 \cs_new:Npn \fp_abs:n #1
14093 { \fp_to_decimal:n { abs __fp_parse:n {#1} } }
(End definition for \fp_abs:n. This function is documented on page 185.)

\fp_max:nn Similar to \fp_abs:n, for consistency with \int_max:nn, etc.
\fp_min:nn 14094 \cs_new:Npn \fp_max:nn #1#2
14095 { \fp_to_decimal:n { max (__fp_parse:n {#1} , __fp_parse:n {#2}) } }
14096 \cs_new:Npn \fp_min:nn #1#2
14097 { \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 185.)

32.9 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
__fp_array_to_clist_loop:Nw 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.

```
14098 \cs_new:Npn \__fp_array_to_clist:n #1
14099 {
14100   \tl_if_empty:nF {#1}
14101   {
14102     \__fp_expand:n
14103     {
14104       { \use_ii:nn }
14105       \__fp_array_to_clist_loop:Nw #1 { ? \__prg_break: } ;
14106       \__prg_break_point:
14107     }
14108   }
14109 }
```

```

14110 \cs_new:Npx \__fp_array_to_clist_loop:Nw #1#2;
14111 {
14112   \exp_not:N \use_none:n #1
14113   \exp_not:N \exp_after:wN
14114   {
14115     \exp_not:N \exp_after:wN ,
14116     \exp_not:N \exp_after:wN \c_space_tl
14117     \exp_not:N \tex_romannumeral:D -‘0
14118     \exp_not:N \__fp_to_tl_dispatch:w #1 #2 ;
14119   }
14120   \exp_not:N \__fp_array_to_clist_loop:Nw
14121 }
(End definition for \__fp_array_to_clist:n. This function is documented on page ??.)
14122 </initex | package>

```

33 l3fp-assign implementation

```

14123 <*initex | package>
14124 <@@=fp>

```

33.1 Assigning values

\fp_new:N Floating point variables are initialized to be +0.

```

14125 \cs_new_protected:Npn \fp_new:N #1
14126 { \cs_new_eq:NN #1 \c_zero_fp }
14127 \cs_generate_variant:Nn \fp_new:N {c}
(End definition for \fp_new:N. This function is documented on page 171.)

```

\fp_set:Nn Simply use __fp_parse:n within various f-expanding assignments.

```

\fp_set:cn 14128 \cs_new_protected:Npn \fp_set:Nn #1#2
\fp_gset:Nn 14129 { \tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_gset:cn 14130 \cs_new_protected:Npn \fp_gset:Nn #1#2
\fp_const:Nn 14131 { \tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_const:cn 14132 \cs_new_protected:Npn \fp_const:Nn #1#2
14133 { \tl_const:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
14134 \cs_generate_variant:Nn \fp_set:Nn {c}
14135 \cs_generate_variant:Nn \fp_gset:Nn {c}
14136 \cs_generate_variant:Nn \fp_const:Nn {c}
(End definition for \fp_set:Nn and others. These functions are documented on page ??.)

```

\fp_set_eq:NN Copying a floating point is the same as copying the underlying token list.

```

\fp_set_eq:cn 14137 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
\fp_set_eq:Nc 14138 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\fp_set_eq:cc 14139 \cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }
\fp_gset_eq:NN 14140 \cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }
\fp_gset_eq:cn (End definition for \fp_set_eq:NN and others. These functions are documented on page ??.)
\fp_gset_eq:Nc
\fp_gset_eq:cc

```


\fp_zero:N Setting a floating point to zero: copy \c_zero_fp.

```

\fp_zero:c 14141 \cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\fp_gzero:N 14142 \cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\fp_gzero:c 14143 \cs_generate_variant:Nn \fp_zero:N { c }
14144 \cs_generate_variant:Nn \fp_gzero:N { c }

```

(End definition for \fp_zero:N and others. These functions are documented on page ??.)

\fp_zero_new:N Set the floating point to zero, or define it if needed.

```

\fp_zero_new:c 14145 \cs_new_protected:Npn \fp_zero_new:N #1
\fp_gzero_new:N 14146 { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\fp_gzero_new:c 14147 \cs_new_protected:Npn \fp_gzero_new:N #1
14148 { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
14149 \cs_generate_variant:Nn \fp_zero_new:N { c }
14150 \cs_generate_variant:Nn \fp_gzero_new:N { c }

```

(End definition for \fp_zero_new:N and others. These functions are documented on page ??.)

33.2 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 is 0)+2, the parsing error would be raised at the last closing parenthesis rather than at the closing parenthesis in the user argument. Thus we evaluate #2 instead of just putting parentheses. As an optimization we use __fp_parse:n rather than \fp_eval:n, which would convert the result away from the internal representation and back.

```

\fp_add:c 14151 \cs_new_protected_nopar:Npn \fp_add:Nn { \__fp_add:NNNn \fp_set:Nn + }
\fp_gadd:Nn 14152 \cs_new_protected_nopar:Npn \fp_gadd:Nn { \__fp_add:NNNn \fp_gset:Nn + }
\fp_gadd:c 14153 \cs_new_protected_nopar:Npn \fp_sub:Nn { \__fp_add:NNNn \fp_set:Nn - }
\fp_sub:Nn 14154 \cs_new_protected_nopar:Npn \fp_gsub:Nn { \__fp_add:NNNn \fp_gset:Nn - }
\fp_gsub:Nn 14155 \cs_new_protected:Npn \__fp_add:NNNn #1#2#3#4
\__fp_add:NNNn 14156 { #1 #3 { #3 #2 \__fp_parse:n {#4} } }
14157 \cs_generate_variant:Nn \fp_add:Nn { c }
14158 \cs_generate_variant:Nn \fp_gadd:Nn { c }
14159 \cs_generate_variant:Nn \fp_sub:Nn { c }
14160 \cs_generate_variant:Nn \fp_gsub:Nn { c }

```

(End definition for \fp_add:Nn and others. These functions are documented on page ??.)

33.3 Showing values

\fp_show:N This shows the result of computing its argument. The __msg_show_variable:n auxiliary expects its input in a slightly odd form, starting with >~, and displays the rest.

```

\fp_show:c 14161 \cs_new_protected:Npn \fp_show:N #1
\fp_show:n 14162 {
14163   \fp_if_exist:NTF #1
14164   { \__msg_show_variable:n { > ~ \fp_to_tl:N #1 } }
14165   {
14166     \__msg_kernel_error:nnx { kernel } { variable-not-defined }
14167     { \token_to_str:N #1 }

```

```

14168     }
14169   }
14170   \cs_new_protected:Npn \fp_show:n #1
14171     { \__msg_show_variable:n { > ~ \fp_to_tl:n {#1} } }
14172   \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 ??.)

33.4 Some useful constants and scratch variables

`\c_one_fp` Some constants.

```

\c_e_fp 14173 \fp_const:Nn \c_e_fp      { 2.718 2818 2845 9045 }
14174 \fp_const:Nn \c_one_fp      { 1 }

```

(End definition for `\c_one_fp` and `\c_e_fp`. These variables are documented on page 176.)

`\c_pi_fp` We simply round π to the closest multiple of 10^{-15} .

```

\c_one_degree_fp 14175 \fp_const:Nn \c_pi_fp      { 3.141 5926 5358 9793 }
14176 \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 176.)

`\l_tmpa_fp` Scratch variables are simply initialized there.

```

\l_tmpb_fp 14177 \fp_new:N \l_tmpa_fp
\g_tmpa_fp 14178 \fp_new:N \l_tmpb_fp
\g_tmpb_fp 14179 \fp_new:N \g_tmpa_fp
14180 \fp_new:N \g_tmpb_fp

```

(End definition for `\l_tmpa_fp` and others. These variables are documented on page 176.)

```

14181 \</initex | package>

```

34 l3fp-old implementation

```

14182 \*initex | package>

```

```

14183 \@@=fp>

```

34.1 Compatibility

`\c_undefined_fp` The old floating point number `\c_undefined_fp` is now implemented as a `nan`.

```

14184 \fp_const:Nn \c_undefined_fp { nan }

```

(End definition for `\c_undefined_fp`. This variable is documented on page ??.)

`\fp_if_undefined_p:N` An old floating point is undefined if it is `inf` or `nan`, *i.e.*, if its type is 2 or 3.

`\fp_if_undefined:NTF`

```

14185 \prg_new_conditional:Npnn \fp_if_undefined:N #1 { p , T , F , TF }
14186 { \exp_after:wN \__fp_if_undefined:w #1 }
14187 \cs_new:Npn \__fp_if_undefined:w \s__fp \__fp_chk:w #1#2;
14188 {
14189   \if_int_compare:w #1 > \c_one
14190     \prg_return_true: \else: \prg_return_false: \fi:
14191 }

```

(End definition for \fp_if_undefined:N. These functions are documented on page ??.)

\fp_if_zero_p:N An old floating point is zero if it is ± 0 , i.e., its type is 0.

```
\fp_if_zero:N $\textcolor{red}{TF}$ 
14192 \prg_new_conditional:Npnn \fp_if_zero:N #1 { p , T , F , TF }
14193 { \exp_after:wN \__fp_if_zero:w #1 }
14194 \cs_new:Npn \__fp_if_zero:w \s__fp \__fp_chk:w #1#2;
14195 { \if_meaning:w #1 0 \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for \fp_if_zero:N. These functions are documented on page ??.)

\fp_abs:N Simply expand the floating point variable to feed it to __fp_abs_o:w or __fp_-_o:w, expanded within an expanding token list assignment. The \prg_do_nothing: is not necessary, but it reminds us more clearly that __fp_abs_o:w and __fp_-_o:w expand after their result.

```
\fp_neg:N
14196 \cs_new_protected_nopar:Npn \fp_abs:N { \__fp_abs:NNN \tl_set:Nx \__fp_abs_o:w }
\fp_neg:c
14197 \cs_new_protected_nopar:Npn \fp_gabs:N { \__fp_abs:NNN \tl_gset:Nx \__fp_abs_o:w }
\fp_gneg:N
14198 \cs_new_protected_nopar:Npx \fp_neg:N
\fp_gneg:c
14199 {
\__fp_abs:NNN
14200 \exp_not:N \__fp_abs:NNN
14201 \exp_not:N \tl_set:Nx
14202 \exp_not:c { \__fp_-_o:w }
14203 }
14204 \cs_new_protected_nopar:Npx \fp_gneg:N
14205 {
14206 \exp_not:N \__fp_abs:NNN
14207 \exp_not:N \tl_gset:Nx
14208 \exp_not:c { \__fp_-_o:w }
14209 }
14210 \cs_new_protected:Npn \__fp_abs:NNN #1#2#3
14211 { #1 #3 { \exp_after:wN #2 #3 \prg_do_nothing: } }
14212 \cs_generate_variant:Nn \fp_abs:N { c }
14213 \cs_generate_variant:Nn \fp_gabs:N { c }
14214 \cs_generate_variant:Nn \fp_neg:N { c }
14215 \cs_generate_variant:Nn \fp_gneg:N { c }
```

(End definition for \fp_abs:N and others. These functions are documented on page ??.)

\fp_mul:Nn See \fp_add:Nn for details.

```
\fp_mul:cn
14216 \cs_new_protected_nopar:Npn \fp_mul:Nn { \__fp_mul:NNNn \fp_set:Nn * }
\fp_gmul:Nn
14217 \cs_new_protected_nopar:Npn \fp_gmul:Nn { \__fp_mul:NNNn \fp_gset:Nn * }
\fp_gmul:cn
14218 \cs_new_protected_nopar:Npn \fp_div:Nn { \__fp_mul:NNNn \fp_set:Nn / }
\fp_div:Nn
14219 \cs_new_protected_nopar:Npn \fp_gdiv:Nn { \__fp_mul:NNNn \fp_gset:Nn / }
\fp_div:cn
14220 \cs_new_protected_nopar:Npn \fp_pow:Nn { \__fp_mul:NNNn \fp_set:Nn ^ }
\fp_gdiv:Nn
14221 \cs_new_protected_nopar:Npn \fp_gpow:Nn { \__fp_mul:NNNn \fp_gset:Nn ^ }
\fp_gdiv:cn
14222 \cs_new_protected:Npn \__fp_mul:NNNn #1#2#3#4
14223 { #1 #3 { #3 #2 \__fp_parse:n {#4} } }
\fp_pow:Nn
14224 \cs_generate_variant:Nn \fp_mul:Nn { c }
\fp_pow:cn
14225 \cs_generate_variant:Nn \fp_gmul:Nn { c }
\fp_gpow:Nn
14226 \cs_generate_variant:Nn \fp_div:Nn { c }
\fp_gpow:cn
14227 \cs_generate_variant:Nn \fp_gdiv:Nn { c }
\__fp_mul:NNNn
14228 \cs_generate_variant:Nn \fp_pow:Nn { c }
14229 \cs_generate_variant:Nn \fp_gpow:Nn { c }
```

(End definition for \fp_mul:Nn and others. These functions are documented on page ??.)

\fp_exp:Nn Here, an added twist is that each value computed by these expensive unary operations is
\fp_exp:cn stored as a constant floating point number.

```

\fp_gexp:Nn 14230 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4#5
\fp_gexp:cn 14231 {
  \fp_ln:Nn 14232 \cs_new_protected_nopar:Npn #1 { #5 {#4} \tl_set_eq:NN #3 }
  \fp_ln:cn 14233 \cs_new_protected_nopar:Npn #2 { #5 {#4} \tl_gset_eq:NN #3 }
  \fp_gln:Nn 14234 \cs_generate_variant:Nn #1 { c }
  \fp_gln:cn 14235 \cs_generate_variant:Nn #2 { c }
  \fp_sin:Nn 14236 }
  \fp_sin:cn 14237 \__fp_tmp:w \fp_exp:Nn \fp_gexp:Nn \__fp_exp_o:w {exp} \__fp_assign_to:nNNNn
  \fp_gsin:Nn 14238 \__fp_tmp:w \fp_ln:Nn \fp_gln:Nn \__fp_ln_o:w {ln} \__fp_assign_to:nNNNn
  \fp_gsin:cn 14239 \__fp_tmp:w \fp_sin:Nn \fp_gsin:Nn \__fp_sin_o:w {sin} \__fp_assign_to:nNNNn
  \fp_cos:Nn 14240 \__fp_tmp:w \fp_cos:Nn \fp_gcos:Nn \__fp_cos_o:w {cos} \__fp_assign_to:nNNNn
  \fp_cos:cn 14241 \__fp_tmp:w \fp_tan:Nn \fp_gtan:Nn \__fp_tan_o:w {tan} \__fp_assign_to:nNNNn
  \fp_gcos:Nn 14242 \cs_new_protected:Npn \__fp_assign_to:nNNNn #1#2#3#4#5
  \fp_gcos:cn 14243 {
    \exp_after:wN \__fp_assign_to_i:wNNNn
    \fp_tan:Nn 14245 \tex_romannumeral:D -'0 \__fp_parse:n {#5} {#1} #2#3#4
    \fp_tan:cn 14246 }
    \fp_gtan:Nn 14247 \cs_new_protected:Npn \__fp_assign_to_i:wNNNn \s__fp \__fp_chk:w #1#2#3; #4
    \fp_gtan:cn 14248 {
      \exp_args:Nc \__fp_assign_to_ii:NnNNN
      \__fp_assign_to:nNNNn 14249 { c__fp_ #4 [ #1 # 2 \if_meaning:w 1 #1 #3 \fi: ] _fp }
      \__fp_assign_to_i:wNNNn 14250 { #1#2#3 }
      \__fp_assign_to_ii:NnNNN 14251 }
      \cs_new_protected:Npn \__fp_assign_to_ii:NnNNN #1#2#3#4#5 14252 {
        \cs_if_exist:NF #1 14253 {
          \tl_const:Nx #1 { #4 \s__fp \__fp_chk:w #2; } } 14254
          #3 #5 #1 14255
        } 14256
      } 14257
    } 14258

```

(End definition for \fp_exp:Nn and others. These functions are documented on page ??.)

\fp_compare:NNNTF Comparisons used to be easier between floating points stored in variables. No more.

```

14259 \cs_new_protected_nopar:Npn \fp_compare:NNNTF { \fp_compare:nNnTF }
14260 \cs_new_protected_nopar:Npn \fp_compare:NNNT { \fp_compare:nNnT }
14261 \cs_new_protected_nopar:Npn \fp_compare:NNNF { \fp_compare:nNnF }

```

(End definition for \fp_compare:NNNTF. This function is documented on page ??.)

\fp_round_places:Nn Rounding to a given number of places is easy, since it is provided by the l3fp-round
\fp_ground_places:Nn module.

```

\__fp_round_places:NNn 14262 \cs_new_protected_nopar:Npn \fp_round_places:Nn
14263 { \__fp_round_places:NNn \tl_set:Nx }
14264 \cs_new_protected_nopar:Npn \fp_ground_places:Nn
14265 { \__fp_round_places:NNn \tl_gset:Nx }
14266 \cs_new_protected:Npn \__fp_round_places:NNn #1#2#3
14267 {

```

```

14268     #1 #2
14269     {
14270         \exp_after:wN \exp_after:wN
14271         \exp_after:wN \__fp_round:Nwn
14272         \exp_after:wN \exp_after:wN
14273         \exp_after:wN \__fp_round_to_nearest:NNN
14274         \exp_after:wN #2
14275         \exp_after:wN { \int_use:N \__int_eval:w #3 }
14276     }
14277 }
14278 \cs_generate_variant:Nn \fp_round_places:Nn { c }
14279 \cs_generate_variant:Nn \fp_ground_places:Nn { c }

```

(End definition for `\fp_round_places:Nn` and `\fp_ground_places:Nn`. These functions are documented on page ??.)

`\fp_round_figures:Nn` Rounding to a given number of figures is the same as rounding to a number of places, after shifting by the exponent of the argument.

`\fp_ground_figures:Nn`

```

14280 \cs_new_protected:Npn \fp_round_figures:Nn #1#2
14281 {
14282     \__fp_round_places:NNn \tl_set:Nx #1
14283     { #2 - \exp_after:wN \__fp_exponent:w #1 }
14284 }
14285 \cs_new_protected:Npn \fp_ground_figures:Nn #1#2
14286 {
14287     \__fp_round_places:NNn \tl_gset:Nx #1
14288     { #2 - \exp_after:wN \__fp_exponent:w #1 }
14289 }
14290 \cs_generate_variant:Nn \fp_round_figures:Nn { c }
14291 \cs_generate_variant:Nn \fp_ground_figures:Nn { c }

```

(End definition for `\fp_round_figures:Nn` and `\fp_ground_figures:Nn`. These functions are documented on page ??.)

```

14292 \</initex | package>

```

35 l3luatex implementation

```

14293 \<*initex | package>

```

Announce and ensure that the required packages are loaded.

```

14294 \<*package>
14295 \ProvidesExplPackage
14296     {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
14297 \__expl_package_check:
14298 \</package>

```

`\lua_now_x:n` When LuaTeX is in use, this is all a question of primitives with new names. On the other hand, for pdfTeX and XeTeX the argument should be removed from the input stream before issuing an error. This is expandable, using `__msg_kernel_expandable_error:nnn` as done for V-type expansion in l3expan.

`\lua_now_x:x`

`\lua_now:n`

`\lua_now:x`

`\lua_shipout_x:n`

`\lua_shipout_x:x`

`\lua_shipout:n`

`\lua_shipout:x`

```

14299 \luatex_if_engine:TF

```

```

14300 {
14301   \cs_new_eq:NN \lua_now_x:n \luatex_directlua:D
14302   \cs_new_eq:NN \lua_shipout_x:n \luatex_latelua:D
14303 }
14304 {
14305   \cs_new:Npn \lua_now_x:n #1
14306   {
14307     \__msg_kernel_expandable_error:nnn
14308     { kernel } { bad-engine } { \lua_now_x:n }
14309   }
14310   \cs_new_protected:Npn \lua_shipout_x:n #1
14311   {
14312     \__msg_kernel_expandable_error:nnn
14313     { kernel } { bad-engine } { \lua_shipout_x:n }
14314   }
14315 }
14316 \cs_generate_variant:Nn \lua_now_x:n { x }
14317 \cs_new:Npn \lua_now:n #1
14318 { \lua_now_x:n { \exp_not:n {#1} } }
14319 \cs_generate_variant:Nn \lua_now:n { x }
14320 \cs_generate_variant:Nn \lua_shipout_x:n { x }
14321 \cs_new_protected:Npn \lua_shipout:n #1
14322 { \lua_shipout_x:n { \exp_not:n {#1} } }
14323 \cs_generate_variant:Nn \lua_shipout:n { x }

```

(End definition for `\lua_now_x:n` and `\lua_now_x:x`. These functions are documented on page ??.)

35.1 Category code tables

14324 `<@@=cctab>`

`\g__cctab_allocate_int` To allocate category code tables, both the read-only and stack tables need to be followed.
`\g__cctab_stack_int` There is also a sequence stack for the dynamic tables themselves.
`\g__cctab_stack_seq`

```

14325 \int_new:N \g__cctab_allocate_int
14326 \int_set:Nn \g__cctab_allocate_int { \c_minus_one }
14327 \int_new:N \g__cctab_stack_int
14328 \seq_new:N \g__cctab_stack_seq

```

(End definition for `\g__cctab_allocate_int`. This function is documented on page ??.)

`\cctab_new:N` Creating a new category code table is done slightly differently from other registers. Low-numbered tables are more efficiently-stored than high-numbered ones. There is also a need to have a stack of flexible tables as well as the set of read-only ones. To satisfy both of these requirements, odd numbered tables are used for read-only tables, and even ones for the stack. Here, therefore, the odd numbers are allocated.

```

14329 \cs_new_protected:Npn \cctab_new:N #1
14330 {
14331   \__chk_if_free_cs:N #1
14332   \int_gadd:Nn \g__cctab_allocate_int { \c_two }
14333   \int_compare:nNnTF
14334     \g__cctab_allocate_int < { \c_max_register_int + \c_one }

```

```

14335     {
14336         \tex_global:D \tex_chardef:D #1 \g__cctab_allocate_int
14337         \luatex_initcatcodetable:D #1
14338     }
14339     { \_msg_kernel_fatal:nxx { kernel } { out-of-registers } { cctab } }
14340 }
14341 \luatex_if_engine:F
14342 {
14343     \cs_set_protected:Npn \cctab_new:N #1
14344     {
14345         \_msg_kernel_error:nxx { kernel } { bad-engine }
14346         { \exp_not:N \cctab_new:N }
14347     }
14348 }
14349 <*package>
14350 \luatex_if_engine:T
14351 {
14352     \cs_set_protected:Npn \cctab_new:N #1
14353     {
14354         \_chk_if_free_cs:N #1
14355         \newcatcodetable #1
14356         \luatex_initcatcodetable:D #1
14357     }
14358 }
14359 </package>

```

(End definition for `\cctab_new:N`. This function is documented on page 189.)

`\cctab_begin:N` The aim here is to ensure that the saved tables are read-only. This is done by using a
`\cctab_end:` stack of tables which are not read only, and actually having them as “in use” copies.
`\l__cctab_internal_tl`

```

14360 \cs_new_protected:Npn \cctab_begin:N #1
14361 {
14362     \seq_gpush:Nx \g__cctab_stack_seq { \tex_the:D \luatex_catcodetable:D }
14363     \luatex_catcodetable:D #1
14364     \int_gadd:Nn \g__cctab_stack_int { \c_two }
14365     \int_compare:nNnT \g__cctab_stack_int > \c_max_register_int
14366     { \_msg_kernel_fatal:nn { kernel } { cctab-stack-full } }
14367     \luatex_savecatcodetable:D \g__cctab_stack_int
14368     \luatex_catcodetable:D \g__cctab_stack_int
14369 }
14370 \cs_new_protected_nopar:Npn \cctab_end:
14371 {
14372     \int_gsub:Nn \g__cctab_stack_int { \c_two }
14373     \seq_if_empty:NTF \g__cctab_stack_seq
14374     { \tl_set:Nn \l__cctab_internal_tl { 0 } }
14375     { \seq_gpop:NN \g__cctab_stack_seq \l__cctab_internal_tl }
14376     \luatex_catcodetable:D \l__cctab_internal_tl \scan_stop:
14377 }
14378 \luatex_if_engine:F
14379 {

```

```

14380 \cs_set_protected:Npn \cctab_begin:N #1
14381 {
14382   \__msg_kernel_error:nxxx { kernel } { bad-engine }
14383   { \exp_not:N \cctab_begin:N } {#1}
14384 }
14385 \cs_set_protected_nopar:Npn \cctab_end:
14386 {
14387   \__msg_kernel_error:nxx { kernel } { bad-engine }
14388   { \exp_not:N \cctab_end: }
14389 }
14390 }
14391 <*package>
14392 \luatex_if_engine:T
14393 {
14394   \cs_set_protected:Npn \cctab_begin:N #1 { \BeginCatcodeRegime #1 }
14395   \cs_set_protected_nopar:Npn \cctab_end: { \EndCatcodeRegime }
14396 }
14397 </package>
14398 \tl_new:N \l__cctab_internal_tl

```

(End definition for \cctab_begin:N. This function is documented on page ??.)

\cctab_gset:Nn Category code tables are always global, so only one version is needed. The set up here is simple, and means that at the point of use there is no need to worry about escaping category codes.

```

14399 \cs_new_protected:Npn \cctab_gset:Nn #1#2
14400 {
14401   \group_begin:
14402   #2
14403   \luatex_savecatcodetable:D #1
14404   \group_end:
14405 }
14406 \luatex_if_engine:F
14407 {
14408   \cs_set_protected:Npn \cctab_gset:Nn #1#2
14409   {
14410     \__msg_kernel_error:nxxx { kernel } { bad-engine }
14411     { \exp_not:N \cctab_gset:Nn } { #1 {#2} }
14412   }
14413 }

```

(End definition for \cctab_gset:Nn. This function is documented on page 189.)

\c_code_cctab Creating category code tables is easy using the function above. The **other** and **string**
\c_document_cctab ones are done by completely ignoring the existing codes as this makes life a lot less
\c_initex_cctab complex. The table for expl3 category codes is always needed, whereas when in package
\c_other_cctab mode the rest can be copied from the existing L^AT_EX 2_ε package `luatex`.
\c_str_cctab

```

14414 \luatex_if_engine:T
14415 {
14416   \cctab_new:N \c_code_cctab
14417   \cctab_gset:Nn \c_code_cctab { }

```



```

14418 }
14419 <*package>
14420 \luatex_if_engine:T
14421 {
14422   \cs_new_eq:NN \c_document_cctab \CatcodeTableLaTeX
14423   \cs_new_eq:NN \c_initex_cctab \CatcodeTableIniTeX
14424   \cs_new_eq:NN \c_other_cctab \CatcodeTableOther
14425   \cs_new_eq:NN \c_str_cctab \CatcodeTableString
14426 }
14427 </package>
14428 <*initex>
14429 \luatex_if_engine:T
14430 {
14431   \cctab_new:N \c_document_cctab
14432   \cctab_new:N \c_other_cctab
14433   \cctab_new:N \c_str_cctab
14434   \cctab_gset:Nn \c_document_cctab
14435   {
14436     \char_set_catcode_space:n { 9 }
14437     \char_set_catcode_space:n { 32 }
14438     \char_set_catcode_other:n { 58 }
14439     \char_set_catcode_math_subscript:n { 95 }
14440     \char_set_catcode_active:n { 126 }
14441   }
14442   \cctab_gset:Nn \c_other_cctab
14443   {
14444     \int_step_inline:nnnn { 0 } { 1 } { 127 }
14445     { \char_set_catcode_other:n {#1} }
14446   }
14447   \cctab_gset:Nn \c_str_cctab
14448   {
14449     \int_step_inline:nnnn { 0 } { 1 } { 127 }
14450     { \char_set_catcode_other:n {#1} }
14451     \char_set_catcode_space:n { 32 }
14452   }
14453 }
14454 </initex>

```

(End definition for `\c_code_cctab`. This function is documented on page 190.)

35.2 Messages

```

14455 \_msg_kernel_new:nnnn { kernel } { bad-engine }
14456 { LuaTeX-engine-not-in-use!~Ignoring~#1. }
14457 {
14458   The~feature~you~are~using~is~only~available~
14459   with~the~LuaTeX~engine.~LaTeX3~ignored~‘#1#2’.
14460 }
14461 \_msg_kernel_new:nnnn { kernel } { cctab-stack-full }
14462 { The~category~code~table~stack~is~exhausted. }

```

```

14463 {
14464     LaTeX~has~been~asked~to~switch~to~a~new~category~code~table,~
14465     but~there~is~no~more~space~to~do~this!
14466 }
14467 </initex | package>

```

36 l3candidates Implementation

```

14468 <*initex | package>

14469 <*package>
14470 \ProvidesExplPackage
14471     {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
14472     \__expl_package_check:
14473 </package>

```

36.1 Additions to l3box

```

14474 <@@=box>

```

36.2 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.

```

14475 \fp_new:N \l__box_angle_fp

```

(End definition for `\l__box_angle_fp`. This variable is documented on page 193.)

\l__box_cos_fp These are used to hold the calculated sine and cosine values while carrying out a rotation.

\l__box_sin_fp

```

14476 \fp_new:N \l__box_cos_fp
14477 \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 193.)

\l__box_top_dim These are the positions of the four edges of a box before manipulation.

\l__box_bottom_dim

\l__box_left_dim

\l__box_right_dim

```

14478 \dim_new:N \l__box_top_dim
14479 \dim_new:N \l__box_bottom_dim
14480 \dim_new:N \l__box_left_dim
14481 \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

\l__box_left_new_dim

\l__box_right_new_dim

```

14482 \dim_new:N \l__box_top_new_dim
14483 \dim_new:N \l__box_bottom_new_dim
14484 \dim_new:N \l__box_left_new_dim
14485 \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.

```

14486 \box_new:N \l__box_internal_box

```

(End definition for `\l__box_internal_box`. This variable is documented on page 193.)

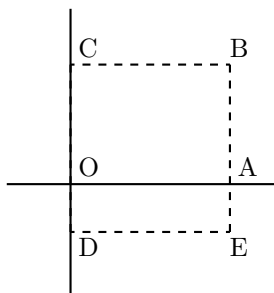


Figure 1: Co-ordinates of a box prior to rotation.

\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:

```

14487 \cs_new_protected:Npn \box_rotate:Nn #1#2
14488 {
14489   \hbox_set:Nn #1
14490   {
14491     \group_begin:
14492       \fp_set:Nn \l__box_angle_fp {#2}
14493       \fp_set:Nn \l__box_sin_fp { sind ( \l__box_angle_fp ) }
14494       \fp_set:Nn \l__box_cos_fp { cosd ( \l__box_angle_fp ) }
14495       \__box_rotate:N #1
14496     \group_end:
14497   }
14498 }

```

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.

```

14499 \cs_new_protected:Npn \__box_rotate:N #1
14500 {
14501   \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
14502   \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
14503   \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
14504   \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.)

```

14505 \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
14506 {
14507   \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
14508   { \__box_rotate_quadrant_one: }
14509   { \__box_rotate_quadrant_two: }
14510 }
14511 {
14512   \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
14513   { \__box_rotate_quadrant_three: }
14514   { \__box_rotate_quadrant_four: }
14515 }

```

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.

```

14516 \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
14517 \hbox_set:Nn \l__box_internal_box
14518 {
14519   \tex_kern:D -\l__box_left_new_dim
14520   \hbox:n
14521   {
14522     \__driver_box_rotate_begin:
14523     \box_use:N \l__box_internal_box
14524     \__driver_box_rotate_end:
14525   }
14526 }

```

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.

```

14527 \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
14528 \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
14529 \box_set_wd:Nn \l__box_internal_box
14530 { \l__box_right_new_dim - \l__box_left_new_dim }
14531 \box_use:N \l__box_internal_box
14532 }

```

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.

```

14533 \cs_new_protected:Npn \__box_rotate_x:nnN #1#2#3
14534 {
14535   \dim_set:Nn #3
14536   {

```

```

14537         \fp_to_dim:n
14538         {
14539             \l__box_cos_fp * \dim_to_fp:n {#1}
14540             - ( \l__box_sin_fp * \dim_to_fp:n {#2} )
14541         }
14542     }
14543 }
14544 \cs_new_protected:Npn \__box_rotate_y:nnN #1#2#3
14545 {
14546     \dim_set:Nn #3
14547     {
14548         \fp_to_dim:n
14549         {
14550             \l__box_sin_fp * \dim_to_fp:n {#1}
14551             + \l__box_cos_fp * \dim_to_fp:n {#2}
14552         }
14553     }
14554 }

```

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.

```

14555 \cs_new_protected:Npn \__box_rotate_quadrant_one:
14556 {
14557     \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
14558     \l__box_top_new_dim
14559     \__box_rotate_y:nnN \l__box_left_dim \l__box_bottom_dim
14560     \l__box_bottom_new_dim
14561     \__box_rotate_x:nnN \l__box_left_dim \l__box_top_dim
14562     \l__box_left_new_dim
14563     \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
14564     \l__box_right_new_dim
14565 }
14566 \cs_new_protected:Npn \__box_rotate_quadrant_two:
14567 {
14568     \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
14569     \l__box_top_new_dim
14570     \__box_rotate_y:nnN \l__box_left_dim \l__box_top_dim
14571     \l__box_bottom_new_dim
14572     \__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
14573     \l__box_left_new_dim
14574     \__box_rotate_x:nnN \l__box_left_dim \l__box_bottom_dim
14575     \l__box_right_new_dim
14576 }
14577 \cs_new_protected:Npn \__box_rotate_quadrant_three:
14578 {
14579     \__box_rotate_y:nnN \l__box_left_dim \l__box_bottom_dim
14580     \l__box_top_new_dim

```

```

14581 \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
14582 \l__box_bottom_new_dim
14583 \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
14584 \l__box_left_new_dim
14585 \__box_rotate_x:nnN \l__box_left_dim \l__box_top_dim
14586 \l__box_right_new_dim
14587 }
14588 \cs_new_protected:Npn \__box_rotate_quadrant_four:
14589 {
14590 \__box_rotate_y:nnN \l__box_left_dim \l__box_top_dim
14591 \l__box_top_new_dim
14592 \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
14593 \l__box_bottom_new_dim
14594 \__box_rotate_x:nnN \l__box_left_dim \l__box_bottom_dim
14595 \l__box_left_new_dim
14596 \__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
14597 \l__box_right_new_dim
14598 }

```

(End definition for `\box_rotate:Nn`. This function is documented on page 192.)

`\l__box_scale_x_fp` Scaling is potentially-different in the two axes.
`\l__box_scale_y_fp`

```

14599 \fp_new:N \l__box_scale_x_fp
14600 \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 193.)

`\box_resize:Nnn` Resizing a box starts by working out the various dimensions of the existing box.
`\box_resize:cnn`
`__box_resize:Nnn`

```

14601 \cs_new_protected:Npn \box_resize:Nnn #1#2#3
14602 {
14603 \hbox_set:Nn #1
14604 {
14605 \group_begin:
14606 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
14607 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
14608 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
14609 \dim_zero:N \l__box_left_dim

```

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.

```

14610 \fp_set:Nn \l__box_scale_x_fp
14611 { \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.

```

14612 \fp_set:Nn \l__box_scale_y_fp
14613 {
14614 \dim_to_fp:n {#3} /
14615 ( \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim } )
14616 }

```

Hand off to the auxiliary which does the work.

```

14617         \__box_resize:Nnn #1 {#2} {#3}
14618     \group_end:
14619 }
14620 }
14621 \cs_generate_variant:Nn \box_resize:Nnn { c }

```

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. This is done using the absolute value of the scale so that the new edge is in the correct place. 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.

```

14622 \cs_new_protected:Npn \__box_resize:Nnn #1#2#3
14623 {
14624     \dim_set:Nn \l__box_right_new_dim { \dim_abs:n {#2} }
14625     \dim_set:Nn \l__box_bottom_new_dim
14626     { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
14627     \dim_set:Nn \l__box_top_new_dim
14628     { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
14629     \__box_resize_common:N #1
14630 }

```

(End definition for `\box_resize:Nnn` and `\box_resize:cnn`. These functions are documented on page ??.)

```

\box_resize_to_ht_plus_dp:Nn
\box_resize_to_ht_plus_dp:cn
\box_resize_to_wd:Nn
\box_resize_to_wd:cn

```

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).

```

14631 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2
14632 {
14633     \hbox_set:Nn #1
14634     {
14635         \group_begin:
14636         \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
14637         \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
14638         \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
14639         \dim_zero:N \l__box_left_dim
14640         \fp_set:Nn \l__box_scale_y_fp
14641         {
14642             \dim_to_fp:n {#2} /
14643             ( \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim } )
14644         }
14645         \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
14646         \__box_resize:Nnn #1 {#2} {#2}
14647     \group_end:
14648 }
14649 }
14650 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
14651 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2

```

```

14652 {
14653   \hbox_set:Nn #1
14654   {
14655     \group_begin:
14656       \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
14657       \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
14658       \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
14659       \dim_zero:N \l__box_left_dim
14660       \fp_set:Nn \l__box_scale_x_fp
14661         { \dim_to_fp:n {#2} / ( \dim_to_fp:n \l__box_right_dim ) }
14662       \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
14663       \__box_resize:Nnn #1 {#2} {#2}
14664     \group_end:
14665   }
14666 }
14667 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }

```

(End definition for `\box_resize_to_ht_plus_dp:Nn` and `\box_resize_to_ht_plus_dp:cn`. These functions are documented on page ??.)

`\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`

```

14668 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
14669 {
14670   \hbox_set:Nn #1
14671   {
14672     \group_begin:
14673       \fp_set:Nn \l__box_scale_x_fp {#2}
14674       \fp_set:Nn \l__box_scale_y_fp {#3}
14675       \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
14676       \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
14677       \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
14678       \dim_zero:N \l__box_left_dim
14679       \dim_set:Nn \l__box_top_new_dim
14680         { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
14681       \dim_set:Nn \l__box_bottom_new_dim
14682         { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
14683       \dim_set:Nn \l__box_right_new_dim
14684         { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
14685       \__box_resize_common:N #1
14686     \group_end:
14687   }
14688 }
14689 \cs_generate_variant:Nn \box_scale:Nnn { c }

```

(End definition for `\box_scale:Nnn` and `\box_scale:cn`. These functions are documented on page ??.)

`__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.

```

14690 \cs_new_protected:Npn \__box_resize_common:N #1
14691 {
14692   \hbox_set:Nn \l__box_internal_box
14693   {
14694     \__driver_box_scale_begin:
14695     \hbox_overlap_right:n { \box_use:N #1 }
14696     \__driver_box_scale_end:
14697   }

```

The new height and depth can be applied directly.

```

14698   \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
14699   \box_set_dp:Nn \l__box_internal_box { \l__box_bottom_new_dim }

```

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.

```

14700   \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
14701   {
14702     \hbox_to_wd:nn { \l__box_right_new_dim }
14703     {
14704       \tex_kern:D \l__box_right_new_dim
14705       \box_use:N \l__box_internal_box
14706       \tex_hss:D
14707     }
14708   }
14709   {
14710     \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
14711     \hbox:n
14712     {
14713       \tex_kern:D \c_zero_dim
14714       \box_use:N \l__box_internal_box
14715       \tex_hss:D
14716     }
14717   }
14718 }

```

(End definition for __box_resize_common:N.)

36.3 Viewing part of a box

`\box_clip:N` A wrapper around the driver-dependent code.

```

\box_clip:c
14719 \cs_new_protected:Npn \box_clip:N #1
14720 { \hbox_set:Nn #1 { \__driver_box_use_clip:N #1 } }
14721 \cs_generate_variant:Nn \box_clip:N { c }

```

(End definition for \box_clip:N and \box_clip:c. These functions are documented on page ??.)

`\box_trim:Nnnnn` Trimming from the left- and right-hand edges of the box is easy: kern the appropriate parts off each side.

```

14722 \cs_new_protected:Npn \box_trim:Nnnnn #1#2#3#4#5
14723 {
14724   \hbox_set:Nn \l__box_internal_box
14725   {
14726     \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
14727     \box_use:N #1
14728     \tex_kern:D -\__dim_eval:w #4 \__dim_eval_end:
14729   }

```

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`.

```

14730   \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
14731   {
14732     \hbox_set:Nn \l__box_internal_box
14733     {
14734       \box_move_down:nn \c_zero_dim
14735       { \box_use:N \l__box_internal_box }
14736     }
14737     \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
14738   }
14739   {
14740     \hbox_set:Nn \l__box_internal_box
14741     {
14742       \box_move_down:nn { #3 - \box_dp:N #1 }
14743       { \box_use:N \l__box_internal_box }
14744     }
14745     \box_set_dp:Nn \l__box_internal_box \c_zero_dim
14746   }

```

Same thing, this time from the top of the box.

```

14747   \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
14748   {
14749     \hbox_set:Nn \l__box_internal_box
14750     { \box_move_up:nn \c_zero_dim { \box_use:N \l__box_internal_box } }
14751     \box_set_ht:Nn \l__box_internal_box
14752     { \box_ht:N \l__box_internal_box - (#5) }
14753   }
14754   {
14755     \hbox_set:Nn \l__box_internal_box
14756     {
14757       \box_move_up:nn { #5 - \box_ht:N \l__box_internal_box }
14758       { \box_use:N \l__box_internal_box }
14759     }
14760     \box_set_ht:Nn \l__box_internal_box \c_zero_dim

```

```

14761     }
14762     \box_set_eq:Nn #1 \l__box_internal_box
14763   }
14764   \cs_generate_variant:Nn \box_trim:Nnnnn { c }

```

(End definition for \box_trim:Nnnnn and \box_trim:cnnnn. These functions are documented on page ??.)

\box_viewport:Nnnnn The same general logic as for the trim operation, but with absolute dimensions. As a
\box_viewport:cnnnn result, there are some things to watch out for in the vertical direction.

```

14765   \cs_new_protected:Npn \box_viewport:Nnnnn #1#2#3#4#5
14766   {
14767     \hbox_set:Nn \l__box_internal_box
14768     {
14769       \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
14770       \box_use:N #1
14771       \tex_kern:D \__dim_eval:w #4 - \box_wd:N #1 \__dim_eval_end:
14772     }
14773     \dim_compare:nNnTF {#3} < \c_zero_dim
14774     {
14775       \hbox_set:Nn \l__box_internal_box
14776       {
14777         \box_move_down:nn \c_zero_dim
14778         { \box_use:N \l__box_internal_box }
14779       }
14780       \box_set_dp:Nn \l__box_internal_box { -\dim_eval:n {#3} }
14781     }
14782     {
14783       \hbox_set:Nn \l__box_internal_box
14784       { \box_move_down:nn {#3} { \box_use:N \l__box_internal_box } }
14785       \box_set_dp:Nn \l__box_internal_box \c_zero_dim
14786     }
14787     \dim_compare:nNnTF {#5} > \c_zero_dim
14788     {
14789       \hbox_set:Nn \l__box_internal_box
14790       { \box_move_up:nn \c_zero_dim { \box_use:N \l__box_internal_box } }
14791       \box_set_ht:Nn \l__box_internal_box
14792       {
14793         #5
14794         \dim_compare:nNnT {#3} > \c_zero_dim
14795         { - (#3) }
14796       }
14797     }
14798     {
14799       \hbox_set:Nn \l__box_internal_box
14800       {
14801         \box_move_up:nn { -\dim_eval:n {#5} }
14802         { \box_use:N \l__box_internal_box }
14803       }
14804       \box_set_ht:Nn \l__box_internal_box \c_zero_dim
14805     }

```

```

14806 \box_set_eq:NN #1 \l__box_internal_box
14807 }
14808 \cs_generate_variant:Nn \box_viewport:Nnnnn { c }

```

(End definition for `\box_viewport:Nnnnn` and `\box_viewport:cnnnn`. These functions are documented on page ??.)

36.4 Additions to `l3clist`

```

14809 <@@=clist>

```

```

\clist_item:Nn
\clist_item:cn
\__clist_item:nnNn
\__clist_item_N_loop:nw

```

To avoid needing to test the end of the list at each step, we first compute the $\langle length \rangle$ of the list. If the item number is 0, less than $-\langle length \rangle$, or more than $\langle length \rangle$, the result is empty. If it is negative, but not less than $-\langle length \rangle$, add $\langle length \rangle + 1$ to the item number before performing the loop. The loop itself is very simple, return the item if the counter reached 1, otherwise, decrease the counter and repeat.

```

14810 \cs_new:Npn \clist_item:Nn #1#2
14811 {
14812   \exp_args:Nfo \__clist_item:nnNn
14813   { \clist_count:N #1 }
14814   #1
14815   \__clist_item_N_loop:nw
14816   {#2}
14817 }
14818 \cs_new:Npn \__clist_item:nnNn #1#2#3#4
14819 {
14820   \int_compare:nNnTF {#4} < \c_zero
14821   {
14822     \int_compare:nNnTF {#4} < { - #1 }
14823     { \use_none_delimit_by_q_stop:w }
14824     { \exp_args:Nf #3 { \int_eval:n { #4 + \c_one + #1 } } }
14825   }
14826   {
14827     \int_compare:nNnTF {#4} > {#1}
14828     { \use_none_delimit_by_q_stop:w }
14829     { #3 {#4} }
14830   }
14831   { } , #2 , \q_stop
14832 }
14833 \cs_new:Npn \__clist_item_N_loop:nw #1 #2,
14834 {
14835   \int_compare:nNnTF {#1} = \c_zero
14836   { \use_i_delimit_by_q_stop:nw { \exp_not:n {#2} } }
14837   { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
14838 }
14839 \cs_generate_variant:Nn \clist_item:Nn { c }

```

(End definition for `\clist_item:Nn` and `\clist_item:cn`. These functions are documented on page ??.)

```

\clist_item:nn
\__clist_item_n:nw
\__clist_item_n_loop:nw
\__clist_item_n_end:n
\__clist_item_n_strip:w

```

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.

```

14840 \cs_new:Npn \clist_item:nn #1#2
14841 {
14842   \exp_args:Nf \__clist_item:nnNn
14843   { \clist_count:n {#1} }
14844   {#1}
14845   \__clist_item_n:nw
14846   {#2}
14847 }
14848 \cs_new:Npn \__clist_item_n:nw #1
14849 { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
14850 \cs_new:Npn \__clist_item_n_loop:nw #1 #2,
14851 {
14852   \exp_args:No \tl_if_blank:nTF {#2}
14853   { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
14854   {
14855     \int_compare:nNnTF {#1} = \c_zero
14856     { \exp_args:No \__clist_item_n_end:n {#2} }
14857     {
14858       \exp_args:Nf \__clist_item_n_loop:nw
14859       { \int_eval:n { #1 - 1 } }
14860       \prg_do_nothing:
14861     }
14862   }
14863 }
14864 \cs_new:Npn \__clist_item_n_end:n #1 #2 \q_stop
14865 {
14866   \__tl_trim_spaces:nn { \q_mark #1 }
14867   { \exp_last_unbraced:No \__clist_item_n_strip:w } ,
14868 }
14869 \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 ??.)

<pre> \clist_set_from_seq:NN \clist_set_from_seq:cN \clist_set_from_seq:Nc \clist_set_from_seq:cc \clist_gset_from_seq:NN \clist_gset_from_seq:cN \clist_gset_from_seq:Nc \clist_gset_from_seq:cc __clist_set_from_seq:NNNN __clist_wrap_item:n __clist_set_from_seq:w </pre>	<p>Setting a comma list from a comma-separated list is done using a simple mapping. We wrap most items with <code>\exp_not:n</code>, 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.</p> <pre> 14870 \cs_new_protected:Npn \clist_set_from_seq:NN 14871 { __clist_set_from_seq:NNNN \clist_clear:N \tl_set:Nx } 14872 \cs_new_protected:Npn \clist_gset_from_seq:NN 14873 { __clist_set_from_seq:NNNN \clist_gclear:N \tl_gset:Nx } 14874 \cs_new_protected:Npn __clist_set_from_seq:NNNN #1#2#3#4 14875 { 14876 \seq_if_empty:NTF #4 14877 { #1 #3 } 14878 { 14879 #2 #3 14880 { 14881 \exp_last_unbraced:Nf \use_none:n </pre>
--	---

```

14882             { \seq_map_function:NN #4 \__clist_wrap_item:n }
14883         }
14884     }
14885 }
14886 \cs_new:Npn \__clist_wrap_item:n #1
14887 {
14888     ,
14889     \tl_if_empty:oTF { \__clist_set_from_seq:w #1 ~ , #1 ~ }
14890     { \exp_not:n {#1} }
14891     { \exp_not:n { {#1} } }
14892 }
14893 \cs_new:Npn \__clist_set_from_seq:w #1 , #2 ~ { }
14894 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , Nc }
14895 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
14896 \cs_generate_variant:Nn \clist_gset_from_seq:NN { c , Nc }
14897 \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 ??.)

`\clist_const:Nn` Creating and initializing a constant comma list is done in a way similar to `\clist_set:Nn`
`\clist_const:cn` and `\clist_gset:Nn`, being careful to strip spaces.

```

\clist_const:Nx 14898 \cs_new_protected:Npn \clist_const:Nn #1#2
\clist_const:cx 14899 { \tl_const:Nx #1 { \__clist_trim_spaces:n {#2} } }
14900 \cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }

```

(End definition for `\clist_const:Nn` and others. These functions are documented on page ??.)

`\clist_if_empty_p:n` As usual, we insert a token (here ?) before grabbing any argument: this avoids losing
`\clist_if_empty:nTF` braces. The argument of `\tl_if_empty:oTF` is empty if #1 is ? followed by blank spaces
`__clist_if_empty_n:w` (besides, this particular variant of the emptiness test is optimized). If the item of the
`__clist_if_empty_n:wNw` comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second
 auxiliary 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.

```

14901 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
14902 {
14903     \__clist_if_empty_n:w ? #1
14904     , \q_mark \prg_return_false:
14905     , \q_mark \prg_return_true:
14906     \q_stop
14907 }
14908 \cs_new:Npn \__clist_if_empty_n:w #1 ,
14909 {
14910     \tl_if_empty:oTF { \use_none:nn #1 ? }
14911     { \__clist_if_empty_n:w ? }
14912     { \__clist_if_empty_n:wNw }
14913 }
14914 \cs_new:Npn \__clist_if_empty_n:wNw #1 \q_mark #2#3 \q_stop {#2}

```

(End definition for `\clist_if_empty:n`. These functions are documented on page 194.)

36.5 Additions to l3coffins

14915 <@@=coffin>

36.6 Rotating coffins

`\l__coffin_sin_fp` Used for rotations to get the sine and cosine values.

`\l__coffin_cos_fp` 14916 `\fp_new:N \l__coffin_sin_fp`

14917 `\fp_new:N \l__coffin_cos_fp`

(End definition for `\l__coffin_sin_fp`. This function 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.

14918 `\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.

14919 `\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
`\l__coffin_right_corner_dim` minimum size of the bounding box after rotation.
`\l__coffin_bottom_corner_dim`

14920 `\dim_new:N \l__coffin_left_corner_dim`

14921 `\dim_new:N \l__coffin_right_corner_dim`

14922 `\dim_new:N \l__coffin_bottom_corner_dim`

14923 `\dim_new:N \l__coffin_top_corner_dim`

(End definition for `\l__coffin_left_corner_dim`. This function is documented on page ??.)

`\coffin_rotate:Nn` Rotating a coffin requires several steps which can be conveniently run together. The sine
`\coffin_rotate:cn` 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.

14924 `\cs_new_protected:Npn \coffin_rotate:Nn #1#2`

14925 {

14926 `\fp_set:Nn \l__coffin_sin_fp { sind (#2) }`

14927 `\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.

14928 `\prop_map_inline:cn { l__coffin_corners_ __int_value:w #1 _prop }`

14929 { `__coffin_rotate_corner:Nnnn #1 {##1} ##2 }`

14930 `\prop_map_inline:cn { l__coffin_poles_ __int_value:w #1 _prop }`

14931 { `__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.

14932 `__coffin_set_bounding:N #1`

14933 `\prop_map_inline:Nn \l__coffin_bounding_prop`

14934 { `__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.

```

14935     \__coffin_find_corner_maxima:N #1
14936     \__coffin_find_bounding_shift:
14937     \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.

```

14938     \hbox_set:Nn \l__coffin_internal_box
14939     {
14940         \tex_kern:D
14941         \__dim_eval:w
14942         \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim
14943         \__dim_eval_end:
14944         \box_move_down:nn { \l__coffin_bottom_corner_dim }
14945         { \box_use:N #1 }
14946     }

```

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.

```

14947     \box_set_ht:Nn \l__coffin_internal_box
14948     { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
14949     \box_set_dp:Nn \l__coffin_internal_box { 0 pt }
14950     \box_set_wd:Nn \l__coffin_internal_box
14951     { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
14952     \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.

```

14953     \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
14954     { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
14955     \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
14956     { \__coffin_shift_pole:Nnnnnn #1 {##1} ##2 }
14957 }
14958 \cs_generate_variant:Nn \coffin_rotate:Nn { c }

```

(End definition for \coffin_rotate:Nn and \coffin_rotate:cn. These functions are documented on page ??.)

__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.

```

14959 \cs_new_protected:Npn \__coffin_set_bounding:N #1
14960 {

```



```

14961 \prop_put:Nnx \l__coffin_bounding_prop { tl }
14962 { { 0 pt } { \dim_use:N \box_ht:N #1 } }
14963 \prop_put:Nnx \l__coffin_bounding_prop { tr }
14964 { { \dim_use:N \box_wd:N #1 } { \dim_use:N \box_ht:N #1 } }
14965 \dim_set:Nn \l__coffin_internal_dim { - \box_dp:N #1 }
14966 \prop_put:Nnx \l__coffin_bounding_prop { bl }
14967 { { 0 pt } { \dim_use:N \l__coffin_internal_dim } }
14968 \prop_put:Nnx \l__coffin_bounding_prop { br }
14969 { { \dim_use:N \box_wd:N #1 } { \dim_use:N \l__coffin_internal_dim } }
14970 }

```

(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
`__coffin_rotate_corner:Nnnn` from the reference point. The same treatment is used for the corners of the material itself
and the bounding box.

```

14971 \cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
14972 {
14973   \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
14974   \prop_put:Nnx \l__coffin_bounding_prop {#1}
14975   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
14976 }
14977 \cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
14978 {
14979   \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
14980   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
14981   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
14982 }

```

(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.

```

14983 \cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
14984 {
14985   \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
14986   \__coffin_rotate_vector:nnNN {#5} {#6}
14987   \l__coffin_x_prime_dim \l__coffin_y_prime_dim
14988   \__coffin_set_pole:Nnx #1 {#2}
14989   {
14990     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
14991     { \dim_use:N \l__coffin_x_prime_dim }
14992     { \dim_use:N \l__coffin_y_prime_dim }
14993   }
14994 }

```

(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.

```

14995 \cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
14996 {
14997   \dim_set:Nn #3
14998   {
14999     \fp_to_dim:n
15000     {
15001       \dim_to_fp:n {#1} * \l__coffin_cos_fp
15002       - ( \dim_to_fp:n {#2} * \l__coffin_sin_fp )
15003     }
15004   }
15005   \dim_set:Nn #4
15006   {
15007     \fp_to_dim:n
15008     {
15009       \dim_to_fp:n {#1} * \l__coffin_sin_fp
15010       + ( \dim_to_fp:n {#2} * \l__coffin_cos_fp )
15011     }
15012   }
15013 }

```

(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.

```

15014 \cs_new_protected:Npn \__coffin_find_corner_maxima:N #1
15015 {
15016   \dim_set:Nn \l__coffin_top_corner_dim { -\c_max_dim }
15017   \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
15018   \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
15019   \dim_set:Nn \l__coffin_left_corner_dim { \c_max_dim }
15020   \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
15021   { \__coffin_find_corner_maxima_aux:nn ##2 }
15022 }
15023 \cs_new_protected:Npn \__coffin_find_corner_maxima_aux:nn #1#2
15024 {
15025   \dim_set:Nn \l__coffin_left_corner_dim
15026   { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
15027   \dim_set:Nn \l__coffin_right_corner_dim
15028   { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
15029   \dim_set:Nn \l__coffin_bottom_corner_dim
15030   { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
15031   \dim_set:Nn \l__coffin_top_corner_dim
15032   { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
15033 }

```

(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.

```

15034 \cs_new_protected_nopar:Npn \__coffin_find_bounding_shift:
15035 {
15036   \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
15037   \prop_map_inline:Nn \l__coffin_bounding_prop
15038     { \__coffin_find_bounding_shift_aux:nn ##2 }
15039 }
15040 \cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
15041 {
15042   \dim_set:Nn \l__coffin_bounding_shift_dim
15043     { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
15044 }

```

(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.

```

15045 \cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
15046 {
15047   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _ prop } {#2}
15048   {
15049     { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
15050     { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
15051   }
15052 }
15053 \cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
15054 {
15055   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _ prop } {#2}
15056   {
15057     { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
15058     { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
15059     {#5} {#6}
15060   }
15061 }

```

(End definition for `__coffin_shift_corner:Nnnn`. This function is documented on page ??.)

36.7 Resizing coffins

`\l__coffin_scale_x_fp`
`\l__coffin_scale_y_fp`

Storage for the scaling factors in x and y , respectively.

```

15062 \fp_new:N \l__coffin_scale_x_fp
15063 \fp_new:N \l__coffin_scale_y_fp

```

(End definition for `\l__coffin_scale_x_fp`. This function is documented on page ??.)

`\l__coffin_scaled_total_height_dim`
`\l__coffin_scaled_width_dim`

When scaling, the values given have to be turned into absolute values.

```

15064 \dim_new:N \l__coffin_scaled_total_height_dim
15065 \dim_new:N \l__coffin_scaled_width_dim

```

(End definition for `\l__coffin_scaled_total_height_dim`. This function 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

```

15066 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
15067 {
15068   \fp_set:Nn \l__coffin_scale_x_fp
15069   { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
15070   \fp_set:Nn \l__coffin_scale_y_fp
15071   {
15072     \dim_to_fp:n {#3} / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
15073   }
15074   \box_resize:Nnn #1 {#2} {#3}
15075   \__coffin_resize_common:Nnn #1 {#2} {#3}
15076 }
15077 \cs_generate_variant:Nn \coffin_resize:Nnn { c }

```

(End definition for \coffin_resize:Nnn and \coffin_resize:cnn. These functions are documented on page ??.)

__coffin_resize_common:Nnn The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```

15078 \cs_new_protected:Npn \__coffin_resize_common:Nnn #1#2#3
15079 {
15080   \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
15081   { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
15082   \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
15083   { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }

```

Negative x -scaling values will place the poles in the wrong location: this is corrected here.

```

15084   \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
15085   {
15086     \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
15087     { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
15088     \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
15089     { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
15090   }
15091 }

```

(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. Only the total height is needed, as this is the shift required for corners and poles. The scaling is done the \TeX way as this works properly with floating point values without needing to use the `fp` module.

\coffin_scale:cnn

```

15092 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
15093 {
15094   \fp_set:Nn \l__coffin_scale_x_fp {#2}
15095   \fp_set:Nn \l__coffin_scale_y_fp {#3}
15096   \box_scale:Nnn #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }

```

```

15097 \dim_set:Nn \l__coffin_internal_dim
15098 { \coffin_ht:N #1 + \coffin_dp:N #1 }
15099 \dim_set:Nn \l__coffin_scaled_total_height_dim
15100 { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
15101 \dim_set:Nn \l__coffin_scaled_width_dim
15102 { -\fp_abs:n { \l__coffin_scale_x_fp } \coffin_wd:N #1 }
15103 \__coffin_resize_common:Nnn #1
15104 { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
15105 }
15106 \cs_generate_variant:Nn \coffin_scale:Nnn { c }

```

(End definition for \coffin_scale:Nnn and \coffin_scale:cnn. These functions are documented on page ??.)

__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.

```

15107 \cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
15108 {
15109   \dim_set:Nn #3
15110   { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
15111   \dim_set:Nn #4
15112   { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
15113 }

```

(End definition for __coffin_scale_vector:nnNN. This function is documented on page ??.)

__coffin_scale_corner:Nnnn Scaling both corners and poles is a simple calculation using the preceding vector scaling.
 __coffin_scale_pole:Nnnnnn

```

15114 \cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
15115 {
15116   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
15117   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
15118   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
15119 }
15120 \cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
15121 {
15122   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
15123   \__coffin_set_pole:Nnx #1 {#2}
15124   {
15125     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
15126     {#5} {#6}
15127   }
15128 }

```

(End definition for __coffin_scale_corner:Nnnn. This function is documented on page ??.)

_coffin_x_shift_corner:Nnnn These functions correct for the x displacement that takes place with a negative horizontal scaling.
 _coffin_x_shift_pole:Nnnnnn

```

15129 \cs_new_protected:Npn \_coffin_x_shift_corner:Nnnn #1#2#3#4
15130 {
15131   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
15132   {

```

```

15133         { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
15134     }
15135 }
15136 \cs_new_protected:Npn \__coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
15137 {
15138     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } {#2}
15139     {
15140         { \dim_eval:n #3 + \box_wd:N #1 } {#4}
15141         {#5} {#6}
15142     }
15143 }

```

(End definition for __coffin_x_shift_corner:Nnnn. This function is documented on page ??.)

36.8 Additions to l3file

```

15144 <@@=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

```

15145 \cs_new_nopar:Npn \ior_map_break:
15146 { \__prg_map_break:Nn \ior_map_break: { } }
15147 \cs_new_nopar:Npn \ior_map_break:n
15148 { \__prg_map_break:Nn \ior_map_break: }

```

(End definition for \ior_map_break: and \ior_map_break:n. These functions are documented on page 196.)

\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

```

15149 \cs_new_protected_nopar:Npn \ior_map_inline:Nn
15150 { \__ior_map_inline:NNn \ior_get:NN }
15151 \cs_new_protected_nopar:Npn \ior_str_map_inline:Nn
15152 { \__ior_map_inline:NNn \ior_get_str:NN }
15153 \cs_new_protected_nopar:Npn \__ior_map_inline:NNn
15154 {
15155     \int_gincr:N \g__prg_map_int
15156     \exp_args:Nc \__ior_map_inline:NNNn
15157     { __prg_map_ \int_use:N \g__prg_map_int :n }
15158 }
15159 \cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
15160 {
15161     \cs_set:Npn #1 ##1 {#4}
15162     \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
15163     \__prg_break_point:Nn \ior_map_break:
15164     { \int_gdecr:N \g__prg_map_int }
15165 }
15166 \cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
15167 {

```

```

15168     #2 #3 \l__ior_internal_tl
15169     \ior_if_eof:NF #3
15170     {
15171         \exp_args:No #1 \l__ior_internal_tl
15172         \__ior_map_inline_loop:NNN #1#2#3
15173     }
15174 }
15175 \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 ??.)

36.9 Additions to l3fp

15176 <@@=fp>

`\fp_set_from_dim:Nn` Use the appropriate function from l3fp-convert.

```

\fp_set_from_dim:cn 15177 \cs_new_protected:Npn \fp_set_from_dim:Nn #1#2
\fp_gset_from_dim:Nn 15178 { \tl_set:Nx #1 { \dim_to_fp:n {#2} } }
\fp_gset_from_dim:cn 15179 \cs_new_protected:Npn \fp_gset_from_dim:Nn #1#2
15180 { \tl_gset:Nx #1 { \dim_to_fp:n {#2} } }
15181 \cs_generate_variant:Nn \fp_set_from_dim:Nn { c }
15182 \cs_generate_variant:Nn \fp_gset_from_dim:Nn { c }

```

(End definition for `\fp_set_from_dim:Nn` and others. These functions are documented on page ??.)

36.10 Additions to l3prop

15183 <@@=prop>

`\prop_map_tokens:Nn` 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 empty, with two exceptions: it is `\s__prop` the first time and `\s_obj_end` the last.

```

15184 \cs_new:Npn \prop_map_tokens:Nn #1#2
15185 {
15186     \exp_last_unbraced:Nno \__prop_map_tokens:nwwn {#2} #1
15187     \__prop_pair:wn \q_recursion_tail \s__prop { }
15188     \__prg_break_point:Nn \prop_map_break: { }
15189 }
15190 \cs_new:Npn \__prop_map_tokens:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
15191 {
15192     \if_meaning:w \q_recursion_tail #3
15193     \exp_after:wN \prop_map_break:
15194     \fi:
15195     \use:n {#1} {#3} {#4}
15196     \__prop_map_tokens:nwwn {#1}
15197 }
15198 \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 ??.)

`\prop_get: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_get_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.

