

The L^AT_EX3 Sources

The L^AT_EX3 Project*

July 20, 2014

Abstract

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

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

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

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

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

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

Contents

I	Introduction to <code>expl3</code> and this document	1
1	Naming functions and variables	1
1.1	Terminological inexactitude	3
2	Documentation conventions	3
3	Formal language conventions which apply generally	5
4	<code>TeX</code> concepts not supported by <code>LaTeX3</code>	6
II	The <code>l3bootstrap</code> package: Bootstrap code	7
1	Using the <code>LaTeX3</code> modules	7
1.1	Internal functions and variables	8
III	The <code>l3names</code> package: Namespace for primitives	9
1	Setting up the <code>LaTeX3</code> programming language	9
IV	The <code>l3basics</code> package: Basic definitions	10
1	No operation functions	10
2	Grouping material	10
3	Control sequences and functions	11
3.1	Defining functions	11
3.2	Defining new functions using parameter text	12
3.3	Defining new functions using the signature	14
3.4	Copying control sequences	16
3.5	Deleting control sequences	17
3.6	Showing control sequences	17
3.7	Converting to and from control sequences	18
4	Using or removing tokens and arguments	19
4.1	Selecting tokens from delimited arguments	21
5	Predicates and conditionals	21
5.1	Tests on control sequences	22
5.2	Engine-specific conditionals	23
5.3	Primitive conditionals	23

6	Internal kernel functions	24
V	The l3expan package: Argument expansion	26
1	Defining new variants	26
2	Methods for defining variants	27
3	Introducing the variants	27
4	Manipulating the first argument	28
5	Manipulating two arguments	29
6	Manipulating three arguments	30
7	Unbraced expansion	31
8	Preventing expansion	31
9	Internal functions and variables	33
VI	The l3prg package: Control structures	34
1	Defining a set of conditional functions	34
2	The boolean data type	36
3	Boolean expressions	38
4	Logical loops	39
5	Producing n copies	40
6	Detecting \TeX 's mode	40
7	Primitive conditionals	41
8	Internal programming functions	41
VII	The l3quark package: Quarks	43
1	Introduction to quarks and scan marks	43
1.1	Quarks	43
2	Defining quarks	44

3	Quark tests	44
4	Recursion	45
5	An example of recursion with quarks	46
6	Internal quark functions	46
7	Scan marks	47
 VIII The l3token package: Token manipulation		48
1	All possible tokens	48
2	Character tokens	49
3	Generic tokens	52
4	Converting tokens	53
5	Token conditionals	53
6	Peeking ahead at the next token	57
7	Decomposing a macro definition	60
 IX The l3int package: Integers		61
1	Integer expressions	61
2	Creating and initialising integers	62
3	Setting and incrementing integers	63
4	Using integers	64
5	Integer expression conditionals	64
6	Integer expression loops	66
7	Integer step functions	68
8	Formatting integers	68
9	Converting from other formats to integers	70
10	Viewing integers	71

11	Constant integers	72
12	Scratch integers	72
13	Primitive conditionals	73
14	Internal functions	73
X	The <code>l3skip</code> package: Dimensions and skips	75
1	Creating and initialising <code>dim</code> variables	75
2	Setting <code>dim</code> variables	76
3	Utilities for dimension calculations	76
4	Dimension expression conditionals	77
5	Dimension expression loops	79
6	Using <code>dim</code> expressions and variables	80
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 <code>muskip</code> variables	86
19	Setting <code>muskip</code> 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	98
9	Working with the content of token lists	98
10	The first token from a token list	100
11	Using a single item	102
12	Viewing token lists	102
13	Constant token lists	103
14	Scratch token lists	103
15	Internal functions	103
XII	The l3str package:Strings	104

1	The first character from a string	104
1.1	Tests on strings	104
2	String manipulation	106
2.1	Internal string functions	106
XIII	The l3seq package: Sequences and stacks	107
1	Creating and initialising sequences	107
2	Appending data to sequences	108
3	Recovering items from sequences	108
4	Recovering values from sequences with branching	110
5	Modifying sequences	111
6	Sequence conditionals	112
7	Mapping to sequences	112
8	Using the content of sequences directly	114
9	Sequences as stacks	114
10	Constant and scratch sequences	116
11	Viewing sequences	116
12	Internal sequence functions	116
XIV	The l3clist package: Comma separated lists	117
1	Creating and initialising comma lists	117
2	Adding data to comma lists	118
3	Modifying comma lists	119
4	Comma list conditionals	120
5	Mapping to comma lists	121
6	Using the content of comma lists directly	123
7	Comma lists as stacks	123

8	Using a single item	125
9	Viewing comma lists	125
10	Constant and scratch comma lists	125
XV The <code>l3prop</code> package: Property lists		127
1	Creating and initialising property lists	127
2	Adding entries to property lists	128
3	Recovering values from property lists	128
4	Modifying property lists	129
5	Property list conditionals	129
6	Recovering values from property lists with branching	130
7	Mapping to property lists	130
8	Viewing property lists	131
9	Scratch property lists	132
10	Constants	132
11	Internal property list functions	132
XVI The <code>l3box</code> package: Boxes		133
1	Creating and initialising boxes	133
2	Using boxes	134
3	Measuring and setting box dimensions	134
4	Box conditionals	135
5	The last box inserted	136
6	Constant boxes	136
7	Scratch boxes	136
8	Viewing box contents	136

9	Horizontal mode boxes	137
10	Vertical mode boxes	138
11	Primitive box conditionals	140
XVII The l3coffins package: Coffin code layer		141
1	Creating and initialising coffins	141
2	Setting coffin content and poles	141
3	Joining and using coffins	143
4	Measuring coffins	143
5	Coffin diagnostics	144
5.1	Constants and variables	144
XVIII The l3color package: Color support		145
1	Color in boxes	145
XIX The l3msg package: Messages		146
1	Creating new messages	146
2	Contextual information for messages	147
3	Issuing messages	148
4	Redirecting messages	150
5	Low-level message functions	151
6	Kernel-specific functions	152
7	Expandable errors	153
8	Internal l3msg functions	154
XX The l3keys package: Key–value interfaces		156
1	Creating keys	157

2	Sub-dividing keys	161
3	Choice and multiple choice keys	161
4	Setting keys	163
5	Handling of unknown keys	164
6	Selective key setting	165
7	Utility functions for keys	166
8	Low-level interface for parsing key-val lists	166
XXI	The l3file package: File and I/O operations	168
1	File operation functions	168
	1.1 Input-output stream management	169
	1.2 Reading from files	170
2	Writing to files	171
	2.1 Wrapping lines in output	173
	2.2 Constant input-output streams	174
	2.3 Primitive conditionals	174
	2.4 Internal file functions and variables	174
	2.5 Internal input-output functions	175
XXII	The l3fp package: floating points	176
1	Creating and initialising floating point variables	177
2	Setting floating point variables	177
3	Using floating point numbers	178
4	Floating point conditionals	179
5	Floating point expression loops	181
6	Some useful constants, and scratch variables	182
7	Floating point exceptions	183
8	Viewing floating points	184

9	Floating point expressions	184
9.1	Input of floating point numbers	184
9.2	Precedence of operators	185
9.3	Operations	186
10	Disclaimer and roadmap	191
XXIII	The l3candidates package: Experimental additions to l3kernel	194
1	Important notice	194
2	Additions to l3box	194
2.1	Affine transformations	194
2.2	Viewing part of a box	196
2.3	Internal variables	197
3	Additions to l3coffins	197
4	Additions to l3file	198
5	Additions to l3prop	200
6	Additions to l3seq	200
7	Additions to l3skip	201
8	Additions to l3tl	201
9	Additions to l3tokens	204
XXIV	The l3drivers package: Drivers	205
1	Box clipping	205
2	Box rotation and scaling	206
3	Color support	206
XXV	Implementation	206

1	l3bootstrap implementation	206
1.1	Format-specific code	207
1.2	The <code>\pdfstrcmp</code> primitive with X _Y TeX and LuaTeX	207
1.3	Engine requirements	208
1.4	Extending allocators	210
1.5	The L ^A T _E X3 code environment	210
2	l3names implementation	212
3	l3basics implementation	223
3.1	Renaming some TeX primitives (again)	223
3.2	Defining some constants	225
3.3	Defining functions	226
3.4	Selecting tokens	227
3.5	Gobbling tokens from input	228
3.6	Conditional processing and definitions	228
3.7	Dissecting a control sequence	234
3.8	Exist or free	236
3.9	Defining and checking (new) functions	238
3.10	More new definitions	240
3.11	Copying definitions	241
3.12	Undefining functions	242
3.13	Generating parameter text from argument count	242
3.14	Defining functions from a given number of arguments	243
3.15	Using the signature to define functions	244
3.16	Checking control sequence equality	246
3.17	Diagnostic functions	247
3.18	Engine specific definitions	248
3.19	Doing nothing functions	248
3.20	Breaking out of mapping functions	249
4	l3expan implementation	249
4.1	General expansion	249
4.2	Hand-tuned definitions	253
4.3	Definitions with the automated technique	255
4.4	Last-unbraced versions	256
4.5	Preventing expansion	258
4.6	Defining function variants	258

5	l3prg implementation	265
5.1	Primitive conditionals	265
5.2	Defining a set of conditional functions	265
5.3	The boolean data type	265
5.4	Boolean expressions	268
5.5	Logical loops	274
5.6	Producing n copies	275
5.7	Detecting T _E X's mode	276
5.8	Internal programming functions	277
6	l3quark implementation	279
6.1	Quarks	279
6.2	Scan marks	282
6.3	Deprecated quark functions	283
7	l3token implementation	283
7.1	Character tokens	283
7.2	Generic tokens	286
7.3	Token conditionals	287
7.4	Peeking ahead at the next token	297
7.5	Decomposing a macro definition	302
8	l3int implementation	303
8.1	Integer expressions	303
8.2	Creating and initialising integers	305
8.3	Setting and incrementing integers	307
8.4	Using integers	308
8.5	Integer expression conditionals	308
8.6	Integer expression loops	312
8.7	Integer step functions	313
8.8	Formatting integers	315
8.9	Converting from other formats to integers	321
8.10	Viewing integer	325
8.11	Constant integers	325
8.12	Scratch integers	326
8.13	Deprecated functions	326

9	l3skip implementation	326
9.1	Length primitives renamed	326
9.2	Creating and initialising <code>dim</code> variables	327
9.3	Setting <code>dim</code> variables	328
9.4	Utilities for dimension calculations	328
9.5	Dimension expression conditionals	329
9.6	Dimension expression loops	331
9.7	Using <code>dim</code> expressions and variables	332
9.8	Viewing <code>dim</code> variables	334
9.9	Constant dimensions	334
9.10	Scratch dimensions	334
9.11	Creating and initialising <code>skip</code> variables	334
9.12	Setting <code>skip</code> variables	335
9.13	Skip expression conditionals	336
9.14	Using <code>skip</code> expressions and variables	337
9.15	Inserting skips into the output	337
9.16	Viewing <code>skip</code> variables	337
9.17	Constant skips	337
9.18	Scratch skips	338
9.19	Creating and initialising <code>muskip</code> variables	338
9.20	Setting <code>muskip</code> variables	339
9.21	Using <code>muskip</code> expressions and variables	339
9.22	Viewing <code>muskip</code> variables	340
9.23	Constant muskips	340
9.24	Scratch muskips	340
9.25	Deprecated functions	340
10	l3tl implementation	341
10.1	Functions	341
10.2	Constant token lists	342
10.3	Adding to token list variables	343
10.4	Reassigning token list category codes	346
10.5	Reassigning token list character codes	347
10.6	Modifying token list variables	347
10.7	Token list conditionals	349
10.8	Mapping to token lists	354
10.9	Using token lists	355
10.10	Working with the contents of token lists	356
10.11	Token by token changes	358
10.12	The first token from a token list	360
10.13	Using a single item	365
10.14	Viewing token lists	366
10.15	Scratch token lists	366
10.16	Deprecated functions	366

11	l3str implementation	367
	11.1 String comparisons	367
	11.2 String manipulation	370
	11.3 Deprecated functions	371
12	l3seq implementation	371
	12.1 Allocation and initialisation	372
	12.2 Appending data to either end	375
	12.3 Modifying sequences	376
	12.4 Sequence conditionals	378
	12.5 Recovering data from sequences	379
	12.6 Mapping to sequences	383
	12.7 Using sequences	386
	12.8 Sequence stacks	386
	12.9 Viewing sequences	387
	12.10Scratch sequences	388
13	l3clist implementation	388
	13.1 Allocation and initialisation	388
	13.2 Removing spaces around items	390
	13.3 Adding data to comma lists	391
	13.4 Comma lists as stacks	392
	13.5 Modifying comma lists	394
	13.6 Comma list conditionals	397
	13.7 Mapping to comma lists	398
	13.8 Using comma lists	401
	13.9 Using a single item	402
	13.10Viewing comma lists	403
	13.11Scratch comma lists	404
14	l3prop implementation	404
	14.1 Allocation and initialisation	405
	14.2 Accessing data in property lists	406
	14.3 Property list conditionals	410
	14.4 Recovering values from property lists with branching	411
	14.5 Mapping to property lists	412
	14.6 Viewing property lists	413
	14.7 Deprecated functions	413

15	l3box implementation	414
15.1	Creating and initialising boxes	414
15.2	Measuring and setting box dimensions	415
15.3	Using boxes	415
15.4	Box conditionals	416
15.5	The last box inserted	416
15.6	Constant boxes	416
15.7	Scratch boxes	417
15.8	Viewing box contents	417
15.9	Horizontal mode boxes	418
15.10	Vertical mode boxes	419
16	l3coffins Implementation	421
16.1	Coffins: data structures and general variables	421
16.2	Basic coffin functions	423
16.3	Measuring coffins	427
16.4	Coffins: handle and pole management	427
16.5	Coffins: calculation of pole intersections	430
16.6	Aligning and typesetting of coffins	433
16.7	Coffin diagnostics	438
16.8	Messages	443
17	l3color Implementation	444
18	l3msg implementation	445
18.1	Creating messages	445
18.2	Messages: support functions and text	447
18.3	Showing messages: low level mechanism	448
18.4	Displaying messages	450
18.5	Kernel-specific functions	457
18.6	Expandable errors	462
18.7	Showing variables	464
19	l3keys Implementation	466
19.1	Low-level interface	466
19.2	Constants and variables	469
19.3	The key defining mechanism	470
19.4	Turning properties into actions	472
19.5	Creating key properties	476
19.6	Setting keys	480
19.7	Utilities	485
19.8	Messages	486
19.9	Deprecated functions	487

20	l3file implementation	488
20.1	File operations	488
20.2	Input operations	494
20.2.1	Variables and constants	494
20.2.2	Stream management	494
20.2.3	Reading input	496
20.3	Output operations	497
20.3.1	Variables and constants	498
20.4	Stream management	498
20.4.1	Deferred writing	499
20.4.2	Immediate writing	500
20.4.3	Special characters for writing	500
20.4.4	Hard-wrapping lines to a character count	501
20.5	Messages	506
21	l3fp implementation	507
22	l3fp-aux implementation	507
23	Internal representation	507
24	Internal storage of floating points numbers	508
24.1	Using arguments and semicolons	509
24.2	Constants, and structure of floating points	510
24.3	Overflow, underflow, and exact zero	512
24.4	Expanding after a floating point number	512
24.5	Packing digits	514
24.6	Decimate (dividing by a power of 10)	516
24.7	Functions for use within primitive conditional branches	518
24.8	Small integer floating points	519
24.9	Length of a floating point array	520
24.10	x-like expansion expandably	520
24.11	Messages	521
25	l3fp-traps Implementation	521
25.1	Flags	521
25.2	Traps	522
25.3	Errors	526
25.4	Messages	526
26	l3fp-round implementation	527
26.1	Rounding tools	527
26.2	The round function	530

27	l3fp-parse implementation	533
27.1	Work plan	533
27.1.1	Storing results	534
27.1.2	Precedence and infix operators	535
27.1.3	Prefix operators, parentheses, and functions	538
27.1.4	Numbers and reading tokens one by one	539
27.2	Main auxiliary functions	541
27.3	Helpers	541
27.4	Parsing one number	543
27.4.1	Numbers: trimming leading zeros	548
27.4.2	Number: small significand	549
27.4.3	Number: large significand	551
27.4.4	Number: beyond 16 digits, rounding	553
27.4.5	Number: finding the exponent	556
27.5	Constants, functions and prefix operators	559
27.5.1	Prefix operators	559
27.5.2	Constants	561
27.5.3	Functions	562
27.6	Main functions	564
27.7	Infix operators	565
27.7.1	Closing parentheses and commas	566
27.7.2	Usual infix operators	568
27.7.3	Juxtaposition	569
27.7.4	Multi-character cases	570
27.7.5	Ternary operator	571
27.7.6	Comparisons	571
27.8	Candidate: defining new l3fp functions	574
27.9	Messages	576
28	l3fp-logic Implementation	576
28.1	Syntax of internal functions	576
28.2	Existence test	576
28.3	Comparison	577
28.4	Floating point expression loops	579
28.5	Extrema	580
28.6	Boolean operations	582
28.7	Ternary operator	582

29	l3fp-basics Implementation	584
29.1	Common to several operations	584
29.2	Addition and subtraction	585
29.2.1	Sign, exponent, and special numbers	585
29.2.2	Absolute addition	587
29.2.3	Absolute subtraction	590
29.3	Multiplication	594
29.3.1	Signs, and special numbers	594
29.3.2	Absolute multiplication	596
29.4	Division	598
29.4.1	Signs, and special numbers	598
29.4.2	Work plan	599
29.4.3	Implementing the significand division	602
29.5	Square root	607
29.6	Setting the sign	615
30	l3fp-extended implementation	615
30.1	Description of fixed point numbers	615
30.2	Helpers for numbers with extended precision	616
30.3	Multiplying a fixed point number by a short one	617
30.4	Dividing a fixed point number by a small integer	618
30.5	Adding and subtracting fixed points	619
30.6	Multiplying fixed points	620
30.7	Combining product and sum of fixed points	621
30.8	Extended-precision floating point numbers	623
30.9	Dividing extended-precision numbers	626
30.10	Inverse square root of extended precision numbers	629
30.11	Converting from fixed point to floating point	631
31	l3fp-expo implementation	633
31.1	Logarithm	633
31.1.1	Work plan	633
31.1.2	Some constants	634
31.1.3	Sign, exponent, and special numbers	634
31.1.4	Absolute ln	634
31.2	Exponential	642
31.2.1	Sign, exponent, and special numbers	642
31.3	Power	647

32	l3fp-trig Implementation	654
32.1	Direct trigonometric functions	654
32.1.1	Filtering special cases	654
32.1.2	Distinguishing small and large arguments	657
32.1.3	Small arguments	658
32.1.4	Argument reduction in degrees	658
32.1.5	Argument reduction in radians	660
32.1.6	Computing the power series	666
32.2	Inverse trigonometric functions	669
32.2.1	Arctangent and arccotangent	670
32.2.2	Arcsine and arccosine	675
32.2.3	Arccosecant and arcsecant	677
33	l3fp-convert implementation	679
33.1	Trimming trailing zeros	679
33.2	Scientific notation	679
33.3	Decimal representation	681
33.4	Token list representation	682
33.5	Formatting	683
33.6	Convert to dimension or integer	683
33.7	Convert from a dimension	684
33.8	Use and eval	685
33.9	Convert an array of floating points to a comma list	686
34	l3fp-assign implementation	686
34.1	Assigning values	686
34.2	Updating values	687
34.3	Showing values	688
34.4	Some useful constants and scratch variables	688
35	l3candidates Implementation	689
35.1	Additions to l3box	689
35.2	Affine transformations	689
35.3	Viewing part of a box	697
35.4	Additions to l3coffins	699
35.5	Rotating coffins	699
35.6	Resizing coffins	704
35.7	Additions to l3file	707
35.8	Additions to l3prop	708
35.9	Additions to l3seq	709
35.10	Additions to l3skip	710
35.11	Additions to l3tl	711
35.11.1	Unicode case changing	715
35.12	Additions to l3tokens	722
35.13	Deprecated candidates	724

36	l3drivers Implementation	724
36.1	Settings for direct PDF output	725
36.2	Driver utility functions	725
36.3	Box clipping	728
36.4	Box rotation and scaling	729
36.5	Color support	730
	Index	732

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

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

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

2 Documentation conventions

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

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

`\ExplSyntaxOn`
`\ExplSyntaxOff`

`\ExplSyntaxOn ... \ExplSyntaxOff`

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

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

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

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

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

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

`\cs_to_str:N` ☆

`\cs_to_str:N` $\langle cs \rangle$

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

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

`\seq_map_function:NN` ☆

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

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

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

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

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

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

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

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

The normal description text.

\TeX hackers note: Detail for the experienced \TeX or $\text{\LaTeX} 2_\epsilon$ programmer. In this case, it would point out that this function is the \TeX primitive `\string`.

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

3 Formal language conventions which apply generally

As this is a formal reference guide for $\text{\LaTeX} 3$ programming, the descriptions of functions are intended to be reasonably “complete”. However, there is also a need to avoid repetition. Formal ideas which apply to general classes of function are therefore summarised here.

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

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

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

Part II

The l3bootstrap package

Bootstrap code

1 Using the L^AT_EX3 modules

The modules documented in `source3` are designed to be used on top of L^AT_EX 2_ε and are loaded all as one with the usual `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions. These modules will also form the basis of the L^AT_EX3 format, but work in this area is incomplete and not included in this documentation at present.

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

`\ExplSyntaxOn`
`\ExplSyntaxOff`
 Updated: 2011-08-13

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

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

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

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

These functions act broadly in the same way as the 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

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

Part III

The l3names package

Namespace for primitives

1 Setting up the L^AT_EX3 programming language

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

- defines new names for all T_EX primitives;
- switches to the category code régime for programming;
- provides support settings for building the code as a T_EX format.

This module is entirely dedicated to primitives, which should not be used directly within L^AT_EX3 code (outside of “kernel-level” code). As such, the primitives are not documented here: *The T_EXbook*, *T_EX by Topic* and the manuals for pdfT_EX, X_YT_EX and LuaT_EX should be consulted for details of the primitives. These are named based on the engine which first introduced them:

`\tex_...` Introduced by T_EX itself;

`\etex_...` Introduced by the ε -T_EX extensions;

`\pdftex_...` Introduced by pdfT_EX;

`\xetex_...` Introduced by X_YT_EX;

`\luatex_...` Introduced by LuaT_EX.

Part IV

The l3basics package

Basic definitions

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

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

1 No operation functions

`\prg_do_nothing: *`

`\prg_do_nothing:`

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

`\scan_stop:`

`\scan_stop:`

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

2 Grouping material

`\group_begin:`
`\group_end:`

`\group_begin:`
`\group_end:`

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

`\group_insert_after:N`

`\group_insert_after:N <token>`

Adds *<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 1).

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

The `\cs_new:` functions below (and friends) do not stop you from using other argument specifiers in your function names, but they do not handle expansion for you. You should define the base function and then use `\cs_generate_variant:Nn` to generate custom variants as described in Section 2.

3.2 Defining new functions using parameter text

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

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

Creates $\langle function \rangle$ 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 definition is global and an error will result if the $\langle function \rangle$ is already defined.

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

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

Creates $\langle function \rangle$ 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 definition is global and an error will result if the $\langle function \rangle$ is already defined.

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

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

Creates $\langle function \rangle$ 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 $\langle function \rangle$ will not expand within an x-type argument. The definition is global and an error will result if the $\langle function \rangle$ is already defined.

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

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

Creates $\langle function \rangle$ 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 $\langle function \rangle$ will not expand within an x-type argument. The definition is global and an error will result if the $\langle function \rangle$ is already defined.

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

```
\cs_set: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.

`\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.

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

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 $\backslash\text{par}$ tokens. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current T_EX group level: the assignment is global. The $\langle function \rangle$ will not expand within an x-type argument.

3.3 Defining new functions using the signature

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Updated: 2012-01-14

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

3.4 Copying control sequences

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

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

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

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

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

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

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

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

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

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

3.5 Deleting control sequences

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

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

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

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

Updated: 2011-09-15

3.6 Showing control sequences

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

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

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

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

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

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

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

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

Updated: 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` ★

New: 2012-11-10

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

`\cs_if_exist_use:NTF` ★

`\cs_if_exist_use:cTF` ★

New: 2012-11-10

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

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

`\cs:w` ★

`\cs_end:` ★

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

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

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

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

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

and

```

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

```

would be equivalent to

```
\abc
```

after one expansion of `\cs:w`.

```
\cs_to_str:N ★ \cs_to_str:N <control sequence>
```

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

4 Using or removing tokens and arguments

Tokens in the input can be read and used or read and discarded. If one or more tokens are wrapped in braces then in absorbing them the outer set will be removed. At the same time, the category code of each token is set when the token is read by a function (if it is read more than once, the category code is determined by the 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 {\langle arg_1 \rangle} {\langle arg_2 \rangle}</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 {\langle arg_1 \rangle} {\langle arg_2 \rangle} {\langle arg_3 \rangle}</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 {\langle arg_1 \rangle} {\langle arg_2 \rangle} {\langle arg_3 \rangle} {\langle arg_4 \rangle}</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 {\langle arg_1 \rangle} {\langle arg_2 \rangle} {\langle arg_3 \rangle}</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 {\langle group_1 \rangle}</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 $\text{T}_{\text{E}}\text{X}$ and $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X } 2_{\epsilon}$. Their use is discouraged in `expl3` (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

`\c_true_bool`
`\c_false_bool`

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

5.1 Tests on control sequences

<code>\cs_if_eq_p:NN</code> ★	<code>\cs_if_eq_p:NN</code> $\{\langle cs_1 \rangle\}$ $\{\langle cs_2 \rangle\}$
<code>\cs_if_eq:NNTF</code> ★	<code>\cs_if_eq:NNTF</code> $\{\langle cs_1 \rangle\}$ $\{\langle cs_2 \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

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`.

<code>\cs_if_exist_p:N</code> ★	<code>\cs_if_exist_p:N</code> $\langle control\ sequence \rangle$
<code>\cs_if_exist_p:c</code> ★	<code>\cs_if_exist:NNTF</code> $\langle control\ sequence \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\cs_if_exist:NNTF</code> ★	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 <code>true</code> .
<code>\cs_if_exist:cTF</code> ★	

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

5.2 Engine-specific conditionals

<code>\luatex_if_engine_p:</code>	★	<code>\luatex_if_engine:TF</code>	$\{\langle$ <i>true code</i> $\rangle\}$ $\{\langle$ <i>false code</i> $\rangle\}$
<code>\luatex_if_engine:TF</code>	★	Detects is the document is being compiled using LuaT _E X.	

Updated: 2011-09-06

<code>\pdftex_if_engine_p:</code>	★	<code>\pdftex_if_engine:TF</code>	$\{\langle$ <i>true code</i> $\rangle\}$ $\{\langle$ <i>false code</i> $\rangle\}$
<code>\pdftex_if_engine:TF</code>	★	Detects is the document is being compiled using pdfT _E X.	

Updated: 2011-09-06

<code>\xetex_if_engine_p:</code>	★	<code>\xetex_if_engine:TF</code>	$\{\langle$ <i>true code</i> $\rangle\}$ $\{\langle$ <i>false code</i> $\rangle\}$
<code>\xetex_if_engine:TF</code>	★	Detects is the document is being compiled using X _T T _E X.	

Updated: 2011-09-06

5.3 Primitive conditionals

The ε -T_EX 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:</code>	\langle <i>true code</i> \rangle <code>\else:</code> \langle <i>false code</i> \rangle <code>\fi:</code>
<code>\if_false:</code>	★	<code>\if_false:</code>	\langle <i>true code</i> \rangle <code>\else:</code> \langle <i>false code</i> \rangle <code>\fi:</code>
<code>\or:</code>	★	<code>\reverse_if:N</code>	\langle <i>primitive conditional</i> \rangle
<code>\else:</code>	★	<code>\if_true:</code> always executes \langle <i>true code</i> \rangle , while <code>\if_false:</code> always executes \langle <i>false code</i> \rangle .	
<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 <arg₁> <arg₂> <true code> \else: <false code> \fi:</code>
----------------------------	---	---

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

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

<code>\if:w</code>	★	<code>\if:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>
<code>\if_charcode:w</code>	★	<code>\if_catcode:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>
<code>\if_catcode:w</code>	★	

These conditionals will expand any following tokens until two unexpandable tokens are left. If you wish to prevent this expansion, prefix the token in question with `\exp_not:N`. `\if_catcode:w` tests if the category codes of the two tokens are the same whereas `\if:w` tests if the character codes are identical. `\if_charcode:w` is an alternative name for `\if:w`.

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

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

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

Execute *<true code>* if currently in horizontal mode, otherwise execute *<false code>*. Similar for the other functions.

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

<code>__chk_if_free_cs:N</code>	<code>__chk_if_free_cs:N <cs></code>
<code>__chk_if_free_cs:c</code>	

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

<code>__chk_if_exist_var:N</code>	<code>__chk_if_exist_var:N <var></code>
------------------------------------	--

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

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

Part V

The l3expan package

Argument expansion

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

Not all possible variations are implemented for every base function. Instead only those that are used within the L^AT_EX3 kernel or otherwise seem to be of general interest are implemented. Consult the module description to find out which functions are actually defined. The next section explains how to define missing variants.

1 Defining new variants

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

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

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

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

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

Providing variants in this way in style files is uncritical as the `\cs_new_nopar:Npn` function will silently accept definitions whenever the new definition is identical to an already given one. Therefore adding such definition to later releases of the kernel will not make such style files obsolete.

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

2 Methods for defining variants

`\cs_generate_variant:Nn`

Updated: 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 L^AT_EX3 approach as this makes them more readily visible in the log and so forth.

`\::c`

`\::o`

`\::f`

`\::x`

`\::v`

`\::V`

`\:::`

Part VI

The 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/> \bool_set_true:N \bool_set_true:c \bool_gset_true:N \bool_gset_true:c <hr/>	\bool_set_true:N $\langle\textit{boolean}\rangle$ Sets $\langle\textit{boolean}\rangle$ logically true.
<hr/> \bool_set_eq:NN \bool_set_eq:(cN Nc cc) \bool_gset_eq:NN \bool_gset_eq:(cN Nc cc) <hr/>	\bool_set_eq:NN $\langle\textit{boolean}_1\rangle$ $\langle\textit{boolean}_2\rangle$ Sets the content of $\langle\textit{boolean}_1\rangle$ equal to that of $\langle\textit{boolean}_2\rangle$.
<hr/> \bool_set:Nn \bool_set:cn \bool_gset:Nn \bool_gset:cn <hr/> Updated: 2012-07-08 <hr/>	\bool_set:Nn $\langle\textit{boolean}\rangle$ $\{\langle\textit{boolexpr}\rangle\}$ Evaluates the $\langle\textit{boolean expression}\rangle$ as described for \bool_if:n(TF), and sets the $\langle\textit{boolean}\rangle$ variable to the logical truth of this evaluation.
<hr/> \bool_if_p:N ★ \bool_if_p:c ★ \bool_if:NTF ★ \bool_if:cTF ★ <hr/>	\bool_if_p:N $\langle\textit{boolean}\rangle$ \bool_if:NTF $\langle\textit{boolean}\rangle$ $\{\langle\textit{true code}\rangle\}$ $\{\langle\textit{false code}\rangle\}$ Tests the current truth of $\langle\textit{boolean}\rangle$, and continues expansion based on this result.
<hr/> \bool_show:N \bool_show:c New: 2012-02-09 <hr/>	\bool_show:N $\langle\textit{boolean}\rangle$ Displays the logical truth of the $\langle\textit{boolean}\rangle$ on the terminal.
<hr/> \bool_show:n New: 2012-02-09 Updated: 2012-07-08 <hr/>	\bool_show:n $\{\langle\textit{boolean expression}\rangle\}$ Displays the logical truth of the $\langle\textit{boolean expression}\rangle$ on the terminal.
<hr/> \bool_if_exist_p:N ★ \bool_if_exist_p:c ★ \bool_if_exist:NTF ★ \bool_if_exist:cTF ★ <hr/> New: 2012-03-03 <hr/>	\bool_if_exist_p:N $\langle\textit{boolean}\rangle$ \bool_if_exist:NTF $\langle\textit{boolean}\rangle$ $\{\langle\textit{true code}\rangle\}$ $\{\langle\textit{false code}\rangle\}$ 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/> \l_tmpa_bool \l_tmpb_bool <hr/>	A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any L ^A T _E X3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> \g_tmpa_bool \g_tmpb_bool <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 T_EX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the `&` token inside `\halign`. This is necessary to allow grabbing of tokens for testing purposes, as T_EX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as `\peek_after:Nw` will result in a forbidden comparison of the internal `\endtemplate` token, yielding a fatal error. Each `\group_align_safe_begin:` must be matched by a `\group_align_safe_end:`, although this does not have to occur within the same function.

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

Updated: 2011-09-06

Stops T_EX'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.

T_EXhackers note: This is a protected version of `\prg_do_nothing:`, which therefore stops T_EX'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 5.

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

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

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

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

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

Updated: 2011-09-06

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

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

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

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

Updated: 2011-09-06

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

5 An example of recursion with quarks

Quarks are mainly used internally in the `expl3` code to define recursion functions such as `\tl_map_inline:nn` and so on. Here is a small example to demonstrate how to use quarks in this fashion. We shall define a command called `\my_map_dbl:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map_dbl:nn {abcd} {[--#1--#2--]~}` would produce “`[-a-b-] [-c-d-]`”. Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here’s the definition of `\my_map_dbl:nn`. First of all, define the function that will do the processing based on the inline function argument `#2`. Then initiate the recursion using an internal function. The token list `#1` is terminated using `\q_recursion_tail`, with delimiters according to the type of recursion (here a pair of `\q_recursion_tail`), concluding with `\q_recursion_stop`. These quarks are used to mark the end of the token list being operated upon.

```

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`.