```

15199 \cs_new:Npn \prop_get:Nn #1#2
15200 {
15201   \exp_last_unbraced:Noo \__prop_get_Nn:nwn { \tl_to_str:n {#2} } #1
15202   \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
15203   \__prg_break_point:
15204 }
15205 \cs_new:Npn \__prop_get_Nn:nwn #1#2 \__prop_pair:wn #3 \s__prop #4
15206 {
15207   \str_if_eq_x:nnTF {#1} {#3}
15208   { \__prg_break:n { \exp_not:n {#4} } }
15209   { \__prop_get_Nn:nwn {#1} }
15210 }
15211 \cs_generate_variant:Nn \prop_get:Nn { c }

```

(End definition for `\prop_get:Nn` and `\prop_get:cn`. These functions are documented on page ??.)

36.11 Additions to l3seq

15212 $\langle @@=seq \rangle$

`\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.

```

15213 \cs_new:Npn \seq_item:Nn #1 { \exp_after:wN \__seq_item:wNn #1 #1 }
15214 \cs_new:Npn \__seq_item:wNn \s__seq #1 \s_obj_end #2#3
15215 {
15216   \exp_args:Nf \__seq_item:nnn
15217   {
15218     \int_eval:n
15219     {
15220       \int_compare:nNnT {#3} < \c_zero
15221       { \seq_count:N #2 + \c_one + }
15222       #3
15223     }
15224   }
15225   #1
15226   { ? \__prg_break: } { }
15227   \__prg_break_point:
15228 }

```



```

15229 \cs_new:Npn \__seq_item:nnn #1#2#3
15230 {
15231   \use_none:n #2
15232   \int_compare:nNnTF {#1} = \c_one
15233     { \__prg_break:n { \exp_not:n {#3} } }
15234     { \exp_args:Nf \__seq_item:nnn { \int_eval:n { #1 - 1 } } }
15235 }
15236 \cs_generate_variant:Nn \seq_item:Nn { c }

```

(End definition for \seq_item:Nn and \seq_item:cn. These functions are documented on page ??.)

\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 here is to first expand both of the 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.

```

15237 \cs_new:Npn \seq_mapthread_function:NNN #1#2#3
15238 { \exp_after:wN \__seq_mapthread_function:wNN #2 #1 #3 }
15239 \cs_new:Npn \__seq_mapthread_function:wNN \s__seq #1 \s_obj_end #2#3
15240 {
15241   \exp_after:wN \__seq_mapthread_function:wNw #2 #3
15242   #1 { ? \__prg_break: } { }
15243   \__prg_break_point:
15244 }
15245 \cs_new:Npn \__seq_mapthread_function:wNw \s__seq #1 \s_obj_end #2
15246 {
15247   \__seq_mapthread_function:Nnnwnn #2
15248   #1 { ? \__prg_break: } { }
15249   \q_stop
15250 }
15251 \cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \q_stop #5#6
15252 {
15253   \use_none:n #2
15254   \use_none:n #5
15255   #1 {#3} {#6}
15256   \__seq_mapthread_function:Nnnwnn #1 #4 \q_stop
15257 }
15258 \cs_generate_variant:Nn \seq_mapthread_function:NNN { Nc }
15259 \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 ??.)

\seq_set_from_clist:NN
 \seq_set_from_clist:cN
 \seq_set_from_clist:Nc
 \seq_set_from_clist:cc
 \seq_set_from_clist:Nn
 \seq_set_from_clist:cn
 \seq_gset_from_clist:NN
 \seq_gset_from_clist:cN
 \seq_gset_from_clist:Nc
 \seq_gset_from_clist:cc
 \seq_gset_from_clist:Nn
 \seq_gset_from_clist:cn

Setting a sequence from a comma-separated list is done using a simple mapping.

```

15260 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
15261 {
15262   \tl_set:Nx #1
15263   { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n \s_obj_end }
15264 }

```

```

15265 \cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
15266 {
15267   \tl_set:Nx #1
15268   { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n \s_obj_end }
15269 }
15270 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
15271 {
15272   \tl_gset:Nx #1
15273   { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n \s_obj_end }
15274 }
15275 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
15276 {
15277   \tl_gset:Nx #1
15278   { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n \s_obj_end }
15279 }
15280 \cs_generate_variant:Nn \seq_set_from_clist:NN { Nc }
15281 \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
15282 \cs_generate_variant:Nn \seq_set_from_clist:Nn { c }
15283 \cs_generate_variant:Nn \seq_gset_from_clist:NN { Nc }
15284 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
15285 \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 ??.)

\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_tmp:w
\__seq_reverse:NN
\__seq_reverse_setup:w
\__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, T_EX'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, T_EX 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. T_EX 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.

```

15286 \cs_new_protected_nopar:Npn \__seq_tmp:w { }
15287 \cs_new_protected_nopar:Npn \seq_reverse:N
15288 { \__seq_reverse:NN \tl_set:Nx }
15289 \cs_new_protected_nopar:Npn \seq_greverse:N
15290 { \__seq_reverse:NN \tl_gset:Nx }
15291 \cs_new_protected:Npn \__seq_reverse:NN #1 #2
15292 {
15293   \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
15294   \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
15295   #1 #2 { \exp_after:wN \__seq_reverse_setup:w #2 }
15296   \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
15297 }
15298 \cs_new:Npn \__seq_reverse_setup:w #1 \s_obj_end
15299 { #1 \exp_not:n { } \s_obj_end }
15300 \cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
15301 {
15302   #2
15303   \exp_not:n { \__seq_item:n {#1} #3 }
15304 }
15305 \cs_generate_variant:Nn \seq_reverse:N { c }
15306 \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End definition for `\seq_reverse:N` and others. These functions are documented on page ??.)

`\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.

```

15307 \cs_new_protected_nopar:Npn \seq_set_filter:NNn
15308 { \__seq_set_filter:NNNn \tl_set:Nx }
15309 \cs_new_protected_nopar:Npn \seq_gset_filter:NNn
15310 { \__seq_set_filter:NNNn \tl_gset:Nx }
15311 \cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
15312 {
15313   \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
15314   #1 #2 { #3 }
15315   \__seq_pop_item_def:
15316 }

```

(End definition for `\seq_set_filter:NNn` and `\seq_gset_filter:NNn`. These functions are documented on page 198.)

`\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.

```

15317 \cs_new_protected_nopar:Npn \seq_set_map:NNn
15318 { \__seq_set_map:NNNn \tl_set:Nx }
15319 \cs_new_protected_nopar:Npn \seq_gset_map:NNn
15320 { \__seq_set_map:NNNn \tl_gset:Nx }
15321 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
15322 {
15323   \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }

```

```

15324     #1 #2 { #3 }
15325     \__seq_pop_item_def:
15326 }

```

(End definition for `\seq_set_map:NNn` and `\seq_gset_map:NNn`. These functions are documented on page 198.)

36.12 Additions to l3skip

```

15327 <@@=dim>

```

`\dim_to_pt:n` A copy of the internal function `__dim_strip_pt:n`, which should perhaps be eliminated in favor of `\dim_to_pt:n`.

```

15328 \cs_new_eq:NN \dim_to_pt:n \__dim_strip_pt:n

```

(End definition for `\dim_to_pt:n`. This function is documented on page 198.)

`\dim_to_unit:nn` An analog of `\dim_ratio:nn` that produces a decimal number as its result, rather than a rational fraction for use within dimension expressions. The naive implementation as

```

\cs_new:Npn \dim_to_unit:nn #1#2
{ \dim_to_pt:n { 1pt * \dim_ratio:nn {#1} {#2} } }

```

would not ignore trailing tokens (see documentation), so we need a bit more work.

```

15329 \cs_new:Npn \dim_to_unit:nn #1#2
15330 {
15331   \dim_to_pt:n
15332   {
15333     1pt * \__dim_to_unit:n { \dim_to_pt:n {#1} pt }
15334     / \__dim_to_unit:n { \dim_to_pt:n {#2} pt }
15335   }
15336 }
15337 \cs_new:Npn \__dim_to_unit:n #1
15338 { \__int_value:w \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_to_unit:nn`. This function is documented on page 198.)

```

15339 <@@=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.

```

15340 \cs_new:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
15341 {
15342   \skip_if_finite:nTF {#1}
15343   {
15344     #3 = \etex_gluestretch:D #1 \scan_stop:
15345     #4 = \etex_glueshrink:D #1 \scan_stop:
15346   }
15347   {
15348     #3 = \c_zero_skip
15349     #4 = \c_zero_skip

```

```

15350         #2
15351     }
15352 }

```

(End definition for `\skip_split_finite_else_action:nnNN`. This function is documented on page 199.)

36.13 Additions to `l3tl`

```

15353 <@@=tl>

```

`\tl_if_single_token_p:n`
`\tl_if_single_token:nTF`

There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. Otherwise, compare with a single space, only case where we have a single token.

```

15354 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
15355 {
15356     \tl_if_head_is_N_type:nTF {#1}
15357     { \__str_if_eq_x_return:nn { \exp_not:o { \use_none:n #1 } } { } }
15358     { \__str_if_eq_x_return:nn { \exp_not:n {#1} } { ~ } }
15359 }

```

(End definition for `\tl_if_single_token:n`. These functions are documented on page 199.)

`\tl_reverse_tokens:n`
`__tl_reverse_group:nn`

The same as `\tl_reverse:n` but with recursion within brace groups.

```

15360 \cs_new:Npn \tl_reverse_tokens:n #1
15361 {
15362     \etex_unexpanded:D \exp_after:wN
15363     {
15364         \tex_romannumeral:D
15365         \__tl_act:NNNnn
15366         \__tl_reverse_normal:nN
15367         \__tl_reverse_group:nn
15368         \__tl_reverse_space:n
15369         { }
15370         {#1}
15371     }
15372 }
15373 \cs_new:Npn \__tl_reverse_group:nn #1
15374 {
15375     \__tl_act_group_recurse:Nnn
15376     \__tl_act_reverse_output:n
15377     { \tl_reverse_tokens:n }
15378 }

```

`__tl_act_group_recurse:Nnn`

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.

```

15379 \cs_new:Npn \__tl_act_group_recurse:Nnn #1#2#3
15380 {
15381     \exp_args:Nf #1
15382     { \exp_after:wN \exp_after:wN \exp_after:wN { #2 {#3} } }
15383 }

```

(End definition for `\tl_reverse_tokens:n`. This function is documented on page 199.)

`\tl_count_tokens:n` The token count is computed through an `\int_eval:n` construction. Each `1+` is output to the *left*, into the integer expression, and the sum is ended by the `\c_zero` inserted by `__tl_act_end:wn`. Somewhat a hack.

```

15384 \cs_new:Npn \tl_count_tokens:n #1
15385 {
15386   \int_eval:n
15387   {
15388     \__tl_act:NNNnn
15389     \__tl_act_count_normal:nN
15390     \__tl_act_count_group:nn
15391     \__tl_act_count_space:n
15392     { }
15393     {#1}
15394   }
15395 }
15396 \cs_new:Npn \__tl_act_count_normal:nN #1 #2 { 1 + }
15397 \cs_new:Npn \__tl_act_count_space:n #1 { 1 + }
15398 \cs_new:Npn \__tl_act_count_group:nn #1 #2
15399 { 2 + \tl_count_tokens:n {#2} + }

```

(End definition for `\tl_count_tokens:n`. This function is documented on page 199.)

`\c__tl_act_uppercase_tl` These constants contain the correspondence between lowercase and uppercase letters, in the form `aAbBcC...` and `AaBbCc...` respectively.

```

15400 \tl_const:Nn \c__tl_act_uppercase_tl
15401 {
15402   aA bB cC dD eE fF gG hH iI jJ kK lL mM
15403   nN oO pP qQ rR sS tT uU vV wW xX yY zZ
15404 }
15405 \tl_const:Nn \c__tl_act_lowercase_tl
15406 {
15407   Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm
15408   Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz
15409 }

```

(End definition for `\c__tl_act_uppercase_tl` and `\c__tl_act_lowercase_tl`. These variables are documented on page ??.)

`\tl_expandable_uppercase:n` The only difference between uppercasing and lowercasing is the table of correspondence that is used. As for other token list actions, we feed `__tl_act:NNNnn` three functions, and this time, we use the *parameters* argument to carry which case-changing we are applying. A space is simply output. A normal token is compared to each letter in the alphabet using `\str_if_eq:nn` tests, and converted if necessary to upper/lowercase, before being output. For a group, we must perform the conversion within the group (the `\exp_after:wn` trigger `\romannumeral`, which expands fully to give the converted group), then output.

```

15410 \cs_new:Npn \tl_expandable_uppercase:n #1
15411 {

```

```

15412 \etex_unexpanded:D \exp_after:wN
15413 {
15414   \tex_romannumeral:D
15415   \__tl_act_case_aux:nn { \c__tl_act_uppercase_tl } {#1}
15416 }
15417 }
15418 \cs_new:Npn \tl_expandable_lowercase:n #1
15419 {
15420   \etex_unexpanded:D \exp_after:wN
15421   {
15422     \tex_romannumeral:D
15423     \__tl_act_case_aux:nn { \c__tl_act_lowercase_tl } {#1}
15424   }
15425 }
15426 \cs_new:Npn \__tl_act_case_aux:nn
15427 {
15428   \__tl_act:NNNnn
15429   \__tl_act_case_normal:nN
15430   \__tl_act_case_group:nn
15431   \__tl_act_case_space:n
15432 }
15433 \cs_new:Npn \__tl_act_case_space:n #1 { \__tl_act_output:n {~} }
15434 \cs_new:Npn \__tl_act_case_normal:nN #1 #2
15435 {
15436   \exp_args:Nf \__tl_act_output:n
15437   {
15438     \exp_args:NNo \str_case:nnF #2 {#1}
15439     { \exp_stop_f: #2 }
15440   }
15441 }
15442 \cs_new:Npn \__tl_act_case_group:nn #1 #2
15443 {
15444   \exp_after:wN \__tl_act_output:n \exp_after:wN
15445   { \exp_after:wN { \tex_romannumeral:D \__tl_act_case_aux:nn {#1} {#2} } }
15446 }

```

(End definition for `\tl_expandable_uppercase:n` and `\tl_expandable_lowercase:n`. These functions are documented on page 199.)

`\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`

```

15447 \cs_new:Npn \tl_item:nn #1#2
15448 {
15449   \exp_args:Nf \__tl_item:nn
15450   {
15451     \int_eval:n
15452     {
15453       \int_compare:nNnT {#2} < \c_zero
15454       { \tl_count:n {#1} + \c_one + }

```

```

15455         #2
15456     }
15457 }
15458 #1
15459 \q_recursion_tail
15460 \__prg_break_point:
15461 }
15462 \cs_new:Npn \__tl_item:nn #1#2
15463 {
15464     \__quark_if_recursion_tail_break:nN {#2} \__prg_break:
15465     \int_compare:nNnTF {#1} = \c_one
15466     { \__prg_break:n { \exp_not:n {#2} } }
15467     { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
15468 }
15469 \cs_new_nopar:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
15470 \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 ??.)

36.14 Additions to l3tokens

```

15471 <@@=char>

```

`\char_set_active:Npn`
`\char_set_active:Npx`
`\char_gset_active:Npn`
`\char_gset_active:Npx`
`\char_set_active_eq:NN`
`\char_gset_active_eq:NN`

```

15472 \group_begin:
15473 \char_set_catcode_active:N \^^@
15474 \cs_set:Npn \char_tmp:NN #1#2
15475 {
15476     \cs_new:Npn #1 ##1
15477     {
15478         \char_set_catcode_active:n { '##1 }
15479         \group_begin:
15480         \char_set_lccode:nn { '\^^@ } { '##1 }
15481         \tl_to_lowercase:n { \group_end: #2 \^^@ }
15482     }
15483 }
15484 \char_tmp:NN \char_set_active:Npn \cs_set:Npn
15485 \char_tmp:NN \char_set_active:Npx \cs_set:Npx
15486 \char_tmp:NN \char_gset_active:Npn \cs_gset:Npn
15487 \char_tmp:NN \char_gset_active:Npx \cs_gset:Npx
15488 \char_tmp:NN \char_set_active_eq:NN \cs_set_eq:NN
15489 \char_tmp:NN \char_gset_active_eq:NN \cs_gset_eq:NN
15490 \group_end:

```

(End definition for `\char_set_active:Npn` and `\char_set_active:Npx`. These functions are documented on page 200.)

```

15491 <@@=peek>

```

`\peek_N_type:TF` All tokens are N-type tokens, except in four cases: begin-group tokens, end-group tokens, space tokens with character code 32, and outer tokens. Since `\l_peek_token` might be

`__peek_execute_branches_N_type:`
`__peek_N_type:w`
`__peek_N_type_aux:nnw`

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.

```

15492 \group_begin:
15493   \char_set_catcode_other:N \O
15494   \char_set_catcode_other:N \U
15495   \char_set_catcode_other:N \T
15496   \char_set_catcode_other:N \E
15497   \char_set_catcode_other:N \R
15498   \tl_to_lowercase:n
15499   {
15500     \cs_new_protected_nopar:Npn \__peek_execute_branches_N_type:
15501     {
15502       \if_int_odd:w
15503         \if_catcode:w \exp_not:N \l_peek_token { \c_two \fi:
15504         \if_catcode:w \exp_not:N \l_peek_token } \c_two \fi:
15505         \if_meaning:w \l_peek_token \c_space_token \c_two \fi:
15506         \c_one
15507         \exp_after:wN \__peek_N_type:w
15508         \token_to_meaning:N \l_peek_token
15509         \q_mark \__peek_N_type_aux:nnw
15510         OUTER \q_mark \use_none_delimit_by_q_stop:w
15511         \q_stop
15512         \exp_after:wN \__peek_true:w
15513       \else:
15514         \exp_after:wN \__peek_false:w
15515       \fi:
15516     }
15517     \cs_new_protected:Npn \__peek_N_type:w #1 OUTER #2 \q_mark #3
15518     { #3 {#1} {#2} }
15519   }
15520 \group_end:
15521 \cs_new_protected:Npn \__peek_N_type_aux:nnw #1 #2 #3 \fi:
15522 {
15523   \fi:
15524   \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
15525   { \__peek_true:w }
15526   { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }

```

```

15527     }
15528     \cs_new_protected_nopar:Npn \peek_N_type:TF
15529     { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }
15530     \cs_new_protected_nopar:Npn \peek_N_type:T
15531     { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
15532     \cs_new_protected_nopar:Npn \peek_N_type:F
15533     { \__peek_token_generic:NNF \__peek_execute_branches_N_type: \scan_stop: }
(End definition for \peek_N_type:. This function is documented on page 201.)
15534 \</initex | package>

```

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
\!	7433
\"	2595, 2610
\#	6, 2610, 9216
\\$	2596, 2610
\%	2610, 9218
.	180
\	2597, 2610, 7433, 7434, 10881, 10955
\)	10837
**	181
*	2575, 2577, 2581, 2588, 9183, 9184, 10787
\+	10885
\,	8058, 8060, 10853
\-	303, 10012, 10884
\.bool_gset:N	8380
\.bool_gset:c	8380
\.bool_gset_inverse:N	8388
\.bool_gset_inverse:c	8388
\.bool_set:N	8380
\.bool_set:c	8380
\.bool_set_inverse:N	8388
\.bool_set_inverse:c	8388
\.choice:	8396
\.choice_code:n	8763
\.choice_code:x	8763
\.choices:Vn	8398
\.choices:nn	8398
\.choices:on	8398
\.choices:xn	8398
\.clist_gset:N	8408
\.clist_gset:c	8408
\.clist_set:N	8408
\.clist_set:c	8408
\.code:n	8406
\.default:V	8416
\.default:n	8416
\.default:o	8416
\.default:x	8416
\.dim_gset:N	8424
\.dim_gset:c	8424
\.dim_set:N	8424
\.dim_set:c	8424
\.fp_gset:N	8432
\.fp_gset:c	8432
\.fp_set:N	8432
\.fp_set:c	8432
\.generate_choices:n	8763
\.groups:n	8440
\.initial:V	8442
\.initial:n	8442
\.initial:o	8442
\.initial:x	8442
\.int_gset:N	8450
\.int_gset:c	8450
\.int_set:N	8450
\.int_set:c	8450
\.meta:n	8458
\.meta:nn	8460
\.multichoice:	8462
\.multichoices:Vn	8462
\.multichoices:nn	8462
\.multichoices:on	8462
\.multichoices:xn	8462
\.skip_gset:N	8472
\.skip_gset:c	8472
\.skip_set:N	8472
\.skip_set:c	8472
\.tl_gset:N	8480
\.tl_gset:c	8480
\.tl_gset_x:N	8480
\.tl_gset_x:c	8480
\.tl_set:N	8480
\.tl_set:c	8480
\.tl_set_x:N	8480
\.tl_set_x:c	8480
\.value_forbidden:	8496
\.value_required:	8496
\/	302, 10883
?:	180
\:	1061, 2687, 2886
\:::	34, 1597, 1598, 1599, 1599, 1600, 1601, 1602, 1603, 1605, 1607, 1614, 1619, 1625, 1744, 1745, 1746, 1747, 1748, 1749, 1750, 1751, 1752, 1753, 1754, 1755, 1756, 1757, 1758, 1759, 1760, 1761, 1762, 1763, 1764, 1765, 1766, 1767, 1768, 1769,

1770, 1771, 1773, 1778, 1780, 1785, 1790, 1827, 1828, 1829, 1830, 1831	8714, 8727, 8728, 8734, 8741, 8754, 8760, 9221, 9358, 9365, 9372, 9798, 9801, 9802, 9803, 9810, 9813, 9814
\::N 34, <u>1601</u> , 1601, 1754, 1760, 1761, 1765, 1766, 1830	_fp_decimate_auxi:Nnnnn <u>9548</u>
\::V 34, <u>1619</u> , 1619, 1750	_fp_decimate_auxii:Nnnnn <u>9548</u>
\::V_unbraced <u>1772</u> , 1780	_fp_decimate_auxiii:Nnnnn <u>9548</u>
\::c 34, <u>1603</u> , 1603, 1745, 1753, 1755, 1762, 1769, 1770	_fp_decimate_auxiv:Nnnnn <u>9548</u>
\::f 34, <u>1607</u> , 1607, 1746, 1747, 1748, 1752, 1829	_fp_decimate_auxv:Nnnnn <u>9548</u>
\::f_unbraced <u>1772</u> , 1773	_fp_decimate_auxvi:Nnnnn <u>9548</u>
\::n 34, <u>1600</u> , 1600, 1745, 1748, 1749, 1750, 1756, 1760, 1762, 1763, 1765, 1767, 1768, 1770, 1827, 1830	_fp_decimate_auxvii:Nnnnn <u>9548</u>
\::o .. 34, <u>1605</u> , 1605, 1746, 1749, 1751, 1752, 1753, 1757, 1758, 1760, 1761, 1763, 1764, 1766, 1768, 1771, 1828	_fp_decimate_auxviii:Nnnnn ... <u>9548</u>
\::o_unbraced <u>1772</u> , 1778, 1827, 1828, 1829, 1830	_fp_decimate_auxxi:Nnnnn <u>9548</u>
\::p 34, <u>1602</u> , 1602	_fp_decimate_auxxii:Nnnnn <u>9548</u>
\::v 34, <u>1619</u> , 1625	_fp_decimate_auxxiii:Nnnnn ... <u>9548</u>
\::v_unbraced <u>1772</u> , 1785	_fp_decimate_auxxiv:Nnnnn <u>9548</u>
\::x 34, <u>1614</u> , 1614, 1744, 1754, 1755, 1756, 1757, 1758, 1759, 1765, 1766, 1767, 1768, 1769, 1770, 1771	_fp_decimate_auxxv:Nnnnn <u>9548</u>
\::x_unbraced <u>1772</u> , 1790, 1831	_fp_decimate_auxxvi:Nnnnn <u>9548</u>
\:N <u>10852</u>	\{ ... 4, 2610, 7431, 8031, 8035, 8036, 9215
\; 2687, 2885, 2886	\} ... 5, 2610, 7432, 8031, 8035, 8036, 9217
\= 8057, 8059	\^ 7, 10, 90, 2598, 2610, 7942, 9190, 10882, 12850, 15473, 15480
\? <u>10982</u>	_ 2599, 2610, 10012
\@ 1061, 1062, 4450, 4451	_bool__0:w <u>2187</u>
\@end 726	_bool__1:w <u>2185</u>
\@hyph 729	_bool_(:Nw <u>2165</u>
\@input 730	_bool_)_0:w <u>2181</u>
\@italiccorr 731	_bool_)_1:w <u>2181</u>
\@underline 732	_bool_:Nw <u>2163</u>
\@addtofilelist 8938	_bool_S_0:w <u>2181</u>
\@currname 8831	_bool_S_1:w <u>2181</u>
\@filelist 8937, 8964, 8966, 8981	_bool_choose:NNN 2167, 2171, <u>2172</u> , 2172
\@ifpackageloaded 142, 223	_bool_eval_skip_to_end_auxi:Nw <u>2187</u> , 2187, 2188, 2190, 2204
\@namedef 205	_bool_eval_skip_to_end_auxii:Nw <u>2187</u> , 2192, 2196
\@nil 200, 208	_bool_eval_skip_to_end_auxiii:Nw <u>2187</u> , 2200, 2202
\@popfilename 170, 188, 190	_bool_get_next:NN 2151, 2153, 2153, 2164, 2168, 2185, 2186
\@pushfilename 170, 171, 186	_bool_if_left_parentheses:wwn <u>2133</u> , 2136, 2143, 2144
\@tempa 55, 57, 65	_bool_if_or:wwn 2133, 2139, 2147, 2148
\\ 1458, 2610, 7265, 7266, 7267, 7268, 7368, 7386, 7388, 7393, 7394, 7408, 7409, 7412, 7413, 7484, 7573, 7581, 7790, 7797, 7809, 7810, 7825, 7826, 8031, 8035, 8040,	_bool_if_parse:NNNww . 2140, <u>2149</u> , 2149
	_bool_if_right_parentheses:wwn <u>2133</u> , 2138, 2145, 2146
	_bool_p:Nw <u>2170</u>
	_box_resize:Nnn <u>14601</u> , 14617, 14622, 14646, 14663

- _box_resize_common:N 14629, 14685, 14690, 14690
- _box_rotate:N 14487, 14495, 14499
- _box_rotate_quadrant_four: 14487, 14514, 14588
- _box_rotate_quadrant_one: 14487, 14508, 14555
- _box_rotate_quadrant_three: 14487, 14513, 14577
- _box_rotate_quadrant_two: 14487, 14509, 14566
- _box_rotate_x:nnN 14487, 14533, 14561, 14563, 14572, 14574, 14583, 14585, 14594, 14596
- _box_rotate_y:nnN 14487, 14544, 14557, 14559, 14568, 14570, 14579, 14581, 14590, 14592
- _box_show:NNnn . 6330, 6340, 6347, 6347
- _chk_if_exist_cs:N 24, 1200, 1200, 1209, 1853
- _chk_if_exist_cs:c 1031, 1036, 1041, 1046, 1200, 1208
- _chk_if_free_cs:N 25, 1177, 1177, 1187, 1199, 1214, 1262, 3234, 3249, 3919, 4158, 4248, 4318, 4324, 4329, 5074, 5966, 6236, 14331, 14354
- _chk_if_free_cs:c 1177, 1198
- _chk_if_free_msg:nn 7319, 7319, 7329, 7342
- _clist_concat:NNNN 5569, 5570, 5572, 5573
- _clist_count:n 5881, 5886, 5899
- _clist_count:w . 5881, 5895, 5900, 5904
- _clist_get:wN .. 5645, 5650, 5653, 5687
- _clist_if_empty_n:w 14901, 14903, 14908, 14911
- _clist_if_empty_n:wNw 14901, 14912, 14914
- _clist_if_in_return:nn 5779, 5781, 5786, 5788
- _clist_item:nnNn 14810, 14812, 14818, 14842
- _clist_item_N_loop:nw 14810, 14815, 14833, 14837
- _clist_item_n:nw .. 14840, 14845, 14848
- _clist_item_n_end:n 14840, 14856, 14864
- _clist_item_n_loop:nw 14840, 14849, 14850, 14853, 14858
- _clist_item_n_strip:w 14840, 14867, 14869
- _clist_map_function:Nw 5804, 5808, 5813, 5817, 5840
- _clist_map_function_n:Nn 5820, 5822, 5826, 5830
- _clist_map_unbrace:Nw 5820, 5829, 5833
- _clist_map_variable:Nnw 5853, 5858, 5869, 5874
- _clist_pop:NNN . 5656, 5657, 5659, 5660
- _clist_pop:wN 5656, 5673, 5679
- _clist_pop:wwNNN 5656, 5665, 5668, 5703
- _clist_pop_TF:NNN 5682, 5695, 5697, 5698
- _clist_put_left:NNNn 5619, 5620, 5622, 5623
- _clist_put_right:NNNn 5632, 5633, 5635, 5636
- _clist_remove_all: 5746, 5756, 5760, 5772
- _clist_remove_all:NNn 5746, 5747, 5749, 5750
- _clist_remove_all:w .. 5746, 5773, 5774
- _clist_remove_duplicates:NN 5730, 5731, 5733, 5734
- _clist_set_from_seq:NNNN 14870, 14871, 14873, 14874
- _clist_set_from_seq:w 14870, 14889, 14893
- _clist_tmp:w 5550, 5550, 5752, 5773, 5790, 5792
- _clist_trim_spaces:n 5592, 5592, 5614, 5616, 14899
- _clist_trim_spaces:nn 5592, 5595, 5599, 5605, 5610
- _clist_trim_spaces_generic:nn 5586, 5589, 5591
- _clist_trim_spaces_generic:nw 5586, 5586, 5594, 5604, 5609, 5822, 5830
- _clist_use:nwn 5906, 5920, 5931
- _clist_use:nwwwnwn 5906, 5917, 5919, 5928
- _clist_use:wn 5906, 5913, 5914, 5927
- _clist_wrap_item:n 14870, 14882, 14886
- _coffin_align:NnnNnnnnN 6856, 6893, 6911, 6919, 6919, 7010
- _coffin_calculate_intersection:Nnn 6748, 6748, 6921, 6924, 7210
- _coffin_calculate_intersection:nnnnnnnn 6748, 6754, 6763, 7156
- _coffin_calculate_intersection_aux:nnnnnnN 6748, 6775, 6790, 6799, 6806, 6832, 6841

_coffin_display_attach:Nnnnn	7122, 7161, 7183, 7202, 7208
_coffin_display_handles_aux:nnnn	7122, 7189, 7194, 7200
_coffin_display_handles_aux:nnnnnn	7122, 7147, 7151
_coffin_find_bounding_shift:	14936, 15034, 15034
_coffin_find_bounding_shift_aux:nn	15034, 15038, 15040
_coffin_find_corner_maxima:N	14935, 15014, 15014
_coffin_find_corner_maxima_aux:nn	15014, 15021, 15023
_coffin_get_pole:NnN	6645, 6645, 6750, 6751, 6974, 6975, 6978, 6979, 7136, 7137, 7140
_coffin_gset_eq_structure:NN	6662, 6669
_coffin_if_exist:NT	6497, 6497, 6508, 6528, 6545, 6575, 6592, 6626, 6678, 6689, 7234
_coffin_mark_handle_aux:nnnnNnn	7067, 7105, 7110, 7114
_coffin_offset_corner:Nnnnn	6957, 6960, 6962
_coffin_offset_corners:Nnn	6876, 6877, 6883, 6884, 6957, 6957
_coffin_offset_pole:Nnnnnnn	6938, 6941, 6943
_coffin_offset_poles:Nnn	6874, 6875, 6880, 6881, 6903, 6904, 6938, 6938	
_coffin_reset_structure:N	6511, 6537, 6558, 6582, 6605, 6655, 6655, 6868, 6898
_coffin_resize_common:Nnn	15075, 15078, 15078, 15103
_coffin_rotate_bounding:nnn	14934, 14971, 14971
_coffin_rotate_corner:Nnnn	14929, 14971, 14977
_coffin_rotate_pole:Nnnnnn	14931, 14983, 14983
_coffin_rotate_vector:nnNN	14973, 14979, 14985, 14986, 14995, 14995	
_coffin_scale_corner:Nnnn	15081, 15114, 15114
_coffin_scale_pole:Nnnnnn	15083, 15114, 15120
_coffin_scale_vector:nnNN	15107, 15107, 15116, 15122
_coffin_set_bounding:N	14932, 14959, 14959
_coffin_set_eq_structure:NN	6629, 6662, 6662
_coffin_set_pole:Nnn	6676, 6698, 6702	
_coffin_set_pole:Nnx	6562, 6609, 6676, 6680, 6691, 6950, 6987, 6991, 6999, 7003, 14988, 15123
_coffin_shift_corner:Nnnn	14954, 15045, 15045
_coffin_shift_pole:Nnnnnn	14956, 15045, 15053
_coffin_update_B:nnnnnnnnN	6972, 6980, 6995
_coffin_update_T:nnnnnnnnN	6972, 6976, 6983
_coffin_update_corners:N	6539, 6560, 6584, 6607, 6703, 6703
_coffin_update_poles:N	6538, 6559, 6583, 6606, 6714, 6714, 6871, 6902	
_coffin_update_vertical_poles:NNN	6887, 6906, 6972, 6972
_coffin_x_shift_corner:Nnnn	15087, 15129, 15129
_coffin_x_shift_pole:Nnnnnn	15089, 15129, 15136
_cs_count_signature:N	25, 1309, 1309, 1320
_cs_count_signature:c	1309, 1319	
_cs_count_signature:nnN	1309, 1310, 1311	
_cs_generate_from_signature:NNn	1338, 1342
_cs_generate_from_signature:nnNNNn	1344, 1347
_cs_generate_internal_variant:n	1996, 2009, 2015
_cs_generate_internal_variant:wwnNwnn	2017, 2028
_cs_generate_internal_variant:wwnw	2009
_cs_generate_internal_variant_loop:n	2009, 2026, 2035, 2038
_cs_generate_variant:N	1854, 1861, 1869	
_cs_generate_variant:Nnnw	1897, 1899, 1899, 1917
_cs_generate_variant:nnNN	1857, 1890, 1890
_cs_generate_variant:ww	1861, 1874, 1882	

_cs_generate_variant:wwNN	_dim_case:nnTF
..... 1906, 1989, 1989	.. 4039, 4042, 4047, 4052, 4057, 4059
_cs_generate_variant:wwNw	_dim_case:nw ... 4039, 4060, 4061, 4065
..... 1861, 1883, 1884	_dim_case_end:nw 4039, 4064, 4067
_cs_generate_variant_loop:nNwN ...	_dim_compare:w 4003, 4005, 4008
..... 1907, 1919, 1919, 1933	_dim_compare:wNN 4003, 4011, 4014, 4026
_cs_generate_variant_loop_end:nwwwNNnn	_dim_compare_:w 4003
..... 1909, 1919, 1940	_dim_compare_<:w 4003
_cs_generate_variant_loop_invalid:NNwNNnn	_dim_compare_>:w 4003
..... 1919, 1926, 1962	_dim_compare_end:w 4011, 4037
_cs_generate_variant_loop_long:wNNnn	_dim_eval:w 89, 3913,
..... 1912, 1919, 1949	3914, 3943, 3954, 3959, 3966, 3972,
_cs_generate_variant_loop_same:w .	3973, 3974, 3980, 3981, 3982, 3997,
..... 1919, 1922, 1935	4000, 4006, 4027, 4032, 4125, 4127,
_cs_generate_variant_same:N	4131, 4148, 6274, 6276, 6278, 6287,
..... 1938, 1977, 1977	6289, 6291, 6293, 6367, 6387, 6400,
_cs_get_function_name:N 25, 1079, 1079	6415, 6443, 10123, 10666, 14726,
_cs_get_function_signature:N	14728, 14769, 14771, 14941, 15338
..... 25, 1079, 1081	_dim_eval_end: 89, 3913, 3915,
_cs_parm_from_arg_count:nnF	3943, 3954, 3959, 3966, 3976, 3984,
..... 909, 1279, 1279, 1323	3997, 4000, 4125, 4127, 4131, 6274,
_cs_parm_from_arg_count_test:nnF .	6276, 6278, 6287, 6289, 6291, 6293,
..... 1279, 1281, 1300	6367, 6387, 6400, 6415, 6443, 14726,
_cs_show:www 1443, 1454, 1458	14728, 14769, 14771, 14943, 15338
_cs_split_function:NN	_dim_maxmin:wwN 3963, 3972, 3980, 3986
.... 25, 891, 904, 991, 992, 1060,	_dim_ratio:n 3994, 3995, 3996
1066, 1080, 1082, 1310, 1344, 1855	_dim_strip_bp:n 89, 4126, 4126
_cs_split_function_auxi:w	_dim_strip_pt:n
..... 1060, 1069, 1074 89, 4127, 4128, 4128, 15328
_cs_split_function_auxii:w	_dim_strip_pt:w 4128, 4131, 4135
..... 1060, 1075, 1076	_dim_to_unit:n 15329, 15333, 15334, 15337
_cs_tmp:w 25,	_driver_box_rotate_begin: 14522
1210, 1218, 1219, 1220, 1221, 1222,	_driver_box_rotate_end: 14524
1223, 1224, 1225, 1226, 1228, 1229,	_driver_box_scale_begin: 14694
1230, 1231, 1232, 1233, 1234, 1235,	_driver_box_scale_end: 14696
1236, 1237, 1238, 1239, 1240, 1241,	_driver_box_use_clip:N 14720
1242, 1243, 1244, 1245, 1246, 1247,	_driver_color_ensure_current:
1248, 1249, 1250, 1251, 1334, 1359, 7286, 7292, 7295
1360, 1361, 1362, 1363, 1364, 1365,	_exp_arg_last_unbraced:nn
1366, 1367, 1368, 1369, 1370, 1371,	.. 1772, 1772, 1775, 1779, 1782, 1787
1372, 1373, 1374, 1375, 1376, 1377,	_exp_arg_next:Nnn 1597, 1598, 1604
1378, 1379, 1380, 1381, 1382, 1383,	_exp_arg_next:nnn 1597,
1391, 1392, 1393, 1394, 1395, 1396,	1597, 1606, 1609, 1617, 1621, 1627
1397, 1398, 1399, 1400, 1401, 1402,	_exp_eval_error_msg:w 1631, 1635, 1644
1403, 1404, 1405, 1406, 1407, 1408,	_exp_eval_register:N
1409, 1410, 1411, 1412, 1413, 1414, 1622, 1631, 1631,
1872, 1887, 1997, 2019, 4212, 4222	1643, 1677, 1695, 1713, 1714, 1721,
_cs_to_str:N ... 1051, 1055, 1057, 1058	1783, 1796, 1808, 1814, 1823, 1843
_cs_to_str:w 1051, 1054, 1058	_exp_eval_register:c .. 1628, 1631,
_dim_abs:N 3963, 3965, 3968	1642, 1672, 1689, 1788, 1798, 1848

```

\__expl_last_two_unbraced:nonN ..... 597, 598, 599, 600, 601, 602, 603,
..... 1832, 1833, 1834
\__expl_package_check: ..... 604, 605, 606, 607, 608, 609, 610,
..... 7, 219, 748, 1595, 611, 612, 613, 614, 615, 616, 617,
2058, 2349, 2475, 3154, 3911, 4314, 618, 619, 620, 621, 622, 623, 624,
5062, 5546, 5957, 6231, 6450, 7276, 625, 626, 627, 628, 629, 630, 631,
7309, 8048, 8820, 9379, 14297, 14472 632, 633, 634, 635, 636, 637, 638,
\__expl_primitive:NN 294, 294, 301, 302, 639, 640, 641, 642, 643, 644, 645,
303, 304, 305, 306, 307, 308, 309, 646, 647, 648, 649, 650, 651, 652,
310, 311, 312, 313, 314, 315, 316, 653, 654, 655, 656, 657, 658, 659,
317, 318, 319, 320, 321, 322, 323, 660, 661, 662, 663, 664, 665, 666,
324, 325, 326, 327, 328, 329, 330, 667, 668, 669, 670, 671, 672, 673,
331, 332, 333, 334, 335, 336, 337, 674, 675, 676, 677, 678, 679, 680,
338, 339, 340, 341, 342, 343, 344, 681, 682, 683, 684, 685, 686, 687,
345, 346, 347, 348, 349, 350, 351, 688, 689, 690, 691, 692, 693, 694,
352, 353, 354, 355, 356, 357, 358, 695, 696, 697, 698, 699, 700, 701,
359, 360, 361, 362, 363, 364, 365, 702, 703, 704, 705, 706, 707, 708,
366, 367, 368, 369, 370, 371, 372, 709, 710, 711, 712, 713, 714, 715,
373, 374, 375, 376, 377, 378, 379, 716, 717, 718, 719, 720, 721, 722, 723
380, 381, 382, 383, 384, 385, 386, \__expl_status_pop:w ..... 207
387, 388, 389, 390, 391, 392, 393, \__file_add_path:nN .... 8871, 8872, 8873
394, 395, 396, 397, 398, 399, 400, \__file_add_path_search:nN .....
401, 402, 403, 404, 405, 406, 407, ..... 8871, 8877, 8881
408, 409, 410, 411, 412, 413, 414, \__file_input:V ..... 8922
415, 416, 417, 418, 419, 420, 421, \__file_input:n ..... 8924, 8930
422, 423, 424, 425, 426, 427, 428, \__file_input:n\__file_input:V .. 8914
429, 430, 431, 432, 433, 434, 435, \__file_input_aux:n 8914, 8927, 8931, 8947
436, 437, 438, 439, 440, 441, 442, \__file_input_aux:o ..... 8914, 8928
443, 444, 445, 446, 447, 448, 449, \__file_name_sanitiz:nn .....
450, 451, 452, 453, 454, 455, 456, ..... 168, 8851, 8851, 8872,
457, 458, 459, 460, 461, 462, 463, 8919, 8949, 8957, 9000, 9010, 9115
464, 465, 466, 467, 468, 469, 470, \__file_path_include:n . 8948, 8949, 8950
471, 472, 473, 474, 475, 476, 477, \__fp_ ..... 11317, 11324
478, 479, 480, 481, 482, 483, 484, \__fp__o:ww ..... 11314
485, 486, 487, 488, 489, 490, 491, \__fp_*o:ww ..... 11697
492, 493, 494, 495, 496, 497, 498, \__fp_+o:ww ..... 11415
499, 500, 501, 502, 503, 504, 505, \__fp_-o:ww ..... 11410
506, 507, 508, 509, 510, 511, 512, \__fp_/o:ww ..... 11807
513, 514, 515, 516, 517, 518, 519, \__fp_^o:ww ..... 12850
520, 521, 522, 523, 524, 525, 526, \__fp_abs:NNN ..... 14196,
527, 528, 529, 530, 531, 532, 533, 14196, 14197, 14200, 14206, 14210
534, 535, 536, 537, 538, 539, 540, \__fp_abs_o:w ..... 14196, 14197
541, 542, 543, 544, 545, 546, 547, \__fp_acos_o:w ..... 13775, 13775
548, 549, 550, 551, 552, 553, 554, \__fp_acot_o:Nw 10689, 10691, 13598, 13603
555, 556, 557, 558, 559, 560, 561, \__fp_acotii_o:Nww .....
562, 563, 564, 565, 566, 567, 568, ..... 13601, 13606, 13622, 13641
569, 570, 571, 572, 573, 574, 575, \__fp_acsc_normal_o:NfwNnw .....
576, 577, 578, 579, 580, 581, 582, ..... 13833, 13848, 13856, 13856
583, 584, 585, 586, 587, 588, 589, \__fp_acsc_o:w ..... 13827, 13827
590, 591, 592, 593, 594, 595, 596, \__fp_add:NNNn ..... 14151,
14151, 14152, 14153, 14154, 14155

```


_fp_add_big_i_o:wNww	_fp_atan_combine_aux:ww
..... 11486, 11493, 11493, 12678 13726, 13740, 13747
_fp_add_big_ii_o:wNww	_fp_atan_combine_o:NwwwwN
..... 11489, 11493, 11501 13645, 13662, 13726, 13726
_fp_add_inf_o:Nww . 11430, 11450, 11450	_fp_atan_dispatch_o:NNnNw
_fp_add_normal_o:Nww 11429, 11470, 11470 13598, 13600, 13605, 13608
_fp_add_npos_o:NnwNnw	_fp_atan_div:wnwnw 13673, 13675, 13675
..... 11473, 11479, 11479	_fp_atan_inf_o:NNNw . 13633, 13634,
_fp_add_return_ii_o:Nww	13635, 13643, 13643, 13778, 13851
..... 11432, 11438, 11438, 11443	_fp_atan_near:www 13675, 13681, 13687
_fp_add_significand_carry_o:wwwNN	_fp_atan_near_aux:wnw
..... 11526, 11541, 11541 13675, 13692, 13694
_fp_add_significand_no_carry_o:wwwNN	_fp_atan_normal_o:NNnwNnw
..... 11528, 11531, 11531 13637, 13653, 13653
_fp_add_significand_o:NnnwnnnN ..	_fp_atan_o:Nw 10693, 10695, 13598, 13598
..... 11496, 11504, 11509, 11509	_fp_atan_test_o:NwnNwN
_fp_add_significand_pack:NNNNNN 13656, 13660, 13660, 13815
..... 11509, 11513, 11516	_fp_atanii_o:Nww
_fp_add_significand_test_o:N 13601, 13606, 13622, 13622, 13642
..... 11509, 11511, 11523	_fp_basics_pack_high:NNNNNw
_fp_add_zeros_o:Nww 11428, 11440, 11440 11384, 11391, 11534,
_fp_and_return:wNw 11314, 11320, 11326	11685, 11786, 11798, 11938, 12405
_fp_array_count:n	_fp_basics_pack_high_carry:w
..... 9624, 9624, 10741, 13611 11384, 11394, 11398
_fp_array_count_loop:Nw	_fp_basics_pack_low:NNNNNw
..... 9624, 9627, 9631, 9632 11384, 11384, 11536,
_fp_array_to_clist:n	11687, 11788, 11800, 11940, 12407
... 9927, 10933, 10934, 14098, 14098	_fp_basics_pack_weird_high:NNNNNNNw
_fp_array_to_clist_loop:Nw 11400, 11408, 11545, 11949
..... 14098, 14105, 14110, 14120	_fp_basics_pack_weird_low:NNNNw ..
_fp_asec_o:w 11400, 11400, 11547, 11951
..... 13840, 13840	_fp_case_return:nw
_fp_asin_auxi_o:NnNww 9579, 9579, 9593, 9595, 9987,
..... 13805, 13808, 13808, 13867	12611, 13633, 13634, 13635, 13898,
_fp_asin_isqrt:wn . 13808, 13811, 13818	13948, 14016, 14018, 14019, 14064
_fp_asin_normal_o:NfwNnnnw	_fp_case_return_i_o:ww . 9586, 9586,
..... 13766, 13782, 13793, 13793	11431, 11445, 11454, 11730, 13625
_fp_asin_o:w	_fp_case_return_ii_o:ww
..... 13760, 13760	9586, 9588, 11731, 12899, 12917, 13626
_fp_assign_to:nNNNn . 14230, 14237,	_fp_case_return_o:Nw
14238, 14239, 14240, 14241, 14242 9580, 9580, 12646,
_fp_assign_to_i:wNNNn	12651, 12654, 12854, 12859, 12881,
..... 14230, 14244, 14247	12890, 13111, 13141, 13834, 13836
_fp_assign_to_ii:NnNNN	_fp_case_return_o:Nww
..... 14230, 14249, 14253 9584, 9584, 11732, 11733,
_fp_atan_Taylor_break:w	11736, 11737, 12901, 12910, 12913
..... 13710, 13713, 13723	_fp_case_return_same_o:w
_fp_atan_Taylor_loop:www 9582, 9582, 12432,
..... 13705, 13710, 13710, 13718	12658, 12878, 13096, 13104, 13119,
_fp_atan_auxi:ww .. 13685, 13699, 13699	
_fp_atan_auxii:w .. 13699, 13700, 13701	

13134, 13149, 13156, 13164, 13179,
 13763, 13771, 13789, 13835, 13852
 __fp_case_use:nw . 9578, 9578, 11456,
 11728, 11729, 11734, 11735, 11815,
 11818, 12425, 12428, 12884, 13097,
 13102, 13112, 13117, 13127, 13132,
 13142, 13147, 13157, 13162, 13172,
 13177, 13765, 13768, 13778, 13780,
 13786, 13830, 13832, 13843, 13846,
 13851, 13901, 13912, 13951, 13961
 __fp_chk:w
 9393, 9394, 9397, 9406, 9407, 9408,
 9409, 9410, 9412, 9413, 9416, 9422,
 9426, 9446, 9449, 9450, 9460, 9470,
 9483, 9502, 9590, 9766, 9771, 9930,
 9939, 9941, 10646, 11137, 11162,
 11163, 11273, 11274, 11277, 11288,
 11289, 11297, 11298, 11306, 11317,
 11320, 11332, 11357, 11416, 11435,
 11436, 11438, 11439, 11440, 11448,
 11451, 11467, 11468, 11470, 11479,
 11555, 11706, 11740, 11741, 11744,
 11823, 11959, 11963, 12422, 12434,
 12436, 12643, 12660, 12662, 12851,
 12870, 12872, 12873, 12875, 12887,
 12892, 12894, 12919, 12920, 12922,
 12938, 13023, 13036, 13038, 13042,
 13046, 13093, 13106, 13108, 13121,
 13123, 13136, 13138, 13151, 13153,
 13166, 13168, 13181, 13191, 13623,
 13638, 13639, 13643, 13654, 13760,
 13773, 13775, 13791, 13794, 13804,
 13827, 13838, 13840, 13854, 13856,
 13861, 13894, 13919, 13922, 13944,
 13968, 13971, 14012, 14035, 14085,
 14086, 14187, 14194, 14247, 14256
 __fp_compare:wNNNNw 11014
 __fp_compare_aux:wn 11145, 11148, 11156
 __fp_compare_back:ww
 .. 11101, 11158, 11161, 11161, 11287
 __fp_compare_nan:w
 .. 11161, 11166, 11167, 11188
 __fp_compare_npos:nwnw
 .. 11172, 11189, 11189, 11557, 12208
 __fp_compare_return:w 11132, 11134, 11137
 __fp_compare_significand:nnnnnnnn .
 .. 11189, 11192, 11197
 __fp_cos_o:w 13108, 13108, 14240
 __fp_cot_o:w 13168, 13168
 __fp_cot_zero_o:Nfw
 .. 13126, 13168, 13171, 13183
 __fp_csc_o:w 13123, 13123
 __fp_decimate:nNnnnn
 ... 9532, 9532, 9606, 9943, 11495,
 11503, 11582, 12694, 12698, 13977
 __fp_decimate_:Nnnnn 9544, 9544
 __fp_decimate_pack:nnnnnnnnnw
 .. 9555, 9574, 9574
 __fp_decimate_pack:nnnnnnnw . 9575, 9576
 __fp_decimate_tiny:Nnnnn . . . 9544, 9546
 __fp_div_npos_o:Nww 11812, 11822, 11822
 __fp_div_significand_calc:wwnnnnnnnn
 .. 11839,
 11848, 11848, 11894, 12504, 12511
 __fp_div_significand_calc_i:wwnnnnnnnn
 .. 11848, 11851, 11856
 __fp_div_significand_calc_ii:wwnnnnnnnn
 .. 11848, 11853, 11873
 __fp_div_significand_i_o:wnnw
 .. 11829, 11835, 11835
 __fp_div_significand_ii:wwn
 .. 11843, 11844, 11845, 11890, 11890
 __fp_div_significand_iii:wwnnnnnn . .
 .. 11846, 11897, 11897
 __fp_div_significand_iv:wwnnnnnnnn .
 .. 11900, 11905, 11905
 __fp_div_significand_large_o:wwwNNNNwN
 .. 11931, 11945, 11945
 __fp_div_significand_pack:NNN 11892,
 11925, 11925, 12491, 12509, 12517
 __fp_div_significand_small_o:wwwNNNNwN
 .. 11929, 11935, 11935
 __fp_div_significand_test_o:w
 .. 11837, 11926, 11926
 __fp_div_significand_v:NN
 .. 11910, 11912, 11915
 __fp_div_significand_v:NNw 11905
 __fp_div_significand_vi:Nw
 .. 11905, 11908, 11916
 __fp_division_by_zero_o:NNww
 .. 9738, 9778, 9782, 11816, 11819, 12886
 __fp_division_by_zero_o:Nnw
 .. 9730, 9778, 9781, 12429, 13187, 13188
 __fp_ep_compare:www 12203, 12203, 13669
 __fp_ep_compare_aux:www
 .. 12203, 12204, 12205
 __fp_ep_div:wwwn 12233, 12233, 12344,
 13588, 13684, 13688, 13697, 13864

_fp_ep_div_eps_pack:NNNNw	_fp_exp_Taylor:Nnnwn
..... 12263 , 12267 , 12269 , 12272 12695 , 12711 , 12711 , 12847
_fp_ep_div_epsi:wnNNNNn	_fp_exp_Taylor_break:Nww
..... 12260 , 12263 , 12263 12711 , 12725 , 12736
_fp_ep_div_epsi:wwNNNNn	_fp_exp_Taylor_ii:ww ... 12717 , 12720
..... 12263 , 12265 , 12274	_fp_exp_Taylor_loop:www
_fp_ep_div_esti:wwwwn 12711 , 12721 , 12722 , 12731
..... 12239 , 12242 , 12242	_fp_exp_after_?_f:nw
_fp_ep_div_estii:wwnnwn 10064
..... 12242 , 12244 , 12250	_fp_exp_after_array_f:w
_fp_ep_div_estiii:NNNNwwwwn 9503 , 9503 , 9506 ,
..... 12242 , 12252 , 12257	10618 , 11336 , 11347 , 11370 , 11378
_fp_ep_inv_to_float:wwN	_fp_exp_after_f:nw
.. 12340 , 12342 , 12348 , 13130 , 13145 9450 , 9470 , 10645 , 10783
_fp_ep_isqrt:wn .. 12286 , 12286 , 13825	_fp_exp_after_mark_f:nw . 10064 , 10073
_fp_ep_isqrt_aux:wn	_fp_exp_after_normal:Nwwww 9492 , 9500
..... 12286	_fp_exp_after_normal:nNNw
_fp_ep_isqrt_auxi:wn .. 12289 , 12291 9453 , 9463 , 9473 , 9490 , 9490
_fp_ep_isqrt_auxii:wwnnwn	_fp_exp_after_o:nw
..... 12286 , 12293 , 12299 9450 , 9450 , 9583 ,
_fp_ep_isqrt_epsi:wN 12323 , 12326 , 12326	9587 , 9589 , 9937 , 9981 , 9999 , 11305 ,
_fp_ep_isqrt_epsi:wwN	11322 , 11439 , 11961 , 13035 , 13040
.. 12326 , 12329 , 12330 , 12331 , 12333	_fp_exp_after_special:nNNw
_fp_ep_isqrt_esti:wwnnwn 9455 , 9465 , 9475 , 9480 , 9480
..... 12301 , 12304 , 12304 , 12309	_fp_exp_after_stop_f:nw ... 9503 , 9509
_fp_ep_isqrt_estii:wwnnwn	_fp_exp_large:w
..... 12304 , 12307 , 12314	12747 , 12752 , 12753 , 12754 , 12755 ,
_fp_ep_isqrt_estiii:NNNNwwwwn	12756 , 12757 , 12758 , 12759 , 12760 ,
..... 12304 , 12316 , 12320	12768 , 12769 , 12770 , 12771 , 12772 ,
_fp_ep_mul:wwwwn	12773 , 12774 , 12775 , 12776 , 12784 ,
..... 12218 , 13545 , 13575 , 13812 , 13823	12785 , 12786 , 12787 , 12788 , 12789 ,
_fp_ep_mul_raw:wwwN	12790 , 12791 , 12792 , 12800 , 12801 ,
.. 12218 , 12224 , 12228 , 13209 , 13495	12802 , 12803 , 12804 , 12805 , 12806 ,
_fp_ep_to_ep:wwN 12169 , 12169 , 12220 ,	12807 , 12808 , 12816 , 12817 , 12818 ,
12223 , 12235 , 12238 , 12288 , 13813	12819 , 12820 , 12821 , 12822 , 12823 ,
_fp_ep_to_ep_end:www 12169 , 12183 , 12187	12824 , 12832 , 12833 , 12834 , 12835 ,
_fp_ep_to_ep_loop:N	12836 , 12837 , 12838 , 12839 , 12840
12174 , 12178 , 12185 , 12188 , 13496	_fp_exp_large:wN . 12738 , 12827 , 12829
_fp_ep_to_ep_zero:ww 12169 , 12193 , 12201	_fp_exp_large_after:wn
_fp_ep_to_fixed:wn 12738 , 12843 , 12845
12151 , 13206 , 13691 , 13700 , 13810	_fp_exp_large_i:wN 12738 , 12811 , 12813
_fp_ep_to_fixed_auxi:www	_fp_exp_large_ii:wN 12738 , 12795 , 12797
..... 12151 , 12153 , 12158	_fp_exp_large_iii:wN 12738 , 12779 , 12781
_fp_ep_to_fixed_auxii:nnnnnnwn ..	_fp_exp_large_iv:wN 12738 , 12763 , 12765
..... 12151 , 12164 , 12167	_fp_exp_large_v:wN 12738 , 12749 , 13022
_fp_ep_to_float:wwN . 12340 , 12340 ,	_fp_exp_normal:w .. 12648 , 12662 , 12662
12345 , 12352 , 13100 , 13115 , 13594	_fp_exp_o:w
_fp_error:nfn 12643 , 12643 , 14237
..... 9707 , 9741 , 9769 , 9792 , 10932	_fp_exp_overflow:
_fp_error:nnfn . 9699 , 9716 , 9733 , 9792 12684 , 12709
_fp_error:nnnn	_fp_exp_pos:NNwnw . 12665 , 12667 , 12670
9792 , 9792 , 9794	_fp_exp_pos:Nnnw
 12662

__fp_exp_pos_large:NnnNwn
 12699, 12738, 12738
 __fp_expand:n 9633, 9633, 14102
 __fp_expand_loop:nwnN
 9633, 9635, 9637, 9640
 __fp_exponent:w 9426, 9426, 14283, 14288
 __fp_fixed_add:Nnnnnwnn
 12044, 12044, 12045, 12046
 __fp_fixed_add:nnNnnwn
 12044, 12052, 12054
 __fp_fixed_add:wn
 12044, 12044, 12284, 12585,
 12593, 12604, 12622, 13696, 13756
 __fp_fixed_add_after:NNNNwn
 12044, 12048, 12062
 __fp_fixed_add_one:wN 11975,
 11975, 12277, 12728, 12737, 13822
 __fp_fixed_add_pack:NNNNwn
 12044, 12050, 12057, 12060
 __fp_fixed_continue:wn 11974,
 11974, 12221, 12226, 12236, 12751,
 12767, 12783, 12799, 12815, 12831,
 12998, 13244, 13533, 13814, 13823
 __fp_fixed_div_int:wnN
 12013, 12018, 12026, 12038
 __fp_fixed_div_int:wwN
 .. 12013, 12013, 12584, 12727, 13715
 __fp_fixed_div_int_after:Nw
 12013, 12015, 12043
 __fp_fixed_div_int_auxi:wnn
 12013, 12019,
 12020, 12021, 12022, 12023, 12033
 __fp_fixed_div_int_auxii:wnn
 12013, 12024, 12041
 __fp_fixed_div_int_pack:Nw
 12013, 12036, 12042
 __fp_fixed_div_myriad:wn
 11980, 11980, 12281
 __fp_fixed_inv_to_float:wN
 12347, 12347, 12667, 12934
 __fp_fixed_mul:nnnnnnwn
 12064, 12084, 12086
 __fp_fixed_mul:wn 12064, 12064,
 12230, 12261, 12276, 12278, 12282,
 12335, 12338, 12351, 12586, 12596,
 12636, 12729, 12748, 12848, 12944,
 13502, 13556, 13703, 13736, 13738
 __fp_fixed_mul_add:Nwnnnwnnn
 .. 12098, 12108, 12118, 12122, 12122
 __fp_fixed_mul_add:nnnnnnnn
 12133, 12135, 12135
 __fp_fixed_mul_add:nnnnwnnwN
 12140, 12146, 12146
 __fp_fixed_mul_add:wwn .. 12092, 12092
 __fp_fixed_mul_after:wn
 11982, 11988, 11988, 11991,
 12066, 12094, 12104, 12114, 12961
 __fp_fixed_mul_one_minus_mul:wn 12092
 __fp_fixed_mul_short:wn
 11989, 11989,
 12259, 12280, 12322, 12324, 13749
 __fp_fixed_mul_sub_back:wwn
 12092, 12102,
 12336, 13523, 13525, 13526, 13527,
 13528, 13529, 13530, 13531, 13532,
 13536, 13538, 13539, 13540, 13541,
 13542, 13543, 13544, 13569, 13571,
 13572, 13573, 13574, 13577, 13579,
 13580, 13581, 13582, 13716, 13724
 __fp_fixed_one_minus_mul:wn ... 12112
 __fp_fixed_sub:wn
 12044, 12045, 12328, 12602, 12618,
 12630, 13248, 13697, 13754, 13820
 __fp_fixed_to_float:Nw
 12354, 12354, 12611
 __fp_fixed_to_float:wN
 12341, 12354, 12354, 12355,
 12631, 12641, 12665, 12930, 13744
 __fp_fixed_to_float_pack:ww 12386, 12396
 __fp_fixed_to_float_rad:wN
 12349, 12349, 13744
 __fp_fixed_to_float_round_up:wnnnnw
 12399, 12403
 __fp_fixed_to_float_zero:w 12382, 12391
 __fp_fixed_to_loop:N 12359, 12369, 12373
 __fp_fixed_to_loop_end:w . 12375, 12379
 __fp_from_dim:wNNnnnnnn
 14054, 14077, 14080
 __fp_from_dim:wNw .. 14054, 14066, 14075
 __fp_from_dim:wnnnnwnn .. 14081, 14082
 __fp_from_dim:wnnnnwnw 14054
 __fp_from_dim_test:ww
 .. 10121, 10664, 14054, 14056, 14061
 __fp_if_undefined:w 14186, 14187
 __fp_if_zero:w 14193, 14194
 __fp_inf_fp:N 9412, 9413, 9754
 __fp_infix_compare:N . 11014, 11016,
 11021, 11026, 11036, 11044, 11052

_fp_invalid_operation:nnw
 9696, 9778,
 9778, 9790, 13903, 13914, 13953, 13963
 _fp_invalid_operation_o:Nww
 9704, 9778, 9779,
 11459, 11462, 11734, 11735, 13029
 _fp_invalid_operation_o:fw
 9789, 13103, 13118,
 13133, 13148, 13163, 13178, 13769,
 13787, 13803, 13831, 13844, 13860
 _fp_invalid_operation_o:nw
 9789, 9789, 9791, 12425
 _fp_invalid_operation_tl_o:nf
 9713, 9778, 9780, 9926
 _fp_ln_Taylor:wwNw 12576, 12577, 12577
 _fp_ln_Taylor_break:w .. 12582, 12593
 _fp_ln_Taylor_loop:www
 12578, 12579, 12588
 _fp_ln_c:NwNw 12568, 12599, 12599
 _fp_ln_div_after:Nw 12472, 12520
 _fp_ln_div_i:w 12493, 12502
 _fp_ln_div_ii:wnw
 .. 12496, 12497, 12498, 12499, 12507
 _fp_ln_div_vi:wnw 12500, 12515
 _fp_ln_exponent:wn 12448, 12608, 12608
 _fp_ln_exponent_one:ww .. 12613, 12627
 _fp_ln_exponent_small:NNww
 12616, 12620, 12633
 _fp_ln_npos_o:w .. 12434, 12436, 12436
 _fp_ln_o:w 12422, 12422, 14238
 _fp_ln_significand:NNNNnnnN
 12447, 12450, 12450, 12942
 _fp_ln_square_t_after:w . 12544, 12575
 _fp_ln_square_t_pack:NNNNw
 .. 12546, 12548, 12550, 12552, 12573
 _fp_ln_t_large:NNw 12525, 12532, 12542
 _fp_ln_t_small:Nw 12523, 12530
 _fp_ln_twice_t_after:w .. 12556, 12572
 _fp_ln_twice_t_pack:Nw 12558,
 12560, 12562, 12564, 12566, 12571
 _fp_ln_x_ii:wnnnn . 12452, 12470, 12470
 _fp_ln_x_iii:NNNNw 12479, 12483
 _fp_ln_x_iii_var:NNNNw . 12477, 12484
 _fp_ln_x_iv:wnnnnnnnn .. 12475, 12489
 _fp_max_fp:N 9414, 9420
 _fp_min_fp:N 9414, 9414
 _fp_minmax_auxi:ww
 11281, 11293, 11300, 11300
 _fp_minmax_auxii:ww
 11283, 11291, 11300, 11302
 _fp_minmax_break_o:w 11274, 11304, 11304
 _fp_minmax_loop:Nww
 .. 11268, 11270, 11276, 11276, 11296
 _fp_minmax_o:Nw
 10697, 10699, 11265, 11265
 _fp_mul:NNNn .. 14216, 14216, 14217,
 14218, 14219, 14220, 14221, 14222
 _fp_mul_cases_o:NnNnw
 11699, 11705, 11809
 _fp_mul_cases_o:nNnnw 11705
 _fp_mul_npos_o:Nww
 11702, 11743, 11743, 14084
 _fp_mul_significand_drop:NNNNw ..
 11752,
 11761, 11763, 11765, 11767, 11771
 _fp_mul_significand_keep:NNNNw ..
 11752, 11757, 11759, 11773
 _fp_mul_significand_large_f:NwNNNN
 11780, 11784, 11784
 _fp_mul_significand_o:nnnnnnnnn ..
 11750, 11752, 11752
 _fp_mul_significand_small_f:NNwwwN
 11778, 11795, 11795
 _fp_mul_significand_test_f:NNN ...
 11754, 11775, 11775
 _fp_neg_sign:N 9435, 9435, 11413
 _fp_not_o:w 11306
 _fp_overflow:w 9442, 9778, 9783
 _fp_pack:NNNNw 9510, 9513
 _fp_pack:NNNNwn . 9510, 9514, 11984,
 11994, 11997, 12000, 12003, 12006,
 12068, 12071, 12074, 12077, 12080,
 12963, 12966, 12969, 12972, 12975
 _fp_pack_Bigg:NNNNNw
 9523, 9526, 11862,
 11865, 11868, 11879, 11882, 11885
 _fp_pack_big:NNNNNw 9516, 9519
 _fp_pack_big:NNNNNwn
 9516, 9521, 12096, 12106,
 12116, 12125, 12128, 12131, 12138
 _fp_pack_eight:wNNNNNNNN
 9530, 9530, 11678, 12160, 13215, 13216
 _fp_pack_twice_four:wNNNNNNNN 9528,
 9528, 9974, 9975, 11621, 11622,
 12161, 12162, 12163, 12195, 12196,
 12197, 12384, 12385, 12714, 12715,
 12716, 13217, 13218, 13431, 14077
 _fp_parse:n
 10005, 10118, 10751, 10751, 11135,
 11149, 11159, 13887, 13942, 14010,

- 14047, 14093, 14095, 14097, 14129,
14131, 14133, 14156, 14223, 14245
- _fp_parse_after:ww 10751, 10754, 10760
- _fp_parse_apply_binary:NwNwN
10774, 10774, 10892, 10937
- _fp_parse_apply_compare:NwNNNNwN .
11085, 11093
- _fp_parse_apply_juxtapose:NwNwN . . .
10914, 10918, 10928
- _fp_parse_apply_round:NNwN 10730, 10739
- _fp_parse_apply_unary:NNNwN
10567, 10567, 10576, 10672, 10681
- _fp_parse_compare:NNNNNNw
11014, 11047, 11055, 11072
- _fp_parse_compare_end:NNNN
11014, 11067, 11081
- _fp_parse_compare_expand:NNNNNw . .
11014,
11063, 11064, 11065, 11066, 11070
- _fp_parse_continue:NwN
10763, 10766, 10773,
10776, 11095, 11344, 11367, 11375
- _fp_parse_digits:N 10041, 10042
- _fp_parse_digits_i:N . . . 10023, 10040
- _fp_parse_digits_ii:N . . 10023, 10039
- _fp_parse_digits_iii:N . . 10023, 10038
- _fp_parse_digits_iv:N . . 10023, 10037
- _fp_parse_digits_v:N . . . 10023, 10036
- _fp_parse_digits_vi:N
10023, 10035, 10277, 10325
- _fp_parse_digits_vii:N
10023, 10264, 10314
- _fp_parse_expand:w
10005, 10005, 10007, 10032, 10069,
10092, 10151, 10182, 10220, 10222,
10241, 10243, 10265, 10282, 10295,
10315, 10345, 10373, 10389, 10400,
10423, 10452, 10462, 10469, 10482,
10498, 10518, 10529, 10586, 10612,
10621, 10677, 10686, 10734, 10737,
10738, 10757, 10830, 10844, 10869,
10896, 10921, 10961, 10973, 10990,
11006, 11032, 11079, 11091, 11340
- _fp_parse_exponent:N 10091,
10256, 10405, 10472, 10474, 10474
- _fp_parse_exponent:Nw
10280, 10293, 10342,
10370, 10421, 10450, 10469, 10469
- _fp_parse_exponent_aux:N
10474, 10477, 10484
- _fp_parse_exponent_body:N
10500, 10504, 10504
- _fp_parse_exponent_digits:N
10508, 10520, 10520, 10524
- _fp_parse_exponent_keep:N 10531
- _fp_parse_exponent_keep:NTF
10511, 10531
- _fp_parse_exponent_sign:N
10490, 10494, 10494, 10497
- _fp_parse_function:NNN
10670, 10679, 10689,
10691, 10693, 10695, 10697, 10699
- _fp_parse_infix:NN
10068, 10160, 10199, 10619, 10633,
10645, 10667, 10783, 10788, 10842
- _fp_parse_infix_ 10616, 10824, 10838,
10848, 10852, 10854, 10859, 10904,
10905, 10907, 10908, 10944, 10956,
10959, 10968, 10971, 10983, 10994
- _fp_parse_infix:Nw 10953
- _fp_parse_infix(:N 10912
- _fp_parse_infix):N 10836
- _fp_parse_infix*:N 10939
- _fp_parse_infix+:N 10005, 10880
- _fp_parse_infix -:N 10880
- _fp_parse_infix/:N 10880
- _fp_parse_infix::N
10981, 10997, 11010, 11329
- _fp_parse_infix:N 11014
- _fp_parse_infix<:N 11014
- _fp_parse_infix>:N 11014
- _fp_parse_infix?:N 10981
- _fp_parse_infix^:N 10880
- _fp_parse_infix_after_operand:NwN
. 10085, 10127, 10593, 10781, 10781
- _fp_parse_infix_and:N
10880, 10909, 10975
- _fp_parse_infix_check:NNN 10809, 10819
- _fp_parse_infix_comma:w . 10862, 10872
- _fp_parse_infix_comma_gobble:w . . .
10865, 10874
- _fp_parse_infix_end:N
10761, 10795, 10834, 10834, 10835
- _fp_parse_infix_excl_aux:NN
11014, 11031, 11034
- _fp_parse_infix_excl_error:
11014, 11039, 11048
- _fp_parse_infix_juxtapose:N 10798,
10807, 10913, 10914, 10914, 10925

_fp_parse_infix_mul:N	_fp_parse_prefix_
..... 10880, 10906, 10947	10602
_fp_parse_infix_or:N 10880, 10910, 10963	_fp_parse_prefix_(:Nw
_fp_parse_large:N . 10227, 10310, 10310	10599
_fp_parse_large_leading:wwNN	_fp_parse_prefix_+:Nw
..... 10312, 10317, 10317	10566
_fp_parse_large_round:NN	_fp_parse_prefix_-:Nw
..... 10353, 10425, 10425	10572
_fp_parse_large_round_aux:wNN	_fp_parse_prefix_:Nw
..... 10425, 10434, 10454	10572
_fp_parse_large_round_test:NN	_fp_parse_prefix_unknown:NNN
..... 10425, 10438, 10443 10184, 10187, 10192
_fp_parse_large_trailing:wwNN	_fp_parse_return_semicolon:w
..... 10323, 10347, 10347 10006, 10006, 10030, 10180,
_fp_parse_letters:N	10387, 10398, 10480, 10512, 10527
..... 10142, 10153, 10164, 10177	_fp_parse_round:Nw
_fp_parse_lparen_after:NwN 10720, 10723, 10726, 10736
..... 10599, 10604, 10614	_fp_parse_round_after:wN
_fp_parse_one:Nw 10402, 10402, 10407, 10416, 10457
10043, 10043, 10204, 10566, 10769	_fp_parse_round_loop:N 10375,
_fp_parse_one_digit:NN	10375, 10380, 10418, 10436, 10461
..... 10056, 10125, 10125	_fp_parse_round_up:N
_fp_parse_one_fp:NN 10048, 10064, 10064 10375, 10383, 10391, 10395
_fp_parse_one_other:NN	_fp_parse_small:N . 10247, 10258, 10258
..... 10059, 10133, 10133	_fp_parse_small_leading:wwNN
_fp_parse_one_register:NN 10262, 10267, 10267, 10329
..... 10051, 10083, 10083	_fp_parse_small_round:NN
_fp_parse_one_register_aux:Nw 10289, 10407, 10407, 10446
..... 10083, 10088, 10102	_fp_parse_small_trailing:wwNN
_fp_parse_one_register_auxii:wwwNw 10275, 10284, 10284, 10361
..... 10083, 10105, 10112	_fp_parse_strim_end:w
_fp_parse_one_register_dim:ww 10233, 10239, 10243
..... 10083, 10107, 10115, 10119	_fp_parse_strim_zeros:N
_fp_parse_one_register_int:www 10214, 10233, 10233, 10237, 10597
..... 10083, 10109, 10117	_fp_parse_trim_end:w 10207, 10217, 10222
_fp_parse_one_register_mu:www	_fp_parse_trim_zeros:N
..... 10083, 10108, 10114 10131, 10207, 10207, 10210
_fp_parse_operand:Nw	_fp_parse_unary_function:nNN
..... 10005, 10582, 10584, 10670, 10670,
10608, 10610, 10677, 10686, 10734,	10701, 10703, 10705, 10713, 10715
10756, 10763, 10763, 10868, 10895,	_fp_parse_word:Nw . 10139, 10153, 10153
10920, 10989, 11005, 11091, 11339	_fp_parse_word_abs:N ... 10700, 10700
_fp_parse_pack_carry:w	_fp_parse_word_acos:N
..... 10297, 10305, 10308	10706
_fp_parse_pack_leading:NNNNnw ...	_fp_parse_word_acosd:N
..... 10260, 10297, 10302, 10320	10706
_fp_parse_pack_trailing:NNNNnw .	_fp_parse_word_acot:N .. 10688, 10688
..... 10270,	_fp_parse_word_acotd:N .. 10688, 10690
10297, 10297, 10339, 10350, 10357	_fp_parse_word_acsc:N
_fp_parse_prefix:NNN 10145, 10184, 10184	10706
	_fp_parse_word_acscd:N
	10706
	_fp_parse_word_asec:N
	10706
	_fp_parse_word_asecd:N
	10706
	_fp_parse_word_asin:N
	10706
	_fp_parse_word_asind:N
	10706
	_fp_parse_word_atan:N .. 10688, 10692
	_fp_parse_word_atand:N .. 10688, 10694

<code>__fp_parse_word_bp:N</code>	10641	<code>__fp_pow_neg_case:w</code>	13025 , 13046 , 13046
<code>__fp_parse_word_cc:N</code>	10641	<code>__fp_pow_neg_case_aux:NNNNNNNNw</code> ...	
<code>__fp_parse_word_cm:N</code>	10641	13046 , 13061 , 13068 , 13078
<code>__fp_parse_word_cos:N</code>	10706	<code>__fp_pow_neg_case_aux:nnnnn</code>	
<code>__fp_parse_word_cosd:N</code>	10706	13046 , 13050 , 13054
<code>__fp_parse_word_cot:N</code>	10706	<code>__fp_pow_normal:ww</code> ..	12866 , 12894 , 12894
<code>__fp_parse_word_cotd:N</code>	10706	<code>__fp_pow_npos:Nww</code> ..	12905 , 12922 , 12922
<code>__fp_parse_word_csc:N</code>	10706	<code>__fp_pow_npos_aux:NNnww</code>	
<code>__fp_parse_word_cscd:N</code>	10706	12928 , 12932 , 12938 , 12938
<code>__fp_parse_word_dd:N</code>	10641	<code>__fp_pow_zero_or_inf:ww</code>	
<code>__fp_parse_word_deg:N</code>	10630	12868 , 12875 , 12875
<code>__fp_parse_word_em:N</code>	10660	<code>__fp_reverse_args:Nww</code>	
<code>__fp_parse_word_ex:N</code>	10660	9390 , 9390 , 13586 , 13671 , 13783 , 13849
<code>__fp_parse_word_exp:N</code> ...	10700 , 10702	<code>__fp_round:NNN</code>	9833 , 9867 , 9870 , 11538 ,
<code>__fp_parse_word_false:N</code>	10630	11549 , 11790 , 11802 , 11942 , 11953
<code>__fp_parse_word_in:N</code>	10641	<code>__fp_round:Nwn</code>	
<code>__fp_parse_word_inf:N</code>	10630	9922 , 9924 , 9930 , 10742 , 14052 , 14271
<code>__fp_parse_word_ln:N</code> ...	10700 , 10704	<code>__fp_round:Nww</code>	9922 , 9922 , 10743
<code>__fp_parse_word_max:N</code> ...	10688 , 10696	<code>__fp_round_digit:Nw</code>	
<code>__fp_parse_word_min:N</code> ...	10688 , 10698	9554 , 9574 , 9884 , 9884 ,
<code>__fp_parse_word_mm:N</code>	10641	11552 , 11694 , 11793 , 11805 , 11956
<code>__fp_parse_word_nan:N</code>	10630	<code>__fp_round_neg:NNN</code>	
<code>__fp_parse_word_nc:N</code>	10641	9895 , 9921 , 11656 , 11671 , 11689
<code>__fp_parse_word_nd:N</code>	10641	<code>__fp_round_normal:NNwNnn</code>	9922 , 9948 , 9968
<code>__fp_parse_word_pc:N</code>	10641	<code>__fp_round_normal:NnnwNNnn</code>	
<code>__fp_parse_word_pi:N</code>	10630	9922 , 9944 , 9946
<code>__fp_parse_word_pt:N</code>	10641	<code>__fp_round_normal:NwNNnw</code>	9922 , 9933 , 9941
<code>__fp_parse_word_round:N</code> ..	10717 , 10717	<code>__fp_round_normal_end:wwNnn</code>	
<code>__fp_parse_word_sec:N</code>	10706	9922 , 9976 , 9979
<code>__fp_parse_word_secd:N</code>	10706	<code>__fp_round_pack:Nw</code> ...	9922 , 9952 , 9966
<code>__fp_parse_word_sin:N</code>	10706	<code>__fp_round_places:NNn</code>	14262 ,
<code>__fp_parse_word_sind:N</code>	10706	14263 , 14265 , 14266 , 14282 , 14287
<code>__fp_parse_word_sp:N</code>	10641	<code>__fp_round_return_one:</code> ..	9833 , 9839 ,
<code>__fp_parse_word_tan:N</code>	10706	9849 , 9857 , 9861 , 9899 , 9907 , 9915
<code>__fp_parse_word_tand:N</code>	10706	<code>__fp_round_s:NNw</code>	9868 , 9868 , 10411 , 10429
<code>__fp_parse_word_true:N</code>	10630	<code>__fp_round_special:NwNnn</code>	
<code>__fp_parse_zero:</code>	9922 , 9971 , 9984
.....	10229 , 10249 , 10253 , 10253	<code>__fp_round_special_aux:Nw</code>	
<code>__fp_pow_B:wwN</code>	12945 , 12980	9922 , 9990 , 9997
<code>__fp_pow_C_neg:w</code>	12983 , 13000	<code>__fp_round_to_nearest:NNN</code>	9833 , 9854 ,
<code>__fp_pow_C_overflow:w</code>	12988 , 12995 , 13016	9867 , 9920 , 10732 , 10737 , 14052 , 14273
<code>__fp_pow_C_pack:w</code> ..	13002 , 13010 , 13021	<code>__fp_round_to_nearest_neg:NNN</code>	
<code>__fp_pow_C_pos:w</code>	12986 , 13005	9895 , 9920 , 9921
<code>__fp_pow_C_pos_loop:wN</code>		<code>__fp_round_to_ninf:NNN</code>	9833 , 9835 , 10726
.....	13006 , 13007 , 13014	<code>__fp_round_to_ninf_neg:NNN</code> ..	9895 , 9895
<code>__fp_pow_exponent:Nwnnnnnwn</code>		<code>__fp_round_to_pinf:NNN</code>	9833 , 9845 , 10720
.....	12951 , 12954 , 12959	<code>__fp_round_to_pinf_neg:NNN</code> ..	9895 , 9911
<code>__fp_pow_exponent:wnN</code> ...	12943 , 12948	<code>__fp_round_to_zero:NNN</code>	9833 , 9844 , 10723
<code>__fp_pow_neg:www</code> ..	12861 , 13023 , 13023	<code>__fp_round_to_zero_neg:NNN</code> ..	9895 , 9904
<code>__fp_pow_neg_aux:wNN</code>	13023 , 13026 , 13038	<code>__fp_rrot:www</code>	9391 , 9391 , 13716

_fp_sanitize:Nw
 ... [9437](#), [9437](#), [9448](#), [9982](#), [10000](#),
 [11481](#), [11575](#), [11746](#), [11825](#), [12438](#),
 [12673](#), [12924](#), [13548](#), [13592](#), [13728](#)
 _fp_sanitize:wN [9437](#), [9448](#), [10130](#), [10596](#)
 _fp_sanitize_zero:w .. [9437](#), [9444](#), [9449](#)
 _fp_sec_o:w [13138](#), [13138](#)
 _fp_set_sign_o:w [11959](#), [11959](#)
 _fp_sin_o:w [13093](#), [13093](#), [14239](#)
 _fp_sin_series_aux_o:NNwww
 [13500](#), [13504](#), [13515](#)
 _fp_sin_series_o:NNwww ... [13099](#),
 [13114](#), [13129](#), [13144](#), [13500](#), [13500](#)
 _fp_small_int:wTF [9590](#), [9590](#), [9924](#)
 _fp_small_int_normal:NnwTF
 [9590](#), [9594](#), [9599](#)
 _fp_small_int_test:NnnwNTF
 [9590](#), [9607](#), [9615](#)
 _fp_small_int_true:wTF
 [9590](#), [9593](#), [9598](#), [9618](#)
 _fp_sub_back_far_o:NnnwnnnN
 [11584](#), [11630](#), [11630](#)
 _fp_sub_back_near_after:wNNNNw ...
 [11590](#), [11592](#), [11599](#), [11667](#)
 _fp_sub_back_near_o:nnnnnnnnN
 [11580](#), [11590](#), [11590](#)
 _fp_sub_back_near_pack:NNNNNNw ...
 [11590](#), [11594](#), [11597](#), [11669](#)
 _fp_sub_back_not_far_o:wwwNN
 [11644](#), [11664](#), [11664](#)
 _fp_sub_back_quite_far_ii:NN
 [11648](#), [11650](#), [11654](#)
 _fp_sub_back_quite_far_o:wwNN
 [11642](#), [11648](#), [11648](#)
 _fp_sub_back_shift:wnnnn
 [11602](#), [11606](#), [11606](#)
 _fp_sub_back_shift_ii:ww
 [11606](#), [11608](#), [11611](#)
 _fp_sub_back_shift_iii:NNNNNNNNw .
 [11606](#), [11616](#), [11619](#), [11628](#)
 _fp_sub_back_shift_iv:nnnw
 [11606](#), [11623](#), [11629](#)
 _fp_sub_back_very_far_ii_o:nnNwNN
 [11676](#), [11679](#), [11683](#)
 _fp_sub_back_very_far_o:wwwNN ...
 [11643](#), [11676](#), [11676](#)
 _fp_sub_eq_o:Nnnw [11555](#), [11558](#), [11566](#)
 _fp_sub_npos_i_o:Nnnw
 [11560](#), [11569](#), [11573](#), [11573](#)
 _fp_sub_npos_ii_o:Nnnw
 [11555](#), [11562](#), [11567](#)
 _fp_sub_npos_o:Nnnw
 [11475](#), [11555](#), [11555](#)
 _fp_tan_o:w [13153](#), [13153](#), [14241](#)
 _fp_tan_series_aux_o:Nnnw
 [13554](#), [13558](#), [13567](#)
 _fp_tan_series_o:NNwww
 [13160](#), [13175](#), [13554](#), [13554](#)
 _fp_ternary:NwN .. [10987](#), [11327](#), [11327](#)
 _fp_ternary_auxi:NwN
 [11327](#), [11333](#), [11365](#)
 _fp_ternary_auxii:NwN
 [11003](#), [11327](#), [11355](#), [11373](#)
 _fp_ternary_break_point:n
 [11327](#), [11333](#), [11352](#), [11364](#)
 _fp_ternary_loop:Nw
 [11327](#), [11330](#), [11357](#), [11362](#)
 _fp_ternary_loop_break:w
 [11327](#), [11332](#), [11352](#)
 _fp_ternary_map_break:
 [11327](#), [11360](#), [11364](#)
 _fp_tmp:w ... [9548](#), [9558](#), [9559](#), [9560](#),
 [9561](#), [9562](#), [9563](#), [9564](#), [9565](#), [9566](#),
 [9567](#), [9568](#), [9569](#), [9570](#), [9571](#), [9572](#),
 [9573](#), [10023](#), [10035](#), [10036](#), [10037](#),
 [10038](#), [10039](#), [10040](#), [10041](#), [10572](#),
 [10589](#), [10590](#), [10630](#), [10635](#), [10636](#),
 [10637](#), [10638](#), [10639](#), [10640](#), [10641](#),
 [10649](#), [10650](#), [10651](#), [10652](#), [10653](#),
 [10654](#), [10655](#), [10656](#), [10657](#), [10658](#),
 [10659](#), [10886](#), [10904](#), [10905](#), [10906](#),
 [10907](#), [10908](#), [10909](#), [10910](#), [14230](#),
 [14237](#), [14238](#), [14239](#), [14240](#), [14241](#)
 _fp_to_decimal_dispatch:w
 .. [13937](#), [13941](#), [13944](#), [13944](#), [14051](#)
 _fp_to_decimal_huge:wnnnn
 [13944](#), [13982](#), [14004](#)
 _fp_to_decimal_large:Nnnw
 [13944](#), [13978](#), [13995](#)
 _fp_to_decimal_normal:wnnnn
 [13944](#), [13949](#), [13970](#), [14032](#)
 _fp_to_int_dispatch:w
 [14042](#), [14046](#), [14049](#), [14049](#)
 _fp_to_scientific_dispatch:w
 [13882](#), [13886](#), [13889](#), [13894](#)
 _fp_to_scientific_normal:wNw
 [13889](#), [13925](#), [13927](#), [13934](#)
 _fp_to_scientific_normal:wnnnn ..
 .. [13889](#), [13899](#), [13921](#), [14025](#), [14029](#)

```

\__fp_to_tl_dispatch:w .....
.. 14005, 14009, 14012, 14012, 14118
\__fp_to_tl_normal:nnnn .....
..... 14012, 14017, 14022
\__fp_trap_division_by_zero_set:N ..
..... 9721, 9722, 9724, 9726, 9727
\__fp_trap_division_by_zero_set_error:
..... 9721, 9721
\__fp_trap_division_by_zero_set_flag:
..... 9721, 9723
\__fp_trap_division_by_zero_set_none:
..... 9721, 9725
\__fp_trap_invalid_operation_set:N .
..... 9687, 9688, 9690, 9692, 9693
\__fp_trap_invalid_operation_set_error:
..... 9687, 9687
\__fp_trap_invalid_operation_set_flag:
..... 9687, 9689
\__fp_trap_invalid_operation_set_none:
..... 9687, 9691
\__fp_trap_overflow_set:N .....
..... 9747, 9748, 9750, 9752, 9753
\__fp_trap_overflow_set:NnNn .....
..... 9747, 9754, 9762, 9763
\__fp_trap_overflow_set_error: ....
..... 9747, 9747
\__fp_trap_overflow_set_flag: 9747, 9749
\__fp_trap_overflow_set_none: 9747, 9751
\__fp_trap_underflow_set:N .....
..... 9747, 9756, 9758, 9760, 9761
\__fp_trap_underflow_set_error: ....
..... 9747, 9755
\__fp_trap_underflow_set_flag: ....
..... 9747, 9757
\__fp_trap_underflow_set_none: ....
..... 9747, 9759
\__fp_trig:NNNNwn 13099, 13114, 13129,
13144, 13159, 13174, 13191, 13191
\__fp_trig_inverse_two_pi: .....
..... 13252, 13252, 13421
\__fp_trig_large:ww . 13199, 13416, 13416
\__fp_trig_large_auxi:wwwww .....
..... 13416, 13418, 13424
\__fp_trig_large_auxii:ww .....
..... 13416, 13426, 13435
\__fp_trig_large_auxiii:wNNNNNNN ..
..... 13416, 13428, 13436
\__fp_trig_large_auxiv:wN .....
..... 13416, 13430, 13438
\__fp_trig_large_auxix:Nw .....
..... 13459, 13469, 13472, 13476
\__fp_trig_large_auxv:www .....
..... 13432, 13439, 13439
\__fp_trig_large_auxvi:wNNNNNNN ...
..... 13439, 13445, 13450
\__fp_trig_large_auxvii:w .....
..... 13442, 13459, 13459
\__fp_trig_large_auxviii:w ..... 13459
\__fp_trig_large_auxviii:ww 13461, 13465
\__fp_trig_large_auxx:wNNNNN .....
..... 13459, 13482, 13486
\__fp_trig_large_auxxi:w .....
..... 13459, 13479, 13493
\__fp_trig_large_pack:NNNNw .....
..... 13439, 13452, 13457, 13488
\__fp_trig_small:ww .....
.. 13201, 13205, 13205, 13211, 13498
\__fp_trigd_large:ww 13199, 13213, 13213
\__fp_trigd_large_auxi:nnnwNNNN ...
..... 13213, 13219, 13225
\__fp_trigd_large_auxii:wNw .....
..... 13213, 13227, 13233
\__fp_trigd_large_auxiii:www .....
..... 13213, 13236, 13240
\__fp_trigd_small:ww .....
..... 13201, 13207, 13207, 13250
\__fp_trim_zeros:w .....
.. 13873, 13873, 13935, 13988, 13997
\__fp_trim_zeros_dot:w 13873, 13876, 13879
\__fp_trim_zeros_end:w 13873, 13879, 13880
\__fp_trim_zeros_loop:w .....
..... 13873, 13875, 13876, 13878
\__fp_type_from_scan:N .....
..... 9505, 10008, 10016, 10066
\__fp_type_from_scan:w 10008, 10018, 10021
\__fp_underflow:w ..... 9443, 9778, 9784
\__fp_use_braced_s:n ..... 9385, 12011
\__fp_use_i:ww .. 9392, 9392, 12198, 13802
\__fp_use_i_until_s:nw .. 9387, 9388,
9431, 9638, 13243, 13441, 13447, 13478
\__fp_use_ii_until_s:nnw 9387, 9389, 9429
\__fp_use_none_stop_f:n .....
..... 9383, 9383, 12362, 12363, 12364
\__fp_use_none_until_s:w .....
..... 9387, 9387, 13032, 13797, 13800
\__fp_use_s:n ..... 9384, 9384
\__fp_use_s:nn ..... 9384, 9386
\__fp_zero_fp:N .. 9412, 9412, 9762, 9988
\__int_abs:N ..... 3167, 3169, 3173

```

<code>__int_case:nnTF</code>	12406, 12408, 12446, 12457, 12476,
.. 3389 , 3392 , 3397 , 3402 , 3407 , 3409	12478, 12480, 12492, 12505, 12510,
<code>__int_case:nw</code> ... 3389 , 3410 , 3411 , 3415	12512, 12518, 12535, 12536, 12537,
<code>__int_case_end:nw</code>	12538, 12539, 12540, 12545, 12547,
<code>__int_compare:NNw</code>	12549, 12551, 12553, 12557, 12559,
<code>__int_compare:Nw</code> 3333 , 3341 , 3343 , 3365	12561, 12563, 12565, 12567, 12589,
<code>__int_compare:nnN</code> .. 3333 , 3360 , 3368 ,	12597, 12675, 12679, 12732, 12941,
3370 , 3372 , 3374 , 3376 , 3378 , 3380	12962, 12964, 12967, 12970, 12973,
<code>__int_compare:w</code>	12976, 12992, 13018, 13028, 13044,
<code>__int_compare_<:NNw</code>	13196, 13228, 13237, 13419, 13420,
<code>__int_compare_=:NNw</code>	13443, 13453, 13462, 13480, 13489,
<code>__int_compare_>:NNw</code>	13496, 13507, 13517, 13550, 13560,
<code>__int_constdef:Nw</code> 3239 , 3250 , 3262 , 3266	13585, 13594, 13611, 13647, 13664,
<code>__int_div_truncate:NwNw</code>	13666, 13678, 13679, 13719, 13730,
..... 3199 , 3202 , 3207 , 3230	13741, 13799, 13929, 14067, 14275
<code>__int_eval:w</code> ... 75 , 1284 , 2256 , 2478 ,	<code>__int_eval_end:</code> 75 , 1284 , 2256 , 2478 ,
2480 , 2482 , 2548 , 2550 , 2552 , 2554 ,	2480 , 2482 , 2548 , 2550 , 2552 , 2554 ,
2556 , 2558 , 2560 , 2562 , 2564 , 2566 ,	2556 , 2558 , 2560 , 2562 , 2564 , 2566 ,
2568 , 2570 , 3156 , 3157 , 3162 , 3165 ,	2568 , 2570 , 3156 , 3158 , 3162 , 3165 ,
3170 , 3178 , 3179 , 3186 , 3187 , 3201 ,	3170 , 3205 , 3221 , 3227 , 3251 , 3288 ,
3203 , 3204 , 3221 , 3224 , 3225 , 3226 ,	3290 , 3312 , 3383 , 3420 , 3428 , 3654 ,
3251 , 3288 , 3290 , 3312 , 3336 , 3365 ,	3681 , 9338 , 9436 , 9629 , 9963 , 9967 ,
3383 , 3420 , 3428 , 3654 , 3681 , 3836 ,	10741 , 11061 , 11150 , 11405 , 11570 ,
3880 , 9335 , 9436 , 9536 , 9539 , 9626 ,	11692 , 11727 , 11915 , 12039 , 12175 ,
9873 , 9877 , 9889 , 9890 , 9949 , 9953 ,	12992 , 13044 , 13229 , 13238 , 13507 ,
9992 , 10131 , 10136 , 10171 , 10261 ,	13517 , 13560 , 13585 , 13611 , 13679
10272 , 10321 , 10352 , 10358 , 10359 ,	<code>__int_from_alph:N</code>
10405 , 10415 , 10417 , 10433 , 10435 ,	<code>__int_from_alph:n</code>
10458 , 10460 , 10597 , 10741 , 10802 ,	<code>__int_from_alph:nn</code> 3760 , 3769 , 3770 , 3775
11061 , 11150 , 11483 , 11491 , 11512 ,	<code>__int_from_base:N</code>
11514 , 11535 , 11537 , 11546 , 11548 ,	<code>__int_from_base:nn</code>
11570 , 11577 , 11583 , 11593 , 11595 ,	<code>__int_from_base:nnN</code> 3781 , 3791 , 3792 , 3797
11668 , 11670 , 11686 , 11688 , 11692 ,	<code>__int_from_roman:NN</code>
11708 , 11748 , 11756 , 11758 , 11760 , 3831 , 3837 , 3840 , 3865 , 3869
11762 , 11764 , 11766 , 11768 , 11787 ,	<code>__int_from_roman_clean_up:w</code>
11789 , 11799 , 11801 , 11827 , 11830 , 3831 , 3848 , 3855 , 3857 , 3876
11838 , 11840 , 11860 , 11863 , 11866 ,	<code>__int_from_roman_end:w</code> 3831 , 3835 , 3874
11869 , 11877 , 11880 , 11883 , 11886 ,	<code>__int_get_digits:n</code> 3726 , 3731 , 3765 , 3787
11893 , 11895 , 11901 , 11909 , 11911 ,	<code>__int_get_sign:n</code> 3726 , 3726 , 3764 , 3785
11913 , 11919 , 11939 , 11941 , 11950 ,	<code>__int_get_sign_and_digits:nnnn</code>
11952 , 11978 , 11983 , 11985 , 11992 , 3726 , 3728 , 3733 , 3736 , 3759
11995 , 11998 , 12001 , 12004 , 12007 ,	<code>__int_get_sign_and_digits:nnnn</code>
12017 , 12029 , 12037 , 12039 , 12049 , 3726 , 3742 , 3746 , 3752
12051 , 12058 , 12067 , 12069 , 12072 ,	<code>__int_maxmin:wwN</code> 3167 , 3177 , 3185 , 3191
12075 , 12078 , 12081 , 12095 , 12097 ,	<code>__int_mod:ww</code>
12105 , 12107 , 12115 , 12117 , 12126 ,	<code>__int_step:NNnnnn</code> 3513 , 3516 , 3523 , 3532
12129 , 12132 , 12139 , 12154 , 12172 ,	<code>__int_step:NnnnN</code>
12175 , 12231 , 12245 , 12247 , 12253 , 3490 , 3493 , 3500 , 3504 , 3509
12266 , 12268 , 12270 , 12294 , 12310 ,	<code>__int_to_Roman_Q:w</code>
12317 , 12318 , 12341 , 12357 , 12361 ,	<code>__int_to_Roman_aux:N</code> .. 3702 , 3705 , 3708

_int_to_Roman_c:w 3690, 3722
 _int_to_Roman_d:w 3690, 3723
 _int_to_Roman_i:w 3690, 3718
 _int_to_Roman_l:w 3690, 3721
 _int_to_Roman_m:w 3690, 3724
 _int_to_Roman_v:w 3690, 3719
 _int_to_Roman_x:w 3690, 3720
 _int_to_base:nn 3623, 3624, 3625
 _int_to_base:nnN
 3623, 3628, 3629, 3631, 3645
 _int_to_base:nnnN 3623, 3636, 3643
 _int_to_letter:n 3623, 3634, 3637, 3651
 _int_to_roman:N
 3690, 3690, 3692, 3695, 3698
 _int_to_roman:w . 74, 777, 778, 878,
 880, 1053, 2254, 3156, 3346, 3693, 3703
 _int_to_roman_Q:w 3690, 3717
 _int_to_roman_c:w 3690, 3714
 _int_to_roman_d:w 3690, 3715
 _int_to_roman_i:w 3690, 3710
 _int_to_roman_l:w 3690, 3713
 _int_to_roman_m:w 3690, 3716
 _int_to_roman_v:w 3690, 3711
 _int_to_roman_x:w 3690, 3712
 _int_to_symbols:nnnn . 3539, 3543, 3553
 _int_value:w 75,
 1059, 2168, 2171, 2256, 3156, 3156,
 3162, 3165, 3169, 3177, 3185, 3221,
 3224, 3225, 3226, 3340, 3365, 3681,
 3836, 3997, 6487, 6518, 6519, 6520,
 6522, 6648, 6657, 6659, 6664, 6665,
 6666, 6667, 6671, 6672, 6673, 6674,
 6699, 6705, 6707, 6709, 6711, 6716,
 6721, 6726, 6733, 6740, 6870, 6900,
 6901, 6940, 6959, 6965, 7135, 7239,
 9335, 9494, 9495, 9496, 9497, 9498,
 9553, 9619, 9935, 9951, 9956, 10090,
 10122, 10123, 10256, 10263, 10276,
 10313, 10324, 10330, 10341, 10362,
 10472, 10549, 10551, 10666, 11165,
 11428, 11429, 11430, 11432, 11486,
 11489, 11552, 11609, 11617, 11625,
 11691, 11694, 11793, 11805, 11843,
 11844, 11845, 11846, 11956, 11965,
 12363, 12364, 12365, 12439, 12453,
 12474, 12496, 12497, 12498, 12499,
 12500, 12613, 12618, 12622, 12675,
 12744, 12926, 12983, 12986, 12988,
 13014, 13016, 13421, 13506, 13559,
 13997, 14059, 14069, 14071, 14928,
 14930, 14953, 14955, 14980, 15020,
 15047, 15055, 15080, 15082, 15086,
 15088, 15117, 15131, 15138, 15338
 _ior_list_streams:Nn
 9059, 9060, 9061, 9154
 _ior_map_inline:NNNn 15149, 15156, 15159
 _ior_map_inline:NNn
 15149, 15150, 15152, 15153
 _ior_map_inline_loop:NNN
 15149, 15162, 15166, 15172
 _ior_open:Nn
 ... 169, 8875, 8895, 9024, 9024, 9040
 _ior_open:No 9007, 9020, 9024
 _ior_open_aux:Nn 8999, 9000, 9002
 _ior_open_aux:NnTF ... 9009, 9010, 9014
 _ior_open_stream:Nn
 9024, 9028, 9036, 9041
 _iow_indent:n 9204, 9205, 9223
 _iow_list_streams:Nn . 9152, 9153, 9154
 _iow_open:Nn ... 9114, 9115, 9117, 9133
 _iow_open_stream:Nn
 9114, 9121, 9129, 9134
 _iow_wrap_end: 9325
 _iow_wrap_end:w 9298
 _iow_wrap_indent: 9313
 _iow_wrap_indent:w 9298
 _iow_wrap_loop:w
 9244, 9253, 9253, 9268, 9303
 _iow_wrap_newline: 9305
 _iow_wrap_newline:w 9298
 _iow_wrap_special:w
 9257, 9298, 9298, 9302
 _iow_wrap_unindent: 9319
 _iow_wrap_unindent:w 9298
 _iow_wrap_word: 9258, 9260, 9260
 _iow_wrap_word_fits: . 9260, 9266, 9270
 _iow_wrap_word_newline: 9260, 9267, 9286
 _kernel_register_show:N .. 25, 1432,
 1432, 1442, 3877, 4145, 4235, 4297
 _kernel_register_show:c 1432, 1441, 3878
 _keys_bool_set:Nn
 8247, 8247, 8262, 8381, 8385
 _keys_bool_set:cn 8247, 8383, 8387
 _keys_bool_set_inverse:Nn
 8263, 8263, 8278, 8389, 8393
 _keys_bool_set_inverse:cn
 8263, 8391, 8395
 _keys_check_groups: 8598, 8611
 _keys_choice_code_store:n
 8763, 8763, 8774, 8776

```

\__keys_choice_code_store:x . 8763, 8778
\__keys_choice_find:n .....
    ..... 8282, 8337, 8689, 8689
\__keys_choice_make: ..... 8250,
    8266, 8279, 8279, 8291, 8397, 8784
\__keys_choices_generate:n .....
    ..... 8763, 8779, 8813
\__keys_choices_generate_aux:n ....
    ..... 8763, 8786, 8793
\__keys_choices_make:nn .....
    .. 8289, 8289, 8399, 8401, 8403, 8405
\__keys_cmd_set:Vn ..... 8305, 8335
\__keys_cmd_set:Vo ..... 8305, 8331
\__keys_cmd_set:nn .....
    ..... 8255, 8271, 8281, 8283,
    8305, 8305, 8310, 8340, 8342, 8407
\__keys_cmd_set:nx .. 8251, 8253, 8267,
    8269, 8296, 8305, 8355, 8376, 8796
\__keys_default_set:n ... 8260, 8276,
    8311, 8311, 8417, 8419, 8421, 8423
\__keys_define:nnn ..... 8178, 8180, 8186
\__keys_define:onn ..... 8178, 8179
\__keys_define_elt:n ... 8183, 8187, 8187
\__keys_define_elt:nn .. 8183, 8187, 8192
\__keys_define_elt_aux:nn .....
    ..... 8187, 8190, 8195, 8197
\__keys_define_key:n ... 8201, 8230, 8230
\__keys_define_key:w ... 8230, 8234, 8245
\__keys_execute: ..... 8589, 8657, 8657
\__keys_execute:nn .....
    .. 8657, 8658, 8664, 8671, 8691, 8692
\__keys_execute_unknown: 8657, 8658, 8659
\__keys_groups_set:n ... 8316, 8316, 8441
\__keys_if_value:n ..... 8647
\__keys_if_value_p:n ... 8572, 8582, 8647
\__keys_initialise:n .....
    .. 8325, 8325, 8443, 8445, 8447, 8449
\__keys_initialise:wn .. 8325, 8326, 8327
\__keys_meta_make:n .... 8329, 8329, 8459
\__keys_meta_make:nn ... 8329, 8334, 8461
\__keys_multichoice_find:n .....
    ..... 8336, 8336, 8341
\__keys_multichoice_make: .....
    ..... 8336, 8338, 8350, 8463
\__keys_multichoices_make:nn .....
    .. 8336, 8348, 8465, 8467, 8469, 8471
\__keys_property_find:n 8199, 8210, 8210
\__keys_property_find:w .....
    ..... 8210, 8214, 8220, 8226
\__keys_set:nnn ..... 8500, 8502, 8509
\__keys_set:onn ..... 8500, 8501
\__keys_set_elt:n ..... 8505, 8549, 8549
\__keys_set_elt:nn ..... 8505, 8549, 8554
\__keys_set_elt_aux: ..... 8549,
    8566, 8568, 8601, 8607, 8629, 8633
\__keys_set_elt_aux:nn .....
    ..... 8549, 8552, 8557, 8559
\__keys_set_elt_selective: .....
    ..... 8549, 8565, 8592
\__keys_store_unused: ..... 8602,
    8608, 8628, 8634, 8657, 8662, 8680
\__keys_value_or_default:n .....
    ..... 8563, 8637, 8637
\__keys_value_requirement:n .....
    ..... 8364, 8364, 8497, 8499
\__keys_variable_set:NnnN .....
    ... 8373, 8373, 8379, 8409, 8413,
    8425, 8429, 8433, 8437, 8451, 8455,
    8473, 8477, 8481, 8485, 8489, 8493
\__keys_variable_set:cnnN .....
    ..... 8373, 8411, 8415,
    8427, 8431, 8435, 8439, 8453, 8457,
    8475, 8479, 8483, 8487, 8491, 8495
\__keyval_parse:n ..... 8056, 8064, 8150
\__keyval_parse_elt:w .....
    ..... 8072, 8078, 8078, 8081, 8085
\__keyval_split_key:n .. 8090, 8108, 8108
\__keyval_split_key:w .. 8108, 8112, 8114
\__keyval_split_key_value:w .....
    ..... 8084, 8088, 8088
\__keyval_split_key_value:WTF .....
    ..... 8088, 8101, 8106
\__keyval_split_value:w 8102, 8116, 8116
\__keyval_split_value_aux:w . 8134, 8137
\__msg_class_chk_exist:nT .....
    .. 7593, 7593, 7607, 7673, 7683, 7688
\__msg_class_new:nn . 7488, 7489, 7526,
    7537, 7548, 7569, 7577, 7585, 7591
\__msg_error:cnnnnn .... 7548, 7550, 7562
\__msg_error_code:nnnnnn ..... 7774
\__msg_expandable_error:n .....
    ..... 149, 7941, 7949, 7966
\__msg_expandable_error:w 7941, 7955, 7961
\__msg_fatal_code:nnnnnn ..... 7770
\__msg_interrupt_more_text:n .....
    ..... 7416, 7418, 7421
\__msg_interrupt_text:n 7419, 7430, 7438
\__msg_interrupt_wrap:nn .....
    ..... 7408, 7412, 7416, 7416

```

```

\__msg_kernel_class_new:nN ..... 7775
.. 7732, 7733, 7770, 7774, 7775, 7776 \__msg_kernel_info:nnnnnn .... 148, 7775
\_msg_kernel_class_new_aux:nN ..... \__msg_kernel_info:nnx ..... 7775
..... 7732, 7734, 7735 \__msg_kernel_info:nnxx ..... 7775
\_msg_kernel_error:nn ..... \__msg_kernel_info:nnxxx ..... 7775
.. 1159, 1173, 6758, 7770, 7773, 8103 \__msg_kernel_info:nnxxxx ..... 7775
\_msg_kernel_error:nnn ..... 7770 \__msg_kernel_new:nnn ..... 7263,
\_msg_kernel_error:nnnn ..... 7770 7724, 7726, 7903, 7905, 7907, 7909,
\_msg_kernel_error:nnnnn ..... 7770 7911, 7913, 7915, 7923, 7930, 7937,
\_msg_kernel_error:nnnnnn ... 148, 7770 7939, 9818, 9820, 9822, 9824, 9826,
\_msg_kernel_error:nnnx 922, 963, 1001, 9828, 11109, 11111, 11113, 11115,
1006, 1159, 1171, 1204, 1355, 1437, 11117, 11119, 11121, 11123, 11125
1893, 2108, 2453, 4513, 5043, 6356, \__msg_kernel_new:nnnn .....
6502, 7596, 7770, 7772, 8014, 8216, 147, 7245, 7253, 7256, 7724, 7724,
8257, 8273, 8576, 8789, 8861, 8920, 7778, 7786, 7794, 7801, 7812, 7820,
9006, 9396, 9684, 14166, 14345, 14387 7829, 7836, 7843, 7850, 7859, 7866,
\_msg_kernel_error:nnxx ... 913, 942, 7873, 7880, 7887, 7895, 8153, 8708,
1020, 1159, 1159, 1172, 1174, 1181, 8711, 8717, 8722, 8731, 8737, 8744,
1191, 1325, 1954, 6650, 7323, 7333, 8751, 8757, 8805, 9349, 9355, 9362,
7621, 7770, 7771, 8205, 8238, 8285, 9369, 9646, 9795, 9807, 14455, 14461
8344, 8586, 8666, 9681, 14382, 14410 \__msg_kernel_set:nnn ..... 7724, 7730
\_msg_kernel_error:nnxxx ..... 7770 \__msg_kernel_set:nnnn ... 147, 7724, 7728
\_msg_kernel_error:nnxxxx ... 1969, 7770 \__msg_kernel_warning:nn ..... 7775
\_msg_kernel_expandable_error:nn .. \__msg_kernel_warning:nnn ..... 7775
..... 2284, \__msg_kernel_warning:nnnn ..... 7775
5066, 5961, 7964, 7988, 10075, 10876 \__msg_kernel_warning:nnnnn ..... 7775
\_msg_kernel_expandable_error:nnn . \__msg_kernel_warning:nnnnnn . 148, 7775
1648, 3329, 3497, 4737, 5478, 5924, \__msg_kernel_warning:nnx ..... 7775
7964, 7983, 10080, 10157, 10196, \__msg_kernel_warning:nnxx ..... 7775
10202, 10538, 10543, 10554, 10561, \__msg_kernel_warning:nnxxx ..... 7775
10624, 10822, 10841, 14307, 14312 \__msg_kernel_warning:nnxxxx 7708, 7775
\_msg_kernel_expandable_error:nnnn \__msg_no_more_text:nnnn 7548, 7564, 7568
.... 7964, 7978, 11000, 11041, 11342 \__msg_redirect:nnn 7677, 7678, 7680, 7681
\_msg_kernel_expandable_error:nnnnn \__msg_redirect_loop_chk:nnn .....
..... 7964, 7973, 9793, 10745, 13616 ..... 7677, 7693, 7698, 7722
\_msg_kernel_expandable_error:nnnnnn \__msg_redirect_loop_chk:onnn .... 7718
148, 7964, 7964, 7975, 7980, 7985, 7990 \__msg_redirect_loop_list:n .....
\_msg_kernel_fatal:nn ..... \__msg_redirect_loop_list:nn .....
..... 7770, 9030, 9123, 14366 ..... 7677, 7714, 7723
\_msg_kernel_fatal:nnn ..... \__msg_show_item:n .....
..... 7770 ..... 149, 5533, 5939, 8029, 8029
\_msg_kernel_fatal:nnnn ..... \__msg_show_item:nn .... 6222, 8029, 8033
7770 \__msg_show_item_unbraced:nn .....
\_msg_kernel_fatal:nnnnnn ... 148, 7770 ..... 149, 7240, 8029, 8038, 9067
\_msg_kernel_fatal:nnx ... 7770, 14339 \__msg_show_variable:Nnn .....
\_msg_kernel_fatal:nnxx ..... 7770 149, 5532, 5938, 6221, 7236, 8006, 8006
\_msg_kernel_fatal:nnxxx ..... 7770 \__msg_show_variable:n .....
\_msg_kernel_fatal:nnxxxx ..... 7770 ..... 149, 1451, 2115, 2116, 5049,
\_msg_kernel_info:nn ..... 7775 8006, 8011, 8018, 9066, 14164, 14171
\_msg_kernel_info:nnn ..... 7775 \__msg_show_variable_aux:n .....
\_msg_kernel_info:nnnn ..... 7775 ..... 8006, 8019, 8020

```

_msg_show_variable_aux:w	_peek_token_generic:NNTF
..... 8006, 8024, 8028 2961, 2961, 2979, 2981, 15529
_msg_term:nn	_peek_token_remove_generic:NNTF ..
..... 7993, 8004 2982, 2982, 2997, 2999
_msg_term:nnn .. 7993, 8002, 8010, 9063	_peek_true:w 2947, 2947, 2965, 2986,
_msg_term:nnnn .. 7993, 8000	3003, 3026, 3036, 15512, 15525, 15526
_msg_term:nnnnV	_peek_true_aux:w 2947, 2948, 2958, 2987
..... 7993	_peek_true_remove:w .. 2955, 2955, 2986
_msg_term:nnnnnn	_prg_break:
..... 149, 7993, 7993, 7999, 8001, 8003, 8005 43, 1584, 1585, 2344, 6159, 9627,
_msg_use:nnnnnn	14105, 15226, 15242, 15248, 15464
..... 7497, 7603, 7603	_prg_break:n
_msg_use_code: 1586, 2344, 5261, 15208, 15233, 15466
..... 7603, 7610, 7623, 7627, 7652, 7663	_prg_break_point: .. 43, 1584, 1584,
_msg_use_hierarchy:nwN	1585, 1586, 2344, 5258, 6144, 9628,
..... 7603, 7630, 7631, 7637	14106, 15203, 15227, 15243, 15460
_msg_use_redirect_module:n	_prg_break_point:Nn
..... 7603, 7634, 7642, 7655 43, 1575,
_msg_use_redirect_name:n	1575, 1576, 2344, 3536, 4689, 4706,
..... 7603, 7618, 7624	4715, 5397, 5432, 5443, 5810, 5824,
_my_map_dbl:nn	5843, 5861, 6188, 6207, 15163, 15188
..... 4, 6, 11	_prg_case_end:nw
_my_map_dbl_fn:nn 25, 1516, 1572, 1574, 3417, 4067, 4684
..... 3, 10	_prg_compare_error:
_peek_N_type:w ... 15492, 15507, 15517 3318, 3322, 3336, 3338, 4006, 4008
_peek_N_type_aux:nnw 15492, 15509, 15521	_prg_compare_error:NNw
_peek_def:nnnn 3324, 3353
..... 3051, 3052,	_prg_generate_F_form:wnnnnnn 951, 972
3068, 3072, 3076, 3080, 3084, 3088,	_prg_generate_TF_form:wnnnnnn 951, 977
3092, 3096, 3100, 3104, 3108, 3112	_prg_generate_T_form:wnnnnnn 951, 967
_peek_def:nnnn	_prg_generate_conditional:nnNnnnnn
..... 3051, 3054, 3055, 3056, 3058 891, 910, 919, 919
_peek_execute_branches: ... 3048, 3063	_prg_generate_conditional:nnnnnnw
_peek_execute_branches_N_type: 919, 928, 934, 949
..... 15492, 15500, 15529, 15531, 15533	_prg_generate_conditional_count:nnNnn
_peek_execute_branches_catcode: 894, 895, 897, 899, 901, 902
..... 3008, 3008, 3071, 3073, 3079, 3081	_prg_generate_conditional_count:nnNnnnn
_peek_execute_branches_catcode_aux: 894, 904, 907
..... 3008, 3009, 3011, 3012	_prg_generate_conditional_parm:nnNpnn
_peek_execute_branches_catcode_auxii:N 881, 882, 884, 886, 888, 889
..... 3008, 3016, 3022	_prg_generate_p_form:wnnnnnn 951, 951
_peek_execute_branches_catcode_auxiii:	_prg_map_break:Nn .. 43, 1575, 1576,
..... 3008, 3019, 3032	1582, 2344, 4728, 4730, 5389, 5391,
_peek_execute_branches_charcode: ..	5878, 5880, 6216, 6218, 15146, 15148
..... 3008, 3010, 3087, 3089, 3095, 3097	_prg_replicate:N 2252, 2259, 2260, 2262
_peek_execute_branches_meaning: ..	_prg_replicate_
..... 3000, 3000, 3103, 3105, 3111, 3113 2252, 2263
_peek_false:w 2947, 2949, 2970,	_prg_replicate_0:n
2988, 3005, 3028, 3038, 15514, 15526 2252
_peek_get_prefix_arg_replacement:wN	_prg_replicate_1:n
..... 3117, 3118, 3125, 3134, 3143 2252
_peek_ignore_spaces_execute_branches:	
..... 3041, 3041, 3045,	
3075, 3083, 3091, 3099, 3107, 3115	
_peek_tmp:w	
..... 2947, 2950, 2959	
_peek_token_generic:NNTF .. 2980, 15533	

_prg_replicate_2:n	2252	_prop_get_Nn:nwnn	15199, 15201, 15205, 15209
_prg_replicate_3:n	2252	_prop_if_in:N	6139, 6149, 6152
_prg_replicate_4:n	2252	_prop_if_in:nwnn	6139, 6141, 6146, 6150
_prg_replicate_5:n	2252	_prop_map_function:Nwnn	6184, 6186, 6190, 6196
_prg_replicate_6:n	2252	_prop_map_tokens:nwnn	15184, 15186, 15190, 15196
_prg_replicate_7:n	2252	_prop_pair:wn	127, 5960, 5960, 6000, 6003, 6084,
_prg_replicate_8:n	2252		6113, 6142, 6146, 6187, 6190, 6203,
_prg_replicate_9:n	2252		6205, 6210, 15187, 15190, 15202, 15205
_prg_replicate_first:N	2252, 2255, 2261	_prop_put:Nnnn	6078, 6078, 6079, 6080
_prg_replicate_first_:n	2252	_prop_put_if_new:Nnnn	6105, 6106, 6108, 6109
_prg_replicate_first_0:n	2252	_prop_split:NnTF	127,
_prg_replicate_first_1:n	2252		5995, 5995, 6008, 6014, 6024, 6032,
_prg_replicate_first_2:n	2252		6041, 6054, 6064, 6087, 6116, 6171
_prg_replicate_first_3:n	2252	_prop_split_aux:NnTF	5995, 5996, 5997
_prg_replicate_first_4:n	2252	_prop_split_aux:w	5995, 5999, 6002, 6005
_prg_replicate_first_5:n	2252	_prop_strip_end:w	5994, 5994, 6092, 6121
_prg_replicate_first_6:n	2252	_quark_if_recursion_tail_break:NN	48, 2390, 2390, 2466, 4722
_prg_replicate_first_7:n	2252	_quark_if_recursion_tail_break:nN	2390,
_prg_replicate_first_8:n	2252		2396, 2468, 4695, 5815, 5828, 15464
_prg_replicate_first_9:n	2252	_scan_new:N	48, 2449, 2449,
_prg_set_eq_conditional:Nnnn	982, 983, 985, 986		2461, 2463, 2464, 5959, 9393, 9399,
_prg_set_eq_conditional:nnNnnNnw	990, 998, 998		9400, 9401, 9402, 9403, 9404, 9405
_prg_set_eq_conditional_F_form:nnn	998	_seq_concat:NN	5135, 5136, 5138, 5139
_prg_set_eq_conditional_F_form:wNnnnn	1044	_seq_concat:w	5135, 5139, 5140
_prg_set_eq_conditional_TF_form:nnn	998	_seq_count:n	5447, 5452, 5455
_prg_set_eq_conditional_TF_form:wNnnnn	1034	_seq_get_left:wnw	5286, 5290, 5294
_prg_set_eq_conditional_T_form:nnn	998	_seq_get_right_loop:nn	5311, 5315, 5324, 5327
_prg_set_eq_conditional_T_form:wNnnnn	1039	_seq_get_right_setup:wN	5311, 5312, 5313
_prg_set_eq_conditional_loop:nnnnNw	998, 1010, 1012, 1027	_seq_if_in:	5243, 5252, 5260
_prg_set_eq_conditional_p_form:nnn	998	_seq_item:n	112,
_prg_set_eq_conditional_p_form:wNnnnn	1029		5064, 5064, 5149, 5157, 5171, 5179,
_prg_variable_get_scope:N	42, 2310, 2316		5187, 5248, 5291, 5294, 5304, 5336,
_prg_variable_get_scope:w	2310, 2319, 2322		5337, 5347, 5409, 5414, 5420, 5424,
_prg_variable_get_type:N	42, 2310, 2331		5467, 5472, 5482, 5487, 5490, 5493,
_prg_variable_get_type:w	2310, 2333, 2336, 2340		15293, 15294, 15296, 15303, 15323
		_seq_item:nnn	15213, 15216, 15229, 15234
		_seq_item:wNn	15213, 15213, 15214
		_seq_map_function:Nnn	5392, 5396, 5399, 5403
		_seq_map_function:wN	5392, 5393, 5394
		_seq_mapthread_function:Nnnwnn	15237, 15247, 15251, 15256