6 Internal quark functions

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

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

7 Scan marks

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

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

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

`__scan_new:N`

`__scan_new:N` $\langle scan\ mark \rangle$

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

`\s__stop`

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

`__use_none_delimit_by_s__stop:w`

`__use_none_delimit_by_s__stop:w` $\langle tokens \rangle$ `\s__stop`

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

Part VIII

The l3token package

Token manipulation

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

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

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

1 All possible tokens

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

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three 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 {⟨integer₁⟩} {⟨integer₂⟩}</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 {⟨integer₁⟩} {⟨integer₂⟩}</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 {⟨integer₁⟩} {⟨integer₂⟩}</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</code>	$\langle token \rangle$
<code>\token_to_meaning:c</code>	★		

Inserts the current meaning of the $\langle token \rangle$ into the input stream as a series of characters of category code 12 (other). This 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</code>	$\langle token \rangle$
<code>\token_to_str:c</code>	★		

Converts the given $\langle token \rangle$ into a series of characters with category code 12 (other). 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</code>	$\langle token \rangle$
<code>\token_if_group_begin:NTF</code>	★	<code>\token_if_group_begin:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

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

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

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

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

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

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

Tests if $\langle token \rangle$ has the category code of an alignment token (`&` when normal \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 <function> <token>`

Locally sets the test variable `\l_peek_token` equal to `<token>` (as an implicit token, *not* as a token list), and then expands the `<function>`. The `<token>` will remain in the input stream as the next item after the `<function>`. The `<token>` here may be `\`, `{` 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 <function> <token>`

Globally sets the test variable `\g_peek_token` equal to `<token>` (as an implicit token, *not* as a token list), and then expands the `<function>`. The `<token>` will remain in the input stream as the next item after the `<function>`. The `<token>` here may be `\`, `{` or `}` (assuming normal 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 <test token> {<true code>} {<false code>}`

Updated: 2012-12-20

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

`\peek_catcode_ignore_spaces:NTF`

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

Updated: 2012-12-20

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

\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.

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

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

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

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

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

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

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

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

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

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

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 TeX-style integer assignment.

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

Updated: 2012-09-26

Evaluates the *⟨integer expression⟩* as described for `\int_eval:n` and leaves the absolute value of the result in the input stream as an *⟨integer denotation⟩* after two expansions.

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

Updated: 2012-09-26

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

<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 divides the first value by the second, and rounds the result towards zero. Note that division using <code>/</code> 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 obtained by subtracting <code>\int_div_truncate:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code> times $\langle integer \rangle_2$ from $\langle integer \rangle_1$. Thus, the result has the same sign as $\langle integer \rangle_1$ and its absolute value is strictly less than that of $\langle integer \rangle_2$. The result is left in the input stream as an $\langle integer \text{ denotation} \rangle$ 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	

```

\int_set_eq:NN
\int_set_eq:(cN|Nc|cc)
\int_gset_eq:NN
\int_gset_eq:(cN|Nc|cc)

```

```

\int_set_eq:NN <integer1> <integer2>

```

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

```

\int_if_exist_p:N ★
\int_if_exist_p:c ★
\int_if_exist:NTF ★
\int_if_exist:cTF ★

```

```

\int_if_exist_p:N <int>
\int_if_exist:NTF <int> {\true code} {\false code}

```

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

New: 2012-03-03

3 Setting and incrementing integers

```

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

```

```

\int_add:Nn <integer> {\integer expression}

```

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

Updated: 2011-10-22

```

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

```

```

\int_decr:N <integer>

```

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

```

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

```

```

\int_incr:N <integer>

```

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

```

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

```

```

\int_set:Nn <integer> {\integer expression}

```

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

Updated: 2011-10-22

```

\int_sub:Nn
\int_sub:cn
\int_gsub:Nn
\int_gsub:cn

```

```

\int_sub:Nn <integer> {\integer expression}

```

Subtracts the result of the $\langle integer \text{ expression} \rangle$ from the current content of the $\langle integer \rangle$.

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: 2014-05-30

`\int_step_function:nnnN` {*initial value*} {*step*} {*final value*} *function*

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

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

would print

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

`\int_step_inline:nnnn`

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

`\int_step_inline:nnnn` {*initial value*} {*step*} {*final value*} {*code*}

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

`\int_step_variable:nnnNn`

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

`\int_step_variable:nnnNn`
{*initial value*} {*step*} {*final value*} *tl var* {*code*}

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

8 Formatting integers

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

`\int_to_arabic:n` ☆

Updated: 2011-10-22

`\int_to_arabic:n` {*integer expression*}

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

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

`\int_to_Alph:n` ★

Updated: 2011-09-17

Evaluates the *⟨integer expression⟩* and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus

`\int_to_alph:n { 1 }`

places a in the input stream,

`\int_to_alph:n { 26 }`

is represented as z and

`\int_to_alph:n { 27 }`

is converted to aa. For conversions using other alphabets, use `\int_to_symbols:nnn` to define an alphabet-specific function. The basic `\int_to_alph:n` and `\int_to_Alph:n` functions should not be modified.

`\int_to_symbols:nnn` ★

Updated: 2011-09-17

`\int_to_symbols:nnn`
`{⟨integer expression⟩} {⟨total symbols⟩}`
`⟨value to symbol mapping⟩`

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

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

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

New: 2014-02-11

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

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

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

9 Converting from other formats to integers

<hr/>	
<code>\int_from_alph:n</code> ★	<code>\int_from_alph:n {⟨letters⟩}</code>
<hr/>	
<hr/>	
<hr/>	
<code>\int_from_bin:n</code> ★	<code>\int_from_bin:n {⟨binary number⟩}</code>
<hr/>	
<hr/>	
<hr/>	
<code>\int_from_hex:n</code> ★	<code>\int_from_hex:n {⟨hexadecimal number⟩}</code>
<hr/>	
<hr/>	
<hr/>	

<hr/> <code>\int_from_oct:n</code> ★ <hr/>	<code>\int_from_oct:n {\langle octal number \rangle}</code>
<hr/> New: 2014-02-11 <hr/>	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> <hr/>	<code>\int_show:n \langle integer expression \rangle</code>
<hr/> New: 2011-11-22 Updated: 2012-05-27 <hr/>	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> ★ $\langle case_1 \rangle$ <code>\or:</code> ... <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 it lower case Roman representation. Expansion ends when a space or non-expandable token is found. Note that this function produces a string of letters with category code 12 and that protected functions <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.

TeXhackers note: This is the TeX 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.

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

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

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

Part X

The l3skip package

Dimensions and skips

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

1 Creating and initialising `dim` variables

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

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

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

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

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

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

New: 2012-03-05

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

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

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

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

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

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

New: 2012-01-07

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

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

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

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

New: 2012-03-03

2 Setting dim variables

<code>\dim_add:Nn</code>	<code>\dim_add:Nn <dimension> {<dimension expression>}</code>
<code>\dim_add:cn</code>	
<code>\dim_gadd:Nn</code>	Adds the result of the $\langle dimension\ expression \rangle$ 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> .

`\dim_use:N` ★ `\dim_use:N <dimension>`

`\dim_use:c` ★ Recovers the content of a $\langle dimension \rangle$ and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ is required (such as in the argument of `\dim_eval:n`).

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

`\dim_to_decimal:n` ★ `\dim_to_decimal:n {<dimexpr>}`

New: 2014-07-15

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

For example

`\dim_to_decimal:n { 1bp }`

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

`\dim_to_decimal_in_bp:n` ★ `\dim_to_decimal_in_bp:n {<dimexpr>}`

New: 2014-07-15

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

For example

`\dim_to_decimal_in_bp:n { 1pt }`

leaves 0.99628 in the input stream, *i.e.* the magnitude of one (T_EX) point when converted to big points.

`\dim_to_decimal_in_unit:nn` ★ `\dim_to_decimal_in_unit:nn {<dimexpr1>} {<dimexpr2>}`

New: 2014-07-15

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.

For example

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

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

<hr/> <code>\dim_to_fp:n</code> ★ <hr/>	<code>\dim_to_fp:n {(dimexpr)}</code>
<hr/> New: 2012-05-08 <hr/>	Expands to an internal floating point number equal to the value of the $\langle dimexpr \rangle$ in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, <code>\dim_to_fp:n</code> can be used to speed up parts of a computation where a low precision is acceptable.

7 Viewing dim variables

<hr/> <code>\dim_show:N</code> <code>\dim_show:c</code> <hr/>	<code>\dim_show:N \langle dimension \rangle</code> Displays the value of the $\langle dimension \rangle$ on the terminal.
--	--

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

8 Constant dimensions

<hr/> <code>\c_max_dim</code> <hr/>	The maximum value that can be stored as a dimension. This can also be used as a component of a skip.
-------------------------------------	--

<hr/> <code>\c_zero_dim</code> <hr/>	A zero length as a dimension. This can also be used as a component of a skip.
--------------------------------------	---

9 Scratch dimensions

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

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

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:NNTF <skip> {(true code)} {(false code)}</code>
<code>\skip_if_exist:NNTF</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:nn ★
\skip_if_eq:nnTF ★
```

```
\skip_if_eq_p:nn {\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 ★
\skip_if_finite:nnTF ★
```

New: 2012-03-05

```
\skip_if_finite_p:n {\skipexpr}
\skip_if_finite:nnTF {\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 ★
```

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 $\langle\textit{muskip expression}\rangle$ to the current content of the $\langle\textit{muskip}\rangle$.
<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 $\langle\textit{muskip}\rangle$ to the value of $\langle\textit{muskip expression}\rangle$, 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 $\langle\textit{muskip}_1\rangle$ equal to that of $\langle\textit{muskip}_2\rangle$.
<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 $\langle\textit{muskip expression}\rangle$ from the current content of the $\langle\textit{skip}\rangle$.
<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 $\langle\textit{muskip expression}\rangle$, expanding any skips and token list variables within the $\langle\textit{expression}\rangle$ 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 $\langle\textit{muglue denotation}\rangle$ 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 $\langle\textit{internal muglue}\rangle$.
<code>\muskip_use:N</code> ★	<code>\muskip_use:N <muskip></code>
<code>\muskip_use:c</code> ★	
	Recovers the content of a $\langle\textit{skip}\rangle$ 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 $\langle\textit{dimension}\rangle$ 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</code> $\langle muskip \rangle$
<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</code> $\langle muskip\ expression \rangle$
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> <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.
<code>\l_tmpb_muskip</code> <hr/>	
<hr/> <code>\g_tmpa_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.
<code>\g_tmpb_muskip</code> <hr/>	

24 Primitive conditional

<hr/> <code>\if_dim:w</code> <hr/>	<code>\if_dim:w</code> $\langle dimen_1 \rangle$ $\langle relation \rangle$ $\langle dimen_2 \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false \rangle$ <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:w</code> $\langle dimexpr \rangle$ <code>_dim_eval_end:</code>
<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`.

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. Trailing spaces at the end of the $\langle tokens \rangle$ are discarded in the rescanning process. The $\langle setup \rangle$ is not limited to changes of category code but may contain any valid input, for example assignment of the expansion of active tokens. See also `\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. Trailing spaces at the end of the $\langle tokens \rangle$ are discarded in the rescanning process. The $\langle setup \rangle$ is not limited to changes of category code but may contain any valid input, for example assignment of the expansion of active tokens. See also `\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 `\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 $\langle tokens \rangle$.

TeXhackers note: This is a wrapper around the TeX primitive `\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:NnTF <tl var> {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_in:cnTF</code>	

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

<code>\tl_if_in:nnTF</code>	<code>\tl_if_in:nnTF {<token list₁>} {<token list₂>} {<true code>} {<false code>}</code>
<code>\tl_if_in:(Vn on no)TF</code>	

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

<code>\tl_if_single_p:N</code> ★	<code>\tl_if_single_p:N <tl var></code>
<code>\tl_if_single_p:c</code> ★	<code>\tl_if_single:NnTF <tl var> {<true code>} {<false code>}</code>
<code>\tl_if_single:NnTF</code> ★	
<code>\tl_if_single:cTF</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_p:n {<token list>}</code>
<code>\tl_if_single:nTF</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:NnTF <test token list variable></code>
<code>\tl_case:cnTF</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> .

<hr/> <code>\tl_map_break:</code> ☆ <hr/>	<code>\tl_map_break:</code>
<hr/> Updated: 2012-06-29 <hr/>	Used to terminate a <code>\tl_map...</code> function before all entries in the <i>⟨token list variable⟩</i> have been processed. This will normally take place within a conditional statement, for example

```

\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 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.

<hr/> <code>\tl_map_break:n</code> ☆ <hr/>	<code>\tl_map_break:n {⟨tokens⟩}</code>
<hr/> Updated: 2012-06-29 <hr/>	Used to terminate a <code>\tl_map...</code> function before all entries in the <i>⟨token list variable⟩</i> have been processed, inserting the <i>⟨tokens⟩</i> after the mapping has ended. This will normally take place within a conditional statement, for example

```

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

```

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

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before the *⟨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:n {(token list)}`

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

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

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

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

`\tl_to_str:N` ★ `\tl_to_str:N <tl var>`

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

`\tl_use:N` ★ `\tl_use:N <tl var>`

`\tl_use:c` ★ 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

`\tl_count:n` ★ `\tl_count:n {(tokens)}`

`\tl_count:(V|o)` ★ 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 $\langle integer denotation \rangle$.

New: 2012-05-13

`\tl_count:N` ★ `\tl_count:N <tl var>`

`\tl_count:c` ★

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 $\langle integer\ denotation \rangle$.

`\tl_reverse:n` ★ `\tl_reverse:n {\token list}`

`\tl_reverse:(V|o)` ★

Updated: 2012-01-08

Reverses the order of the $\langle items \rangle$ in the $\langle token\ list \rangle$, so that $\langle item_1 \rangle \langle item_2 \rangle \langle item_3 \rangle \dots \langle item_n \rangle$ becomes $\langle item_n \rangle \dots \langle item_3 \rangle \langle item_2 \rangle \langle item_1 \rangle$. This process will preserve unprotected space within the $\langle token\ list \rangle$. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider `\tl_reverse_items:n`. See also `\tl_reverse:N`.

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

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

`\tl_reverse:c`

`\tl_greverse:N`

`\tl_greverse:c`

Updated: 2012-01-08

Reverses the order of the $\langle items \rangle$ stored in $\langle tl\ var \rangle$, so that $\langle item_1 \rangle \langle item_2 \rangle \langle item_3 \rangle \dots \langle item_n \rangle$ becomes $\langle item_n \rangle \dots \langle item_3 \rangle \langle item_2 \rangle \langle item_1 \rangle$. This process will preserve unprotected spaces within the $\langle token\ list\ variable \rangle$. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. See also `\tl_reverse:n`, and, for improved performance, `\tl_reverse_items:n`.

`\tl_reverse_items:n` ★ `\tl_reverse_items:n {\token list}`

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 `\tl_reverse:n`.

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.

`\tl_trim_spaces:n` ★ `\tl_trim_spaces:n {\token list}`

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.

```

\tl_trim_spaces:N
\tl_trim_spaces:c
\tl_gtrim_spaces:N
\tl_gtrim_spaces:c

```

New: 2011-07-09

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

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

10 The first token from a token list

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.

```

\tl_head:N ★
\tl_head:(n|V|v|f) ★

```

Updated: 2012-09-29

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

Leaves in the input stream the first $\langle item \rangle$ in the $\langle token\ list \rangle$, discarding the rest of the $\langle token\ list \rangle$. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example

```
\tl_head:n { abc }
```

and

```
\tl_head:n { ~ abc }
```

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

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

yields `ab`. A blank $\langle token\ list \rangle$ (see `\tl_if_blank:nTF`) will result in `\tl_head:n` leaving nothing in the input stream.

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

```

\tl_head:w ★

```

```
\tl_head:w <token list> { } \q_stop
```

Leaves in the input stream the first $\langle item \rangle$ in the $\langle token\ list \rangle$, discarding the rest of the $\langle token\ list \rangle$. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank $\langle token\ list \rangle$ (which consists only of space characters) will result in a low-level T_EX error, which may be avoided by the inclusion of an empty group in the input (as shown), without the need for an explicit test. Alternatively, `\tl_if_blank:nF` may be used to avoid using the function with a “blank” argument. This function requires only a single expansion, and thus is suitable for use within an `o`-type expansion. In general, `\tl_head:n` should be preferred if the number of expansions is not critical.

<code>\tl_tail:N</code>	★	<code>\tl_tail:n {⟨token list⟩}</code>
-------------------------	---	--

<code>\tl_tail:(n V v f)</code>	★
---------------------------------	---

Updated: 2012-09-01

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

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

and

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

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.

<code>\tl_if_head_eq_catcode_p:nN</code>	★	<code>\tl_if_head_eq_catcode_p:nN {⟨token list⟩} ⟨test token⟩</code>
--	---	--

<code>\tl_if_head_eq_catcode:nNTF</code>	★	<code>\tl_if_head_eq_catcode:nNTF {⟨token list⟩} ⟨test token⟩</code>
--	---	--

<code>{⟨true code⟩} {⟨false code⟩}</code>

Updated: 2012-07-09

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

<code>\tl_if_head_eq_charcode_p:nN</code>	★	<code>\tl_if_head_eq_charcode_p:nN {⟨token list⟩} ⟨test token⟩</code>
---	---	---

<code>\tl_if_head_eq_charcode_p:fN</code>	★	<code>\tl_if_head_eq_charcode:nNTF {⟨token list⟩} ⟨test token⟩</code>
---	---	---

<code>\tl_if_head_eq_charcode:nNTF</code>	★	<code>{⟨true code⟩} {⟨false code⟩}</code>
---	---	---

<code>\tl_if_head_eq_charcode:fNTF</code>	★
---	---

Updated: 2012-07-09

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

<code>\tl_if_head_eq_meaning_p:nN</code>	★	<code>\tl_if_head_eq_meaning_p:nN {⟨token list⟩} ⟨test token⟩</code>
--	---	--

<code>\tl_if_head_eq_meaning:nNTF</code>	★	<code>\tl_if_head_eq_meaning:nNTF {⟨token list⟩} ⟨test token⟩</code>
--	---	--

<code>{⟨true code⟩} {⟨false code⟩}</code>

Updated: 2012-07-09

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

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

New: 2012-07-08

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

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

Updated: 2012-07-08

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

11 Using a single item

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

New: 2014-07-17

Indexing items in the *⟨token list⟩* from 1 on the left, this function will evaluate the *⟨integer expression⟩* and leave the appropriate item from the *⟨token list⟩* in the input stream. If the *⟨integer expression⟩* is negative, indexing occurs from the right of the token list, starting at -1 for the right-most item. If the index is out of bounds, then the function expands to nothing.

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

12 Viewing token lists

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

Updated: 2012-09-09

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

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

<hr/> <hr/>	<code>\tl_show:n</code>	<code>\tl_show:n</code> $\langle token\ list \rangle$
<hr/>	Updated: 2012-09-09	Displays the $\langle token\ list \rangle$ on the terminal.

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

13 Constant token lists

<hr/> <hr/>	<code>\c_empty_tl</code>	Constant that is always empty.
-------------	--------------------------	--------------------------------

<hr/> <hr/>	<code>\c_job_name_tl</code>	Constant that gets the “job name” assigned when T _E X starts.
-------------	-----------------------------	--

T_EXhackers note: This copies the contents of the primitive `\jobname`. It is a constant that is set by T_EX and should not be overwritten by the package.

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

14 Scratch token lists

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

15 Internal functions

<hr/> <hr/>	<code>__tl_trim_spaces:nn</code>	<code>__tl_trim_spaces:nn { \q_mark $\langle token\ list \rangle$ } {$\langle continuation \rangle$}</code>
<hr/>		This function removes all leading and trailing explicit space characters from the $\langle token\ list \rangle$, and expands to the $\langle continuation \rangle$, followed by a brace group containing <code>\use_none:n \q_mark $\langle trimmed\ token\ list \rangle$</code> . For instance, <code>\tl_trim_spaces:n</code> is implemented by taking the $\langle continuation \rangle$ to be <code>\exp_not:o</code> , and the 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 l3str package

Strings

T_EX associates each character with a category code: as such, there is no concept of a “string” as commonly understood in many other programming languages. However, there are places where we wish to manipulate token lists while in some sense “ignoring” category codes: this is done by treating token lists as strings in a T_EX sense.

A T_EX string (and thus an expl3 string) is a series of characters which have category code 12 (“other”) with the exception of space characters which have category code 10 (“space”). Thus at a technical level, a T_EX string is a token list with the appropriate category codes. In this documentation, these will simply be referred to as strings: note that they can be stored in token lists as normal.

The functions documented here take literal token lists, convert to strings and then carry out manipulations. Thus they may informally be described as “ignoring” category code. Note that the functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) will generate strings from the appropriate input: these are documented in `l3basics`, `l3tl` and `l3token`, respectively.

1 The first character from a string

<code>\str_head:n</code>	★	<code>\str_head:n {⟨token list⟩}</code>
<code>\str_tail:n</code>	★	<code>\str_tail:n {⟨token list⟩}</code>

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.

1.1 Tests on strings

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

Compares the two *⟨token lists⟩* on a character by character basis, and is **true** if the two lists contain the same characters in the same order. Thus for example

```
\str_if_eq_p:no { abc } { \tl_to_str:n { abc } }
```

is logically **true**.

<code>\str_if_eq_x_p:nn</code> ★ <code>\str_if_eq_x:nnTF</code> ★	<code>\str_if_eq_x_p:nn</code> {<tl ₁ >} {<tl ₂ >} <code>\str_if_eq_x:nnTF</code> {<tl ₁ >} {<tl ₂ >} {<true code>} {<false code>}
--	---

New: 2012-06-05

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

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

is logically **true**.

<code>\str_case:nnTF</code> ★ <code>\str_case:onTF</code> ★	<code>\str_case:nnTF</code> {<test string>} { {<string case ₁ >} {<code case ₁ >} {<string case ₂ >} {<code case ₂ >} ... {<string case _n >} {<code case _n >} } {<true code>} {<false code>}
--	--

New: 2013-07-24

This function compares the <test string> in turn with each of the <string cases>. If the two are equal (as described for `\str_if_eq:nnTF` then the associated <code> is left in the input stream. If any of the cases are matched, the <true code> is also inserted into the input stream (after the code for the appropriate case), while if none match then the <false code> is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

<code>\str_case_x:nnTF</code> ★	<code>\str_case_x:nnn</code> {<test string>} { {<string case ₁ >} {<code case ₁ >} {<string case ₂ >} {<code case ₂ >} ... {<string case _n >} {<code case _n >} } {<true code>} {<false code>}
---------------------------------	---

New: 2013-07-24

This function compares the full expansion of the <test string> in turn with the full expansion of the <string cases>. If the two full expansions are equal (as described for `\str_if_eq:nnTF` then the associated <code> is left in the input stream. If any of the cases are matched, the <true code> is also inserted into the input stream (after the code for the appropriate case), while if none match then the <false code> is inserted. The function `\str_case_x:nn`, which does nothing if there is no match, is also available. The <test string> is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

2 String manipulation

`\str_fold_case:n` ☆

New: 2014-06-19

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

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

String folding is a process used for material such as identifiers rather than for “text”. The folding provided by `\str_fold_case:n` follows the mappings provided by the [Unicode Consortium](#), who [state](#):

Case folding is primarily used for caseless comparison of text, such as identifiers in a computer program, rather than actual text transformation. Case folding in Unicode is based on the lowercase mapping, but includes additional changes to the source text to help make it language-insensitive and consistent. As a result, case-folded text should be used solely for internal processing and generally should not be stored or displayed to the end user.

The folding approach implemented by `\str_fold_case:n` follows the “full” scheme defined by the Unicode Consortium (*e.g.* SSfolds to SS). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (*i.e.* I always folds to i and not to ı).

TeXhackers note: As with all `expl3` functions, the input supported by `\str_fold_case:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfTeX *only* the Latin alphabet characters A–Z will be case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both XeTeX and LuaTeX, subject only to the fact that XeTeX in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

2.1 Internal string functions

`__str_if_eq_x:nn` ★

`__str_if_eq_x:nn` $\{\langle tl_1 \rangle\} \{\langle tl_2 \rangle\}$

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. Leaves 0 in the input stream if the condition is true, and +1 or -1 otherwise.

`\@@_if_eq_x_return:nn`

`\@@_if_eq_x_return:nn` $\{\langle tl_1 \rangle\} \{\langle tl_2 \rangle\}$

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. 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 XIII

The l3seq package

Sequences and stacks

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

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

1 Creating and initialising sequences

`\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_from_clist:NN`
`\seq_set_from_clist:(cN|Nc|cc|Nn|cn)`
`\seq_gset_from_clist:NN`
`\seq_gset_from_clist:(cN|Nc|cc|Nn|cn)`

`\seq_set_from_clist:NN` *⟨sequence⟩* *⟨comma-list⟩*

New: 2014-07-17

Converts the data in the *⟨comma list⟩* into a *⟨sequence⟩*: the original *⟨comma list⟩* is unchanged.

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

2 Appending data to sequences

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

3 Recovering items from sequences

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

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

`\seq_item:Nn` ★
`\seq_item:cn` ★

New: 2014-07-17

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

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

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

4 Recovering values from sequences with branching

The functions in this section combine tests for non-empty sequences with recovery of an item from the sequence. They offer increased readability and performance over separate testing and recovery phases.

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

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

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

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

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

New: 2012-05-19

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

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

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

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

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

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

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

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

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

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

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

New: 2012-05-19

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

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

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

New: 2012-05-19

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

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

5 Modifying sequences

While sequences are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update sequences, while retaining the order of the unaffected entries.

```
\seq_remove_duplicates:N
\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
```

```
\seq_remove_duplicates:N <sequence>
```

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

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

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

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

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

```
\seq_reverse:N
\seq_reverse:c
\seq_greverse:N
\seq_greverse:c
```

New: 2014-07-18

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

Reverses the order of the items stored in the $\langle sequence \rangle$.

6 Sequence conditionals

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

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

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

7 Mapping to sequences

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

Updated: 2012-06-29

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

Updated: 2012-06-29

<code>\seq_map_variable:NN</code>	<code>\seq_map_variable:NN</code> $\langle sequence \rangle$ $\langle tl var. \rangle$ $\{\langle function using tl var. \rangle\}$
<code>\seq_map_variable:(Ncn cN ccn)</code>	

Updated: 2012-06-29

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

`\seq_map_break:` ☆

Updated: 2012-06-29

`\seq_map_break:`

Used to terminate a `\seq_map...` function before all entries in the $\langle sequence \rangle$ have been processed. This will normally take place within a conditional statement, for example

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

Use outside of a `\seq_map...` scenario will lead to low level 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
 New: 2012-04-26 other non-kernel code and so should only be used for short-term storage.

`\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
 New: 2012-04-26 by other non-kernel code and so should only be used for short-term storage.

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 XIV

The l3clist package

Comma separated lists

Comma lists contain ordered data where items can be added to the left or right end of the list. The resulting ordered list can then be mapped over using `\clist_map_function:NN`. Several items can be added at once, and spaces are removed from both sides of each item on input. Hence,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~ a ~ , ~ {b} ~ }
\clist_put_right:Nn \l_my_clist { ~ { c ~ } , d }
```

results in `\l_my_clist` containing `a,{b},{c~},d`. Comma lists cannot contain empty items, thus

```
\clist_clear_new:N \l_my_clist
\clist_put_right:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

will leave `true` in the input stream. To include an item which contains a comma, or starts or ends with a space, surround it with braces. The sequence data type should be preferred to comma lists if items are to contain `{`, `}`, or `#` (assuming the usual \TeX category codes apply).

1 Creating and initialising comma lists

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

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

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

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

```
\clist_const:Nn <clist var> {<comma list>}
```

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

New: 2014-07-05

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

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

Clears all items from the *<comma list>*.

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

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

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

New: 2014-07-17

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

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

new comma list.

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

list really is a comma list.

New: 2012-03-03

2 Adding data to comma lists

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

New: 2011-09-06

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

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

Updated: 2011-09-05

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

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

Updated: 2011-09-05

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

3 Modifying comma lists

While comma lists are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update comma lists, while retaining the order of the unaffected entries.

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

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

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

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

Updated: 2011-09-06

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

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

New: 2014-07-18

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

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

```
\clist_reverse:n
```

New: 2014-07-18

```
\clist_reverse:n {<comma list>}
```

Leaves the items in the *<comma list>* in the input stream in reverse order. Braces and spaces are preserved by this process.

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

4 Comma list conditionals

```
\clist_if_empty_p:N *
\clist_if_empty_p:c *
\clist_if_empty:NTF *
\clist_if_empty:cTF *
```

```
\clist_if_empty_p:N <comma list>
\clist_if_empty:NTF <comma list> {<true code>} {<false code>}
```

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

```
\clist_if_empty_p:n *
\clist_if_empty:nTF *
```

New: 2014-07-05

```
\clist_if_empty_p:n {<comma list>}
\clist_if_empty:nTF {<comma list>} {<true code>} {<false code>}
```

Tests if the *<comma list>* is empty (containing no items). The rules for space trimming are as for other *n*-type comma-list functions, hence the comma list `{~,~,~}` (without outer braces) is empty, while `{~,{}},}` (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

```
\clist_if_in:NnTF
\clist_if_in:(NV|No|cn|cV|co|nn|nV|no)TF
```

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

Updated: 2011-09-06

Tests if the *<item>* is present in the *<comma list>*. In the case of an *n*-type *<comma list>*, spaces are stripped from each item, but braces are not removed. Hence,

```
\clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}
```

yields **false**.

T_EXhackers note: The *<item>* may not contain `{`, `}`, or `#` (assuming the usual T_EX category codes apply), and should not contain `,` nor start or end with a space.

5 Mapping to comma lists

The functions described in this section apply a specified function to each item of a comma list.

When the comma list is given explicitly, as an *n*-type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if your comma list that is being mapped is $\{a, \{b\}, \{c\}, \}$ then the arguments passed to the mapped function are ‘a’, ‘{b}’, an empty argument, and ‘c’.

When the comma list is given as an *N*-type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using *n*-type comma lists.

<code>\clist_map_function:NN</code>	☆	<code>\clist_map_function:NN</code> $\langle comma\ list \rangle$ $\langle function \rangle$
<code>\clist_map_function:(cN nN)</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:Nn</code> $\langle comma\ list \rangle$ $\{ \langle inline\ function \rangle \}$
<code>\clist_map_inline:(cn nn)</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:NNn</code>	<code>\clist_map_variable:NNn</code> $\langle comma\ list \rangle$ $\langle tl\ var. \rangle$ $\{ \langle function\ using\ tl\ var. \rangle \}$
<code>\clist_map_variable:(cNn nNn)</code>	

Updated: 2012-06-29

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

`\clist_map_break:` ☆

Updated: 2012-06-29

`\clist_map_break:`

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

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\clist_map...` scenario will lead to low level 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 {⟨tokens⟩}`

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

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break:n { <tokens> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\clist_map...` scenario will lead to low level 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.

`\clist_count:N` ☆

`\clist_count:(c|n)` ☆

New: 2012-07-13

`\clist_count:N ⟨comma list⟩`

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

6 Using the content of comma lists directly

<code>\clist_use:Nnnn</code> ★ <code>\clist_use: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.

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

<code>\clist_push:Nn</code>	<code>\clist_push:Nn <comma list> {<items>}</code>
<code>\clist_push:(NV No Nx cn cV co cx)</code>	
<code>\clist_gpush:Nn</code>	
<code>\clist_gpush:(NV No Nx cn cV co cx)</code>	

Adds the $\{\langle items \rangle\}$ to the top of the $\langle comma list \rangle$. Spaces are removed from both sides of each item.

8 Using a single item

<code>\clist_item:Nn</code> ★	<code>\clist_item:Nn <comma list> {<integer expression>}</code>
<code>\clist_item:(cn nn)</code> ★	

New: 2014-07-17

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

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

9 Viewing comma lists

<code>\clist_show:N</code>	<code>\clist_show:N <comma list></code>
<code>\clist_show:c</code>	Displays the entries in the $\langle comma list \rangle$ 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.

10 Constant and scratch comma lists

<code>\c_empty_clist</code>	Constant that is always empty.
New: 2012-07-02	

<code>\l_tmpa_clist</code>	Scratch comma lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_clist</code>	
New: 2011-09-06	

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

Part XV

The l3prop package

Property lists

L^AT_EX3 implements a “property list” data type, which contain an unordered list of entries each of which consists of a $\langle key \rangle$ and an associated $\langle value \rangle$. The $\langle key \rangle$ and $\langle value \rangle$ may both be any $\langle balanced\ text \rangle$. It is possible to map functions to property lists such that the function is applied to every key–value pair within the list.

Each entry in a property list must have a unique $\langle key \rangle$: if an entry is added to a property list which already contains the $\langle key \rangle$ then the new entry will overwrite the existing one. The $\langle keys \rangle$ are compared on a string basis, using the same method as `\str_if_eq:nn`.

Property lists are intended for storing key-based information for use within code. This is in contrast to key–value lists, which are a form of *input* parsed by the `keys` module.

1 Creating and initialising property lists

 \backslash prop_new:N
 \backslash prop_new:c

 \backslash 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.

 \backslash prop_clear:N
 \backslash prop_clear:c
 \backslash prop_gclear:N
 \backslash prop_gclear:c

 \backslash prop_clear:N $\langle property\ list \rangle$

Clears all entries from the $\langle property\ list \rangle$.

 \backslash prop_clear_new:N
 \backslash prop_clear_new:c
 \backslash prop_gclear_new:N
 \backslash prop_gclear_new:c

 \backslash 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.

 \backslash prop_set_eq:NN
 \backslash prop_set_eq:(cN|Nc|cc)
 \backslash prop_gset_eq:NN
 \backslash prop_gset_eq:(cN|Nc|cc)

 \backslash prop_set_eq:NN $\langle property\ list_1 \rangle$ $\langle property\ list_2 \rangle$

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

2 Adding entries to property lists

<code>\prop_put:Nnn</code> <code>\prop_put:(NnV Nno Nnx NVn NVV Non Noo cnn cnV cno cnx cVn cVV con coo)</code> <code>\prop_gput:Nnn</code> <code>\prop_gput:(NnV Nno Nnx NVn NVV Non Noo cnn cnV cno cnx cVn cVV con coo)</code>	<code>\prop_put:Nnn <property list></code> <code>{<key>} {<value>}</code>
--	--

Updated: 2012-07-09

Adds an entry to the *<property list>* which may be accessed using the *<key>* and which has *<value>*. Both the *<key>* and *<value>* may contain any *<balanced text>*. The *<key>* is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored. If the *<key>* is already present in the *<property list>*, the existing entry is overwritten by the new *<value>*.