```

\__seq_mapthread_function:wNN . . . . . 15237, 15238, 15239
\__seq_mapthread_function:wNw . . . . . 15237, 15241, 15245
\__seq_pop:NNNN . . . . . 5268, 5268, 5298, 5300, 5331, 5333
\__seq_pop_TF:NNNN . . . . . 5268, 5276, 5359, 5361, 5369, 5371, 5373, 5375
\__seq_pop_item_def: . . . . . 112, 5227, 5406, 5422, 5432, 5443, 15315, 15325
\__seq_pop_left:NNN . . . . . 5297, 5298, 5300, 5301, 5369, 5371
\__seq_pop_left:wNwNNN . 5297, 5302, 5303
\__seq_pop_right_aux:NNN . . . . . 5330, 5331, 5333, 5334, 5373, 5375
\__seq_pop_right_loop:nn . . . . . 5330, 5350, 5351, 5354
\__seq_pop_right_setup:w 5330, 5340, 5349
\__seq_push_item_def: . . . . . 5406, 5408, 5413, 5416
\__seq_push_item_def:n . . . . . 112, 5211, 5406, 5406, 5430, 15313, 15323
\__seq_push_item_def:x . 5406, 5411, 5437
\__seq_put_left_aux:w . . . . . 5145, 5150, 5158, 5161
\__seq_put_right_aux:w . . . . . 5166, 5170, 5178, 5182
\__seq_remove_all_aux:NNn . . . . . 5205, 5206, 5208, 5209
\__seq_remove_duplicates:NN . . . . . 5189, 5190, 5192, 5193
\__seq_reverse:NN . . . . . 15286, 15288, 15290, 15291
\__seq_reverse_item:nwn . . . . . 15286, 15294, 15300
\__seq_reverse_setup:w 15286, 15295, 15298
\__seq_set_filter:NNNn . . . . . 15307, 15308, 15310, 15311
\__seq_set_map:NNNn . . . . . 15317, 15318, 15320, 15321
\__seq_set_split:NNnn . . . . . 5098, 5099, 5101, 5102
\__seq_set_split_auxi:w . . . . . 5098, 5112, 5119, 5125
\__seq_set_split_auxii:w 5098, 5127, 5131
\__seq_set_split_end: . . . . . 5098, 5114, 5118, 5125, 5129, 5131
\__seq_tmp:w . . 15286, 15286, 15293, 15296
\__seq_use:nwnn . . . . . 5457, 5474, 5493
\__seq_use:nwwwnwn 5457, 5473, 5485, 5486
\__seq_use:wNwnn . 5457, 5466, 5469, 5481
\__seq_use_setup:w . . . . . 5457, 5472, 5484
\__seq_wrap_item:n . . . . . 5107, 5132, 5187, 5187, 5223, 15263, 15268, 15273, 15278, 15313
\__skip_if_finite:wNwN . 4212, 4216, 4220
\__str_case:nnTF . . . . . 1516, 1519, 1524, 1529, 1534, 1536
\__str_case:nw . . . 1516, 1537, 1538, 1542
\__str_case_end:nw 1516, 1541, 1569, 1574
\__str_case_x:nnTF . . . . . 1516, 1547, 1552, 1557, 1562, 1564
\__str_case_x:nw . 1516, 1565, 1566, 1570
\__str_count_ignore_spaces:N . . . . . 9263, 9331, 9331
\__str_count_ignore_spaces:n . . . . . 9331, 9332, 9333
\__str_count_loop:NNNNNNNN . . . . . 9331, 9336, 9340, 9346
\__str_head:w . . . . . 4917, 4919, 4923
\__str_if_eq_x_return:nn . . . . . 26, 1508, 1508, 2737, 2746, 2762, 2784, 2804, 2822, 2840, 2853, 2864, 2874, 4651, 5007, 15357, 15358
\__str_tail:w . . . . . 4917, 4927, 4931
\__tl_act:NNNnn . . . . . 4815, 4815, 4866, 15365, 15388, 15428
\__tl_act_case_aux:nn . . . . . 15415, 15423, 15426, 15445
\__tl_act_case_group:nn . . . . . 15410, 15430, 15442
\__tl_act_case_normal:nN . . . . . 15410, 15429, 15434
\__tl_act_case_space:n 15410, 15431, 15433
\__tl_act_count_group:nn . . . . . 15384, 15390, 15398
\__tl_act_count_normal:nN . . . . . 15384, 15389, 15396
\__tl_act_count_space:n . . . . . 15384, 15391, 15397
\__tl_act_end:w . . . . . 4815
\__tl_act_end:wn . . . . . 4836, 4842
\__tl_act_group:nwnNNN . 4815, 4828, 4844
\__tl_act_group_recurse:Nnn . . . . . 15375, 15379, 15379
\__tl_act_loop:w . . . . . 4815, 4818, 4822, 4839, 4847, 4854
\__tl_act_normal:NwnNNN 4815, 4825, 4833
\__tl_act_output:n . . . . . 4815, 4857, 15433, 15436, 15444

```

_tl_act_result:n	4820, 4842, 4857, 4858, 4859, 4860
_tl_act_reverse_output:n	4815, 4859, 4876, 4878, 4880, 15376
_tl_act_space:wnnn	4815, 4829, 4851
_tl_case:NnTF	4655, 4660, 4665, 4670, 4672
_tl_case:Nw	4652, 4673, 4674, 4678
_tl_case:nnTF	4652
_tl_case_end:nw	4652, 4677, 4684
_tl_count:n	4740, 4743, 4748, 4750
_tl_head_auxi:nw	4887, 4890, 4892, 4903
_tl_head_auxii:nw	4887, 4893, 4894
_tl_if_blank_p:NNw	4559
_tl_if_empty_return:o	4560, 4593, 4593, 4603
_tl_if_head_eq_meaning_normal:nN	4973, 4977
_tl_if_head_eq_meaning_special:nN	4974, 4986
_tl_if_head_is_space:w	5026, 5029, 5031
_tl_item:nn	15447, 15449, 15462, 15467
_tl_map_function:Nn	4685, 4687, 4693, 4696, 4703
_tl_map_variable:Nnn	4711, 4713, 4719, 4724
_tl_replace:NNNnn	4497, 4498, 4500, 4502, 4504, 4509
_tl_replace:w	4497, 4534, 4537, 4542
_tl_replace_all:	4497, 4502, 4504, 4535, 4538
_tl_replace_once:	4497, 4498, 4500, 4540
_tl_replace_once_end:w	4497, 4543, 4545
_tl_rescan:w	4459, 4477, 4485
_tl_reverse_group:nn	15360, 15367, 15373
_tl_reverse_group_preserve:nn	4861, 4868, 4877
_tl_reverse_items:nwNwn	4753, 4755, 4756, 4760, 4763
_tl_reverse_items:wn	4753, 4757, 4764, 4767
_tl_reverse_normal:nN	4861, 4867, 4875, 15366
_tl_reverse_space:n	4861, 4869, 4879, 15368
_tl_set_rescan:NNnn	4459, 4460, 4462, 4464, 4465
_tl_tmp:w	4518, 4538, 4543, 4639, 4640, 4777, 4814
_tl_trim_spaces:nn	103, 4770, 4777, 4779, 5588, 14866
_tl_trim_spaces_auxi:w	4777, 4781, 4791, 4794, 4800
_tl_trim_spaces_auxii:w	4785, 4799
_tl_trim_spaces_auxii:w_tl_trim_spaces_auxiii:w	4777
_tl_trim_spaces_auxiii:w	4786, 4802, 4805, 4809
_tl_trim_spaces_auxiv:w	4777, 4788, 4811
_token_if_chardef:w	2724, 2739, 2748, 2753
_token_if_dim_register:w	2724, 2764, 2771
_token_if_int_register:w	2724, 2786, 2795
_token_if_long_macro:w	2724, 2866, 2876, 2881
_token_if_macro_p:w	2684, 2695, 2698
_token_if_muskip_register:w	2724, 2806, 2813
_token_if_primitive:NNw	2883, 2896, 2900
_token_if_primitive:Nw	2883, 2919, 2925
_token_if_primitive_loop:N	2883, 2903, 2916, 2922
_token_if_primitive_nullfont:N	2883, 2904, 2908
_token_if_primitive_space:w	2883, 2902, 2907
_token_if_primitive_undefined:N	2883, 2928, 2934
_token_if_protected_macro:w	2724, 2855, 2860
_token_if_skip_register:w	2724, 2824, 2831
_token_if_toks_register:w	2724, 2842, 2849
_use_none_delimit_by_s_stop:w	48, 2462, 2462
\	10954
\~	2600, 2610, 9219
Numbers	
\8	8059
\9	8060
_	301, 1054, 2610, 2767, 2789, 2809, 2827, 2845, 2858, 2869, 2879, 7431, 7432, 7782, 7827, 7846, 7943, 8031, 8035, 8036, 8040, 8041, 9184, 9222

A

\A	1863, 2686, 4450, 4452	.bool_gset_inverse:N	152
\above	422	\bool_gset_true:c	2065
\abovedisplayshortskip	435	\bool_gset_true:N	38, 2065, 2069, 2075
\abovedisplayskip	436	\bool_if:cTF	2091
\abovewithdelims	423	\bool_if:N	2091
\accent	473	\bool_if:n	2125
\adjdemerits	510	\bool_if:Nf	249, 2101, 2216, 2222, 3754, 7157, 8685
\advance	317	\bool_if:nf	2240, 2249
\afterassignment	327	\bool_if:NT	2100, 2214, 2220, 3754, 3755, 6756
\aftergroup	328	\bool_if:nT	2227, 2236, 15313
false	185	\bool_if:NTF	38, 1349, 2091,
nan	184		2102, 3740, 8232, 8564, 8600, 8606,
tan	182		8625, 8627, 8632, 8639, 8661, 9272
tand	182	\bool_if:nTF	39, 2114, 2125, 8570, 8580
\AtBeginDocument	7290, 8979	\bool_if_exist:c	2124
\atop	424	\bool_if_exist:cTF	2123
\atopwithdelims	425	\bool_if_exist:N	2123
max	181	\bool_if_exist:Nf	8249, 8265

B

\B	4451, 4453	\bool_if_exist:NTF	38, 2105, 2123
\badness	572	\bool_if_exist_p:c	2123
\baselineskip	500	\bool_if_exist_p:N	38, 2123
\batchmode	393	\bool_if_p:c	2091
\BeginCatcodeRegime	14394	\bool_if_p:N	38, 2091, 2099
\begingroup	13, 62, 68, 72, 293, 331	\bool_if_p:n	39, 2086, 2088,
\beginL	685		2125, 2127, 2133, 2133, 2206, 2209
\beginR	687	\bool_new:c	2063
\belowdisplayshortskip	437	\bool_new:N	37, 2063, 2063, 2064, 2119,
\belowdisplayskip	438		2120, 2121, 2122, 6474, 8168, 8169,
\binoppenalty	461		8172, 8173, 8177, 8249, 8265, 9181
\bodydir	719	\bool_not_p:n	40, 2206, 2206
\bool_do_until:cn	2219	.bool_set:c	152
\bool_do_until:Nn	40, 2219, 2221, 2222, 2224	\bool_set:cn	2085
\bool_do_until:nn	40, 2225, 2246, 2249	.bool_set:N	152
\bool_do_while:cn	2219	\bool_set:Nn	38, 2085, 2085, 2089
\bool_do_while:Nn	40, 2219, 2219, 2220, 2223	\bool_set_eq:cc	2077, 2080
\bool_do_while:nn	40, 2225, 2233, 2236	\bool_set_eq:cN	2077, 2079
.bool_gset:c	152	\bool_set_eq:Nc	2077, 2078
\bool_gset:cn	2085	\bool_set_eq:NN	38, 2077, 2077
.bool_gset:N	152	\bool_set_false:c	2065
\bool_gset:Nn	38, 2085, 2087, 2090	\bool_set_false:N	37, 264, 2065,
\bool_gset_eq:cc	2077, 2084		2067, 2074, 6752, 7155, 8194, 8521,
\bool_gset_eq:cN	2077, 2083		8537, 8543, 8546, 8556, 8613, 9274
\bool_gset_eq:Nc	2077, 2082	.bool_set_inverse:c	152
\bool_gset_eq:NN	38, 2077, 2081	.bool_set_inverse:N	152
\bool_gset_false:c	2065	\bool_set_true:c	2065
\bool_gset_false:N	37, 2065, 2071, 2076	\bool_set_true:N	38, 278, 2065, 2065, 2073,
.bool_gset_inverse:c	152		

- 6770, 6785, 6816, 8189, 8519, 8533,
- 8534, 8542, 8551, 8620, 9240, 9311
- \bool_show:c 2103
- \bool_show:N 38, 2103, 2103, 2118
- \bool_show:n 38, 2103, 2106, 2112
- \bool_until_do:cn 2213
- \bool_until_do:Nn 40, 2213, 2215, 2216, 2218
- \bool_until_do:nn .. 41, 2225, 2238, 2243
- \bool_while_do:cn 2213
- \bool_while_do:Nn 40, 2213, 2213, 2214, 2217
- \bool_while_do:nn .. 41, 2225, 2225, 2230
- \bool_xor_p:nn 40, 2207, 2207
- \botmark 408
- \botmarks 634
- \box 616
- \box_clear:c 6241
- \box_clear:N 128, 6241,
- 6241, 6245, 6248, 6510, 6569, 6618
- \box_clear_new:c 6247
- \box_clear_new:N .. 128, 6247, 6247, 6251
- \box_clip:c 14719
- \box_clip:N 192, 14719, 14719, 14721
- \box_dp:c 6267, 6640
- \box_dp:N 129,
- 6267, 6268, 6271, 6274, 6639, 6710,
- 6712, 6729, 6743, 6896, 6914, 7227,
- 7267, 14502, 14607, 14637, 14657,
- 14676, 14730, 14737, 14742, 14965
- \box_gclear:c 6241
- \box_gclear:N . 128, 6241, 6243, 6246, 6250
- \box_gclear_new:c 6247
- \box_gclear_new:N . 128, 6247, 6249, 6252
- \box_gset_eq:cc 6253
- \box_gset_eq:cN 6253
- \box_gset_eq:Nc 6253
- \box_gset_eq:NN 128, 6244, 6253, 6255, 6258
- \box_gset_eq_clear:cc 6259
- \box_gset_eq_clear:cN 6259
- \box_gset_eq_clear:Nc 6259
- \box_gset_eq_clear:NN 128, 6259, 6261, 6264
- \box_gset_to_last:c 6315
- \box_gset_to_last:N 131, 6315, 6317, 6320
- \box_ht:c 6267, 6642
- \box_ht:N 130, 6267, 6267, 6270,
- 6276, 6565, 6613, 6641, 6706, 6708,
- 6729, 6736, 6895, 6913, 7225, 7266,
- 14501, 14606, 14636, 14656, 14675,
- 14747, 14752, 14757, 14962, 14964
- \box_if_empty:cTF 6309
- \box_if_empty:N 6309
- \box_if_empty:Nf 6313
- \box_if_empty:NTF 6312
- \box_if_empty:Ntf 130, 6309, 6314
- \box_if_empty_p:c 6309
- \box_if_empty_p:N 130, 6309, 6311
- \box_if_exist:c 6266
- \box_if_exist:cTF 6265
- \box_if_exist:N 6265
- \box_if_exist:Ntf 129, 6248, 6250, 6265, 6353
- \box_if_exist_p:c 6265
- \box_if_exist_p:N 129, 6265
- \box_if_horizontal:cTF 6297
- \box_if_horizontal:N 6297
- \box_if_horizontal:Nf 6303
- \box_if_horizontal:NT 6302
- \box_if_horizontal:Ntf . 130, 6297, 6304
- \box_if_horizontal_p:c 6297
- \box_if_horizontal_p:N . 130, 6297, 6301
- \box_if_vertical:cTF 6297
- \box_if_vertical:N 6299
- \box_if_vertical:Nf 6307
- \box_if_vertical:NT 6306
- \box_if_vertical:Ntf ... 130, 6297, 6308
- \box_if_vertical_p:c 6297
- \box_if_vertical_p:N ... 130, 6297, 6305
- \box_log:c 6332
- \box_log:cn 6332
- \box_log:N 131, 6332, 6332, 6334
- \box_log:Nnn .. 132, 6332, 6333, 6335, 6346
- \box_move_down:nn .. 129, 6286, 6292,
- 14734, 14742, 14777, 14784, 14944
- \box_move_left:nn 129, 6286, 6286
- \box_move_right:nn 129, 6286, 6288
- \box_move_up:nn 129, 6286, 6290,
- 6934, 7222, 14750, 14757, 14790, 14801
- \box_new:c 6233
- \box_new:N 128, 6233,
- 6234, 6240, 6248, 6250, 6321, 6322,
- 6323, 6324, 6325, 6452, 6517, 14486
- \box_resize:cn 14601
- \box_resize:Nnn 191, 14601, 14601, 14621, 15074
- \box_resize_to_ht_plus_dp:cn 14631
- \box_resize_to_ht_plus_dp:Nn 192, 14631, 14631, 14650
- \box_resize_to_wd:cn 14631
- \box_resize_to_wd:Nn 192, 14631, 14651, 14667
- \box_rotate:Nn .. 192, 14487, 14487, 14937

- \box_scale:cnn [14668](#)
 - \box_scale:Nnn
 [192](#), [14668](#), [14668](#), [14689](#), [15096](#)
 - \box_set_dp:cn [6273](#)
 - \box_set_dp:Nn . [130](#), [6273](#), [6273](#), [6280](#),
 [6896](#), [6914](#), [7226](#), [14528](#), [14699](#),
 [14737](#), [14745](#), [14780](#), [14785](#), [14949](#)
 - \box_set_eq:cc [6253](#)
 - \box_set_eq:cN [6253](#)
 - \box_set_eq:Nc [6253](#)
 - \box_set_eq:NN
 [128](#), [6242](#), [6253](#), [6253](#), [6256](#),
 [6257](#), [6628](#), [6916](#), [7230](#), [14762](#), [14806](#)
 - \box_set_eq_clear:cc [6259](#)
 - \box_set_eq_clear:cN [6259](#)
 - \box_set_eq_clear:Nc [6259](#)
 - \box_set_eq_clear:NN
 [128](#), [6259](#), [6259](#), [6262](#), [6263](#)
 - \box_set_ht:cn [6273](#)
 - \box_set_ht:Nn . [130](#), [6273](#), [6275](#), [6279](#),
 [6895](#), [6913](#), [7224](#), [14527](#), [14698](#),
 [14751](#), [14760](#), [14791](#), [14804](#), [14947](#)
 - \box_set_to_last:c [6315](#)
 - \box_set_to_last:N
 [131](#), [6315](#), [6315](#), [6318](#), [6319](#)
 - \box_set_wd:cn [6273](#)
 - \box_set_wd:Nn . [130](#), [6273](#), [6277](#), [6281](#),
 [6897](#), [6915](#), [7228](#), [14529](#), [14710](#), [14950](#)
 - \box_show:c [6326](#)
 - \box_show:cnn [6326](#)
 - \box_show:N [131](#), [6326](#), [6326](#), [6328](#)
 - \box_show:Nnn . [131](#), [6326](#), [6327](#), [6329](#), [6331](#)
 - \box_trim:cnnnn [14722](#)
 - \box_trim:Nnnnn . [193](#), [14722](#), [14722](#), [14764](#)
 - \box_use:c [6282](#)
 - \box_use:N [129](#), [6282](#),
 [6283](#), [6285](#), [6931](#), [6934](#), [7012](#), [7149](#),
 [7219](#), [7222](#), [14516](#), [14523](#), [14531](#),
 [14695](#), [14705](#), [14714](#), [14727](#), [14735](#),
 [14743](#), [14750](#), [14758](#), [14770](#), [14778](#),
 [14784](#), [14790](#), [14802](#), [14945](#), [14952](#)
 - \box_use_clear:c [6282](#)
 - \box_use_clear:N . [129](#), [6282](#), [6282](#), [6284](#)
 - \box_viewport:cnnnn [14765](#)
 - \box_viewport:Nnnnn
 [193](#), [14765](#), [14765](#), [14808](#)
 - \box_wd:c [6267](#), [6644](#)
 - \box_wd:N . [130](#), [6267](#), [6269](#), [6272](#), [6278](#),
 [6643](#), [6708](#), [6712](#), [6718](#), [6723](#), [6864](#),
 [6897](#), [6915](#), [6932](#), [7220](#), [7229](#), [7268](#),
 [14503](#), [14608](#), [14638](#), [14658](#), [14677](#),
 [14771](#), [14964](#), [14969](#), [15133](#), [15140](#)
 - \boxmaxdepth 578
 - \brokenpenalty 535
 - abs 181
- C**
- cc 184
 - nc 184
 - pc 184
 - \c__coffin_corners_prop . [6455](#), [6455](#),
 [6456](#), [6457](#), [6458](#), [6459](#), [6521](#), [6658](#)
 - \c__coffin_poles_prop [6460](#),
 [6460](#), [6462](#), [6463](#), [6464](#), [6466](#), [6467](#),
 [6468](#), [6469](#), [6470](#), [6471](#), [6523](#), [6660](#)
 - \c__fp_big_leading_shift_int
 [9516](#), [9516](#), [12095](#), [12105](#), [12115](#)
 - \c__fp_big_middle_shift_int
 [9516](#), [9517](#), [12097](#),
 [12107](#), [12117](#), [12126](#), [12129](#), [12132](#)
 - \c__fp_big_trailing_shift_int
 [9516](#), [9518](#), [12139](#)
 - \c__fp_Bigg_leading_shift_int
 [9523](#), [9523](#), [11860](#), [11877](#)
 - \c__fp_Bigg_middle_shift_int
 [9523](#), [9524](#), [11863](#), [11866](#), [11880](#), [11883](#)
 - \c__fp_Bigg_trailing_shift_int
 [9523](#), [9525](#), [11869](#), [11886](#)
 - \c__fp_leading_shift_int
 [9510](#), [9510](#), [11983](#),
 [11992](#), [12067](#), [12962](#), [13443](#), [13480](#)
 - \c__fp_ln_i_fixed_tl [12413](#), [12413](#)
 - \c__fp_ln_ii_fixed_tl [12413](#), [12414](#)
 - \c__fp_ln_iii_fixed_tl ... [12413](#), [12415](#)
 - \c__fp_ln_iv_fixed_tl [12413](#), [12416](#)
 - \c__fp_ln_ix_fixed_tl [12413](#), [12420](#)
 - \c__fp_ln_vi_fixed_tl [12413](#), [12417](#)
 - \c__fp_ln_vii_fixed_tl ... [12413](#), [12418](#)
 - \c__fp_ln_viii_fixed_tl .. [12413](#), [12419](#)
 - \c__fp_ln_x_fixed_tl
 [12413](#), [12421](#), [12630](#), [12637](#)
 - \c__fp_max_exponent_int
 [9411](#), [9411](#), [9417](#),
 [9423](#), [9439](#), [9440](#), [12192](#), [12393](#),
 [12683](#), [12710](#), [12997](#), [13907](#), [13956](#)
 - \c__fp_middle_shift_int
 [9510](#), [9511](#), [11995](#),
 [11998](#), [12001](#), [12004](#), [12069](#), [12072](#),
 [12075](#), [12078](#), [12964](#), [12967](#), [12970](#),
 [12973](#), [13446](#), [13453](#), [13483](#), [13489](#)

- \c__fp_one_fixed_tl [11972](#), [11972](#), [12584](#), [12742](#),
[12998](#), [13022](#), [13649](#), [13715](#), [13820](#)
- \c__fp_trailing_shift_int [9510](#), [9512](#), [11985](#),
[12007](#), [12081](#), [12976](#), [13446](#), [13483](#)
- \c__int_from_roman_C_int [3817](#)
- \c__int_from_roman_c_int [3817](#)
- \c__int_from_roman_D_int [3817](#)
- \c__int_from_roman_d_int [3817](#)
- \c__int_from_roman_I_int [3817](#)
- \c__int_from_roman_i_int [3817](#)
- \c__int_from_roman_L_int [3817](#)
- \c__int_from_roman_l_int [3817](#)
- \c__int_from_roman_M_int [3817](#)
- \c__int_from_roman_m_int [3817](#)
- \c__int_from_roman_V_int [3817](#)
- \c__int_from_roman_v_int [3817](#)
- \c__int_from_roman_X_int [3817](#)
- \c__int_from_roman_x_int [3817](#)
- \c__iow_wrap_end_marker_tl .. [9187](#), [9246](#)
- \c__iow_wrap_indent_marker_tl [9187](#), [9207](#)
- \c__iow_wrap_marker_tl [9187](#), [9189](#), [9198](#), [9256](#), [9301](#)
- \c__iow_wrap_newline_marker_tl [9187](#), [9221](#)
- \c__iow_wrap_unindent_marker_tl [9187](#), [9209](#)
- \c__keys_code_root_tl ... [8160](#), [8160](#),
[8308](#), [8673](#), [8675](#), [8696](#), [8702](#), [8707](#)
- \c__keys_info_root_tl [8160](#), [8161](#), [8307](#),
[8313](#), [8314](#), [8318](#), [8321](#), [8366](#), [8368](#),
[8369](#), [8370](#), [8594](#), [8596](#), [8641](#), [8649](#),
[8651](#), [8766](#), [8769](#), [8771](#), [8782](#), [8802](#)
- \c__keys_props_root_tl .. [8162](#), [8162](#),
[8200](#), [8236](#), [8243](#), [8380](#), [8382](#), [8384](#),
[8386](#), [8388](#), [8390](#), [8392](#), [8394](#), [8396](#),
[8398](#), [8400](#), [8402](#), [8404](#), [8406](#), [8408](#),
[8410](#), [8412](#), [8414](#), [8416](#), [8418](#), [8420](#),
[8422](#), [8424](#), [8426](#), [8428](#), [8430](#), [8432](#),
[8434](#), [8436](#), [8438](#), [8440](#), [8442](#), [8444](#),
[8446](#), [8448](#), [8450](#), [8452](#), [8454](#), [8456](#),
[8458](#), [8460](#), [8462](#), [8464](#), [8466](#), [8468](#),
[8470](#), [8472](#), [8474](#), [8476](#), [8478](#), [8480](#),
[8482](#), [8484](#), [8486](#), [8488](#), [8490](#), [8492](#),
[8494](#), [8496](#), [8498](#), [8775](#), [8777](#), [8812](#)
- \c__max_constdef_int [3239](#), [3243](#), [3263](#), [3267](#)
- \c__msg_more_text_prefix_tl [7312](#), [7313](#), [7351](#), [7360](#), [7551](#)
- \c__msg_text_prefix_tl [7312](#),
[7312](#), [7316](#), [7349](#), [7358](#), [7531](#), [7542](#),
[7557](#), [7574](#), [7582](#), [7588](#), [7969](#), [7996](#)
- \c__tl_act_lowercase_tl [15400](#), [15405](#), [15423](#)
- \c__tl_act_uppercase_tl [15400](#), [15400](#), [15415](#)
- \c__tl_rescan_marker_tl [4449](#), [4457](#), [4468](#), [4486](#)
- \c_alignment_token .. [53](#), [2572](#), [2578](#), [2628](#)
- \c_catcode_active_tl [53](#), [2587](#), [2589](#), [2666](#)
- \c_catcode_letter_token [53](#), [2572](#), [2584](#), [2656](#)
- \c_catcode_other_space_tl [168](#), [9182](#), [9185](#), [9197](#), [9199](#), [9201](#), [9222](#)
- \c_catcode_other_token [53](#), [2572](#), [2585](#), [2661](#)
- \c_code_cctab ... [189](#), [14414](#), [14416](#), [14417](#)
- \c_document_cctab [190](#), [14414](#), [14422](#), [14431](#), [14434](#)
- \c_e_fp [176](#), [14173](#), [14173](#)
- \c_eight [73](#), [2500](#), [2532](#), [3816](#), [3881](#), [3886](#),
[9602](#), [10319](#), [10351](#), [12676](#), [13056](#),
[13069](#), [13427](#), [13429](#), [13431](#), [13490](#)
- \c_eleven [73](#), [2506](#),
[2538](#), [3881](#), [3889](#), [11593](#), [11668](#), [11709](#)
- \c_empty_box [131](#), [6242](#), [6244](#), [6321](#), [6321](#), [7009](#)
- \c_empty_clist [121](#), [5548](#), [5548](#), [5647](#), [5662](#), [5684](#), [5700](#)
- \c_empty_coffin [139](#), [6633](#), [6633](#), [6634](#), [7010](#)
- \c_empty_prop [127](#), [5963](#), [5963](#), [5967](#), [5971](#), [5974](#), [6132](#)
- \c_empty_seq [112](#), [5071](#), [5071](#),
[5075](#), [5079](#), [5082](#), [5233](#), [5270](#), [5278](#)
- \c_empty_tl [103](#), [3629](#), [4319](#),
[4335](#), [4337](#), [4362](#), [4362](#), [4571](#), [5548](#)
- \c_false_bool [21](#),
[921](#), [960](#), [1000](#), [1005](#), [1049](#), [1050](#),
[1071](#), [1295](#), [1302](#), [1471](#), [1473](#), [1482](#),
[1494](#), [1892](#), [2063](#), [2068](#), [2072](#), [2181](#),
[2183](#), [2187](#), [2210](#), [3729](#), [3734](#), [3743](#)
- \c_fifteen [73](#),
[2514](#), [2546](#), [3881](#), [3892](#), [10677](#), [10904](#)
- \c_five [73](#), [2494](#), [2526](#), [3881](#),
[3885](#), [9856](#), [10909](#), [11907](#), [12317](#), [12683](#)
- \c_four [73](#),
[2492](#), [2524](#), [3881](#), [3884](#), [9315](#), [9321](#),
[10910](#), [12156](#), [12172](#), [12357](#), [12398](#),
[12456](#), [12635](#), [13229](#), [13235](#), [13238](#),
[13507](#), [13634](#), [13778](#), [13851](#), [14067](#)

- \c_fourteen [73](#), [2512](#), [2544](#), [3881](#), [3891](#), [10904](#)
- \c_group_begin_token [53](#), [2572](#),
[2572](#), [2613](#), [4962](#), [4997](#), [6373](#), [6421](#)
- \c_group_end_token
[53](#), [2572](#), [2573](#), [2618](#), [6378](#), [6379](#), [6429](#)
- \c_inf_fp [176](#),
[9406](#), [9408](#), [10635](#), [11736](#), [11816](#),
[12652](#), [12886](#), [12890](#), [12911](#), [13187](#)
- \c_initex_cctab [190](#), [14414](#), [14423](#)
- \c_job_name_tl [103](#), [4363](#), [4376](#), [4380](#)
- \c_log_iow ... [168](#), [9099](#), [9099](#), [9164](#), [9165](#)
- \c_math_subscript_token
..... [53](#), [2572](#), [2582](#), [2646](#)
- \c_math_superscript_token
..... [53](#), [2572](#), [2580](#), [2641](#)
- \c_math_toggle_token [53](#), [2572](#), [2576](#), [2623](#)
- \c_max_dim [82](#), [4149](#), [4150](#), [4240](#),
[15016](#), [15017](#), [15018](#), [15019](#), [15036](#)
- \c_max_int [73](#), [3899](#), [3899](#), [6327](#), [6333](#)
- \c_max_muskip [88](#), [4301](#), [4302](#)
- \c_max_register_int
[73](#), [808](#), [809](#), [811](#), [7846](#), [14334](#), [14365](#)
- \c_max_skip [85](#), [4239](#), [4240](#)
- \c_minus_inf_fp [176](#),
[9406](#), [9409](#), [11737](#), [11819](#), [12429](#), [13188](#)
- \c_minus_one
[73](#), [796](#), [797](#), [800](#), [801](#), [1156](#), [1316](#),
[3241](#), [3302](#), [3881](#), [4469](#), [4470](#), [7475](#),
[9049](#), [9092](#), [9099](#), [9142](#), [9188](#), [9214](#),
[10756](#), [11059](#), [11270](#), [11425](#), [12017](#),
[13049](#), [13058](#), [13084](#), [13712](#), [14326](#)
- \c_minus_zero_fp
..... [175](#), [9406](#), [9407](#), [11733](#), [13836](#)
- \c_msg_coding_error_text_tl
..... [7248](#), [7259](#), [7365](#),
[7365](#), [7781](#), [7789](#), [7815](#), [7823](#), [7832](#),
[7839](#), [7853](#), [7862](#), [7869](#), [7876](#), [7883](#),
[7890](#), [7898](#), [8725](#), [8740](#), [8747](#), [8808](#)
- \c_msg_continue_text_tl [7365](#), [7370](#), [7409](#)
- \c_msg_critical_text_tl [7365](#), [7372](#), [7545](#)
- \c_msg_fatal_text_tl ... [7365](#), [7374](#), [7534](#)
- \c_msg_help_text_tl [7365](#), [7376](#), [7413](#)
- \c_msg_no_info_text_tl . [7365](#), [7378](#), [7408](#)
- \c_msg_on_line_text_tl . [7365](#), [7383](#), [7400](#)
- \c_msg_return_text_tl
.. [7365](#), [7381](#), [7384](#), [7784](#), [7792](#), [7799](#)
- \c_msg_trouble_text_tl [7365](#), [7391](#)
- \c_nan_fp [9406](#), [9410](#), [9710](#),
[9718](#), [9790](#), [10076](#), [10081](#), [10159](#),
[10198](#), [10636](#), [10747](#), [12859](#), [13618](#)
- \c_nine ... [73](#), [2502](#), [2534](#), [3881](#), [3887](#),
[10027](#), [10054](#), [10226](#), [10246](#), [10273](#),
[10287](#), [10322](#), [10349](#), [10377](#), [10393](#),
[10409](#), [10427](#), [10506](#), [10522](#), [10907](#),
[10908](#), [11719](#), [13229](#), [13235](#), [13238](#)
- \c_one [73](#), [2486](#), [2518](#), [3215](#), [3300](#),
[3881](#), [3881](#), [4750](#), [5455](#), [5899](#), [5903](#),
[6330](#), [7709](#), [7715](#), [9235](#), [9439](#), [9632](#),
[9834](#), [9886](#), [9887](#), [9890](#), [9967](#), [9992](#),
[10236](#), [10299](#), [10309](#), [10378](#), [10394](#),
[10398](#), [10435](#), [10608](#), [10741](#), [10856](#),
[10861](#), [10868](#), [11268](#), [11387](#), [11399](#),
[11403](#), [11519](#), [11544](#), [11583](#), [11637](#),
[11640](#), [11662](#), [11666](#), [11686](#), [11710](#),
[11726](#), [11797](#), [11830](#), [11919](#), [11921](#),
[11948](#), [12029](#), [12181](#), [12245](#), [12247](#),
[12296](#), [12297](#), [12372](#), [12408](#), [12440](#),
[12486](#), [12581](#), [12677](#), [12679](#), [12724](#),
[13018](#), [13028](#), [13051](#), [13071](#), [13074](#),
[13087](#), [13160](#), [13198](#), [13246](#), [13585](#),
[13611](#), [13631](#), [13690](#), [13796](#), [13858](#),
[13924](#), [13929](#), [14189](#), [14334](#), [14824](#),
[15221](#), [15232](#), [15454](#), [15465](#), [15506](#)
- \c_one_degree_fp [176](#), [10638](#), [14175](#), [14176](#)
- \c_one_fp
. [175](#), [10639](#), [11016](#), [11021](#), [11026](#),
[11037](#), [11309](#), [12646](#), [12854](#), [12901](#),
[13111](#), [13141](#), [13612](#), [14173](#), [14174](#)
- \c_one_hundred [73](#), [3896](#), [3896](#)
- \c_one_thousand [73](#), [3896](#), [3897](#)
- \c_other_cctab
..... [190](#), [14414](#), [14424](#), [14432](#), [14442](#)
- \c_parameter_token
..... [53](#), [2572](#), [2579](#), [2632](#), [2635](#)
- \c_pi_fp [176](#), [10637](#), [14175](#), [14175](#)
- \c_seven [73](#), [796](#),
[806](#), [2498](#), [2530](#), [3881](#), [10358](#), [11046](#),
[11091](#), [11614](#), [11721](#), [13648](#), [13666](#)
- \c_six [73](#), [796](#), [805](#), [2496](#), [2528](#), [3881](#)
- \c_sixteen [73](#), [796](#),
[803](#), [1158](#), [3814](#), [3881](#), [8986](#), [9049](#),
[9074](#), [9100](#), [9142](#), [9536](#), [9606](#), [9943](#),
[10607](#), [10686](#), [10734](#), [12698](#), [13057](#),
[13062](#), [13975](#), [13977](#), [13983](#), [14024](#)
- \c_space_tl
[103](#), [4382](#), [4382](#), [4851](#), [5895](#), [5904](#),
[7401](#), [8943](#), [9233](#), [9247](#), [9317](#), [14116](#)
- \c_space_token [53](#), [2572](#),
[2583](#), [2651](#), [3043](#), [4963](#), [4998](#), [15505](#)
- \c_str_cctab [190](#), [14414](#), [14425](#), [14433](#), [14447](#)

- `\c_ten` 73, 2504, 2536, 3654, 3881, 3888,
10905, 10906, 11718, 12457, 13629
- `\c_ten_thousand` 73, 3896, 3898,
11978, 12009, 12010, 12088, 12149
- `\c_term_ior` 168, 8986, 8986, 8997, 9055, 9148
- `\c_term_iow` 168, 9099, 9100, 9112, 9166, 9167
- `\c_thirteen` 73,
2510, 2542, 3881, 3890, 13444, 13481
- `\c_thirty_two` 73, 3893, 3893,
10137, 10172, 10803, 10916, 10920
- `\c_three` 73, 2490, 2522, 3881, 3883, 9441,
10138, 10175, 10805, 10985, 10989,
10999, 11423, 11713, 13175, 13670
- `\c_token_A_int` 2883, 2918
- `\c_true_bool` 21,
960, 1049, 1049, 1070, 1313, 1472,
1483, 1493, 2066, 2070, 2093, 2182,
2184, 2188, 2211, 3729, 3734, 3747
- `\c_twelve` . . . 73, 796, 807, 2312, 2327,
2508, 2540, 2731, 3881, 10589, 10590
- `\c_two` 73, 2488,
2520, 3215, 3812, 3881, 3882, 7710,
9436, 9440, 11005, 11188, 11339,
11570, 11701, 11716, 11719, 11721,
11901, 12041, 12192, 12296, 12297,
12311, 12393, 12589, 12710, 12997,
13028, 13044, 13115, 13145, 13235,
13507, 13517, 13560, 13585, 13631,
13633, 13719, 13741, 14027, 14332,
14364, 14372, 15503, 15504, 15505
- `\c_two_hundred_fifty_five` 73, 3894, 3894
- `\c_two_hundred_fifty_six` . 73, 3894, 3895
- `\c_undefined_fp` 14184, 14184
- `\c_zero` 73, 796, 804, 960, 970,
975, 980, 1057, 1059, 1500, 1505,
1510, 1573, 1640, 1649, 2283, 2286,
2287, 2288, 2289, 2290, 2291, 2292,
2293, 2294, 2295, 2305, 2307, 2375,
2382, 2399, 2426, 2435, 2484, 2516,
2700, 3210, 3269, 3270, 3320, 3328,
3492, 3495, 3627, 3881, 4139, 4206,
4843, 5034, 5035, 6339, 6340, 7957,
9441, 9601, 9626, 9838, 9842, 9844,
9848, 9852, 9865, 9877, 9888, 9889,
9898, 9902, 9906, 9909, 9914, 9918,
9950, 9955, 10131, 10169, 10271,
10340, 10387, 10417, 10460, 10465,
10487, 10536, 10550, 10597, 10610,
10762, 10793, 10840, 11180, 11354,
11359, 11635, 11919, 12611, 12615,
12629, 12687, 12693, 12707, 12897,
12909, 12927, 12950, 12982, 13065,
13066, 13080, 13082, 13100, 13130,
13198, 13496, 13620, 13629, 13635,
13669, 13683, 13688, 13973, 13998,
14820, 14835, 14855, 15220, 15453
- `\c_zero_dim` 82, 3930, 4149,
4149, 4239, 6388, 6402, 6766, 6769,
6772, 6781, 6784, 6787, 6796, 6803,
6860, 6865, 6872, 14713, 14734,
14745, 14750, 14760, 14773, 14777,
14785, 14787, 14790, 14794, 14804
- `\c_zero_fp` 175, 9406, 9406, 9449, 10640,
11017, 11022, 11027, 11036, 11311,
11566, 11732, 12655, 12881, 12914,
13834, 14064, 14126, 14141, 14142,
14505, 14507, 14512, 14700, 15084
- `\c_zero_muskip` 88, 4260, 4301, 4301
- `\c_zero_skip`
. . 85, 4169, 4239, 4239, 15348, 15349
- `\catcode` 4, 5, 6, 7, 10, 112, 113, 114, 115,
116, 117, 118, 119, 120, 126, 127,
128, 129, 130, 131, 132, 133, 236,
237, 238, 239, 240, 241, 242, 243, 620
- `\catcodetable` 713
- `\CatcodeTableIniTeX` 14423
- `\CatcodeTableLaTeX` 14422
- `\CatcodeTableOther` 14424
- `\CatcodeTableString` 14425
- `\cctab_begin:N`
189, 14360, 14360, 14380, 14383, 14394
- `\cctab_end:`
189, 14360, 14370, 14385, 14388, 14395
- `\cctab_gset:Nn` 189, 14399, 14399, 14408,
14411, 14417, 14434, 14442, 14447
- `\cctab_new:N`
. 189, 14329, 14329, 14343, 14346,
14352, 14416, 14431, 14432, 14433
- `\char` 474, 2742
- `\char_gset_active:Npn` . 200, 15472, 15486
- `\char_gset_active:Npx` 15472, 15487
- `\char_gset_active_eq:NN` 200, 15472, 15489
- `\char_set_active:Npn` . . 200, 15472, 15484
- `\char_set_active:Npx` 15472, 15485
- `\char_set_active_eq:NN` 200, 15472, 15488
- `\char_set_catcode:nn` 51, 253,
254, 255, 256, 257, 258, 259, 260,
261, 2477, 2477, 2484, 2486, 2488,
2490, 2492, 2494, 2496, 2498, 2500,
2502, 2504, 2506, 2508, 2510, 2512,

- 2514, 2516, 2518, 2520, 2522, 2524,
2526, 2528, 2530, 2532, 2534, 2536,
2538, 2540, 2542, 2544, 2546, 2731
- \char_set_catcode_active:N
. 50, 2483, 2509, 2588, 2595, 2596,
2597, 2598, 2599, 2600, 7434, 15473
- \char_set_catcode_active:n 50,
2515, 2541, 8057, 8058, 14440, 15478
- \char_set_catcode_alignment:N
..... 50, 2483, 2491, 2577
- \char_set_catcode_alignment:n
..... 50, 271, 2515, 2523
- \char_set_catcode_comment:N
..... 50, 2483, 2511
- \char_set_catcode_comment:n
..... 50, 2515, 2543
- \char_set_catcode_end_line:N
..... 50, 2483, 2493
- \char_set_catcode_end_line:n
..... 50, 2515, 2525
- \char_set_catcode_escape:N 50, 2483, 2483
- \char_set_catcode_escape:n 50, 2515, 2515
- \char_set_catcode_group_begin:N
..... 50, 2483, 2485
- \char_set_catcode_group_begin:n
..... 50, 2515, 2517
- \char_set_catcode_group_end:N
..... 50, 2483, 2487
- \char_set_catcode_group_end:n
..... 50, 2515, 2519
- \char_set_catcode_ignore:N 50, 2483, 2501
- \char_set_catcode_ignore:n
..... 50, 268, 269, 2515, 2533
- \char_set_catcode_invalid:N
..... 50, 2483, 2513
- \char_set_catcode_invalid:n
..... 50, 2515, 2545
- \char_set_catcode_letter:N
..... 50, 2483, 2505,
10600, 10601, 10787, 10837, 10853,
10882, 10883, 10884, 10885, 10940,
10954, 10955, 10982, 11315, 11316
- \char_set_catcode_letter:n
..... 50, 272, 274, 2515, 2537
- \char_set_catcode_math_subscript:N .
..... 50, 2483, 2499, 2581
- \char_set_catcode_math_subscript:n .
..... 50, 2515, 2531, 14439
- \char_set_catcode_math_superscript:N
..... 50, 2483, 2497, 7942
- \char_set_catcode_math_superscript:n
..... 50, 273, 2515, 2529
- \char_set_catcode_math_toggle:N
..... 50, 2483, 2489, 2575
- \char_set_catcode_math_toggle:n
..... 50, 2515, 2521
- \char_set_catcode_other:N
. 50, 2483, 2507, 2685, 2686, 2885,
9183, 10009, 10010, 10011, 10095,
10096, 10097, 10098, 10881, 13890,
15493, 15494, 15495, 15496, 15497
- \char_set_catcode_other:n .. 50, 270,
275, 2515, 2539, 14438, 14445, 14450
- \char_set_catcode_parameter:N
..... 50, 2483, 2495
- \char_set_catcode_parameter:n
..... 50, 2515, 2527
- \char_set_catcode_space:N 50, 2483, 2503
- \char_set_catcode_space:n 50,
276, 2515, 2535, 14436, 14437, 14451
- \char_set_lccode:nn 51, 2547,
2553, 2687, 2688, 2689, 2725, 2726,
2727, 2728, 2729, 2886, 2887, 2888,
7431, 7432, 7433, 7943, 7944, 7945,
7946, 8059, 8060, 9184, 10012, 15480
- \char_set_mathcode:nn ... 52, 2547, 2547
- \char_set_sfcode:nn 52, 2547, 2565
- \char_set_uccode:nn 52, 2547, 2559
- \char_show_value_catcode:n 51, 2477, 2481
- \char_show_value_lccode:n 51, 2547, 2557
- \char_show_value_mathcode:n
..... 52, 2547, 2551
- \char_show_value_sfcode:n 53, 2547, 2569
- \char_show_value_uccode:n 52, 2547, 2563
- \char_tmp:NN 15474, 15484,
15485, 15486, 15487, 15488, 15489
- \char_value_catcode:n 51, 253, 254, 255,
256, 257, 258, 259, 260, 261, 2477, 2479
- \char_value_lccode:n 51, 2547, 2555
- \char_value_mathcode:n .. 52, 2547, 2549
- \char_value_sfcode:n 52, 2547, 2567
- \char_value_uccode:n 52, 2547, 2561
- \chardef 122, 135, 138, 283, 309
- .choice: 152
- .choices:nn 152
- .choices:on 152
- .choices:Vn 152
- .choices:xn 152
- \cleaders 492
- \clist_clear:c 5553, 5554

\clist_clear:N	113,	\clist_gput_left:Nx	5619, 5724
5553, 5553, 5736, 8512, 8526, 14871		\clist_gput_right:cn	5632
\clist_clear_new:c	5557, 5558	\clist_gput_right:co	5632
\clist_clear_new:N	113, 5557, 5557	\clist_gput_right:cV	5632
\clist_concat:ccc	5569	\clist_gput_right:cx	5632
\clist_concat:NNN		\clist_gput_right:Nn	
... 114, 5569, 5569, 5582, 5620, 5633		114, 5632, 5634, 5643, 5644	
\clist_const:cn	14898	\clist_gput_right:No	5632
\clist_const:cx	14898	\clist_gput_right:NV	5632
\clist_const:Nn	194, 14898, 14898, 14900	\clist_gput_right:Nx	5632
\clist_const:Nx	14898	\clist_gremove_all:cn	5746
\clist_count:c	5881	\clist_gremove_all:Nn	115, 5746, 5748, 5776
\clist_count:N		\clist_gremove_duplicates:c	5730
... 118, 5881, 5881, 5889, 5910, 14813		\clist_gremove_duplicates:N	
\clist_count:n	5881, 5890, 14843	115, 5730, 5732, 5745	
\clist_gclear:c	5553, 5556	.clist_gset:c	152
\clist_gclear:N	113, 5553, 5555, 14873	\clist_gset:cn	5613
\clist_gclear_new:c	5557, 5560	\clist_gset:co	5613
\clist_gclear_new:N	113, 5557, 5559	\clist_gset:cV	5613
\clist_gconcat:ccc	5569	\clist_gset:cx	5613
\clist_gconcat:NNN		.clist_gset:N	152
... 114, 5569, 5571, 5583, 5622, 5635		\clist_gset:Nn	114, 5613, 5615, 5618
\clist_get:cN	5645	\clist_gset:No	5613
\clist_get:cNTF	5682	\clist_gset:NV	5613
\clist_get:NN	120, 5645, 5645, 5655, 5682	\clist_gset:Nx	5613
\clist_get:NNF	5692	\clist_gset_eq:cc	5561, 5568
\clist_get:NNT	5691	\clist_gset_eq:cN	5561, 5567
\clist_get:NNTF	120, 5682, 5693	\clist_gset_eq:Nc	5561, 5566
\clist_gpop:cN	5656	\clist_gset_eq:NN	114, 5561, 5565, 5733
\clist_gpop:cNTF	5682	\clist_gset_from_seq:cc	14870
\clist_gpop:NN	120, 5656, 5658, 5681, 5696	\clist_gset_from_seq:cN	14870
\clist_gpop:NNF	5711	\clist_gset_from_seq:Nc	14870
\clist_gpop:NNT	5710	\clist_gset_from_seq:NN	
\clist_gpop:NNTF	120, 5682, 5712	194, 14870, 14872, 14896, 14897	
\clist_gpush:cn	5713, 5725	\clist_if_empty:c	5778
\clist_gpush:co	5713, 5727	\clist_if_empty:cTF	5777
\clist_gpush:cV	5713, 5726	\clist_if_empty:N	5777
\clist_gpush:cx	5713, 5728	\clist_if_empty:n	14901
\clist_gpush:Nn	121, 5713, 5721	\clist_if_empty:NF	
\clist_gpush:No	5713, 5723	5578, 5763, 5806, 5836, 5855	
\clist_gpush:NV	5713, 5722	\clist_if_empty:NTF	115, 5777, 7919
\clist_gpush:Nx	5713, 5724	\clist_if_empty:nTF	194, 14901
\clist_gput_left:cn	5619, 5725	\clist_if_empty_p:c	5777
\clist_gput_left:co	5619, 5727	\clist_if_empty_p:N	115, 5777
\clist_gput_left:cV	5619, 5726	\clist_if_empty_p:n	194, 14901
\clist_gput_left:cx	5619, 5728	\clist_if_exist:c	5585
\clist_gput_left:Nn		\clist_if_exist:cTF	5584
... 114, 5619, 5621, 5630, 5631, 5721		\clist_if_exist:N	5584
\clist_gput_left:No	5619, 5723	\clist_if_exist:NT	8964
\clist_gput_left:NV	5619, 5722	\clist_if_exist:NTF	114, 5584, 5908, 8937

\clist_if_exist_p:c	5584	\clist_push:Nn	121, 5713, 5713
\clist_if_exist_p:N	114, 5584	\clist_push:No	5713, 5715
\clist_if_in:cnTF	5779	\clist_push:Nv	5713, 5714
\clist_if_in:coTF	5779	\clist_push:Nx	5713, 5716
\clist_if_in:cVTF	5779	\clist_put_left:cn	5619, 5717
\clist_if_in:Nn	5779	\clist_put_left:co	5619, 5719
\clist_if_in:nn	5783	\clist_put_left:cV	5619, 5718
\clist_if_in:NnF	5739, 5797, 5798	\clist_put_left:cx	5619, 5720
\clist_if_in:nnF	5802	\clist_put_left:Nn	
\clist_if_in:NnT	5795, 5796	... 114, 5619, 5619, 5628, 5629, 5713	
\clist_if_in:nnT	5801	\clist_put_left:No	5619, 5715
\clist_if_in:NnTF	116, 5779, 5799, 5800	\clist_put_left:Nv	5619, 5714
\clist_if_in:nnTF	5779, 5803, 9677	\clist_put_left:Nx	5619, 5716
\clist_if_in:NoTF	5779	\clist_put_right:cn	5632
\clist_if_in:noTF	5779	\clist_put_right:co	5632
\clist_if_in:NvTF	5779	\clist_put_right:cV	5632
\clist_if_in:nvTF	5779	\clist_put_right:cx	5632
\clist_item:cn	14810	\clist_put_right:Nn	
\clist_item:Nn	194, 14810, 14810, 14839	... 114, 5632, 5632, 5641, 5642, 5740	
\clist_item:nn	14840, 14840	\clist_put_right:No	5632
\clist_map_break:		\clist_put_right:Nv	5632
... 117, 5810, 5815, 5824, 5828,		\clist_put_right:Nx	5632, 8682
5843, 5861, 5877, 5877, 5878, 5880		\clist_remove_all:cn	5746
\clist_map_break:n	118, 5877, 5879, 8621	\clist_remove_all:Nn	115, 5746, 5746, 5775
\clist_map_function:cN	5804	\clist_remove_duplicates:c	5730
\clist_map_function:NN	116, 5804,	\clist_remove_duplicates:N	
5804, 5819, 5886, 5939, 15263, 15273		... 115, 5730, 5730, 5744	
\clist_map_function:nNclist_set:c	152
. 5820, 5820, 8337, 8786, 15268, 15278		\clist_set:cn	5613
\clist_map_inline:cn	5834	\clist_set:co	5613
\clist_map_inline:Nn	116, 5737, 5834,	\clist_set:cV	5613
5834, 5850, 5852, 8616, 8966, 8981		\clist_set:cx	5613
\clist_map_inline:nn	5834, 5847, 8293, 8352	.clist_set:N	152
\clist_map_variable:cNn	5853	\clist_set:Nn	114,
\clist_map_variable:NNn		5613, 5613, 5617, 5620, 5622, 5633,	
... 117, 5853, 5853, 5867, 5876		5635, 5785, 5849, 5866, 5943, 8320	
\clist_map_variable:nNn	5853, 5864	\clist_set:No	5613
\clist_new:c	5551, 5552	\clist_set:Nv	5613
\clist_new:N	113, 5551, 5551,	\clist_set:Nx	5613
5729, 5947, 5948, 5949, 5950, 8165		\clist_set_eq:cc	5561, 5564
\clist_pop:cN	5656	\clist_set_eq:cN	5561, 5563
\clist_pop:cnTF	5682	\clist_set_eq:Nc	5561, 5562
\clist_pop:NN	120, 5656, 5656, 5680, 5694	\clist_set_eq:NN	114, 5561, 5561, 5731
\clist_pop:NNF	5708	\clist_set_from_seq:cc	14870
\clist_pop:NNT	5707	\clist_set_from_seq:cN	14870
\clist_pop:NNTF	120, 5682, 5709	\clist_set_from_seq:Nc	14870
\clist_push:cn	5713, 5717	\clist_set_from_seq:NN	
\clist_push:co	5713, 5719	... 194, 14870, 14870, 14894, 14895	
\clist_push:cV	5713, 5718	\clist_show:c	5936
\clist_push:cx	5713, 5720	\clist_show:N	121, 5936, 5936, 5944, 5946

\clist_show:n	121, 5936, 5941	\coffin_set_eq:cc	6624
\clist_use:cn	5906	\coffin_set_eq:cN	6624
\clist_use:cnnn	5906	\coffin_set_eq:Nc	6624
\clist_use:Nn	119, 5906, 5933, 5935	\coffin_set_eq:NN	136, 6624, 6624, 6632, 6888, 6907, 6936, 7143
\clist_use:Nnnn	119, 5906, 5906, 5926, 5934	\coffin_set_horizontal_pole:cnn	6676
\closein	368	\coffin_set_horizontal_pole:Nnn	137, 6676, 6676, 6700
\closeout	363	\coffin_set_vertical_pole:cnn	6676
\clubpenalties	676	\coffin_set_vertical_pole:Nnn	137, 6676, 6687, 6701
\clubpenalty	503	\coffin_show_structure:c	7232
.code:n	152	\coffin_show_structure:N	139, 7232, 7232, 7244
\coffin_attach:cnnncnnn	6891	\coffin_typeset:cnnnn	7007
\coffin_attach:cnnNnnnn	6891	\coffin_typeset:Nnnnn	138, 7007, 7007, 7014
\coffin_attach:Nnnncnnn	6891	\coffin_wd:c	6639, 6644
\coffin_attach:NnnNnnnn	138, 6891, 6891, 6918	\coffin_wd:N	139, 6639, 6643, 15069, 15102
\coffin_attach_mark:NnnNnnnn	6891, 6909, 7079, 7100, 7116	\color	7075, 7087, 7130, 7170
\coffin_clear:c	6506	\color_ensure_current:	140, 6533, 6577, 6600, 7284, 7285, 7289, 7294, 7299
\coffin_clear:N	136, 6506, 6506, 6514	\color_group_begin:	140, 6532, 6554, 6577, 6600, 7278, 7278
\coffin_display_handles:cn	7122	\color_group_end:	140, 6535, 6556, 6580, 6603, 7278, 7279
\coffin_display_handles:Nn	139, 7122, 7122, 7207	\columnwidth	6552, 6598
\coffin_dp:c	6639, 6640	\copy	560
\coffin_dp:N	138, 6639, 6639, 15072, 15098	acos	183
\coffin_ht:c	6639, 6642	acosd	183
\coffin_ht:N	139, 6639, 6641, 15072, 15098	acot	183
\coffin_if_exist:cTF	6483	acotd	184
\coffin_if_exist:N	6483	\count	611
\coffin_if_exist:Nf	6495	\countdef	310
\coffin_if_exist:NT	6494	\cr	335
\coffin_if_exist:NTF	136, 6483, 6496, 6499	\crr	336
\coffin_if_exist_p:c	6483	\cs:w	17, 764, 766, 781, 783, 842, 1103, 1131, 1339, 1388, 1604, 1643, 1657, 1659, 1663, 1664, 1665, 1700, 1706, 1726, 1728, 1733, 1740, 1741, 1803, 1807, 1837, 2037, 2260, 2262, 3317, 9505, 9534, 10066, 10148, 10331, 10363, 10579, 10675, 10778, 10810, 12473, 12741, 13255
\coffin_if_exist_p:N	136, 6483, 6493	\cs_end:	17, 764, 767, 781, 783, 787, 842, 1097, 1103, 1125, 1131, 1272, 1339, 1388, 1604, 1643, 1657, 1659, 1663, 1664, 1665, 1700, 1706, 1726, 1728, 1733, 1740, 1741, 1803, 1807, 1837, 2037, 2257, 2263, 2264, 2265, 2266, 2268, 2270, 2272, 2274, 2276, 2278,
\coffin_join:cnnncnnn	6854		
\coffin_join:cnnNnnnn	6854		
\coffin_join:Nnnncnnn	6854		
\coffin_join:NnnNnnnn	138, 6854, 6854, 6890		
\coffin_mark_handle:cnnn	7067		
\coffin_mark_handle:Nnnn	139, 7067, 7067, 7121		
\coffin_new:c	6515		
\coffin_new:N	136, 6515, 6515, 6525, 6633, 6635, 6636, 6637, 6638, 7015, 7016, 7017		
\coffin_resize:cnn	15066		
\coffin_resize:Nnn	194, 15066, 15066, 15077		
\coffin_rotate:cn	14924		
\coffin_rotate:Nn	194, 14924, 14924, 14958		
\coffin_scale:cnn	15092		
\coffin_scale:Nnn	195, 15092, 15092, 15106		

2280, 3317, 9505, 9542, 9663, 10066,
 10148, 10335, 10367, 10579, 10675,
 10778, 10813, 12473, 12741, 13414
 \cs_generate_from_arg_count:cNnn ...
 1321, 1330
 \cs_generate_from_arg_count:Ncnn ...
 1321, 1332
 \cs_generate_from_arg_count:NNnn ...
 15, 1321, 1321, 1331, 1333, 1351
 \cs_generate_variant:Nn ... 28, 1851,
 1851, 2040, 2041, 2042, 2043, 2044,
 2045, 2046, 2047, 2048, 2049, 2050,
 2051, 2064, 2073, 2074, 2075, 2076,
 2089, 2090, 2099, 2100, 2101, 2102,
 2118, 2217, 2218, 2223, 2224, 2388,
 2389, 2419, 2420, 2421, 2422, 2441,
 2442, 2443, 2444, 3238, 3259, 3271,
 3272, 3277, 3278, 3280, 3281, 3283,
 3284, 3295, 3296, 3297, 3298, 3307,
 3308, 3309, 3310, 3314, 3315, 3759,
 3923, 3929, 3932, 3933, 3938, 3939,
 3945, 3946, 3948, 3949, 3951, 3952,
 3956, 3957, 3961, 3962, 4144, 4146,
 4162, 4168, 4171, 4172, 4177, 4178,
 4184, 4185, 4187, 4188, 4190, 4191,
 4195, 4196, 4200, 4201, 4226, 4233,
 4234, 4236, 4252, 4258, 4262, 4263,
 4268, 4269, 4275, 4276, 4278, 4279,
 4281, 4282, 4286, 4287, 4291, 4292,
 4296, 4298, 4321, 4332, 4333, 4338,
 4339, 4344, 4345, 4358, 4359, 4395,
 4396, 4397, 4398, 4399, 4400, 4417,
 4418, 4419, 4420, 4421, 4422, 4423,
 4424, 4441, 4442, 4443, 4444, 4445,
 4446, 4447, 4448, 4489, 4490, 4491,
 4492, 4505, 4506, 4507, 4508, 4551,
 4552, 4557, 4558, 4561, 4562, 4563,
 4564, 4565, 4566, 4567, 4568, 4577,
 4578, 4579, 4580, 4589, 4590, 4591,
 4592, 4612, 4613, 4614, 4615, 4634,
 4635, 4636, 4643, 4644, 4645, 4680,
 4681, 4682, 4683, 4698, 4710, 4726,
 4733, 4739, 4751, 4752, 4775, 4776,
 4874, 4885, 4886, 4905, 4915, 4947,
 4948, 4949, 4950, 5047, 5077, 5080,
 5083, 5086, 5089, 5133, 5134, 5141,
 5142, 5162, 5163, 5164, 5165, 5183,
 5184, 5185, 5186, 5203, 5204, 5229,
 5230, 5239, 5240, 5241, 5242, 5262,
 5263, 5264, 5265, 5266, 5267, 5296,
 5309, 5310, 5329, 5356, 5357, 5362,
 5363, 5364, 5365, 5366, 5367, 5376,
 5377, 5378, 5379, 5380, 5381, 5382,
 5383, 5384, 5385, 5386, 5387, 5405,
 5434, 5445, 5446, 5456, 5480, 5497,
 5535, 5582, 5583, 5617, 5618, 5628,
 5629, 5630, 5631, 5641, 5642, 5643,
 5644, 5655, 5680, 5681, 5691, 5692,
 5693, 5707, 5708, 5709, 5710, 5711,
 5712, 5744, 5745, 5775, 5776, 5795,
 5796, 5797, 5798, 5799, 5800, 5801,
 5802, 5803, 5819, 5852, 5876, 5889,
 5926, 5935, 5946, 5969, 5972, 5975,
 5978, 5981, 6018, 6019, 6020, 6021,
 6028, 6029, 6048, 6049, 6050, 6051,
 6072, 6073, 6074, 6075, 6076, 6077,
 6097, 6099, 6101, 6103, 6126, 6127,
 6135, 6136, 6137, 6138, 6161, 6162,
 6163, 6164, 6165, 6166, 6167, 6168,
 6178, 6179, 6180, 6181, 6182, 6183,
 6198, 6199, 6214, 6224, 6240, 6245,
 6246, 6251, 6252, 6257, 6258, 6263,
 6264, 6270, 6271, 6272, 6279, 6280,
 6281, 6284, 6285, 6301, 6302, 6303,
 6304, 6305, 6306, 6307, 6308, 6311,
 6312, 6313, 6314, 6319, 6320, 6328,
 6331, 6334, 6346, 6364, 6365, 6370,
 6371, 6376, 6377, 6395, 6396, 6406,
 6407, 6412, 6413, 6418, 6419, 6424,
 6425, 6440, 6441, 6493, 6494, 6495,
 6496, 6514, 6525, 6542, 6572, 6589,
 6623, 6632, 6700, 6701, 6702, 6890,
 6918, 7014, 7121, 7207, 7244, 7722,
 7999, 8186, 8262, 8278, 8310, 8379,
 8508, 8509, 8516, 8523, 8530, 8539,
 8548, 8774, 8930, 8947, 8998, 9001,
 9011, 9012, 9013, 9040, 9058, 9113,
 9116, 9133, 9151, 9157, 9160, 9163,
 9791, 9794, 13883, 13938, 14006,
 14039, 14043, 14090, 14127, 14134,
 14135, 14136, 14139, 14140, 14143,
 14144, 14149, 14150, 14157, 14158,
 14159, 14160, 14172, 14212, 14213,
 14214, 14215, 14224, 14225, 14226,
 14227, 14228, 14229, 14234, 14235,
 14278, 14279, 14290, 14291, 14316,
 14319, 14320, 14323, 14621, 14650,
 14667, 14689, 14721, 14764, 14808,
 14839, 14894, 14895, 14896, 14897,
 14900, 14958, 15077, 15106, 15181,

- 15182, 15198, 15211, 15236, 15258,
 15259, 15280, 15281, 15282, 15283,
 15284, 15285, 15305, 15306, 15470
 \cs_gset:cn 1383
 \cs_gset:cpn
 .. 1234, 1236, 4702, 5839, 7358, 7360
 \cs_gset:cpx 1234, 1237
 \cs_gset:cx 1383
 \cs_gset:Nn 14, 1334
 \cs_gset:Npn 12,
 827, 829, 1220, 1236, 5409, 6205, 15486
 \cs_gset:Npx
 ... 827, 831, 1221, 1237, 5414, 15487
 \cs_gset:Nx 1334
 \cs_gset_eq:cc ... 1252, 1259, 2084, 4353
 \cs_gset_eq:cN 1252, 1258, 1277,
 2083, 4351, 5419, 6202, 8146, 8148
 \cs_gset_eq:Nc
 .. 1252, 1257, 2082, 4352, 5424, 6210
 \cs_gset_eq:NN 16, 1252, 1256,
 1257, 1258, 1259, 1269, 1476, 1477,
 1478, 1479, 1480, 1481, 1482, 1483,
 1487, 1488, 1489, 1490, 1491, 1492,
 1493, 1494, 2070, 2072, 2081, 4319,
 4350, 5075, 5967, 9055, 9148, 15489
 \cs_gset_nopar:cn 1383
 \cs_gset_nopar:cpn 1226, 1230
 \cs_gset_nopar:cpx 1226, 1231
 \cs_gset_nopar:cx 1383
 \cs_gset_nopar:Nn 14, 1334
 \cs_gset_nopar:Npn 12, 827, 827,
 830, 834, 838, 1218, 1230, 3517, 7398
 \cs_gset_nopar:Npx 827, 828, 832,
 836, 840, 1219, 1231, 3524, 4325,
 4330, 4390, 4392, 4394, 4410, 4412,
 4414, 4416, 4434, 4436, 4438, 4440
 \cs_gset_nopar:Nx 1334
 \cs_gset_protected:cn 1383
 \cs_gset_protected:cpn 1246, 1248
 \cs_gset_protected:cpx 1246, 1249
 \cs_gset_protected:cx 1383
 \cs_gset_protected:Nn 15, 1334
 \cs_gset_protected:Npn
 12, 827, 837, 1224, 1248, 7329
 \cs_gset_protected:Npx 827, 839, 1225, 1249
 \cs_gset_protected:Nx 1334
 \cs_gset_protected_nopar:cn 1383
 \cs_gset_protected_nopar:cpn 1240, 1242
 \cs_gset_protected_nopar:cpx 1240, 1243
 \cs_gset_protected_nopar:cx 1383
 \cs_gset_protected_nopar:Nn ... 15, 1334
 \cs_gset_protected_nopar:Npn
 13, 827, 833, 1222, 1242
 \cs_gset_protected_nopar:Npx
 827, 835, 1223, 1243
 \cs_gset_protected_nopar:Nx 1334
 \cs_if_eq:ccF 1431
 \cs_if_eq:ccT 1430
 \cs_if_eq:ccTF 1415, 1429
 \cs_if_eq:cNF 1423
 \cs_if_eq:cNT 1422
 \cs_if_eq:cNTF 1415, 1421, 7564
 \cs_if_eq:NcF 1427
 \cs_if_eq:NcT 1426
 \cs_if_eq:NcTF 1415, 1425
 \cs_if_eq:NN 1415
 \cs_if_eq:NNF 1423, 1427, 1431
 \cs_if_eq:NNT 1422, 1426, 1430
 \cs_if_eq:NNTF . 21, 1415, 1421, 1425, 1429
 \cs_if_eq_p:cc 1415, 1428
 \cs_if_eq_p:cN 1415, 1420
 \cs_if_eq_p:Nc 1415, 1424
 \cs_if_eq_p:NN . 21, 1415, 1420, 1424, 1428
 \cs_if_exist:c
 1095, 2124, 3286, 3941, 4180, 4271,
 4361, 5144, 5585, 6129, 6266, 11131
 \cs_if_exist:cF 8765
 \cs_if_exist:cTF 1083,
 1148, 1150, 1152, 1154, 6487, 7316,
 8200, 8673, 8696, 8702, 8781, 10194
 \cs_if_exist:N
 1083, 2123, 3285, 3940, 4179, 4270,
 4360, 5143, 5584, 6128, 6265, 11130
 \cs_if_exist:NF 1202, 14255
 \cs_if_exist:NT 1474, 1485, 7298, 8885, 8903
 \cs_if_exist:NTF
 21, 1083, 1140, 1142, 1144,
 1146, 1434, 6485, 7292, 8008, 9072
 \cs_if_exist_p:c 1083
 \cs_if_exist_p:N 21, 1083
 \cs_if_exist_use:c 1139, 1153
 \cs_if_exist_use:cF ... 1149, 9675, 10155
 \cs_if_exist_use:cT 1151
 \cs_if_exist_use:cTF 1139, 1147
 \cs_if_exist_use:N 1139, 1145
 \cs_if_exist_use:NF 1141
 \cs_if_exist_use:NT 1143
 \cs_if_exist_use:NTF
 17, 17, 191, 1139, 1139
 \cs_if_free:c 1123

<code>\cs_if_free:cT</code>	2032	3343, 3349, 3360, 3389, 3394, 3399,
<code>\cs_if_free:cTF</code>	<u>1111</u> , 7595	3404, 3409, 3411, 3434, 3442, 3450,
<code>\cs_if_free:N</code>	1111	3456, 3462, 3470, 3478, 3484, 3490,
<code>\cs_if_free:NF</code>	1179, 1189	3504, 3538, 3539, 3553, 3559, 3591,
<code>\cs_if_free:NTF</code>	22, <u>1111</u> , 1993	3623, 3625, 3631, 3643, 3651, 3684,
<code>\cs_if_free_p:c</code>	<u>1111</u>	3686, 3688, 3690, 3695, 3700, 3705,
<code>\cs_if_free_p:N</code>	22, <u>1111</u>	3725, 3726, 3731, 3736, 3760, 3768,
<code>\cs_meaning:c</code>	<u>784</u> , 785, 795	3770, 3779, 3781, 3790, 3792, 3802,
<code>\cs_meaning:N</code>	16, <u>771</u> , <u>773</u> , 792, 1454	3811, 3813, 3815, 3831, 3840, 3874,
<code>\cs_new:cn</code>	<u>1383</u>	3876, 3963, 3968, 3986, 3994, 3996,
<code>\cs_new:cpn</code>	<u>1234</u> ,	4008, 4014, 4037, 4039, 4044, 4049,
	1238, 2163, 2165, 2170, 2264, 2265,	4054, 4059, 4061, 4124, 4126, 4128,
	2266, 2267, 2269, 2271, 2273, 2275,	4135, 4220, 4223, 4228, 4231, 4293,
	2277, 2279, 2281, 2286, 2287, 2288,	4485, 4534, 4545, 4593, 4646, 4647,
	2289, 2290, 2291, 2292, 2293, 2294,	4648, 4649, 4652, 4657, 4662, 4667,
	2295, 3355, 3367, 3369, 3371, 3373,	4672, 4674, 4685, 4693, 4732, 4734,
	3375, 3377, 3379, 4029, 4031, 4033,	4740, 4745, 4750, 4753, 4760, 4767,
	4035, 9550, 10025, 10078, 10574,	4769, 4779, 4791, 4799, 4805, 4811,
	10591, 10912, 10941, 11014, 11019,	4815, 4822, 4833, 4842, 4844, 4851,
	11024, 11029, 11306, 11415, 12850	4857, 4859, 4861, 4875, 4877, 4879,
<code>\cs_new:cpx</code>	<u>1234</u> , 1239	4887, 4892, 4894, 4906, 4908, 4917,
<code>\cs_new:cx</code>	<u>1383</u>	4923, 4925, 4931, 4977, 4986, 5031,
<code>\cs_new:Nn</code>	6, <u>13</u> , <u>1334</u>	5064, 5125, 5131, 5139, 5140, 5161,
<code>\cs_new:Npn</code> 1, <u>11</u> , <u>1210</u> , 1220, 1238, 1309,		5182, 5187, 5294, 5349, 5351, 5392,
	1311, 1458, 1508, 1516, 1521, 1526,	5394, 5399, 5447, 5455, 5457, 5481,
	1531, 1536, 1538, 1544, 1549, 1554,	5484, 5486, 5493, 5495, 5586, 5591,
	1559, 1564, 1566, 1572, 1576, 1585,	5592, 5599, 5679, 5772, 5774, 5804,
	1586, 1597, 1598, 1599, 1600, 1601,	5813, 5820, 5826, 5833, 5881, 5899,
	1602, 1603, 1605, 1607, 1619, 1625,	5906, 5927, 5928, 5931, 5933, 5960,
	1631, 1642, 1644, 1651, 1652, 1654,	5994, 6005, 6146, 6152, 6184, 6190,
	1656, 1658, 1660, 1667, 1669, 1674,	7477, 7478, 7479, 7480, 7481, 7482,
	1679, 1685, 1691, 1697, 1703, 1709,	7568, 7593, 7723, 7961, 7964, 7973,
	1716, 1723, 1730, 1737, 1772, 1773,	7978, 7983, 7988, 8028, 8029, 8033,
	1778, 1780, 1785, 1795, 1797, 1799,	8038, 8106, 8336, 8671, 8689, 9333,
	1800, 1802, 1804, 1810, 1816, 1818,	9340, 9383, 9384, 9385, 9386, 9387,
	1825, 1832, 1834, 1836, 1837, 1838,	9388, 9389, 9390, 9391, 9392, 9412,
	1840, 1845, 1919, 1935, 1940, 1949,	9413, 9414, 9420, 9426, 9435, 9437,
	1962, 1977, 2035, 2133, 2143, 2145,	9448, 9449, 9450, 9460, 9470, 9480,
	2147, 2149, 2153, 2172, 2190, 2196,	9490, 9500, 9503, 9513, 9514, 9519,
	2202, 2206, 2207, 2213, 2215, 2219,	9521, 9526, 9528, 9530, 9532, 9544,
	2221, 2225, 2233, 2238, 2246, 2252,	9546, 9548, 9574, 9576, 9578, 9579,
	2259, 2261, 2263, 2316, 2322, 2331,	9580, 9582, 9584, 9586, 9588, 9590,
	2336, 2358, 2364, 2372, 2379, 2390,	9598, 9599, 9615, 9624, 9631, 9633,
	2396, 2462, 2465, 2467, 2479, 2549,	9640, 9659, 9778, 9779, 9780, 9781,
	2555, 2561, 2567, 2698, 2753, 2771,	9782, 9783, 9784, 9792, 9833, 9835,
	2795, 2813, 2831, 2849, 2860, 2881,	9844, 9845, 9854, 9868, 9884, 9895,
	2900, 2907, 2908, 2916, 2925, 2934,	9904, 9911, 9922, 9930, 9941, 9946,
	2950, 3022, 3118, 3121, 3130, 3139,	9966, 9968, 9979, 9984, 9997, 10005,
	3165, 3167, 3173, 3191, 3199, 3207,	10006, 10016, 10021, 10043, 10064,
	3220, 3222, 3229, 3317, 3324, 3338,	10073, 10083, 10102, 10112, 10114,

10117, 10119, 10125, 10133, 10153,
 10164, 10184, 10192, 10207, 10222,
 10233, 10243, 10253, 10258, 10267,
 10284, 10297, 10302, 10308, 10310,
 10317, 10347, 10375, 10391, 10402,
 10407, 10425, 10443, 10454, 10469,
 10474, 10484, 10494, 10504, 10520,
 10567, 10602, 10614, 10670, 10679,
 10700, 10717, 10736, 10739, 10751,
 10760, 10763, 10773, 10774, 10781,
 10788, 10819, 10834, 10838, 10854,
 10872, 10874, 10888, 10914, 10928,
 10956, 10968, 10983, 10997, 11034,
 11039, 11044, 11055, 11070, 11081,
 11093, 11137, 11156, 11161, 11188,
 11189, 11197, 11209, 11215, 11221,
 11229, 11237, 11243, 11249, 11257,
 11265, 11276, 11300, 11302, 11304,
 11317, 11326, 11327, 11352, 11357,
 11364, 11365, 11373, 11384, 11391,
 11398, 11400, 11408, 11438, 11440,
 11450, 11470, 11479, 11493, 11501,
 11509, 11516, 11523, 11531, 11541,
 11555, 11566, 11567, 11573, 11590,
 11597, 11599, 11606, 11611, 11628,
 11629, 11630, 11648, 11654, 11664,
 11676, 11683, 11705, 11743, 11752,
 11771, 11773, 11775, 11784, 11795,
 11822, 11835, 11848, 11856, 11873,
 11890, 11897, 11905, 11915, 11916,
 11925, 11926, 11935, 11945, 11959,
 11974, 11975, 11980, 11988, 11989,
 12013, 12026, 12033, 12041, 12042,
 12043, 12046, 12054, 12060, 12062,
 12064, 12086, 12092, 12102, 12112,
 12122, 12135, 12146, 12151, 12158,
 12167, 12169, 12178, 12187, 12201,
 12203, 12205, 12218, 12228, 12233,
 12242, 12250, 12257, 12263, 12272,
 12274, 12286, 12291, 12299, 12304,
 12314, 12320, 12326, 12333, 12340,
 12342, 12347, 12349, 12354, 12355,
 12369, 12379, 12391, 12396, 12403,
 12422, 12436, 12450, 12470, 12483,
 12484, 12489, 12502, 12507, 12515,
 12520, 12530, 12542, 12571, 12572,
 12573, 12575, 12577, 12579, 12593,
 12599, 12608, 12627, 12633, 12643,
 12662, 12670, 12709, 12711, 12720,
 12722, 12736, 12738, 12747, 12749,
 12765, 12781, 12797, 12813, 12829,
 12845, 12875, 12894, 12922, 12938,
 12948, 12959, 12980, 12995, 13000,
 13005, 13007, 13021, 13023, 13038,
 13046, 13054, 13078, 13093, 13108,
 13123, 13138, 13153, 13168, 13183,
 13191, 13205, 13207, 13213, 13225,
 13233, 13240, 13416, 13424, 13435,
 13436, 13438, 13439, 13450, 13457,
 13459, 13465, 13486, 13493, 13500,
 13515, 13554, 13567, 13608, 13622,
 13641, 13643, 13660, 13675, 13687,
 13694, 13699, 13701, 13710, 13723,
 13726, 13747, 13760, 13775, 13793,
 13808, 13818, 13827, 13840, 13856,
 13873, 13878, 13879, 13880, 13881,
 13894, 13921, 13934, 13936, 13944,
 13970, 13995, 14004, 14005, 14012,
 14022, 14042, 14049, 14054, 14061,
 14075, 14080, 14082, 14092, 14094,
 14096, 14098, 14187, 14194, 14305,
 14317, 14810, 14818, 14833, 14840,
 14848, 14850, 14864, 14869, 14886,
 14893, 14908, 14914, 15184, 15190,
 15199, 15205, 15213, 15214, 15229,
 15237, 15239, 15245, 15251, 15298,
 15300, 15329, 15337, 15340, 15360,
 15373, 15379, 15384, 15396, 15397,
 15398, 15410, 15418, 15426, 15433,
 15434, 15442, 15447, 15462, 15476
 \cs_new:Npx . . . 1210, 1221, 1239, 5890,
 5900, 7949, 9205, 14037, 14040, 14110
 \cs_new:Nx 1334
 \cs_new_eq:cc 985, 1252, 1267
 \cs_new_eq:cN 1252, 1265, 10566
 \cs_new_eq:Nc 1252, 1266
 \cs_new_eq:NN . . . 15, 1252, 1260, 1265,
 1266, 1267, 1462, 1463, 1464, 1465,
 1466, 1467, 1468, 1469, 1470, 1471,
 1472, 1473, 1574, 1575, 1584, 1587,
 1588, 1613, 2052, 2063, 2077, 2078,
 2079, 2080, 2081, 2082, 2083, 2084,
 2458, 2571, 2572, 2573, 2943, 2944,
 2945, 3156, 3157, 3158, 3159, 3160,
 3262, 3266, 3316, 3417, 3877, 3878,
 3904, 3913, 3914, 3915, 4067, 4143,
 4145, 4225, 4227, 4230, 4235, 4295,
 4297, 4307, 4346, 4347, 4348, 4349,
 4350, 4351, 4352, 4353, 4684, 4731,
 5054, 5055, 5090, 5091, 5092, 5093,

- 5094, 5095, 5096, 5097, 5498, 5499,
 5500, 5501, 5502, 5503, 5504, 5505,
 5506, 5507, 5508, 5509, 5510, 5511,
 5512, 5513, 5514, 5515, 5516, 5517,
 5518, 5519, 5520, 5521, 5522, 5523,
 5548, 5551, 5552, 5553, 5554, 5555,
 5556, 5557, 5558, 5559, 5560, 5561,
 5562, 5563, 5564, 5565, 5566, 5567,
 5568, 5713, 5714, 5715, 5716, 5717,
 5718, 5719, 5720, 5721, 5722, 5723,
 5724, 5725, 5726, 5727, 5728, 5982,
 5983, 5984, 5985, 5986, 5987, 5988,
 5989, 6267, 6268, 6269, 6282, 6283,
 6294, 6295, 6296, 6378, 6379, 6380,
 6381, 6382, 6383, 6384, 6385, 6393,
 6394, 6431, 6432, 6433, 6434, 6435,
 6436, 6437, 6438, 6439, 6639, 6640,
 6641, 6642, 6643, 6644, 7278, 8986,
 8997, 9069, 9099, 9100, 9112, 9154,
 9169, 9509, 9669, 9670, 9671, 9672,
 9867, 9920, 9921, 14089, 14091,
 14126, 14137, 14138, 14301, 14302,
 14422, 14423, 14424, 14425, 15328
 \cs_new_nopar:cn 1383
 \cs_new_nopar:cpn 1226,
 1232, 2181, 2182, 2183, 2184, 2185,
 2186, 2187, 2188, 10632, 10643,
 10662, 10712, 10714, 11697, 11807
 \cs_new_nopar:cpx 1226, 1233, 2023, 11410
 \cs_new_nopar:cx 1383
 \cs_new_nopar:Nn 13, 1334
 \cs_new_nopar:Npn 11, 1210, 1218, 1232,
 1319, 1420, 1421, 1422, 1423, 1424,
 1425, 1426, 1427, 1428, 1429, 1430,
 1431, 1496, 1745, 1746, 1747, 1748,
 1749, 1750, 1751, 1752, 1753, 1760,
 1761, 1762, 1763, 1764, 1827, 1828,
 1829, 1830, 2304, 2306, 2947, 2948,
 2949, 3000, 3008, 3010, 3012, 3032,
 3710, 3711, 3712, 3713, 3714, 3715,
 3716, 3717, 3718, 3719, 3720, 3721,
 3722, 3723, 3724, 4535, 4540, 4691,
 4727, 4729, 4907, 4916, 5260, 5388,
 5390, 5877, 5879, 6215, 6217, 7397,
 9168, 9331, 9789, 10042, 10688,
 10690, 10692, 10694, 10696, 10698,
 10702, 10704, 11324, 12044, 12045,
 13476, 13598, 13603, 13884, 13939,
 14007, 14044, 15145, 15147, 15469
 \cs_new_nopar:Npx
 1210, 1219, 1233, 1878, 13252
 \cs_new_nopar:Nx 1334
 \cs_new_protected:cn 1383
 \cs_new_protected:cpn
 1246, 1250, 7492, 7493,
 7737, 8380, 8382, 8384, 8386, 8388,
 8390, 8392, 8394, 8398, 8400, 8402,
 8404, 8406, 8408, 8410, 8412, 8414,
 8416, 8418, 8420, 8422, 8424, 8426,
 8428, 8430, 8432, 8434, 8436, 8438,
 8440, 8442, 8444, 8446, 8448, 8450,
 8452, 8454, 8456, 8458, 8460, 8464,
 8466, 8468, 8470, 8472, 8474, 8476,
 8478, 8480, 8482, 8484, 8486, 8488,
 8490, 8492, 8494, 8775, 8777, 8812
 \cs_new_protected:cpx
1246, 1251, 7502, 7504, 7506, 7508,
 7510, 7519, 7521, 7523, 7746, 7748,
 7750, 7752, 7754, 7763, 7765, 7767
 \cs_new_protected:cx 1383
 \cs_new_protected:Nn 13, 1334
 \cs_new_protected:Npn 11, 1210,
 1224, 1250, 1252, 1260, 1268, 1270,
 1321, 1342, 1347, 1432, 1449, 1614,
 1790, 1851, 1869, 1882, 1884, 1890,
 1899, 1989, 2015, 2028, 2063, 2065,
 2067, 2069, 2071, 2085, 2087, 2103,
 2112, 2351, 2449, 2477, 2481, 2483,
 2485, 2487, 2489, 2491, 2493, 2495,
 2497, 2499, 2501, 2503, 2505, 2507,
 2509, 2511, 2513, 2515, 2517, 2519,
 2521, 2523, 2525, 2527, 2529, 2531,
 2533, 2535, 2537, 2539, 2541, 2543,
 2545, 2547, 2551, 2553, 2557, 2559,
 2563, 2565, 2569, 2571, 2955, 2961,
 2978, 2980, 2982, 2996, 2998, 3232,
 3239, 3269, 3270, 3273, 3275, 3279,
 3282, 3287, 3289, 3299, 3301, 3311,
 3520, 3532, 3879, 3917, 3924, 3930,
 3931, 3934, 3936, 3942, 3944, 3947,
 3950, 3953, 3955, 3958, 3960, 4147,
 4156, 4163, 4169, 4170, 4173, 4175,
 4181, 4183, 4186, 4189, 4192, 4194,
 4197, 4199, 4237, 4246, 4253, 4259,
 4261, 4264, 4266, 4272, 4274, 4277,
 4280, 4283, 4285, 4288, 4290, 4299,
 4316, 4322, 4327, 4334, 4336, 4340,
 4342, 4354, 4356, 4383, 4385, 4387,
 4389, 4391, 4393, 4401, 4403, 4405,
 4407, 4409, 4411, 4413, 4415, 4425,

4427, 4429, 4431, 4433, 4435, 4437,
 4439, 4465, 4493, 4495, 4509, 4547,
 4549, 4553, 4555, 4699, 4708, 4711,
 4719, 4771, 4773, 4881, 4883, 5038,
 5048, 5072, 5078, 5081, 5084, 5087,
 5102, 5135, 5137, 5145, 5153, 5166,
 5174, 5189, 5191, 5193, 5205, 5207,
 5209, 5268, 5276, 5286, 5301, 5303,
 5311, 5313, 5324, 5334, 5406, 5411,
 5416, 5428, 5435, 5530, 5550, 5573,
 5613, 5615, 5623, 5636, 5645, 5653,
 5660, 5668, 5698, 5730, 5732, 5734,
 5746, 5748, 5750, 5788, 5834, 5847,
 5853, 5864, 5869, 5936, 5941, 5964,
 5970, 5973, 5976, 5979, 5995, 5997,
 6006, 6012, 6022, 6030, 6039, 6080,
 6109, 6200, 6219, 6234, 6241, 6243,
 6247, 6249, 6253, 6255, 6259, 6261,
 6273, 6275, 6277, 6286, 6288, 6290,
 6292, 6315, 6317, 6326, 6332, 6335,
 6347, 6361, 6362, 6363, 6366, 6368,
 6372, 6374, 6386, 6388, 6389, 6391,
 6397, 6398, 6399, 6401, 6403, 6405,
 6408, 6410, 6414, 6416, 6420, 6422,
 6426, 6442, 6497, 6506, 6515, 6526,
 6543, 6573, 6590, 6624, 6645, 6655,
 6662, 6669, 6676, 6687, 6698, 6703,
 6714, 6748, 6763, 6841, 6854, 6891,
 6909, 6919, 6938, 6943, 6957, 6962,
 6972, 6983, 6995, 7007, 7067, 7114,
 7122, 7151, 7200, 7208, 7232, 7319,
 7340, 7345, 7347, 7354, 7356, 7363,
 7404, 7416, 7421, 7438, 7462, 7468,
 7562, 7603, 7624, 7637, 7642, 7668,
 7679, 7681, 7698, 7724, 7726, 7728,
 7730, 7993, 8000, 8002, 8004, 8006,
 8018, 8020, 8064, 8078, 8088, 8108,
 8114, 8116, 8137, 8143, 8178, 8180,
 8187, 8192, 8197, 8210, 8220, 8230,
 8245, 8247, 8263, 8289, 8305, 8311,
 8316, 8325, 8327, 8329, 8334, 8348,
 8364, 8373, 8502, 8510, 8517, 8524,
 8531, 8540, 8549, 8554, 8559, 8637,
 8706, 8763, 8779, 8793, 8851, 8871,
 8873, 8881, 8914, 8924, 8931, 8948,
 8950, 8955, 8997, 8999, 9002, 9014,
 9024, 9041, 9047, 9061, 9086, 9088,
 9112, 9114, 9117, 9134, 9140, 9155,
 9158, 9161, 9204, 9211, 9253, 9298,
 9394, 9657, 9673, 9693, 9727, 9753,
 9761, 9763, 13653, 14125, 14128,
 14130, 14132, 14141, 14142, 14145,
 14147, 14155, 14161, 14170, 14210,
 14222, 14242, 14247, 14253, 14266,
 14280, 14285, 14310, 14321, 14329,
 14360, 14399, 14487, 14499, 14533,
 14544, 14555, 14566, 14577, 14588,
 14601, 14622, 14631, 14651, 14668,
 14690, 14719, 14722, 14765, 14870,
 14872, 14874, 14898, 14924, 14959,
 14971, 14977, 14983, 14995, 15014,
 15023, 15040, 15045, 15053, 15066,
 15078, 15092, 15107, 15114, 15120,
 15129, 15136, 15159, 15166, 15177,
 15179, 15260, 15265, 15270, 15275,
 15291, 15311, 15321, 15517, 15521
 \cs_new_protected:Npx .. [1210](#), [1225](#), [1251](#)
 \cs_new_protected:Nx [1334](#)
 \cs_new_protected_nopar:cn [1383](#)
 \cs_new_protected_nopar:cpn
 .. [1240](#), [1244](#), [8396](#), [8462](#), [8496](#), [8498](#)
 \cs_new_protected_nopar:cpx
 .. [1240](#), [1245](#), [1336](#), [1385](#), [2020](#), [3060](#)
 \cs_new_protected_nopar:cx [1383](#)
 \cs_new_protected_nopar:Nn [13](#), [1334](#)
 \cs_new_protected_nopar:Npn [11](#),
[1210](#), [1222](#), [1227](#), [1244](#), [1253](#), [1254](#),
[1255](#), [1256](#), [1257](#), [1258](#), [1259](#), [1265](#),
[1266](#), [1267](#), [1330](#), [1332](#), [1441](#), [1460](#),
[1744](#), [1754](#), [1755](#), [1756](#), [1757](#), [1758](#),
[1759](#), [1765](#), [1766](#), [1767](#), [1768](#), [1769](#),
[1770](#), [1771](#), [1831](#), [2308](#), [2951](#), [2953](#),
[3041](#), [3291](#), [3293](#), [3303](#), [3305](#), [3313](#),
[3318](#), [3513](#), [4459](#), [4461](#), [4463](#), [4497](#),
[4499](#), [4501](#), [4503](#), [4631](#), [4632](#), [4633](#),
[4717](#), [5098](#), [5100](#), [5297](#), [5299](#), [5330](#),
[5332](#), [5422](#), [5569](#), [5571](#), [5619](#), [5621](#),
[5632](#), [5634](#), [5656](#), [5658](#), [6078](#), [6079](#),
[6105](#), [6107](#), [6329](#), [6588](#), [6622](#), [7279](#),
[7285](#), [7289](#), [7623](#), [7677](#), [8279](#), [8338](#),
[8500](#), [8568](#), [8592](#), [8611](#), [8657](#), [8659](#),
[8680](#), [8960](#), [9059](#), [9152](#), [9165](#), [9167](#),
[9260](#), [9270](#), [9286](#), [9305](#), [9319](#), [9325](#),
[9687](#), [9689](#), [9691](#), [9721](#), [9723](#), [9725](#),
[9747](#), [9749](#), [9751](#), [9755](#), [9757](#), [9759](#),
[14151](#), [14152](#), [14153](#), [14154](#), [14196](#),
[14197](#), [14216](#), [14217](#), [14218](#), [14219](#),
[14220](#), [14221](#), [14232](#), [14233](#), [14259](#),
[14260](#), [14261](#), [14262](#), [14264](#), [14370](#),
[15034](#), [15149](#), [15151](#), [15153](#), [15286](#),

- 15287, 15289, 15307, 15309, 15317,
15319, 15500, 15528, 15530, 15532
- \cs_new_protected_nopar:Npx
..... [1210](#), [1223](#), [1245](#),
[1872](#), [1876](#), [2019](#), [9313](#), [14198](#), [14204](#)
- \cs_new_protected_nopar:Nx [1334](#)
- \cs_set:cn [1383](#)
- \cs_set:cpn
.. [1234](#), [1234](#), [7349](#), [7351](#), [8308](#), [9766](#)
- \cs_set:cpx [1234](#), [1235](#)
- \cs_set:cx [1383](#)
- \cs_set:Nn [14](#), [1334](#)
- \cs_set:Npn [3](#), [11](#), [813](#), [815](#),
[842](#), [848](#), [849](#), [850](#), [851](#), [852](#), [853](#),
[854](#), [855](#), [856](#), [857](#), [858](#), [859](#), [860](#),
[861](#), [862](#), [863](#), [864](#), [865](#), [866](#), [867](#),
[868](#), [869](#), [870](#), [871](#), [872](#), [873](#), [874](#),
[875](#), [876](#), [1029](#), [1034](#), [1039](#), [1044](#),
[1057](#), [1058](#), [1066](#), [1074](#), [1076](#), [1079](#),
[1081](#), [1139](#), [1141](#), [1143](#), [1145](#), [1147](#),
[1149](#), [1151](#), [1153](#), [1210](#), [1226](#), [1234](#),
[1334](#), [1383](#), [3052](#), [3058](#), [3162](#), [3175](#),
[3183](#), [3970](#), [3978](#), [4068](#), [4076](#), [4084](#),
[4090](#), [4096](#), [4104](#), [4112](#), [4118](#), [4639](#),
[4777](#), [5752](#), [5790](#), [5999](#), [9696](#), [9704](#),
[9713](#), [9730](#), [9738](#), [15161](#), [15474](#), [15484](#)
- \cs_set:Npx ... [813](#), [817](#), [1235](#), [4518](#), [15485](#)
- \cs_set:Nx [1334](#)
- \cs_set_eq:cc . [983](#), [1252](#), [1255](#), [2080](#), [4349](#)
- \cs_set_eq:cN [1252](#), [1253](#), [2079](#), [4347](#), [9658](#)
- \cs_set_eq:Nc [1252](#), [1254](#), [2078](#), [4348](#)
- \cs_set_eq:NN [16](#),
[1252](#), [1252](#), [1253](#), [1254](#), [1255](#), [1256](#),
[1263](#), [1872](#), [1887](#), [2019](#), [2066](#), [2068](#),
[2077](#), [2632](#), [2959](#), [2963](#), [2984](#), [2986](#),
[3063](#), [4346](#), [5336](#), [5337](#), [5347](#), [9221](#),
[9222](#), [9223](#), [15293](#), [15294](#), [15296](#), [15488](#)
- \cs_set_nopar:cn [1383](#)
- \cs_set_nopar:cpn [1226](#), [1228](#)
- \cs_set_nopar:cpx [1226](#), [1229](#)
- \cs_set_nopar:cx [1383](#)
- \cs_set_nopar:Nn [14](#), [1334](#)
- \cs_set_nopar:Npn [12](#), [813](#),
[813](#), [815](#), [816](#), [817](#), [819](#), [820](#), [821](#),
[824](#), [841](#), [877](#), [879](#), [1051](#), [1175](#), [1228](#)
- \cs_set_nopar:Npx
.... [813](#), [814](#), [818](#), [822](#), [826](#), [845](#),
[1229](#), [1616](#), [1792](#), [2965](#), [2970](#), [2987](#),
[2988](#), [4384](#), [4386](#), [4388](#), [4402](#), [4404](#),
[4406](#), [4408](#), [4426](#), [4428](#), [4430](#), [4432](#),
[8855](#), [9215](#), [9216](#), [9217](#), [9218](#), [9219](#)
- \cs_set_nopar:Nx [1334](#)
- \cs_set_protected:cn [1383](#)
- \cs_set_protected:cpn [1246](#), [1246](#)
- \cs_set_protected:cpx [1246](#), [1247](#)
- \cs_set_protected:cx [1383](#)
- \cs_set_protected:Nn [14](#), [1334](#)
- \cs_set_protected:Npn [12](#),
[813](#), [823](#), [843](#), [889](#), [902](#), [907](#), [919](#),
[934](#), [951](#), [967](#), [972](#), [977](#), [986](#), [998](#),
[1012](#), [1159](#), [1171](#), [1173](#), [1177](#), [1187](#),
[1200](#), [1212](#), [1246](#), [1279](#), [1300](#), [4212](#),
[5248](#), [6601](#), [7489](#), [7733](#), [7735](#), [10023](#),
[10572](#), [10630](#), [10641](#), [10886](#), [14230](#),
[14343](#), [14352](#), [14380](#), [14394](#), [14408](#)
- \cs_set_protected:Npx ... [813](#), [825](#), [1247](#)
- \cs_set_protected:Nx [1334](#)
- \cs_set_protected_nopar:cn [1383](#)
- \cs_set_protected_nopar:cpn . [1240](#), [1240](#)
- \cs_set_protected_nopar:cpx . [1240](#), [1241](#)
- \cs_set_protected_nopar:cx [1383](#)
- \cs_set_protected_nopar:Nn [14](#), [1334](#)
- \cs_set_protected_nopar:Npn [12](#), [265](#),
[813](#), [819](#), [823](#), [825](#), [829](#), [831](#), [833](#),
[835](#), [837](#), [839](#), [881](#), [883](#), [885](#), [887](#),
[894](#), [896](#), [898](#), [900](#), [982](#), [984](#), [1155](#),
[1157](#), [1198](#), [1208](#), [1240](#), [6578](#), [7294](#),
[7299](#), [7448](#), [9164](#), [9166](#), [14385](#), [14395](#)
- \cs_set_protected_nopar:Npx
..... [251](#), [813](#), [821](#), [1241](#), [7610](#)
- \cs_set_protected_nopar:Nx [1334](#)
- \cs_show:c [1443](#), [1460](#), [8707](#)
- \cs_show:N [16](#), [1443](#), [1449](#), [1461](#), [5041](#)
- \cs_to_str:N
.. [4](#), [18](#), [1051](#), [1051](#), [1070](#), [2320](#), [9169](#)
- \cs_undefine:c [1268](#), [1270](#)
- \cs_undefine:N
.... [16](#), [1268](#), [1268](#), [7771](#), [7772](#), [7773](#)
- acsc [183](#)
- acscd [183](#)
- \csname [14](#), [33](#), [36](#),
[63](#), [69](#), [73](#), [122](#), [135](#), [138](#), [174](#), [177](#),
[183](#), [191](#), [196](#), [198](#), [208](#), [211](#), [221](#), [398](#)
- \currentgrouplevel [650](#)
- \currentgroupstype [651](#)
- \currentifbranch [647](#)
- \currentiflevel [646](#)
- \currentifttype [648](#)

D

- dd 184
- nd 184
- \day 606
- \deadcycles 540
- \def 55, 57, 140, 152,
159, 164, 210, 220, 247, 280, 294, 305
- .default:n 153
- .default:o 153
- .default:V 153
- .default:x 153
- \defaultthyphenchar 590
- \defaultskewchar 591
- \delcode 621
- \delimiter 415
- \delimiterfactor 464
- \delimitershortfall 463
- \detokenize 33,
36, 122, 135, 138, 174, 177, 183,
192, 197, 199, 205, 208, 211, 221, 638
- \dim_abs:n 77, 3963, 3963, 14624
- \dim_add:cn 3953
- \dim_add:Nn 77, 3953, 3953, 3955, 3956
- \dim_case:nn 4039, 4054
- \dim_case:nnF 4049, 4307
- \dim_case:nnn 4307, 4307
- \dim_case:nnT 4044
- \dim_case:nnTF 80, 4039, 4039
- \dim_compare:n 4003
- \dim_compare:nF 4078, 4093
- \dim_compare:nNn 3998
- \dim_compare:nNnF 4106, 4121
- \dim_compare:nNnT
..... 4098, 4115, 6860, 6865, 14794
- \dim_compare:nNnTF
. 78, 3998, 4063, 6766, 6769, 6772,
6781, 6784, 6787, 6796, 6803, 6872,
6985, 6997, 14730, 14747, 14773, 14787
- \dim_compare:nT 4070, 4087
- \dim_compare:nTF 79, 4003
- \dim_compare_p:n 79, 4003
- \dim_compare_p:nNn 78, 3998
- \dim_const:cn 3924
- \dim_const:Nn
.... 76, 3924, 3924, 3929, 4149, 4150
- \dim_do_until:nn ... 81, 4068, 4090, 4094
- \dim_do_until:nNnn .. 80, 4096, 4118, 4122
- \dim_do_while:nn ... 81, 4068, 4084, 4088
- \dim_do_while:nNnn .. 80, 4096, 4112, 4116
- \dim_eval:n 81, 4042, 4047, 4052,
4057, 4124, 4124, 6565, 6613, 6682,
6693, 6710, 6712, 6718, 6729, 6743,
6968, 6969, 14780, 14801, 15049,
15050, 15057, 15058, 15133, 15140
- \dim_gadd:cn 3953
- \dim_gadd:Nn 77, 3953, 3955, 3957
- .dim_gset:c 153
- \dim_gset:cn 3942
- .dim_gset:N 153
- \dim_gset:Nn ... 77, 3927, 3942, 3944, 3946
- \dim_gset_eq:cc 3947
- \dim_gset_eq:cN 3947
- \dim_gset_eq:Nc 3947
- \dim_gset_eq:NN 77, 3947, 3950, 3951, 3952
- \dim_gsub:cn 3953
- \dim_gsub:Nn 77, 3953, 3960, 3962
- \dim_gzero:c 3930
- \dim_gzero:N ... 76, 3930, 3931, 3933, 3937
- \dim_gzero_new:c 3934
- \dim_gzero_new:N ... 76, 3934, 3936, 3939
- \dim_if_exist:c 3941
- \dim_if_exist:cTF 3940
- \dim_if_exist:N 3940
- \dim_if_exist:NTF .. 76, 3935, 3937, 3940
- \dim_if_exist_p:c 3940
- \dim_if_exist_p:N 76, 3940
- \dim_max:nn .. 77, 3963, 3970, 15028, 15032
- \dim_min:nn
. 77, 3963, 3978, 15026, 15030, 15043
- \dim_new:c 3916
- \dim_new:N 76, 3916, 3917,
3923, 3926, 3935, 3937, 4151, 4152,
4153, 4154, 6453, 6475, 6476, 6479,
6480, 6481, 6482, 7055, 7057, 7058,
14478, 14479, 14480, 14481, 14482,
14483, 14484, 14485, 14919, 14920,
14921, 14922, 14923, 15064, 15065
- \dim_ratio:nn 78, 3994, 3994
- .dim_set:c 153
- \dim_set:cn 3942
- .dim_set:N 153
- \dim_set:Nn
. 77, 3942, 3942, 3944, 3945, 6549,
6595, 6768, 6773, 6783, 6788, 6798,
6805, 6818, 6843, 6863, 6922, 6923,
6925, 6927, 6945, 6946, 7056, 7159,
7160, 7211, 7212, 7213, 7215, 14501,
14502, 14503, 14535, 14546, 14606,
14607, 14608, 14624, 14625, 14627,

14636, 14637, 14638, 14656, 14657, 14658, 14675, 14676, 14677, 14679, 14681, 14683, 14965, 14997, 15005, 15016, 15017, 15018, 15019, 15025, 15027, 15029, 15031, 15036, 15042, 15097, 15099, 15101, 15109, 15111	\displaylimits 450
\dim_set_eq:cc 3947	\displaystyle 428
\dim_set_eq:cN 3947	\displaywidowpenalties 678
\dim_set_eq:Nc 3947	\displaywidowpenalty 439
\dim_set_eq:NN 77, 3947, 3947, 3948, 3949, 6551, 6552, 6597, 6598	\displaywidth 441
\dim_show:c 4145	\divide 318
\dim_show:N 82, 4145, 4145, 4146	\doublehyphendemerits 508
\dim_show:n 82, 4147, 4147	\dp 619
\dim_sub:cn 3953	\dump 602
\dim_sub:Nn 77, 3953, 3958, 3960, 3961	
\dim_to_fp:n 185, 6811, 6813, 6823, 6824, 6825, 6826, 6847, 6848, 6849, 6850, 14054, 14054, 14539, 14540, 14550, 14551, 14611, 14614, 14615, 14642, 14643, 14661, 15001, 15002, 15009, 15010, 15069, 15072, 15110, 15112, 15178, 15180	E
\dim_to_pt:n 198, 15328, 15328, 15331, 15333, 15334	\E 15496
\dim_to_unit:nn 198, 15329, 15329	sec 182
\dim_until_do:nn 81, 4068, 4076, 4081	secd 182
\dim_until_do:nNnn 81, 4096, 4104, 4109	\edef 34, 110, 124, 171, 173, 188, 208, 306
\dim_use:c 4143	deg 184
\dim_use:N 82, 3966, 3972, 3973, 3974, 3980, 3981, 3982, 4006, 4027, 4125, 4131, 4143, 4143, 4144, 4148, 6706, 6708, 6712, 6723, 6736, 6953, 7266, 7267, 7268, 14962, 14964, 14967, 14969, 14975, 14981, 14990, 14991, 14992, 15118, 15125	\else 15, 64, 179, 194, 214, 359
\dim_while_do:nn 81, 4068, 4068, 4073	\else: 24, 750, 753, 789, 955, 1087, 1090, 1099, 1105, 1115, 1118, 1127, 1133, 1274, 1295, 1304, 1315, 1418, 1501, 1506, 1512, 1637, 1873, 1923, 1924, 1925, 1981, 1984, 2095, 2129, 2158, 2177, 2297, 2299, 2301, 2303, 2368, 2384, 2407, 2415, 2428, 2437, 2614, 2619, 2624, 2629, 2636, 2642, 2647, 2652, 2657, 2662, 2667, 2672, 2677, 2682, 2702, 2710, 2716, 2719, 2758, 2761, 2780, 2783, 2800, 2803, 2818, 2821, 2836, 2839, 2912, 2921, 2929, 2938, 3004, 3018, 3027, 3037, 3047, 3174, 3195, 3211, 3214, 3358, 3385, 3422, 3430, 3681, 3969, 3990, 4001, 4011, 4038, 4208, 4573, 4585, 4598, 4608, 4624, 4899, 4943, 4966, 4982, 4990, 5000, 5022, 5235, 5272, 5281, 5649, 5664, 5686, 5702, 6156, 6298, 6300, 6310, 9076, 9079, 9344, 9430, 9439, 9440, 9441, 9454, 9464, 9474, 9538, 9595, 9605, 9610, 9620, 9665, 9858, 9886, 9887, 9936, 9954, 9989, 9993, 10029, 10049, 10053, 10057, 10144, 10170, 10179, 10212, 10216, 10228, 10238, 10248, 10279, 10292, 10327, 10337, 10356, 10369, 10382, 10386, 10397, 10420, 10437, 10449, 10463, 10479, 10487, 10489, 10499, 10510, 10526, 10541, 10547, 10552, 10559, 10583, 10609, 10721, 10724, 10744, 10796, 10800, 10808, 10827, 10845, 10860, 10864, 10897, 10922,
\dim_while_do:nNnn 81, 4096, 4096, 4101	
\dim_zero:c 3930	
\dim_zero:N 76, 3930, 3930, 3931, 3932, 3935, 6759, 6760, 14504, 14609, 14639, 14659, 14678	
\dim_zero_new:c 3934	
\dim_zero_new:N 76, 3934, 3934, 3938	
\dimen 612	
\dimendef 311	
\dimexpr 665	
\directlua 16, 714	
\discretionary 475	
\displayindent 440	

10931, 10946, 10962, 10974, 10991, 11007, 11049, 11060, 11067, 11105, 11141, 11152, 11173, 11176, 11179, 11182, 11193, 11202, 11205, 11269, 11282, 11285, 11292, 11310, 11324, 11341, 11421, 11424, 11432, 11444, 11455, 11461, 11474, 11487, 11527, 11561, 11581, 11618, 11636, 11639, 11644, 11658, 11693, 11711, 11714, 11717, 11720, 11779, 11852, 11920, 11921, 11930, 12182, 12211, 12215, 12374, 12383, 12431, 12442, 12458, 12466, 12524, 12603, 12614, 12619, 12653, 12666, 12682, 12686, 12689, 12860, 12867, 12889, 12912, 12927, 12931, 12953, 12984, 12987, 13012, 13015, 13051, 13059, 13064, 13070, 13073, 13083, 13086, 13104, 13119, 13134, 13149, 13164, 13179, 13200, 13245, 13471, 13509, 13510, 13519, 13563, 13615, 13629, 13630, 13631, 13733, 13755, 13770, 13788, 13836, 13852, 13926, 14019, 14026, 14030, 14065, 14070, 14190, 14195, 15513	\etex_detokenize:D 638, 930, 994, 1859, 4731, 4732 \etex_dimexpr:D 665, 3914 \etex_displaywidowpenalties:D 678 \etex_endL:D 686 \etex_endR:D 688 \etex_eTeXrevision:D 630 \etex_eTeXversion:D 629 \etex_everyeof:D 690, 4468 \etex_firstmarks:D 633 \etex_fontchardp:D 658 \etex_fontcharht:D 657 \etex_fontcharic:D 660 \etex_fontcharwd:D 659 \etex_glueexpr:D 666, 4182, 4193, 4198, 4217, 4224, 4229, 4232, 4238, 14059 \etex_glueshrink:D 669, 15345 \etex_glueshrinkorder:D 671 \etex_gluestretch:D 668, 15344 \etex_gluestretchorder:D 670 \etex_gluetomu:D 672 \etex_ifcsname:D 627, 765 \etex_ifdefined:D 626, 764, 808 \etex_iffontchar:D 656 \etex_interactionmode:D 654, 6339, 6342, 6343 \etex_interlinepenalties:D 675 \etex_lastlinefit:D 674 \etex_lastnodetype:D 655 \etex_marks:D 631 \etex_middle:D 679 \etex_muexpr:D 667, 4273, 4284, 4289, 4294, 4300 \etex_mutogluue:D 673 \etex_numexpr:D 664, 3157 \etex_pagediscards:D 682 \etex_parshapedimen:D 663 \etex_parshapeindent:D 661 \etex_parshapelength:D 662 \etex_predisplaydirection:D 689 \etex_protected:D 691, 815, 817, 819, 820, 821, 822, 824, 826, 834, 836, 838, 840 \etex_readline:D 641, 9093 \etex_savinghyphcodes:D 680 \etex_savingvdiscards:D 681 \etex_scantokens:D 639, 4479 \etex_showgroups:D 652 \etex_showifs:D 653
---	--

<code>\etex_showtokens:D</code>	636	2167, 2171, 2255, 2318, 2319, 2333,
... 640, 3880, 4148, 4238, 4300, 8025		2361, 2367, 2369, 2376, 2383, 2385,
<code>\etex_splitbotmarks:D</code>	636	2393, 2400, 2695, 2714, 2739, 2748,
<code>\etex_splitdiscards:D</code>	683	2764, 2786, 2806, 2824, 2842, 2855,
<code>\etex_splitfirstmarks:D</code>	635	2866, 2876, 2896, 2919, 2920, 2922,
<code>\etex_TeXeTstate:D</code>	684	2928, 2931, 3003, 3005, 3015, 3016,
<code>\etex_topmarks:D</code>	632	3017, 3019, 3025, 3026, 3028, 3035,
<code>\etex_tracingassigns:D</code>	642	3036, 3038, 3044, 3045, 3048, 3125,
<code>\etex_tracinggroups:D</code>	649	3134, 3143, 3169, 3174, 3177, 3178,
<code>\etex_tracingifs:D</code>	645	3185, 3186, 3202, 3203, 3224, 3225,
<code>\etex_tracingnesting:D</code>	644	3335, 3340, 3345, 3363, 3365, 3653,
<code>\etex_tracingscantokens:D</code>	643	3681, 3692, 3702, 3835, 3880, 3965,
<code>\etex_unexpanded:D</code>	637,	3969, 3972, 3973, 3980, 3981, 4005,
770, 1836, 1839, 1842, 1847, 3351,		4010, 4023, 4026, 4130, 4148, 4216,
4863, 4889, 4910, 15362, 15412, 15420		4238, 4300, 4366, 4373, 4477, 4478,
<code>\etex_unless:D</code>	628, 755	4529, 4537, 4542, 4583, 4595, 4596,
<code>\etex_widowpenalties:D</code>	677	4732, 4836, 4863, 4893, 4896, 4897,
<code>\eTeXrevision</code>	630	4898, 4900, 4913, 4919, 4927, 4938,
<code>\eTeXversion</code>	629	4957, 4979, 4989, 4992, 5014, 5015,
<code>\everycr</code>	341	5016, 5034, 5035, 5036, 5139, 5150,
<code>\everydisplay</code>	442	5158, 5170, 5178, 5252, 5290, 5302,
<code>\everyeof</code>	690	5312, 5340, 5350, 5393, 5466, 5469,
<code>\everyhbox</code>	581	5472, 5650, 5665, 5687, 5703, 5760,
<code>\everyjob</code>	32, 610	5768, 5773, 5913, 5914, 5917, 5918,
<code>\everymath</code>	466	6002, 6092, 6121, 6193, 7457, 7954,
<code>\everypar</code>	529	7955, 7956, 7957, 7968, 8025, 8026,
<code>\everyvbox</code>	582	8072, 8074, 8234, 8326, 8332, 8824,
<code>\exhyphenpenalty</code>	505	8836, 9336, 9343, 9346, 9429, 9431,
<code>\exp_after:wN</code> 32, 768, 768, 781, 783, 788,		9432, 9442, 9443, 9444, 9453, 9455,
790, 878, 880, 924, 937, 954, 956,		9463, 9465, 9473, 9475, 9482, 9483,
1003, 1008, 1015, 1055, 1059, 1068,		9484, 9485, 9486, 9487, 9492, 9493,
1069, 1098, 1100, 1103, 1126, 1128,		9494, 9495, 9496, 9497, 9498, 9552,
1131, 1273, 1275, 1283, 1303, 1305,		9554, 9581, 9585, 9594, 9603, 9604,
1339, 1388, 1454, 1580, 1597, 1604,		9611, 9618, 9621, 9642, 9710, 9718,
1606, 1609, 1610, 1617, 1621, 1622,		9735, 9744, 9790, 9834, 9870, 9871,
1627, 1628, 1633, 1638, 1640, 1643,		9872, 9933, 9934, 9937, 9948, 9952,
1651, 1653, 1655, 1657, 1659, 1662,		9959, 9960, 9971, 9972, 9981, 9988,
1663, 1664, 1668, 1671, 1676, 1681,		9990, 9991, 9999, 10018, 10028,
1682, 1683, 1687, 1688, 1689, 1693,		10047, 10048, 10050, 10051, 10055,
1694, 1695, 1699, 1700, 1701, 1705,		10056, 10058, 10059, 10068, 10069,
1706, 1707, 1711, 1712, 1713, 1714,		10076, 10081, 10085, 10086, 10088,
1718, 1719, 1720, 1721, 1725, 1726,		10089, 10091, 10104, 10105, 10106,
1727, 1732, 1733, 1734, 1735, 1739,		10121, 10122, 10127, 10128, 10130,
1740, 1741, 1742, 1775, 1776, 1779,		10139, 10140, 10141, 10142, 10145,
1782, 1783, 1787, 1788, 1796, 1798,		10146, 10147, 10148, 10159, 10176,
1799, 1801, 1803, 1806, 1807, 1812,		10177, 10187, 10188, 10198, 10210,
1813, 1817, 1820, 1821, 1822, 1826,		10214, 10227, 10229, 10237, 10247,
1833, 1835, 1836, 1837, 1839, 1842,		10249, 10255, 10260, 10262, 10264,
1847, 1855, 1856, 1857, 1858, 1871,		10270, 10271, 10275, 10277, 10289,
1874, 1895, 1902, 1922, 2037, 2164,		10290, 10312, 10314, 10320, 10323,

10325, 10329, 10334, 10339, 10340,
10350, 10351, 10353, 10354, 10357,
10361, 10366, 10380, 10383, 10395,
10404, 10411, 10412, 10413, 10414,
10416, 10418, 10429, 10430, 10431,
10432, 10434, 10436, 10438, 10439,
10440, 10446, 10447, 10457, 10461,
10462, 10464, 10465, 10466, 10471,
10477, 10488, 10490, 10497, 10498,
10500, 10501, 10508, 10514, 10524,
10576, 10577, 10578, 10579, 10593,
10594, 10596, 10604, 10605, 10619,
10620, 10633, 10664, 10665, 10666,
10672, 10673, 10674, 10675, 10681,
10682, 10683, 10684, 10730, 10731,
10732, 10747, 10754, 10766, 10767,
10769, 10770, 10776, 10777, 10794,
10795, 10797, 10798, 10806, 10807,
10809, 10812, 10824, 10825, 10826,
10828, 10829, 10830, 10842, 10843,
10844, 10846, 10847, 10848, 10857,
10858, 10859, 10862, 10865, 10869,
10891, 10892, 10893, 10896, 10898,
10899, 10900, 10917, 10918, 10921,
10923, 10924, 10925, 10944, 10945,
10947, 10948, 10949, 10959, 10960,
10961, 10963, 10964, 10965, 10971,
10972, 10973, 10975, 10976, 10977,
10986, 10987, 10990, 10992, 10993,
10994, 11002, 11003, 11006, 11008,
11009, 11010, 11031, 11032, 11047,
11048, 11050, 11051, 11052, 11072,
11073, 11074, 11075, 11076, 11077,
11078, 11079, 11084, 11085, 11086,
11087, 11088, 11089, 11095, 11096,
11098, 11099, 11100, 11134, 11148,
11158, 11166, 11167, 11268, 11270,
11309, 11311, 11333, 11334, 11337,
11344, 11345, 11348, 11349, 11355,
11360, 11367, 11368, 11375, 11376,
11428, 11429, 11430, 11432, 11443,
11459, 11460, 11462, 11463, 11473,
11475, 11481, 11482, 11486, 11489,
11511, 11513, 11526, 11528, 11534,
11536, 11539, 11545, 11547, 11549,
11550, 11551, 11553, 11558, 11560,
11562, 11566, 11569, 11575, 11576,
11580, 11582, 11583, 11584, 11592,
11594, 11595, 11602, 11608, 11615,
11616, 11621, 11622, 11623, 11624,
11642, 11643, 11644, 11650, 11651,
11652, 11657, 11659, 11667, 11669,
11671, 11672, 11674, 11685, 11687,
11689, 11690, 11695, 11746, 11747,
11754, 11755, 11757, 11759, 11761,
11763, 11765, 11767, 11769, 11778,
11780, 11786, 11788, 11790, 11791,
11792, 11798, 11800, 11802, 11803,
11804, 11825, 11826, 11829, 11837,
11839, 11843, 11844, 11845, 11846,
11851, 11853, 11859, 11862, 11865,
11868, 11876, 11879, 11882, 11885,
11892, 11894, 11900, 11908, 11910,
11912, 11929, 11931, 11938, 11940,
11943, 11949, 11951, 11953, 11954,
11955, 11957, 11961, 11962, 11963,
11964, 11977, 11982, 11984, 11991,
11994, 11997, 12000, 12003, 12006,
12011, 12015, 12016, 12028, 12036,
12038, 12048, 12050, 12057, 12066,
12068, 12071, 12074, 12077, 12080,
12094, 12096, 12104, 12106, 12114,
12116, 12125, 12128, 12131, 12138,
12153, 12154, 12171, 12173, 12174,
12231, 12244, 12246, 12252, 12265,
12267, 12269, 12293, 12307, 12309,
12316, 12318, 12358, 12359, 12360,
12362, 12363, 12364, 12366, 12367,
12373, 12375, 12376, 12382, 12384,
12385, 12386, 12387, 12399, 12405,
12407, 12438, 12445, 12452, 12472,
12473, 12475, 12477, 12479, 12491,
12496, 12497, 12498, 12499, 12500,
12504, 12509, 12511, 12517, 12523,
12525, 12526, 12532, 12533, 12534,
12535, 12536, 12537, 12538, 12539,
12544, 12546, 12548, 12550, 12552,
12556, 12558, 12560, 12562, 12564,
12566, 12584, 12588, 12596, 12597,
12602, 12604, 12613, 12616, 12617,
12618, 12620, 12621, 12622, 12630,
12636, 12648, 12651, 12652, 12654,
12655, 12673, 12674, 12678, 12684,
12688, 12690, 12707, 12725, 12731,
12740, 12741, 12742, 12743, 12751,
12767, 12783, 12799, 12815, 12831,
12857, 12861, 12862, 12866, 12868,
12899, 12905, 12906, 12908, 12910,
12911, 12913, 12914, 12924, 12925,
12928, 12929, 12930, 12932, 12933,

- 12934, 12951, 12952, 12954, 12955,
 12961, 12963, 12966, 12969, 12972,
 12975, 12983, 12986, 12988, 12991,
 12998, 13002, 13010, 13011, 13014,
 13016, 13018, 13022, 13026, 13031,
 13032, 13040, 13041, 13042, 13043,
 13060, 13061, 13067, 13068, 13193,
 13194, 13195, 13197, 13215, 13216,
 13217, 13218, 13219, 13220, 13227,
 13236, 13243, 13244, 13254, 13418,
 13419, 13420, 13441, 13442, 13452,
 13461, 13469, 13470, 13472, 13473,
 13478, 13479, 13488, 13495, 13504,
 13505, 13518, 13520, 13548, 13549,
 13558, 13561, 13586, 13592, 13593,
 13612, 13618, 13620, 13645, 13646,
 13648, 13662, 13663, 13671, 13681,
 13715, 13718, 13728, 13729, 13732,
 13734, 13740, 13754, 13756, 13797,
 13800, 13820, 13882, 13886, 13896,
 13899, 13905, 13906, 13925, 13927,
 13928, 13937, 13941, 13946, 13949,
 13955, 13981, 13982, 13988, 13989,
 13990, 13997, 14005, 14009, 14014,
 14017, 14025, 14028, 14029, 14031,
 14032, 14042, 14046, 14051, 14056,
 14057, 14058, 14064, 14066, 14069,
 14071, 14113, 14115, 14116, 14186,
 14193, 14211, 14244, 14270, 14271,
 14272, 14273, 14274, 14275, 14283,
 14288, 15193, 15213, 15238, 15241,
 15295, 15362, 15382, 15412, 15420,
 15444, 15445, 15507, 15512, 15514
 \exp_args:cc
 ... 780, 782, 959, 969, 974, 979, 1656
 \exp_args:Nc
 29, 780, 780, 784, 792, 1199, 1209,
 1227, 1253, 1258, 1265, 1320, 1331,
 1387, 1420, 1421, 1422, 1423, 1442,
 1656, 4703, 8675, 9660, 14249, 15156
 \exp_args:Ncc 1255, 1259, 1267,
 1428, 1429, 1430, 1431, 1656, 1658
 \exp_args:Nccc 1656, 1660
 \exp_args:Ncco 1716, 1737
 \exp_args:Nccx 1760, 1769
 \exp_args:Ncf 1679, 1703
 \exp_args:NcNc 1716, 1723
 \exp_args:NcNo 1716, 1730
 \exp_args:Ncnx 1760, 1770
 \exp_args:Nco 1679, 1697
 \exp_args:Ncx 1745, 1755
 \exp_args:Nf 30, 1667, 1667, 3392, 3397,
 3402, 3407, 3555, 3624, 3636, 3645,
 3738, 3751, 3765, 3775, 3786, 3797,
 4042, 4047, 4052, 4057, 5902, 7966,
 14824, 14837, 14842, 14858, 15216,
 15234, 15381, 15436, 15449, 15467
 \exp_args:Nff 1745, 1747
 \exp_args:Nfo 1745, 1746, 14812
 \exp_args:NNc 1254, 1257,
 1266, 1333, 1424, 1425, 1426, 1427,
 1461, 1656, 1656, 3516, 3523, 7968
 \exp_args:Nnc 1745, 1745
 \exp_args:NNf 1679,
 1679, 3493, 3500, 3509, 13187, 13188
 \exp_args:Nnf 1745, 1748
 \exp_args:Nnnc 1760, 1762
 \exp_args:NNNo 31, 1651, 1654
 \exp_args:NNno 1760, 1760
 \exp_args:Nnno 1760, 1763
 \exp_args:NNNV 1716, 1716
 \exp_args:NNnx 31, 1760, 1765
 \exp_args:Nnnx 1760, 1767
 \exp_args:NNo 30,
 1651, 1652, 3543, 5996, 9250, 15438
 \exp_args:Nno
 . 30, 1745, 1749, 3117, 4013, 5857,
 9695, 9703, 9712, 9729, 9737, 9765
 \exp_args:NNoo 31, 1760, 1761
 \exp_args:NNox 1760, 1766
 \exp_args:Nnox 1760, 1768
 \exp_args:NNV 1679, 1691
 \exp_args:NNv 1679, 1685
 \exp_args:NnV 1745, 1750
 \exp_args:NNx 31, 1745, 1754
 \exp_args:Nnx 1745, 1756
 \exp_args:No 29, 1651,
 1651, 3543, 3628, 4222, 4468, 4631,
 4632, 4633, 4646, 4647, 4648, 4649,
 4692, 4709, 4718, 4772, 4774, 4882,
 4884, 4907, 4916, 5128, 5589, 5767,
 5781, 5786, 8403, 8421, 8447, 8469,
 8675, 9332, 14852, 14856, 15171, 15469
 \exp_args:Noc 1745, 1753
 \exp_args:Nof 1745, 1752
 \exp_args:Noo 1745, 1751
 \exp_args:Nooo 1760, 1764
 \exp_args:Noox 1760, 1771
 \exp_args:Nox 1745, 1757

- `\exp_args:NV` 30, [1667](#), 1674, 8401, 8419, 8445, 8467
- `\exp_args:Nv` 30, [1667](#), 1669
- `\exp_args:NVV` [1679](#), 1709
- `\exp_args:Nx` 30, [1281](#), [1744](#),
1744, 7423, 8405, 8423, 8449, 8471
- `\exp_args:Nxo` [1745](#), 1758
- `\exp_args:Nxx` [1745](#), 1759
- `\exp_last_two_unbraced:Noo`
.... 32, [1832](#), 1832, 6753, 6976, 6980
- `\exp_last_unbraced:Nco` . [1795](#), 1802, 5840
- `\exp_last_unbraced:NcV` [1795](#), 1804
- `\exp_last_unbraced:Nf`
..... 32, [1795](#), 1800, 3634, 14881
- `\exp_last_unbraced:Nfo` [1795](#), 1829
- `\exp_last_unbraced:NNNo` [1795](#), 1825
- `\exp_last_unbraced:NnNo` [1795](#), 1830
- `\exp_last_unbraced:NNNV` [1795](#), 1818
- `\exp_last_unbraced:NNo` .. [1795](#), 1816,
4850, 5136, 5138, 5808, 6186, 6950
- `\exp_last_unbraced:Nno` [1795](#), 1827, 15186
- `\exp_last_unbraced:NNV` [1795](#), 1810
- `\exp_last_unbraced:No` [1795](#),
1799, 7105, 7110, 7188, 7194, 14867
- `\exp_last_unbraced:Noo`
..... [1795](#), 1828, 6141, 15201
- `\exp_last_unbraced:NV` [1795](#), 1795
- `\exp_last_unbraced:Nv` [1795](#), 1797
- `\exp_last_unbraced:Nx` ... 32, [1795](#), 1831
- `\exp_not:c` 33, [1836](#), 1837,
1947, 1997, 3064, 7503, 7505, 7507,
7509, 7515, 7520, 7522, 7524, 7747,
7749, 7751, 7753, 7759, 7764, 7766,
7768, 7959, 8095, 8120, 8252, 8254,
8268, 8270, 8377, 11412, 14202, 14208
- `\exp_not:f` 33, [1836](#),
1838, 5150, 5158, 14129, 14131, 14133
- `\exp_not:N` 33, [768](#), 769,
928, 990, 993, 1338, 1339, 1387,
1388, [1597](#), 1633, 1837, 1871, 1906,
1915, 1944, 1945, 1946, 1997, 2037,
2375, 2382, 2399, 2426, 2435, 2589,
2613, 2618, 2623, 2628, 2635, 2641,
2646, 2651, 2656, 2661, 2666, 2676,
2681, 2709, 2714, 2967, 2972, 2990,
3014, 3017, 3024, 3025, 3034, 3035,
3528, 4135, 4136, 4139, 4468, 4475,
4485, 4487, 4521, 4522, 4935, 4938,
4954, 4957, 4988, 4995, 5127, 5129,
5439, 5892, 5895, 5903, 5904, 6084,
6113, 6340, 7514, 7758, 7959, 8252,
8254, 8268, 8270, 8298, 8299, 8357,
8358, 8377, 8798, 8799, 9248, 9316,
10045, 10106, 10168, 10209, 10213,
10235, 10328, 10360, 10445, 10459,
10476, 10486, 10496, 10533, 10536,
10790, 10792, 10943, 10958, 10970,
14038, 14041, 14112, 14113, 14115,
14116, 14117, 14118, 14120, 14200,
14201, 14206, 14207, 14346, 14383,
14388, 14411, 15323, 15503, 15504
- `\exp_not:n` 33,
[768](#), 770, 929, 931, 995, 1283, 1499,
[1597](#), 1792, 1951, 1966, 2375, 2382,
2399, 2426, 2435, 2968, 2973, 2987,
2991, 3063, 3065, 3529, 4325, 4384,