<code>\prop_put_if_new:Nnn</code> <code>\prop_put_if_new:cnn</code> <code>\prop_gput_if_new:Nnn</code> <code>\prop_gput_if_new:cnn</code>	<code>\prop_put_if_new:Nnn <property list> {<key>} {<value>}</code> <p>If the <i><key></i> is present in the <i><property list></i> then no action is taken. If the <i><key></i> is not present in the <i><property list></i> then a new entry is added. Both the <i><key></i> and <i><value></i> may contain any <i><balanced text></i>. The <i><key></i> is stored after processing with <code>\tl_to_str:n</code>, meaning that category codes are ignored.</p>
--	---

3 Recovering values from property lists

<code>\prop_get:NnN</code> <code>\prop_get:(NVN NoN cnN cVN coN)</code>	<code>\prop_get:NnN <property list> {<key>} <tl var></code>
--	---

Updated: 2011-08-28

Recovers the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* will contain the special marker `\q_no_value`. The *<token list variable>* is set within the current \TeX group. See also `\prop_get:NnNTF`.

<code>\prop_pop:NnN</code> <code>\prop_pop:(NoN cnN coN)</code>	<code>\prop_pop:NnN <property list> {<key>} <tl var></code> <p>Recovers the <i><value></i> stored with <i><key></i> from the <i><property list></i>, and places this in the <i><token list variable></i>. If the <i><key></i> is not found in the <i><property list></i> then the <i><token list variable></i> will contain the special marker <code>\q_no_value</code>. The <i><key></i> and <i><value></i> are then deleted from the property list. Both assignments are local. See also <code>\prop_pop:NnNTF</code>.</p>
--	---

Updated: 2011-08-18

<code>\prop_gpop:NnN</code> <code>\prop_gpop:(NoN cnN coN)</code>	<code>\prop_gpop:NnN <property list> {<key>} <tl var></code> <p>Recovers the <i><value></i> stored with <i><key></i> from the <i><property list></i>, and places this in the <i><token list variable></i>. If the <i><key></i> is not found in the <i><property list></i> then the <i><token list variable></i> will contain the special marker <code>\q_no_value</code>. The <i><key></i> and <i><value></i> are then deleted from the property list. The <i><property list></i> is modified globally, while the assignment of the <i><token list variable></i> is local. See also <code>\prop_gpop:NnNTF</code>.</p>
--	---

Updated: 2011-08-18

<code>\prop_item:Nn</code> ★	<code>\prop_item:Nn</code> $\langle property list \rangle$ $\{\langle key \rangle\}$
<code>\prop_item:cn</code> ★	
New: 2014-07-17	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.

4 Modifying property lists

<code>\prop_remove:Nn</code>	<code>\prop_remove:Nn</code> $\langle property list \rangle$ $\{\langle key \rangle\}$
<code>\prop_remove:(NV cn cV)</code>	
<code>\prop_gremove:Nn</code>	Removes the entry listed under $\langle key \rangle$ from the $\langle property list \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ no change occurs, <i>i.e.</i> there is no need to test for the existence of a key before deleting it.
<code>\prop_gremove:(NV cn cV)</code>	
New: 2012-05-12	

5 Property list conditionals

<code>\prop_if_exist_p:N</code> ★	<code>\prop_if_exist_p:N</code> $\langle property list \rangle$
<code>\prop_if_exist_p:c</code> ★	<code>\prop_if_exist:NTF</code> $\langle property list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\prop_if_exist:N\underline{TF}</code> ★	Tests whether the $\langle property list \rangle$ is currently defined. This does not check that the $\langle property list \rangle$ really is a property list variable.
<code>\prop_if_exist:c\underline{TF}</code> ★	
New: 2012-03-03	

<code>\prop_if_empty_p:N</code> ★	<code>\prop_if_empty_p:N</code> $\langle property list \rangle$
<code>\prop_if_empty_p:c</code> ★	<code>\prop_if_empty:NTF</code> $\langle property list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\prop_if_empty:N\underline{TF}</code> ★	Tests if the $\langle property list \rangle$ is empty (containing no entries).
<code>\prop_if_empty:c\underline{TF}</code> ★	

<code>\prop_if_in_p:Nn</code> ★	<code>\prop_if_in:NnTF</code> $\langle property list \rangle$ $\{\langle key \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\prop_if_in_p:(NV No cn cV co)</code> ★	
<code>\prop_if_in:Nn\underline{TF}</code> ★	
<code>\prop_if_in:(NV No cn cV co)\underline{TF}</code> ★	
Updated: 2011-09-15	

Tests if the $\langle key \rangle$ is present in the $\langle property list \rangle$, making the comparison using the method described by `\str_if_eq:nnTF`.

T_EXhackers note: This function iterates through every key–value pair in the $\langle property list \rangle$ and is therefore slower than using the non-expandable `\prop_get:NnNTF`.

6 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated valued. This makes them useful for cases where different cases follow dependent on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

<code>\prop_get:NnNTF</code> <code>\prop_get:(NVN NoN cnN cVN coN)TF</code>	<code>\prop_get:NnNTF <property list> {<key>} <token list variable></code> <code>{<true code>} {<false code>}</code>
--	---

Updated: 2012-05-19

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

<code>\prop_pop:NnNTF</code> <code>\prop_pop:cnNTF</code>	<code>\prop_pop:NnNTF <property list> {<key>} <token list variable> {<true code>}</code> <code>{<false code>}</code>
--	---

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

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

<code>\prop_gpop:NnNTF</code> <code>\prop_gpop:cnNTF</code>	<code>\prop_gpop:NnNTF <property list> {<key>} <token list variable> {<true code>}</code> <code>{<false code>}</code>
--	--

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

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

7 Mapping to property lists

<code>\prop_map_function:NN</code> ☆ <code>\prop_map_function:cN</code> ☆	<code>\prop_map_function:NN <property list> <function></code>
--	---

Updated: 2013-01-08

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

<code>\prop_map_inline:Nn</code>	<code>\prop_map_inline:Nn <property list> {<inline function>}</code>
<code>\prop_map_inline:cn</code>	Applies <i><inline function></i> to every <i><entry></i> stored within the <i><property list></i> . The <i><inline function></i> should consist of code which will receive the <i><key></i> as #1 and the <i><value></i> as #2. The order in which <i><entries></i> are returned is not defined and should not be relied upon.
Updated: 2013-01-08	

<code>\prop_map_break: ☆</code>	<code>\prop_map_break:</code>
Updated: 2012-06-29	Used to terminate a <code>\prop_map...</code> function before all entries in the <i><property list></i> have been processed. This will normally take place within a conditional statement, for example

```

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

```

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

<code>\prop_map_break:n ☆</code>	<code>\prop_map_break:n {<tokens>}</code>
Updated: 2012-06-29	Used to terminate a <code>\prop_map...</code> function before all entries in the <i><property list></i> have been processed, inserting the <i><tokens></i> after the mapping has ended. This will normally take place within a conditional statement, for example

```

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

```

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

8 Viewing property lists

<code>\prop_show:N</code>	<code>\prop_show:N <property list></code>
<code>\prop_show:c</code>	Displays the entries in the <i><property list></i> in the terminal.
Updated: 2012-09-09	

9 Scratch property lists

<code>\l_tmpa_prop</code>	Scratch property lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_prop</code>	
New: 2012-06-23	

<code>\g_tmpa_prop</code>	Scratch property lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_prop</code>	
New: 2012-06-23	

10 Constants

<code>\c_empty_prop</code>	A permanently-empty property list used for internal comparisons.
----------------------------	--

11 Internal property list functions

<code>\s__prop</code>	The internal token used at the beginning of property lists. This is also used after each $\langle key \rangle$ (see <code>__prop_pair:wn</code>).
-----------------------	---

<code>__prop_pair:wn</code>	<code>__prop_pair:wn $\langle key \rangle$ \s__prop {$\langle item \rangle$}</code>
	The internal token used to begin each key–value pair in the property list. If expanded outside of a mapping or manipulation function, an error will be raised. The definition should always be set globally.

<code>\l__prop_internal_tl</code>	Token list used to store new key–value pairs to be inserted by functions of the <code>\prop_put:Nnn</code> family.
-----------------------------------	--

<code>__prop_split:NnTF</code>	<code>__prop_split:NnTF $\langle property list \rangle$ {$\langle key \rangle$} {$\langle true code \rangle$} {$\langle false code \rangle$}</code>
Updated: 2013-01-08	Splits the $\langle property list \rangle$ at the $\langle key \rangle$, giving three token lists: the $\langle extract \rangle$ of $\langle property list \rangle$ before the $\langle key \rangle$, the $\langle value \rangle$ associated with the $\langle key \rangle$ and the $\langle extract \rangle$ of the $\langle property list \rangle$ after the $\langle value \rangle$. Both $\langle extracts \rangle$ retain the internal structure of a property list, and the concatenation of the two $\langle extracts \rangle$ is a property list. If the $\langle key \rangle$ is present in the $\langle property list \rangle$ then the $\langle true code \rangle$ is left in the input stream, with #1, #2, and #3 replaced by the first $\langle extract \rangle$, the $\langle value \rangle$, and the second extract. If the $\langle key \rangle$ is not present in the $\langle property list \rangle$ then the $\langle false code \rangle$ is left in the input stream, with no trailing material. Both $\langle true code \rangle$ and $\langle false code \rangle$ are used in the replacement text of a macro defined internally, hence macro parameter characters should be doubled, except #1, #2, and #3 which stand in the $\langle true code \rangle$ for the three extracts from the property list. The $\langle key \rangle$ comparison takes place as described for <code>\str_if_eq:nn</code> .

Part XVI

The l3box package

Boxes

There are three kinds of box operations: horizontal mode denoted with prefix `\hbox_`, vertical mode with prefix `\vbox_`, and the generic operations working in both modes with prefix `\box_`.

1 Creating and initialising boxes

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

<code>\box_clear:N</code>	<code>\box_clear:N <box></code>
<code>\box_clear:c</code>	Clears the content of the $\langle box \rangle$ by setting the box equal to <code>\c_void_box</code> .
<code>\box_gclear:N</code>	
<code>\box_gclear:c</code>	

<code>\box_clear_new:N</code>	<code>\box_clear_new:N <box></code>
<code>\box_clear_new:c</code>	Ensures that the $\langle box \rangle$ exists globally by applying <code>\box_new:N</code> if necessary, then applies <code>\box_(g)clear:N</code> to leave the $\langle box \rangle$ empty.
<code>\box_gclear_new:N</code>	
<code>\box_gclear_new:c</code>	

<code>\box_set_eq:NN</code>	<code>\box_set_eq:NN <box₁> <box₂></code>
<code>\box_set_eq:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$.
<code>\box_gset_eq:NN</code>	
<code>\box_gset_eq:(cN Nc cc)</code>	

<code>\box_set_eq_clear:NN</code>	<code>\box_set_eq_clear:NN <box₁> <box₂></code>
<code>\box_set_eq_clear:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ within the current TeX group equal to that of $\langle box_2 \rangle$, then clears $\langle box_2 \rangle$ globally.

<code>\box_gset_eq_clear:NN</code>	<code>\box_gset_eq_clear:NN <box₁> <box₂></code>
<code>\box_gset_eq_clear:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$, then clears $\langle box_2 \rangle$. These assignments are global.

<code>\box_if_exist_p:N</code> ★	<code>\box_if_exist_p:N</code> $\langle box \rangle$
<code>\box_if_exist_p:c</code> ★	<code>\box_if_exist:N</code> $\langle box \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\box_if_exist:N</code> ★	Tests whether the $\langle box \rangle$ is currently defined. This does not check that the $\langle box \rangle$ really is a box.
<code>\box_if_exist:c</code> ★	

New: 2012-03-03

2 Using boxes

<code>\box_use:N</code>	<code>\box_use:N</code> $\langle box \rangle$
<code>\box_use:c</code>	Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting.

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

<code>\box_use_clear:N</code>	<code>\box_use_clear:N</code> $\langle box \rangle$
<code>\box_use_clear:c</code>	Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting, then globally clears the content of the $\langle box \rangle$.

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

<code>\box_move_right:nn</code>	<code>\box_move_right:nn</code> $\{\langle dimexpr \rangle\}$ $\{\langle box\ function \rangle\}$
<code>\box_move_left:nn</code>	This function operates in vertical mode, and inserts the material specified by the $\langle box\ function \rangle$ such that its reference point is displaced horizontally by the given $\langle dimexpr \rangle$ from the reference point for typesetting, to the right or left as appropriate. The $\langle box\ function \rangle$ should be a box operation such as <code>\box_use:N</code> $\langle box \rangle$ or a “raw” box specification such as <code>\vbox:n</code> $\{ xyz \}$.

<code>\box_move_up:nn</code>	<code>\box_move_up:nn</code> $\{\langle dimexpr \rangle\}$ $\{\langle box\ function \rangle\}$
<code>\box_move_down:nn</code>	This function operates in horizontal mode, and inserts the material specified by the $\langle box\ function \rangle$ such that its reference point is displaced vertical by the given $\langle dimexpr \rangle$ from the reference point for typesetting, up or down as appropriate. The $\langle box\ function \rangle$ should be a box operation such as <code>\box_use:N</code> $\langle box \rangle$ or a “raw” box specification such as <code>\vbox:n</code> $\{ xyz \}$.

3 Measuring and setting box dimensions

<code>\box_dp:N</code>	<code>\box_dp:N</code> $\langle box \rangle$
<code>\box_dp:c</code>	Calculates the depth (below the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension\ expression \rangle$.

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

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

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

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

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

<code>\box_set_dp:Nn</code>	<code>\box_set_dp:Nn <box> {<dimension expression>}</code>
<code>\box_set_dp:cn</code>	Set the depth (below the baseline) of the $\langle box \rangle$ to the value of the $\{ \langle dimension expression \rangle \}$. This is a global assignment.

Updated: 2011-10-22

<code>\box_set_ht:Nn</code>	<code>\box_set_ht:Nn <box> {<dimension expression>}</code>
<code>\box_set_ht:cn</code>	Set the height (above the baseline) of the $\langle box \rangle$ to the value of the $\{ \langle dimension expression \rangle \}$. This is a global assignment.

Updated: 2011-10-22

<code>\box_set_wd:Nn</code>	<code>\box_set_wd:Nn <box> {<dimension expression>}</code>
<code>\box_set_wd:cn</code>	Set the width of the $\langle box \rangle$ to the value of the $\{ \langle dimension expression \rangle \}$. This is a global assignment.

Updated: 2011-10-22

4 Box conditionals

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

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

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

5 The last box inserted

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

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> <hr/>	
<code>\hbox_gset:Nn</code> <hr/>	Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$.
<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> <hr/>	
<code>\hbox_gset_to_wd:Nnn</code> <hr/>	Typesets the $\langle contents \rangle$ to the width given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$.
<code>\hbox_gset_to_wd: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`.

<code>\vbox_unpack:N</code>	<code>\vbox_unpack:N <box></code>
<code>\vbox_unpack:c</code>	Unpacks the content of the vertical <code><box></code> , retaining any stretching or shrinking applied when the <code><box></code> was set.

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

<code>\vbox_unpack_clear:N</code>	<code>\vbox_unpack:N <box></code>
<code>\vbox_unpack_clear:c</code>	Unpacks the content of the vertical <code><box></code> , retaining any stretching or shrinking applied when the <code><box></code> was set. The <code><box></code> is then cleared globally.

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

11 Primitive box conditionals

<code>\if_hbox:N</code> ★	<code>\if_hbox:N <box></code> <code><true code></code> <code>\else:</code> <code><false code></code> <code>\fi:</code>
---------------------------	--

Tests is `<box>` is a horizontal box.

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

<code>\if_vbox:N</code> ★	<code>\if_vbox:N <box></code> <code><true code></code> <code>\else:</code> <code><false code></code> <code>\fi:</code>
---------------------------	--

Tests is `<box>` is a vertical box.

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

<code>\if_box_empty:N</code> ★	<code>\if_box_empty:N <box></code> <code><true code></code> <code>\else:</code> <code><false code></code> <code>\fi:</code>
--------------------------------	---

Tests is `<box>` is an empty (void) box.

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

Part XVII

The l3coffins package

Coffin code layer

The material in this module provides the low-level support system for coffins. For details about the design concept of a coffin, see the xcoffins module (in the l3experimental bundle).

1 Creating and initialising coffins

`\coffin_new:N`

`\coffin_new:c`

New: 2011-08-17

`\coffin_new:N` $\langle coffin \rangle$

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

`\coffin_clear:N`

`\coffin_clear:c`

New: 2011-08-17

`\coffin_clear:N` $\langle coffin \rangle$

Clears the content of the $\langle coffin \rangle$ within the current TeX group level.

`\coffin_set_eq:NN`

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

New: 2011-08-17

`\coffin_set_eq:NN` $\langle coffin_1 \rangle$ $\langle coffin_2 \rangle$

Sets both the content and poles of $\langle coffin_1 \rangle$ equal to those of $\langle coffin_2 \rangle$ within the current TeX group level.

`\coffin_if_exist_p:N` ★

`\coffin_if_exist_p:c` ★

`\coffin_if_exist:NTF` ★

`\coffin_if_exist:cTF` ★

New: 2012-06-20

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

`\coffin_if_exist:NTF` $\langle box \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests whether the $\langle coffin \rangle$ is currently defined.

2 Setting coffin content and poles

All coffin functions create and manipulate coffins locally within the current TeX group level.

`\hcoffin_set:Nn`

`\hcoffin_set:cn`

New: 2011-08-17

Updated: 2011-09-03

`\hcoffin_set:Nn` $\langle coffin \rangle$ $\{\langle material \rangle\}$

Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.

<hr/> <code>\hcoffin_set:Nw</code> <code>\hcoffin_set:cw</code> <code>\hcoffin_set_end:</code> <hr/> New: 2011-09-10	<code>\hcoffin_set:Nw <coffin> <material> \hcoffin_set_end:</code> Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.
<hr/> <code>\vcoffin_set:Nnn</code> <code>\vcoffin_set:cnn</code> <hr/> New: 2011-08-17 Updated: 2012-05-22	<code>\vcoffin_set:Nnn <coffin> {\width} {\material}</code> Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.
<hr/> <code>\vcoffin_set:Nnw</code> <code>\vcoffin_set:cnw</code> <code>\vcoffin_set_end:</code> <hr/> New: 2011-09-10 Updated: 2012-05-22	<code>\vcoffin_set:Nnw <coffin> {\width} <material> \vcoffin_set_end:</code> Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.
<hr/> <code>\coffin_set_horizontal_pole:Nnn</code> <code>\coffin_set_horizontal_pole:cnn</code> <hr/> New: 2012-07-20	<code>\coffin_set_horizontal_pole:Nnn <coffin> {\pole} {\offset}</code> Sets the $\langle pole \rangle$ to run horizontally through the $\langle coffin \rangle$. The $\langle pole \rangle$ will be located at the $\langle offset \rangle$ from the bottom edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.
<hr/> <code>\coffin_set_vertical_pole:Nnn</code> <code>\coffin_set_vertical_pole:cnn</code> <hr/> New: 2012-07-20	<code>\coffin_set_vertical_pole:Nnn <coffin> {\pole} {\offset}</code> Sets the $\langle pole \rangle$ to run vertically through the $\langle coffin \rangle$. The $\langle pole \rangle$ will be located at the $\langle offset \rangle$ from the left-hand edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.

3 Joining and using coffins

<hr/>	<hr/>
<code>\coffin_attach:NnnNnnnn</code>	<code>\coffin_attach:NnnNnnnn</code>
<code>\coffin_attach:(cnnNnnnn Nnnnnnn cnnnnnn)</code>	<code>\coffin_attach:(cnnNnnnn Nnnnnnn cnnnnnn)</code>
	<code>\coffin_1 \{ \coffin_1-pole_1 \} \{ \coffin_1-pole_2 \}</code>
	<code>\coffin_2 \{ \coffin_2-pole_1 \} \{ \coffin_2-pole_2 \}</code>
	<code>\{ \langle x-offset \rangle \} \{ \langle y-offset \rangle \}</code>

This function attaches $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ is not altered, *i.e.* $\langle coffin_2 \rangle$ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

<hr/>	<hr/>
<code>\coffin_join:NnnNnnnn</code>	<code>\coffin_join:NnnNnnnn</code>
<code>\coffin_join:(cnnNnnnn Nnnnnnn cnnnnnn)</code>	<code>\coffin_join:(cnnNnnnn Nnnnnnn cnnnnnn)</code>
	<code>\coffin_1 \{ \coffin_1-pole_1 \} \{ \coffin_1-pole_2 \}</code>
	<code>\coffin_2 \{ \coffin_2-pole_1 \} \{ \coffin_2-pole_2 \}</code>
	<code>\{ \langle x-offset \rangle \} \{ \langle y-offset \rangle \}</code>

This function joins $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ may expand. The new bounding box will cover the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

<hr/>	<hr/>
<code>\coffin_typeset:Nnnnn</code>	<code>\coffin_typeset:Nnnnn \langle coffin \rangle \{ \langle pole_1 \rangle \} \{ \langle pole_2 \rangle \}</code>
<code>\coffin_typeset:cnnnn</code>	<code>\{ \langle x-offset \rangle \} \{ \langle y-offset \rangle \}</code>
<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> $\langle\textit{coffin}\rangle$ $\{\langle\textit{color}\rangle\}$
<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 $\langle\textit{color}\rangle$ specified.
Updated: 2011-09-02 <hr/>	

<hr/> <code>\coffin_mark_handle:Nnnn</code> <hr/>	<code>\coffin_mark_handle:Nnnn</code> $\langle\textit{coffin}\rangle$ $\{\langle\textit{pole}_1\rangle\}$ $\{\langle\textit{pole}_2\rangle\}$ $\{\langle\textit{color}\rangle\}$
<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 $\langle\textit{color}\rangle$ 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	

Part XVIII

The l3color package

Color support

This module provides support for color in L^AT_EX3. At present, the material here is mainly intended to support a small number of low-level requirements in other l3kernel modules.

1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

```
\color_group_begin:  
\color_group_end:
```

New: 2011-09-03

```
\color_group_begin:  
...  
\color_group_end:
```

Creates a color group: one used to “trap” color settings.

```
\color_ensure_current:
```

New: 2011-09-03

```
\color_ensure_current:
```

Ensures that material inside a box will use the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin: ... \color_group_end: group`.

Part XIX

The l3msg package

Messages

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

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

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

1 Creating new messages

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

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

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

will allow only those messages from the `submodule` to be filtered out.

```
\msg_new:nnnn
```

```
\msg_new:nnn
```

Updated: 2011-08-16

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

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

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

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

Sets up the text for a *<message>* for a given *<module>*. The message 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.

```
\msg_if_exist_p:nn ★
\msg_if_exist:nnTF ★
```

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

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

New: 2012-03-03

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

2 Contextual information for messages

```
\msg_line_context: ★
```

```
\msg_line_context:
```

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

```
\msg_line_number: ★
```

```
\msg_line_number:
```

Prints the current line number when a message is given.

```
\msg_fatal_text:n ★
```

```
\msg_fatal_text:n {<module>}
```

Produces the standard text

Fatal *<module>* **error**

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *<module>* to be included.

```
\msg_critical_text:n ★
```

```
\msg_critical_text:n {<module>}
```

Produces the standard text

Critical *<module>* **error**

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *<module>* to be included.

```
\msg_error_text:n ★
```

```
\msg_error_text:n {<module>}
```

Produces the standard text

<module> **error**

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *<module>* to be included.

<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 T_EX 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`.

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

Updated: 2012-08-11

```
\msg_none:nnnnnn {<module>} {<message>} {<arg one>}
{<arg two>} {<arg three>} {<arg four>}
```

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

4 Redirecting messages

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

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

to define a message, with

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

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

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

to turn all errors into warnings, or with

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

to alter only messages from that module, or even

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

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

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

```
\msg_redirect_class:nn
```

Updated: 2012-04-27

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

Changes the behaviour of messages of $\langle class one \rangle$ so that they are processed using the code for those of $\langle class two \rangle$.

<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.

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.

`\c__msg_coding_error_text_tl`

The text

This is a coding error.

used by kernel functions when erroneous programming input is encountered.

Part XX

The l3keys package

Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

```
\MyModuleSetup{
  key-one = value one,
  key-two = value two
}
```

or

```
\MyModuleMacro[
  key-one = value one,
  key-two = value two
]{argument}
```

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { mymodule }
{
  key-one .code:n    = code including parameter #1,
  key-two .tl_set:N = \l_mymodule_store_tl
}
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }
{
  key-one = value one,
  key-two = value two
}
```

At a document level, `\keys_set:nn` will be used within a document function, for example

```
\DeclareDocumentCommand \MyModuleSetup { m }
{ \keys_set:nn { mymodule } { #1 } }
\DeclareDocumentCommand \MyModuleMacro { o m }
{
```

```

\group_begin:
  \keys_set:nn { mymodule } { #1 }
  % Main code for \MyModuleMacro
\group_end:
}

```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As will be discussed in section 2, it is suggested that the character `/` is reserved for sub-division of keys into logical groups. Functions and variables are *not* expanded when creating key names, and so

```

\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
{
  \l_mymodule_tmp_tl .code:n = code
}

```

will create a key called `\l_mymodule_tmp_tl`, and not one called `key`.

1 Creating keys

```

\keys_define:nn \keys_define:nn {<module>} {<keyval list>}

```

Parses the *<keyval list>* and defines the keys listed there for *<module>*. The *<module>* name should be a text value, but there are no restrictions on the nature of the text. In practice the *<module>* should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The *<keyval list>* should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```

\keys_define:nn { mymodule }
{
  keyname .code:n = Some~code~using~#1,
  keyname .value_required:
}

```

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 $\langle code \rangle$, $\backslash l_keys_choice_tl$ will be the name of the choice made, and $\backslash l_keys_choice_int$ 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>	Creates a <i><default></i> value for <i><key></i> , which is used if no value is given. This will be used if only the key name is given, but not if a blank <i><value></i> is given:
<code>.default:o</code>	
<code>.default:x</code> <hr/>	
Updated: 2013-07-09 <hr/>	<pre> \keys_define:nn { mymodule } { key .code:n = Hello~#1, key .default:n = World } \keys_set:nn { mymodule } { key = Fred, % Prints 'Hello Fred' key, % Prints 'Hello World' key = , % Prints 'Hello ' } </pre>
<hr/> <code>.dim_set:N</code> <hr/>	<code><key> .dim_set:N = <dimension></code>
<code>.dim_set:c</code>	Defines <i><key></i> to set <i><dimension></i> to <i><value></i> (which must a dimension expression). If the variable does not exist, it will be created globally at the point that the key is set up.
<code>.dim_gset:N</code>	
<code>.dim_gset:c</code> <hr/>	
<hr/> <code>.fp_set:N</code> <hr/>	<code><key> .fp_set:N = <floating point></code>
<code>.fp_set:c</code>	Defines <i><key></i> to set <i><floating point></i> to <i><value></i> (which must a floating point expression). If the variable does not exist, it will be created globally at the point that the key is set up.
<code>.fp_gset:N</code>	
<code>.fp_gset:c</code> <hr/>	
<hr/> <code>.groups:n</code> <hr/>	<code><key> .groups:n = {<groups>}</code>
New: 2013-07-14 <hr/>	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>	Initialises the <i><key></i> with the <i><value></i> , equivalent to
<code>.initial:o</code>	
<code>.initial:x</code> <hr/>	
Updated: 2013-07-09 <hr/>	<code>\keys_set:nn {<module>} { <key> = <value> }</code>
<hr/> <code>.int_set:N</code> <hr/>	<code><key> .int_set:N = <integer></code>
<code>.int_set:c</code>	Defines <i><key></i> to set <i><integer></i> to <i><value></i> (which must be an integer expression). If the variable does not exist, it will be created globally at the point that the key is set up.
<code>.int_gset:N</code>	
<code>.int_gset:c</code> <hr/>	

<hr/> <code>.meta:n</code> <hr/>	<code><key> .meta:n = {<keyval list>}</code>
Updated: 2013-07-10	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

Updated: 2014-04-27

In some cases, the desired behavior is to simply ignore unknown keys, collecting up information on these for later processing. The `\keys_set_known:nnN` function parses the `<keyval list>`, and sets those keys which are defined for `<module>`. Any keys which are unknown are not processed further by the parser. The key–value pairs for each *unknown* key name will be stored in the `<tl>` in a comma-separated form (*i.e.* an edited version of the `<keyval list>`). The `\keys_set_known:nn` version skips this stage.

Use of `\keys_set_known:nnN` can be nested, with the correct residual `<keyval list>` returned at each stage.

6 Selective key setting

In some cases it may be useful to be able to select only some keys for setting, even though these keys have the same path. For example, with a set of keys defined using

```
\keys define:nn { mymodule }
{
  key-one   .code:n   = { \my_func:n {#1} } ,
  key-two   .tl_set:N = \l_my_a_tl           ,
  key-three .tl_set:N = \l_my_b_tl           ,
  key-four  .fp_set:N = \l_my_a_fp           ,
}
```

the use of `\keys_set:nn` will attempt to set all four keys. However, in some contexts it may only be sensible to set some keys, or to control the order of setting. To do this, keys may be assigned to *groups*: arbitrary sets which are independent of the key tree. Thus modifying the example to read

```
\keys define:nn { mymodule }
{
  key-one   .code:n   = { \my_func:n {#1} } ,
  key-one   .groups:n = { first }           ,
  key-two   .tl_set:N = \l_my_a_tl           ,
  key-two   .groups:n = { first }           ,
  key-three .tl_set:N = \l_my_b_tl           ,
  key-three .groups:n = { second }          ,
  key-four  .fp_set:N = \l_my_a_fp           ,
}
```

will assign `key-one` and `key-two` to group `first`, `key-three` to group `second`, while `key-four` is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made “active”, or by marking one or more groups to be ignored in key setting.

<code>\keys_set_filter:nnnN</code> <code>\keys_set_filter:(nnVN nnvN nnoN nnn nnV nnv nno)</code>	<code>\keys_set_filter:nnnN {<module>} {<groups>} {<keyval</code> <code>list>} <tl></code>
--	---

New: 2013-07-14
Updated: 2014-04-27

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

Use of `\keys_set_filter:nnnN` can be nested, with the correct residual *<keyval list>* returned at each stage.

```
\keys_set_groups:nnn
\keys_set_groups:(nnV|nnv|nno)
```

New: 2013-07-14

```
\keys_set_groups:nnn {<module>} {<groups>} {<keyval list>}
```

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

7 Utility functions for keys

```
\keys_if_exist_p:nn ★
\keys_if_exist:nnTF ★
```

```
\keys_if_exist_p:nn {<module>} {<key>}
```

```
\keys_if_exist:nnTF {<module>} {<key>} {<true code>} {<false code>}
```

Tests if the $\langle key \rangle$ exists for $\langle module \rangle$, *i.e.* if any code has been defined for $\langle key \rangle$.

```
\keys_if_choice_exist_p:nnn ★
\keys_if_choice_exist:nnnTF ★
```

New: 2011-08-21

```
\keys_if_choice_exist_p:nnn {<module>} {<key>} {<choice>}
```

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

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

```
\keys_show:nn
```

```
\keys_show:nn {<module>} {<key>}
```

Shows the function which is used to actually implement a $\langle key \rangle$ for a $\langle module \rangle$.

8 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

```
KeyOne = ValueOne ,
KeyTwo = ValueTwo ,
KeyThree
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

While the functions described earlier are used as a high-level interface for processing such input, in special circumstances you may wish to use a lower-level approach. The low-level parsing system converts a $\langle key\text{--}value\text{ list} \rangle$ into $\langle keys \rangle$ and associated $\langle values \rangle$. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and

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

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

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

`\keyval_parse:NNn`

Updated: 2011-09-08

`\keyval_parse:NNn` $\langle function_1 \rangle$ $\langle function_2 \rangle$ $\{ \langle key-value list \rangle \}$

Parses the $\langle key-value list \rangle$ into a series of $\langle keys \rangle$ and associated $\langle values \rangle$, or keys alone (if no $\langle value \rangle$ was given). $\langle function_1 \rangle$ should take one argument, while $\langle function_2 \rangle$ should absorb two arguments. After `\keyval_parse:NNn` has parsed the $\langle key-value list \rangle$, $\langle function_1 \rangle$ will be used to process keys given with no value and $\langle function_2 \rangle$ will be used to process keys given with a value. The order of the $\langle keys \rangle$ in the $\langle key-value list \rangle$ will be preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

will be converted into an input stream

```
\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the $\langle key \rangle$ and $\langle value \rangle$, then one *outer* set of braces is removed from the $\langle key \rangle$ and $\langle value \rangle$ as part of the processing.

Part XXI

The l3file package

File and I/O operations

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

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

For functions which expect a $\langle file\ name \rangle$ argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Any active characters (as declared in `\l_char_active_seq`) will *not* be expanded, allowing the direct use of these in file names. File names will be quoted using `"` tokens if they contain spaces: as a result, `"` tokens are *not* permitted in file names.

1 File operation functions

<hr/> <code>\g_file_current_name_tl</code> <hr/>	Contains the name of the current <code>L^AT_EX</code> file. This variable should not be modified: it is intended for information only. It will be equal to <code>\c_job_name_tl</code> at the start of a <code>L^AT_EX</code> run and will be modified each time a file is read using <code>\file_input:n</code> .
<hr/> <code>\file_if_exist:nTF</code> <hr/> <div>Updated: 2012-02-10</div>	<code>\file_if_exist:nTF {$\langle file\ name \rangle$} {$\langle true\ code \rangle$} {$\langle false\ code \rangle$}</code> Searches for $\langle file\ name \rangle$ using the current <code>T_EX</code> search path and the additional paths controlled by <code>\file_path_include:n</code> .
<hr/> <code>\file_add_path:nN</code> <hr/> <div>Updated: 2012-02-10</div>	<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>	<code>\file_input:n {$\langle file\ name \rangle$}</code> Searches for $\langle file\ name \rangle$ in the path as detailed for <code>\file_if_exist:nTF</code> , and if found reads in the file as additional <code>L^AT_EX</code> source. All files read are recorded for information and the file name stack is updated by this function. An error will be raised if the file is not found.

<hr/> <code>\file_path_include:n</code> <hr/>	<code>\file_path_include:n {<path>}</code>
Updated: 2012-07-04	Adds $\langle path \rangle$ to the list of those used to search when reading files. The assignment is local. The $\langle path \rangle$ is processed in the same way as a $\langle file\ name \rangle$, <i>i.e.</i> , with <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>
<hr/> <code>\iow_new:N</code> <hr/>	
<code>\iow_new:c</code>	Globally reserves the name of the $\langle stream \rangle$, either for reading or for writing as appropriate. The $\langle stream \rangle$ is not opened until the appropriate <code>\..._open:Nn</code> function is used. Attempting to use a $\langle stream \rangle$ which has not been opened is an error, and the $\langle stream \rangle$ will behave as the corresponding <code>\c_term_...</code>
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|cn|cx)**

Updated: 2012-06-05

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

This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ immediately (*i.e.* the write operation is called on expansion of **\iow_now:Nn**).

\iow_log:n**\iow_log:x****\iow_log:n** $\{\langle tokens \rangle\}$

This function writes the given $\langle tokens \rangle$ to the log (transcript) file immediately: it is a dedicated version of **\iow_now:Nn**.

\iow_term:n**\iow_term:x****\iow_term:n** $\{\langle tokens \rangle\}$

This function writes the given $\langle tokens \rangle$ to the terminal file immediately: it is a dedicated version of **\iow_now:Nn**.

\iow_shipout:Nn**\iow_shipout:(Nx|cn|cx)****\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|cn|cx)`

Updated: 2012-09-08

`\iow_shipout_x:Nn <stream> {<tokens>}`

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:

`\iow_now:Nx \g_my_iow { \iow_char:N \{ text \iow_char:N \} }`

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

`\iow_newline:` ★ `\iow_newline:`

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

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_T_EX systems.

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

2.2 Constant input–output streams

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

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

2.3 Primitive conditionals

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

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

2.4 Internal file functions and variables

<hr/> <code>\g__file_internal_ior</code> <hr/>	Used to test for the existence of files when opening.
<hr/> <code>\l__file_internal_name_tl</code> <hr/>	Used to return the full name of a file for internal use. This is set by <code>\file_if_exist:n(TF)</code> and <code>__file_if_exist:nT</code> , and the value may then be used to load a file directly provided no further operations intervene.
<hr/> <code>__file_name_sanitize:nn</code> <hr/>	<code>__file_name_sanitize:nn {<name>} {<tokens>}</code>
<hr/> <small>New: 2012-02-09</small> <hr/>	Exhaustively-expands the <code><name></code> with the exception of any category <code><active></code> (catcode 13) tokens, which are not expanded. The list of <code><active></code> tokens is taken from <code>\l_char_active_seq</code> . The <code><sanitized name></code> is then inserted (in braces) after the <code><tokens></code> , which should further process the file name. If any spaces are found in the name after expansion, an error is raised.

2.5 Internal input–output functions

`__ior_open:Nn` `__ior_open:Nn` $\langle stream \rangle$ $\{\langle file\ name \rangle\}$

`__ior_open:No`

New: 2012-01-23

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

Part XXII

The l3fp package: floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. Floating point expressions support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x * y$, division x / y , square root \sqrt{x} , and parentheses.
- Comparison operators: $x < y$, $x \leq y$, $x > y$, $x \neq y$ etc.
- Boolean logic: negation $!x$, conjunction $x \&\& y$, disjunction $x || y$, ternary operator $x ? y : z$.
- Exponentials: $\exp x$, $\ln x$, x^y .
- Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$ expecting their arguments in radians, and $\text{sin}d\ x$, $\text{cos}d\ x$, $\text{tan}d\ x$, $\text{cot}d\ x$, $\text{sec}d\ x$, $\text{csc}d\ x$ expecting their arguments in degrees.
- Inverse trigonometric functions: $\text{asin}\ x$, $\text{acos}\ x$, $\text{atan}\ x$, $\text{acot}\ x$, $\text{asec}\ x$, $\text{acsc}\ x$ giving a result in radians, and $\text{asind}\ x$, $\text{acosd}\ x$, $\text{atand}\ x$, $\text{acotd}\ x$, $\text{asecd}\ x$, $\text{acscd}\ x$ giving a result in degrees.
- (not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\text{sech}\ x$, $\text{csch}\ x$, and $\text{asinh}\ x$, $\text{acosh}\ x$, $\text{atanh}\ x$, $\text{acoth}\ x$, $\text{asech}\ x$, $\text{acsch}\ x$.
- Extrema: $\max(x, y, \dots)$, $\min(x, y, \dots)$, $\text{abs}(x)$.
- Rounding functions: $\text{round}(x, n)$ rounds to closest, $\text{trunc}(x, n)$ rounds towards zero, $\text{floor}(x, n)$ rounds towards $-\infty$, $\text{ceil}(x, n)$ rounds towards $+\infty$. 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 +0.
<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 *<fp var₁>* equal to the current value of *<fp var₂>*.

```
\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 *<floating point expression>* to the *<fp var>*.

```
\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 *<floating point expression>* from the *<fp var>*.

3 Using floating point numbers

```
\fp_eval:n ★
```

New: 2012-05-08

Updated: 2012-07-08

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

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

```
\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 *<floating point expression>* and expresses the result as a dimension (in **pt**) suitable for use in dimension expressions. The output is identical to `\fp_to_decimal:n`, with an additional trailing **pt**. In particular, the result may be outside the range $[-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]$ of valid T_EX 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 <fp var></code>
<code>\fp_to_int:(c n)</code> ★	<code>\fp_to_int:n {\floating point expression}</code>
Updated: 2012-07-08	Evaluates the <i><floating point expression></i> , and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range $[-2^{31} + 1, 2^{31} - 1]$ of valid T _E X integers, leading to overflow errors if used in an integer expression. The values $\pm\infty$ and NaN trigger an “invalid operation” exception.

<code>\fp_to_scientific:N</code> ★	<code>\fp_to_scientific:N <fp var></code>
<code>\fp_to_scientific:(c n)</code> ★	<code>\fp_to_scientific:n {\floating point expression}</code>
New: 2012-05-08 Updated: 2012-07-08	Evaluates the <i><floating point expression></i> and expresses the result in scientific notation: <i><optional -><digit>.<15 digits>e<optional sign><exponent></i>
	The leading <i><digit></i> 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 <fp var></code>
<code>\fp_to_tl:(c n)</code> ★	<code>\fp_to_tl:n {\floating point expression}</code>
Updated: 2012-07-08	Evaluates the <i><floating point expression></i> and expresses the result in (almost) the shortest possible form. Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (see <code>\fp_to_scientific:n</code>). Numbers in the range $[10^{-3}, 10^{16})$ are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see <code>\fp_to_decimal:n</code>). Negative numbers start with <code>-</code> . The special values ± 0 , $\pm\infty$ and NaN are rendered as <code>0</code> , <code>-0</code> , <code>inf</code> , <code>-inf</code> , and <code>nan</code> respectively.

<code>\fp_use:N</code> ★	<code>\fp_use:N <fp var></code>
<code>\fp_use:c</code> ★	Inserts the value of the <i><fp var></i> into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an “invalid operation” exception. This function is identical to <code>\fp_to_decimal:N</code> .
Updated: 2012-07-08	

4 Floating point conditionals

<code>\fp_if_exist_p:N</code> ★	<code>\fp_if_exist_p:N <fp var></code>
<code>\fp_if_exist_p:c</code> ★	<code>\fp_if_exist:NTF <fp var> {\true code} {\false code}</code>
<code>\fp_if_exist:NTF</code> ★	Tests whether the <i><fp var></i> is currently defined. This does not check that the <i><fp var></i> really is a floating point variable.
<code>\fp_if_exist:cTF</code> ★	
Updated: 2012-05-08	

<code>\fp_compare_p:nNn</code> ★	<code>\fp_compare_p:nNn {<fpexpr₁>} <relation> {<fpexpr₂>}</code>
<code>\fp_compare:nNnTF</code> ★	<code>\fp_compare:nNnTF {<fpexpr₁>} <relation> {<fpexpr₂>} {<true code>} {<false code>}</code>
Updated: 2012-05-08	Compares the $\langle fpexpr_1 \rangle$ and the $\langle fpexpr_2 \rangle$, and returns true if the $\langle relation \rangle$ is obeyed. Two floating point numbers x and y may obey four mutually exclusive relations: $x \langle y, x=y, x \rangle y$, or x and y are not ordered. The latter case occurs exactly when either operand is NaN, and this relation is denoted by the symbol ?. Note that a NaN is distinct from any value, even another NaN, hence $x = x$ is not true for a NaN. To test if a value is NaN, compare it to an arbitrary number with the “not ordered” relation.

```

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

```

<code>\fp_compare_p:n</code> ★	<code>\fp_compare_p:n</code>
<code>\fp_compare:nTF</code> ★	<code>{</code>
Updated: 2012-12-14	<code> <fpexpr₁> <relation₁></code>
	<code> ...</code>
	<code> <fpexpr_N> <relation_N></code>
	<code> <fpexpr_{N+1}></code>
	<code>}</code>
	<code>\fp_compare:nTF</code>
	<code>{</code>
	<code> <fpexpr₁> <relation₁></code>
	<code> ...</code>
	<code> <fpexpr_N> <relation_N></code>
	<code> <fpexpr_{N+1}></code>
	<code>}</code>
	<code>{<true code>} {<false code>}</code>

Evaluates the $\langle floating\ point\ expressions \rangle$ as described for `\fp_eval:n` and compares consecutive result using the corresponding $\langle relation \rangle$, namely it compares $\langle intexpr_1 \rangle$ and $\langle intexpr_2 \rangle$ using the $\langle relation_1 \rangle$, then $\langle intexpr_2 \rangle$ and $\langle intexpr_3 \rangle$ using the $\langle relation_2 \rangle$, until finally comparing $\langle intexpr_N \rangle$ and $\langle intexpr_{N+1} \rangle$ using the $\langle relation_N \rangle$. The test yields **true** if all comparisons are **true**. Each $\langle floating\ point\ expression \rangle$ is evaluated only once. Contrarily to `\int_compare:nTF`, all $\langle floating\ point\ expressions \rangle$ are computed, even if one comparison is **false**. Two floating point numbers x and y may obey four mutually exclusive relations: $x \langle y, x=y, x \rangle y$, or x and y are not ordered. The latter case occurs exactly when one of the operands is NaN, and this relation is denoted by the symbol ?. Each $\langle relation \rangle$ can be any (non-empty) combination of $<$, $=$, $>$, and $?$, plus an optional leading ! (which negates the $\langle relation \rangle$), with the restriction that the $\langle relation \rangle$ may not start with $?$, as this symbol has a different meaning (in combination with $:$) within floatin point expressions. The comparison $x \langle relation \rangle y$ is then **true** if the $\langle relation \rangle$ does not start with ! and the actual relation ($<$, $=$, $>$, or $?$) between x and y appears within the $\langle relation \rangle$, or on the contrary if the $\langle relation \rangle$ starts with ! and the relation between x and y does not appear within the $\langle relation \rangle$. Common choices of $\langle relation \rangle$ include $>=$ (greater or equal), $!=$ (not equal), $!?$ or $<=>$ (comparable).

5 Floating point expression loops

<code>\fp_do_until:nNnn</code> ☆	<code>\fp_do_until:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
New: 2012-08-16	Places the <code><code></code> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> . If the test is <code>false</code> then the <code><code></code> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is <code>true</code> .
<code>\fp_do_while:nNnn</code> ☆	<code>\fp_do_while:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
New: 2012-08-16	Places the <code><code></code> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> . If the test is <code>true</code> then the <code><code></code> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is <code>false</code> .
<code>\fp_until_do:nNnn</code> ☆	<code>\fp_until_do:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <code><code></code> in the input stream if the <i><relation></i> is <code>false</code> . After the <code><code></code> has been processed by T _E X the test will be repeated, and a loop will occur until the test is <code>true</code> .
<code>\fp_while_do:nNnn</code> ☆	<code>\fp_while_do:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <code><code></code> in the input stream if the <i><relation></i> is <code>true</code> . After the <code><code></code> has been processed by T _E X the test will be repeated, and a loop will occur until the test is <code>false</code> .
<code>\fp_do_until:nn</code> ☆	<code>\fp_do_until:nn { <fpexpr₁> <relation> <fpexpr₂> } {<code>}</code>
New: 2012-08-16	Places the <code><code></code> 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 <code>false</code> then the <code><code></code> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is <code>true</code> .
<code>\fp_do_while:nn</code> ☆	<code>\fp_do_while:nn { <fpexpr₁> <relation> <fpexpr₂> } {<code>}</code>
New: 2012-08-16	Places the <code><code></code> 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 <code>true</code> then the <code><code></code> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is <code>false</code> .
<code>\fp_until_do:nn</code> ☆	<code>\fp_until_do:nn { <fpexpr₁> <relation> <fpexpr₂> } {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <code><code></code> in the input stream if the <i><relation></i> is <code>false</code> . After the <code><code></code> has been processed by T _E X the test will be repeated, and a loop will occur until the test is <code>true</code> .

<hr/> <code>\fp_while_do:nn</code> ☆ <hr/>	<code>\fp_while_do:nn { <fpexpr₁> <relation> <fpexpr₂> } {<code>}</code>
<hr/> New: 2012-08-16 <hr/>	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.
<hr/> New: 2012-05-08 <hr/>	
<hr/> <code>\c_one_fp</code> <hr/>	One as an fp : useful for comparisons in some places.
<hr/> New: 2012-05-08 <hr/>	
<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 inf and -inf .
<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 pi .
<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 deg .
<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^{9999} . The IEEE standard defines 5 types of exceptions.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm\infty$.
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ± 0 .
- *Invalid operation* occurs for operations with no defined outcome, for instance $0/0$, or $\sin(\infty)$, and almost any operation involving a NaN. This normally results in a NaN, except for conversion functions whose target type does not have a notion of NaN (e.g., `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating e.g., $\ln(0)$ or $\cot(0)$. This results in $\pm\infty$.
- *Inexact* occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in L^AT_EX3.

To each exception is associated a “flag”, which can be either *on* or *off*. By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions only raise the corresponding flag. The state of the flag can be tested and modified. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and turn the flag on, or only turn the flag on, or do nothing at all.

<code>\fp_if_flag_on_p:n</code> ★	<code>\fp_if_flag_on_p:n {<exception>}</code>
<code>\fp_if_flag_on:nTF</code> ★	<code>\fp_if_flag_on:nTF {<exception>} {<true code>} {<false code>}</code>
New: 2012-08-08	Tests if the flag for the <code><exception></code> is on, which normally means the given <code><exception></code> has occurred. <i>This function is experimental, and may be altered or removed.</i>
<code>\fp_flag_off:n</code>	<code>\fp_flag_off:n {<exception>}</code>
New: 2012-08-08	Locally turns off the flag which indicates whether the <code><exception></code> has occurred. <i>This function is experimental, and may be altered or removed.</i>
<code>\fp_flag_on:n</code> ★	<code>\fp_flag_on:n {<exception>}</code>
New: 2012-08-08	Locally turns on the flag to indicate (or pretend) that the <code><exception></code> has occurred. Note that this function is expandable: it is used internally by l3fp to signal when exceptions do occur. <i>This function is experimental, and may be altered or removed.</i>

<hr/> <code>\fp_trap:nn</code> <hr/>	<code>\fp_trap:nn {⟨exception⟩} {⟨trap type⟩}</code>
New: 2012-07-19 Updated: 2012-08-08	All occurrences of the <i>⟨exception⟩</i> (<code>invalid_operation</code> , <code>division_by_zero</code> , <code>overflow</code> , or <code>underflow</code>) within the current group are treated as <i>⟨trap type⟩</i> , which can be <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 \TeX 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

<hr/> <code>\fp_show:N</code> <code>\fp_show:(c n)</code> <hr/>	<code>\fp_show:N ⟨fp var⟩</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:

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

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

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

The result is thus ± 0 if and only if $\langle significand \rangle$ contains no non-zero digit (*i.e.*, consists only in 0 characters, and an optional . character), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to +0, but that is not guaranteed to remain true.

Special numbers are input as follows:

- **inf** represents $+\infty$, and can be preceded by any $\langle sign \rangle$, yielding $\pm\infty$ as appropriate.
- **nan** represents a (quiet) non-number. It can be preceded by any sign, but that will be ignored.
- Any unrecognizable string triggers an error, and produces a NaN.

Note that **e-1** is not a representation of 10^{-1} , because it could be mistaken with the difference of “e” and 1. This is consistent with several other programming languages. However, in order to avoid confusions, **e-1** is not considered to be this difference either. To input the base of natural logarithms, use **exp(1)** or **\c_e_fp**.

9.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (**sin**, **ln**, *etc*).
- Binary ****** and **^** (right associative).
- Unary **+**, **-**, **!**.
- Binary *****, **/**, and implicit multiplication by juxtaposition (**2pi**, **3(4+5)**, *etc*).
- Binary **+** and **-**.
- Comparisons **>=**, **!=**, **<?**, *etc*.
- Logical **and**, denoted by **&&**.
- Logical **or**, denoted by **||**.
- Ternary operator **?:** (right associative).

The precedence of operations can be overridden using parentheses. In particular, those precedences imply that

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

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 $\langle \text{operand}_2 \rangle$ if $\langle \text{operand}_1 \rangle$ is true, and $\langle \text{operand}_3 \rangle$ if it is false (equal to ± 0). All three $\langle \text{operands} \rangle$ 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 $\langle \text{operand}_1 \rangle$ is true (non-zero), use that value, otherwise the value of $\langle \text{operand}_2 \rangle$. Both $\langle \text{operands} \rangle$ are evaluated in all cases.

&& `\fp_eval:n { <operand1> && <operand2> }`

If $\langle \text{operand}_1 \rangle$ is false (equal to ± 0), use that value, otherwise the value of $\langle \text{operand}_2 \rangle$. Both $\langle \text{operands} \rangle$ are evaluated in all cases.

```

<      \fp_eval:n
=      {
>      \langle operand_1 \rangle \langle relation_1 \rangle
?      ...
Updated: 2013-12-14 \langle operand_N \rangle \langle relation_N \rangle
                    \langle operand_{N+1} \rangle
                    }

```

Each $\langle relation \rangle$ consists of a non-empty string of $<$, $=$, $>$, and $?$, optionally preceded by $!$, and may not start with $?$. This evaluates to $+1$ if all comparisons $\langle operand_i \rangle \langle relation_j \rangle$ are true, and $+0$ otherwise. All $\langle operands \rangle$ are evaluated in all cases. See `\fp_compare:nTF` for details.

```

-
+ \fp_eval:n { \langle operand_1 \rangle + \langle operand_2 \rangle }
- \fp_eval:n { \langle operand_1 \rangle - \langle operand_2 \rangle }
-

```

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

```

-
* \fp_eval:n { \langle operand_1 \rangle * \langle operand_2 \rangle }
/ \fp_eval:n { \langle operand_1 \rangle / \langle operand_2 \rangle }
-

```

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

```

-
+ \fp_eval:n { + \langle operand \rangle }
- \fp_eval:n { - \langle operand \rangle }
! \fp_eval:n { ! \langle operand \rangle }
-

```

The unary $+$ does nothing, the unary $-$ changes the sign of the $\langle operand \rangle$, and $!$ $\langle operand \rangle$ evaluates to 1 if $\langle operand \rangle$ is false and 0 otherwise (this is the `not` boolean function). Those operations never raise exceptions.

```

--
** \fp_eval:n { \langle operand_1 \rangle ** \langle operand_2 \rangle }
^ \fp_eval:n { \langle operand_1 \rangle ^ \langle operand_2 \rangle }
--

```

Raises $\langle operand_1 \rangle$ to the power $\langle operand_2 \rangle$. This operation is right associative, hence $2 ** 2 ** 3$ equals $2^{2^3} = 256$. The “invalid operation” exception occurs if $\langle operand_1 \rangle$ is negative or -0 , and $\langle operand_2 \rangle$ is not an integer, unless the result is zero (in that case, the sign is chosen arbitrarily to be $+0$). “Division by zero” occurs when raising ± 0 to a strictly negative power. “Underflow” and “overflow” occur when appropriate.

```

abs \fp_eval:n { abs( \langle fpexpr \rangle ) }

```

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

```

exp \fp_eval:n { exp( \langle fpexpr \rangle ) }

```

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

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

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

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

round	<code>\fp_eval:n { round (<fpexpr>) }</code>
trunc	<code>\fp_eval:n { round (<fpexpr₁> , <fpexpr₂>) }</code>

Evaluates $\langle fpexpr_1 \rangle = x$ and $\langle fpexpr_2 \rangle = n$, then rounds x to n places. If n is an integer, this rounds x to a multiple of 10^{-n} ; if $n = +\infty$, this always yields x ; if $n = -\infty$, this yields one of ± 0 , $\pm\infty$, or NaN; if n is neither $\pm\infty$ nor an integer, then an “invalid operation” exception is raised. When $\langle fpexpr_2 \rangle$ is omitted, $n = 0$, *i.e.*, $\langle fpexpr_1 \rangle$ is rounded to an integer. The rounding direction depends on the function:

- **round** yields the multiple of 10^{-n} closest to x , and if x is half-way between two such multiples, the even multiple is chosen (“ties to even”);
- **floor**, or the deprecated **round-**, yields the largest multiple of 10^{-n} smaller or equal to x (“round towards negative infinity”);
- **ceil**, or the deprecated **round+**, yields the smallest multiple of 10^{-n} greater or equal to x (“round towards positive infinity”);
- **trunc**, or the deprecated **round0**, yields a multiple of 10^{-n} with the same sign as x and with the largest absolute value less than that of x (“round towards zero”).

“Overflow” occurs if x is finite and the result is infinite (this can only happen if $\langle fpexpr_2 \rangle < -9984$).

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

Updated: 2013-11-17

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

<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.

atan	<code>\fp_eval:n { atan(<fpexpr>) }</code>
acot	<code>\fp_eval:n { atan(<fpexpr₁> , <fpexpr₂>) }</code>
	<code>\fp_eval:n { acot(<fpexpr>) }</code>
New: 2013-11-02	<code>\fp_eval:n { acot(<fpexpr₁> , <fpexpr₂>) }</code>

Those functions yield an angle in radians: **atand** and **acotd** are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-\pi/2, \pi/2]$, and arccotangent in the range $[0, \pi]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by π depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by π . Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm \pi/4, \pm 3\pi/4\}$ depending on signs. Only the “underflow” exception can occur.

atand	<code>\fp_eval:n { atand(<fpexpr>) }</code>
acotd	<code>\fp_eval:n { atand(<fpexpr₁> , <fpexpr₂>) }</code>
	<code>\fp_eval:n { acotd(<fpexpr>) }</code>
New: 2013-11-02	<code>\fp_eval:n { acotd(<fpexpr₁> , <fpexpr₂>) }</code>

Those functions yield an angle in degrees: **atand** and **acotd** are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-90, 90]$, and arccotangent in the range $[0, 180]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by 180 depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm 45, \pm 135\}$ depending on signs. Only the “underflow” exception can occur.

sqrt	<code>\fp_eval:n { sqrt(<fpexpr>) }</code>
-------------	--

New: 2013-12-14 Computes the square root of the $\langle fpexpr \rangle$. The “invalid operation” is raised when the $\langle fpexpr \rangle$ is negative; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\text{NaN}} = \text{NaN}$.

inf The special values $+\infty$, $-\infty$, and NaN are represented as **inf**, **-inf** and **nan** (see `\c_-inf_fp`, `\c_minus_inf_fp` and `\c_nan_fp`).

pi The value of π (see `\c_pi_fp`).

deg The value of 1° in radians (see `\c_one_degree_fp`).

—	
em	Those units of measurement are equal to their values in pt, namely
ex	
in	$1\text{in} = 72.27\text{pt}$
pt	$1\text{pt} = 1\text{pt}$
pc	$1\text{pc} = 12\text{pt}$
cm	
mm	$1\text{cm} = \frac{1}{2.54}\text{in} = 28.45275590551181\text{pt}$
dd	
cc	$1\text{mm} = \frac{1}{25.4}\text{in} = 2.845275590551181\text{pt}$
nd	
nc	$1\text{dd} = 0.376065\text{mm} = 1.07000856496063\text{pt}$
bp	$1\text{cc} = 12\text{dd} = 12.84010277952756\text{pt}$
sp	$1\text{nd} = 0.375\text{mm} = 1.066978346456693\text{pt}$
—	
	$1\text{nc} = 12\text{nd} = 12.80374015748031\text{pt}$
	$1\text{bp} = \frac{1}{72}\text{in} = 1.00375\text{pt}$
	$1\text{sp} = 2^{-16}\text{pt} = 1.52587890625e - 5\text{pt}.$

The values of the (font-dependent) units `em` and `ex` are gathered from \TeX when the surrounding floating point expression is evaluated.

—	
true	Other names for 1 and +0.
false	
—	
<hr/>	
<code>\fp_abs:n</code> ★	<code>\fp_abs:n {⟨floating point expression⟩}</code>
New: 2012-05-14	Evaluates the <i>⟨floating point expression⟩</i> as described for <code>\fp_eval:n</code> and leaves the
Updated: 2012-07-08	absolute value of the result in the input stream. This function does not raise any exception
<hr/>	beyond those raised when evaluating its argument. Within floating point expressions,
	<code>abs()</code> can be used.
<hr/>	
<code>\fp_max:nn</code> ★	<code>\fp_max:nn {⟨fp expression 1⟩} {⟨fp expression 2⟩}</code>
<code>\fp_min:nn</code> ★	Evaluates the <i>⟨floating point expressions⟩</i> as described for <code>\fp_eval:n</code> and leaves the
New: 2012-09-26	resulting larger (<code>max</code>) or smaller (<code>min</code>) value in the input stream. This function does not
<hr/>	raise any exception beyond those raised when evaluating its argument. Within floating
	point expressions, <code>max()</code> and <code>min()</code> 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.

- Decide what exponent range to consider.

- Support signalling `nan`.
- Modulo and remainder, and rounding functions `quantize`, `quantize0`, `quantize+`, `quantize-`, `quantize=`, `round=`. Should the modulo also be provided as (catcode 12) `%`?
- `\fp_format:nn {<fpepr>} {<format>}`, but what should `<format>` be? More general pretty printing?
- Add `and`, `or`, `xor`? Perhaps under the names `all`, `any`, and `xor`?
- Add `log(x, b)` for logarithm of x in base b .
- `hypot` (Euclidean length). Cartesian-to-polar transform.
- Hyperbolic functions `cosh`, `sinh`, `tanh`.
- Inverse hyperbolics.
- Base conversion, input such as `0xAB.CDEF`.
- Random numbers (`pgfmath` provides `rnd`, `rand`, `random`), with seed reset at every `\fp_set:Nn`.
- Factorial (not with `!`), gamma function.
- Improve coefficients of the `sin` and `tan` series.
- Treat upper and lower case letters identically in identifiers, and ignore underscores.
- Add an `array(1,2,3)` and `i=complex(0,1)`.
- Provide an experimental `map` function? Perhaps easier to implement if it is a single character, `@sin(1,2)`?
- Provide `\fp_if_nan:nTF`, and an `isnan` function?
- Support keyword arguments?

`Pgfmath` also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs. (Exclamation points mark important bugs.)

- Check that functions are monotonic when they should.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- Logarithms of numbers very close to 1 are inaccurate.
- When rounding towards $-\infty$, `\dim_to_fp:n {Opt}` should return -0 , not $+0$.
- The result of $(\pm 0) + (\pm 0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.

- `0e9999999999` gives a T_EX “number too large” error.
- Subnormals are not implemented.
- The overflow trap receives the wrong argument in `l3fp-expo` (see `exp(1e5678)` in `m3fp-traps001`).

Possible optimizations/improvements.

- Document that `l3trial/l3fp-types` introduces tools for adding new types.
- In subsection 9.1, write a grammar.
- Fix the `TWO BARS` business with the index.
- It would be nice if the `parse` auxiliaries for each operation were set up in the corresponding module, rather than centralizing in `l3fp-parse`.
- Some functions should get an `_o` ending to indicate that they expand after their result.
- More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.
- The code for the `ternary` set of functions is ugly.
- There are many `~` missing in the doc to avoid bad line-breaks.
- The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking $c = 2000/(\lfloor 200x \rfloor + 1) \in [10, 95]$ instead of $c \in [1, 10]$. Also, it would then be possible to simplify the computation of t . However, we would then have to hard-code the logarithms of 44 small integers instead of 9.
- Improve notations in the explanations of the division algorithm (`l3fp-basics`).
- Understand and document `_fp_basics_pack_weird_low:NNNNw` and `_fp_basics_pack_weird_high:NNNNNNNNw` better. Move the other `basics_pack` auxiliaries to `l3fp-aux` under a better name.
- Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.
- Add bibliography. Some of Kahan’s articles, some previous T_EX fp packages, the international standards,...
- Also take into account the “inexact” exception?
- Support multi-character prefix operators (*e.g.*, `@/` or whatever)? Perhaps for including comments inside the computation itself??

Part XXIII

The l3candidates package

Experimental additions to l3kernel

1 Important notice

This module provides a space in which functions can be added to l3kernel (expl3) while still being experimental.

As such, the functions here may not remain in their current form, or indeed at all, in l3kernel in the future.

In contrast to the material in l3experimental, the functions here are all *small* additions to the kernel. We encourage programmers to test them out and report back on the LaTeX-L mailing list.

Thus, if you intend to use any of these functions from the candidate module in a public package offered to others for productive use (e.g., being placed on CTAN) please consider the following points carefully:

- Be prepared that your public packages might require updating when such functions are being finalized.
- Consider informing us that you use a particular function in your public package, e.g., by discussing this on the LaTeX-L mailing list. This way it becomes easier to coordinate any updates necessary without a issues for the users of your package.
- Discussing and understanding use cases for a particular addition or concept also helps to ensure that we provide the right interfaces in the final version so please give us feedback if you consider a certain candidate function useful (or not).

We only add functions in this space if we consider them being serious candidates for a final inclusion into the kernel. However, real use sometimes leads to better ideas, so functions from this module are **not necessarily stable** and we may have to adjust them!

2 Additions to 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.

<code>\box_resize:Nnn</code>	<code>\box_resize:Nnn <box> {<x-size>} {<y-size>}</code>
<code>\box_resize:cnn</code>	

Resize the $\langle box \rangle$ to $\langle x-size \rangle$ horizontally and $\langle y-size \rangle$ vertically (both of the sizes are dimension expressions). The $\langle y-size \rangle$ is the vertical size (height plus depth) of the box. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus negative y -sizes will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current \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. Thus negative y -sizes will result in a box with depth dependent on the height of the original box and height dependent on the depth of the original. The resizing applies within the current \TeX group level.

<code>\box_resize_to_ht:Nn</code>	<code>\box_resize_to_ht:Nn <box> {<y-size>}</code>
<code>\box_resize_to_ht:cn</code>	

Resize the $\langle box \rangle$ to $\langle y-size \rangle$ vertically, scaling the horizontal size by the same amount ($\langle y-size \rangle$ is a dimension expression). The $\langle y-size \rangle$ is the height only, not including depth, of the box. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative size will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus negative y -sizes will result in a box with depth dependent on the height of the original box and height dependent on the depth of the original. The resizing applies within the current \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. Thus negative y -sizes will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current \TeX group level.

<code>\box_resize_to_wd_and_ht:Nnn</code>	<code>\box_resize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}</code>
<code>\box_resize_to_wd_and_ht:cnn</code>	

New: 2014-07-03

Resize the $\langle box \rangle$ to a *height* of $\langle x-size \rangle$ horizontally and $\langle y-size \rangle$ vertically (both of the sizes are dimension expressions). The $\langle y-size \rangle$ is the *height* of the box, ignoring any depth. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged.

<code>\box_rotate:Nn</code>	<code>\box_rotate:Nn <box> {<angle>}</code>
<code>\box_rotate:cn</code>	

Rotates the $\langle box \rangle$ by $\langle angle \rangle$ (in degrees) anti-clockwise about its reference point. The reference point of the updated box will be moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the rotation is applied. The rotation applies within the current \TeX group level.

<code>\box_scale:Nnn</code>	<code>\box_scale:Nnn <box> {<x-scale>} {<y-scale>}</code>
<code>\box_scale:cnn</code>	

Scales the $\langle box \rangle$ by factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the scaling is applied. Negative scalings will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus negative y -scales will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current \TeX group level.

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 $\text{\LaTeX}3$ native drivers: they will not work with the $\text{\LaTeX}2_{\epsilon}$ graphics drivers!

\TeX hackers note: Clipping is implemented by the driver, and as such the full content of the box is placed in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

<code>\box_trim:Nnnnn</code>	<code>\box_trim:Nnnnn <box> {\left}&{\bottom}&{\right}&{\top}</code>
<code>\box_trim:cnnnn</code>	

Adjusts the bounding box of the `<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 l3coffins

<code>\coffin_resize:Nnn</code>	<code>\coffin_resize:Nnn <coffin> {\width}&{\total-height}</code>
<code>\coffin_resize:cnn</code>	

Resized the `<coffin>` to `<width>` and `<total-height>`, both of which should be given as dimension expressions.

<code>\coffin_rotate:Nn</code>	<code>\coffin_rotate:Nn <coffin> {<angle>}</code>
<code>\coffin_rotate:cn</code>	

Rotates the *<coffin>* by the given *<angle>* (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.

<code>\coffin_scale:Nnn</code>	<code>\coffin_scale:Nnn <coffin> {<x-scale>} {<y-scale>}</code>
<code>\coffin_scale:cnn</code>	

Scales the *<coffin>* by a factors *<x-scale>* and *<y-scale>* in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

4 Additions to l3file

<code>\file_if_exist_input:nTF</code>	<code>\file_if_exist_input:n {<file name>}</code>
	<code>\file_if_exist_input:nTF {<file name>} {<true code>} {<false code>}</code>

New: 2014-07-02

Searches for *<file name>* using the current T_EX search path and the additional paths controlled by `\file_path_include:n`. If found, inserts the *<true code>* then reads in the file as additional L^AT_EX source as described for `\file_input:n`. Note that `\file_if_exist_input:n` does not raise an error if the file is not found, in contrast to `\file_input:n`.

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

New: 2012-02-11

Applies the *<inline function>* to *<lines>* obtained by reading one or more lines (until an equal number of left and right braces are found) from the *<stream>*. The *<inline function>* should consist of code which will receive the *<line>* as #1. Note that 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.

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

New: 2012-02-11

Applies the *<inline function>* to every *<line>* in the *<stream>*. The material is read from the *<stream>* as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The *<inline function>* should consist of code which will receive the *<line>* as #1. Note that 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.

<hr/> <code>\ior_map_break:</code> <hr/>	<code>\ior_map_break:</code>
<hr/> New: 2012-06-29 <hr/>	Used to terminate a <code>\ior_map_...</code> 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.

<hr/> <code>\ior_map_break:n</code> <hr/>	<code>\ior_map_break:n {<tokens>}</code>
<hr/> New: 2012-06-29 <hr/>	Used to terminate a <code>\ior_map_...</code> function before all lines in the $\langle stream \rangle$ have been processed, inserting the $\langle tokens \rangle$ after the mapping has ended. This will normally take place within a conditional statement, for example

```

\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 $\langle tokens \rangle$ are inserted into the input stream. This will depend on the design of the mapping function.