4390, 4402, 4410, 4426, 4434, 4523,
5009, 5149, 5157, 5171, 5179, 5182,
5187, 5295, 5353, 5440, 5483, 5494,
5608, 5774, 5896, 5902, 5927, 5932,
5994, 6085, 6088, 6114, 7497, 7514,
7612, 7741, 7758, 7951, 8301, 8360,
8377, 8868, 9093, 9159, 9162, 9243,
11413, 13254, 14318, 14322, 14836,
14869, 14890, 14891, 15208, 15233,
15299, 15300, 15303, 15358, 15466
- `\exp_not:o` 33, [1836](#), 1836, 4355,
4357, 4386, 4392, 4402, 4404, 4406,
4408, 4410, 4412, 4414, 4416, 4426,
4428, 4430, 4432, 4434, 4436, 4438,
4440, 4487, 4534, 4546, 4651, 4768,
4770, 5008, 5219, 5577, 5579, 5674,
5767, 8097, 8111, 8122, 8125, 8132,
8141, 8514, 8528, 8684, 8686, 15357
- `\exp_not:V`
33, [1836](#), 1840, 4404, 4412, 4428, 4436
- `\exp_not:v` 33, [1836](#), 1845, 8801
- `\exp_stop_f:` 33, [1607](#), 1613, 2320,
3171, 3181, 3189, 3357, 3362, 4021,
5161, 7968, 9383, 9592, 9860, 9876,
10027, 10054, 10226, 10246, 10273,
10287, 10322, 10349, 10358, 10377,
10393, 10409, 10427, 10487, 10506,
10522, 11101, 11186, 11188, 11191,
11199, 11200, 11420, 11422, 11442,
11484, 11557, 11579, 11633, 11634,
11966, 12208, 12209, 12306, 12427,
12445, 12454, 12610, 12645, 12744,
12856, 12898, 12903, 12985, 13048,
13081, 13095, 13110, 13125, 13140,

- 13155, 13170, 13242, 13468, 13680,
 13753, 13762, 13777, 13829, 13842,
 13897, 13947, 14001, 14015, 15439
 \expandafter 13, 14, 32, 36, 62,
 63, 65, 68, 69, 72, 73, 87, 106, 138,
 173, 176, 182, 186, 190, 191, 195,
 196, 198, 208, 210, 213, 215, 220, 329
 \ExplFileDate 50, 289, 747, 1594,
 2057, 2348, 2474, 3153, 3910, 4313,
 5061, 5545, 5956, 6230, 6449, 7275,
 7308, 8047, 8819, 9378, 14296, 14471
 \ExplFileDescription 289, 747, 1594,
 2057, 2348, 2474, 3153, 3910, 4313,
 5061, 5545, 5956, 6230, 6449, 7275,
 7308, 8047, 8819, 9378, 14296, 14471
 \ExplFileName 289, 747, 1594,
 2057, 2348, 2474, 3153, 3910, 4313,
 5061, 5545, 5956, 6230, 6449, 7275,
 7308, 8047, 8819, 9378, 14296, 14471
 \ExplFileVersion 50, 289, 747, 1594,
 2057, 2348, 2474, 3153, 3910, 4313,
 5061, 5545, 5956, 6230, 6449, 7275,
 7308, 8047, 8819, 9378, 14296, 14471
 \ExplSyntaxOff 4, 6, 109,
 110, 185, 193, 215, 246, 251, 265, 280
 \ExplSyntaxOn 4, 6,
109, 124, 155, 162, 167, 213, 246, 247
- F**
- \F 2689, 2888, 10010
 \fam 321
 \fi 45, 66, 71, 108, 181, 201, 216, 360
 \fi: 24, 750, 754, 791, 925, 938, 957,
 1004, 1009, 1016, 1054, 1059, 1092,
 1093, 1101, 1107, 1120, 1121, 1129,
 1135, 1196, 1276, 1296, 1306, 1317,
 1418, 1501, 1506, 1514, 1581, 1636,
 1639, 1646, 1647, 1880, 1896, 1903,
 1912, 1927, 1928, 1929, 1943, 1963,
 1965, 1986, 1987, 2097, 2131, 2158,
 2177, 2297, 2299, 2301, 2303, 2305,
 2307, 2362, 2370, 2377, 2386, 2394,
 2401, 2409, 2417, 2430, 2439, 2614,
 2619, 2624, 2629, 2636, 2642, 2647,
 2652, 2657, 2662, 2667, 2672, 2677,
 2682, 2704, 2710, 2721, 2722, 2768,
 2769, 2790, 2791, 2810, 2811, 2828,
 2829, 2846, 2847, 2914, 2923, 2932,
 2940, 3006, 3020, 3029, 3039, 3049,
 3174, 3197, 3214, 3215, 3217, 3320,
 3328, 3352, 3358, 3364, 3387, 3424,
 3432, 3682, 3969, 3992, 4001, 4020,
 4024, 4038, 4210, 4575, 4587, 4600,
 4610, 4627, 4837, 4890, 4893, 4901,
 4903, 4931, 4945, 4968, 4984, 4993,
 5002, 5024, 5028, 5036, 5215, 5218,
 5237, 5253, 5274, 5284, 5339, 5343,
 5651, 5666, 5689, 5705, 6158, 6194,
 6298, 6300, 6310, 7338, 9081, 9082,
 9347, 9433, 9441, 9445, 9456, 9466,
 9476, 9540, 9578, 9579, 9580, 9581,
 9582, 9583, 9584, 9585, 9586, 9587,
 9588, 9589, 9596, 9609, 9612, 9619,
 9622, 9667, 9770, 9840, 9841, 9850,
 9851, 9862, 9863, 9864, 9874, 9875,
 9879, 9880, 9888, 9891, 9892, 9900,
 9901, 9908, 9916, 9917, 9938, 9957,
 9958, 9967, 9973, 9993, 9994, 10007,
 10031, 10052, 10060, 10061, 10137,
 10150, 10172, 10174, 10181, 10189,
 10218, 10219, 10222, 10224, 10225,
 10230, 10240, 10243, 10245, 10250,
 10281, 10294, 10299, 10305, 10308,
 10309, 10343, 10344, 10371, 10372,
 10385, 10388, 10399, 10422, 10441,
 10451, 10467, 10481, 10487, 10491,
 10496, 10502, 10517, 10528, 10546,
 10556, 10558, 10564, 10585, 10611,
 10727, 10728, 10729, 10748, 10799,
 10803, 10814, 10815, 10831, 10849,
 10867, 10870, 10901, 10926, 10936,
 10950, 10966, 10978, 10995, 11011,
 11053, 11062, 11068, 11081, 11083,
 11106, 11143, 11154, 11166, 11167,
 11168, 11175, 11177, 11178, 11184,
 11185, 11194, 11195, 11203, 11204,
 11206, 11207, 11271, 11284, 11294,
 11295, 11300, 11301, 11302, 11303,
 11304, 11305, 11312, 11321, 11326,
 11350, 11352, 11354, 11361, 11388,
 11395, 11398, 11399, 11404, 11426,
 11427, 11433, 11446, 11464, 11466,
 11476, 11490, 11520, 11529, 11563,
 11585, 11603, 11620, 11637, 11638,
 11640, 11641, 11645, 11660, 11693,
 11722, 11723, 11724, 11725, 11726,
 11739, 11781, 11854, 11919, 11921,
 11922, 11932, 11966, 12184, 12188,
 12190, 12194, 12201, 12202, 12212,
 12213, 12216, 12308, 12377, 12388,

12400, 12426, 12433, 12444, 12460,	\fp_compare:n	11132
12467, 12527, 12583, 12593, 12595,	\fp_compare:nF	11212, 11223
12605, 12623, 12624, 12656, 12659,	\fp_compare:nNn	11145
12668, 12670, 12672, 12691, 12705,	\fp_compare:NNNF	14261
12706, 12726, 12747, 12748, 12761,	\fp_compare:nNnF	11240, 11251, 14261
12777, 12793, 12809, 12825, 12841,	\fp_compare:NNNT	14260
12855, 12863, 12869, 12879, 12882,	\fp_compare:nNnT 11246, 11259, 14260, 15084	
12891, 12900, 12902, 12908, 12915,	\fp_compare:NNNTF	14259, 14259
12918, 12927, 12935, 12956, 12989,	\fp_compare:nNnTF	174, 6814, 11145,
12990, 13017, 13019, 13033, 13034,	14259, 14505, 14507, 14512, 14700	
13052, 13063, 13072, 13075, 13076,	\fp_compare:nT	11218, 11231
13085, 13088, 13105, 13120, 13135,	\fp_compare:nTF	11132
13150, 13165, 13180, 13183, 13185,	\fp_compare_p:n	11132
13202, 13247, 13474, 13510, 13511,	\fp_compare_p:nNn	174, 11145
13521, 13562, 13563, 13587, 13619,	\fp_const:cn	14128
13625, 13626, 13629, 13631, 13632,	\fp_const:Nn 171, 14128, 14132, 14136,	
13636, 13648, 13667, 13672, 13679,	14173, 14174, 14175, 14176, 14184	
13682, 13714, 13724, 13725, 13735,	\fp_cos:cn	14230
13757, 13772, 13790, 13798, 13801,	\fp_cos:Nn	14230, 14240
13829, 13837, 13853, 13896, 13918,	\fp_div:cn	14216
13930, 13946, 13967, 14000, 14014,	\fp_div:Nn	14216, 14218, 14226
14020, 14033, 14034, 14072, 14073,	\fp_do_until:nn	175, 11209, 11209, 11213
14190, 14195, 14250, 15194, 15503,	\fp_do_until:nNnn 174, 11237, 11237, 11241	
15504, 15505, 15515, 15521, 15523	\fp_do_while:nn	175, 11209, 11215, 11219
\file_add_path:nN	\fp_do_while:nNnn 174, 11237, 11243, 11247	
162, 8871, 8871, 8909, 8916, 9004, 9016	\fp_eval:n	172, 14089, 14091
\file_if_exist:n	8907	
\file_if_exist:nTF	162, 8907	
\file_input:n	162, 8914, 8914	
\file_list:	163, 8960, 8960	
\file_path_include:n	163, 8948, 8948	
\file_path_remove:n	163, 8948, 8955	
\finalhyphendemerits	509	
\firstmark	407	
\firstmarks	633	
\floatingpenalty	554	
\font	320	
\fontcharhp	658	
\fontcharht	657	
\fontcharic	660	
\fontcharwd	659	
\fontdimen	587	
\fontname	411	
\fp_abs:c	14196	
\fp_abs:N	14196, 14196, 14212	
\fp_abs:n	185, 14092, 14092, 14626, 14628,	
14680, 14682, 14684, 15100, 15102		
\fp_add:cn	14151	
\fp_add:Nn	172, 14151, 14151, 14157	
\fp_compare:n	11132	
\fp_compare:nF	11212, 11223	
\fp_compare:nNn	11145	
\fp_compare:NNNF	14261	
\fp_compare:nNnF	11240, 11251, 14261	
\fp_compare:NNNT	14260	
\fp_compare:nNnT 11246, 11259, 14260, 15084		
\fp_compare:NNNTF	14259, 14259	
\fp_compare:nNnTF	174, 6814, 11145,	
14259, 14505, 14507, 14512, 14700		
\fp_compare:nT	11218, 11231	
\fp_compare:nTF	11132	
\fp_compare_p:n	11132	
\fp_compare_p:nNn	174, 11145	
\fp_const:cn	14128	
\fp_const:Nn 171, 14128, 14132, 14136,		
14173, 14174, 14175, 14176, 14184		
\fp_cos:cn	14230	
\fp_cos:Nn	14230, 14240	
\fp_div:cn	14216	
\fp_div:Nn	14216, 14218, 14226	
\fp_do_until:nn	175, 11209, 11209, 11213	
\fp_do_until:nNnn 174, 11237, 11237, 11241		
\fp_do_while:nn	175, 11209, 11215, 11219	
\fp_do_while:nNnn 174, 11237, 11243, 11247		
\fp_eval:n	172, 14089, 14091	
\fp_exp:cn	14230	
\fp_exp:Nn	14230, 14237	
\fp_flag_off:n	177, 9657, 9657	
\fp_flag_on:n	177, 9659, 9659,	
9700, 9709, 9717, 9734, 9743, 9774		
\fp_gabs:c	14196	
\fp_gabs:N	14196, 14197, 14213	
\fp_gadd:cn	14151	
\fp_gadd:Nn	172, 14151, 14152, 14158	
\fp_gcos:cn	14230	
\fp_gcos:Nn	14230, 14240	
\fp_gdiv:cn	14216	
\fp_gdiv:Nn	14216, 14219, 14227	
\fp_gexp:cn	14230	
\fp_gexp:Nn	14230, 14237	
\fp_gln:cn	14230	
\fp_gln:Nn	14230, 14238	
\fp_gmul:cn	14216	
\fp_gmul:Nn	14216, 14217, 14225	
\fp_gneg:c	14196	
\fp_gneg:N	14196, 14204, 14215	
\fp_gpow:cn	14216	
\fp_gpow:Nn	14216, 14221, 14229	
\fp_ground_figures:Nn 14280, 14285, 14291		

- \fp_ground_places:Nn [14262](#), [14264](#), [14279](#)
- .fp_gset:c [153](#)
- \fp_gset:cn [14128](#)
- .fp_gset:N [153](#)
- \fp_gset:Nn . [171](#), [14128](#), [14130](#), [14135](#),
[14152](#), [14154](#), [14217](#), [14219](#), [14221](#)
- \fp_gset_eq:cc [14137](#)
- \fp_gset_eq:cN [14137](#)
- \fp_gset_eq:Nc [14137](#)
- \fp_gset_eq:NN
..... [172](#), [14137](#), [14138](#), [14140](#), [14142](#)
- \fp_gset_from_dim:cn [15177](#)
- \fp_gset_from_dim:Nn
..... [196](#), [15177](#), [15179](#), [15182](#)
- \fp_gsin:cn [14230](#)
- \fp_gsin:Nn [14230](#), [14239](#)
- \fp_gsub:cn [14151](#)
- \fp_gsub:Nn [172](#), [14151](#), [14154](#), [14160](#)
- \fp_gtan:cn [14230](#)
- \fp_gtan:Nn [14230](#), [14241](#)
- \fp_gzero:c [14141](#)
- \fp_gzero:N [171](#), [14141](#), [14142](#), [14144](#), [14148](#)
- \fp_gzero_new:c [14145](#)
- \fp_gzero_new:N . [171](#), [14145](#), [14147](#), [14150](#)
- \fp_if_exist:c [11131](#)
- \fp_if_exist:cTF [11130](#)
- \fp_if_exist:N [11130](#)
- \fp_if_exist:NTF
..... [173](#), [11130](#), [14146](#), [14148](#), [14163](#)
- \fp_if_exist_p:c [11130](#)
- \fp_if_exist_p:N [173](#), [11130](#)
- \fp_if_flag_on:n [9661](#)
- \fp_if_flag_on:nTF [177](#), [9661](#)
- \fp_if_flag_on_p:n [177](#), [9661](#)
- \fp_if_undefined:N [14185](#)
- \fp_if_undefined:NTF [14185](#)
- \fp_if_undefined_p:N [14185](#)
- \fp_if_zero:N [14192](#)
- \fp_if_zero:NTF [14192](#)
- \fp_if_zero_p:N [14192](#)
- \fp_ln:cn [14230](#)
- \fp_ln:Nn [14230](#), [14238](#)
- \fp_max:nn [185](#), [14094](#), [14094](#)
- \fp_min:nn [185](#), [14094](#), [14096](#)
- \fp_mul:cn [14216](#)
- \fp_mul:Nn [14216](#), [14216](#), [14224](#)
- \fp_neg:c [14196](#)
- \fp_neg:N [14196](#), [14198](#), [14214](#)
- \fp_new:N [171](#),
[6472](#), [6473](#), [14125](#), [14125](#), [14127](#),
[14146](#), [14148](#), [14177](#), [14178](#), [14179](#),
[14180](#), [14475](#), [14476](#), [14477](#), [14599](#),
[14600](#), [14916](#), [14917](#), [15062](#), [15063](#)
- \fp_pow:cn [14216](#)
- \fp_pow:Nn [14216](#), [14220](#), [14228](#)
- \fp_round_figures:Nn [14280](#), [14280](#), [14290](#)
- \fp_round_places:Nn . [14262](#), [14262](#), [14278](#)
- .fp_set:c [153](#)
- \fp_set:cn [14128](#)
- .fp_set:N [153](#)
- \fp_set:Nn [171](#),
[6810](#), [6812](#), [14128](#), [14128](#), [14134](#),
[14151](#), [14153](#), [14216](#), [14218](#), [14220](#),
[14492](#), [14493](#), [14494](#), [14610](#), [14612](#),
[14640](#), [14660](#), [14673](#), [14674](#), [14926](#),
[14927](#), [15068](#), [15070](#), [15094](#), [15095](#)
- \fp_set_eq:cc [14137](#)
- \fp_set_eq:cN [14137](#)
- \fp_set_eq:Nc [14137](#)
- \fp_set_eq:NN [172](#), [14137](#),
[14137](#), [14139](#), [14141](#), [14645](#), [14662](#)
- \fp_set_from_dim:cn [15177](#)
- \fp_set_from_dim:Nn
..... [196](#), [15177](#), [15177](#), [15181](#)
- \fp_show:c [14161](#)
- \fp_show:N [177](#), [14161](#), [14161](#), [14172](#)
- \fp_show:n [14161](#), [14170](#)
- \fp_sin:cn [14230](#)
- \fp_sin:Nn [14230](#), [14239](#)
- \fp_sub:cn [14151](#)
- \fp_sub:Nn [172](#), [14151](#), [14153](#), [14159](#)
- \fp_tan:cn [14230](#)
- \fp_tan:Nn [14230](#), [14241](#)
- \fp_to_decimal:c [13936](#)
- \fp_to_decimal:N [172](#), [9650](#),
[13936](#), [13936](#), [13938](#), [14038](#), [14089](#)
- \fp_to_decimal:n [13936](#), [13939](#),
[14041](#), [14091](#), [14093](#), [14095](#), [14097](#)
- \fp_to_dim:c [14037](#)
- \fp_to_dim:N [172](#), [14037](#), [14037](#), [14039](#)
- \fp_to_dim:n
..... [6820](#), [6845](#), [14037](#), [14040](#), [14537](#),
[14548](#), [14999](#), [15007](#), [15110](#), [15112](#)
- \fp_to_int:c [14042](#)
- \fp_to_int:N [173](#), [14042](#), [14042](#), [14043](#)
- \fp_to_int:n [14042](#), [14044](#)
- \fp_to_scientific:c [13881](#)
- \fp_to_scientific:N
..... [173](#), [9651](#), [13881](#), [13881](#), [13883](#)
- \fp_to_scientific:n [13881](#), [13884](#)

- \fp_to_tl:c [14005](#)
 - \fp_to_tl:N [173](#), [14005](#), [14005](#), [14006](#), [14164](#)
 - \fp_to_tl:n [9397](#), [9699](#), [9708](#),
[9733](#), [9742](#), [9771](#), [14005](#), [14007](#), [14171](#)
 - \fp_trap:nn
[177](#), [9673](#), [9673](#), [9785](#), [9786](#), [9787](#), [9788](#)
 - \fp_until_do:nn . [175](#), [11209](#), [11221](#), [11226](#)
 - \fp_until_do:nNnn [174](#), [11237](#), [11249](#), [11254](#)
 - \fp_use:c [14089](#)
 - \fp_use:N [173](#), [14089](#), [14089](#), [14090](#)
 - \fp_while_do:nn . [175](#), [11209](#), [11229](#), [11234](#)
 - \fp_while_do:nNnn [175](#), [11237](#), [11257](#), [11262](#)
 - \fp_zero:c [14141](#)
 - \fp_zero:N [171](#), [14141](#), [14141](#), [14143](#), [14146](#)
 - \fp_zero_new:c [14145](#)
 - \fp_zero_new:N .. [171](#), [14145](#), [14145](#), [14149](#)
 - \frozen@everydisplay [727](#)
 - \frozen@everymath [728](#)
 - \futurelet [316](#)
- G**
- \g_cctab_allocate_int [14325](#),
[14325](#), [14326](#), [14332](#), [14334](#), [14336](#)
 - \g_cctab_stack_int ... [14325](#), [14327](#),
[14364](#), [14365](#), [14367](#), [14368](#), [14372](#)
 - \g_cctab_stack_seq
.. [14325](#), [14328](#), [14362](#), [14373](#), [14375](#)
 - \g_file_internal_ior [8875](#),
[8876](#), [8879](#), [8895](#), [8896](#), [9097](#), [9097](#)
 - \g_file_record_seq [8834](#),
[8834](#), [8839](#), [8934](#), [8939](#), [8962](#), [8982](#)
 - \g_file_stack_seq [8833](#), [8833](#), [8941](#), [8944](#)
 - \g_ior_streams_prop
.. [8993](#), [8993](#), [8995](#), [9044](#), [9052](#), [9060](#)
 - \g_ior_streams_seq [8987](#),
[8987](#), [8989](#), [9027](#), [9053](#), [9054](#), [9103](#)
 - \g_iow_streams_prop [9106](#), [9106](#),
[9108](#), [9109](#), [9110](#), [9137](#), [9145](#), [9153](#)
 - \g_iow_streams_seq
.. [9101](#), [9101](#), [9103](#), [9120](#), [9146](#), [9147](#)
 - \g_keyval_level_int [8051](#), [8051](#),
[8096](#), [8121](#), [8145](#), [8147](#), [8149](#), [8151](#)
 - \g_prg_map_int [43](#), [2343](#), [2343](#),
[3515](#), [3518](#), [3522](#), [3525](#), [3536](#), [4701](#),
[4702](#), [4704](#), [4706](#), [5418](#), [5419](#), [5425](#),
[5426](#), [5838](#), [5839](#), [5841](#), [5844](#), [6203](#),
[6204](#), [6209](#), [6211](#), [15155](#), [15157](#), [15164](#)
 - \g_scan_marks_tl [2448](#), [2448](#), [2451](#), [2457](#)
 - \g_file_current_name_tl
..... [162](#), [8822](#), [8822](#),
[8827](#), [8831](#), [8839](#), [8941](#), [8942](#), [8945](#)
 - \g_peek_token [58](#), [2943](#), [2944](#), [2954](#)
 - \g_tmpa_bool [38](#), [2119](#), [2121](#)
 - \g_tmpa_box [131](#), [6322](#), [6324](#)
 - \g_tmpa_clist [121](#), [5947](#), [5949](#)
 - \g_tmpa_dim [82](#), [4151](#), [4153](#)
 - \g_tmpa_fp [176](#), [14177](#), [14179](#)
 - \g_tmpa_int [73](#), [3900](#), [3902](#)
 - \g_tmpa_muskip [88](#), [4303](#), [4305](#)
 - \g_tmpa_prop [127](#), [5990](#), [5992](#)
 - \g_tmpa_seq [112](#), [5536](#), [5538](#)
 - \g_tmpa_skip [85](#), [4241](#), [4243](#)
 - \g_tmpa_tl [103](#), [5050](#), [5050](#)
 - \g_tmpb_bool [38](#), [2119](#), [2122](#)
 - \g_tmpb_box [131](#), [6322](#), [6325](#)
 - \g_tmpb_clist [121](#), [5947](#), [5950](#)
 - \g_tmpb_dim [82](#), [4151](#), [4154](#)
 - \g_tmpb_fp [176](#), [14177](#), [14180](#)
 - \g_tmpb_int [73](#), [3900](#), [3903](#)
 - \g_tmpb_muskip [88](#), [4303](#), [4306](#)
 - \g_tmpb_prop [127](#), [5990](#), [5993](#)
 - \g_tmpb_seq [112](#), [5536](#), [5539](#)
 - \g_tmpb_skip [85](#), [4241](#), [4244](#)
 - \g_tmpb_tl [103](#), [5050](#), [5051](#)
 - \gdef [307](#)
 - \GetIdInfo [6](#), [139](#)
 - \global [291](#), [322](#)
 - \globaldefs [326](#)
 - \glueexpr [666](#)
 - \glueshrink [669](#)
 - \glueshrinkorder [671](#)
 - \gluestretch [668](#)
 - \gluestretchorder [670](#)
 - \gluetomu [672](#)
 - \group_align_safe_begin: ... [42](#), [2135](#),
[2304](#), [2304](#), [2975](#), [2993](#), [4517](#), [4817](#)
 - \group_align_safe_end:
..... [42](#), [2183](#), [2184](#), [2304](#), [2306](#),
[2957](#), [2967](#), [2972](#), [2990](#), [4526](#), [4843](#)
 - \group_begin: [9](#),
[774](#), [775](#), [1060](#), [1443](#), [1461](#), [1861](#),
[1995](#), [2009](#), [2310](#), [2325](#), [2574](#), [2587](#),
[2594](#), [2631](#), [2684](#), [2724](#), [2884](#), [3051](#),
[4449](#), [4467](#), [4618](#), [5246](#), [6349](#), [7278](#),
[7430](#), [7447](#), [7488](#), [7732](#), [7941](#), [8056](#),
[8066](#), [8853](#), [9182](#), [9187](#), [9213](#), [10008](#),
[10094](#), [10599](#), [10786](#), [10836](#), [10852](#),
[10880](#), [10939](#), [10953](#), [10981](#), [11314](#),

- 13889, 14401, 14491, 14605, 14635,
14655, 14672, 15472, 15479, 15492
- \group_end: 9, 774, 776,
1065, 1448, 1461, 1868, 1998, 2014,
2315, 2330, 2586, 2590, 2603, 2638,
2692, 2734, 2890, 3116, 4456, 4474,
4622, 4625, 5256, 5261, 6359, 7282,
7437, 7458, 7592, 7777, 7963, 8063,
8074, 8867, 9186, 9191, 9250, 10015,
10101, 10629, 10833, 10851, 10879,
10911, 10952, 10980, 11013, 11325,
13893, 14404, 14496, 14618, 14647,
14664, 14686, 15481, 15490, 15520
- \group_insert_after:N 9, 779, 779
- .groups:n 153
- ## H
- \halign 333
- \hangafter 511
- \hangindent 512
- \hbadness 573
- \hbox 568
- \hbox:n 132,
6361, 6361, 7072, 7127, 14520, 14711
- \hbox_gset:cn 6362
- \hbox_gset:cw 6372, 6384
- \hbox_gset:Nn 132, 6362, 6363, 6365
- \hbox_gset:Nw 133, 6372, 6374, 6377, 6383
- \hbox_gset_end: 133, 6372, 6379, 6385
- \hbox_gset_inline_begin:c 6380, 6384
- \hbox_gset_inline_begin:N 6380, 6383
- \hbox_gset_inline_end: 6380, 6385
- \hbox_gset_to_wd:cn 6366
- \hbox_gset_to_wd:Nnn 132, 6366, 6368, 6371
- \hbox_overlap_left:n 132, 6389, 6389
- \hbox_overlap_right:n 132, 6389, 6391, 14695
- \hbox_set:cn 6362
- \hbox_set:cw 6372, 6381
- \hbox_set:Nn 132, 6362, 6362, 6363, 6364,
6530, 6634, 6858, 6929, 7217, 14489,
14516, 14517, 14603, 14633, 14653,
14670, 14692, 14720, 14724, 14732,
14740, 14749, 14755, 14767, 14775,
14783, 14789, 14799, 14938, 14952
- \hbox_set:Nw 133, 6372, 6372, 6375, 6376, 6380, 6577
- \hbox_set_end: 133, 6372, 6378, 6382, 6581
- \hbox_set_inline_begin:c 6380, 6381
- \hbox_set_inline_begin:N 6380, 6380
- \hbox_set_inline_end: 6380, 6382
- \hbox_set_to_wd:cn 6366
- \hbox_set_to_wd:Nnn 132, 6366, 6366, 6369, 6370
- \hbox_to_wd:nn 132, 6386, 6386, 14702
- \hbox_to_zero:n 132, 6386, 6388, 6390, 6392
- \hbox_unpack:c 6393
- \hbox_unpack:N 133, 6393, 6393, 6395, 6862, 7009
- \hbox_unpack_clear:c 6393
- \hbox_unpack_clear:N 133, 6393, 6394, 6396
- \hcoffin_set:cn 6526
- \hcoffin_set:cw 6573
- \hcoffin_set:Nn 136, 6526,
6526, 6542, 7069, 7081, 7124, 7164
- \hcoffin_set:Nw 137, 6573, 6573, 6589
- \hcoffin_set_end: 137, 6573, 6578, 6588
- \hfil 476
- \hfill 478
- \hfilneg 477
- \hfuzz 575
- \hoffset 550
- \holdinginserts 553
- \hrule 489
- \hsize 514
- \hskip 479
- \hss 480
- \ht 618
- \hyphenation 604
- \hyphenchar 588
- \hyphenpenalty 506
- ## I
- pi 184
- \if 191, 342
- \if:w 24, 750, 756,
1054, 1921, 1924, 1925, 1979, 1982,
2927, 10209, 10213, 10235, 10328,
10360, 10379, 10445, 10459, 10476,
10496, 10943, 10958, 10970, 12927
- \if_bool:N 42, 2060, 2060
- \if_box_empty:N 135, 6294, 6296, 6310
- \if_case:w 74,
1284, 3156, 3160, 3654, 9439, 9592,
10741, 11057, 11101, 11418, 11557,
11632, 11656, 11708, 11966, 12207,
12427, 12454, 12610, 12645, 12751,
12767, 12783, 12799, 12815, 12831,
12856, 12903, 13025, 13048, 13095,
13110, 13125, 13140, 13155, 13170,

- 13610, 13627, 13679, 13762, 13777,
13829, 13842, 13897, 13947, 14015
- `\if_catcode:w` 24, 750, 758,
2613, 2618, 2623, 2628, 2635, 2641,
2646, 2651, 2656, 2661, 2666, 2676,
2709, 3009, 3014, 4953, 4995, 5013,
9342, 10045, 10168, 10486, 10533,
10790, 10930, 11058, 15503, 15504
- `\if_charcode:w` 24,
750, 757, 2681, 3011, 4928, 4934, 4988
- `\if_cs_exist:N`
. 24, 764, 764, 1088, 1116, 2717, 2936
- `\if_cs_exist:w`
764, 765, 787, 1097, 1125, 1272, 9663
- `\if_dim:w` 88, 3913, 3913, 3988, 4000, 4025
- `\if_eof:w` 168, 9069, 9069, 9077
- `\if_false:` 24, 750, 751,
2305, 3340, 4010, 4890, 4893, 4903,
5028, 5036, 5215, 5218, 5339, 5343
- `\if_hbox:N` 135, 6294, 6294, 6298
- `\if_int_compare:w`
.. 74, 777, 777, 1499, 1505, 1510,
2305, 2307, 2374, 2381, 2398, 2425,
2434, 2700, 2918, 3156, 3193, 3320,
3368, 3370, 3372, 3374, 3376, 3378,
3380, 3383, 4204, 9074, 9439, 9440,
9536, 9601, 9602, 9838, 9848, 9856,
9877, 9889, 9898, 9906, 9914, 9950,
9955, 10027, 10054, 10135, 10137,
10167, 10172, 10226, 10246, 10273,
10287, 10322, 10349, 10377, 10393,
10409, 10427, 10486, 10506, 10522,
10535, 10548, 10581, 10607, 10791,
10801, 10803, 10840, 10856, 10861,
10890, 10916, 10985, 10999, 11046,
11147, 11177, 11180, 11191, 11194,
11199, 11200, 11203, 11206, 11286,
11359, 11422, 11442, 11484, 11579,
11633, 11634, 11637, 11640, 11709,
11718, 11919, 12209, 12212, 12306,
12398, 12440, 12456, 12581, 12615,
12676, 12683, 12687, 12724, 12896,
12898, 12909, 12927, 12950, 12982,
12985, 13028, 13056, 13057, 13065,
13066, 13080, 13198, 13242, 13631,
13668, 13677, 13712, 13796, 13799,
13924, 13998, 14024, 14027, 14189
- `\if_int_odd:w` 74,
3156, 3159, 3420, 3428, 9860, 9874,
9886, 11692, 13081, 13468, 13507,
13517, 13560, 13584, 13753, 15502
- `\if_meaning:w` .. 24, 750, 759, 921, 936,
953, 1000, 1005, 1014, 1085, 1103,
1113, 1131, 1302, 1313, 1417, 1579,
1633, 1634, 1871, 1892, 1901, 2093,
2158, 2177, 2360, 2366, 2392, 2405,
2413, 2671, 2714, 2756, 2759, 2778,
2781, 2798, 2801, 2816, 2819, 2834,
2837, 2910, 3002, 3043, 3174, 3209,
3214, 3215, 3352, 3969, 4018, 4571,
4583, 4595, 4606, 4621, 4835, 4896,
4979, 5233, 5251, 5270, 5278, 5647,
5662, 5684, 5700, 6154, 6192, 9428,
9441, 9452, 9462, 9472, 9617, 9619,
9770, 9837, 9847, 9859, 9874, 9875,
9886, 9887, 9897, 9913, 9932, 9967,
9970, 9986, 9993, 10046, 10186,
10299, 10305, 10534, 10719, 10722,
10725, 10821, 11139, 11166, 11167,
11168, 11169, 11170, 11171, 11267,
11279, 11280, 11308, 11319, 11329,
11386, 11393, 11402, 11419, 11453,
11458, 11472, 11518, 11525, 11601,
11613, 11712, 11715, 11726, 11777,
11850, 11918, 11921, 11928, 12180,
12191, 12371, 12381, 12424, 12522,
12601, 12650, 12664, 12853, 12865,
12877, 12880, 12883, 12908, 13009,
13013, 13510, 13563, 13625, 13626,
13628, 13629, 13648, 13665, 13731,
13829, 13896, 13946, 14014, 14063,
14068, 14195, 14250, 15192, 15505
- `\if_mode_horizontal:` .. 24, 760, 761, 2299
- `\if_mode_inner:` 24, 760, 763, 2301
- `\if_mode_math:` 24, 760, 760, 2303
- `\if_mode_vertical:` 24, 760, 762, 2297
- `\if_predicate:w` 42, 2060, 2061, 2127
- `\if_true:` 24, 750, 750
- `\if_vbox:N` 135, 6294, 6295, 6300
- `\ifcase` 343
- `\ifcat` 344
- `\ifcsname` 627
- `\ifdefined` 626
- `\ifdim` 347
- `\ifeof` 348
- `\iffalse` 353
- `\iffontchar` 656
- `\ifhbox` 349
- `\ifhmode` 355
- `\ifinner` 358

<code>\ifmmode</code>	356	<code>\int_const:cn</code>	<u>3239</u> , 3817, 3818, 3819, 3820, 3821, 3822, 3823, 3824, 3825, 3826, 3827, 3828, 3829, 3830
<code>\ifnum</code>	345	<code>\int_const:Nn</code>	63, <u>3239</u> , 3239, 3259, 3881, 3882, 3883, 3884, 3885, 3886, 3887, 3888, 3889, 3890, 3891, 3892, 3893, 3894, 3895, 3896, 3897, 3898, 3899, 9411, 9510, 9511, 9512, 9516, 9517, 9518, 9523, 9524, 9525
<code>\ifodd</code>	176, 212, 346	<code>\int_decr:c</code>	<u>3299</u>
<code>\iftrue</code>	354	<code>\int_decr:N</code>	64, <u>3299</u> , 3301, 3306, 3308
<code>\ifvbox</code>	350	<code>\int_div_round:nn</code>	62, <u>3199</u> , 3220
<code>\ifvmode</code>	357	<code>\int_div_truncate:nn</code>	63, <u>3199</u> , 3199, 3556, 3646
<code>\ifvoid</code>	351	<code>\int_do_until:nn</code> ...	68, <u>3434</u> , 3456, 3460
<code>\ifx</code>	14, 63, 69, 73, 352	<code>\int_do_until:nNnn</code> ..	67, <u>3462</u> , 3484, 3488
<code>\ignorespaces</code>	400	<code>\int_do_while:nn</code> ...	68, <u>3434</u> , 3450, 3454
<code>\immediate</code>	362	<code>\int_do_while:nNnn</code> ..	68, <u>3462</u> , 3478, 3482
<code>min</code>	181	<code>\int_eval:n</code> 62, 1310, 1326, <u>3161</u> , 3162, 3165, 3392, 3397, 3402, 3407, 3502, 3510, 3538, 3624, 3693, 3703, 3762, 3776, 3780, 3783, 3798, 3807, 4742, 4747, 5449, 5883, 5892, 6341, 9035, 9128, 9417, 14824, 14837, 14859, 15218, 15234, 15386, 15451, 15467	
<code>sin</code>	182	<code>\int_from_alph:n</code>	71, <u>3760</u> , 3760
<code>sind</code>	182	<code>\int_from_base:nn</code>	72, <u>3781</u> , 3781, 3812, 3814, 3816
<code>\indent</code>	496	<code>\int_from_binary:n</code>	71, <u>3811</u> , 3811
<code>\initcatcodetable</code>	715	<code>\int_from_hexadecimal:n</code> ..	71, <u>3811</u> , 3813
<code>.initial:n</code>	153	<code>\int_from_octal:n</code>	72, <u>3811</u> , 3815
<code>.initial:o</code>	153	<code>\int_from_roman:n</code>	72, <u>3831</u> , 3831
<code>.initial:V</code>	153	<code>\int_gadd:cn</code>	<u>3287</u>
<code>.initial:x</code>	153	<code>\int_gadd:Nn</code>	64, <u>3287</u> , 3291, 3296, 14332, 14364
<code>\input</code>	370	<code>\int_gdecr:c</code>	<u>3299</u>
<code>\input@path</code>	8885, 8888, 8903	<code>\int_gdecr:N</code> 64, <u>3299</u> , 3305, 3310, 3536, 4706, 5426, 5844, 6209, 8151, 15164	
<code>\inputlineno</code>	372	<code>\int_gincr:c</code>	<u>3299</u>
<code>\insert</code>	552	<code>\int_gincr:N</code>	64, <u>3299</u> , 3303, 3309, 3515, 3522, 4701, 5418, 5838, 6204, 8145, 15155
<code>\insertpenalties</code>	555	<code>.int_gset:c</code>	153
<code>\int_abs:n</code>	62, <u>3167</u> , 3167	<code>\int_gset:cn</code>	<u>3311</u>
<code>\int_add:cn</code>	<u>3287</u>	<code>.int_gset:N</code>	153
<code>\int_add:Nn</code>	64, <u>3287</u> , 3287, 3292, 3295, 9264, 9277, 9315	<code>\int_gset:Nn</code> 64, 3246, 3256, <u>3311</u> , 3313, 3315	
<code>\int_case:nn</code>	3389, 3404, 3545, 3551	<code>\int_gset_eq:cc</code>	<u>3279</u>
<code>\int_case:nnF</code>	3399, 3904, 5461, 5910	<code>\int_gset_eq:cN</code>	<u>3279</u>
<code>\int_case:nnn</code>	<u>3904</u> , 3904	<code>\int_gset_eq:Nc</code>	<u>3279</u>
<code>\int_case:nnT</code>	3394		
<code>\int_case:nnTF</code>	67, <u>3389</u> , 3389		
<code>\int_compare:n</code>	3333		
<code>\int_compare:nF</code>	3444, 3459		
<code>\int_compare:nNn</code>	3381		
<code>\int_compare:nNnF</code>	3472, 3487, 3506		
<code>\int_compare:nNnT</code>	3464, 3481, 4139, 14365, 15220, 15453		
<code>\int_compare:nNnTF</code>	65, 2209, 3241, 3243, <u>3381</u> , 3413, 3492, 3495, 3541, 3627, 3633, 3780, 3804, 3808, 3858, 9265, 13858, 13973, 13975, 14333, 14820, 14822, 14827, 14835, 14855, 15232, 15465		
<code>\int_compare:nT</code> ..	3436, 3453, 9049, 9142		
<code>\int_compare:nTF</code>	66, <u>3333</u>		
<code>\int_compare_p:n</code>	66, <u>3333</u>		
<code>\int_compare_p:nNn</code>	65, <u>3381</u>		

`\int_gset_eq:NN` [63](#), [3279](#), [3282](#), [3283](#), [3284](#), [7475](#)
`\int_gsub:cn` [3287](#)
`\int_gsub:Nn` .. [64](#), [3287](#), [3293](#), [3298](#), [14372](#)
`\int_gzero:c` [3269](#)
`\int_gzero:N` ... [63](#), [3269](#), [3270](#), [3272](#), [3276](#)
`\int_gzero_new:c` [3273](#)
`\int_gzero_new:N` ... [63](#), [3273](#), [3275](#), [3278](#)
`\int_if_even:n` [3426](#)
`\int_if_even:nTF` [67](#), [3418](#)
`\int_if_even_p:n` [67](#), [3418](#)
`\int_if_exist:c` [3286](#)
`\int_if_exist:cF` [3847](#), [3854](#), [3856](#)
`\int_if_exist:cTF` [3285](#)
`\int_if_exist:N` [3285](#)
`\int_if_exist:NTF` .. [64](#), [3274](#), [3276](#), [3285](#)
`\int_if_exist_p:c` [3285](#)
`\int_if_exist_p:N` [64](#), [3285](#)
`\int_if_odd:n` [3418](#)
`\int_if_odd:nTF` [67](#), [3418](#), [12295](#)
`\int_if_odd_p:n` [67](#), [3418](#)
`\int_incr:c` [3299](#)
`\int_incr:N` [64](#), [3299](#), [3299](#),
[3304](#), [3307](#), [8295](#), [8354](#), [8795](#), [9283](#)
`\int_max:nn` .. [63](#), [3167](#), [3175](#), [12156](#), [13222](#)
`\int_min:nn` [63](#), [3167](#), [3183](#)
`\int_mod:nn` ... [63](#), [3199](#), [3222](#), [3546](#), [3637](#)
`\int_new:c` [3231](#)
`\int_new:N` [63](#), [2343](#), [3231](#), [3232](#),
[3238](#), [3245](#), [3255](#), [3274](#), [3276](#), [3900](#),
[3901](#), [3902](#), [3903](#), [8051](#), [8163](#), [9170](#),
[9172](#), [9173](#), [9174](#), [9175](#), [14325](#), [14327](#)
`.int_set:c` [153](#)
`\int_set:cn` [3311](#)
`.int_set:N` [153](#)
`\int_set:Nn` [64](#), [3311](#), [3311](#), [3313](#), [3314](#),
[6350](#), [6351](#), [8299](#), [8358](#), [8799](#), [9094](#),
[9171](#), [9220](#), [9234](#), [9262](#), [9290](#), [14326](#)
`\int_set_eq:cc` [3279](#)
`\int_set_eq:cN` [3279](#)
`\int_set_eq:Nc` [3279](#)
`\int_set_eq:NN` [63](#), [3279](#), [3279](#),
[3280](#), [3281](#), [6352](#), [9092](#), [9188](#), [9214](#)
`\int_show:c` [3877](#), [3878](#)
`\int_show:N` [72](#), [3877](#), [3877](#)
`\int_show:n` [72](#), [3879](#), [3879](#)
`\int_step_function:nnnN`
..... [69](#), [3490](#), [3490](#), [3535](#)
`\int_step_inline:nnnn`
..... [69](#), [3513](#), [3513](#), [14444](#), [14449](#)
`\int_step_variable:nnnNn` . [69](#), [3513](#), [3520](#)
`\int_sub:cn` [3287](#)
`\int_sub:Nn` [64](#), [3287](#), [3289](#), [3294](#), [3297](#), [9321](#)
`\int_to_Alph:n` [70](#), [3559](#), [3591](#)
`\int_to_alph:n` [70](#), [3559](#), [3559](#)
`\int_to_arabic:n` [69](#), [3538](#), [3538](#)
`\int_to_base:nn`
..... [71](#), [3623](#), [3623](#), [3685](#), [3687](#), [3689](#)
`\int_to_binary:n` [70](#), [3684](#), [3684](#)
`\int_to_hexadecimal:n` ... [71](#), [3684](#), [3686](#)
`\int_to_octal:n` [71](#), [3684](#), [3688](#)
`\int_to_Roman:n` [71](#), [3690](#), [3700](#)
`\int_to_roman:n` [71](#), [3690](#), [3690](#)
`\int_to_symbols:nnn`
..... [70](#), [3539](#), [3539](#), [3555](#), [3561](#), [3593](#)
`\int_until_do:nn` ... [68](#), [3434](#), [3442](#), [3447](#)
`\int_until_do:nNnn` .. [68](#), [3462](#), [3470](#), [3475](#)
`\int_use:c` [3316](#), [3317](#)
`\int_use:N` [65](#), [3170](#), [3178](#),
[3179](#), [3186](#), [3187](#), [3201](#), [3203](#), [3204](#),
[3316](#), [3316](#), [3317](#), [3336](#), [3518](#), [3525](#),
[3880](#), [4702](#), [4704](#), [5419](#), [5425](#), [5839](#),
[5841](#), [6203](#), [6211](#), [7397](#), [7846](#), [8096](#),
[8121](#), [8147](#), [8149](#), [8300](#), [8359](#), [8800](#),
[9094](#), [9423](#), [9626](#), [9873](#), [9949](#), [9953](#),
[9992](#), [10131](#), [10261](#), [10272](#), [10321](#),
[10352](#), [10358](#), [10359](#), [10405](#), [10415](#),
[10417](#), [10433](#), [10435](#), [10458](#), [10597](#),
[11483](#), [11512](#), [11514](#), [11535](#), [11537](#),
[11546](#), [11548](#), [11570](#), [11577](#), [11583](#),
[11593](#), [11595](#), [11668](#), [11670](#), [11686](#),
[11688](#), [11748](#), [11756](#), [11758](#), [11760](#),
[11762](#), [11764](#), [11766](#), [11768](#), [11787](#),
[11789](#), [11799](#), [11801](#), [11827](#), [11830](#),
[11838](#), [11840](#), [11860](#), [11863](#), [11866](#),
[11869](#), [11877](#), [11880](#), [11883](#), [11886](#),
[11893](#), [11895](#), [11901](#), [11909](#), [11911](#),
[11913](#), [11939](#), [11941](#), [11950](#), [11952](#),
[11978](#), [11983](#), [11985](#), [11992](#), [11995](#),
[11998](#), [12001](#), [12004](#), [12007](#), [12017](#),
[12029](#), [12037](#), [12039](#), [12049](#), [12051](#),
[12058](#), [12067](#), [12069](#), [12072](#), [12075](#),
[12078](#), [12081](#), [12095](#), [12097](#), [12105](#),
[12107](#), [12115](#), [12117](#), [12126](#), [12129](#),
[12132](#), [12139](#), [12154](#), [12172](#), [12175](#),
[12231](#), [12245](#), [12247](#), [12253](#), [12266](#),
[12268](#), [12270](#), [12294](#), [12310](#), [12317](#),
[12318](#), [12361](#), [12406](#), [12408](#), [12446](#),
[12476](#), [12478](#), [12480](#), [12492](#), [12505](#),
[12510](#), [12512](#), [12518](#), [12535](#), [12536](#),

- 12537, 12538, 12539, 12540, 12545,
 12547, 12549, 12551, 12553, 12557,
 12559, 12561, 12563, 12565, 12567,
 12589, 12597, 12679, 12732, 12962,
 12964, 12967, 12970, 12973, 12976,
 12992, 13044, 13196, 13228, 13237,
 13419, 13420, 13443, 13453, 13462,
 13480, 13489, 13496, 13550, 13594,
 13647, 13664, 13719, 13730, 13741,
 13907, 13929, 14067, 14275, 15157
 \int_while_do:nn ... 68, [3434](#), 3434, 3439
 \int_while_do:nnn ... 68, [3462](#), 3462, 3467
 \int_zero:c [3269](#)
 \int_zero:N 63, [3269](#), 3269, 3271, 3274,
 8292, 8351, 8785, 9236, 9238, 9309
 \int_zero_new:c [3273](#)
 \int_zero_new:N 63, [3273](#), 3273, 3277
 \interactionmode 654
 \interlinepenalties 675
 \interlinepenalty 534
 \ior_close:c [9047](#)
 \ior_close:N
 ... 164, 8879, 9026, [9047](#), 9047, 9058
 \ior_get:NN 164, [9086](#), 9086, 15150
 \ior_get_str:NN .. 165, [9088](#), 9088, 15152
 \ior_if_eof:N 9070
 \ior_if_eof:Nf 8896, 15162, 15169
 \ior_if_eof:Ntf 165, 8876, [9070](#)
 \ior_if_eof_p:N 165, [9070](#)
 \ior_list_streams: 164, [9059](#), 9059
 \ior_map_break:
 195, [15145](#), 15145, 15146, 15148, 15163
 \ior_map_break:n 196, [15145](#), 15147
 \ior_map_inline:Nn 195, [15149](#), 15149
 \ior_new:c [8997](#)
 \ior_new:N ... 163, [8997](#), 8997, 8998, 9097
 \ior_open:cn [8999](#)
 \ior_open:cnTF [9009](#)
 \ior_open:Nn .. 163, [8999](#), 8999, 9001, 9009
 \ior_open:Nnf 9012
 \ior_open:Nnt 9011
 \ior_open:Nntf 163, [9009](#), 9013
 \ior_str_map_inline:Nn 195, [15149](#), 15151
 \iow_char:N 166, [9169](#), 9169, 12850
 \iow_close:c [9140](#)
 \iow_close:N .. 164, 9119, [9140](#), 9140, 9151
 \iow_indent:n 167, 7810,
 8728, [9204](#), 9204, 9223, 9799, 9811
 \iow_list_streams: 164, [9152](#), 9152
 \iow_log:n 165, 7464, 7465, 7466,
 7589, 8974, 8975, 8976, [9164](#), 9165
 \iow_log:x [1155](#),
 1155, 1194, 2001, 7336, [9164](#), 9164
 \iow_new:c [9112](#)
 \iow_new:N 163, [9112](#), 9112, 9113
 \iow_newline:
 166, 7426, 7442, 7444, [9168](#), 9168, 9231
 \iow_now:Nn 165, [9161](#), 9161, 9163, 9165, 9167
 \iow_now:Nx [9161](#), 9164, 9166
 \iow_open:cn [9114](#)
 \iow_open:Nn 164, [9114](#), 9114, 9116
 \iow_shipout:Nn ... 165, [9158](#), 9158, 9160
 \iow_shipout:Nx [9158](#)
 \iow_shipout_x:Nn . 166, [9155](#), 9155, 9157
 \iow_shipout_x:Nx [9155](#)
 \iow_term:n
 165, 7470, 7471, 7472, 7997, [9164](#), 9167
 \iow_term:x .. 1155, 1157, 7440, [9164](#), 9166
 \iow_wrap:nnnN . 167, 7418, 7419, 7465,
 7471, 7587, 7995, 8019, [9211](#), 9211
- J**
- \jobname 609
- K**
- \kern 487
 \keys_define:nn ... 151, [8178](#), 8178, 8726
 \keys_if_choice_exist:nnn 8700
 \keys_if_choice_exist:nnnTF .. 160, [8700](#)
 \keys_if_choice_exist_p:nnn .. 160, [8700](#)
 \keys_if_exist:nn 8694
 \keys_if_exist:nnTF 160, [8694](#)
 \keys_if_exist_p:nn 160, [8694](#)
 \keys_set:nn ... 157, 8328, 8332, 8335,
 8500, 8500, 8508, 8520, 8536, 8545
 \keys_set:no 8500
 \keys_set:nV 8500
 \keys_set:nv 8500
 \keys_set_filter:nnn [8524](#), 8527, 8531, 8539
 \keys_set_filter:nnnN 159, [8524](#), 8524, 8530
 \keys_set_filter:nnV [8524](#)
 \keys_set_filter:nnv\keys_set_filter:nno
 [8524](#)
 \keys_set_filter:nnVN [8524](#)
 \keys_set_filter:nnvN\keys_set_filter:nnoN
 [8524](#)
 \keys_set_groups:nnn 160, [8524](#), 8540, 8548
 \keys_set_groups:nnV [8524](#)

\keys_set_groups:nnv\keys_set_groups:nno	\l__box_right_new_dim .
..... 8524	14482, 14485,
\keys_set_known:nn 8510, 8513, 8517, 8523	14530, 14564, 14575, 14586, 14597,
\keys_set_known:nnN 158, 8510, 8510, 8516	14624, 14683, 14702, 14704, 14710
\keys_set_known:no 8510	\l__box_scale_x_fp
\keys_set_known:noN 8510	. 193, 14599, 14599, 14610, 14645,
\keys_set_known:nV 8510	14660, 14662, 14673, 14684, 14700
\keys_set_known:nv 8510	\l__box_scale_y_fp 193, 14599,
\keys_set_known:nVN 8510	14600, 14612, 14626, 14628, 14640,
\keys_set_known:nvN 8510	14645, 14662, 14674, 14680, 14682
\keys_show:nn 160, 8706, 8706	\l__box_sin_fp 193, 14476,
\keyval_parse:NNn	14477, 14493, 14505, 14540, 14550
..... 161, 8143, 8143, 8183, 8505	\l__box_top_dim
 14478, 14478, 14501, 14557,
	14561, 14570, 14572, 14581, 14585,
	14590, 14596, 14606, 14615, 14628,
	14636, 14643, 14656, 14675, 14680
	\l__box_top_new_dim
	14482, 14482, 14527, 14558, 14569,
	14580, 14591, 14627, 14679, 14698
	\l__cctab_internal_tl
	.. 14360, 14374, 14375, 14376, 14398
	\l__clist_internal_clist
 5549, 5549, 5625,
	5626, 5638, 5639, 5785, 5786, 5849,
	5850, 5866, 5867, 5943, 5944, 7918
	\l__clist_internal_remove_clist
	.. 5729, 5729, 5736, 5739, 5740, 5742
	\l__coffin_aligned_coffin
 6633, 6635, 6857,
	6858, 6862, 6868, 6870, 6871, 6887,
	6888, 6894, 6895, 6896, 6897, 6898,
	6900, 6902, 6906, 6907, 6912, 6913,
	6914, 6915, 6916, 6950, 6965, 7011,
	7012, 7217, 7224, 7226, 7228, 7230
	\l__coffin_aligned_internal_coffin .
 6633, 6636, 6929, 6936
	\l__coffin_bottom_corner_dim
 14920, 14922, 14944, 14948,
	15018, 15029, 15030, 15050, 15058
	\l__coffin_bounding_prop
 14918, 14918, 14933, 14961,
	14963, 14966, 14968, 14974, 15037
	\l__coffin_bounding_shift_dim 14919,
	14919, 14942, 15036, 15042, 15043
	\l__coffin_cos_fp
	.. 14916, 14917, 14927, 15001, 15010
	\l__coffin_display_coffin
 7015, 7015, 7143, 7149,
	7219, 7220, 7225, 7227, 7229, 7230

L

\l@expl@log@functions@bool .. 1186, 7328	
\l__box_angle_fp	
193, 14475, 14475, 14492, 14493, 14494	
\l__box_bottom_dim	
.... 14478, 14479, 14502, 14559,	
14563, 14568, 14574, 14579, 14583,	
14592, 14594, 14607, 14615, 14626,	
14637, 14643, 14657, 14676, 14682	
\l__box_bottom_new_dim	
14482, 14483, 14528, 14560, 14571,	
14582, 14593, 14625, 14681, 14699	
\l__box_cos_fp 193, 14476, 14476,	
14494, 14507, 14512, 14539, 14551	
\l__box_internal_box 193, 14486,	
14486, 14516, 14517, 14523, 14527,	
14528, 14529, 14531, 14692, 14698,	
14699, 14705, 14710, 14714, 14724,	
14732, 14735, 14737, 14740, 14743,	
14745, 14747, 14749, 14750, 14751,	
14752, 14755, 14757, 14758, 14760,	
14762, 14767, 14775, 14778, 14780,	
14783, 14784, 14785, 14789, 14790,	
14791, 14799, 14802, 14804, 14806	
\l__box_left_dim	
14478, 14480, 14504, 14559, 14561,	
14570, 14574, 14579, 14585, 14590,	
14594, 14609, 14639, 14659, 14678	
\l__box_left_new_dim	
..... 14482, 14484, 14519,	
14530, 14562, 14573, 14584, 14595	
\l__box_right_dim 14478, 14481, 14503,	
14557, 14563, 14568, 14572, 14581,	
14583, 14592, 14596, 14608, 14611,	
14638, 14658, 14661, 14677, 14684	

\l__coffin_display_coord_coffin 7015, 7016,
 7081, 7101, 7117, 7164, 7184, 7203
 \l__coffin_display_font_tl 7060, 7060, 7062, 7065, 7089, 7172
 \l__coffin_display_handles_prop 7018, 7018, 7019, 7021, 7023, 7025,
 7027, 7029, 7031, 7033, 7035, 7037,
 7039, 7041, 7043, 7045, 7047, 7049,
 7051, 7053, 7092, 7096, 7175, 7179
 \l__coffin_display_offset_dim 7055,
 7055, 7056, 7118, 7119, 7204, 7205
 \l__coffin_display_pole_coffin 7015, 7017, 7069, 7080, 7124, 7162
 \l__coffin_display_poles_prop 7059, 7059,
 7134, 7139, 7142, 7144, 7146, 7153
 \l__coffin_display_x_dim 7057, 7057, 7159, 7214
 \l__coffin_display_y_dim 7057, 7058, 7160, 7216
 \l__coffin_error_bool 6474, 6474, 6752,
 6756, 6770, 6785, 6816, 7155, 7157
 \l__coffin_internal_box . 6452, 6452,
 6561, 6565, 6569, 6608, 6613, 6618,
 14938, 14947, 14949, 14950, 14952
 \l__coffin_internal_dim 6452, 6453, 6863, 6865, 6866,
 14965, 14967, 14969, 15097, 15100
 \l__coffin_internal_tl 6452, 6454, 6461, 6462,
 6463, 6464, 6465, 6466, 6467, 6468,
 6469, 6470, 6471, 6948, 6949, 6951,
 7093, 7094, 7097, 7098, 7106, 7111,
 7176, 7177, 7180, 7181, 7190, 7195
 \l__coffin_left_corner_dim 14920, 14920, 14942, 14951,
 15019, 15025, 15026, 15049, 15057
 \l__coffin_offset_x_dim 6475, 6475, 6860,
 6861, 6864, 6872, 6874, 6876, 6882,
 6885, 6905, 6925, 6933, 7213, 7221
 \l__coffin_offset_y_dim 6475, 6476, 6875, 6877, 6882,
 6885, 6905, 6927, 6934, 7215, 7222
 \l__coffin_pole_a_tl 6477, 6477, 6750, 6755, 6974,
 6977, 6978, 6981, 7136, 7138, 7141
 \l__coffin_pole_b_tl 6477, 6478, 6751, 6755, 6975, 6977,
 6979, 6981, 7137, 7138, 7140, 7141
 \l__coffin_right_corner_dim . 14920,
 14921, 14951, 15017, 15027, 15028
 \l__coffin_scale_x_fp 15062, 15062, 15068,
 15084, 15094, 15096, 15102, 15110
 \l__coffin_scale_y_fp . 15062, 15063,
 15070, 15095, 15096, 15100, 15112
 \l__coffin_scaled_total_height_dim 15064, 15064, 15099, 15104
 \l__coffin_scaled_width_dim 15064, 15065, 15101, 15104
 \l__coffin_sin_fp 14916, 14916, 14926, 15002, 15009
 \l__coffin_slope_x_fp 6472, 6472, 6810, 6815, 6823, 6829
 \l__coffin_slope_y_fp 6472, 6473, 6812, 6815, 6824, 6829
 \l__coffin_top_corner_dim . . 14920,
 14923, 14948, 15016, 15031, 15032
 \l__coffin_x_dim 6479, 6479, 6759, 6768, 6788,
 6791, 6798, 6805, 6807, 6818, 6833,
 6922, 6926, 6945, 6953, 7159, 7211,
 14973, 14975, 14979, 14981, 14985,
 14990, 15116, 15118, 15122, 15125
 \l__coffin_x_prime_dim . . 6479, 6481,
 6922, 6926, 7211, 7214, 14987, 14991
 \l__coffin_y_dim 6479, 6480,
 6760, 6773, 6776, 6783, 6800, 6834,
 6923, 6928, 6946, 6953, 7160, 7212,
 14973, 14975, 14979, 14981, 14985,
 14990, 15116, 15118, 15122, 15125
 \l__coffin_y_prime_dim . . 6479, 6482,
 6923, 6928, 7212, 7216, 14987, 14992
 \l__exp_internal_tl . . . 34, 841, 841,
 845, 846, 1597, 1616, 1617, 1792, 1793
 \l__expl_status_stack_tl 204
 \l__file_internal_name_ior 168
 \l__file_internal_name_tl 168, 8843, 8843,
 8856, 8857, 8858, 8859, 8862, 8863,
 8868, 8909, 8910, 8916, 8917, 8922,
 9004, 9005, 9007, 9016, 9017, 9020
 \l__file_internal_seq . . 8848, 8849,
 8888, 8890, 8962, 8968, 8973, 8975
 \l__file_internal_tl 8842, 8842, 8944, 8945
 \l__file_saved_search_path_seq 8845, 8846, 8887, 8904

- \l__file_search_path_seq
..... [8844](#), [8844](#), [8887](#), [8889](#),
[8890](#), [8893](#), [8904](#), [8952](#), [8953](#), [8958](#)
- \l__fp_division_by_zero_flag_token .
..... [9669](#), [9670](#)
- \l__fp_invalid_operation_flag_token
..... [9669](#), [9669](#)
- \l__fp_overflow_flag_token .. [9669](#), [9671](#)
- \l__fp_underflow_flag_token . [9669](#), [9672](#)
- \l__ior_internal_tl
..... [15149](#), [15168](#), [15171](#), [15175](#)
- \l__ior_stream_tl
..... [8992](#), [8992](#), [9027](#), [9035](#), [9043](#)
- \l__iow_current_indentation_int
..... [9173](#), [9175](#),
[9236](#), [9278](#), [9293](#), [9315](#), [9321](#), [9323](#)
- \l__iow_current_indentation_tl [9176](#),
[9178](#), [9237](#), [9276](#), [9296](#), [9316](#), [9322](#)
- \l__iow_current_line_int
..... [9173](#), [9173](#), [9238](#),
[9264](#), [9265](#), [9277](#), [9283](#), [9290](#), [9309](#)
- \l__iow_current_line_tl
..... [9176](#), [9176](#), [9239](#), [9275](#),
[9281](#), [9289](#), [9295](#), [9308](#), [9310](#), [9328](#)
- \l__iow_current_word_int
..... [9173](#), [9174](#), [9262](#), [9264](#), [9292](#)
- \l__iow_current_word_tl . [9176](#), [9177](#),
[9255](#), [9256](#), [9263](#), [9276](#), [9282](#), [9296](#)
- \l__iow_line_start_bool
.. [9181](#), [9181](#), [9240](#), [9272](#), [9274](#), [9311](#)
- \l__iow_newline_tl [9180](#), [9180](#),
[9231](#), [9232](#), [9233](#), [9235](#), [9289](#), [9308](#)
- \l__iow_stream_tl
..... [9105](#), [9105](#), [9120](#), [9128](#), [9136](#)
- \l__iow_target_count_int
..... [9172](#), [9172](#), [9234](#), [9265](#)
- \l__iow_wrap_tl [9179](#), [9179](#), [9226](#), [9229](#),
[9243](#), [9245](#), [9251](#), [9288](#), [9307](#), [9327](#)
- \l__kernel_expl_bool
..... [7](#), [138](#), [249](#), [264](#), [278](#), [282](#), [283](#)
- \l__keys_filtered_bool .. [8172](#), [8173](#),
[8534](#), [8543](#), [8600](#), [8606](#), [8627](#), [8632](#)
- \l__keys_groups_clist
.. [8165](#), [8165](#), [8320](#), [8322](#), [8597](#), [8616](#)
- \l__keys_module_tl [8167](#),
[8167](#), [8179](#), [8182](#), [8184](#), [8212](#), [8332](#),
[8501](#), [8504](#), [8506](#), [8562](#), [8664](#), [8667](#)
- \l__keys_no_value_bool
... [8168](#), [8168](#), [8189](#), [8194](#), [8232](#),
[8551](#), [8556](#), [8573](#), [8583](#), [8639](#), [8685](#)
- \l__keys_only_known_bool
..... [8169](#), [8169](#), [8519](#), [8521](#), [8661](#)
- \l__keys_property_tl
... [8171](#), [8171](#), [8200](#), [8203](#), [8206](#),
[8217](#), [8228](#), [8235](#), [8236](#), [8239](#), [8243](#)
- \l__keys_selective_bool [8172](#),
[8172](#), [8533](#), [8537](#), [8542](#), [8546](#), [8564](#)
- \l__keys_selective_seq
..... [8174](#), [8174](#), [8535](#), [8544](#), [8614](#)
- \l__keys_tmp_bool