5 Additions to l3prop

`\prop_map_tokens:Nn` ☆
`\prop_map_tokens:cn` ☆

`\prop_map_tokens:Nn` $\langle property\ list \rangle$ $\{ \langle code \rangle \}$

Analogue of `\prop_map_function:NN` which maps several tokens instead of a single function. The $\langle code \rangle$ receives each key–value pair in the $\langle property\ list \rangle$ as two trailing brace groups. For instance,

`\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }`

will expand to the value corresponding to `mykey`: for each pair in `\l_my_prop` the function `\str_if_eq:nnT` receives `mykey`, the $\langle key \rangle$ and the $\langle value \rangle$ as its three arguments. For that specific task, `\prop_item:Nn` is faster.

6 Additions to l3seq

`\seq_mapthread_function:NNN` ☆
`\seq_mapthread_function:(NcN|cNN|ccN)` ☆

`\seq_mapthread_function:NNN` $\langle seq_1 \rangle$ $\langle seq_2 \rangle$ $\langle function \rangle$

Applies $\langle function \rangle$ to every pair of items $\langle seq_1\text{-}item \rangle$ – $\langle seq_2\text{-}item \rangle$ from the two sequences, returning items from both sequences from left to right. The $\langle function \rangle$ will receive two n-type arguments for each iteration. The mapping will terminate when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

`\seq_set_filter:NNn`
`\seq_gset_filter:NNn`

`\seq_set_filter:NNn` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\{ \langle inline\ boolexpr \rangle \}$

Evaluates the $\langle inline\ boolexpr \rangle$ for every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline\ boolexpr \rangle$ will receive the $\langle item \rangle$ as #1. The sequence of all $\langle items \rangle$ for which the $\langle inline\ boolexpr \rangle$ evaluated to `true` is assigned to $\langle sequence_1 \rangle$.

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level T_EX errors.

`\seq_set_map:NNn`
`\seq_gset_map:NNn`

New: 2011–12–22

`\seq_set_map:NNn` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\{ \langle inline\ function \rangle \}$

Applies $\langle inline\ function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline\ function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. The sequence resulting from x-expanding $\langle inline\ function \rangle$ applied to each $\langle item \rangle$ is assigned to $\langle sequence_1 \rangle$. As such, the code in $\langle inline\ function \rangle$ should be expandable.

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level T_EX errors.

7 Additions to l3skip

<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.

8 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>*.

```

\tl_expandable_uppercase:n ☆ \tl_expandable_uppercase:n {\tokens}
\tl_expandable_lowercase:n ☆ \tl_expandable_lowercase:n {\tokens}

```

The `\tl_expandable_uppercase:n` function works through all of the $\langle tokens \rangle$, 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.

```

\tl_lower_case:n ☆ \tl_upper_case:n {\tokens}
\tl_lower_case:nn ☆ \tl_upper_case:nn {\language} {\tokens}
\tl_upper_case:n ☆
\tl_upper_case:nn ☆
\tl_mixed_case:n ☆
\tl_mixed_case:nn ☆

```

New: 2014-06-30

These functions are intended to be applied to input which may be regarded broadly as “text”. They traverse the $\langle tokens \rangle$ and change the case of characters as discussed below. The character code of the characters replaced may be arbitrary: the replacement characters will have stand document-level category codes (11 for letters, 12 for letter-like characters which can also be case-changed).

The functions are x-type expandable: tokens are returned protected from further expansion where appropriate. Begin-group and end-group characters in the $\langle tokens \rangle$ are normalized and become { and }, respectively. Any tokens within such a group will *not* be case-changed, and thus for example

```
\tl_upper_case:n { Some~text~{\$y = mx + c\$}~with~{Protection} }
```

will become

```
SOME~TEXT~{\$y = mx + c\$}~WITH~{Protection}
```

‘Mixed’ case conversion may be regarded informally as converting the first character of the $\langle tokens \rangle$ to upper case and the rest to lower case. However, the process is more complex than this as there are some cases where a single lower case character maps to a special form, for example ij in Dutch which becomes IJ. As such, `\tl_mixed_case:n(n)` implement a more sophisticated mapping which accounts for this and for modifying accents on the first letter. Spaces at the start of the $\langle tokens \rangle$ are ignored when finding the first “letter” for conversion, while a brace group will terminate this search. For example

```

\tl_mixed_case:n { hello~WORLD } % => "Hello world"
\tl_mixed_case:n { ~hello~WORLD } % => " Hello world"
\tl_mixed_case:n { {hello}~WORLD } % => "{hello} world"

```

where the brace group is retained. (Note that the Unicode Consortium describe this as ‘title case’, but that in English title case applies on a word-by-word basis. The ‘mixed’

case implemented here is a lower level concept needed for both ‘title’ and ‘sentence’ casing of text.)

As is generally true for `expl3`, these functions are designed to work with engine-native input only. As such, when used with `pdfTeX` *only* the characters `a–zA–Z` are modified. When used with `XYTeX` or `LuaTeX` a full range of Unicode transformations are enabled. Specifically, the standard mappings here follow those defined by the [Unicode Consortium](#) in `UnicodeData.txt` and `SpecialCasing.txt`. Note that in some cases, `pdfTeX` can interpret the input to a case change but not generate the correct output (for example in the mapping `i` to `I-dot` in Turkish): in these cases the input is left unchanged.

Context-sensitive mappings are enabled: language-dependent cases are discussed below. The “final sigma” rule for Greek letters is enabled and active for all inputs. It is implemented here in a modified form which takes account of the requirements of the likely real use cases, performance and expandability. Thus a capital sigma will map to a final-sigma if it is followed by a space or one of the characters: `!'),.,:;?[]}`. (Feedback on this area is very welcome.)

Language-sensitive conversions are enabled using the `<language>` argument, and follow Unicode Consortium guidelines. Currently, the languages recognised for special handling are as follows.

- Azeri and Turkish (`az` and `tr`). The case pairs `I/i-dotless` and `I-dot/i` are activated for these languages. The combining dot mark is removed when lower casing `I-dot` and introduced when upper casing `i-dotless`.
- Lithuanian (`lt`). The lower case letters `i` and `j` should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lower casing of the relevant upper case letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when upper casing in these cases. Note that *only* the accents used in Lithuanian are covered: the behaviour of other accents are not modified.
- Dutch (`nl`). Capitalisation of `ij` at the beginning of mixed cased input produces `IJ` rather than `Ij`. The output retains two separate letters, thus this transformation *is* available using `pdfTeX`.

Creating additional context-sensitive mappings requires knowledge of the underlying mapping implementation used here. The team are happy to add these to the kernel where they are well-documented (*e.g.* in Unicode Consortium or relevant government publications).

```
\tl_set_from_file:Nnn
\tl_set_from_file:cnn
\tl_gset_from_file:Nnn
\tl_gset_from_file:cnn
```

```
\tl_set_from_file:Nnn <tl> {\setup} {\filename}
```

Defines `<tl>` to the contents of `<filename>`. Category codes may need to be set appropriately via the `<setup>` argument.

New: 2014-06-25

```

\tl_set_from_file_x:Nnn
\tl_set_from_file_x:cnn
\tl_gset_from_file_x:Nnn
\tl_gset_from_file_x:cnn

```

New: 2014-06-25

```
\tl_set_from_file_x:Nnn <tl> {\<setup>} {\<filename>}
```

Defines $\langle tl \rangle$ to the contents of $\langle filename \rangle$, expanding the contents of the file as it is read. Category codes and other definitions may need to be set appropriately via the $\langle setup \rangle$ argument.

9 Additions to l3tokens

```

\char_set_active:Npn
\char_set_active:Npx

```

```
\char_set_active:Npn <char> <parameters> {\<code>}
```

Makes $\langle char \rangle$ an active character 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. The $\langle char \rangle$ is made active within the current T_EX group level, and the definition is also local.

```

\char_gset_active:Npn
\char_gset_active:Npx

```

```
\char_gset_active:Npn <char> <parameters> {\<code>}
```

Makes $\langle char \rangle$ an active character 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. The $\langle char \rangle$ 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 $\langle char \rangle$ is again made active).

```
\char_set_active_eq:NN
```

```
\char_set_active_eq:NN <char> <function>
```

Makes $\langle char \rangle$ an active character equivalent in meaning to the $\langle function \rangle$ (which may itself be an active character). The $\langle char \rangle$ is made active within the current T_EX group level, and the definition is also local.

```
\char_gset_active_eq:NN
```

```
\char_gset_active_eq:NN <char> <function>
```

Makes $\langle char \rangle$ an active character equivalent in meaning to the $\langle function \rangle$ (which may itself be an active character). The $\langle char \rangle$ 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 $\langle char \rangle$ is again made active).

```
\peek_N_type:TF
```

Updated: 2012-12-20

```
\peek_N_type:TF {\<true code>} {\<false code>}
```

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 $\backslash c_space_token$, the test will take the $\langle false \rangle$ branch, even though the next $\langle token \rangle$ is in fact a valid N-type argument. The $\langle token \rangle$ will be left in the input stream after the $\langle true code \rangle$ or $\langle false code \rangle$ (as appropriate to the result of the test).

Part XXIV

The l3drivers package

Drivers

T_EX relies on drivers in order to carry out a number of tasks, such as using color, including graphics and setting up hyper-links. The nature of the code required depends on the exact driver in use. Currently, L^AT_EX3 is aware of the following drivers:

- **pdfmode**: The “driver” for direct PDF output by *both* pdfT_EX and LuaT_EX (no separate driver is used in this case: the engine deals with PDF creation itself).
- **dvips**: The dvips program, which works in conjugation with pdfT_EX or LuaT_EX in DVI mode.
- **dvipdfmx**: The dvipdfmx program, which works in conjugation with pdfT_EX or LuaT_EX in DVI mode.
- **xdvipdfmx**: The driver used by X_YT_EX.

The code here is all very low-level, and should not in general be used outside of the kernel. It is also important to note that many of the functions here are closely tied to the immediate level “up”: several variable values must be in the correct locations for the driver code to function.

1 Box clipping

`_driver_box_use_clip:N`

New: 2011-11-11

`_driver_box_use_clip:N <box>`

Inserts the content of the `<box>` at the current insertion point such that any material outside of the bounding box will not be displayed by the driver. The material in the `<box>` is still placed in the output stream: the clipping takes place at a driver level.

This function should only be used within a surrounding horizontal box construct.

2 Box rotation and scaling

<code>__driver_box_rotate_begin:</code>	<code>__driver_box_rotate_begin:</code>
<code>__driver_box_rotate_end:</code>	<code>\box_use:N \l__box_internal_box</code>
	<code>__driver_box_rotate_end:</code>

New: 2011-09-01

Updated: 2013-12-27

Rotates the $\langle box material \rangle$ anti-clockwise around the current insertion point. The angle of rotation (in degrees counter-clockwise) and the sine and cosine of this angle should be stored in `\l__box_angle_fp`, `\l__box_sin_fp` and `\l__box_cos_fp`, respectively. Typically, the box material inserted between the beginning and end markers will be stored in `\l__box_internal_box`: this fact is required by some drivers to obtain the correct output.

<code>__driver_box_scale_begin:</code>	<code>__driver_box_scale_begin:</code>
<code>__driver_box_scale_end:</code>	$\langle box material \rangle$
	<code>__driver_box_scale_end:</code>

New: 2011-09-02

Updated: 2013-12-27

Scales the $\langle box material \rangle$ (which should be either a `\box_use:N` or `\hbox:n` construct). The $\langle box material \rangle$ is scaled by the values stored in `\l__box_scale_x_fp` and `\l__box_scale_y_fp` in the horizontal and vertical directions, respectively. This function is also reused when resizing boxes: at a driver level, only scalings are supported and so the higher-level code must convert the absolute sizes to scale factors.

3 Color support

<code>__driver_color_ensure_current:</code>	<code>__driver_color_ensure_current:</code>
--	--

New: 2011-09-03

Updated: 2012-05-18

Ensures that the color used to typeset material is that which was set when the material was placed in a box. This function is therefore required inside any “color safe” box to ensure that the box may be inserted in a location where the foreground color has been altered, while preserving the color used in the box.

Part XXV

Implementation

1 l3bootstrap implementation

```

1  $\langle *initex | package \rangle$ 
2  $\langle @@=expl \rangle$ 

```

1.1 Format-specific code

The very first thing to do is to bootstrap the `iniTeX` system so that everything else will actually work. `TeX` does not start with some pretty basic character codes set up.

```
3 <*initex>
4 \catcode '\{ = 1 \relax
5 \catcode '\} = 2 \relax
6 \catcode '\# = 6 \relax
7 \catcode '\^ = 7 \relax
8 </initex>
```

Tab characters should not show up in the code, but to be on the safe side.

```
9 <*initex>
10 \catcode '\^^I = 10 \relax
11 </initex>
```

For `LuaTeX`, the extra primitives need to be enabled. This is not needed in package mode: plain `TeX` and `ConTeXt` have the primitives enabled while `LATeX 2ε` has them with the prefix `luatex` (which is handled in `l3names`).

```
12 <*initex>
13 \begingroup\expandafter\expandafter\expandafter\endgroup
14 \expandafter\ifx\csname directlua\endcsname\relax
15 \else
16 \directlua{tex.enableprimitives ('', tex.extraprimitives ( ))}
17 \fi
18 </initex>
```

1.2 The `\pdfstrcmp` primitive with `XYTeX` and `LuaTeX`

Only `pdfTeX` has a primitive called `\pdfstrcmp`. The `XYTeX` version is just `\strcmp`, so there is some shuffling to do. As this is still a real primitive, using the `pdfTeX` name is “safe”.

```
19 \begingroup\expandafter\expandafter\expandafter\endgroup
20 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
21 \let\pdfstrcmp\strcmp
22 \fi
```

If `LuaTeX` is in use then no primitive `\(pdf)strcmp` is available. However, it can be emulated using some Lua code. In earlier versions of the code, the `pdftexcmds` package was loaded to do this task. However, that raises some issues in “generic” (it fails with `ConTeXt MkIV`), and also adds a hardly-needed dependency. Note that `LuaTeX` prior to version 0.36 is not supported by `expl3`: here that means simply skipping the definition, which will then be picked up later. This definition may need to be done twice: one “now” and once at the start of every job. The latter can occur in package mode if for example a custom format is being constructed. To achieve this while not requiring a separate file, the Lua code is saved into a macro then used twice. (In the long term, the Lua code here may be best moved to a separate file.)

No macro definition is given just yet: that is left until `l3basics`.

```
23 \begingroup
```

```

24 \expandafter\ifx\csname directlua\endcsname\relax
25 \else
26   \ifnum\luatexversion<36 %
27   \else
28     \catcode'\_ =11 %
29     \catcode'\:=11 %
30     \def\tempa
31     {%
32       l3kernel = l3kernel or { }
33       function l3kernel.strcmp (A, B)
34         if A == B then
35           tex.write ("0")
36         elseif A < B then
37           tex.write ("-1")
38         else
39           tex.write ("1")
40         end
41       end
42     }
43     \directlua{\tempa}

```

A test for Lua_{TEX} in Ini_{TEX} mode.

```

44   \ifnum 0%
45   \directlua
46   {%
47     if status.ini_version then
48       tex.write("1")
49     end
50   }>0 %
51   \global\everyjob\expandafter
52   {%
53     \the\expandafter\everyjob
54     \expandafter\luatex_directlua:D\expandafter{\tempa}%
55   }
56   \fi
57   \fi
58   \fi
59 \endgroup

```

1.3 Engine requirements

The code currently requires functionality equivalent to `\pdfstrcmp` in addition to ε -_{TEX}. This is picked up by testing for the `\pdfstrcmp` primitive or a version of Lua_{TEX} capable of emulating it.

```

60 \begingroup
61 \def\next{\endgroup}
62 \def\ShortText{Required primitives not found}%
63 \def\LongText%
64   {%

```

```

65     LaTeX3 requires the e-TeX primitives and \string\pdfstrcmp.\LineBreak
66     \LineBreak
67     These are available in engine versions:\LineBreak
68     - pdfTeX 1.30\LineBreak
69     - XeTeX 0.9994\LineBreak
70     - LuaTeX 0.40\LineBreak
71     or later.\LineBreak
72     \LineBreak
73 }%
74 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
75   \expandafter\ifx\csname directlua\endcsname\relax
76   \else
77     \ifnum\luatexversion<36 %
78   \langle*initex\rangle
79     \def\LineBreak{^^J}%
80     \edef\next
81     {%
82       \newlinechar'\noexpand^^J\relax
83       \errhelp
84       {%
85         \LongText
86         For pdfTeX and XeTeX the '-etex' command-line switch is also
87         needed.\LineBreak
88         \LineBreak
89         Format building will abort!\LineBreak
90       }%
91       \errmessage{\ShortText}%
92       \endgroup
93       \noexpand\end
94     }%
95   \rangle
96   \langle*package\rangle
97     \def\LineBreak{\noexpand\MessageBreak}%
98     \expandafter\ifx\csname PackageError\endcsname\relax
99     \def\LineBreak{^^J}%
100     \begingroup
101       \def\PackageError#1#2#3%
102       {%
103         \endgroup
104         \errhelp{#3}%
105         \errmessage{#1 Error: #2!}%
106       }%
107     \fi
108     \edef\next
109     {%
110       \noexpand\PackageError{expl3}{\ShortText}
111       {\LongText Loading of expl3 will abort!}%
112     \endgroup
113     \noexpand\endinput
114   }%

```

```

115 </package>
116     \fi
117     \fi
118     \fi
119 \next

```

1.4 Extending allocators

In format mode, allocating registers is handled by `l3alloc`. However, in package mode it's much safer to rely on more general code. For example, the ability to extend \TeX 's allocation routine to allow for $\varepsilon\text{-}\text{\TeX}$ has been around since 1997 in the `etex` package.

Loading this support is delayed until here as we are now sure that the $\varepsilon\text{-}\text{\TeX}$ extensions and `\pdfstrcmp` or equivalent are available. Thus there is no danger of an “uncontrolled” error if the engine requirements are not met.

For \LaTeX only, load `etex` as otherwise we are likely to get into trouble with registers. Some inserts are reserved also as these have to be from the standard pool. Note that `\reserveinserts` is `\outer` and so is accessed here by `csname`. In earlier versions, loading `etex` was done directly and so `\reserveinserts` appeared in the code: this then required a `\relax` after `\RequirePackage` to prevent an error with “unsafe” definitions as seen for example with `capoptions`. The optional loading here is done using a group and `\ifx` test as we are not quite in the position to have a single name for `\pdfstrcmp` just yet.

```

120 <*package>
121 \begingroup
122   \def\@tempa{LaTeX2e}
123   \def\next{}
124   \ifx\fmtname\@tempa
125     \def\next
126       {%
127         \RequirePackage{etex}%
128         \csname reserveinserts\endcsname{32}%
129       }
130   \fi
131 \expandafter\endgroup
132 \next
133 </package>

```

1.5 The \LaTeX3 code environment

The code environment is now set up.

`\ExplSyntaxOff` Before changing any category codes, in package mode we need to save the situation before loading. Note the set up here means that once applied `\ExplSyntaxOff` will be a “do nothing” command until `\ExplSyntaxOn` is used. For format mode, there is no need to save category codes so that step is skipped.

```

134 \protected\def\ExplSyntaxOff{}
135 <*package>

```

```

136 \protected\edef\ExplSyntaxOff
137   {%
138     \protected\def\ExplSyntaxOff{%
139       \catcode 9 = \the\catcode 9\relax
140       \catcode 32 = \the\catcode 32\relax
141       \catcode 34 = \the\catcode 34\relax
142       \catcode 38 = \the\catcode 38\relax
143       \catcode 58 = \the\catcode 58\relax
144       \catcode 94 = \the\catcode 94\relax
145       \catcode 95 = \the\catcode 95\relax
146       \catcode 124 = \the\catcode 124\relax
147       \catcode 126 = \the\catcode 126\relax
148       \endlinechar = \the\endlinechar\relax
149       \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
150     }
151   }</package>

```

(End definition for \ExplSyntaxOff. This function is documented on page 7.)

The code environment is now set up.

```

152 \catcode 9 = 9\relax
153 \catcode 32 = 9\relax
154 \catcode 34 = 12\relax
155 \catcode 58 = 11\relax
156 \catcode 94 = 7\relax
157 \catcode 95 = 11\relax
158 \catcode 124 = 12\relax
159 \catcode 126 = 10\relax
160 \endlinechar = 32\relax

```

\l__kernel_expl_bool The status for experimental code syntax: this is on at present.

```

161 \chardef\l__kernel_expl_bool = 1 ~

```

(End definition for \l__kernel_expl_bool. This variable is documented on page 8.)

\ExplSyntaxOn The idea here is that multiple \ExplSyntaxOn calls are not going to mess up category codes, and that multiple calls to \ExplSyntaxOff are also not wasting time. Applying \ExplSyntaxOn will alter the definition of \ExplSyntaxOff and so in package mode this function should not be used until after the end of the loading process!

```

162 \protected \def \ExplSyntaxOn
163   {
164     \bool_if:NF \l__kernel_expl_bool
165     {
166       \cs_set_protected_nopar:Npx \ExplSyntaxOff
167       {
168         \char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
169         \char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
170         \char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
171         \char_set_catcode:nn { 38 } { \char_value_catcode:n { 38 } }
172         \char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
173         \char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
174         \char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }

```



```

175         \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
176         \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
177         \tex_endlinechar:D =
178         \tex_the:D \tex_endlinechar:D \scan_stop:
179         \bool_set_false:N \l__kernel_expl_bool
180         \cs_set_protected_nopar:Npn \ExplSyntaxOff { }
181     }
182 }
183 \char_set_catcode_ignore:n { 9 } % tab
184 \char_set_catcode_ignore:n { 32 } % space
185 \char_set_catcode_other:n { 34 } % double quote
186 \char_set_catcode_alignment:n { 38 } % ampersand
187 \char_set_catcode_letter:n { 58 } % colon
188 \char_set_catcode_math_superscript:n { 94 } % circumflex
189 \char_set_catcode_letter:n { 95 } % underscore
190 \char_set_catcode_other:n { 124 } % pipe
191 \char_set_catcode_space:n { 126 } % tilde
192 \tex_endlinechar:D = 32 \scan_stop:
193 \bool_set_true:N \l__kernel_expl_bool
194 }

```

(End definition for \ExplSyntaxOn. This function is documented on page 7.)

```

195 </initex | package>

```

2 l3names implementation

```

196 <*initex | package>

```

No prefix substitution here.

```

197 <@@=>

```

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.

```

198 \let \tex_global:D \global
199 \let \tex_let:D \let

```

Everything is inside a (rather long) group, which keeps `__kernel_primitive:NN` trapped.

```

200 \begingroup

```

`__kernel_primitive:NN` A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

```

201 \long \def \__kernel_primitive:NN #1#2
202 {

```

```

203     \tex_global:D \tex_let:D #2 #1
204 \*initex>
205     \tex_global:D \tex_let:D #1 \tex_undefined:D
206 \initex>
207 }

```

(End definition for _kernel_primitive:NN.)

To allow extracting “just the names”, a bit of DocStrip fiddling.

```

208 \initex | package>
209 \*initex | names | package>

```

In the current incarnation of this package, all TeX primitives are given a new name of the form `\tex_oldname:D`. But first three special cases which have symbolic original names. These are given modified new names, so that they may be entered without catcode tricks.

```

210 \_kernel_primitive:NN \           \tex_space:D
211 \_kernel_primitive:NN \/         \tex_italiccorrection:D
212 \_kernel_primitive:NN \-         \tex_hyphen:D

```

Now all the other primitives.

```

213 \_kernel_primitive:NN \let       \tex_let:D
214 \_kernel_primitive:NN \def       \tex_def:D
215 \_kernel_primitive:NN \edef      \tex_edef:D
216 \_kernel_primitive:NN \gdef      \tex_gdef:D
217 \_kernel_primitive:NN \xdef      \tex_xdef:D
218 \_kernel_primitive:NN \chardef   \tex_chardef:D
219 \_kernel_primitive:NN \countdef  \tex_countdef:D
220 \_kernel_primitive:NN \dimendef  \tex_dimendef:D
221 \_kernel_primitive:NN \skipdef   \tex_skipdef:D
222 \_kernel_primitive:NN \muskipdef \tex_muskipdef:D
223 \_kernel_primitive:NN \mathchardef \tex_mathchardef:D
224 \_kernel_primitive:NN \toksdef   \tex_toksdef:D
225 \_kernel_primitive:NN \futurelet \tex_futurelet:D
226 \_kernel_primitive:NN \advance   \tex_advance:D
227 \_kernel_primitive:NN \divide    \tex_divide:D
228 \_kernel_primitive:NN \multiply  \tex_multiply:D
229 \_kernel_primitive:NN \font      \tex_font:D
230 \_kernel_primitive:NN \fam       \tex_fam:D
231 \_kernel_primitive:NN \global    \tex_global:D
232 \_kernel_primitive:NN \long      \tex_long:D
233 \_kernel_primitive:NN \outer     \tex_outer:D
234 \_kernel_primitive:NN \setlanguage \tex_setlanguage:D
235 \_kernel_primitive:NN \globaldefs \tex_globaldefs:D
236 \_kernel_primitive:NN \afterassignment \tex_afterassignment:D
237 \_kernel_primitive:NN \aftergroup \tex_aftergroup:D
238 \_kernel_primitive:NN \expandafter \tex_expandafter:D
239 \_kernel_primitive:NN \noexpand  \tex_noexpand:D
240 \_kernel_primitive:NN \begingroup \tex_begingroup:D
241 \_kernel_primitive:NN \endgroup  \tex_endgroup:D
242 \_kernel_primitive:NN \halign    \tex_halign:D
243 \_kernel_primitive:NN \valign    \tex_valign:D

```

244	_kernel_primitive:NN \cr	\tex_cr:D
245	_kernel_primitive:NN \crr	\tex_crr:D
246	_kernel_primitive:NN \noalign	\tex_noalign:D
247	_kernel_primitive:NN \omit	\tex_omit:D
248	_kernel_primitive:NN \span	\tex_span:D
249	_kernel_primitive:NN \tabskip	\tex_tabskip:D
250	_kernel_primitive:NN \everycr	\tex_everycr:D
251	_kernel_primitive:NN \if	\tex_if:D
252	_kernel_primitive:NN \ifcase	\tex_ifcase:D
253	_kernel_primitive:NN \ifcat	\tex_ifcat:D
254	_kernel_primitive:NN \ifnum	\tex_ifnum:D
255	_kernel_primitive:NN \ifodd	\tex_ifodd:D
256	_kernel_primitive:NN \ifdim	\tex_ifdim:D
257	_kernel_primitive:NN \ifeof	\tex_ifeof:D
258	_kernel_primitive:NN \ifhbox	\tex_ifhbox:D
259	_kernel_primitive:NN \ifvbox	\tex_ifvbox:D
260	_kernel_primitive:NN \ifvoid	\tex_ifvoid:D
261	_kernel_primitive:NN \ifx	\tex_ifx:D
262	_kernel_primitive:NN \iffalse	\tex_iffalse:D
263	_kernel_primitive:NN \iftrue	\tex_iftrue:D
264	_kernel_primitive:NN \ifhmode	\tex_ifhmode:D
265	_kernel_primitive:NN \ifmmode	\tex_ifmmode:D
266	_kernel_primitive:NN \ifvmode	\tex_ifvmode:D
267	_kernel_primitive:NN \ifinner	\tex_ifinner:D
268	_kernel_primitive:NN \else	\tex_else:D
269	_kernel_primitive:NN \fi	\tex_fi:D
270	_kernel_primitive:NN \or	\tex_or:D
271	_kernel_primitive:NN \immediate	\tex_immediate:D
272	_kernel_primitive:NN \closeout	\tex_closeout:D
273	_kernel_primitive:NN \openin	\tex_openin:D
274	_kernel_primitive:NN \openout	\tex_openout:D
275	_kernel_primitive:NN \read	\tex_read:D
276	_kernel_primitive:NN \write	\tex_write:D
277	_kernel_primitive:NN \closein	\tex_closein:D
278	_kernel_primitive:NN \newlinechar	\tex_newlinechar:D
279	_kernel_primitive:NN \input	\tex_input:D
280	_kernel_primitive:NN \endinput	\tex_endinput:D
281	_kernel_primitive:NN \inputlineno	\tex_inputlineno:D
282	_kernel_primitive:NN \errmessage	\tex_errmessage:D
283	_kernel_primitive:NN \message	\tex_message:D
284	_kernel_primitive:NN \show	\tex_show:D
285	_kernel_primitive:NN \showthe	\tex_showthe:D
286	_kernel_primitive:NN \showbox	\tex_showbox:D
287	_kernel_primitive:NN \showlists	\tex_showlists:D
288	_kernel_primitive:NN \errhelp	\tex_errhelp:D
289	_kernel_primitive:NN \errorcontextlines	\tex_errorcontextlines:D
290	_kernel_primitive:NN \tracingcommands	\tex_tracingcommands:D
291	_kernel_primitive:NN \tracinglostchars	\tex_tracinglostchars:D
292	_kernel_primitive:NN \tracingmacros	\tex_tracingmacros:D
293	_kernel_primitive:NN \tracingonline	\tex_tracingonline:D

294	_kernel_primitive:NN	\tracingoutput	\tex_tracingoutput:D
295	_kernel_primitive:NN	\tracingpages	\tex_tracingpages:D
296	_kernel_primitive:NN	\tracingparagraphs	\tex_tracingparagraphs:D
297	_kernel_primitive:NN	\tracingrestores	\tex_tracingrestores:D
298	_kernel_primitive:NN	\tracingstats	\tex_tracingstats:D
299	_kernel_primitive:NN	\pausing	\tex_pausing:D
300	_kernel_primitive:NN	\showboxbreadth	\tex_showboxbreadth:D
301	_kernel_primitive:NN	\showboxdepth	\tex_showboxdepth:D
302	_kernel_primitive:NN	\batchmode	\tex_batchmode:D
303	_kernel_primitive:NN	\errorstopmode	\tex_errorstopmode:D
304	_kernel_primitive:NN	\nonstopmode	\tex_nonstopmode:D
305	_kernel_primitive:NN	\scrollmode	\tex_scrollmode:D
306	_kernel_primitive:NN	\end	\tex_end:D
307	_kernel_primitive:NN	\csname	\tex_csname:D
308	_kernel_primitive:NN	\endcsname	\tex_endcsname:D
309	_kernel_primitive:NN	\ignorespaces	\tex_ignorespaces:D
310	_kernel_primitive:NN	\relax	\tex_relax:D
311	_kernel_primitive:NN	\the	\tex_the:D
312	_kernel_primitive:NN	\mag	\tex_mag:D
313	_kernel_primitive:NN	\language	\tex_language:D
314	_kernel_primitive:NN	\mark	\tex_mark:D
315	_kernel_primitive:NN	\topmark	\tex_topmark:D
316	_kernel_primitive:NN	\firstmark	\tex_firstmark:D
317	_kernel_primitive:NN	\botmark	\tex_botmark:D
318	_kernel_primitive:NN	\splitfirstmark	\tex_splitfirstmark:D
319	_kernel_primitive:NN	\splitbotmark	\tex_splitbotmark:D
320	_kernel_primitive:NN	\fontname	\tex_fontname:D
321	_kernel_primitive:NN	\escapechar	\tex_escapechar:D
322	_kernel_primitive:NN	\endlinechar	\tex_endlinechar:D
323	_kernel_primitive:NN	\mathchoice	\tex_mathchoice:D
324	_kernel_primitive:NN	\delimiter	\tex_delimiter:D
325	_kernel_primitive:NN	\mathaccent	\tex_mathaccent:D
326	_kernel_primitive:NN	\mathchar	\tex_mathchar:D
327	_kernel_primitive:NN	\mskip	\tex_mskip:D
328	_kernel_primitive:NN	\radical	\tex_radical:D
329	_kernel_primitive:NN	\vcenter	\tex_vcenter:D
330	_kernel_primitive:NN	\mkern	\tex_mkern:D
331	_kernel_primitive:NN	\above	\tex_above:D
332	_kernel_primitive:NN	\abovewithdelims	\tex_abovewithdelims:D
333	_kernel_primitive:NN	\atop	\tex_atop:D
334	_kernel_primitive:NN	\atopwithdelims	\tex_atopwithdelims:D
335	_kernel_primitive:NN	\over	\tex_over:D
336	_kernel_primitive:NN	\overwithdelims	\tex_overwithdelims:D
337	_kernel_primitive:NN	\displaystyle	\tex_displaystyle:D
338	_kernel_primitive:NN	\textstyle	\tex_textstyle:D
339	_kernel_primitive:NN	\scriptstyle	\tex_scriptstyle:D
340	_kernel_primitive:NN	\scriptscriptstyle	\tex_scriptscriptstyle:D
341	_kernel_primitive:NN	\nonscript	\tex_nonscript:D
342	_kernel_primitive:NN	\eqno	\tex_eqno:D
343	_kernel_primitive:NN	\leqno	\tex_leqno:D