..... [8177](#), [8177](#), [8613](#), [8620](#), [8625](#)
- \l__keys_unused_clist [8175](#),
[8175](#), [8512](#), [8514](#), [8526](#), [8528](#), [8682](#)
- \l__keyval_key_tl
.. [8052](#), [8052](#), [8097](#), [8111](#), [8115](#), [8122](#)
- \l__keyval_parse_tl . [8054](#), [8055](#), [8071](#),
[8075](#), [8093](#), [8118](#), [8127](#), [8131](#), [8140](#)
- \l__keyval_sanitise_tl [8054](#),
[8054](#), [8067](#), [8068](#), [8069](#), [8070](#), [8073](#)
- \l__keyval_value_tl [8052](#),
[8053](#), [8124](#), [8126](#), [8129](#), [8139](#), [8141](#)
- \l__msg_class_loop_seq .. [7602](#), [7602](#),
[7692](#), [7700](#), [7709](#), [7710](#), [7713](#), [7715](#)
- \l__msg_class_tl
[7598](#), [7598](#), [7614](#), [7626](#), [7647](#), [7651](#),
[7654](#), [7662](#), [7701](#), [7703](#), [7705](#), [7718](#)
- \l__msg_current_class_tl
..... [7598](#), [7599](#), [7609](#),
[7646](#), [7651](#), [7654](#), [7662](#), [7691](#), [7705](#)
- \l__msg_hierarchy_seq
..... [7601](#), [7601](#), [7629](#), [7639](#), [7644](#)
- \l__msg_internal_tl
..... [7311](#), [7311](#), [8023](#), [8024](#), [8026](#)
- \l__msg_redirect_prop
..... [7600](#), [7600](#), [7626](#), [7671](#), [7674](#)
- \l__peek_search_tl
.. [2946](#), [2946](#), [2964](#), [2985](#), [3025](#), [3035](#)
- \l__peek_search_token
..... [2945](#), [2945](#), [2963](#), [2984](#), [3002](#)
- \l__prop_internal_tl [127](#), [5962](#),
[5962](#), [6082](#), [6088](#), [6093](#), [6111](#), [6122](#)
- \l__seq_internal_a_tl
... [5069](#), [5069](#), [5106](#), [5110](#), [5116](#),
[5121](#), [5123](#), [5220](#), [5225](#), [5247](#), [5251](#)
- \l__seq_internal_b_tl
.. [5069](#), [5070](#), [5216](#), [5220](#), [5250](#), [5251](#)
- \l__seq_remove_seq
.. [5188](#), [5188](#), [5195](#), [5198](#), [5199](#), [5201](#)
- \l__tl_internal_a_tl [4616](#), [4619](#), [4621](#), [4629](#)
- \l__tl_internal_b_tl [4616](#), [4620](#), [4621](#), [4630](#)

- \l_char_active_seq 53, 2591, 2591, 2604, 8854
- \l_char_special_seq . 53, 2591, 2608, 2609
- \l_iow_line_count_int 167, 9170, 9170, 9171, 9235
- \l_keys_choice_int .. 156, 8163, 8163, 8292, 8295, 8299, 8300, 8351, 8354, 8358, 8359, 8785, 8795, 8799, 8800
- \l_keys_choice_tl 156, 8163, 8164, 8298, 8357, 8798
- \l_keys_key_tl 158, 8166, 8166, 8258, 8274, 8561, 8562, 8684
- \l_keys_path_tl 158, 8170, 8170, 8206, 8212, 8222, 8225, 8240, 8251, 8253, 8255, 8267, 8269, 8271, 8281, 8283, 8286, 8296, 8313, 8314, 8318, 8321, 8326, 8331, 8335, 8340, 8342, 8345, 8355, 8366, 8368, 8369, 8370, 8376, 8407, 8562, 8577, 8587, 8594, 8596, 8641, 8649, 8651, 8658, 8667, 8691, 8692, 8766, 8769, 8771, 8782, 8790, 8796, 8802
- \l_keys_value_tl ... 158, 8176, 8176, 8587, 8642, 8643, 8645, 8676, 8686
- \l_peek_token 58, 2943, 2943, 2952, 3002, 3014, 3034, 3043, 15503, 15504, 15505, 15508
- \l_tmpa_bool 38, 2119, 2119
- \l_tmpa_box 131, 6322, 6322
- \l_tmpa_clist 121, 5947, 5947
- \l_tmpa_coffin 139, 6637, 6637
- \l_tmpa_dim 82, 4151, 4151
- \l_tmpa_fp 176, 14177, 14177
- \l_tmpa_int 73, 3900, 3900
- \l_tmpa_muskip 88, 4303, 4303
- \l_tmpa_prop 127, 5990, 5990
- \l_tmpa_seq 112, 5536, 5536
- \l_tmpa_skip 85, 4241, 4241
- \l_tmpa_tl 5, 103, 5052, 5052
- \l_tmpb_bool 38, 2119, 2120
- \l_tmpb_box 131, 6322, 6323
- \l_tmpb_clist 121, 5947, 5948
- \l_tmpb_coffin 139, 6637, 6638
- \l_tmpb_dim 82, 4151, 4152
- \l_tmpb_fp 176, 14177, 14178
- \l_tmpb_int 73, 3900, 3901
- \l_tmpb_muskip 88, 4303, 4304
- \l_tmpb_prop 127, 5990, 5991
- \l_tmpb_seq 112, 5536, 5537
- \l_tmpb_skip 85, 4241, 4242
- \l_tmpb_tl 103, 5052, 5053
- \language 404
- \lastbox 561
- \lastkern 494
- \lastlinefit 674
- \lastnodetype 655
- \lastpenalty 600
- \lastskip 495
- \latelua 716
- \lccode 623
- \leaders 491
- \left 459
- \lefthyphenmin 515
- \leftskip 517
- \legno 434
- \let 60, 70, 291, 292, 304
- \limits 451
- \linepenalty 507
- \lineskip 501
- \lineskiplimit 502
- \linewidth 6551, 6597
- \ln 12951, 12954
- \long 34, 294, 323
- \looseness 519
- \lower 556
- \lowercase 595
- \lua_now:n 188, 14299, 14317, 14319
- \lua_now:x 14299
- \lua_now_x:n 188, 4369, 14299, 14301, 14305, 14308, 14316, 14318
- \lua_now_x:x 14299
- \lua_shipout:n .. 188, 14299, 14321, 14323
- \lua_shipout:x 14299
- \lua_shipout_x:n 189, 14299, 14302, 14310, 14313, 14320, 14322
- \lua_shipout_x:x 14299
- \luaescapestring 40, 41
- \luatex_bodydir:D 719, 737
- \luatex_catcodetable:D 713, 733, 14362, 14363, 14368, 14376
- \luatex_directlua:D 714, 1485, 14301
- \luatex_if_engine:F 1463, 1488, 14341, 14378, 14406
- \luatex_if_engine:T . 1462, 1487, 4364, 14350, 14392, 14414, 14420, 14429
- \luatex_if_engine:TF 23, 1462, 1464, 1489, 14299
- \luatex_if_engine_p: 23, 1462, 1471, 1493
- \luatex_initcatcodetable:D 715, 734, 14337, 14356

<code>\luatex_latelua:D</code>	716, 735, 14302	<code>.meta:n</code>	154
<code>\luatex luatexversion:D</code>	717, 808	<code>.meta:nn</code>	154
<code>\luatex_mathdir:D</code>	720, 738	<code>\middle</code>	679
<code>\luatex_pagedir:D</code>	721, 739	<code>\mkern</code>	421
<code>\luatex_pardir:D</code>	722, 740	<code>\mode_if_horizontal:</code>	2298
<code>\luatex_savecatcodetable:D</code>	718, 736, 14367, 14403	<code>\mode_if_horizontal:TF</code>	41, 2298
<code>\luatex_textdir:D</code>	723, 741	<code>\mode_if_horizontal_p:</code>	41, 2298
<code>\luatexbodydir</code>	737	<code>\mode_if_inner:</code>	2300
<code>\luatexcatcodetable</code>	733	<code>\mode_if_inner:TF</code>	41, 2300
<code>\luatexinitcatcodetable</code>	734	<code>\mode_if_inner_p:</code>	41, 2300
<code>\luatexlatelua</code>	735	<code>\mode_if_math:</code>	2302
<code>\luatexmathdir</code>	738	<code>\mode_if_math:TF</code>	41, 2302
<code>\luatexpagedir</code>	739	<code>\mode_if_math_p:</code>	41, 2302
<code>\luatexpardir</code>	740	<code>\mode_if_vertical:</code>	2296
<code>\luatexsavecatcodetable</code>	736	<code>\mode_if_vertical:TF</code>	41, 2296
<code>\luatextextdir</code>	741	<code>\mode_if_vertical_p:</code>	41, 2296
<code>\luatexversion</code>	717	<code>\month</code>	607
M			
<code>\M</code>	1862, 2685, 10097	<code>\moveleft</code>	557
<code>cm</code>	184	<code>\moveright</code>	558
<code>em</code>	184	<code>\msg_critical:nn</code>	7537
<code>mm</code>	184	<code>\msg_critical:nnn</code>	7537
<code>\m@ne</code>	797	<code>\msg_critical:nnnn</code>	7537
<code>\mag</code>	403	<code>\msg_critical:nnnnn</code>	7537
<code>\mark</code>	405	<code>\msg_critical:nnnnnn</code>	144, 7537
<code>\marks</code>	631	<code>\msg_critical:nnx</code>	7537
<code>\mathaccent</code>	416	<code>\msg_critical:nnxx</code>	7537
<code>\mathbin</code>	446	<code>\msg_critical:nnxxx</code>	7537
<code>\mathchar</code>	417, 2751	<code>\msg_critical:nnxxxx</code>	7537
<code>\mathchardef</code>	314	<code>\msg_critical:nnxxxxx</code>	7537
<code>\mathchoice</code>	414	<code>\msg_critical_text:n</code>	142, 7477, 7478, 7540
<code>\mathclose</code>	447	<code>\msg_error:nn</code>	7548
<code>\mathcode</code>	625	<code>\msg_error:nnn</code>	7548
<code>\mathdir</code>	720	<code>\msg_error:nnnn</code>	7548
<code>\mathinner</code>	448	<code>\msg_error:nnnnn</code>	7548
<code>\mathop</code>	449	<code>\msg_error:nnnnnn</code>	144, 7548
<code>\mathopen</code>	453	<code>\msg_error:nnx</code>	7548
<code>\mathord</code>	454	<code>\msg_error:nnxx</code>	7548
<code>\mathpunct</code>	455	<code>\msg_error:nnxxx</code>	7548
<code>\mathrel</code>	456	<code>\msg_error:nnxxxx</code>	7548
<code>\mathsurround</code>	467	<code>\msg_error:nnxxxxx</code>	7548
<code>\maxdeadcycles</code>	537	<code>\msg_error_text:n</code>	142, 7477, 7479, 7555
<code>\maxdepth</code>	538	<code>\msg_fatal:nn</code>	7526
<code>\meaning</code>	597	<code>\msg_fatal:nnn</code>	7526
<code>\medmuskip</code>	468	<code>\msg_fatal:nnnn</code>	7526
<code>\message</code>	374	<code>\msg_fatal:nnnnn</code>	7526
<code>\MessageBreak</code>	77, 78, 79, 80, 81, 82, 83, 84, 148, 229	<code>\msg_fatal:nnnnnn</code>	143, 7526
		<code>\msg_fatal:nnx</code>	7526
		<code>\msg_fatal:nnxx</code>	7526
		<code>\msg_fatal:nnxxx</code>	7526
		<code>\msg_fatal:nnxxxx</code>	7526
		<code>\msg_fatal_text:n</code>	142, 7477, 7477, 7529
		<code>\msg_gset:nnn</code>	7340, 7363

<code>\msg_gset:nnnn</code>	142 , 7340 , 7343 , 7356 , 7364	<code>\msg_warning:nn</code>	7569
<code>\msg_if_exist:nn</code>	7314	<code>\msg_warning:nnn</code>	7569
<code>\msg_if_exist:nnT</code>	7321 , 7331	<code>\msg_warning:nnnn</code>	7569
<code>\msg_if_exist:nnTF</code>	142 , 7314 , 7605	<code>\msg_warning:nnnnn</code>	7569
<code>\msg_if_exist_p:nn</code>	142 , 7314	<code>\msg_warning:nnnnnn</code>	144 , 7569
<code>\msg_info:nn</code>	7577	<code>\msg_warning:nnx</code>	7569
<code>\msg_info:nnn</code>	7577	<code>\msg_warning:nnxx</code>	7569
<code>\msg_info:nnnn</code>	7577	<code>\msg_warning:nnxxx</code>	7569
<code>\msg_info:nnnnn</code>	7577	<code>\msg_warning:nnxxxx</code>	7569 , 7775
<code>\msg_info:nnnnnn</code>	144 , 7577	<code>\msg_warning_text:n</code>	143 , 7477 , 7480 , 7573
<code>\msg_info:nnx</code>	7577	<code>\mskip</code>	418
<code>\msg_info:nnxx</code>	7577	<code>\muexpr</code>	667
<code>\msg_info:nnxxx</code>	7577	<code>.multichoice:</code>	154
<code>\msg_info:nnxxxx</code>	7577 , 7776	<code>.multichoices:nn</code>	154
<code>\msg_info_text:n</code>	143 , 7477 , 7481 , 7581	<code>.multichoices:on</code>	154
<code>\msg_interrupt:nnn</code>	146 , 7404 , 7404 , 7528 , 7539 , 7554	<code>.multichoices:Vn</code>	154
<code>\msg_line_context:</code>	142 , 1175 , 1175 , 1194 , 7336 , 7397 , 7398	<code>.multichoices:xn</code>	154
<code>\msg_line_number:</code>	142 , 7397 , 7397 , 7402 , 8154	<code>\multiply</code>	319
<code>\msg_log:n</code>	147 , 7462 , 7462 , 7579	<code>\muskip</code>	615
<code>\msg_log:nn</code>	7585	<code>\muskip_add:cn</code>	4283
<code>\msg_log:nnn</code>	7585	<code>\muskip_add:Nn</code>	87 , 4283 , 4283 , 4285 , 4286
<code>\msg_log:nnnn</code>	7585	<code>\muskip_const:cn</code>	4253
<code>\msg_log:nnnnn</code>	144 , 7585	<code>\muskip_const:Nn</code>	86 , 4253 , 4253 , 4258 , 4301 , 4302
<code>\msg_log:nnx</code>	7585	<code>\muskip_eval:n</code>	87 , 4293 , 4293
<code>\msg_log:nnxx</code>	7585	<code>\muskip_gadd:cn</code>	4283
<code>\msg_log:nnxxx</code>	7585	<code>\muskip_gadd:Nn</code>	87 , 4283 , 4285 , 4287
<code>\msg_log:nnxxxx</code>	7585	<code>\muskip_gset:cn</code>	4272
<code>\msg_log:nnxxxxx</code>	7585	<code>\muskip_gset:Nn</code>	87 , 4256 , 4272 , 4274 , 4276
<code>\msg_new:nnn</code>	7340 , 7345 , 7727	<code>\muskip_gset_eq:cc</code>	4277
<code>\msg_new:nnnn</code>	141 , 7340 , 7340 , 7346 , 7725	<code>\muskip_gset_eq:cN</code>	4277
<code>\msg_none:nn</code>	7591	<code>\muskip_gset_eq:Nc</code>	4277
<code>\msg_none:nnn</code>	7591	<code>\muskip_gset_eq:NN</code>	87 , 4277 , 4280 , 4281 , 4282
<code>\msg_none:nnnn</code>	7591	<code>\muskip_gsub:cn</code>	4283
<code>\msg_none:nnnnn</code>	7591	<code>\muskip_gsub:Nn</code>	87 , 4283 , 4290 , 4292
<code>\msg_none:nnnnnn</code>	145 , 7591	<code>\muskip_gzero:c</code>	4259
<code>\msg_none:nnx</code>	7591	<code>\muskip_gzero:N</code>	86 , 4259 , 4261 , 4263 , 4267
<code>\msg_none:nnxx</code>	7591	<code>\muskip_gzero_new:c</code>	4264
<code>\msg_none:nnxxx</code>	7591	<code>\muskip_gzero_new:N</code>	86 , 4264 , 4266 , 4269
<code>\msg_none:nnxxxx</code>	7591	<code>\muskip_if_exist:c</code>	4271
<code>\msg_none:nnxxxxx</code>	7591	<code>\muskip_if_exist:cTF</code>	4270
<code>\msg_redirect_class:nn</code>	145 , 7677 , 7677	<code>\muskip_if_exist:N</code>	4270
<code>\msg_redirect_module:nnn</code>	146 , 7677 , 7679	<code>\muskip_if_exist:NTF</code>	86 , 4265 , 4267 , 4270
<code>\msg_redirect_name:nnn</code>	146 , 7668 , 7668	<code>\muskip_if_exist_p:c</code>	4270
<code>\msg_see_documentation_text:n</code>	143 , 7482 , 7482 , 7532 , 7543 , 7558	<code>\muskip_if_exist_p:N</code>	86 , 4270
<code>\msg_set:nnn</code>	7340 , 7354 , 7731	<code>\muskip_new:c</code>	4245
<code>\msg_set:nnnn</code>	142 , 7340 , 7347 , 7355 , 7729	<code>\muskip_new:N</code>	86 , 4245 , 4246 , 4252 , 4255 , 4265 , 4267 , 4303 , 4304 , 4305 , 4306
<code>\msg_term:n</code>	147 , 7462 , 7468 , 7571	<code>\muskip_set:cn</code>	4272

\muskip_set:Nn . 87, 4272 , 4272 , 4274 , 4275	O
\muskip_set_eq:cc 4277	\O 15493
\muskip_set_eq:cN 4277	\omit 338
\muskip_set_eq:Nc 4277	\openin 364
\muskip_set_eq:NN 87, 4277 , 4277 , 4278 , 4279	\openout 365
\muskip_show:c 4297	\or 361
\muskip_show:N 88, 4297 , 4297 , 4298	\or: 24, 74, 750 , 752, 1286, 1287,
\muskip_show:n 88, 4299 , 4299	1288, 1289, 1290, 1291, 1292, 1293,
\muskip_sub:cn 4283	1294, 3656, 3657, 3658, 3659, 3660,
\muskip_sub:Nn . 87, 4283 , 4288 , 4290 , 4291	3661, 3662, 3663, 3664, 3665, 3666,
\muskip_use:c 4295	3667, 3668, 3669, 3670, 3671, 3672,
\muskip_use:N . . 87, 4294 , 4295 , 4295 , 4296	3673, 3674, 3675, 3676, 3677, 3678,
\muskip_zero:c 4259	3679, 3680, 9442, 9443, 9444, 9594,
\muskip_zero:N	10743, 11064, 11065, 11066, 11103,
. 86, 4259 , 4259 , 4261 , 4262 , 4265	11104, 11429, 11430, 11431, 11559,
\muskip_zero_new:c 4264	11643, 11729, 11730, 11731, 11732,
\muskip_zero_new:N . . 86, 4264 , 4264 , 4268	11733, 11734, 11735, 11736, 11737,
\muskipdef 313	11814, 11817, 11966, 12214, 12430,
\mutogluue 673	12455, 12461, 12462, 12463, 12464,
\my_map_dbl:nn 1	12465, 12612, 12647, 12649, 12657,
	12747, 12751, 12752, 12753, 12754,
	12755, 12756, 12757, 12758, 12759,
N	12760, 12767, 12768, 12769, 12770,
\N 2011	12771, 12772, 12773, 12774, 12775,
in 184	12776, 12783, 12784, 12785, 12786,
ln 181	12787, 12788, 12789, 12790, 12791,
\newbox 6237	12792, 12799, 12800, 12801, 12802,
\newcatcodetable 14355	12803, 12804, 12805, 12806, 12807,
\newcount 3235	12808, 12815, 12816, 12817, 12818,
\newdimen 3920	12819, 12820, 12821, 12822, 12823,
\newlinechar 90, 369	12824, 12831, 12832, 12833, 12834,
\newmuskip 4249	12835, 12836, 12837, 12838, 12839,
\newread 9034	12840, 12858, 12904, 12907, 12916,
\newskip 4159	13027, 13050, 13097, 13102, 13112,
\newwrite 9127	13117, 13127, 13132, 13142, 13147,
inf 184	13157, 13162, 13172, 13177, 13614,
\noalign 337	13634, 13635, 13679, 13764, 13767,
\noboundary 472	13779, 13785, 13832, 13834, 13835,
\noexpand 36, 40, 41, 173, 176, 179, 181,	13845, 13851, 13899, 13900, 13911,
182, 191, 194, 195, 196, 198, 200,	13949, 13950, 13960, 14017, 14018
201, 210, 212, 213, 214, 215, 216, 330	cos 182
\noindent 498	cosd 182
\nolimits 452	cot 182
\nonscript 432	cotd 182
\nonstopmode 395	round 182
\nulldelimiterspace 465	round+ 182
\nullfont 583	round- 182
\number 592	round0 182
\numexpr 664	\outer 324
	\output 539

<code>\outputpenalty</code>	549	<code>\pdfstrcmp</code>	34, 60, 70, 77, 92, 711
<code>\over</code>	426	<code>\pdfTeX_if_engine:F</code>	1466, 1477, 1491
<code>\overfullrule</code>	577	<code>\pdfTeX_if_engine:T</code>	1465, 1476, 1490
<code>\overline</code>	457	<code>\pdfTeX_if_engine:TF</code>	
<code>\overwithdelims</code>	427	23, 1462, 1467, 1478, 1492, 3260
P			
<code>\P</code>	1864, 10011, 10095	<code>\pdfTeX_if_engine_p:</code>	
<code>bp</code>	184	23, 1462, 1472, 1482, 1494
<code>sp</code>	184	<code>\pdfTeX_pdfcolorstack:D</code>	693
<code>\PackageError</code>	75, 145, 226	<code>\pdfTeX_pdfcompresslevel:D</code>	694
<code>\pagedepth</code>	541	<code>\pdfTeX_pdfcreationdate:D</code>	692
<code>\pagedir</code>	721	<code>\pdfTeX_pdfdecimaldigits:D</code>	695
<code>\pagediscards</code>	682	<code>\pdfTeX_pdfdhorigin:D</code>	696
<code>\pagefilllstretch</code>	545	<code>\pdfTeX_pdfinfo:D</code>	697
<code>\pagefillstretch</code>	544	<code>\pdfTeX_pdflastxform:D</code>	698
<code>\pagefilstretch</code>	543	<code>\pdfTeX_pdfliteral:D</code>	699
<code>\pagegoal</code>	547	<code>\pdfTeX_pdfminorversion:D</code>	700
<code>\pageshrink</code>	546	<code>\pdfTeX_pdfobjcompresslevel:D</code>	701
<code>\pagestretch</code>	542	<code>\pdfTeX_pdfoutput:D</code>	702
<code>\pagetotal</code>	548	<code>\pdfTeX_pdfpkresolution:D</code>	707
<code>\par</code>	497, 6397, 6398, 6400, 6402, 6404, 6409, 6415, 6428	<code>\pdfTeX_pdfrefxform:D</code>	703
<code>\pardir</code>	722	<code>\pdfTeX_pdfrestore:D</code>	704
<code>\parfillskip</code>	528	<code>\pdfTeX_pdfsave:D</code>	705
<code>\parindent</code>	521	<code>\pdfTeX_pdfsetmatrix:D</code>	706
<code>\parshape</code>	513	<code>\pdfTeX_pdftextrevision:D</code>	708
<code>\parshapedimen</code>	663	<code>\pdfTeX_pdfvorigin:D</code>	709
<code>\parshapeindent</code>	661	<code>\pdfTeX_pdfxform:D</code>	710
<code>\parshapelength</code>	662	<code>\pdfTeX_strcmp:D</code> ...	711, 1499, 1505, 1510, 2374, 2381, 2398, 2425, 2434, 2700, 4205, 10536, 10549, 10792, 12896
<code>\parskip</code>	520	<code>\pdfTeXrevision</code>	708
<code>\patterns</code>	603	<code>\pdfvorigin</code>	709
<code>\pausing</code>	390	<code>\pdfxform</code>	710
<code>\pdf@strcmp</code>	60	<code>\peek_after:Nw</code>	
<code>\pdfcolorstack</code>	693	58, 2951, 2951, 2976, 2994, 3044
<code>\pdfcompresslevel</code>	694	<code>\peek_catcode:Ntf</code>	58, 3068
<code>\pdfcreationdate</code>	692	<code>\peek_catcode_ignore_spaces:Ntf</code> 58, 3068	
<code>\pdfdecimaldigits</code>	695	<code>\peek_catcode_remove:Ntf</code>	59, 3068
<code>\pdfdhorigin</code>	696	<code>\peek_catcode_remove_ignore_spaces:Ntf</code>	
<code>\pdfinfo</code>	697	59, 3068
<code>\pdflastxform</code>	698	<code>\peek_charcode:Ntf</code>	59, 3084
<code>\pdfliteral</code>	699	<code>\peek_charcode_ignore_spaces:Ntf</code> ...	
<code>\pdfminorversion</code>	700	59, 3084
<code>\pdfobjcompresslevel</code>	701	<code>\peek_charcode_remove:Ntf</code>	59, 3084
<code>\pdfoutput</code>	702	<code>\peek_charcode_remove_ignore_spaces:Ntf</code>	
<code>\pdfpkresolution</code>	707	60, 3084
<code>\pdfrefxform</code>	703	<code>\peek_gafter:Nw</code>	58, 2951, 2953
<code>\pdfrestore</code>	704	<code>\peek_meaning:Ntf</code>	60, 3100
<code>\pdfsave</code>	705	<code>\peek_meaning_ignore_spaces:Ntf</code> 60, 3100	
<code>\pdfsetmatrix</code>	706	<code>\peek_meaning_remove:Ntf</code>	60, 3100

`\peek_meaning_remove_ignore_spaces:NTF` 5372, 5374, 5682, 5694, 5696, 5779, 5783, 6052, 6062, 6169, 8907, 9009
 60, 3100
`\peek_N_type:F` 15532
`\peek_N_type:T` 15530
`\peek_N_type:TF` 201, 15492, 15528
`\penalty` 598
`\postdisplaypenalty` 445
`\predisplaydirection` 689
`\predisplaypenalty` 444
`\predisplaysize` 443
`\pretolerance` 524
`\prevdepth` 571
`\prevgraf` 530
`\prg_break:` 2466, 2468
`\prg_do_nothing:`
 ... 9, 991, 992, 1455, 1496, 1496,
 1584, 1931, 1936, 4464, 4478, 4538,
 4543, 5112, 5119, 5359, 5361, 5673,
 9636, 9688, 9722, 9748, 9756, 11073,
 14087, 14211, 14849, 14853, 14860
`\prg_new_conditional:Nnn` 894,
 896, 2062, 2403, 2411, 2423, 2432, 9070
`\prg_new_conditional:Npnn`
 35, 881, 883, 1415,
 1497, 1503, 2062, 2091, 2125, 2296,
 2298, 2300, 2302, 2611, 2616, 2621,
 2626, 2633, 2639, 2644, 2649, 2654,
 2659, 2664, 2669, 2674, 2679, 2693,
 2707, 2712, 2735, 2744, 2754, 2772,
 2796, 2814, 2832, 2850, 2862, 2871,
 2891, 3333, 3381, 3418, 3426, 3998,
 4003, 4202, 4214, 4559, 4569, 4581,
 4602, 4604, 4650, 4932, 4951, 4970,
 5005, 5011, 5026, 5231, 6130, 6139,
 6297, 6299, 6309, 6483, 7314, 8647,
 8694, 8700, 9661, 10531, 11132,
 11145, 14185, 14192, 14901, 15354
`\prg_new_eq_conditional:NNn`
 37, 982, 984, 2062,
 2123, 2124, 3285, 3286, 3940, 3941,
 4179, 4180, 4270, 4271, 4360, 4361,
 5143, 5144, 5524, 5525, 5526, 5527,
 5528, 5529, 5584, 5585, 5777, 5778,
 6128, 6129, 6265, 6266, 11130, 11131
`\prg_new_protected_conditional:Nnn` .
 894, 900, 2062
`\prg_new_protected_conditional:Npnn`
 35, 881, 887, 2062, 4616,
 4637, 5243, 5358, 5360, 5368, 5370,
 5372, 5374, 5682, 5694, 5696, 5779,
 5783, 6052, 6062, 6169, 8907, 9009
`\prg_replicate:nn` 41, 2252, 2252, 7910,
 9323, 12156, 13003, 13062, 13069,
 13222, 13426, 13427, 13429, 13431,
 13444, 13481, 13956, 13983, 13991
`\prg_return_false:` 37,
 877, 879, 1086, 1091, 1104, 1109,
 1117, 1134, 1418, 1501, 1506, 1513,
 2062, 2096, 2130, 2297, 2299, 2301,
 2303, 2408, 2416, 2429, 2438, 2614,
 2619, 2624, 2629, 2636, 2642, 2647,
 2652, 2657, 2662, 2667, 2672, 2677,
 2682, 2703, 2710, 2715, 2720, 2757,
 2760, 2779, 2782, 2799, 2802, 2817,
 2820, 2835, 2838, 2894, 2913, 2930,
 2939, 3331, 3358, 3363, 3386, 3423,
 3429, 4001, 4022, 4037, 4038, 4209,
 4217, 4574, 4586, 4599, 4609, 4626,
 4641, 4944, 4967, 4983, 4991, 5001,
 5021, 5035, 5236, 5257, 5280, 5685,
 5701, 5793, 6060, 6070, 6133, 6155,
 6176, 6298, 6300, 6310, 6489, 6491,
 7317, 8653, 8655, 8698, 8704, 8911,
 9018, 9080, 9666, 10545, 10557,
 11140, 11153, 14190, 14195, 14904
`\prg_return_true:` 37, 877, 877,
 1089, 1106, 1114, 1119, 1132, 1137,
 1418, 1501, 1506, 1511, 2062, 2094,
 2128, 2297, 2299, 2301, 2303, 2406,
 2414, 2427, 2436, 2614, 2619, 2624,
 2629, 2636, 2642, 2647, 2652, 2657,
 2662, 2667, 2672, 2677, 2682, 2701,
 2710, 2718, 2774, 2776, 2911, 2937,
 3358, 3384, 3421, 3431, 4001, 4038,
 4207, 4218, 4572, 4584, 4597, 4607,
 4623, 4641, 4942, 4965, 4981, 4999,
 5023, 5034, 5234, 5261, 5283, 5688,
 5704, 5793, 6058, 6068, 6133, 6157,
 6174, 6298, 6300, 6310, 6488, 7317,
 8652, 8697, 8703, 8912, 9021, 9075,
 9078, 9084, 9664, 10540, 10563,
 11142, 11151, 14190, 14195, 14905
`\prg_set_conditional:Nnn` . 894, 894, 2062
`\prg_set_conditional:Npnn` 35,
 881, 881, 1083, 1095, 1111, 1123, 2062
`\prg_set_eq_conditional:NNn`
 37, 982, 982, 2062
`\prg_set_protected_conditional:Nnn` .
 894, 898, 2062

\prg_set_protected_conditional:Npnn	
.....	35 , 881 , 885 , 2062
\prop_clear:c	5970 , 6869
\prop_clear:N	122 , 5970 , 5970 , 5972 , 5977
\prop_clear_new:c	5976 , 6518 , 6519 , 8307
\prop_clear_new:N	122 , 5976 , 5976 , 5978
\prop_gclear:c	5970
\prop_gclear:N	122 , 5970 , 5973 , 5975 , 5980
\prop_gclear_new:c	5976
\prop_gclear_new:N	122 , 5976 , 5979 , 5981
\prop_get:cn	15199
\prop_get:cnN	6022
\prop_get:cnNF	6647 , 8641
\prop_get:cnNT	7701
\prop_get:cnNTF	6169 , 7646 , 8596
\prop_get:coN	6022
\prop_get:coNTF	6169
\prop_get:cVN	6022
\prop_get:cVNTF	6169
\prop_get:Nn	197 , 15199 , 15199 , 15211
\prop_get:NnN	123 , 6022 , 6022 , 6028 , 6029 , 6169 , 7092 , 7096 , 7175 , 7179
\prop_get:NnNF	6179 , 6182
\prop_get:NnNT	6178 , 6181
\prop_get:NnNTF	125 , 6169 , 6180 , 6183 , 7626
\prop_get:NoN	6022
\prop_get:NoNTF	6169
\prop_get:NVN	6022
\prop_get:NVNTF	6169
\prop_gpop:cnN	6030
\prop_gpop:cnNTF	6052
\prop_gpop:coN	6030
\prop_gpop:NnN	123 , 6030 , 6039 , 6050 , 6051 , 6062
\prop_gpop:NnNF	6076
\prop_gpop:NnNT	6075
\prop_gpop:NnNTF	125 , 6052 , 6077
\prop_gpop:NoN	6030
\prop_gput:cnn	6078
\prop_gput:cno	6078
\prop_gput:cnV	6078
\prop_gput:cnx	6078
\prop_gput:con	6078
\prop_gput:coo	6078
\prop_gput:cVn	6078
\prop_gput:cVV	6078
\prop_gput:Nnn	123 , 6078 , 6079 , 6101 , 6103 , 8995
\prop_gput:Nno	6078
\prop_gput:NnV	6078
\prop_gput:Nnx	6078
\prop_gput:Non	6078
\prop_gput:Noo	6078
\prop_gput:NVn	6078 , 9044 , 9137
\prop_gput:NVV	6078
\prop_gput_if_new:cnn	6105
\prop_gput_if_new:Nnn	123 , 6105 , 6107 , 6127
\prop_gremove:cn	6006
\prop_gremove:cV	6006
\prop_gremove:Nn	124 , 6006 , 6012 , 6020 , 6021
\prop_gremove:NV	6006 , 9052 , 9145
\prop_gset_eq:cc	5982 , 5989 , 6671 , 6673
\prop_gset_eq:cN	5982 , 5988 , 6520 , 6522
\prop_gset_eq:Nc	5982 , 5987
\prop_gset_eq:NN	122 , 5974 , 5982 , 5986
\prop_if_empty:cTF	6130
\prop_if_empty:N	6130
\prop_if_empty:NF	6137
\prop_if_empty:NT	6136
\prop_if_empty:NTF	124 , 6130 , 6138 , 7926 , 9064
\prop_if_empty_p:c	6130
\prop_if_empty_p:N	124 , 6130 , 6135
\prop_if_exist:c	6129
\prop_if_exist:cT	8313 , 8318 , 8366
\prop_if_exist:cTF	6128 , 8594 , 8649
\prop_if_exist:N	6128
\prop_if_exist:NTF	124 , 5977 , 5980 , 6128
\prop_if_exist_p:c	6128
\prop_if_exist_p:N	124 , 6128
\prop_if_in:cnTF	6139 , 8651
\prop_if_in:coTF	6139
\prop_if_in:cVTF	6139
\prop_if_in:Nn	6139
\prop_if_in:NnF	6165 , 6166
\prop_if_in:NnT	6163 , 6164
\prop_if_in:NnTF	124 , 6139 , 6167 , 6168
\prop_if_in:NoTF	6139
\prop_if_in:NVTF	6139
\prop_if_in_p:cn	6139
\prop_if_in_p:co	6139
\prop_if_in_p:cV	6139
\prop_if_in_p:Nn	124 , 6139 , 6161 , 6162
\prop_if_in_p:No	6139
\prop_if_in_p:NV	6139
\prop_map_break:	126 , 6188 , 6193 , 6207 , 6215 , 6215 , 6216 , 6218 , 15188 , 15193
\prop_map_break:n	126 , 6215 , 6217
\prop_map_function:cc	6184
\prop_map_function:cN	6184 , 7238

<code>\prop_map_function:Nc</code>	6184	<code>\prop_put:NvN</code>	6078
<code>\prop_map_function:NN</code>	125 , 6184 , 6184 , 6198 , 6199 , 6222 , 9067	<code>\prop_put:NvV</code>	6078
<code>\prop_map_inline:cn</code>	6200 , 6940 , 6959 , 14928 , 14930 , 14953 , 14955 , 15020 , 15080 , 15082 , 15086 , 15088	<code>\prop_put_if_new:cnn</code>	6105
<code>\prop_map_inline:Nn</code>	125 , 6200 , 6200 , 6214 , 7144 , 7153 , 14933 , 15037	<code>\prop_put_if_new:Nnn</code> 123 , 6105 , 6105 , 6126	
<code>\prop_map_tokens:cn</code>	15184	<code>\prop_remove:cn</code> ..	6006 , 7686 , 8368 , 8369
<code>\prop_map_tokens:Nn</code>	196 , 15184 , 15184 , 15198	<code>\prop_remove:cV</code>	6006
<code>\prop_new:c</code>	5964 , 7491	<code>\prop_remove:Nn</code>	124 , 6006 , 6006 , 6018 , 6019 , 7139 , 7142 , 7146 , 7671
<code>\prop_new:N</code>	122 , 5964 , 5964 , 5969 , 5977 , 5980 , 5990 , 5991 , 5992 , 5993 , 6455 , 6460 , 7018 , 7059 , 7600 , 8993 , 9106 , 14918	<code>\prop_remove:Nv</code>	6006
<code>\prop_pop:cnN</code>	6030	<code>\prop_set_eq:cc</code> 5982 , 5985 , 6664 , 6666 , 6899	
<code>\prop_pop:cnNTF</code>	6052	<code>\prop_set_eq:cN</code> ..	5982 , 5984 , 6657 , 6659
<code>\prop_pop:coN</code>	6030	<code>\prop_set_eq:Nc</code>	5982 , 5983 , 7134
<code>\prop_pop:NnN</code>	123 , 6030 , 6030 , 6048 , 6049 , 6052	<code>\prop_set_eq:NN</code>	122 , 5971 , 5982 , 5982
<code>\prop_pop:NnNF</code>	6073	<code>\prop_show:c</code>	6219
<code>\prop_pop:NnNT</code>	6072	<code>\prop_show:N</code>	126 , 6219 , 6219 , 6224
<code>\prop_pop:NnNTF</code>	125 , 6052 , 6074	<code>\protected</code>	110 , 124 , 140 , 152 , 159 , 164 , 220 , 247 , 280 , 691 , 2879
<code>\prop_pop:NoN</code>	6030	<code>\protected@edef</code>	9229
<code>\prop_put:cnN</code>	6078 , 6699 , 7690 , 7707 , 8314 , 8370	<code>\ProvidesClass</code>	161
<code>\prop_put:cnO</code>	6078	<code>\ProvidesExplClass</code>	6 , 139 , 159
<code>\prop_put:cnV</code>	6078 , 8321	<code>\ProvidesExplFile</code>	6 , 139 , 164
<code>\prop_put:cnx</code>	6078 , 6705 , 6707 , 6709 , 6711 , 6716 , 6721 , 6726 , 6733 , 6740 , 6964 , 14980 , 15047 , 15055 , 15117 , 15131 , 15138	<code>\ProvidesExplPackage</code>	6 , 139 , 140 , 152 , 157 , 288 , 746 , 1593 , 2056 , 2347 , 2473 , 3152 , 3909 , 4312 , 5060 , 5544 , 5955 , 6229 , 6448 , 7274 , 7307 , 8046 , 8818 , 9377 , 14295 , 14470
<code>\prop_put:con</code>	6078	<code>\ProvidesFile</code>	166
<code>\prop_put:coo</code>	6078	<code>\ProvidesPackage</code>	48 , 154
<code>\prop_put:cVn</code>	6078		
<code>\prop_put:cVv</code>	6078		
<code>\prop_put:Nnn</code>	123 , 6078 , 6078 , 6097 , 6099 , 6456 , 6457 , 6458 , 6459 , 7019 , 7021 , 7023 , 7025 , 7027 , 7029 , 7031 , 7033 , 7035 , 7037 , 7039 , 7041 , 7043 , 7045 , 7047 , 7049 , 7051 , 7053 , 7674 , 9108 , 9109 , 9110		
<code>\prop_put:Nno</code> .	6078 , 6462 , 6463 , 6464 , 6466 , 6467 , 6468 , 6469 , 6470 , 6471		
<code>\prop_put:NnV</code>	6078		
<code>\prop_put:Nnx</code>	6078 , 14961 , 14963 , 14966 , 14968 , 14974		
<code>\prop_put:Non</code>	6078		
<code>\prop_put:Noo</code>	6078		

Q

<code>\q__tl_act_mark</code>	4815 , 4818 , 4835
<code>\q__tl_act_stop</code> 4815 , 4818 , 4822 , 4831 , 4833 , 4839 , 4844 , 4847 , 4851 , 4854	
<code>\q_tl_act_mark</code>	2445 , 2445
<code>\q_tl_act_stop</code>	2445 , 2446
<code>\q_mark</code>	45 , 1070 , 1071 , 1074 , 1075 , 1076 , 1537 , 1565 , 1572 , 1875 , 1876 , 1878 , 1882 , 1885 , 1910 , 1919 , 1933 , 1941 , 1944 , 1953 , 1968 , 1990 , 2018 , 2021 , 2029 , 2137 , 2138 , 2139 , 2140 , 2143 , 2144 , 2145 , 2146 , 2147 , 2148 , 2149 , 2352 , 2353 , 3346 , 3349 , 3410 , 4060 , 4521 , 4530 , 4534 , 4545 , 4673 , 4756 , 4757 , 4760 , 4763 , 4764 , 4770 , 4784 , 4785 , 4791 , 4795 , 4797 , 4800 , 5473 , 5474 , 5488 , 5491 , 5596 , 5605 , 5610 , 5665 , 5675 , 5679 , 5703 , 5755 , 5761 , 5774 , 5823 , 5831 , 5919 , 5920 , 5929 , 5930 , 6000 , 6002 , 6003 , 7631 , 7632 , 7637 ,

7640, 10019, 10021, 14866, 14904, 14905, 14914, 15509, 15510, 15517	\quark_if_nil:n 2423
\q_nil 862,	\quark_if_nil:nF 2444
865, 2136, 2138, 2139, 2143, 2144,	\quark_if_nil:nT 2443
2145, 2146, 2147, 2148, 2352, 2352,	\quark_if_nil:NTF <u>45</u> , <u>2403</u> , 3772, 3794, 8129
2405, 2426, 3769, 3791, 4583, 4595,	\quark_if_nil:nTF <u>45</u> , <u>2423</u> , <u>2442</u>
4596, 4783, 4787, 4805, 4808, 4811,	\quark_if_nil:oTF <u>2423</u> , 8110
4896, 4897, 8073, 8081, 8085, 8102,	\quark_if_nil:VTF <u>2423</u>
8110, 8111, 8114, 8125, 8132, 8137	\quark_if_nil_p:N <u>45</u> , <u>2403</u>
\q_no_value 45, 2193, 2352, 2354, 2413, 2435,	\quark_if_nil_p:n <u>45</u> , <u>2423</u> , <u>2441</u>
5271, 5279, 5291, 5316, 5648, 5663,	\quark_if_nil_p:o <u>2423</u>
6026, 6037, 6046, 8101, 8107, 8883	\quark_if_nil_p:V <u>2423</u>
\q_recursion_stop 4, <u>46</u> , 864, 867, 931, 995, 1859,	\quark_if_no_value:cTF <u>2403</u>
<u>2356</u> , <u>2357</u> , 5597, 5860, 5896, 8073	\quark_if_no_value:N 2411
\q_recursion_tail 4, <u>46</u> , 931, 936, 995, 1014, <u>2356</u> ,	\quark_if_no_value:n 2432
2356, 2360, 2366, 2375, 2382, 2392,	\quark_if_no_value:N 2421
2399, 4688, 4705, 4714, 5597, 5809,	\quark_if_no_value:NT 2420
5823, 5842, 5860, 5896, 6143, 6154,	\quark_if_no_value:NTF 45, 2198, <u>2403</u> , <u>2422</u> , 7094, 7098,
6187, 6192, 8073, 15187, 15192, 15459	7177, 7181, 8910, 8917, 9005, 9017
\q_stop <u>45</u> , 863, 866,	\quark_if_no_value:nTF <u>45</u> , <u>2423</u>
947, 951, 967, 972, 977, 1025, 1029,	\quark_if_no_value_p:c <u>2403</u>
1034, 1039, 1044, 1072, 1074, 1075,	\quark_if_no_value_p:N .. <u>45</u> , <u>2403</u> , <u>2419</u>
1076, 1537, 1565, 1572, 1879, 1885,	\quark_if_no_value_p:n <u>45</u> , <u>2423</u>
1914, 1941, 1945, 1949, 1957, 1963,	\quark_if_recursion_tail_break:N ...
1972, 1990, 2024, 2029, 2141, 2149, <u>2465</u> , <u>2465</u>
2193, 2196, 2320, 2322, 2334, 2336,	\quark_if_recursion_tail_break:n ...
2340, <u>2352</u> , 2355, 2696, 2698, 2740, <u>2465</u> , <u>2467</u>
2749, 2753, 2765, 2771, 2787, 2795,	\quark_if_recursion_tail_stop:N
2807, 2813, 2825, 2831, 2843, 2849, <u>46</u> , <u>2358</u> , <u>2358</u> , 5872
2856, 2861, 2867, 2877, 2881, 2897,	\quark_if_recursion_tail_stop:n .. 8,
2900, 2903, 2925, 3119, 3126, 3135,	<u>9</u> , <u>46</u> , <u>2372</u> , <u>2372</u> , 2388, 5601, 5902
3144, 3325, 3341, 3343, 3347, 3355,	\quark_if_recursion_tail_stop:o
3410, 3837, 3874, 4011, 4037, 4060, <u>2372</u> , 8083
4131, 4136, 4218, 4220, 4530, 4545,	\quark_if_recursion_tail_stop_do:Nn
4673, 4758, 4760, 4765, 4767, 4789, <u>46</u> , <u>2358</u> , 2364
4811, 4890, 4892, 4906, 4921, 4929,	\quark_if_recursion_tail_stop_do:nn
4931, 4939, 4958, 4980, 5291, 5294, <u>46</u> , <u>2372</u> , <u>2379</u> , 2389
5302, 5304, 5475, 5488, 5491, 5493,	\quark_if_recursion_tail_stop_do:on
5650, 5653, 5665, 5668, 5676, 5679, <u>2372</u>
5687, 5703, 5761, 5921, 5929, 5930,	\quark_new:N <u>45</u> , <u>2351</u> , 2351, 2352, 2353,
5931, 6000, 6003, 7633, 8084, 8088,	2354, 2355, 2356, 2357, 2445, 2446
8101, 8106, 8112, 8114, 8134, 8137,	
8214, 8220, 8226, 8235, 8245, 8326,	
8327, 9248, 9337, 10019, 10021,	
10110, 10113, 14831, 14864, 14906,	
14914, 15249, 15251, 15256, 15511	
\quark_if_nil:N 2403	

R

\R 1865, 15497
\radical 419
\raise 559
\read 366
\readline 641
\relax 4, 5, 6, 7,
10, 14, 63, 69, 73, 90, 112, 113, 114,

- 115, 116, 117, 118, 119, 120, 121,
 122, 126, 127, 128, 129, 130, 131,
 132, 133, 134, 135, 138, 236, 237,
 238, 239, 240, 241, 242, 243, 244, 401
 \relpenalty 462
 \RequirePackage 58, 59
 \reverse_if:N 24, 750, 755,
 3368, 3370, 3372, 3374, 4025, 4030,
 4034, 4036, 4928, 12908, 13561, 13584
 \right 460
 \righthyphenmin 516
 \rightskip 518
 \romannumeral 593
 true 185
 \rule 7076, 7131
- S**
- \S 10009
 \s__fp ... 9393, 9393, 9397, 9406, 9407,
 9408, 9409, 9410, 9412, 9413, 9416,
 9422, 9426, 9446, 9449, 9450, 9460,
 9470, 9482, 9502, 9578, 9580, 9582,
 9583, 9584, 9586, 9587, 9588, 9590,
 9766, 9771, 9930, 9939, 9941, 10536,
 10646, 11137, 11162, 11163, 11273,
 11274, 11277, 11288, 11289, 11297,
 11298, 11300, 11301, 11302, 11304,
 11305, 11306, 11317, 11320, 11332,
 11357, 11410, 11413, 11416, 11435,
 11436, 11438, 11439, 11440, 11448,
 11451, 11467, 11468, 11470, 11479,
 11555, 11706, 11740, 11741, 11744,
 11823, 11959, 11962, 12422, 12434,
 12436, 12643, 12660, 12662, 12851,
 12870, 12872, 12873, 12875, 12887,
 12892, 12894, 12919, 12920, 12922,
 12938, 13023, 13036, 13038, 13041,
 13046, 13093, 13106, 13108, 13121,
 13123, 13136, 13138, 13151, 13153,
 13166, 13168, 13181, 13191, 13623,
 13638, 13639, 13643, 13654, 13760,
 13773, 13775, 13791, 13794, 13804,
 13827, 13838, 13840, 13854, 13856,
 13861, 13894, 13919, 13922, 13944,
 13968, 13971, 14012, 14035, 14085,
 14086, 14187, 14194, 14247, 14256
 \s__fp_division 9401, 9404
 \s__fp_exact 9401, 9405,
 9406, 9407, 9408, 9409, 9410, 11273
 \s__fp_invalid 9401, 9401
 \s__fp_mark 9399, 9399, 9637,
 9638, 9640, 9644, 10757, 10761, 10792
 \s__fp_overflow 9401, 9403, 9413
 \s__fp_stop .. 9399, 9400, 10618, 10758,
 10761, 11336, 11347, 11370, 11378
 \s__fp_underflow 9401, 9402, 9412
 \s__prop
 127, 5959, 5959, 5960, 5963, 6000,
 6003, 6085, 6114, 6142, 6146, 6187,
 6190, 6205, 15187, 15190, 15202, 15205
 \s__seq ... 112, 2463, 2463, 5064, 5071,
 5123, 5149, 5157, 5161, 5313, 5339,
 5349, 5394, 5484, 15214, 15239,
 15245, 15263, 15268, 15273, 15278
 \s__stop 48, 2461,
 2461, 2462, 12721, 12736, 13876, 13880
 \s_obj_end
 2464, 2464, 5071, 5123, 5140, 5171,
 5179, 5182, 5313, 5343, 5349, 5394,
 5467, 5482, 5484, 5963, 5994, 6093,
 6122, 15214, 15239, 15245, 15263,
 15268, 15273, 15278, 15298, 15299
 \savecatcodetable 718
 \savingshyphcodes 680
 \savingsvdiscards 681
 csc 182
 \scan_align_safe_stop: .. 42, 2308, 2308
 \scan_stop: 9, 263, 277, 774,
 774, 953, 1061, 1085, 1103, 1113,
 1131, 1444, 1445, 1634, 1859, 1862,
 1863, 1864, 1865, 1901, 1943, 2010,
 2011, 2311, 2326, 2458, 2632, 2709,
 3014, 3128, 3137, 3146, 3536, 4182,
 4193, 4198, 4224, 4229, 4232, 4273,
 4284, 4289, 4294, 4450, 4451, 4452,
 4453, 4929, 5337, 6343, 6361, 7072,
 7127, 9043, 9045, 9136, 9138, 10045,
 10046, 10168, 10186, 10486, 10533,
 10534, 10790, 10821, 11058, 14376,
 15344, 15345, 15529, 15531, 15533
 \scantokens 639
 cscd 182
 \scriptfont 585
 \scriptscriptfont 586
 \scriptscriptstyle 431
 \scriptspace 471
 \scriptstyle 430
 \scrollmode 396
 asec 183
 asecd 183

<code>\seq_clear:c</code>	5078	<code>\seq_gpop_right:NNF</code>	5386
<code>\seq_clear:N</code>	104 , 5078 , 5078 , 5080 , 5085 , 5195 , 7629 , 7692	<code>\seq_gpop_right:NNT</code>	5385
<code>\seq_clear_new:c</code>	5084	<code>\seq_gpop_right:NNTF</code> ...	107 , 5368 , 5387
<code>\seq_clear_new:N</code> ..	104 , 5084 , 5084 , 5086	<code>\seq_gpush:cn</code>	5498 , 5513
<code>\seq_concat:ccc</code>	5135	<code>\seq_gpush:co</code>	5498 , 5516
<code>\seq_concat:NNN</code> 105 , 5135 , 5135 , 5141 , 8889		<code>\seq_gpush:cV</code>	5498 , 5514
<code>\seq_count:c</code>	5447	<code>\seq_gpush:cv</code>	5498 , 5515
<code>\seq_count:N</code>		<code>\seq_gpush:cx</code>	5498 , 5517
... 109 , 5447 , 5447 , 5456 , 5461 , 15221		<code>\seq_gpush:Nn</code>	111 , 5498 , 5508
<code>\seq_gclear:c</code>	5078	<code>\seq_gpush:No</code>	5498 , 5511 , 8941
<code>\seq_gclear:N</code> ..	104 , 5078 , 5081 , 5083 , 5088	<code>\seq_gpush:Nv</code>	5498 , 5509 , 9054 , 9147
<code>\seq_gclear_new:c</code>	5084	<code>\seq_gpush:Nv</code>	5498 , 5510
<code>\seq_gclear_new:N</code> ..	104 , 5084 , 5087 , 5089	<code>\seq_gpush:Nx</code>	5498 , 5512 , 14362
<code>\seq_gconcat:ccc</code>	5135	<code>\seq_gput_left:cn</code>	5145 , 5513
<code>\seq_gconcat:NNN</code> ..	105 , 5135 , 5137 , 5142	<code>\seq_gput_left:co</code>	5145 , 5516
<code>\seq_get:cN</code>	5518 , 5519 , 5525	<code>\seq_gput_left:cV</code>	5145 , 5514
<code>\seq_get:cNTF</code>	5524	<code>\seq_gput_left:cv</code>	5145 , 5515
<code>\seq_get:NN</code>	111 , 5518 , 5518 , 5524	<code>\seq_gput_left:cx</code>	5145 , 5517
<code>\seq_get:NNTF</code>	111 , 5524	<code>\seq_gput_left:Nn</code>	
<code>\seq_get_left:cN</code>	5286 , 5519 , 5525	... 105 , 5145 , 5153 , 5164 , 5165 , 5508	
<code>\seq_get_left:cNTF</code>	5358	<code>\seq_gput_left:No</code>	5145 , 5511
<code>\seq_get_left:NN</code>	105 , 5286 , 5286 , 5296 , 5358 , 5359 , 5518 , 5524	<code>\seq_gput_left:Nv</code>	5145 , 5509
<code>\seq_get_left:NNF</code>	5363	<code>\seq_gput_left:Nv</code>	5145 , 5510
<code>\seq_get_left:NNT</code>	5362	<code>\seq_gput_left:Nx</code>	5145 , 5512
<code>\seq_get_left:NNTF</code>	106 , 5358 , 5364	<code>\seq_gput_right:cn</code>	5166
<code>\seq_get_right:cN</code>	5311	<code>\seq_gput_right:co</code>	5166
<code>\seq_get_right:cNTF</code>	5358	<code>\seq_gput_right:cV</code>	5166
<code>\seq_get_right:NN</code>		<code>\seq_gput_right:cv</code>	5166
... 105 , 5311 , 5311 , 5329 , 5360 , 5361		<code>\seq_gput_right:cx</code>	5166
<code>\seq_get_right:NNF</code>	5366	<code>\seq_gput_right:Nn</code>	
<code>\seq_get_right:NNT</code>	5365	... 105 , 5166 , 5174 , 5183 , 5184 , 8934 , 8939	
<code>\seq_get_right:NNTF</code>	106 , 5358 , 5367	<code>\seq_gput_right:No</code>	5166 , 8982
<code>\seq_gpop:cN</code>	5518 , 5523 , 5529	<code>\seq_gput_right:Nv</code>	5166 , 8839
<code>\seq_gpop:cNTF</code>	5524	<code>\seq_gput_right:Nv</code>	5166
<code>\seq_gpop:NN</code>		<code>\seq_gput_right:Nx</code>	5166
... 111 , 5518 , 5522 , 5528 , 8944 , 14375		<code>\seq_gremove_all:cn</code>	5205
<code>\seq_gpop:NNTF</code>	111 , 5524 , 9027 , 9120	<code>\seq_gremove_all:Nn</code> 108 , 5205 , 5207 , 5230	
<code>\seq_gpop_left:cN</code>	5297 , 5523 , 5529	<code>\seq_gremove_duplicates:c</code>	5189
<code>\seq_gpop_left:cNTF</code>	5368	<code>\seq_gremove_duplicates:N</code>	
<code>\seq_gpop_left:NN</code> 107 , 5189 , 5191 , 5204	
... 106 , 5297 , 5299 , 5310 , 5370 , 5522 , 5528		<code>\seq_greverse:c</code>	15286
<code>\seq_gpop_left:NNF</code>	5380	<code>\seq_greverse:N</code> ..	197 , 15286 , 15289 , 15306
<code>\seq_gpop_left:NNT</code>	5379	<code>\seq_gset_eq:cc</code>	5090 , 5097
<code>\seq_gpop_left:NNTF</code>	107 , 5368 , 5381	<code>\seq_gset_eq:cN</code>	5090 , 5096
<code>\seq_gpop_right:cN</code>	5330	<code>\seq_gset_eq:Nc</code>	5090 , 5095
<code>\seq_gpop_right:cNTF</code>	5368	<code>\seq_gset_eq:NN</code>	
<code>\seq_gpop_right:NN</code> 104 , 5082 , 5090 , 5094 , 5192 , 9103	
... 106 , 5330 , 5332 , 5357 , 5374		<code>\seq_gset_filter:NNn</code> ..	198 , 15307 , 15309
		<code>\seq_gset_from_clist:cc</code>	15260
		<code>\seq_gset_from_clist:cN</code>	15260

<code>\seq_gset_from_clist:cn</code>	15260	<code>\seq_map_inline:Nn</code>	
<code>\seq_gset_from_clist:Nc</code>	15260	108 , 5196 , 5428 , 5428 ,
<code>\seq_gset_from_clist:NN</code>		5434 , 7644 , 8614 , 8854 , 8893 , 8975	
.....	197 , 15260 , 15270 , 15283 , 15284	<code>\seq_map_variable:ccn</code>	5435
<code>\seq_gset_from_clist:Nn</code>		<code>\seq_map_variable:cNn</code>	5435
.....	15260 , 15275 , 15285	<code>\seq_map_variable:Ncn</code>	5435
<code>\seq_gset_map:NNn</code>	198 , 15317 , 15319	<code>\seq_map_variable:NNn</code>	
<code>\seq_gset_split:Nnn</code>	108 , 5435 , 5435 , 5445 , 5446
.....	104 , 5098 , 5100 , 5134 , 8989	<code>\seq_mapthread_function:ccN</code>	15237
<code>\seq_gset_split:NnV</code>	5098	<code>\seq_mapthread_function:cNN</code>	15237
<code>\seq_if_empty:cTF</code>	5231	<code>\seq_mapthread_function:NcN</code>	15237
<code>\seq_if_empty:N</code>	5231	<code>\seq_mapthread_function:NNN</code>	
<code>\seq_if_empty:NF</code>	5241	197 , 15237 , 15237 , 15258 , 15259
<code>\seq_if_empty:NT</code>	5240	<code>\seq_new:c</code>	5072
<code>\seq_if_empty:NTF</code>		<code>\seq_new:N</code>	4 ,
.....	108 , 5231 , 5242 , 7933 , 14373 , 14876	104 , 2591 , 2608 , 5072 , 5072 , 5077 ,
<code>\seq_if_empty_p:c</code>	5231	5085 , 5088 , 5188 , 5536 , 5537 , 5538 ,	
<code>\seq_if_empty_p:N</code>	108 , 5231 , 5239	5539 , 7601 , 7602 , 8174 , 8833 , 8834 ,	
<code>\seq_if_exist:c</code>	5144	8844 , 8846 , 8849 , 8987 , 9101 , 14328	
<code>\seq_if_exist:cTF</code>	5143	<code>\seq_pop:cN</code>	5518 , 5521 , 5527
<code>\seq_if_exist:N</code>	5143	<code>\seq_pop:cNTF</code>	5524
<code>\seq_if_exist:NTF</code>		<code>\seq_pop:NN</code>	111 , 5518 , 5520 , 5526
.....	105 , 5085 , 5088 , 5143 , 5459	<code>\seq_pop:NNTF</code>	111 , 5524
<code>\seq_if_exist_p:c</code>	5143	<code>\seq_pop_left:cN</code>	5297 , 5521 , 5527
<code>\seq_if_exist_p:N</code>	105 , 5143	<code>\seq_pop_left:cNTF</code>	5368
<code>\seq_if_in:cnTF</code>	5243	<code>\seq_pop_left:NN</code>	
<code>\seq_if_in:coTF</code>	5243	106 , 5297 , 5297 , 5309 , 5368 , 5520 , 5526
<code>\seq_if_in:cvTF</code>	5243	<code>\seq_pop_left:NNTF</code>	5377
<code>\seq_if_in:cvTF</code>	5243	<code>\seq_pop_left:NNT</code>	5376
<code>\seq_if_in:cxTF</code>	5243	<code>\seq_pop_left:NNTF</code>	107 , 5368 , 5378
<code>\seq_if_in:Nn</code>	5243	<code>\seq_pop_right:cN</code>	5330
<code>\seq_if_in:NnF</code> ...	5198 , 5264 , 5265 , 8952	<code>\seq_pop_right:cNTF</code>	5368
<code>\seq_if_in:NnT</code>	5262 , 5263	<code>\seq_pop_right:NN</code>	
<code>\seq_if_in:NnTF</code> ...	108 , 5243 , 5266 , 5267	106 , 5330 , 5330 , 5356 , 5372
<code>\seq_if_in:NoTF</code>	5243	<code>\seq_pop_right:NNTF</code>	5383
<code>\seq_if_in:NVF</code>	9053 , 9146	<code>\seq_pop_right:NNT</code>	5382
<code>\seq_if_in:NVTF</code>	5243	<code>\seq_pop_right:NNTF</code>	107 , 5368 , 5384
<code>\seq_if_in:NvTF</code>	5243	<code>\seq_push:cn</code>	5498 , 5503
<code>\seq_if_in:NxTF</code>	5243	<code>\seq_push:co</code>	5498 , 5506
<code>\seq_item:cn</code>	15213	<code>\seq_push:cV</code>	5498 , 5498 , 5504
<code>\seq_item:Nn</code>	197 ,	<code>\seq_push:cv</code>	5505
.....	7709 , 7710 , 7715 , 15213 , 15213 , 15236	<code>\seq_push:cx</code>	5498 , 5507
<code>\seq_map_break:</code>		<code>\seq_push:Nn</code>	111 , 5498 , 5498
.....	109 , 5388 , 5388 , 5389 , 5391 ,	<code>\seq_push:No</code>	5498 , 5501
5396 , 5397 , 5432 , 5443 , 8621 , 8899		<code>\seq_push:Nv</code>	5498 , 5499
<code>\seq_map_break:n</code> 109 , 5388 , 5390 , 7649 , 7663		<code>\seq_push:Nx</code>	5498 , 5500
<code>\seq_map_function:cN</code>	5392	<code>\seq_put_left:cn</code>	5145 , 5503
<code>\seq_map_function:NN</code> ...	4 , 108 , 5392 ,	<code>\seq_put_left:co</code>	5145 , 5506
5392 , 5405 , 5452 , 5533 , 7713 , 14882		<code>\seq_put_left:cV</code>	5145 , 5504
<code>\seq_map_inline:cn</code>	5428		

<code>\seq_put_left:cv</code>	5145 , 5505	<code>\seq_use:Nn</code>	110 , 5457 , 5495 , 5497
<code>\seq_put_left:cx</code>	5145 , 5507	<code>\seq_use:Nnnn</code> ..	110 , 5457 , 5457 , 5480 , 5496
<code>\seq_put_left:Nn</code>	105 , 5145 , 5145 , 5162 , 5163 , 5498 , 7639	<code>\set@color</code>	7298 , 7299
<code>\seq_put_left:No</code>	5145 , 5501	<code>\setbox</code>	567
<code>\seq_put_left:Nv</code>	5145 , 5499	<code>\setlanguage</code>	325
<code>\seq_put_left:Nv</code>	5145 , 5500	<code>\sfcode</code>	622
<code>\seq_put_left:Nx</code>	5145 , 5502	<code>\sffamily</code>	7065
<code>\seq_put_right:cn</code>	5166	<code>\shipout</code>	532
<code>\seq_put_right:co</code>	5166	<code>\show</code>	375
<code>\seq_put_right:cV</code>	5166	<code>\showbox</code>	377
<code>\seq_put_right:cv</code>	5166	<code>\showboxbreadth</code>	391
<code>\seq_put_right:cx</code>	5166	<code>\showboxdepth</code>	392
<code>\seq_put_right:Nn</code>	105 , 5166 , 5166 , 5185 , 5186 , 5199 , 7700 , 8953	<code>\showgroups</code>	652
<code>\seq_put_right:No</code>	5166 , 8968	<code>\showifs</code>	653
<code>\seq_put_right:Nv</code>	5166	<code>\showlists</code>	378
<code>\seq_put_right:Nv</code>	5166	<code>\showthe</code>	376
<code>\seq_put_right:Nx</code>	5166	<code>\showtokens</code>	640
<code>\seq_remove_all:cn</code>	5205	<code>asin</code>	183
<code>\seq_remove_all:Nn</code>	108 , 5205 , 5205 , 5229 , 8958	<code>asind</code>	183
<code>\seq_remove_duplicates:c</code>	5189	<code>\skewchar</code>	589
<code>\seq_remove_duplicates:N</code>	107 , 5189 , 5189 , 5203 , 8973	<code>\skip</code>	613
<code>\seq_reverse:c</code>	15286	<code>\skip_add:cn</code>	4192
<code>\seq_reverse:N</code> ..	197 , 15286 , 15287 , 15305	<code>\skip_add:Nn</code> ...	83 , 4192 , 4192 , 4194 , 4195
<code>\seq_set_eq:cc</code>	5090 , 5093	<code>\skip_const:cn</code>	4163