344	_kernel_primitive:NN	\abovedisplayshortskip	\tex_abovedisplayshortskip:D
345	_kernel_primitive:NN	\abovedisplayskip	\tex_abovedisplayskip:D
346	_kernel_primitive:NN	\belowdisplayshortskip	\tex_belowdisplayshortskip:D
347	_kernel_primitive:NN	\belowdisplayskip	\tex_belowdisplayskip:D
348	_kernel_primitive:NN	\displaywidowpenalty	\tex_displaywidowpenalty:D
349	_kernel_primitive:NN	\displayindent	\tex_displayindent:D
350	_kernel_primitive:NN	\displaywidth	\tex_displaywidth:D
351	_kernel_primitive:NN	\everydisplay	\tex_everydisplay:D
352	_kernel_primitive:NN	\predisplaysize	\tex_predisplaysize:D
353	_kernel_primitive:NN	\predisplaypenalty	\tex_predisplaypenalty:D
354	_kernel_primitive:NN	\postdisplaypenalty	\tex_postdisplaypenalty:D
355	_kernel_primitive:NN	\mathbin	\tex_mathbin:D
356	_kernel_primitive:NN	\mathclose	\tex_mathclose:D
357	_kernel_primitive:NN	\mathinner	\tex_mathinner:D
358	_kernel_primitive:NN	\mathop	\tex_mathop:D
359	_kernel_primitive:NN	\displaylimits	\tex_displaylimits:D
360	_kernel_primitive:NN	\limits	\tex_limits:D
361	_kernel_primitive:NN	\nolimits	\tex_nolimits:D
362	_kernel_primitive:NN	\mathopen	\tex_mathopen:D
363	_kernel_primitive:NN	\mathord	\tex_mathord:D
364	_kernel_primitive:NN	\mathpunct	\tex_mathpunct:D
365	_kernel_primitive:NN	\mathrel	\tex_mathrel:D
366	_kernel_primitive:NN	\overline	\tex_overline:D
367	_kernel_primitive:NN	\underline	\tex_underline:D
368	_kernel_primitive:NN	\left	\tex_left:D
369	_kernel_primitive:NN	\right	\tex_right:D
370	_kernel_primitive:NN	\binoppenalty	\tex_binoppenalty:D
371	_kernel_primitive:NN	\relpenalty	\tex_relpenalty:D
372	_kernel_primitive:NN	\delimitershortfall	\tex_delimitershortfall:D
373	_kernel_primitive:NN	\delimiterfactor	\tex_delimiterfactor:D
374	_kernel_primitive:NN	\nulldelimiterspace	\tex_nulldelimiterspace:D
375	_kernel_primitive:NN	\everymath	\tex_everymath:D
376	_kernel_primitive:NN	\mathsurround	\tex_mathsurround:D
377	_kernel_primitive:NN	\medmuskip	\tex_medmuskip:D
378	_kernel_primitive:NN	\thinmuskip	\tex_thinmuskip:D
379	_kernel_primitive:NN	\thickmuskip	\tex_thickmuskip:D
380	_kernel_primitive:NN	\scriptspace	\tex_scriptspace:D
381	_kernel_primitive:NN	\noboundary	\tex_noboundary:D
382	_kernel_primitive:NN	\accent	\tex_accent:D
383	_kernel_primitive:NN	\char	\tex_char:D
384	_kernel_primitive:NN	\discretionary	\tex_discretionary:D
385	_kernel_primitive:NN	\hfil	\tex_hfil:D
386	_kernel_primitive:NN	\hfilneg	\tex_hfilneg:D
387	_kernel_primitive:NN	\hfill	\tex_hfill:D
388	_kernel_primitive:NN	\hskip	\tex_hskip:D
389	_kernel_primitive:NN	\hss	\tex_hss:D
390	_kernel_primitive:NN	\vfil	\tex_vfil:D
391	_kernel_primitive:NN	\vfilneg	\tex_vfilneg:D
392	_kernel_primitive:NN	\vfill	\tex_vfill:D
393	_kernel_primitive:NN	\vskip	\tex_vskip:D

394	_kernel_primitive:NN \vss	\tex_vss:D
395	_kernel_primitive:NN \unskip	\tex_unskip:D
396	_kernel_primitive:NN \kern	\tex_kern:D
397	_kernel_primitive:NN \unkern	\tex_unkern:D
398	_kernel_primitive:NN \hrule	\tex_hrule:D
399	_kernel_primitive:NN \vrule	\tex_vrule:D
400	_kernel_primitive:NN \leaders	\tex_leaders:D
401	_kernel_primitive:NN \cleaders	\tex_cleaders:D
402	_kernel_primitive:NN \xleaders	\tex_xleaders:D
403	_kernel_primitive:NN \lastkern	\tex_lastkern:D
404	_kernel_primitive:NN \lastskip	\tex_lastskip:D
405	_kernel_primitive:NN \indent	\tex_indent:D
406	_kernel_primitive:NN \par	\tex_par:D
407	_kernel_primitive:NN \noindent	\tex_noindent:D
408	_kernel_primitive:NN \vadjust	\tex_vadjust:D
409	_kernel_primitive:NN \baselineskip	\tex_baselineskip:D
410	_kernel_primitive:NN \lineskip	\tex_lineskip:D
411	_kernel_primitive:NN \lineskiplimit	\tex_lineskiplimit:D
412	_kernel_primitive:NN \clubpenalty	\tex_clubpenalty:D
413	_kernel_primitive:NN \widowpenalty	\tex_widowpenalty:D
414	_kernel_primitive:NN \exhyphenpenalty	\tex_exhyphenpenalty:D
415	_kernel_primitive:NN \hyphenpenalty	\tex_hyphenpenalty:D
416	_kernel_primitive:NN \linepenalty	\tex_linepenalty:D
417	_kernel_primitive:NN \doublehyphendemerits	\tex_doublehyphendemerits:D
418	_kernel_primitive:NN \finalhyphendemerits	\tex_finalhyphendemerits:D
419	_kernel_primitive:NN \adjdemerits	\tex_adjdemerits:D
420	_kernel_primitive:NN \hangafter	\tex_hangafter:D
421	_kernel_primitive:NN \hangindent	\tex_hangindent:D
422	_kernel_primitive:NN \parshape	\tex_parshape:D
423	_kernel_primitive:NN \hsize	\tex_hsize:D
424	_kernel_primitive:NN \lefthyphenmin	\tex_lefthyphenmin:D
425	_kernel_primitive:NN \righthyphenmin	\tex_righthyphenmin:D
426	_kernel_primitive:NN \leftskip	\tex_leftskip:D
427	_kernel_primitive:NN \rightskip	\tex_rightskip:D
428	_kernel_primitive:NN \looseness	\tex_looseness:D
429	_kernel_primitive:NN \parskip	\tex_parskip:D
430	_kernel_primitive:NN \parindent	\tex_parindent:D
431	_kernel_primitive:NN \uchyph	\tex_uchyph:D
432	_kernel_primitive:NN \emergencystretch	\tex_emergencystretch:D
433	_kernel_primitive:NN \pretolerance	\tex_pretolerance:D
434	_kernel_primitive:NN \tolerance	\tex_tolerance:D
435	_kernel_primitive:NN \spaceskip	\tex_spaceskip:D
436	_kernel_primitive:NN \xspaceskip	\tex_xspaceskip:D
437	_kernel_primitive:NN \parfillskip	\tex_parfillskip:D
438	_kernel_primitive:NN \everypar	\tex_everypar:D
439	_kernel_primitive:NN \prevgraf	\tex_prevgraf:D
440	_kernel_primitive:NN \spacefactor	\tex_spacefactor:D
441	_kernel_primitive:NN \shipout	\tex_shipout:D
442	_kernel_primitive:NN \vsize	\tex_vsize:D
443	_kernel_primitive:NN \interlinepenalty	\tex_interlinepenalty:D

444	_kernel_primitive:NN	\brokenpenalty	\tex_brokenpenalty:D
445	_kernel_primitive:NN	\topskip	\tex_topskip:D
446	_kernel_primitive:NN	\maxdeadcycles	\tex_maxdeadcycles:D
447	_kernel_primitive:NN	\maxdepth	\tex_maxdepth:D
448	_kernel_primitive:NN	\output	\tex_output:D
449	_kernel_primitive:NN	\deadcycles	\tex_deadcycles:D
450	_kernel_primitive:NN	\pagedepth	\tex_pagedepth:D
451	_kernel_primitive:NN	\pagestretch	\tex_pagestretch:D
452	_kernel_primitive:NN	\pagefilstretch	\tex_pagefilstretch:D
453	_kernel_primitive:NN	\pagefillstretch	\tex_pagefillstretch:D
454	_kernel_primitive:NN	\pagefilllstretch	\tex_pagefilllstretch:D
455	_kernel_primitive:NN	\pageshrink	\tex_pageshrink:D
456	_kernel_primitive:NN	\pagegoal	\tex_pagegoal:D
457	_kernel_primitive:NN	\pagetotal	\tex_pagetotal:D
458	_kernel_primitive:NN	\outputpenalty	\tex_outputpenalty:D
459	_kernel_primitive:NN	\hoffset	\tex_hoffset:D
460	_kernel_primitive:NN	\voffset	\tex_voffset:D
461	_kernel_primitive:NN	\insert	\tex_insert:D
462	_kernel_primitive:NN	\holdinginserts	\tex_holdinginserts:D
463	_kernel_primitive:NN	\floatingpenalty	\tex_floatingpenalty:D
464	_kernel_primitive:NN	\insertpenalties	\tex_insertpenalties:D
465	_kernel_primitive:NN	\lower	\tex_lower:D
466	_kernel_primitive:NN	\moveleft	\tex_moveleft:D
467	_kernel_primitive:NN	\moveright	\tex_moveright:D
468	_kernel_primitive:NN	\raise	\tex_raise:D
469	_kernel_primitive:NN	\copy	\tex_copy:D
470	_kernel_primitive:NN	\lastbox	\tex_lastbox:D
471	_kernel_primitive:NN	\vsplit	\tex_vsplit:D
472	_kernel_primitive:NN	\unhbox	\tex_unhbox:D
473	_kernel_primitive:NN	\unhcopy	\tex_unhcopy:D
474	_kernel_primitive:NN	\unvbox	\tex_unvbox:D
475	_kernel_primitive:NN	\unvcopy	\tex_unvcopy:D
476	_kernel_primitive:NN	\setbox	\tex_setbox:D
477	_kernel_primitive:NN	\hbox	\tex_hbox:D
478	_kernel_primitive:NN	\vbox	\tex_vbox:D
479	_kernel_primitive:NN	\vtop	\tex_vtop:D
480	_kernel_primitive:NN	\prevdepth	\tex_prevdepth:D
481	_kernel_primitive:NN	\badness	\tex_badness:D
482	_kernel_primitive:NN	\hbadness	\tex_hbadness:D
483	_kernel_primitive:NN	\vbadness	\tex_vbadness:D
484	_kernel_primitive:NN	\hfuzz	\tex_hfuzz:D
485	_kernel_primitive:NN	\vfuzz	\tex_vfuzz:D
486	_kernel_primitive:NN	\overfullrule	\tex_overfullrule:D
487	_kernel_primitive:NN	\boxmaxdepth	\tex_boxmaxdepth:D
488	_kernel_primitive:NN	\splitmaxdepth	\tex_splitmaxdepth:D
489	_kernel_primitive:NN	\splittopskip	\tex_splittopskip:D
490	_kernel_primitive:NN	\everyhbox	\tex_everyhbox:D
491	_kernel_primitive:NN	\everyvbox	\tex_everyvbox:D
492	_kernel_primitive:NN	\nullfont	\tex_nullfont:D
493	_kernel_primitive:NN	\textfont	\tex_textfont:D

494	__kernel_primitive:NN	\scriptfont	\tex_scriptfont:D
495	__kernel_primitive:NN	\scriptscriptfont	\tex_scriptscriptfont:D
496	__kernel_primitive:NN	\fontdimen	\tex_fontdimen:D
497	__kernel_primitive:NN	\hyphenchar	\tex_hyphenchar:D
498	__kernel_primitive:NN	\skewchar	\tex_skewchar:D
499	__kernel_primitive:NN	\defaultthyphenchar	\tex_defaultthyphenchar:D
500	__kernel_primitive:NN	\defaultskewchar	\tex_defaultskewchar:D
501	__kernel_primitive:NN	\number	\tex_number:D
502	__kernel_primitive:NN	\romannumeral	\tex_romannumeral:D
503	__kernel_primitive:NN	\string	\tex_string:D
504	__kernel_primitive:NN	\lowercase	\tex_lowercase:D
505	__kernel_primitive:NN	\uppercase	\tex_uppercase:D
506	__kernel_primitive:NN	\meaning	\tex_meaning:D
507	__kernel_primitive:NN	\penalty	\tex_penalty:D
508	__kernel_primitive:NN	\unpenalty	\tex_unpenalty:D
509	__kernel_primitive:NN	\lastpenalty	\tex_lastpenalty:D
510	__kernel_primitive:NN	\special	\tex_special:D
511	__kernel_primitive:NN	\dump	\tex_dump:D
512	__kernel_primitive:NN	\patterns	\tex_patterns:D
513	__kernel_primitive:NN	\hyphenation	\tex_hyphenation:D
514	__kernel_primitive:NN	\time	\tex_time:D
515	__kernel_primitive:NN	\day	\tex_day:D
516	__kernel_primitive:NN	\month	\tex_month:D
517	__kernel_primitive:NN	\year	\tex_year:D
518	__kernel_primitive:NN	\jobname	\tex_jobname:D
519	__kernel_primitive:NN	\everyjob	\tex_everyjob:D
520	__kernel_primitive:NN	\count	\tex_count:D
521	__kernel_primitive:NN	\dimen	\tex_dimen:D
522	__kernel_primitive:NN	\skip	\tex_skip:D
523	__kernel_primitive:NN	\toks	\tex_toks:D
524	__kernel_primitive:NN	\muskip	\tex_muskip:D
525	__kernel_primitive:NN	\box	\tex_box:D
526	__kernel_primitive:NN	\wd	\tex_wd:D
527	__kernel_primitive:NN	\ht	\tex_ht:D
528	__kernel_primitive:NN	\dp	\tex_dp:D
529	__kernel_primitive:NN	\catcode	\tex_catcode:D
530	__kernel_primitive:NN	\delcode	\tex_delcode:D
531	__kernel_primitive:NN	\sfcode	\tex_sfcode:D
532	__kernel_primitive:NN	\lccode	\tex_lccode:D
533	__kernel_primitive:NN	\uccode	\tex_uccode:D
534	__kernel_primitive:NN	\mathcode	\tex_mathcode:D

Since L^AT_EX3 requires at least the ε -T_EX extensions, we also rename the additional primitives. These are all given the prefix `\etex_`.

535	__kernel_primitive:NN	\ifdefined	\etex_ifdefined:D
536	__kernel_primitive:NN	\ifcsname	\etex_ifcsname:D
537	__kernel_primitive:NN	\unless	\etex_unless:D
538	__kernel_primitive:NN	\eTeXversion	\etex_eTeXversion:D
539	__kernel_primitive:NN	\eTeXrevision	\etex_eTeXrevision:D
540	__kernel_primitive:NN	\marks	\etex_marks:D

541	_kernel_primitive:NN	\topmarks	\etex_topmarks:D
542	_kernel_primitive:NN	\firstmarks	\etex_firstmarks:D
543	_kernel_primitive:NN	\botmarks	\etex_botmarks:D
544	_kernel_primitive:NN	\splitfirstmarks	\etex_splitfirstmarks:D
545	_kernel_primitive:NN	\splitbotmarks	\etex_splitbotmarks:D
546	_kernel_primitive:NN	\unexpanded	\etex_unexpanded:D
547	_kernel_primitive:NN	\detokenize	\etex_detokenize:D
548	_kernel_primitive:NN	\scantokens	\etex_scantokens:D
549	_kernel_primitive:NN	\showtokens	\etex_showtokens:D
550	_kernel_primitive:NN	\readline	\etex_readline:D
551	_kernel_primitive:NN	\tracingassigns	\etex_tracingassigns:D
552	_kernel_primitive:NN	\tracingscantokens	\etex_tracingscantokens:D
553	_kernel_primitive:NN	\tracingnesting	\etex_tracingnesting:D
554	_kernel_primitive:NN	\tracingifs	\etex_tracingifs:D
555	_kernel_primitive:NN	\currentiflevel	\etex_currentiflevel:D
556	_kernel_primitive:NN	\currentifbranch	\etex_currentifbranch:D
557	_kernel_primitive:NN	\currentifttype	\etex_currentifttype:D
558	_kernel_primitive:NN	\tracinggroups	\etex_tracinggroups:D
559	_kernel_primitive:NN	\currentgrouplevel	\etex_currentgrouplevel:D
560	_kernel_primitive:NN	\currentgrouptype	\etex_currentgrouptype:D
561	_kernel_primitive:NN	\showgroups	\etex_showgroups:D
562	_kernel_primitive:NN	\showifs	\etex_showifs:D
563	_kernel_primitive:NN	\interactionmode	\etex_interactionmode:D
564	_kernel_primitive:NN	\lastnodetype	\etex_lastnodetype:D
565	_kernel_primitive:NN	\iffontchar	\etex_iffontchar:D
566	_kernel_primitive:NN	\fontcharht	\etex_fontcharht:D
567	_kernel_primitive:NN	\fontchardp	\etex_fontchardp:D
568	_kernel_primitive:NN	\fontcharwd	\etex_fontcharwd:D
569	_kernel_primitive:NN	\fontcharic	\etex_fontcharic:D
570	_kernel_primitive:NN	\parshapeindent	\etex_parshapeindent:D
571	_kernel_primitive:NN	\parshapelength	\etex_parshapelength:D
572	_kernel_primitive:NN	\parshapedimen	\etex_parshapedimen:D
573	_kernel_primitive:NN	\numexpr	\etex_numexpr:D
574	_kernel_primitive:NN	\dimexpr	\etex_dimexpr:D
575	_kernel_primitive:NN	\glueexpr	\etex_glueexpr:D
576	_kernel_primitive:NN	\muexpr	\etex_muexpr:D
577	_kernel_primitive:NN	\gluestretch	\etex_gluestretch:D
578	_kernel_primitive:NN	\glueshrink	\etex_glueshrink:D
579	_kernel_primitive:NN	\gluestretchorder	\etex_gluestretchorder:D
580	_kernel_primitive:NN	\glueshrinkorder	\etex_glueshrinkorder:D
581	_kernel_primitive:NN	\gluetomu	\etex_gluetomu:D
582	_kernel_primitive:NN	\mutoglue	\etex_mutoglue:D
583	_kernel_primitive:NN	\lastlinefit	\etex_lastlinefit:D
584	_kernel_primitive:NN	\interlinepenalties	\etex_interlinepenalties:D
585	_kernel_primitive:NN	\clubpenalties	\etex_clubpenalties:D
586	_kernel_primitive:NN	\widowpenalties	\etex_widowpenalties:D
587	_kernel_primitive:NN	\displaywidowpenalties	\etex_displaywidowpenalties:D
588	_kernel_primitive:NN	\middle	\etex_middle:D
589	_kernel_primitive:NN	\savinghyphcodes	\etex_savinghyphcodes:D
590	_kernel_primitive:NN	\savingvdiscards	\etex_savingvdiscards:D

591	<code>__kernel_primitive:NN \pagediscards</code>	<code>\etex_pagediscards:D</code>
592	<code>__kernel_primitive:NN \splitdiscards</code>	<code>\etex_splitdiscards:D</code>
593	<code>__kernel_primitive:NN \TeXXeTstate</code>	<code>\etex_TeXXeTstate:D</code>
594	<code>__kernel_primitive:NN \beginL</code>	<code>\etex_beginL:D</code>
595	<code>__kernel_primitive:NN \endL</code>	<code>\etex_endL:D</code>
596	<code>__kernel_primitive:NN \beginR</code>	<code>\etex_beginR:D</code>
597	<code>__kernel_primitive:NN \endR</code>	<code>\etex_endR:D</code>
598	<code>__kernel_primitive:NN \predisplaydirection</code>	<code>\etex_predisplaydirection:D</code>
599	<code>__kernel_primitive:NN \everyeof</code>	<code>\etex_everyeof:D</code>
600	<code>__kernel_primitive:NN \protected</code>	<code>\etex_protected:D</code>

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.

601	<code>__kernel_primitive:NN \pdfcreationdate</code>	<code>\pdfTEX_pdfcreationdate:D</code>
602	<code>__kernel_primitive:NN \pdfcolorstack</code>	<code>\pdfTEX_pdfcolorstack:D</code>
603	<code>__kernel_primitive:NN \pdfcompresslevel</code>	<code>\pdfTEX_pdfcompresslevel:D</code>
604	<code>__kernel_primitive:NN \pdfdecimaldigits</code>	<code>\pdfTEX_pdfdecimaldigits:D</code>
605	<code>__kernel_primitive:NN \pdfhorigin</code>	<code>\pdfTEX_pdfhorigin:D</code>
606	<code>__kernel_primitive:NN \pdfinfo</code>	<code>\pdfTEX_pdfinfo:D</code>
607	<code>__kernel_primitive:NN \pdflastxform</code>	<code>\pdfTEX_pdflastxform:D</code>
608	<code>__kernel_primitive:NN \pdfliteral</code>	<code>\pdfTEX_pdfliteral:D</code>
609	<code>__kernel_primitive:NN \pdfminorversion</code>	<code>\pdfTEX_pdfminorversion:D</code>
610	<code>__kernel_primitive:NN \pdfobjcompresslevel</code>	<code>\pdfTEX_pdfobjcompresslevel:D</code>
611	<code>__kernel_primitive:NN \pdfoutput</code>	<code>\pdfTEX_pdfoutput:D</code>
612	<code>__kernel_primitive:NN \pdfrefxform</code>	<code>\pdfTEX_pdfrefxform:D</code>
613	<code>__kernel_primitive:NN \pdfrestore</code>	<code>\pdfTEX_pdfrestore:D</code>
614	<code>__kernel_primitive:NN \pdfsave</code>	<code>\pdfTEX_pdfsave:D</code>
615	<code>__kernel_primitive:NN \pdfsetmatrix</code>	<code>\pdfTEX_pdfsetmatrix:D</code>
616	<code>__kernel_primitive:NN \pdfpkresolution</code>	<code>\pdfTEX_pdfpkresolution:D</code>
617	<code>__kernel_primitive:NN \pdfTEXrevision</code>	<code>\pdfTEX_pdfTEXrevision:D</code>
618	<code>__kernel_primitive:NN \pdfvorigin</code>	<code>\pdfTEX_pdfvorigin:D</code>
619	<code>__kernel_primitive:NN \pdfxform</code>	<code>\pdfTEX_pdfxform:D</code>

While these are not.

620	<code>__kernel_primitive:NN \pdfstrcmp</code>	<code>\pdfTEX_strcmp:D</code>
-----	--	-------------------------------

X_YTeX-specific primitives. Note that X_YTeX's `\strcmp` is handled earlier and is “rolled up” into `\pdfstrcmp`.

621	<code>__kernel_primitive:NN \XeTeXversion</code>	<code>\xetex_XeTeXversion:D</code>
-----	---	------------------------------------

Primitives from LuaTeX.

622	<code>__kernel_primitive:NN \catcodetable</code>	<code>\luaTEX_catcodetable:D</code>
623	<code>__kernel_primitive:NN \directlua</code>	<code>\luaTEX_directlua:D</code>
624	<code>__kernel_primitive:NN \expanded</code>	<code>\luaTEX_expanded:D</code>
625	<code>__kernel_primitive:NN \initcatcodetable</code>	<code>\luaTEX_initcatcodetable:D</code>
626	<code>__kernel_primitive:NN \latelua</code>	<code>\luaTEX_latelua:D</code>
627	<code>__kernel_primitive:NN \luaescapestring</code>	<code>\luaTEX_luaescapestring:D</code>
628	<code>__kernel_primitive:NN \luaTEXversion</code>	<code>\luaTEX_luaTEXversion:D</code>

```

629 \__kernel_primitive:NN \savecatcodetable \luatex_savecatcodetable:D
630 \__kernel_primitive:NN \Uchar \luatex_Uchar: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.

```

631 \__kernel_primitive:NN \bodydir \luatex_bodydir:D
632 \__kernel_primitive:NN \mathdir \luatex_mathdir:D
633 \__kernel_primitive:NN \pagedir \luatex_pagedir:D
634 \__kernel_primitive:NN \pardir \luatex_pardir:D
635 \__kernel_primitive:NN \textdir \luatex_textdir:D

```

End of the “just the names” part of the source.

```

636 </initex | names | package>
637 <*initex | package>

```

The job is done: close the group (using the primitive renamed!).

```

638 \tex_endgroup:D

```

L^AT_EX 2_ε will have moved a few primitives, so these are sorted out. A convenient test for L^AT_EX 2_ε is the \@@end saved primitive.

```

639 <*package>
640 \etex_ifdefined:D \@@end
641 \tex_let:D \tex_end:D \@@end
642 \tex_let:D \tex_everydisplay:D \frozen@everydisplay
643 \tex_let:D \tex_everymath:D \frozen@everymath
644 \tex_let:D \tex_hyphen:D \@@hyph
645 \tex_let:D \tex_input:D \@@input
646 \tex_let:D \tex_italiccorrection:D \@@italiccorr
647 \tex_let:D \tex_underline:D \@@underline

```

That is also true for the LuaTeX primitives under L^AT_EX 2_ε.

```

648 \tex_let:D \luatex_catcodetable:D \luatexcatcodetable
649 \tex_let:D \luatex_initcatcodetable:D \luatexinitcatcodetable
650 \tex_let:D \luatex_latelua:D \luatexlatelua
651 \tex_let:D \luatex_luaescapestring:D \luatexluaescapestring
652 \tex_let:D \luatex_savecatcodetable:D \luatexsavecatcodetable
653 \tex_let:D \luatex_Uchar:D \luatexUchar

```

Which also covers those slightly odd ones.

```

654 \tex_let:D \luatex_bodydir:D \luatexbodydir
655 \tex_let:D \luatex_mathdir:D \luatexmathdir
656 \tex_let:D \luatex_pagedir:D \luatexpagedir
657 \tex_let:D \luatex_pardir:D \luatexpardir
658 \tex_let:D \luatex_textdir:D \luatextextdir
659 \tex_fi:D

```

For ConT_EXt, two tests are needed. Both Mark II and Mark IV move several primitives: these are all covered by the first test, again using \end as a marker. For Mark IV, a few more primitives are moved: they are implemented using some Lua code in the current ConT_EXt.

```

660 \etex_ifdefined:D \normalend

```

```

661 \tex_let:D \tex_outer:D \normalouter
662 \tex_let:D \tex_input:D \normalinput
663 \tex_let:D \tex_end:D \normalend
664 \tex_let:D \tex_language:D \normallanguage
665 \tex_let:D \tex_vcenter:D \normalvcneter
666 \tex_let:D \tex_over:D \normalover
667 \tex_let:D \tex_mathop:D \normalmathop
668 \tex_let:D \tex_month:D \normalmonth
669 \tex_let:D \tex_everyjob:D \normaleveryjob
670 \tex_let:D \etex_unexpanded:D \normalunexpanded
671 \tex_fi:D
672 \etex_ifdefined:D \normalitaliccorrection
673 \tex_let:D \tex_hoffset:D \normalhoffset
674 \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
675 \tex_let:D \tex_voffset:D \normalvoffset
676 \tex_let:D \etex_showtokens:D \normalshowtokens
677 \tex_let:D \luatex_bodydir:D \spac_directions_normal_body_dir
678 \tex_let:D \luatex_pagedir:D \spac_directions_normal_page_dir
679 \tex_fi:D
680 \etex_ifdefined:D \normalleft
681 \tex_let:D \tex_left:D \normalleft
682 \tex_let:D \tex_middle:D \normalmiddle
683 \tex_let:D \tex_right:D \normalright
684 \tex_fi:D
685 </package>
686 </initex | package>

```

3 l3basics implementation

```

687 <*initex | package>

```

3.1 Renaming some T_EX primitives (again)

Having given all the T_EX primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but do a few now, just to get started.²

```

\if_true: Then some conditionals.
\if_false:
\or:
\else:
\fi:
\reverse_if:N
\if:w
\if_charcode:w
\if_catcode:w
\if_meaning:w

```

688 \tex_let:D \if_true:	\tex_iftrue:D
689 \tex_let:D \if_false:	\tex_iffalse:D
690 \tex_let:D \or:	\tex_or:D
691 \tex_let:D \else:	\tex_else:D
692 \tex_let:D \fi:	\tex_fi:D
693 \tex_let:D \reverse_if:N	\etex_unless:D
694 \tex_let:D \if:w	\tex_if:D
695 \tex_let:D \if_charcode:w	\tex_if:D

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.

```

696 \tex_let:D \if_catcode:w      \tex_ifcat:D
697 \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:` T_EX lets us detect some of its modes.

```

\if_mode_horizontal: 698 \tex_let:D \if_mode_math:      \tex_ifmmode:D
\if_mode_vertical:    699 \tex_let:D \if_mode_horizontal: \tex_ifhmode:D
\if_mode_inner:       700 \tex_let:D \if_mode_vertical:    \tex_ifvmode:D
                       701 \tex_let:D \if_mode_inner:      \tex_ifinner:D

```

(End definition for `\if_mode_math:` and others. These functions are documented on page 24.)

`\if_cs_exist:N` Building csnames and testing if control sequences exist.

```

\if_cs_exist:w        702 \tex_let:D \if_cs_exist:N      \etex_ifdefined:D
\cs:w                 703 \tex_let:D \if_cs_exist:w      \etex_ifcsname:D
\cs_end:              704 \tex_let:D \cs:w              \tex_csname:D
                       705 \tex_let:D \cs_end:         \tex_endcsname:D

```

(End definition for `\if_cs_exist:N` and others. These functions are documented on page 18.)

`\exp_after:wN` The three `\exp_` functions are used in the `l3expan` module where they are described.

```

\exp_not:N            706 \tex_let:D \exp_after:wN      \tex_expandafter:D
\exp_not:n            707 \tex_let:D \exp_not:N        \tex_noexpand:D
                       708 \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 32.)

`\token_to_meaning:N` Examining a control sequence or token.

```

\token_to_str:N       709 \tex_let:D \token_to_meaning:N \tex_meaning:D
\cs_meaning:N         710 \tex_let:D \token_to_str:N    \tex_string:D
                       711 \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 17.)

`\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:         712 \tex_let:D \scan_stop:        \tex_relax:D
\group_end:           713 \tex_let:D \group_begin:     \tex_begingroup:D
                       714 \tex_let:D \group_end:      \tex_endgroup:D

```

(End definition for `\scan_stop:`, `\group_begin:`, and `\group_end:`. These functions are documented on page 10.)

`\if_int_compare:w` For integers.

```

\__int_to_roman:w     715 \tex_let:D \if_int_compare:w  \tex_ifnum:D
                       716 \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 73.)

`\group_insert_after:N` Adding material after the end of a group.

```

717 \tex_let:D \group_insert_after:N \tex_aftergroup:D

```

(End definition for `\group_insert_after:N`. This function is documented on page 10.)

`\exp_args:Nc` Discussed in `l3expan`, but needed much earlier.

```

\exp_args:cc 718 \tex_long:D \tex_def:D \exp_args:Nc #1#2
              719 { \exp_after:wN #1 \cs:w #2 \cs_end: }
              720 \tex_long:D \tex_def:D \exp_args:cc #1#2
              721 { \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.

```

722 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
723 \tex_long:D \tex_def:D \cs_meaning:c #1
724 {
725   \if_cs_exist:w #1 \cs_end:
726   \exp_after:wN \use_i:nn
727   \else:
728   \exp_after:wN \use_ii:nn
729   \fi:
730   { \exp_args:Nc \cs_meaning:N {#1} }
731   { \tl_to_str:n {undefined} }
732 }
733 \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_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`.

```

734 \*package>
735 \tex_let:D \c_minus_one \m@ne
736 \*package>
737 \*initex>
738 \tex_countdef:D \c_minus_one = 10 ~
739 \c_minus_one = -1 ~
740 \*initex>
741 \tex_chardef:D \c_sixteen = 16 ~
742 \tex_chardef:D \c_zero = 0 ~
743 \tex_chardef:D \c_six = 6 ~
744 \tex_chardef:D \c_seven = 7 ~
745 \tex_chardef:D \c_twelve = 12 ~

```

(End definition for `\c_minus_one`, `\c_zero`, and `\c_sixteen`. These variables are documented on page 72.)

`\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`.

```

746 \etex_ifdefined:D \luatex luatexversion:D
747 \tex_chardef:D \c_max_register_int = 65 535 ~
748 \tex_else:D
749 \tex_mathchardef:D \c_max_register_int = 32 767 ~
750 \tex_fi:D

```

(End definition for `\c_max_register_int`. This variable is documented on page 72.)

3.3 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

`\cs_set_nopar:Npn` All assignment functions in L^AT_EX₃ should be naturally protected; after all, the T_EX primitives for assignments are and it can be a cause of problems if others aren't.

```

\cs_set_nopar:Npx
\cs_set:Npn
\cs_set:Npx
\cs_set_protected_nopar:Npn
\cs_set_protected_nopar:Npx
\cs_set_protected:Npn
\cs_set_protected:Npx
751 \tex_let:D \cs_set_nopar:Npn \tex_def:D
752 \tex_let:D \cs_set_nopar:Npx \tex_edef:D
753 \etex_protected:D \cs_set_nopar:Npn \cs_set:Npn
754 { \tex_long:D \cs_set_nopar:Npn }
755 \etex_protected:D \cs_set_nopar:Npn \cs_set:Npx
756 { \tex_long:D \cs_set_nopar:Npx }
757 \etex_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npn
758 { \etex_protected:D \cs_set_nopar:Npn }
759 \etex_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npx
760 { \etex_protected:D \cs_set_nopar:Npx }
761 \cs_set_protected_nopar:Npn \cs_set_protected:Npn
762 { \etex_protected:D \tex_long:D \cs_set_nopar:Npn }
763 \cs_set_protected_nopar:Npn \cs_set_protected:Npx
764 { \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` Global versions of the above functions.

```

\cs_gset_nopar:Npx
\cs_gset:Npn
\cs_gset:Npx
\cs_gset_protected_nopar:Npn
\cs_gset_protected_nopar:Npx
\cs_gset_protected:Npn
\cs_gset_protected:Npx
765 \tex_let:D \cs_gset_nopar:Npn \tex_gdef:D
766 \tex_let:D \cs_gset_nopar:Npx \tex_xdef:D
767 \cs_set_protected_nopar:Npn \cs_gset:Npn
768 { \tex_long:D \cs_gset_nopar:Npn }
769 \cs_set_protected_nopar:Npn \cs_gset:Npx
770 { \tex_long:D \cs_gset_nopar:Npx }
771 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npn
772 { \etex_protected:D \cs_gset_nopar:Npn }
773 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npx
774 { \etex_protected:D \cs_gset_nopar:Npx }
775 \cs_set_protected_nopar:Npn \cs_gset_protected:Npn
776 { \etex_protected:D \tex_long:D \cs_gset_nopar:Npn }
777 \cs_set_protected_nopar:Npn \cs_gset_protected:Npx
778 { \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.

```
779 \cs_set_nopar:Npn \l__exp_internal_tl { }
```

(End definition for `\l__exp_internal_tl`. This variable is documented on page 33.)

`\use:c` This macro grabs its argument and returns a `csname` from it.

```
780 \cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }
```

(End definition for `\use:c`. This function is documented on page 18.)