<code>\seq_set_eq:cN</code>	5090 , 5092	<code>\skip_const:Nn</code>	83 , 4163 , 4163 , 4168 , 4239 , 4240
<code>\seq_set_eq:Nc</code>	5090 , 5091	<code>\skip_eval:n</code>	84 , 4205 , 4223 , 4223
<code>\seq_set_eq:NN</code>	104 , 5079 , 5090 , 5090 , 5190 , 8887 , 8904 , 8962	<code>\skip_gadd:cn</code>	4192
<code>\seq_set_filter:NNn</code> ...	198 , 15307 , 15307	<code>\skip_gadd:Nn</code>	83 , 4192 , 4194 , 4196
<code>\seq_set_from_clist:cc</code>	15260	<code>.skip_gset:c</code>	154
<code>\seq_set_from_clist:cN</code>	15260	<code>\skip_gset:cn</code>	4181
<code>\seq_set_from_clist:cn</code>	15260	<code>.skip_gset:N</code>	154
<code>\seq_set_from_clist:Nc</code>	15260	<code>\skip_gset:Nn</code> ..	83 , 4166 , 4181 , 4183 , 4185
<code>\seq_set_from_clist:NN</code>	197 , 15260 , 15260 , 15280 , 15281	<code>\skip_gset_eq:cc</code>	4186
<code>\seq_set_from_clist:Nn</code>	8535 , 8544 , 15260 , 15265 , 15282	<code>\skip_gset_eq:cN</code>	4186
<code>\seq_set_map:NNn</code>	198 , 15317 , 15317	<code>\skip_gset_eq:Nc</code>	4186
<code>\seq_set_split:Nnn</code>	104 , 2604 , 2609 , 5098 , 5098 , 5133	<code>\skip_gset_eq:NN</code>	84 , 4186 , 4189 , 4190 , 4191
<code>\seq_set_split:NnV</code>	5098 , 8888	<code>\skip_gsub:cn</code>	4192
<code>\seq_show:c</code>	5530	<code>\skip_gsub:Nn</code>	84 , 4192 , 4199 , 4201
<code>\seq_show:N</code>	112 , 5530 , 5530 , 5535	<code>\skip_gzero:c</code>	4169
<code>\seq_tmp:w</code>	5336 , 5347	<code>\skip_gzero:N</code> ..	83 , 4169 , 4170 , 4172 , 4176
<code>\seq_use:cn</code>	5457	<code>\skip_gzero_new:c</code>	4173
<code>\seq_use:cnnn</code>	5457	<code>\skip_gzero_new:N</code> ..	83 , 4173 , 4175 , 4178
		<code>\skip_horizontal:c</code>	4227
		<code>\skip_horizontal:N</code>	86 , 4227 , 4227 , 4229 , 4233
		<code>\skip_horizontal:n</code>	4227 , 4228
		<code>\skip_if_eq:nn</code>	4202
		<code>\skip_if_eq:nnTF</code>	84 , 4202
		<code>\skip_if_eq_p:nn</code>	84 , 4202

<code>\skip_if_exist:c</code>	4180	<code>\str_case:nnF</code>	1526, 1587, 2050, 15438
<code>\skip_if_exist:cTF</code>	4179	<code>\str_case:nnn</code>	1587, 1587
<code>\skip_if_exist:N</code>	4179	<code>\str_case:nnT</code>	1521, 2049
<code>\skip_if_exist:NTF</code> ..	83, 4174, 4176, 4179	<code>\str_case:nnTF</code>	22, 1516, 1531, 2051
<code>\skip_if_exist_p:c</code>	4179	<code>\str_case:on</code>	2040
<code>\skip_if_exist_p:N</code>	83, 4179	<code>\str_case:onF</code>	2052
<code>\skip_if_finite:n</code>	4214	<code>\str_case:onn</code>	2052, 2052
<code>\skip_if_finite:nTF</code>	84, 4212, 15342	<code>\str_case:onTF</code>	2040
<code>\skip_if_finite_p:n</code>	84, 4212	<code>\str_case_x:nn</code>	1516, 1544
<code>\skip_new:c</code>	4155	<code>\str_case_x:nnF</code>	1554, 1588
<code>\skip_new:N</code> 83, 4155, 4156, 4162, 4165,		<code>\str_case_x:nnn</code>	1587, 1588
4174, 4176, 4241, 4242, 4243, 4244		<code>\str_case_x:nnT</code>	1549
<code>.skip_set:c</code>	154	<code>\str_case_x:nnTF</code>	23, 1516, 1559
<code>\skip_set:cn</code>	4181	<code>\str_head:n</code> ...	101, 4917, 4917, 4941, 4988
<code>.skip_set:N</code>	154	<code>\str_if_eq:nn</code>	1497
<code>\skip_set:Nn</code> ...	83, 4181, 4181, 4183, 4184	<code>\str_if_eq:nnF</code>	2044, 2045, 7918
<code>\skip_set_eq:cc</code>	4186	<code>\str_if_eq:nnT</code> 2042, 2043, 5213, 7660, 8618	
<code>\skip_set_eq:cN</code>	4186	<code>\str_if_eq:nnTF</code>	22, 1497, 1540,
<code>\skip_set_eq:Nc</code>	4186	2046, 2047, 3842, 3845, 7485, 8091	
<code>\skip_set_eq:NN</code> 84, 4186, 4186, 4187, 4188		<code>\str_if_eq:noTF</code>	2040
<code>\skip_show:c</code>	4235	<code>\str_if_eq:nVTF</code>	2040
<code>\skip_show:N</code>	85, 4235, 4235, 4236	<code>\str_if_eq:onTF</code>	2040
<code>\skip_show:n</code>	85, 4237, 4237	<code>\str_if_eq:VnTF</code>	2040
<code>\skip_split_finite_else_action:nnNN</code>		<code>\str_if_eq:VVTF</code>	2040
.....	199, 15340, 15340	<code>\str_if_eq_p:nn</code>	22, 1497, 2040, 2041
<code>\skip_sub:cn</code>	4192	<code>\str_if_eq_p:no</code>	2040
<code>\skip_sub:Nn</code> ...	84, 4192, 4197, 4199, 4200	<code>\str_if_eq_p:nV</code>	2040
<code>\skip_use:c</code>	4225	<code>\str_if_eq_p:on</code>	2040
<code>\skip_use:N</code> 85, 4217, 4224, 4225, 4225, 4226		<code>\str_if_eq_p:Vn</code>	2040
<code>\skip_vertical:c</code>	4227	<code>\str_if_eq_p:VV</code>	2040
<code>\skip_vertical:N</code> 86, 4227, 4230, 4232, 4234		<code>\str_if_eq_x:nn</code>	1503
<code>\skip_vertical:n</code>	4227, 4231	<code>\str_if_eq_x:nnF</code>	7703, 8203
<code>\skip_zero:c</code>	4169	<code>\str_if_eq_x:nnTF</code>	
<code>\skip_zero:N</code> 83, 4169, 4169, 4170, 4171, 4174		... 22, 1497, 1568, 6148, 9301, 15207	
<code>\skip_zero_new:c</code>	4173	<code>\str_if_eq_x_p:nn</code>	22, 1497
<code>\skip_zero_new:N</code> ...	83, 4173, 4173, 4177	<code>\str_tail:n</code>	101, 4917, 4925
<code>\skipdef</code>	312	<code>\strcmp</code>	70
<code>\space</code>	50, 212	<code>\string</code>	77, 149, 230, 594
<code>\spacefactor</code>	531		
<code>\spaceskip</code>	526		
<code>\span</code>	339		
<code>\special</code>	601		
<code>\splitbotmark</code>	410		
<code>\splitbotmarks</code>	636		
<code>\splitdiscards</code>	683		
<code>\splitfirstmark</code>	409		
<code>\splitfirstmarks</code>	635		
<code>\splitmaxdepth</code>	579		
<code>\splittopskip</code>	580		
<code>\str_case:nn</code>	1516, 1516, 2048		

T

<code>\T</code>	2688, 2887, 10096, 15495
<code>pt</code>	184
<code>\tabskip</code>	340
<code>atan</code>	183
<code>atand</code>	184
<code>\tex_above:D</code>	422
<code>\tex_abovedisplayshortskip:D</code>	435
<code>\tex_abovedisplayskip:D</code>	436
<code>\tex_abovewithdelims:D</code>	423
<code>\tex_accent:D</code>	473

<code>\tex_adjdemerits:D</code>	510	<code>\tex_displaylimits:D</code>	450
<code>\tex_advance:D</code>		<code>\tex_displaystyle:D</code>	428
..... 317, 3288, 3290, 3300, 3302,		<code>\tex_displaywidowpenalty:D</code>	439
3954, 3959, 4193, 4198, 4284, 4289		<code>\tex_displaywidth:D</code>	441
<code>\tex_afterassignment:D</code>		<code>\tex_divide:D</code>	318
..... 327, 2958, 5320, 5344		<code>\tex_doublehyphendemerits:D</code>	508
<code>\tex_aftergroup:D</code>	328, 779	<code>\tex_dp:D</code>	619, 6268
<code>\tex_atop:D</code>	424	<code>\tex_dump:D</code>	602
<code>\tex_atopwithdelims:D</code>	425	<code>\tex_edef:D</code>	306, 814
<code>\tex_badness:D</code>	572	<code>\tex_else:D</code>	359, 753, 810
<code>\tex_baselineskip:D</code>	500	<code>\tex_emergencystretch:D</code>	523
<code>\tex_batchmode:D</code>	393	<code>\tex_end:D</code>	397, 726, 1169, 7535
<code>\tex_begingroup:D</code>	331, 775	<code>\tex_endcsname:D</code>	399, 767
<code>\tex_belowdisplayshortskip:D</code>	437	<code>\tex_endgroup:D</code>	332, 724, 776
<code>\tex_belowdisplayskip:D</code>	438	<code>\tex_endinput:D</code>	371, 7546
<code>\tex_binoppenalty:D</code>	461	<code>\tex_endlinechar:D</code>	
<code>\tex_botmark:D</code>	408 262, 263, 277, 413, 4469, 9092, 9094	
<code>\tex_box:D</code>	616, 6260, 6282	<code>\tex_eqno:D</code>	433
<code>\tex_boxmaxdepth:D</code>	578	<code>\tex_errhelp:D</code>	379, 7423
<code>\tex_brokenpenalty:D</code>	535	<code>\tex_errmessage:D</code>	373, 1161, 7450
<code>\tex_catcode:D</code>	620, 1062, 1445,	<code>\tex_errorcontextlines:D</code>	380, 7475
1862, 1863, 1864, 1865, 2010, 2312,		<code>\tex_errorstopmode:D</code>	394
2327, 2478, 2480, 2482, 4452, 4453		<code>\tex_escapechar:D</code>	412, 9188, 9214, 9220
<code>\tex_char:D</code>	474	<code>\tex_everycr:D</code>	341
<code>\tex_chardef:D</code>	309, 803, 804, 805, 806,	<code>\tex_everydisplay:D</code>	442, 727
807, 809, 1049, 1050, 2086, 2088,		<code>\tex_everyhbox:D</code>	581
2883, 3266, 3267, 9043, 9136, 14336		<code>\tex_everyjob:D</code>	610, 4366, 4368,
<code>\tex_cleaders:D</code>	492	4373, 4375, 8824, 8826, 8836, 8838	
<code>\tex_closein:D</code>	368, 9051	<code>\tex_everymath:D</code>	466, 728
<code>\tex_closeout:D</code>	363, 9144	<code>\tex_everypar:D</code>	529
<code>\tex_clubpenalty:D</code>	503	<code>\tex_everyvbox:D</code>	582
<code>\tex_copy:D</code>	560, 6254, 6283	<code>\tex_exhyphenpenalty:D</code>	505
<code>\tex_count:D</code>	611, 2778	<code>\tex_expandafter:D</code>	329, 768
<code>\tex_countdef:D</code>	310, 800, 2781	<code>\tex_fam:D</code>	321
<code>\tex_cr:D</code>	335	<code>\tex_fi:D</code>	360, 754, 812
<code>\tex_crcr:D</code>	336	<code>\tex_finalhyphendemerits:D</code>	509
<code>\tex_csname:D</code>	398, 766	<code>\tex_firstmark:D</code>	407
<code>\tex_day:D</code>	606	<code>\tex_floatingpenalty:D</code>	554
<code>\tex_deadcycles:D</code>	540	<code>\tex_font:D</code>	320
<code>\tex_def:D</code> ... 305, 780, 782, 784, 785, 813		<code>\tex_fontdimen:D</code>	587
<code>\tex_defaultthyphenchar:D</code>	590	<code>\tex_fontname:D</code>	411
<code>\tex_defaultskewchar:D</code>	591	<code>\tex_futurelet:D</code>	316, 2952, 2954
<code>\tex_delcode:D</code>	621	<code>\tex_gdef:D</code>	307, 827
<code>\tex_delimiter:D</code>	415	<code>\tex_global:D</code>	291, 296,
<code>\tex_delimiterfactor:D</code>	464	298, 322, 1256, 1263, 2088, 2954,	
<code>\tex_delimitershortfall:D</code>	463	3250, 3270, 3282, 3292, 3294, 3304,	
<code>\tex_dimen:D</code>	612, 2756	3306, 3313, 3931, 3944, 3950, 3955,	
<code>\tex_dimendef:D</code>	311, 2759	3960, 4170, 4183, 4189, 4194, 4199,	
<code>\tex_discretionary:D</code>	475	4261, 4274, 4280, 4285, 4290, 6256,	
<code>\tex_displayindent:D</code>	440		

6262, 6318, 6363, 6369, 6375, 6405, 6411, 6417, 6423, 9043, 9136, 14336	<code>\tex_insertpenalties:D</code> 555
<code>\tex_globaldefs:D</code> 326	<code>\tex_interlinepenalty:D</code> 534
<code>\tex_halign:D</code> 333	<code>\tex_italiccorrection:D</code> 302, 731
<code>\tex_hangafter:D</code> 511	<code>\tex_jobname:D</code> 609, 4376, 4380, 8827
<code>\tex_hangindent:D</code> 512	<code>\tex_kern:D</code> 487, 6861, 6866, 6932, 6933, 7220, 7221, 14519, 14704, 14713, 14726, 14728, 14769, 14771, 14940
<code>\tex_hbadness:D</code> 573	<code>\tex_language:D</code> 404
<code>\tex_hbox:D</code> 568, 6361, 6362, 6367, 6373, 6387, 6388	<code>\tex_lastbox:D</code> 561, 6316
<code>\tex_hfil:D</code> 476	<code>\tex_lastkern:D</code> 494
<code>\tex_hfill:D</code> 478	<code>\tex_lastpenalty:D</code> 600
<code>\tex_hfilneg:D</code> 477	<code>\tex_lastskip:D</code> 495
<code>\tex_hfuzz:D</code> 575	<code>\tex_lccode:D</code> 623, 1061, 1444, 2011, 2311, 2326, 2554, 2556, 2558, 4450, 4451
<code>\tex_hoffset:D</code> 550	<code>\tex_leaders:D</code> 491
<code>\tex_holdinginserts:D</code> 553	<code>\tex_left:D</code> 459
<code>\tex_hruler:D</code> 489	<code>\tex_lefthyphenmin:D</code> 515
<code>\tex_hsize:D</code> 514, 6549, 6551, 6552, 6595, 6597, 6598	<code>\tex_leftskip:D</code> 517
<code>\tex_hskip:D</code> 479, 4227	<code>\tex_leqno:D</code> 434
<code>\tex_hss:D</code> 480, 6390, 6392, 14706, 14715	<code>\tex_let:D</code> 292, 296, 298, 304, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 795, 797, 813, 814, 827, 828, 1252, 2060, 2061
<code>\tex_ht:D</code> 618, 6267	<code>\tex_limits:D</code> 451
<code>\tex_hyphen:D</code> 303, 729	<code>\tex_linepenalty:D</code> 507
<code>\tex_hyphenation:D</code> 604	<code>\tex_lineskip:D</code> 501
<code>\tex_hyphenchar:D</code> 588	<code>\tex_lineskiplimit:D</code> 502
<code>\tex_hyphenpenalty:D</code> 506	<code>\tex_long:D</code> 323, 780, 782, 785, 816, 818, 824, 826, 830, 832, 838, 840
<code>\tex_if:D</code> 342, 756, 757	<code>\tex_looseness:D</code> 519
<code>\tex_ifcase:D</code> 343, 3160	<code>\tex_lower:D</code> 556, 6293
<code>\tex_ifcat:D</code> 344, 758	<code>\tex_lowercase:D</code> 595, 1063, 1446, 1866, 2012, 4454, 4494
<code>\tex_ifdim:D</code> 347, 3913	<code>\tex_mag:D</code> 403
<code>\tex_ifeof:D</code> 348, 9069	<code>\tex_mark:D</code> 405
<code>\tex_iffalse:D</code> 353, 751	<code>\tex_mathaccent:D</code> 416
<code>\tex_ifhbox:D</code> 349, 6294	<code>\tex_mathbin:D</code> 446
<code>\tex_ifhmode:D</code> 355, 761	<code>\tex_mathchar:D</code> 417
<code>\tex_ifinner:D</code> 358, 763	<code>\tex_mathchardef:D</code> 314, 811, 3262, 3263
<code>\tex_ifmmode:D</code> 356, 760	<code>\tex_mathchoice:D</code> 414
<code>\tex_ifnum:D</code> 345, 777	<code>\tex_mathclose:D</code> 447
<code>\tex_ifodd:D</code> 346, 1186, 2060, 2061, 3159, 7328	<code>\tex_mathcode:D</code> 625, 2548, 2550, 2552
<code>\tex_iftrue:D</code> 354, 750	<code>\tex_mathinner:D</code> 448
<code>\tex_ifvbox:D</code> 350, 6295	<code>\tex_mathop:D</code> 449
<code>\tex_ifvmode:D</code> 357, 762	
<code>\tex_ifvoid:D</code> 351, 6296	
<code>\tex_ifx:D</code> 352, 759	
<code>\tex_ignorespaces:D</code> 400	
<code>\tex_immediate:D</code> 362, 1156, 1158, 9138, 9144, 9162	
<code>\tex_indent:D</code> 496	
<code>\tex_input:D</code> 370, 730, 8943	
<code>\tex_inputlineno:D</code> 372, 1176, 2005, 7397	
<code>\tex_insert:D</code> 552	

<code>\tex_mathopen:D</code>	453	<code>\tex_parindent:D</code>	521
<code>\tex_mathord:D</code>	454	<code>\tex_parshape:D</code>	513
<code>\tex_mathpunct:D</code>	455	<code>\tex_parskip:D</code>	520
<code>\tex_mathrel:D</code>	456	<code>\tex_patterns:D</code>	603
<code>\tex_mathsurround:D</code>	467	<code>\tex_pausing:D</code>	390
<code>\tex_maxdeadcycles:D</code>	537	<code>\tex_penalty:D</code>	598
<code>\tex_maxdepth:D</code>	538	<code>\tex_postdisplaypenalty:D</code>	445
<code>\tex_meaning:D</code>	597, 771, 773	<code>\tex_predisdisplaypenalty:D</code>	444
<code>\tex_medmuskip:D</code>	468	<code>\tex_predisplaysize:D</code>	443
<code>\tex_message:D</code>	374	<code>\tex_pretolerance:D</code>	524
<code>\tex_mkern:D</code>	421	<code>\tex_prevdepth:D</code>	571
<code>\tex_month:D</code>	607	<code>\tex_prevgraf:D</code>	530
<code>\tex_moveleft:D</code>	557, 6287	<code>\tex_radical:D</code>	419
<code>\tex moveright:D</code>	558, 6289	<code>\tex_raise:D</code>	559, 6291
<code>\tex_mskip:D</code>	418	<code>\tex_read:D</code>	366, 9087
<code>\tex_multiply:D</code>	319	<code>\tex_relax:D</code>	401, 774, 3158, 3915
<code>\tex_muskip:D</code>	615, 2798	<code>\tex_relpenalty:D</code>	462
<code>\tex_muskipdef:D</code>	313, 2801	<code>\tex_right:D</code>	460
<code>\tex_newlinechar:D</code>	369, 4470	<code>\tex_righthyphenmin:D</code>	516
<code>\tex_noalign:D</code>	337	<code>\tex_rightskip:D</code>	518
<code>\tex_noboundary:D</code>	472	<code>\tex_romannumeral:D</code>	593,
<code>\tex_noexpand:D</code>	330, 769		778, 1518, 1523, 1528, 1533, 1546,
<code>\tex_noindent:D</code>	498		1551, 1556, 1561, 1610, 1622, 1628,
<code>\tex_nolimits:D</code>	452		1668, 1672, 1677, 1683, 1689, 1695,
<code>\tex_nonscript:D</code>	432		1707, 1712, 1714, 1721, 1776, 1783,
<code>\tex_nonstopmode:D</code>	395		1788, 1796, 1798, 1801, 1808, 1814,
<code>\tex_nulldelimiterspace:D</code>	465		1823, 1839, 1843, 1848, 3046, 3391,
<code>\tex_nullfont:D</code>	583, 2910		3396, 3401, 3406, 4010, 4041, 4046,
<code>\tex_number:D</code>	592, 3156		4051, 4056, 4654, 4659, 4664, 4669,
<code>\tex_omit:D</code>	338		4865, 5028, 7953, 9477, 9539, 9642,
<code>\tex_openin:D</code>	364, 9045		9834, 9981, 9999, 10028, 10069,
<code>\tex_openout:D</code>	365, 9138		10076, 10081, 10087, 10092, 10129,
<code>\tex_or:D</code>	361, 752		10142, 10143, 10149, 10159, 10177,
<code>\tex_outer:D</code>	324		10178, 10198, 10211, 10215, 10237,
<code>\tex_output:D</code>	539		10265, 10278, 10291, 10315, 10326,
<code>\tex_outputpenalty:D</code>	549		10333, 10336, 10355, 10365, 10368,
<code>\tex_over:D</code>	426		10381, 10384, 10396, 10419, 10448,
<code>\tex_overfullrule:D</code>	577		10462, 10478, 10498, 10509, 10515,
<code>\tex_overline:D</code>	457		10525, 10570, 10580, 10595, 10606,
<code>\tex_overwithdelims:D</code>	427		10621, 10633, 10667, 10676, 10685,
<code>\tex_pagedepth:D</code>	541		10733, 10742, 10743, 10747, 10753,
<code>\tex_pagefilllstretch:D</code>	545		10755, 10768, 10771, 10778, 10779,
<code>\tex_pagefillstretch:D</code>	544		10830, 10844, 10863, 10866, 10894,
<code>\tex_pagefilstretch:D</code>	543		10919, 10961, 10973, 10988, 11004,
<code>\tex_pagegoal:D</code>	547		11032, 11079, 11090, 11097, 11107,
<code>\tex_pageshrink:D</code>	546		11135, 11149, 11159, 11335, 11338,
<code>\tex_pagestretch:D</code>	542		11346, 11369, 11377, 12155, 12473,
<code>\tex_pagetotal:D</code>	548		12681, 12685, 12704, 12741, 12862,
<code>\tex_par:D</code>	497, 7281		13030, 13221, 13613, 13614, 13618,
<code>\tex_parfillskip:D</code>	528		13887, 13896, 13942, 13946, 13991,

14010, 14014, 14047, 14051, 14117, 14245, 15364, 15414, 15422, 15445	\tex_tracingcommands:D 381
\tex_scriptfont:D 585	\tex_tracinglostchars:D 382
\tex_scriptscriptfont:D 586	\tex_tracingmacros:D 383
\tex_scriptscriptstyle:D 431	\tex_tracingonline:D 384, 6352
\tex_scriptspace:D 471	\tex_tracingoutput:D 385
\tex_scriptstyle:D 430	\tex_tracingpages:D 386
\tex_scrollmode:D 396	\tex_tracingparagraphs:D 387
\tex_setbox:D	\tex_tracingrestores:D 388
567, 6254, 6260, 6316, 6362, 6367, 6373, 6404, 6409, 6415, 6421, 6443	\tex_tracingstats:D 389
\tex_setlanguage:D 325	\tex_uccode:D 624, 2560, 2562, 2564
\tex_sfcode:D 622, 2566, 2568, 2570	\tex_uchyph:D 522
\tex_shipout:D 532	\tex_undefined:D 291, 298, 1269, 1277, 9658, 9669, 9670, 9671, 9672
\tex_show:D 375	\tex_underline:D 458, 732
\tex_showbox:D 377, 6354	\tex_unhbox:D 563, 6394
\tex_showboxbreadth:D 391, 6350	\tex_unhcopy:D 564, 6393
\tex_showboxdepth:D 392, 6351	\tex_unkern:D 488
\tex_showlists:D 378	\tex_unpenalty:D 599
\tex_showthe:D	\tex_unskip:D 486
376, 1435, 2482, 2552, 2558, 2564, 2570	\tex_unvbox:D 565, 6439
\tex_skewchar:D 589	\tex_unvcopy:D 566, 6438
\tex_skip:D 613, 2816	\tex_uppercase:D 596, 4496
\tex_skipdef:D 312, 2819	\tex_vadjust:D 499
\tex_space:D 301	\tex_valign:D 334
\tex_spacefactor:D 531	\tex_vbadness:D 574
\tex_spaceskip:D 526	\tex_vbox:D
\tex_span:D 339	569, 6397, 6400, 6402, 6404, 6415, 6421
\tex_special:D 601	\tex_vcenter:D 420
\tex_splitbotmark:D 410	\tex_vfil:D 481
\tex_splitfirstmark:D 409	\tex_vfill:D 483
\tex_splitmaxdepth:D 579	\tex_vfilneg:D 482
\tex_splittopskip:D 580	\tex_vfuzz:D 576
\tex_string:D 594, 772	\tex_voffset:D 551
\tex_tabskip:D 340	\tex_vrule:D 490, 7072, 7127
\tex_textfont:D 584	\tex_vsize:D 533
\tex_textstyle:D 429	\tex_vskip:D 484, 4230
\tex_the:D 263, 402, 1176, 1640, 1644, 2005, 2480, 2550, 2556, 2562, 2568, 3316, 4143, 4225, 4238, 4295, 4300, 4368, 4375, 6343, 8826, 8838, 10106, 10549, 14362	\tex_vsplit:D 562, 6443
\tex_thickmuskip:D 470	\tex_vss:D 485
\tex_thinmuskip:D 469	\tex_vtop:D 570, 6398, 6409
\tex_time:D 605	\tex_wd:D 617, 6269
\tex_toks:D 614, 2834	\tex_widowpenalty:D 504
\tex_toksdef:D 315, 2837	\tex_write:D
\tex_tolerance:D 525	. . . 367, 1156, 1158, 9156, 9159, 9162
\tex_topmark:D 406	\tex_xdef:D 308, 828
\tex_topskip:D 536	\tex_xleaders:D 493
	\tex_xspaceskip:D 527
	\tex_year:D 608
	\texdir 723
	\textfont 584
	\textstyle 429

\TeXeTstate	684	\tl_gclear_new:N	91, 4340, 4342, 4345, 5559
\the	112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 402	\tl_gconcat:ccc	4354
\thickmuskip	470	\tl_gconcat:NNN	91, 4354, 4356, 4359
\thinmuskip	469	\tl_gput_left:cn	4401
\time	605	\tl_gput_left:co	4401
\tiny	7065	\tl_gput_left:cV	4401
\tl_case:cn	4652	\tl_gput_left:cx	4401
\tl_case:cnF	5055	\tl_gput_left:Nn ...	92, 4401, 4409, 4421
\tl_case:cnn	5054, 5055	\tl_gput_left:No	4401, 4413, 4423
\tl_case:cnTF	4652	\tl_gput_left:NV	4401, 4411, 4422
\tl_case:Nn	4652, 4652, 4680	\tl_gput_left:Nx	4401, 4415, 4424
\tl_case:NnF	4662, 4682, 5054	\tl_gput_right:cn	4425
\tl_case:Nnn	5054, 5054	\tl_gput_right:co	4425
\tl_case:NnT	4657, 4681	\tl_gput_right:cV	4425
\tl_case:NnTF	95, 4652, 4667, 4683	\tl_gput_right:cx	4425
\tl_clear:c	4334, 5554	\tl_gput_right:Nn	92, 2457, 4425, 4433, 4445
\tl_clear:N	91, 4334, 4334, 4338, 4341, 5553, 8023, 8067, 8071, 8643, 9237, 9239, 9243, 9310	\tl_gput_right:No	4425, 4437, 4447
\tl_clear_new:c	4340, 5558	\tl_gput_right:NV	4425, 4435, 4446
\tl_clear_new:N	91, 4340, 4340, 4344, 5557	\tl_gput_right:Nx	4425, 4439, 4448
\tl_concat:ccc	4354	\tl_gremove_all:cn	4553
\tl_concat:NNN	91, 4354, 4354, 4358	\tl_gremove_all:Nn ..	93, 4553, 4555, 4558
\tl_const:cn	4322	\tl_gremove_once:cn	4547
\tl_const:cx	4322, 9195	\tl_gremove_once:Nn .	92, 4547, 4549, 4552
\tl_const:Nn	91, 2351, 2589, 4322, 4322, 4332, 4362, 4382, 4457, 5071, 5963, 7312, 7313, 7365, 7370, 7372, 7374, 7376, 7378, 7383, 7384, 7391, 8160, 8161, 8162, 9185, 9406, 9407, 9408, 9409, 9410, 11972, 12413, 12414, 12415, 12416, 12417, 12418, 12419, 12420, 12421, 15400, 15405	\tl_greplace_all:cnn	4497
\tl_const:Nx	4322, 4327, 4333, 4376, 4380, 9189, 14133, 14256, 14899	\tl_greplace_all:Nnn	92, 4497, 4503, 4508, 4556
\tl_count:c	4740	\tl_greplace_once:cn	4497
\tl_count:N	98, 4740, 4745, 4752, 9235	\tl_greplace_once:Nnn	92, 4497, 4499, 4506, 4550
\tl_count:n	98, 911, 915, 1314, 1352, 4740, 4740, 4751, 15454	\tl_greverse:c	4881
\tl_count:o	4740	\tl_greverse:N	99, 4881, 4883, 4886
\tl_count:V	4740	.tl_gset:c	154
\tl_count_tokens:n	199, 15384, 15384, 15399	\tl_gset:cf	4383
\tl_expandable_lowercase:n	199, 15410, 15418	\tl_gset:cn	4383
\tl_expandable_uppercase:n	199, 15410, 15410	\tl_gset:co	4383
\tl_gclear:c	4334, 5556	\tl_gset:cV	4383
\tl_gclear:N	91, 4334, 4336, 4339, 4343, 5555	\tl_gset:cv	4383
\tl_gclear_new:c	4340, 5560	\tl_gset:cx	4383
		.tl_gset:N	154
		\tl_gset:Nf	4383, 5138
		\tl_gset:Nn	92, 4383, 4389, 4398, 4400, 4462, 5300, 5371, 6015, 6044, 6067, 8942
		\tl_gset:No	4383, 4391
		\tl_gset:NV	4383
		\tl_gset:Nv	4383
		\tl_gset:Nx	4357, 4383, 4393, 4399, 4500, 4504, 4774, 4884, 5101, 5155, 5176, 5208, 5333, 5375, 5572, 5616, 5659, 5697, 5749,

6079, 6108, 8827, 14131, 14197, 14207, 14265, 14287, 14873, 15180, 15272, 15277, 15290, 15310, 15320	\tl_if_empty:o 4602
\tl_gset_eq:cc 4346 , 4353 , 5097 , 5568 , 5989	\tl_if_empty:oTF 2902, 4593 , 4640 , 5033 , 5791 , 14889 , 14910
\tl_gset_eq:cN 4346 , 4351 , 5096 , 5567 , 5988	\tl_if_empty:VTF 4581
\tl_gset_eq:Nc 4346 , 4352 , 5095 , 5566 , 5987	\tl_if_empty_p:c 4569
\tl_gset_eq:NN 91 , 4337 , 4346 , 4350 , 5094 , 5565 , 5986 , 8831 , 8945 , 14138 , 14233	\tl_if_empty_p:N 94 , 4569 , 4577
\tl_gset_rescan:cn 4459	\tl_if_empty_p:n 94 , 4581 , 4589
\tl_gset_rescan:cno 4459	\tl_if_empty_p:o 4593
\tl_gset_rescan:cnx 4459	\tl_if_empty_p:V 4581
\tl_gset_rescan:Nnn 93 , 4459 , 4461 , 4491 , 4492	\tl_if_eq:ccTF 4604
\tl_gset_rescan:Nno 4459	\tl_if_eq:cNTF 4604
\tl_gset_rescan:Nnx 4459	\tl_if_eq:NcTF 4604
.tl_gset_x:c 154	\tl_if_eq:NN 4604
.tl_gset_x:N 154	\tl_if_eq:nn 4616
\tl_gtrim_spaces:c 4769	\tl_if_eq:NNF 4615
\tl_gtrim_spaces:N .. 99 , 4769 , 4773 , 4776	\tl_if_eq:NNT 4614 , 5220 , 7138 , 7141
\tl_head:f 4887	\tl_if_eq:NNTF 94 , 4604 , 4613 , 4676 , 6132 , 7651 , 7705 , 9256
\tl_head:N 100 , 4887 , 4907	\tl_if_eq:nnTF 94 , 4616
\tl_head:n 4887 , 4887 , 4905 , 4907	\tl_if_eq_p:cc 4604
\tl_head:V 4887	\tl_if_eq_p:cN 4604
\tl_head:v 4887	\tl_if_eq_p:Nc 4604
\tl_head:w 100 , 4887 , 4906 , 4924 , 4939 , 4958 , 4980	\tl_if_eq_p:NN 94 , 4604 , 4612
\tl_if_blank:n 4559	\tl_if_exist:c 4361
\tl_if_blank:nF .. 3833 , 4563 , 4567 , 5903	\tl_if_exist:cTF 4360
\tl_if_blank:nT 4562 , 4566	\tl_if_exist:N 4360
\tl_if_blank:nTF 94 , 4559 , 4564 , 4568 , 4911 , 14852	\tl_if_exist:NTF 91 , 4341 , 4343 , 4360 , 4736 , 5040
\tl_if_blank:oTF 4559 , 8080	\tl_if_exist_p:c 4360
\tl_if_blank:VTF 4559	\tl_if_exist_p:N 91 , 4360
\tl_if_blank_p:n ... 94 , 4559 , 4561 , 4565	\tl_if_head_eq_catcode:nN 4951
\tl_if_blank_p:o 4559	\tl_if_head_eq_catcode:nNTF .. 101 , 4932
\tl_if_blank_p:V 4559	\tl_if_head_eq_catcode_p:nN .. 101 , 4932
\tl_if_empty:c 5778	\tl_if_head_eq_charcode:fNTF 4932
\tl_if_empty:cTF 4569	\tl_if_head_eq_charcode:nN 4932
\tl_if_empty:N 4569 , 5777	\tl_if_head_eq_charcode:nNF 4950
\tl_if_empty:n 4581	\tl_if_head_eq_charcode:nNT 4949
\tl_if_empty:NF 4579	\tl_if_head_eq_charcode:nNTF 101 , 3738 , 3751 , 4932 , 4948
\tl_if_empty:nF 940 , 1018 , 3062 , 4592 , 5829 , 7804 , 7808 , 14100	\tl_if_head_eq_charcode_p:fN 4932
\tl_if_empty:NT 4578	\tl_if_head_eq_charcode_p:nN 101 , 4932 , 4947
\tl_if_empty:nT 4591	\tl_if_head_eq_meaning:nN 4970
\tl_if_empty:NTF ... 94 , 4569 , 4580 , 8126	\tl_if_head_eq_meaning:nNTF 101 , 4932 , 8107
\tl_if_empty:nTF 94 , 3875 , 4511 , 4581 , 4590 , 5104 , 5602 , 7406 , 7670 , 7685 , 8022 , 8246 , 15526	\tl_if_head_eq_meaning_p:nN .. 101 , 4932
	\tl_if_head_is_group:n 5011
	\tl_if_head_is_group:nTF 102 , 4827 , 4961 , 4996 , 5011
	\tl_if_head_is_group_p:n 102 , 5011

<code>\tl_if_head_is_N_type:n</code>	5005	<code>\tl_new:N</code>	91,
<code>\tl_if_head_is_N_type:nTF</code>	102,		2448, 2946, <u>4316</u> , 4316, 4321, 4341,
	4824, 4936, 4955, 4972, <u>5005</u> , 15356		4343, 4629, 4630, 5050, 5051, 5052,
<code>\tl_if_head_is_N_type_p:n</code>	102, <u>5005</u>		5053, 5069, 5070, 5549, 5551, 5962,
<code>\tl_if_head_is_space:n</code>	5026		6454, 6477, 6478, 7060, 7311, 7598,
<code>\tl_if_head_is_space:nTF</code>	102, <u>5026</u>		7599, 8052, 8053, 8054, 8055, 8164,
<code>\tl_if_head_is_space_p:n</code>	102, <u>5026</u>		8166, 8167, 8170, 8171, 8175, 8176,
<code>\tl_if_in:cnTF</code>	<u>4631</u>		8822, 8842, 8843, 8992, 9105, 9176,
<code>\tl_if_in:nn</code>	4637		9177, 9178, 9179, 9180, 14398, 15175
<code>\tl_if_in:NnF</code>	4632, 4635	<code>\tl_put_left:cn</code>	<u>4401</u>
<code>\tl_if_in:nnF</code>	4632, 4644	<code>\tl_put_left:co</code>	<u>4401</u>
<code>\tl_if_in:NnT</code>	4631, 4634, 8859	<code>\tl_put_left:cV</code>	<u>4401</u>
<code>\tl_if_in:nnT</code>	4631, 4643	<code>\tl_put_left:cx</code>	<u>4401</u>
<code>\tl_if_in:NnTF</code> ..	95, 2451, <u>4631</u> , 4633, 4636	<code>\tl_put_left:Nn</code>	92, <u>4401</u> , 4401, 4417
<code>\tl_if_in:nnTF</code>	95, 4633,	<code>\tl_put_left:No</code>	<u>4401</u> , 4405, 4419
	<u>4637</u> , 4645, 6947, 8213, 8223, 8926	<code>\tl_put_left:NV</code>	<u>4401</u> , 4403, 4418
<code>\tl_if_in:noTF</code>	<u>4637</u> , 15524	<code>\tl_put_left:Nx</code>	<u>4401</u> , 4407, 4420
<code>\tl_if_in:onTF</code>	<u>4637</u>	<code>\tl_put_right:cn</code>	<u>4425</u>
<code>\tl_if_in:VnTF</code>	<u>4637</u>	<code>\tl_put_right:co</code>	<u>4425</u>
<code>\tl_if_single:n</code>	4650	<code>\tl_put_right:cV</code>	<u>4425</u>
<code>\tl_if_single:Nf</code>	4648	<code>\tl_put_right:cx</code>	<u>4425</u>
<code>\tl_if_single:nF</code>	4648	<code>\tl_put_right:Nn</code> ..	92, <u>4425</u> , 4425, 4441, 8127
<code>\tl_if_single:NT</code>	4647	<code>\tl_put_right:No</code>	<u>4425</u> , 4429, 4443
<code>\tl_if_single:nT</code>	4647	<code>\tl_put_right:NV</code>	<u>4425</u> , 4427, 4442
<code>\tl_if_single:NfF</code>	95, <u>4646</u> , 4649	<code>\tl_put_right:Nx</code>	<u>4425</u> ,
<code>\tl_if_single:nTF</code>	95, 4649, <u>4650</u>		4431, 4444, 8093, 8118, 8131, 8140,
<code>\tl_if_single_p:N</code>	95, <u>4646</u> , 4646		9275, 9281, 9288, 9307, 9316, 9327
<code>\tl_if_single_p:n</code>	95, <u>4646</u> , <u>4650</u>	<code>\tl_remove_all:cn</code>	<u>4553</u>
<code>\tl_if_single_token:n</code>	15354	<code>\tl_remove_all:Nn</code> ..	93, <u>4553</u> , 4553, 4557, 8863
<code>\tl_if_single_token:nTF</code>	199, <u>15354</u>	<code>\tl_remove_once:cn</code>	<u>4547</u>
<code>\tl_if_single_token_p:n</code>	199, <u>15354</u>	<code>\tl_remove_once:Nn</code> ..	92, <u>4547</u> , 4547, 4551
<code>\tl_item:cn</code>	<u>15447</u>	<code>\tl_replace_all:cnn</code>	<u>4497</u>
<code>\tl_item:Nn</code>	<u>15447</u> , 15469, 15470	<code>\tl_replace_all:Nnn</code> ..	92, <u>4497</u> , 4501,
<code>\tl_item:nn</code>	200, <u>15447</u> , 15447, 15469		4507, 4554, 5116, 8069, 8070, 9233
<code>\tl_map_break:</code> ..	97, 4689, 4695, 4706,	<code>\tl_replace_once:cnn</code>	<u>4497</u>
	4715, 4722, <u>4727</u> , 4727, 4728, 4730	<code>\tl_replace_once:Nnn</code>	
<code>\tl_map_break:n</code>	97, <u>4727</u> , 4729		92, <u>4497</u> , 4497, 4505, 4548
<code>\tl_map_function:cN</code>	<u>4685</u>	<code>\tl_rescan:nn</code>	93, <u>4459</u> , 4463
<code>\tl_map_function:NN</code>		<code>\tl_reverse:c</code>	<u>4881</u>
	96, <u>4685</u> , 4691, 4698, 4748	<code>\tl_reverse:N</code>	99, <u>4881</u> , 4881, 4885
<code>\tl_map_function:nN</code>		<code>\tl_reverse:n</code>	
	96, <u>4685</u> , 4685, 4692, 4743, 5107		98, <u>4861</u> , 4861, 4874, 4882, 4884
<code>\tl_map_inline:cn</code>	<u>4699</u>	<code>\tl_reverse:o</code>	<u>4861</u>
<code>\tl_map_inline:Nn</code> ..	96, <u>4699</u> , 4708, 4710	<code>\tl_reverse:V</code>	<u>4861</u>
<code>\tl_map_inline:nn</code>	96, 2730,	<code>\tl_reverse_items:n</code>	99, <u>4753</u> , 4753
	<u>4699</u> , 4699, 4709, 9192, 10660, 10706	<code>\tl_reverse_tokens:n</code>	
<code>\tl_map_variable:cNn</code>	<u>4711</u>		199, <u>15360</u> , 15360, 15377
<code>\tl_map_variable:NNn</code> ..	96, <u>4711</u> , 4717, 4726	<code>.tl_set:c</code>	154
<code>\tl_map_variable:nNn</code> ..	96, <u>4711</u> , 4711, 4718	<code>\tl_set:cf</code>	<u>4383</u>
<code>\tl_new:c</code>	<u>4316</u> , 5552, 8768	<code>\tl_set:cn</code>	<u>4383</u> , 8771

<code>\tl_set:co</code>	4383	<code>\tl_tail:N</code>	101 , 4887 , 4916
<code>\tl_set:cV</code>	4383	<code>\tl_tail:n</code>	4887 , 4908 , 4915 , 4916
<code>\tl_set:cv</code>	4383	<code>\tl_tail:V</code>	4887
<code>\tl_set:cx</code>	4383	<code>\tl_tail:v</code>	4887
<code>.tl_set:N</code>	154	<code>\tl_to_lowercase:n</code>	
<code>\tl_set:Nf</code>	4383 , 5136 , 8024		93 , 2313 , 2328 , 2690 , 2732 , 2889 ,
<code>\tl_set:Nn</code>			4493 , 4493 , 7435 , 7947 , 8061 , 9185 ,
	92 , 2964 , 2985 , 3528 , 4383 , 4383 ,		10013 , 10099 , 13891 , 15481 , 15498
	4395 , 4397 , 4460 , 4619 , 4620 , 4721 ,	<code>\tl_to_str:c</code>	4732
	5106 , 5110 , 5216 , 5225 , 5247 , 5250 ,	<code>\tl_to_str:N</code>	97 , 4732 ,
	5271 , 5279 , 5298 , 5307 , 5321 , 5369 ,		4732 , 4733 , 8858 , 9232 , 9245 , 9246
	5439 , 5648 , 5654 , 5663 , 5670 , 5871 ,	<code>\tl_to_str:n</code>	98 , 793 , 3119 , 4015 ,
	6009 , 6025 , 6026 , 6034 , 6035 , 6037 ,		4136 , 4222 , 4514 , 4583 , 4596 , 4731 ,
	6043 , 6046 , 6056 , 6057 , 6066 , 6082 ,		4731 , 4897 , 4920 , 4929 , 5049 , 5996 ,
	6111 , 6173 , 6461 , 6465 , 6652 , 6948 ,		6084 , 6113 , 6141 , 6142 , 7090 , 7173 ,
	6949 , 7062 , 7065 , 7609 , 7691 , 8068 ,		7498 , 7499 , 7742 , 7743 , 8031 , 8035 ,
	8184 , 8217 , 8228 , 8298 , 8357 , 8506 ,		8036 , 8040 , 8041 , 8182 , 8222 , 8504 ,
	8645 , 8798 , 8878 , 8883 , 9255 , 14374		8561 , 8691 , 8707 , 8928 , 8969 , 8982 ,
<code>\tl_set:No</code>	4383 , 4385		9190 , 9336 , 10930 , 14018 , 14019 ,
<code>\tl_set:NV</code>	4383		14038 , 14041 , 15201 , 15202 , 15524
<code>\tl_set:Nv</code>	4383	<code>\tl_to_uppercase:n</code>	94 , 4493 , 4495
<code>\tl_set:Nx</code>	4355 , 4383 ,	<code>\tl_trim_spaces:c</code>	4769
	4387 , 4396 , 4498 , 4502 , 4772 , 4882 ,	<code>\tl_trim_spaces:N</code> ..	99 , 4769 , 4771 , 4775
	5099 , 5121 , 5147 , 5168 , 5206 , 5288 ,	<code>\tl_trim_spaces:n</code>	99 , 4769 ,
	5331 , 5345 , 5373 , 5570 , 5614 , 5657 ,		4769 , 4772 , 4774 , 5128 , 8115 , 8139
	5695 , 5747 , 6078 , 6106 , 8111 , 8115 ,	<code>\tl_use:c</code>	4734
	8124 , 8139 , 8182 , 8212 , 8222 , 8225 ,	<code>\tl_use:N</code>	98 , 4734 , 4734 , 4739
	8504 , 8514 , 8528 , 8561 , 8562 , 8856 ,	<code>\token_get_arg_spec:N</code> ...	61 , 3117 , 3130
	8857 , 8898 , 9035 , 9128 , 9226 , 9231 ,	<code>\token_get_prefix_spec:N</code> .	61 , 3117 , 3121
	9232 , 9295 , 9322 , 14129 , 14196 ,	<code>\token_get_replacement_spec:N</code>	
	14201 , 14263 , 14282 , 14871 , 15178 ,		61 , 3117 , 3139
	15262 , 15267 , 15288 , 15308 , 15318	<code>\token_if_active:N</code>	2664
<code>\tl_set_eq:cc</code> 4346 , 4349 , 5093 , 5564 , 5985		<code>\token_if_active:NTF</code>	55 , 2664
<code>\tl_set_eq:cN</code> 4346 , 4347 , 5092 , 5563 , 5984		<code>\token_if_active_p:N</code>	55 , 2664
<code>\tl_set_eq:Nc</code> 4346 , 4348 , 5091 , 5562 , 5983		<code>\token_if_alignment:N</code>	2626
<code>\tl_set_eq:NN</code> 91 , 4335 , 4346 , 4346 , 5090 ,		<code>\token_if_alignment:NTF</code>	54 , 2626
	5561 , 5982 , 7654 , 7662 , 14137 , 14232	<code>\token_if_alignment_p:N</code>	54 , 2626
<code>\tl_set_rescan:cn</code>	4459	<code>\token_if_chardef:N</code>	2735
<code>\tl_set_rescan:cno</code>	4459	<code>\token_if_chardef:NTF</code> ...	56 , 2724 , 2774
<code>\tl_set_rescan:cnx</code>	4459	<code>\token_if_chardef_p:N</code>	56 , 2724
<code>\tl_set_rescan:Nnn</code>		<code>\token_if_cs:N</code>	2707
	93 , 4459 , 4459 , 4489 , 4490	<code>\token_if_cs:NTF</code>	56 , 2707
<code>\tl_set_rescan:Nno</code>	4459	<code>\token_if_cs_p:N</code>	56 , 2707
<code>\tl_set_rescan:Nnx</code>	4459	<code>\token_if_dim_register:N</code>	2754
<code>.tl_set_x:c</code>	154	<code>\token_if_dim_register:NTF</code>	57 , 2724
<code>.tl_set_x:N</code>	154	<code>\token_if_dim_register_p:N</code>	57 , 2724
<code>\tl_show:c</code>	5038	<code>\token_if_eq_catcode:NN</code>	2674
<code>\tl_show:N</code>	102 , 5038 , 5038 , 5047	<code>\token_if_eq_catcode:NNTF</code>	55 , 2674
<code>\tl_show:n</code>	102 , 5048 , 5048	<code>\token_if_eq_catcode_p:NN</code>	55 , 2674
<code>\tl_tail:f</code>	4887	<code>\token_if_eq_charcode:NN</code>	2679

<code>\token_if_eq_charcode:NNTF</code>	55 , 2679	<code>\token_if_parameter_p:N</code>	55 , 2631
<code>\token_if_eq_charcode_p:NN</code>	55 , 2679	<code>\token_if_primitive:N</code>	2891
<code>\token_if_eq_meaning:NN</code>	2669	<code>\token_if_primitive:NTF</code>	57 , 2883
<code>\token_if_eq_meaning:NNTF</code>	9772	<code>\token_if_primitive_p:N</code>	57 , 2883
<code>\token_if_eq_meaning:NNT</code>	2323	<code>\token_if_protected_long_macro:N</code> .	2871
<code>\token_if_eq_meaning:NNTF</code>	56 , 2338 , 2669 , 10616 , 13186	<code>\token_if_protected_long_macro:NTF</code> .	56 , 2724
<code>\token_if_eq_meaning_p:NN</code>	56 , 2669	<code>\token_if_protected_long_macro_p:N</code> .	56 , 2724
<code>\token_if_expandable:N</code>	2712	<code>\token_if_protected_macro:N</code>	2850
<code>\token_if_expandable:NTF</code>	56 , 2712	<code>\token_if_protected_macro:NTF</code> .	56 , 2724
<code>\token_if_expandable_p:N</code>	56 , 2712	<code>\token_if_protected_macro_p:N</code> .	56 , 2724
<code>\token_if_group_begin:N</code>	2611	<code>\token_if_skip_register:N</code>	2814
<code>\token_if_group_begin:NTF</code>	54 , 2611	<code>\token_if_skip_register:NTF</code> . . .	57 , 2724
<code>\token_if_group_begin_p:N</code>	54 , 2611	<code>\token_if_skip_register_p:N</code> . . .	57 , 2724
<code>\token_if_group_end:N</code>	2616	<code>\token_if_space:N</code>	2649
<code>\token_if_group_end:NTF</code>	54 , 2616	<code>\token_if_space:NTF</code>	55 , 2649
<code>\token_if_group_end_p:N</code>	54 , 2616	<code>\token_if_space_p:N</code>	55 , 2649
<code>\token_if_int_register:N</code>	2772	<code>\token_if_toks_register:N</code>	2832
<code>\token_if_int_register:NTF</code>	57 , 2724	<code>\token_if_toks_register:NTF</code> . . .	57 , 2724
<code>\token_if_int_register_p:N</code>	57 , 2724	<code>\token_if_toks_register_p:N</code> . . .	57 , 2724
<code>\token_if_letter:N</code>	2654	<code>\token_new:Nn</code> 53 , 2571 , 2571 , 2576 , 2578 , 2579 , 2580 , 2582 , 2583 , 2584 , 2585	
<code>\token_if_letter:NTF</code>	55 , 2654	<code>\token_to_meaning:c</code>	784 , 795 , 2571
<code>\token_if_letter_p:N</code>	55 , 2654	<code>\token_to_meaning:N</code>	54 , 771 , 771 , 1182 , 1192 , 1875 , 2571 , 2696 , 2740 , 2749 , 2765 , 2787 , 2807 , 2825 , 2843 , 2856 , 2867 , 2877 , 2897 , 3126 , 3135 , 3144 , 15508
<code>\token_if_long_macro:N</code>	2862	<code>\token_to_str:c</code> . . . 784 , 784 , 914 , 923 , 944 , 964 , 1002 , 1007 , 1022 , 1894 , 2571	
<code>\token_if_long_macro:NTF</code>	56 , 2724	<code>\token_to_str:N</code>	5 , 54 , 771 , 772 , 784 , 1054 , 1055 , 1182 , 1192 , 1194 , 1205 , 1326 , 1356 , 1438 , 1453 , 1955 , 1970 , 2003 , 2109 , 2334 , 2454 , 2571 , 2742 , 2751 , 2767 , 2789 , 2809 , 2827 , 2845 , 2858 , 2869 , 2879 , 5017 , 5044 , 6357 , 6503 , 6651 , 7265 , 7269 , 7918 , 7925 , 7932 , 8015 , 8855 , 9215 , 9216 , 9217 , 9218 , 9219 , 9650 , 9651 , 10019 , 10027 , 10028 , 10054 , 10226 , 10246 , 10261 , 10273 , 10274 , 10287 , 10288 , 10313 , 10322 , 10324 , 10349 , 10352 , 10377 , 10379 , 10393 , 10409 , 10427 , 10496 , 10506 , 10507 , 10522 , 10523 , 13254 , 14167
<code>\token_if_long_macro_p:N</code>	56 , 2724	<code>\toks</code>	614
<code>\token_if_macro:N</code>	2693	<code>\toksdef</code>	315
<code>\token_if_macro:NTF</code>	56 , 2684 , 2893 , 3123 , 3132 , 3141	<code>\tolerance</code>	525
<code>\token_if_macro_p:N</code>	56 , 2684	<code>\topmark</code>	406
<code>\token_if_math_subscript:N</code>	2644		
<code>\token_if_math_subscript:NTF</code> . . .	55 , 2644		
<code>\token_if_math_subscript_p:N</code> . . .	55 , 2644		
<code>\token_if_math_superscript:N</code>	2639		
<code>\token_if_math_superscript:NTF</code> . . .	55 , 2639		
<code>\token_if_math_superscript_p:N</code> . . .	55 , 2639		
<code>\token_if_math_toggle:N</code>	2621		
<code>\token_if_math_toggle:NTF</code>	54 , 2621		
<code>\token_if_math_toggle_p:N</code>	54 , 2621		
<code>\token_if_mathchardef:N</code>	2744		
<code>\token_if_mathchardef:NTF</code>	57 , 2724 , 2776		
<code>\token_if_mathchardef_p:N</code>	57 , 2724		
<code>\token_if_muskip_register:N</code>	2796		
<code>\token_if_muskip_register:NTF</code> . . .	57 , 2724		
<code>\token_if_muskip_register_p:N</code> . . .	57 , 2724		
<code>\token_if_other:N</code>	2659		
<code>\token_if_other:NTF</code>	55 , 2659		
<code>\token_if_other_p:N</code>	55 , 2659		
<code>\token_if_parameter:N</code>	2633		
<code>\token_if_parameter:NTF</code>	55 , 2631		

<code>\topmarks</code>	632	<code>\use_i:nn</code>	19,
<code>\topskip</code>	536		788, <u>852</u> , 852, 878, 954, 1098, 1126,
<code>\tracingassigns</code>	642		1305, 1467, 1481, 1489, 2026, 2151,
<code>\tracingcommands</code>	381		2164, 2168, 2185, 2186, 4898, 6002,
<code>\tracinggroups</code>	649		9621, 10689, 10693, 10713, 11657,
<code>\tracingifs</code>	645		12173, 12688, 12857, 13518, 13732
<code>\tracinglostchars</code>	382	<code>\use_i:nnn</code>	19, <u>854</u> , 854, 1080, 3126, 11615
<code>\tracingmacros</code>	383	<code>\use_i:nnnn</code>	19, <u>854</u> , 858, 11633, 11640, 11830
<code>\tracingnesting</code>	644	<code>\use_i_delimit_by_q_nil:nw</code> ..	20, <u>865</u> , 865
<code>\tracingonline</code>	384	<code>\use_i_delimit_by_q_recursion_stop:nw</code> 20, 47, 865, 867, 2367, 2383
<code>\tracingoutput</code>	385	<code>\use_i_delimit_by_q_stop:nw</code> 20, <u>865</u> , 866, 14836
<code>\tracingpages</code>	386	<code>\use_i_ii:nnn</code>	19, <u>854</u> , 857, 1638
<code>\tracingparagraphs</code>	387	<code>\use_ii:nn</code>	19, 790, <u>852</u> , 853, 880,
<code>\tracingrestores</code>	388		956, 1100, 1128, 1303, 1464, 1470,
<code>\tracingscantokens</code>	643		1478, 1492, 1575, 2164, 4900, 6003,
<code>\tracingstats</code>	389		8083, 9595, 10691, 10695, 10715,
			11659, 12690, 13520, 13734, 14104
U			
<code>\U</code>	10098, 15494	<code>\use_ii:nnn</code> 19, <u>854</u> , 855, 1082, 3135, 8111, 8132
<code>\uccode</code>	624	<code>\use_ii:nnnn</code>	19, <u>854</u> , 859
<code>\uchyph</code>	522	<code>\use_iii:nnn</code> 19, <u>854</u> , 856, 1580, 3144, 9604, 9611
<code>\underline</code>	458	<code>\use_iii:nnnn</code>	19, <u>854</u> , 860
<code>\unexpanded</code>	186, 190, 637	<code>\use_iv:nnnn</code>	19, <u>854</u> , 861
<code>\unhbox</code>	563	<code>\use_none:n</code>	19, <u>868</u> , 868,
<code>\unhcopy</code>	564		970, 1275, 1462, 1466, 1468, 1476,
<code>\unkern</code>	488		1480, 1488, 1490, 2369, 2385, 2931,
<code>\unless</code>	628		3628, 3743, 3747, 3752, 4560, 4812,
<code>\unpenalty</code>	599		4897, 4913, 4992, 5014, 5033, 5036,
<code>\unskip</code>	486		5067, 5320, 5344, 5350, 5401, 5675,
<code>\unvbox</code>	565		5768, 7453, 7804, 7808, 8080, 9560,
<code>\unvcopy</code>	566		9564, 9568, 9572, 9632, 9660, 10626,
<code>\uppercase</code>	596		10835, 10847, 10858, 10873, 10877,
<code>\use:c</code>	17, <u>842</u> , 842, 939,		10899, 10924, 10993, 11009, 11051,
	1017, 1148, 1150, 1152, 1154, 2155,		11634, 11637, 12360, 13254, 14112,
	2174, 3352, 3697, 3707, 3850, 3859,		14881, 15231, 15253, 15254, 15357
	3861, 3863, 3864, 3868, 4019, 7531,	<code>\use_none:nn</code> ..	<u>868</u> , 869, 916, 924, 1455,
	7542, 7557, 7566, 7574, 7582, 7588,		4651, 4768, 4939, 4958, 5221, 5326,
	7614, 7996, 8236, 8243, 8375, 9300		9509, 9559, 9563, 9567, 9571, 14910
<code>\use:n</code>	18, <u>848</u> , 848, 970,	<code>\use_none:nnn</code>	<u>868</u> , 870, 4980,
	1273, 1435, 1463, 1465, 1469, 1477,		8110, 8125, 9558, 9562, 9566, 9570
	1479, 1487, 1491, 2592, 4464, 4723,	<code>\use_none:nnnn</code> ..	<u>868</u> , 871, 1956, 1971, 3498
	4989, 5008, 5873, 6354, 7959, 9695,	<code>\use_none:nnnnn</code> <u>868</u> , 872, 9690, 9724, 9750, 9758
	9703, 9712, 9729, 9737, 9765, 15195	<code>\use_none:nnnnnn</code>	<u>868</u> , 873, 1024
<code>\use:nn</code>	<u>848</u> , 849, 1613,	<code>\use_none:nnnnnnn</code>	<u>868</u> , 874, 946,
	2601, 3117, 4013, 5857, 10104, 12862		9692, 9726, 9752, 9760, 9961, 11673
<code>\use:nnn</code>	<u>848</u> , 850, 1323	<code>\use_none:nnnnnnnn</code>	<u>868</u> , 875
<code>\use:nnnn</code>	<u>848</u> , 851		
<code>\use:x</code>	20, <u>843</u> , 843, 926, 988,		
	1904, 4133, 4472, 4483, 6337, 7495,		
	7512, 7739, 7756, 8865, 9090, 9241		

<code>\use_none:nnnnnnnnnn</code>	868, 876	<code>\vbox_to_ht:nn</code>	134, 6399, 6399, 6399
<code>\use_none_delimit_by_q_nil:w</code> 20, 862, 862		<code>\vbox_to_zero:n</code>	134, 6399, 6399, 6401
<code>\use_none_delimit_by_q_recursion_stop:w</code> 20, 47, 862, 864, 937, 1003, 1008, 1015, 1895, 1902, 2361, 2376		<code>\vbox_top:n</code>	133, 6397, 6398
<code>\use_none_delimit_by_q_stop:w</code> 20, 862, 863, 3363, 4023, 4522, 5755, 7632, 9329, 9343, 14823, 14828, 15510		<code>\vbox_unpack:c</code>	6438
<code>\usepackage</code>	149, 230	<code>\vbox_unpack:N</code>	135, 6438, 6438, 6440, 6561, 6608
V			
<code>\vadjust</code>	499	<code>\vbox_unpack_clear:c</code>	6438
<code>\valign</code>	334	<code>\vbox_unpack_clear:N</code> 135, 6438, 6439, 6441	
<code>.value_forbidden:</code>	154	<code>\vcenter</code>	420
<code>.value_required:</code>	154	<code>\vcoffin_set:cnn</code>	6543
<code>\vbadness</code>	574	<code>\vcoffin_set:cnw</code>	6590
<code>\vbox</code>	569	<code>\vcoffin_set:Nnn</code> 137, 6543, 6543, 6572	
<code>\vbox:n</code>	133, 6397, 6397	<code>\vcoffin_set:Nnw</code> 137, 6590, 6590, 6623	
<code>\vbox_gset:cn</code>	6403	<code>\vcoffin_set_end:</code> 137, 6590, 6601, 6622	
<code>\vbox_gset:cw</code>	6420, 6436	<code>\vfil</code>	481
<code>\vbox_gset:Nn</code> 134, 6403, 6405, 6407		<code>\vfill</code>	483
<code>\vbox_gset:Nw</code> 134, 6420, 6422, 6425, 6435		<code>\vfilneg</code>	482
<code>\vbox_gset_end:</code> 134, 6420, 6431, 6437		<code>\vfuzz</code>	576
<code>\vbox_gset_inline_begin:c</code> 6432, 6436		<code>\voffset</code>	551
<code>\vbox_gset_inline_begin:N</code> 6432, 6435		<code>\vrule</code>	490
<code>\vbox_gset_inline_end:</code> 6432, 6437		<code>\vsize</code>	533
<code>\vbox_gset_to_ht:cnn</code>	6414	<code>\vskip</code>	484
<code>\vbox_gset_to_ht:Nnn</code> 134, 6414, 6416, 6419		<code>\vsplit</code>	562
<code>\vbox_gset_top:cn</code>	6408	<code>\vss</code>	485
<code>\vbox_gset_top:Nn</code> 134, 6408, 6410, 6413		<code>\vtop</code>	570
<code>\vbox_set:cn</code>	6403	W	
<code>\vbox_set:cw</code>	6420, 6433	<code>\wd</code>	617
<code>\vbox_set:Nn</code>		<code>\widowpenalties</code>	677
. 134, 6403, 6403, 6405, 6406, 6547		<code>\widowpenalty</code>	504
<code>\vbox_set:Nw</code>		<code>TWOBARS</code>	180
134, 6420, 6420, 6423, 6424, 6432, 6594		<code>\write</code>	367
<code>\vbox_set_end:</code>		X	
. 134, 6420, 6426, 6431, 6434, 6604		<code>\X</code>	2010
<code>\vbox_set_inline_begin:c</code> 6432, 6433		<code>ex</code>	184
<code>\vbox_set_inline_begin:N</code> 6432, 6432		<code>\xdef</code>	308
<code>\vbox_set_inline_end:</code> 6432, 6434		<code>\xetex_if_engine:F</code> 1469, 1480	
<code>\vbox_set_split_to_ht:NNn</code> 134, 6442, 6442		<code>\xetex_if_engine:T</code> 1468, 1479	
<code>\vbox_set_to_ht:cnn</code>	6414	<code>\xetex_if_engine:TF</code> 5, 23, 1462, 1470, 1481	
<code>\vbox_set_to_ht:Nnn</code>		<code>\xetex_if_engine_p:</code> 23, 1462, 1473, 1483	
. 134, 6414, 6414, 6417, 6418		<code>\xetex_XeTeXversion:D</code> 712, 1474	
<code>\vbox_set_top:cn</code>	6408	<code>\XeTeXversion</code>	712
<code>\vbox_set_top:Nn</code>		<code>\xleaders</code>	493
134, 6408, 6408, 6411, 6412, 6561, 6608		<code>exp</code>	181
Y		<code>\xspaceskip</code>	527
<code>\year</code>	608		