`\use:x` Fully expands its argument and passes it to the input stream. Uses the reserved `\l__exp_internal_tl` which will be set up in `l3expan`.

```
781 \cs_set_protected:Npn \use:x #1
782 {
783   \cs_set_nopar:Npx \l__exp_internal_tl {#1}
784   \l__exp_internal_tl
785 }
```

(End definition for `\use:x`. This function is documented on page 21.)

`\use:n` These macros grab their arguments and returns them back to the input (with outer braces removed).

```
\use:nnn 786 \cs_set:Npn \use:n #1 {#1}
\use:nnnn 787 \cs_set:Npn \use:nn #1#2 {#1#2}
788 \cs_set:Npn \use:nnn #1#2#3 {#1#2#3}
789 \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 790 \cs_set:Npn \use_i:nn #1#2 {#1}
791 \cs_set:Npn \use_ii:nn #1#2 {#2}
```

(End definition for `\use_i:nn` and `\use_ii:nn`. These functions are documented on page 20.)

`\use_i:nnnn` We also need something for picking up arguments from a longer list.

```
\use_ii:nnnn 792 \cs_set:Npn \use_i:nnnn #1#2#3 {#1}
\use_iii:nnnn 793 \cs_set:Npn \use_ii:nnnn #1#2#3 {#2}
\use_i_ii:nnnn 794 \cs_set:Npn \use_iii:nnnn #1#2#3 {#3}
\use_i:nnnnn 795 \cs_set:Npn \use_i_ii:nnnn #1#2#3 {#1#2}
\use_ii:nnnnn 796 \cs_set:Npn \use_i:nnnnn #1#2#3#4 {#1}
\use_iii:nnnnn 797 \cs_set:Npn \use_ii:nnnnn #1#2#3#4 {#2}
\use_iv:nnnnn 798 \cs_set:Npn \use_iii:nnnnn #1#2#3#4 {#3}
799 \cs_set:Npn \use_iv:nnnnn #1#2#3#4 {#4}
```

(End definition for `\use_i:nnn` and others. These functions are documented on page 20.)

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.

```
\use_none_delimit_by_q_stop:w 800 \cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
\use_none_delimit_by_q_recursion_stop:w 801 \cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
802 \cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }
```


(End definition for `\use_none_delimit_by_q_nil:w`, `\use_none_delimit_by_q_stop:w`, and `\use_none_delimit_by_q_recursion_stop:w`. These functions are documented on page 21.)

`\use_i_delimit_by_q_nil:nw`
`\use_i_delimit_by_q_stop:nw`
`\use_i_delimit_by_q_recursion_stop:nw`

Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

```
803 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
804 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
805 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw #1#2 \q_recursion_stop {#1}
```

(End definition for `\use_i_delimit_by_q_nil:nw`, `\use_i_delimit_by_q_stop:nw`, and `\use_i_delimit_by_q_recursion_stop:nw`. These functions are documented on page 21.)

3.5 Gobbling tokens from input

`\use_none:n`
`\use_none:nn`
`\use_none:nnn`
`\use_none:nnnn`
`\use_none:nnnnn`
`\use_none:nnnnnn`
`\use_none:nnnnnnn`
`\use_none:nnnnnnnn`
`\use_none:nnnnnnnnn`

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of `n`'s following the `:` in the name. Although we could define functions to remove ten arguments or more using separate calls of `\use_none:nnnnnnnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once will take care of gobbling all the tokens in one go.

```
806 \cs_set:Npn \use_none:n #1 { }
807 \cs_set:Npn \use_none:nn #1#2 { }
808 \cs_set:Npn \use_none:nnn #1#2#3 { }
809 \cs_set:Npn \use_none:nnnn #1#2#3#4 { }
810 \cs_set:Npn \use_none:nnnnn #1#2#3#4#5 { }
811 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6 { }
812 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7 { }
813 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8 { }
814 \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 T_EX 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 T_EX 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
`\prg_return_false:` 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.

```
815 \cs_set_nopar:Npn \prg_return_true:
816   { \exp_after:wN \use_i:nn \__int_to_roman:w }
817 \cs_set_nopar:Npn \prg_return_false:
818   { \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 36.)

`\prg_set_conditional:Npnn` The user functions for the types using parameter text from the programmer. The various
`\prg_new_conditional:Npnn` functions only differ by which function is used for the assignment. For those `Npnn` type
`\prg_set_protected_conditional:Npnn` functions, we must grab the parameter text, reading everything up to a left brace before
`\prg_new_protected_conditional:Npnn` continuing. Then split the base function into name and signature, and feed `{⟨name⟩}`
`__prg_generate_conditional_parm:nnNpnn` `{⟨signature⟩}` `⟨boolean⟩` `{⟨set or new⟩}` `{⟨maybe protected⟩}` `{⟨parameters⟩}` `{TF,...}`
`{⟨code⟩}` to the auxiliary function responsible for defining all conditionals.

```
819 \cs_set_protected_nopar:Npn \prg_set_conditional:Npnn
820   { \__prg_generate_conditional_parm:nnNpnn { set } { } }
821 \cs_set_protected_nopar:Npn \prg_new_conditional:Npnn
822   { \__prg_generate_conditional_parm:nnNpnn { new } { } }
823 \cs_set_protected_nopar:Npn \prg_set_protected_conditional:Npnn
824   { \__prg_generate_conditional_parm:nnNpnn { set } { _protected } }
825 \cs_set_protected_nopar:Npn \prg_new_protected_conditional:Npnn
826   { \__prg_generate_conditional_parm:nnNpnn { new } { _protected } }
827 \cs_set_protected:Npn \__prg_generate_conditional_parm:nnNpnn #1#2#3#4#
828   {
829     \__cs_split_function:NN #3 \__prg_generate_conditional:nnNnnnnn
830     {#1} {#2} {#4}
831   }
```

(End definition for \prg_set_conditional:Npnn and others. These functions are documented on page 34.)

`\prg_set_conditional:Nnn` The user functions for the types automatically inserting the correct parameter text based
`\prg_new_conditional:Nnn` on the signature. The various functions only differ by which function is used for the
`\prg_set_protected_conditional:Nnn` assignment. Split the base function into name and signature. The second auxiliary
`\prg_new_protected_conditional:Nnn` generates the parameter text from the number of letters in the signature. Then feed
`__prg_generate_conditional_count:nnNnn` `{⟨name⟩}` `{⟨signature⟩}` `⟨boolean⟩` `{⟨set or new⟩}` `{⟨maybe protected⟩}` `{⟨parameters⟩}`
`__prg_generate_conditional_count:nnNnnnn` `{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.

```

832 \cs_set_protected_nopar:Npn \prg_set_conditional:Nnn
833   { \prg_generate_conditional_count:nnNnn { set } { } }
834 \cs_set_protected_nopar:Npn \prg_new_conditional:Nnn
835   { \prg_generate_conditional_count:nnNnn { new } { } }
836 \cs_set_protected_nopar:Npn \prg_set_protected_conditional:Nnn
837   { \prg_generate_conditional_count:nnNnn { set } { _protected } }
838 \cs_set_protected_nopar:Npn \prg_new_protected_conditional:Nnn
839   { \prg_generate_conditional_count:nnNnn { new } { _protected } }
840 \cs_set_protected:Npn \prg_generate_conditional_count:nnNnn #1#2#3
841   {
842     \cs_split_function:NN #3 \prg_generate_conditional_count:nnNnnnn
843     {#1} {#2}
844   }
845 \cs_set_protected:Npn \prg_generate_conditional_count:nnNnnnn #1#2#3#4#5
846   {
847     \cs_parm_from_arg_count:nnF
848     { \prg_generate_conditional:nnNnnnnn {#1} {#2} #3 {#4} {#5} }
849     { \tl_count:n {#2} }
850     {
851       \msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
852       { \token_to_str:c { #1 : #2 } }
853       { \tl_count:n {#2} }
854       \use_none:nn
855     }
856   }

```

(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.

```

857 \cs_set_protected:Npn \prg_generate_conditional:nnNnnnnn #1#2#3#4#5#6#7#8
858   {
859     \if_meaning:w \c_false_bool #3
860     \msg_kernel_error:nnx { kernel } { missing-colon }
861     { \token_to_str:c {#1} }
862     \exp_after:wN \use_none:nn
863     \fi:
864     \use:x
865     {
866       \exp_not:N \prg_generate_conditional:nnnnnnw

```

```

867 \exp_not:n { {#4} {#5} {#1} {#2} {#6} {#8} }
868 \etex_detokenize:D {#7}
869 \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
870 }
871 }

```

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.

```

872 \cs_set_protected:Npn \__prg_generate_conditional:nnnnnnw #1#2#3#4#5#6#7 ,
873 {
874   \if_meaning:w \q_recursion_tail #7
875   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
876   \fi:
877   \use:c { __prg_generate_ #7 _form:wnnnnnn }
878   \tl_if_empty:nF {#7}
879   {
880     \__msg_kernel_error:nnxx
881     { kernel } { conditional-form-unknown }
882     {#7} { \token_to_str:c { #3 : #4 } }
883   }
884   \use_none:nnnnnnn
885   \q_stop
886   {#1} {#2} {#3} {#4} {#5} {#6}
887   \__prg_generate_conditional:nnnnnnw {#1} {#2} {#3} {#4} {#5} {#6}
888 }

```

(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.

```

889 \cs_set_protected:Npn \__prg_generate_p_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
890 {
891   \if_meaning:w \scan_stop: #3 \scan_stop:
892   \exp_after:wN \use_i:nn
893   \else:
894   \exp_after:wN \use_ii:nn
895   \fi:
896   {
897     \exp_args:cc { cs_ #2 #3 :Npn } { #4 _p: #5 } #6
898     { #7 \c_zero \c_true_bool \c_false_bool }
899   }
900   {

```

```

901         \_msg_kernel_error:nmx { kernel } { protected-predicate }
902         { \token_to_str:c { #4 _p: #5 } }
903     }
904 }
905 \cs_set_protected:Npn \__prg_generate_T_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
906 {
907     \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 T } #6
908     { #7 \c_zero \use:n \use_none:n }
909 }
910 \cs_set_protected:Npn \__prg_generate_F_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
911 {
912     \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 F } #6
913     { #7 \c_zero { } }
914 }
915 \cs_set_protected:Npn \__prg_generate_TF_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
916 {
917     \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 TF } #6
918     { #7 \c_zero }
919 }

```

(End definition for __prg_generate_p_form:wnnnnnn and others.)

\prg_set_eq_conditional:NNn The setting-equal functions. Split the two functions and feed a first auxiliary $\{\langle name_1 \rangle\}$
\prg_new_eq_conditional:NNn $\{\langle signature_1 \rangle\} \langle boolean_1 \rangle \{\langle name_2 \rangle\} \{\langle signature_2 \rangle\} \langle boolean_2 \rangle \langle copying\ function \rangle \langle conditions \rangle$
 $_prg_set_eq_conditional:NNn$, \q_recursion_tail , \q_recursion_stop

```

920 \cs_set_protected_nopar:Npn \prg_set_eq_conditional:NNn
921 { \_prg_set_eq_conditional:NNNn \cs_set_eq:cc }
922 \cs_set_protected_nopar:Npn \prg_new_eq_conditional:NNn
923 { \_prg_set_eq_conditional:NNNn \cs_new_eq:cc }
924 \cs_set_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
925 {
926     \use:x
927     {
928         \exp_not:N \_prg_set_eq_conditional:nnNnnNNw
929         \__cs_split_function:NN #2 \prg_do_nothing:
930         \__cs_split_function:NN #3 \prg_do_nothing:
931         \exp_not:N #1
932         \etex_detokenize:D {#4}
933         \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
934     }
935 }

```

(End definition for \prg_set_eq_conditional:NNn and \prg_new_eq_conditional:NNn. These functions are documented on page 36.)

$_prg_set_eq_conditional:nnNnnNNw$ Split the function to be defined, and setup a manual clist loop over argument #6 of the
 $_prg_set_eq_conditional_loop:nnnnNw$ first auxiliary. The second auxiliary receives twice three arguments coming from splitting
 $_prg_set_eq_conditional_p_form:nnn$ the function to be defined and the function to copy. Make sure that both functions
 $_prg_set_eq_conditional_TF_form:nnn$ contained a colon, otherwise we don't know how to build conditionals, hence abort. Call
 $_prg_set_eq_conditional_T_form:nnn$ the looping macro, with arguments $\{\langle name_1 \rangle\} \{\langle signature_1 \rangle\} \{\langle name_2 \rangle\} \{\langle signature_2 \rangle\}$
 $_prg_set_eq_conditional_F_form:nnn$

<copying function> and followed by the comma list. At each step in the loop, make sure that the conditional form we copy is defined, and copy it, otherwise abort.

```

936 \cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
937 {
938   \if_meaning:w \c_false_bool #3
939   \__msg_kernel_error:nnx { kernel } { missing-colon }
940   { \token_to_str:c {#1} }
941   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
942   \fi:
943   \if_meaning:w \c_false_bool #6
944   \__msg_kernel_error:nnx { kernel } { missing-colon }
945   { \token_to_str:c {#4} }
946   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
947   \fi:
948   \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}
949 }
950 \cs_set_protected:Npn \__prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
951 {
952   \if_meaning:w \q_recursion_tail #6
953   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
954   \fi:
955   \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
956   \tl_if_empty:nF {#6}
957   {
958     \__msg_kernel_error:nnxx
959     { kernel } { conditional-form-unknown }
960     {#6} { \token_to_str:c { #1 : #2 } }
961   }
962   \use_none:nnnnnn
963   \q_stop
964   #5 {#1} {#2} {#3} {#4}
965   \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
966 }
967 \cs_set:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \q_stop #2#3#4#5#6
968 {
969   \__chk_if_exist_cs:c { #5 _p : #6 }
970   #2 { #3 _p : #4 } { #5 _p : #6 }
971 }
972 \cs_set:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \q_stop #2#3#4#5#6
973 {
974   \__chk_if_exist_cs:c { #5 : #6 TF }
975   #2 { #3 : #4 TF } { #5 : #6 TF }
976 }
977 \cs_set:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \q_stop #2#3#4#5#6
978 {
979   \__chk_if_exist_cs:c { #5 : #6 T }
980   #2 { #3 : #4 T } { #5 : #6 T }
981 }
982 \cs_set:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \q_stop #2#3#4#5#6

```

```

983 {
984   \_chk_if_exist_cs:c { #5      : #6 F }
985   #2 { #3      : #4 F } { #5      : #6 F }
986 }

```

(End definition for `_prg_set_eq_conditional:nnNnnNNw` and `_prg_set_eq_conditional_loop:nnnnNw`.)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either TT or TF to be tested using `\if:w` but this was later changed to 00 and 01, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

```

\c_true_bool  Here are the canonical boolean values.
\c_false_bool
987 \tex_chardef:D \c_true_bool  = 1 ~
988 \tex_chardef:D \c_false_bool = 0 ~

```

(End definition for `\c_true_bool` and `\c_false_bool`. These variables are documented on page 22.)

3.7 Dissecting a control sequence

```

\cs_to_str:N This converts a control sequence into the character string of its name, removing the
\__cs_to_str:N leading escape character. This turns out to be a non-trivial matter as there are different
\__cs_to_str:w cases:

```

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of the `\escapechar` is negative;
- when the escape character is a space.

One approach to solve this is to test how many tokens result from `\token_to_str:N \a`. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, `\token_to_str:N _` yields the escape character itself and a space. The character codes are different, thus the `\if:w` test is false, and TeX reads `__cs_to_str:N` after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial `__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.

```

989 \cs_set_nopar:Npn \cs_to_str:N
990 {
991   \__int_to_roman:w
992   \if:w \token_to_str:N \__cs_to_str:w \fi:
993   \exp_after:wN \__cs_to_str:N \token_to_str:N
994 }
995 \cs_set:Npn \__cs_to_str:N #1 { \c_zero }
996 \cs_set:Npn \__cs_to_str:w #1 \__cs_to_str:N
997 { - \__int_value:w \fi: \exp_after:wN \c_zero }

```

(End definition for `\cs_to_str:N`. This function is documented on page 19.)

```

\__cs_split_function:NN
\__cs_split_function_auxi:w
\__cs_split_function_auxii:w

```

This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean `<true>` or `<false>` is returned with `<true>` for when there is a colon in the function and `<false>` if there is not. Lastly, the second argument of `__cs_split_function:NN` is supposed to be a function taking three variables, one for name, one for signature, and one for the boolean. For example, `__cs_split_function:NN \foo_bar:cnx \use_i:nnn` as input becomes `\use_i:nnn {foo_bar} {cnx} \c_true_bool`.

We 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 `<processor>`. The second auxiliary trims the trailing `\q_mark` from the function name if present (that is, if the original function had no colon).

```

998 \group_begin:
999 \tex_lccode:D '\@ = '\: \scan_stop:
1000 \tex_catcode:D '\@ = 12 ~
1001 \tex_lowercase:D
1002 {
1003   \group_end:
1004   \cs_set:Npn \__cs_split_function:NN #1
1005   {
1006     \exp_after:wN \exp_after:wN
1007     \exp_after:wN \__cs_split_function_auxi:w
1008     \cs_to_str:N #1 \q_mark \c_true_bool
1009     @ \q_mark \c_false_bool
1010     \q_stop
1011   }
1012   \cs_set:Npn \__cs_split_function_auxi:w #1 @ #2 \q_mark #3#4 \q_stop #5
1013   { \__cs_split_function_auxii:w #5 #1 \q_mark \q_stop {#2} #3 }
1014   \cs_set:Npn \__cs_split_function_auxii:w #1#2 \q_mark #3 \q_stop
1015   { #1 {#2} }
1016 }

```


(End definition for `_cs_split_function:NN`.)

`_cs_get_function_name:N`
`_cs_get_function_signature:N`

Simple wrappers.

```
1017 \cs_set:Npn \_cs_get_function_name:N #1
1018 { \_cs_split_function:NN #1 \use_i:nnn }
1019 \cs_set:Npn \_cs_get_function_signature:N #1
1020 { \_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`
`\cs_if_exist_p:c`
`\cs_if_exist:NTF`
`\cs_if_exist:cTF`

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:`.

```
1021 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1022 {
1023   \if_meaning:w #1 \scan_stop:
1024   \prg_return_false:
1025   \else:
1026     \if_cs_exist:N #1
1027     \prg_return_true:
1028   \else:
1029     \prg_return_false:
1030   \fi:
1031 \fi:
1032 }
```

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.

```
1033 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1034 {
1035   \if_cs_exist:w #1 \cs_end:
1036   \exp_after:wN \use_i:nn
1037   \else:
1038     \exp_after:wN \use_ii:nn
1039   \fi:
1040   {
1041     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1042     \prg_return_false:
1043   \else:
1044     \prg_return_true:
1045   \fi:
1046 }
```

```

1046     }
1047     \prg_return_false:
1048 }

```

(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 1049 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
\cs_if_free:NTF 1050 {
\cs_if_free:cTF 1051   \if_meaning:w #1 \scan_stop:
1052   \prg_return_true:
1053   \else:
1054     \if_cs_exist:N #1
1055     \prg_return_false:
1056     \else:
1057       \prg_return_true:
1058     \fi:
1059   \fi:
1060 }
1061 \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1062 {
1063   \if_cs_exist:w #1 \cs_end:
1064   \exp_after:wN \use_i:nn
1065   \else:
1066     \exp_after:wN \use_ii:nn
1067   \fi:
1068   {
1069     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1070     \prg_return_true:
1071     \else:
1072       \prg_return_false:
1073     \fi:
1074   }
1075   { \prg_return_true: }
1076 }

```

(End definition for \cs_if_free:N and \cs_if_free:c. These functions are documented on page ??.)

\cs_if_exist_use:NTF**** The \cs_if_exist_use:... functions cannot be implemented as conditionals because
\cs_if_exist_use:cTF**** the true branch must leave both the control sequence itself and the true code in the input
\cs_if_exist_use:N stream. For the c variants, we are careful not to put the control sequence in the hash
\cs_if_exist_use:c table if it does not exist.

```

1077 \cs_set:Npn \cs_if_exist_use:NTF #1#2
1078 { \cs_if_exist:NTF #1 { #1 #2 } }
1079 \cs_set:Npn \cs_if_exist_use:NF #1
1080 { \cs_if_exist:NTF #1 { #1 } }
1081 \cs_set:Npn \cs_if_exist_use:NT #1 #2
1082 { \cs_if_exist:NTF #1 { #1 #2 } { } }
1083 \cs_set:Npn \cs_if_exist_use:N #1
1084 { \cs_if_exist:NTF #1 { #1 } { } }
1085 \cs_set:Npn \cs_if_exist_use:cTF #1#2

```

```

1086 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
1087 \cs_set:Npn \cs_if_exist_use:cF #1
1088 { \cs_if_exist:cTF {#1} { \use:c {#1} } }
1089 \cs_set:Npn \cs_if_exist_use:cT #1#2
1090 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
1091 \cs_set:Npn \cs_if_exist_use:c #1
1092 { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }

```

(End definition for `\cs_if_exist_use:NTF` and `\cs_if_exist_use:cTF`. These functions are documented on page ??.)

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
`\iow_term:x` the log file and the terminal. These will be redefined later by `l3io`.

```

1093 \cs_set_protected_nopar:Npn \iow_log:x
1094 { \tex_immediate:D \tex_write:D \c_minus_one }
1095 \cs_set_protected_nopar:Npn \iow_term:x
1096 { \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
`__msg_kernel_error:nxx` usable if terse error message and halt. This can only happen if a coding error is made by
`__msg_kernel_error:nn` the team, so this is a reasonable response.

```

1097 \cs_set_protected:Npn __msg_kernel_error:nxxx #1#2#3#4
1098 {
1099   \tex_errmessage:D
1100   {
1101     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
1102     Argh,~internal~LaTeX3~error! ^^J ^^J
1103     Module ~ #1 , ~ message-name~"#2": ^^J
1104     Arguments~'#3'~and~'#4' ^^J ^^J
1105     This~is~one~for~The~LaTeX3~Project:~bailing~out
1106   }
1107   \tex_end:D
1108 }
1109 \cs_set_protected:Npn __msg_kernel_error:nxx #1#2#3
1110 { \__msg_kernel_error:nxxx {#1} {#2} {#3} { } }
1111 \cs_set_protected:Npn __msg_kernel_error:nn #1#2
1112 { \__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.

```
1113 \cs_set_nopar:Npn \msg_line_context:
1114 { on~line~ \tex_the:D \tex_inputlineno:D }
```

(End definition for `\msg_line_context:`. This function is documented on page 147.)

`__chk_if_free_cs:N` This command is called by `\cs_new_nopar:Npn` and `\cs_new_eq:NN` etc. to make sure that the argument sequence is not already in use. If it is, an error is signalled. It checks if `<csname>` is undefined or `\scan_stop:`. Otherwise an error message is issued. We have to make sure we don't put the argument into the conditional processing since it may be an `\if...` type function!

```
1115 \cs_set_protected:Npn \__chk_if_free_cs:N #1
1116 {
1117   \cs_if_free:NF #1
1118   {
1119     \__msg_kernel_error:nxx { kernel } { command-already-defined }
1120     { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1121   }
1122 }
1123 <*package>
1124 \tex_ifodd:D \l@expl@log@functions@bool
1125 \cs_set_protected:Npn \__chk_if_free_cs:N #1
1126 {
1127   \cs_if_free:NF #1
1128   {
1129     \__msg_kernel_error:nxx { kernel } { command-already-defined }
1130     { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1131   }
1132   \iow_log:x { Defining~\token_to_str:N #1~ \msg_line_context: }
1133 }
1134 \fi:
1135 </package>
1136 \cs_set_protected_nopar:Npn \__chk_if_free_cs:c
1137 { \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_var:N` Create the checking function for variable definitions when the option is set.

```
1138 <*package>
1139 \tex_ifodd:D \l@expl@check@declarations@bool
1140 \cs_set_protected:Npn \__chk_if_exist_var:N #1
1141 {
1142   \cs_if_exist:NF #1
1143   {
1144     \__msg_kernel_error:nxx { check } { non-declared-variable }
1145     { \token_to_str:N #1 }
1146   }
1147 }
1148 \fi:
1149 </package>
```

(End definition for `_chk_if_exist_var:N`.)

`_chk_if_exist_cs:N` This function issues an error message when the control sequence in its argument does
`_chk_if_exist_cs:c` not exist.

```

1150 \cs_set_protected:Npn \_chk_if_exist_cs:N #1
1151 {
1152   \cs_if_exist:NF #1
1153   {
1154     \_msg_kernel_error:nxx { kernel } { command-not-defined }
1155     { \token_to_str:N #1 }
1156   }
1157 }
1158 \cs_set_protected_nopar:Npn \_chk_if_exist_cs:c
1159 { \exp_args:Nc \_chk_if_exist_cs:N }

```

(End definition for `_chk_if_exist_cs:N` and `_chk_if_exist_cs:c`.)

3.10 More new definitions

Function which check that the control sequence is free before defining it.

```

\cs_new_nopar:Npn
\cs_new_nopar:Npx
\cs_new:Npn
\cs_new:Npx
\cs_new_protected_nopar:Npn
\cs_new_protected_nopar:Npx
\cs_new_protected:Npn
\cs_new_protected:Npx
\_cs_tmp:w
1160 \cs_set:Npn \_cs_tmp:w #1#2
1161 {
1162   \cs_set_protected:Npn #1 ##1
1163   {
1164     \_chk_if_free_cs:N ##1
1165     #2 ##1
1166   }
1167 }
1168 \_cs_tmp:w \cs_new_nopar:Npn \cs_gset_nopar:Npn
1169 \_cs_tmp:w \cs_new_nopar:Npx \cs_gset_nopar:Npx
1170 \_cs_tmp:w \cs_new:Npn \cs_gset:Npn
1171 \_cs_tmp:w \cs_new:Npx \cs_gset:Npx
1172 \_cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1173 \_cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1174 \_cs_tmp:w \cs_new_protected:Npn \cs_gset_protected:Npn
1175 \_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 25.)

`\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_new_nopar:cpn` `\cs_set_nopar:cpn` $\langle string \rangle \langle rep-text \rangle$ will turn $\langle string \rangle$ into a `csname` and then assign $\langle rep-text \rangle$ to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```

1176 \cs_set:Npn \_cs_tmp:w #1#2
1177 { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
1178 \_cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
1179 \_cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx

```

```

1180 \_cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1181 \_cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1182 \_cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
1183 \_cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

(End definition for \cs_set_nopar:cpn and others. These functions are documented on page ??.)

\cs_set:cpn Variants of the \cs_set:Npn versions which make a csname out of the first arguments.
 \cs_set:cpx We may also do this globally.

```

1184 \_cs_tmp:w \cs_set:cpn \cs_set:Npn
1185 \_cs_tmp:w \cs_set:cpx \cs_set:Npx
1186 \cs_new:cpn \_cs_tmp:w \cs_gset:cpn \cs_gset:Npn
1187 \cs_new:cpx \_cs_tmp:w \cs_gset:cpx \cs_gset:Npx
1188 \_cs_tmp:w \cs_new:cpn \cs_new:Npn
1189 \_cs_tmp:w \cs_new:cpx \cs_new:Npx

```

(End definition for \cs_set:cpn and others. These functions are documented on page ??.)

\cs_set_protected_nopar:cpn Variants of the \cs_set_protected_nopar:Npn versions which make a csname out of
 \cs_set_protected_nopar:cpx the first arguments. We may also do this globally.

```

1190 \_cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
1191 \_cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
1192 \_cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
1193 \_cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1194 \_cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
1195 \_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.

```

1196 \_cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
1197 \_cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
1198 \_cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
1199 \_cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
1200 \_cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
1201 \_cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx

```

(End definition for \cs_set_protected:cpn and others. These functions are documented on page ??.)

3.11 Copying definitions

\cs_set_eq:NN These macros allow us to copy the definition of a control sequence to another control
 \cs_set_eq:cN sequence.

\cs_set_eq:Nc The = sign allows us to define funny char tokens like = itself or \sqcup with this function.
 \cs_set_eq:cc For the definition of \c_space_char{~} to work we need the ~ after the =.

\cs_gset_eq:NN \cs_set_eq:NN is long to avoid problems with a literal argument of \par. While
 \cs_gset_eq:cN \cs_new_eq:NN will probably never be correct with a first argument of \par, define it
 \cs_gset_eq:Nc long in order to throw an “already defined” error rather than “runaway argument”.

```

1202 \cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
1203 \cs_new_protected_nopar:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
\cs_new_eq:cN
\cs_new_eq:Nc
\cs_new_eq:cc

```

```

1204 \cs_new_protected_nopar:Npn \cs_set_eq:Nc { \exp_args:Nnc \cs_set_eq:NN }
1205 \cs_new_protected_nopar:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
1206 \cs_new_protected_nopar:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }
1207 \cs_new_protected_nopar:Npn \cs_gset_eq:Nc { \exp_args:Nnc \cs_gset_eq:NN }
1208 \cs_new_protected_nopar:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
1209 \cs_new_protected_nopar:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }
1210 \cs_new_protected:Npn \cs_new_eq:NN #1
1211 {
1212   \__chk_if_free_cs:N #1
1213   \tex_global:D \cs_set_eq:NN #1
1214 }
1215 \cs_new_protected_nopar:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
1216 \cs_new_protected_nopar:Npn \cs_new_eq:Nc { \exp_args:Nnc \cs_new_eq:NN }
1217 \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
`\cs_undefine:c` 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.

```

1218 \cs_new_protected:Npn \cs_undefine:N #1
1219 { \cs_gset_eq:NN #1 \tex_undefined:D }
1220 \cs_new_protected:Npn \cs_undefine:c #1
1221 {
1222   \if_cs_exist:w #1 \cs_end:
1223     \exp_after:wN \use:n
1224   \else:
1225     \exp_after:wN \use_none:n
1226   \fi:
1227   { \cs_gset_eq:cN {#1} \tex_undefined:D }
1228 }

```

(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
`_cs_parm_from_arg_count_test:nnF` 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.

```

1229 \cs_set_protected:Npn \_cs_parm_from_arg_count:nnF #1#2
1230 {
1231   \exp_args:Nx \_cs_parm_from_arg_count_test:nnF

```

```

1232 {
1233   \exp_after:wN \exp_not:n
1234   \if_case:w \__int_eval:w #2 \__int_eval_end:
1235     { }
1236   \or: { ##1 }
1237   \or: { ##1##2 }
1238   \or: { ##1##2##3 }
1239   \or: { ##1##2##3##4 }
1240   \or: { ##1##2##3##4##5 }
1241   \or: { ##1##2##3##4##5##6 }
1242   \or: { ##1##2##3##4##5##6##7 }
1243   \or: { ##1##2##3##4##5##6##7##8 }
1244   \or: { ##1##2##3##4##5##6##7##8##9 }
1245   \else: { \c_false_bool }
1246   \fi:
1247 }
1248 {#1}
1249 }
1250 \cs_set_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
1251 {
1252   \if_meaning:w \c_false_bool #1
1253     \exp_after:wN \use_ii:nn
1254   \else:
1255     \exp_after:wN \use_i:nn
1256   \fi:
1257   { #2 {#1} }
1258 }
(End definition for \__cs_parm_from_arg_count:nnF.)

```

3.14 Defining functions from a given number of arguments

`__cs_count_signature:N` Counting the number of tokens in the signature, *i.e.*, the number of arguments the function should take. Since this is not used in any time-critical function, we simply use `\tl_count:n` if there is a signature, otherwise `-1` arguments to signal an error. We need a variant form right away.

```

1259 \cs_new:Npn \__cs_count_signature:N #1
1260 { \int_eval:n { \__cs_split_function:NN #1 \__cs_count_signature:nnN } }
1261 \cs_new:Npn \__cs_count_signature:nnN #1#2#3
1262 {
1263   \if_meaning:w \c_true_bool #3
1264     \tl_count:n {#2}
1265   \else:
1266     \c_minus_one
1267   \fi:
1268 }
1269 \cs_new_nopar:Npn \__cs_count_signature:c
1270 { \exp_args:Nc \__cs_count_signature:N }
(End definition for \__cs_count_signature:N and \__cs_count_signature:c.)

```


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.

```

1271 \cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
1272 {
1273   \__cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
1274   {
1275     \__msg_kernel_error:nxxx { kernel } { bad-number-of-arguments }
1276     { \token_to_str:N #1 } { \int_eval:n {#3} }
1277   }
1278   {#4}
1279 }

```

```

1280 \cs_new_protected_nopar:Npn \cs_generate_from_arg_count:cNnn
1281   { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
1282 \cs_new_protected_nopar:Npn \cs_generate_from_arg_count:Ncnn
1283   { \exp_args:NNc \cs_generate_from_arg_count:NNnn }

```

3.15 Using the signature to define functions

We want to define `\cs_set:Nn` as

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.

```

1290     }
1291   }
1292   \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
1293   {
1294     \__cs_split_function:NN #2 \__cs_generate_from_signature:nnNNNn
1295     #1 #2
1296   }
1297   \cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
1298   {
1299     \bool_if:NTF #3
1300     {
1301       \cs_generate_from_arg_count:NNnn
1302       #5 #4 { \tl_count:n {#2} } {#6}
1303     }
1304     {
1305       \__msg_kernel_error:nmx { kernel } { missing-colon }
1306       { \token_to_str:N #5 }
1307     }
1308   }

```

Then we define the 24 variants beginning with N.

```

1309 \__cs_tmp:w { set } { Nn } { Npn }
1310 \__cs_tmp:w { set } { Nx } { Npx }
1311 \__cs_tmp:w { set_nopar } { Nn } { Npn }
1312 \__cs_tmp:w { set_nopar } { Nx } { Npx }
1313 \__cs_tmp:w { set_protected } { Nn } { Npn }
1314 \__cs_tmp:w { set_protected } { Nx } { Npx }
1315 \__cs_tmp:w { set_protected_nopar } { Nn } { Npn }
1316 \__cs_tmp:w { set_protected_nopar } { Nx } { Npx }
1317 \__cs_tmp:w { gset } { Nn } { Npn }
1318 \__cs_tmp:w { gset } { Nx } { Npx }
1319 \__cs_tmp:w { gset_nopar } { Nn } { Npn }
1320 \__cs_tmp:w { gset_nopar } { Nx } { Npx }
1321 \__cs_tmp:w { gset_protected } { Nn } { Npn }
1322 \__cs_tmp:w { gset_protected } { Nx } { Npx }
1323 \__cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
1324 \__cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
1325 \__cs_tmp:w { new } { Nn } { Npn }
1326 \__cs_tmp:w { new } { Nx } { Npx }
1327 \__cs_tmp:w { new_nopar } { Nn } { Npn }
1328 \__cs_tmp:w { new_nopar } { Nx } { Npx }
1329 \__cs_tmp:w { new_protected } { Nn } { Npn }
1330 \__cs_tmp:w { new_protected } { Nx } { Npx }
1331 \__cs_tmp:w { new_protected_nopar } { Nn } { Npn }
1332 \__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

```

```

1335 \cs_new_protected_nopar:cpx { cs_ #1 : c #2 }
1336 {
1337   \exp_not:N \exp_args:Nc
1338   \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:
1339 }
1340 }
1341 \__cs_tmp:w { set } { n }
1342 \__cs_tmp:w { set } { x }
1343 \__cs_tmp:w { set_nopar } { n }
1344 \__cs_tmp:w { set_nopar } { x }
1345 \__cs_tmp:w { set_protected } { n }
1346 \__cs_tmp:w { set_protected } { x }
1347 \__cs_tmp:w { set_protected_nopar } { n }
1348 \__cs_tmp:w { set_protected_nopar } { x }
1349 \__cs_tmp:w { gset } { n }
1350 \__cs_tmp:w { gset } { x }
1351 \__cs_tmp:w { gset_nopar } { n }
1352 \__cs_tmp:w { gset_nopar } { x }
1353 \__cs_tmp:w { gset_protected } { n }
1354 \__cs_tmp:w { gset_protected } { x }
1355 \__cs_tmp:w { gset_protected_nopar } { n }
1356 \__cs_tmp:w { gset_protected_nopar } { x }
1357 \__cs_tmp:w { new } { n }
1358 \__cs_tmp:w { new } { x }
1359 \__cs_tmp:w { new_nopar } { n }
1360 \__cs_tmp:w { new_nopar } { x }
1361 \__cs_tmp:w { new_protected } { n }
1362 \__cs_tmp:w { new_protected } { x }
1363 \__cs_tmp:w { new_protected_nopar } { n }
1364 \__cs_tmp:w { new_protected_nopar } { x }

```

(End definition for \cs_set:cn and others. These functions are documented on page ??.)

3.16 Checking control sequence equality

\cs_if_eq_p:NN Check if two control sequences are identical.

```

\cs_if_eq_p:cN 1365 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
\cs_if_eq_p:Nc 1366 {
\cs_if_eq_p:cc 1367   \if_meaning:w #1#2
\cs_if_eq:NNTF 1368   \prg_return_true: \else: \prg_return_false: \fi:
\cs_if_eq:cNTF 1369 }
\cs_if_eq:NcTF 1370 \cs_new_nopar:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
\cs_if_eq:ccTF 1371 \cs_new_nopar:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
1372 \cs_new_nopar:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }
1373 \cs_new_nopar:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
1374 \cs_new_nopar:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
1375 \cs_new_nopar:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
1376 \cs_new_nopar:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNT }
1377 \cs_new_nopar:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
1378 \cs_new_nopar:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }

```

```

1379 \cs_new_nopar:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
1380 \cs_new_nopar:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
1381 \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 Check that the variable exists, then apply the `\showthe` primitive to the variable. The odd-looking `\use:n` gives a nicer output.

```

\__kernel_register_show:c
1382 \cs_new_protected:Npn \__kernel_register_show:N #1
1383 {
1384   \cs_if_exist:NTF #1
1385   { \tex_showthe:D \use:n {#1} }
1386   {
1387     \__msg_kernel_error:nnx { kernel } { variable-not-defined }
1388     { \token_to_str:N #1 }
1389   }
1390 }
1391 \cs_new_protected_nopar:Npn \__kernel_register_show:c
1392 { \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.

```

\cs_show:c
\__cs_show:www
1393 \group_begin:
1394   \tex_lccode:D '?' = ': \scan_stop:
1395   \tex_catcode:D '?' = 12 \scan_stop:
1396   \tex_lowercase:D
1397   {
1398     \group_end:
1399     \cs_new_protected:Npn \cs_show:N #1
1400     {
1401       \__msg_show_variable:n
1402       {
1403         > ~ \token_to_str:N #1 =
1404         \exp_after:wN \__cs_show:www \cs_meaning:N #1
1405         \use_none:nn ? \prg_do_nothing:
1406       }
1407     }
1408     \cs_new:Npn \__cs_show:www #1 ? { #1 ? \ }
1409   }
1410   \cs_new_protected_nopar:Npn \cs_show:c
1411   { \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
1412 \cs_new_eq:NN \luatex_if_engine:T \use_none:n
1413 \cs_new_eq:NN \luatex_if_engine:F \use:n
1414 \cs_new_eq:NN \luatex_if_engine:TF \use_ii:nn
1415 \cs_new_eq:NN \pdfTeX_if_engine:T \use:n
1416 \cs_new_eq:NN \pdfTeX_if_engine:F \use_none:n
1417 \cs_new_eq:NN \pdfTeX_if_engine:TF \use_i:nn
1418 \cs_new_eq:NN \xetex_if_engine:T \use_none:n
1419 \cs_new_eq:NN \xetex_if_engine:F \use:n
1420 \cs_new_eq:NN \xetex_if_engine:TF \use_ii:nn
1421 \cs_new_eq:NN \luatex_if_engine_p: \c_false_bool
1422 \cs_new_eq:NN \pdfTeX_if_engine_p: \c_true_bool
1423 \cs_new_eq:NN \xetex_if_engine_p: \c_false_bool
1424 \cs_if_exist:NT \xetex_XeTeXversion:D
1425 {
1426   \cs_gset_eq:NN \pdfTeX_if_engine:T \use_none:n
1427   \cs_gset_eq:NN \pdfTeX_if_engine:F \use:n
1428   \cs_gset_eq:NN \pdfTeX_if_engine:TF \use_ii:nn
1429   \cs_gset_eq:NN \xetex_if_engine:T \use:n
1430   \cs_gset_eq:NN \xetex_if_engine:F \use_none:n
1431   \cs_gset_eq:NN \xetex_if_engine:TF \use_i:nn
1432   \cs_gset_eq:NN \pdfTeX_if_engine_p: \c_false_bool
1433   \cs_gset_eq:NN \xetex_if_engine_p: \c_true_bool
1434 }
1435 \cs_if_exist:NT \luatex_directlua:D
1436 {
1437   \cs_gset_eq:NN \luatex_if_engine:T \use:n
1438   \cs_gset_eq:NN \luatex_if_engine:F \use_none:n
1439   \cs_gset_eq:NN \luatex_if_engine:TF \use_i:nn
1440   \cs_gset_eq:NN \pdfTeX_if_engine:T \use_none:n
1441   \cs_gset_eq:NN \pdfTeX_if_engine:F \use:n
1442   \cs_gset_eq:NN \pdfTeX_if_engine:TF \use_ii:nn
1443   \cs_gset_eq:NN \luatex_if_engine_p: \c_true_bool
1444   \cs_gset_eq:NN \pdfTeX_if_engine_p: \c_false_bool
1445 }
```

(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!

```
1446 \cs_new_nopar:Npn \prg_do_nothing: { }
```

(End definition for `\prg_do_nothing:`. This function is documented on page 10.)

3.20 Breaking out of mapping functions

`__prg_break_point:Nn`
`__prg_map_break:Nn`

In inline mappings, the nesting level must be reset at the end of the mapping, even when the user decides to break out. This is done by putting the code that must be performed as an argument of `__prg_break_point:Nn`. The breaking functions are then defined to jump to that point and perform the argument of `__prg_break_point:Nn`, before the user's code (if any). There is a check that we close the correct loop, otherwise we continue breaking.

```

1447 \cs_new_eq:NN \__prg_break_point:Nn \use_ii:nn
1448 \cs_new:Npn \__prg_map_break:Nn #1#2#3 \__prg_break_point:Nn #4#5
1449 {
1450     #5
1451     \if_meaning:w #1 #4
1452     \exp_after:wN \use_iii:nnn
1453     \fi:
1454     \__prg_map_break:Nn #1 {#2}
1455 }
```

(End definition for `__prg_break_point:Nn` and `__prg_map_break:Nn`. These functions are documented on page 42.)

`__prg_break_point:`
`__prg_break:`
`__prg_break:n`

Very simple analogues of `__prg_break_point:Nn` and `__prg_map_break:Nn`, for use in fast short-term recursions which are not mappings, do not need to support nesting, and in which nothing has to be done at the end of the loop.

```

1456 \cs_new_eq:NN \__prg_break_point: \prg_do_nothing:
1457 \cs_new:Npn \__prg_break: #1 \__prg_break_point: { }
1458 \cs_new:Npn \__prg_break:n #1#2 \__prg_break_point: {#1}
```

(End definition for `__prg_break_point:..` This function is documented on page ??.)

```

1459 </initex | package>
```

4 l3expan implementation

```

1460 <*initex | package>
1461 <@@=exp>
```

`\exp_after:wN`
`\exp_not:N`
`\exp_not:n`

These are defined in `l3basics`.
(End definition for `\exp_after:wN`. This function is documented on page 32.)

4.1 General expansion

In this section a general mechanism for defining functions to handle argument handling is defined. These general expansion functions are expandable unless `x` is used. (Any version of `x` is going to have to use one of the L^AT_EX3 names for `\cs_set_nopar:Npx` at some point, and so is never going to be expandable.)

The definition of expansion functions with this technique happens in section 4.3. In section 4.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

`\l__exp_internal_tl` This scratch token list variable is defined in `l3basics`, as it is needed “early”. This is just a reminder that is the case!

(End definition for \l__exp_internal_tl. This variable is documented on page 33.)

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.

```
1462 \cs_new:Npn \__exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
1463 \cs_new:Npn \__exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }
```

(End definition for __exp_arg_next:nnn.)

`\:::` The end marker is just another name for the identity function.

```
1464 \cs_new:Npn \::: #1 {#1}
```

(End definition for \:::.)

`\::n` This function is used to skip an argument that doesn’t need to be expanded.

```
1465 \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.

```
1466 \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.

```
1467 \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.

```
1468 \cs_new:Npn \::c #1 \::: #2#3
1469 { \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.

```

1470 \cs_new:Npn \::o #1 \::: #2#3
1471 { \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 TeX had fully expanded `\cs_set_eq:Nc \aaa { b \l_tmpa_tl b }` into `\cs_set_eq:NN \aaa = \blurb` which then turned out to contain the non-expandable token `\cs_set_eq:NN`. Since the expansion of `\romannumeral -'0` is $\langle null \rangle$, we wind up with a fully expanded list, only TeX has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```

1472 \cs_new:Npn \::f #1 \::: #2#3
1473 {
1474   \exp_after:wN \__exp_arg_next:nnn
1475   \exp_after:wN { \tex_romannumeral:D -'0 #3 }
1476   {#1} {#2}
1477 }
1478 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
(End definition for \::f.)

```

\::x This function is used to expand an argument fully.

```

1479 \cs_new_protected:Npn \::x #1 \::: #2#3
1480 {
1481   \cs_set_nopar:Npx \l__exp_internal_tl { {#3} }
1482   \exp_after:wN \__exp_arg_next:nnn \l__exp_internal_tl {#1} {#2}
1483 }
(End definition for \::x.)

```

\::v These functions return the value of a register, i.e., one of `tl`, `clist`, `int`, `skip`, `dim` 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.

```

1484 \cs_new:Npn \::V #1 \::: #2#3
1485 {
1486   \exp_after:wN \__exp_arg_next:nnn
1487   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #3 }
1488   {#1} {#2}
1489 }
1490 \cs_new:Npn \::v # 1\::: #2#3
1491 {

```



```

1492     \exp_after:wN \__exp_arg_next:nnn
1493     \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:c {#3} }
1494     {#1} {#2}
1495 }
(End definition for \::v.)

```

`__exp_eval_register:N` This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in TeX register such as `\count`. For the TeX registers we have to utilize a `\the` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\the` and when not to. What we do here is try to find out whether the token will expand to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the register in question when it has been prefixed with `\exp_not:N` and the register itself. If it is a macro, the prefixed `\exp_not:N` will temporarily turn it into the primitive `\scan_stop:`.

```

1496 \cs_new:Npn \__exp_eval_register:N #1
1497 {
1498     \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.

```

1499     \if_meaning:w \scan_stop: #1
1500     \__exp_eval_error_msg:w
1501     \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`.

```

1502     \else:
1503     \exp_after:wN \use_i_ii:nnn
1504     \fi:
1505     \exp_after:wN \c_zero \tex_the:D #1
1506 }
1507 \cs_new:Npn \__exp_eval_register:c #1
1508 { \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}

```

```

1509 \cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
1510 {
1511     \fi:
1512     \fi:
1513     \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
1514     \c_zero
1515 }

```

(End definition for `__exp_eval_register:N` and `__exp_eval_register:c`.)

4.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable and therefore allow to prefix them with `\tex_global:D` for example.

`\exp_args:No` Those lovely runs of expansion!

```

1516 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
1517 \cs_new:Npn \exp_args:NNo #1#2#3
1518 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
1519 \cs_new:Npn \exp_args:NNNo #1#2#3#4
1520 { \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 30.)

`\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.

`\exp_args:Ncc`

`\exp_args:Nccc`

```

1521 \cs_new:Npn \exp_args:NNc #1#2#3
1522 { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
1523 \cs_new:Npn \exp_args:Ncc #1#2#3
1524 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
1525 \cs_new:Npn \exp_args:Nccc #1#2#3#4
1526 {
1527     \exp_after:wN #1
1528     \cs:w #2 \exp_after:wN \cs_end:
1529     \cs:w #3 \exp_after:wN \cs_end:
1530     \cs:w #4 \cs_end:
1531 }

```

(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`

```

1532 \cs_new:Npn \exp_args:Nf #1#2
1533 { \exp_after:wN #1 \exp_after:wN { \tex_romannumeral:D -'0 #2 } }
1534 \cs_new:Npn \exp_args:Nv #1#2
1535 {
1536     \exp_after:wN #1 \exp_after:wN
1537     { \tex_romannumeral:D \__exp_eval_register:c {#2} }
1538 }
1539 \cs_new:Npn \exp_args:Nv #1#2

```

```

1540 {
1541   \exp_after:wN #1 \exp_after:wN
1542   { \tex_romannumeral:D \__exp_eval_register:N #2 }
1543 }

```

(End definition for `\exp_args:Nf`, `\exp_args:Nv`, and `\exp_args:Nv`. These functions are documented on page 29.)

`\exp_args:NNV` Some more hand-tuned function with three arguments. If we forced that an `o` argument always has braces, we could implement `\exp_args:Nco` with less tokens and only two arguments.

```

\exp_args:NNV 1544 \cs_new:Npn \exp_args:NNf #1#2#3
\exp_args:NNv 1545 {
\exp_args:NNf 1546   \exp_after:wN #1
\exp_args:NNv 1547   \exp_after:wN #2
\exp_args:Nco 1548   \exp_after:wN { \tex_romannumeral:D -'0 #3 }
1549 }
1550 \cs_new:Npn \exp_args:NNv #1#2#3
1551 {
1552   \exp_after:wN #1
1553   \exp_after:wN #2
1554   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:c {#3} }
1555 }
1556 \cs_new:Npn \exp_args:NNV #1#2#3
1557 {
1558   \exp_after:wN #1
1559   \exp_after:wN #2
1560   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #3 }
1561 }
1562 \cs_new:Npn \exp_args:Nco #1#2#3
1563 {
1564   \exp_after:wN #1
1565   \cs:w #2 \exp_after:wN \cs_end:
1566   \exp_after:wN {#3}
1567 }
1568 \cs_new:Npn \exp_args:Ncf #1#2#3
1569 {
1570   \exp_after:wN #1
1571   \cs:w #2 \exp_after:wN \cs_end:
1572   \exp_after:wN { \tex_romannumeral:D -'0 #3 }
1573 }
1574 \cs_new:Npn \exp_args:NVV #1#2#3
1575 {
1576   \exp_after:wN #1
1577   \exp_after:wN { \tex_romannumeral:D \exp_after:wN
1578     \__exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
1579   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #3 }
1580 }

```

(End definition for `\exp_args:NNV` and others. These functions are documented on page ??.)

A few more that we can hand-tune.

```

\exp_args:Ncco 1581 \cs_new:Npn \exp_args:NNNV #1#2#3#4
\exp_args:NcNc 1582 {
\exp_args:NcNo 1583   \exp_after:wN #1
\exp_args:NNNV 1584   \exp_after:wN #2
1585   \exp_after:wN #3
1586   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #4 }
1587 }
1588 \cs_new:Npn \exp_args:NcNc #1#2#3#4
1589 {
1590   \exp_after:wN #1
1591   \cs:w #2 \exp_after:wN \cs_end:
1592   \exp_after:wN #3
1593   \cs:w #4 \cs_end:
1594 }
1595 \cs_new:Npn \exp_args:NcNo #1#2#3#4
1596 {
1597   \exp_after:wN #1
1598   \cs:w #2 \exp_after:wN \cs_end:
1599   \exp_after:wN #3
1600   \exp_after:wN {#4}
1601 }
1602 \cs_new:Npn \exp_args:Ncco #1#2#3#4
1603 {
1604   \exp_after:wN #1
1605   \cs:w #2 \exp_after:wN \cs_end:
1606   \cs:w #3 \exp_after:wN \cs_end:
1607   \exp_after:wN {#4}
1608 }

```

(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`

```
1609 \cs_new_protected_nopar:Npn \exp_args:Nx { \::x \::: }
```

(End definition for `\exp_args:Nx`. This function is documented on page 29.)

Here are the actual function definitions, using the helper functions above.

```

\exp_args:Nnc 1610 \cs_new_nopar:Npn \exp_args:Nnc { \::n \::c \::: }
\exp_args:Nfo 1611 \cs_new_nopar:Npn \exp_args:Nfo { \::f \::o \::: }
\exp_args:Nff 1612 \cs_new_nopar:Npn \exp_args:Nff { \::f \::f \::: }
\exp_args:Nnf 1613 \cs_new_nopar:Npn \exp_args:Nnf { \::n \::f \::: }
\exp_args:Nno 1614 \cs_new_nopar:Npn \exp_args:Nno { \::n \::o \::: }
\exp_args:NnV 1615 \cs_new_nopar:Npn \exp_args:NnV { \::n \::V \::: }
\exp_args:Noo
\exp_args:Nof
\exp_args:Noc
\exp_args:NNx
\exp_args:Ncx
\exp_args:Nnx
\exp_args:Nox
\exp_args:Nxo
\exp_args:Nxx

```

```

1616 \cs_new_nopar:Npn \exp_args:Noo { \::o \::o \::: }
1617 \cs_new_nopar:Npn \exp_args:Nof { \::o \::f \::: }
1618 \cs_new_nopar:Npn \exp_args:Noc { \::o \::c \::: }
1619 \cs_new_protected_nopar:Npn \exp_args:NNx { \::N \::x \::: }
1620 \cs_new_protected_nopar:Npn \exp_args:Ncx { \::c \::x \::: }
1621 \cs_new_protected_nopar:Npn \exp_args:Nnx { \::n \::x \::: }
1622 \cs_new_protected_nopar:Npn \exp_args:Nox { \::o \::x \::: }
1623 \cs_new_protected_nopar:Npn \exp_args:Nxo { \::x \::o \::: }
1624 \cs_new_protected_nopar:Npn \exp_args:Nxx { \::x \::x \::: }

```

(End definition for `\exp_args:Nnc` and others. These functions are documented on page ??.)

```

\exp_args:NNno
\exp_args:NNoo
\exp_args:Nnnc
\exp_args:Nnno
\exp_args:Nooo
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Noox

```

```

1625 \cs_new_nopar:Npn \exp_args:NNno { \::N \::n \::o \::: }
1626 \cs_new_nopar:Npn \exp_args:NNoo { \::N \::o \::o \::: }
1627 \cs_new_nopar:Npn \exp_args:Nnnc { \::n \::n \::c \::: }
1628 \cs_new_nopar:Npn \exp_args:Nnno { \::n \::n \::o \::: }
1629 \cs_new_nopar:Npn \exp_args:Nooo { \::o \::o \::o \::: }
1630 \cs_new_protected_nopar:Npn \exp_args:NNnx { \::N \::n \::x \::: }
1631 \cs_new_protected_nopar:Npn \exp_args:NNox { \::N \::o \::x \::: }
1632 \cs_new_protected_nopar:Npn \exp_args:Nnnx { \::n \::n \::x \::: }
1633 \cs_new_protected_nopar:Npn \exp_args:Nnox { \::n \::o \::x \::: }
1634 \cs_new_protected_nopar:Npn \exp_args:Nccx { \::c \::c \::x \::: }
1635 \cs_new_protected_nopar:Npn \exp_args:Ncnx { \::c \::n \::x \::: }
1636 \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
\::f_unbraced
\::o_unbraced
\::V_unbraced
\::v_unbraced
\::x_unbraced

```

There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

1637 \cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
1638 \cs_new:Npn \::f_unbraced \::: #1#2
1639 {
1640   \exp_after:wN \__exp_arg_last_unbraced:nn
1641   \exp_after:wN { \tex_romannumeral:D -'0 #2 } {#1}
1642 }
1643 \cs_new:Npn \::o_unbraced \::: #1#2
1644 { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
1645 \cs_new:Npn \::V_unbraced \::: #1#2
1646 {
1647   \exp_after:wN \__exp_arg_last_unbraced:nn
1648   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #2 } {#1}
1649 }
1650 \cs_new:Npn \::v_unbraced \::: #1#2
1651 {
1652   \exp_after:wN \__exp_arg_last_unbraced:nn
1653   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:c {#2} } {#1}
1654 }
1655 \cs_new_protected:Npn \::x_unbraced \::: #1#2

```

```

1656 {
1657   \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
1658   \l__exp_internal_tl
1659 }

```

(End definition for __exp_arg_last_unbraced:nn.)

\exp_last_unbraced:NV \exp_last_unbraced:Nv
\exp_last_unbraced:Nf

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```

\exp_last_unbraced:Nv 1660 \cs_new:Npn \exp_last_unbraced:NV #1#2
\exp_last_unbraced:Nf 1661 { \exp_after:wN #1 \tex_romannumeral:D \__exp_eval_register:N #2 }
\exp_last_unbraced:Nc 1662 \cs_new:Npn \exp_last_unbraced:Nv #1#2
\exp_last_unbraced:NcV 1663 { \exp_after:wN #1 \tex_romannumeral:D \__exp_eval_register:c {#2} }
\exp_last_unbraced:NNV 1664 \cs_new:Npn \exp_last_unbraced:Nc #1#2 { \exp_after:wN #1 #2 }
\exp_last_unbraced:NN 1665 \cs_new:Npn \exp_last_unbraced:Nf #1#2
\exp_last_unbraced:NNN 1666 { \exp_after:wN #1 \tex_romannumeral:D -'0 #2 }
\exp_last_unbraced:NNNV 1667 \cs_new:Npn \exp_last_unbraced:Nc #1#2#3
\exp_last_unbraced:NNNo 1668 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
\exp_last_unbraced:Nn 1669 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
\exp_last_unbraced:Noo 1670 {
\exp_last_unbraced:Nfo 1671   \exp_after:wN #1
\exp_last_unbraced:NnNo 1672   \cs:w #2 \exp_after:wN \cs_end:
\exp_last_unbraced:Nx 1673   \tex_romannumeral:D \__exp_eval_register:N #3
1674 }
1675 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3
1676 {
1677   \exp_after:wN #1
1678   \exp_after:wN #2
1679   \tex_romannumeral:D \__exp_eval_register:N #3
1680 }
1681 \cs_new:Npn \exp_last_unbraced:NN 1682 \cs_new:Npn \exp_last_unbraced:Nc #1#2#3
1683 { \exp_after:wN #1 \exp_after:wN #2 #3 }
1684 \cs_new:Npn \exp_last_unbraced:NNNV 1685 \cs_new:Npn \exp_last_unbraced:NN #1#2#3#4
1686 {
1687   \exp_after:wN #1
1688   \exp_after:wN #2
1689   \exp_after:wN #3
1690   \tex_romannumeral:D \__exp_eval_register:N #4
1691 }
1692 \cs_new:Npn \exp_last_unbraced:NNNo 1693 \cs_new:Npn \exp_last_unbraced:NN #1#2#3#4
1694 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
1695 \cs_new_nopar:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \:: }
1696 \cs_new_nopar:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \:: }
1697 \cs_new_nopar:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \:: }
1698 \cs_new_nopar:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \:: }
1699 \cs_new_protected_nopar:Npn \exp_last_unbraced:Nx { \::x_unbraced \:: }

```

(End definition for \exp_last_unbraced:NV. This function is documented on page 31.)

\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.

```

1697 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
1698 { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
1699 \cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
1700 { \exp_after:wN #3 #2 #1 }

```

(End definition for `\exp_last_two_unbraced:Noo`. This function is documented on page 31.)

4.5 Preventing expansion

```

\exp_not:o
\exp_not:c
\exp_not:f
\exp_not:V
\exp_not:v
1701 \cs_new:Npn \exp_not:o #1 { \etex_unexpanded:D \exp_after:wN {#1} }
1702 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
1703 \cs_new:Npn \exp_not:f #1
1704 { \etex_unexpanded:D \exp_after:wN { \tex_romannumeral:D -'0 #1 } }
1705 \cs_new:Npn \exp_not:V #1
1706 {
1707   \etex_unexpanded:D \exp_after:wN
1708   { \tex_romannumeral:D \__exp_eval_register:N #1 }
1709 }
1710 \cs_new:Npn \exp_not:v #1
1711 {
1712   \etex_unexpanded:D \exp_after:wN
1713   { \tex_romannumeral:D \__exp_eval_register:c {#1} }
1714 }

```

(End definition for `\exp_not:o`. This function is documented on page 32.)

4.6 Defining function variants

```

1715 <@@=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.

```

1716 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
1717 {
1718   \__chk_if_exist_cs:N #1
1719   \__cs_generate_variant:N #1
1720   \exp_after:wN \__cs_split_function:NN
1721   \exp_after:wN #1

```

```

1722 \exp_after:wN \__cs_generate_variant:nnNN
1723 \exp_after:wN #1
1724 \etex_detokenize:D {#2} , \scan_stop: , \q_recursion_stop
1725 }

```

(End definition for `\cs_generate_variant:Nn`. This function is documented on page 27.)

```

\__cs_generate_variant:N
\__cs_generate_variant:ww
\__cs_generate_variant:wwNw

```

The goal here is to pick up protected parent functions. There are four cases: the parent function can be a primitive or a macro, and can be expandable or not. For non-expandable primitives, all variants should be protected; skipping the `\else:` branch is safe because all primitive TeX conditionals are expandable.

The other case where variants should be protected is when the parent function is a protected macro: then `protected` appears in the meaning before the 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`.

```

1726 \group_begin:
1727 \tex_catcode:D '\M = 12 \scan_stop:
1728 \tex_catcode:D '\A = 12 \scan_stop:
1729 \tex_catcode:D '\P = 12 \scan_stop:
1730 \tex_catcode:D '\R = 12 \scan_stop:
1731 \tex_lowercase:D
1732 {
1733   \group_end:
1734   \cs_new_protected:Npn \__cs_generate_variant:N #1
1735   {
1736     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
1737     \cs_set_eq:NN \__cs_tmp:w \cs_new_protected_nopar:Npx
1738     \else:
1739       \exp_after:wN \__cs_generate_variant:ww
1740       \token_to_meaning:N #1 MA \q_mark
1741       \q_mark \cs_new_protected_nopar:Npx
1742       PR
1743       \q_mark \cs_new_nopar:Npx
1744       \q_stop
1745     \fi:
1746   }
1747   \cs_new_protected:Npn \__cs_generate_variant:ww #1 MA #2 \q_mark
1748   { \__cs_generate_variant:wwNw #1 }
1749   \cs_new_protected:Npn \__cs_generate_variant:wwNw
1750   #1 PR #2 \q_mark #3 #4 \q_stop
1751   {
1752     \cs_set_eq:NN \__cs_tmp:w #3
1753   }

```



```

1754 }
(End definition for \_cs_generate_variant:N.)

```

```

\_cs_generate_variant:nnNN #1 : Base name.
#2 : Base signature.
#3 : Boolean.
#4 : Base function.

```

If the boolean is `\c_false_bool`, the base function has no colon and we abort with an error; otherwise, set off a loop through the desired variant forms. The original function is retained as `#4` for efficiency.

```

1755 \cs_new_protected:Npn \_cs_generate_variant:nnNN #1#2#3#4
1756 {
1757   \if_meaning:w \c_false_bool #3
1758     \_msg_kernel_error:nnx { kernel } { missing-colon }
1759     { \token_to_str:c {#1} }
1760     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1761   \fi:
1762   \_cs_generate_variant:Nnnw #4 {#1}{#2}
1763 }
(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`.

```

1764 \cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
1765 {
1766   \if_meaning:w \scan_stop: #4
1767   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1768   \fi:
1769   \use:x
1770   {
1771     \exp_not:N \__cs_generate_variant:wwNN
1772     \__cs_generate_variant_loop:nNwN { }
1773     #4
1774     \__cs_generate_variant_loop_end:nwwwNNnn
1775     \q_mark
1776     #3 ~
1777     { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
1778     { }
1779     \q_stop
1780     \exp_not:N #1 {#2} {#4}
1781   }
1782   \__cs_generate_variant:Nnnw #1 {#2} {#3}
1783 }
(End definition for \__cs_generate_variant:Nnnw.)

```

<code>__cs_generate_variant_loop:nNwN</code> <code>__cs_generate_variant_loop_same:w</code> <code>__cs_generate_variant_loop_end:nwwwNNnn</code> <code>__cs_generate_variant_loop_long:wNNnn</code> <code>__cs_generate_variant_loop_invalid:NNwNNnn</code>	<p>#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).</p> <p>#2 : Next variant letter.</p> <p>#3 : Remainder of variant form.</p> <p>#4 : Next base letter.</p>
--	---

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:wvNN`.
- 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:wvNN`.

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.

```

1784 \cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \q_mark #4
1785 {
1786   \if:w #2 #4
1787     \exp_after:wN \__cs_generate_variant_loop_same:w
1788   \else:
1789     \if:w N #4 \else:
1790       \if:w n #4 \else:
1791         \__cs_generate_variant_loop_invalid:NNwNNnn #4#2
1792       \fi:
1793     \fi:
1794   \fi:
1795   #1
1796   \prg_do_nothing:
1797   #2
1798   \__cs_generate_variant_loop:nNwN { } #3 \q_mark
1799 }
1800 \cs_new:Npn \__cs_generate_variant_loop_same:w
1801   #1 \prg_do_nothing: #2#3#4
1802 {
1803   #3 { #1 \__cs_generate_variant_same:N #2 }
1804 }
1805 \cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
1806   #1#2 \q_mark #3 ~ #4 \q_stop #5#6#7#8
1807 {
1808   \scan_stop: \scan_stop: \fi:
1809   \exp_not:N \q_mark
1810   \exp_not:N \q_stop
1811   \exp_not:N #6
1812   \exp_not:c { #7 : #8 #1 #3 }

```

```

1813 }
1814 \cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \q_stop #2#3#4#5
1815 {
1816   \exp_not:n
1817   {
1818     \q_mark
1819     \__msg_kernel_error:nxxx { kernel } { variant-too-long }
1820     {#5} { \token_to_str:N #3 }
1821     \use_none:nnnn
1822     \q_stop
1823     #3
1824     #3
1825   }
1826 }
1827 \cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
1828   #1#2 \fi: \fi: \fi: #3 \q_stop #4#5#6#7
1829 {
1830   \fi: \fi: \fi:
1831   \exp_not:n
1832   {
1833     \q_mark
1834     \__msg_kernel_error:nxxxx { kernel } { invalid-variant }
1835     {#7} { \token_to_str:N #5 } {#1} {#2}
1836     \use_none:nnnn
1837     \q_stop
1838     #5
1839     #5
1840   }
1841 }

```

(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.

```

1842 \cs_new:Npn \__cs_generate_variant_same:N #1
1843 {
1844   \if:w N #1
1845     N
1846   \else:
1847     \if:w p #1
1848       p
1849     \else:
1850       n
1851     \fi:
1852   \fi:
1853 }

```

(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.

```

1854 \cs_new_protected:Npn \__cs_generate_variant:wwNN
1855   #1 \q_mark #2 \q_stop #3#4
1856 {
1857   #2
1858   \cs_if_free:NTF #4
1859   {
1860     \group_begin:
1861       \__cs_generate_internal_variant:n {#1}
1862       \__cs_tmp:w #4 { \exp_not:c { exp_args:N #1 } \exp_not:N #3 }
1863     \group_end:
1864   }
1865   {
1866     \iow_log:x
1867     {
1868       Variant~\token_to_str:N #4~%
1869       already~defined;~ not~ changing~ it~on~line~%
1870       \tex_the:D \tex_inputlineno:D
1871     }
1872   }
1873 }

```

(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.

```

1874 \group_begin:
1875   \tex_catcode:D '\X = 12 \scan_stop:
1876   \tex_lccode:D '\N = '\N \scan_stop:
1877   \tex_lowercase:D
1878   {
1879     \group_end:
1880     \cs_new_protected:Npn \__cs_generate_internal_variant:n #1
1881     {
1882       \__cs_generate_internal_variant:wwnNwnn
1883       #1 \q_mark
1884         { \cs_set_eq:NN \__cs_tmp:w \cs_new_protected_nopar:Npx }
1885       \cs_new_protected_nopar:cpx
1886       X \q_mark
1887       { }
1888       \cs_new_nopar:cpx
1889     \q_stop
1890     { exp_args:N #1 }
1891     { \__cs_generate_internal_variant_loop:n #1 { : \use_i:nn } }
1892   }
1893   \cs_new_protected:Npn \__cs_generate_internal_variant:wwnNwnn
1894     #1 X #2 \q_mark #3 #4 #5 \q_stop #6 #7

```

```

1895     {
1896       #3
1897       \cs_if_free:cT {#6} { #4 {#6} {#7} }
1898     }
1899   }

```

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.

```

1900 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
1901 {
1902   \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
1903   \__cs_generate_internal_variant_loop:n
1904 }
(End definition for \__cs_generate_internal_variant:n.)
1905 </initex | package>

```

5 l3prg implementation

The following test files are used for this code: *m3prg001.lvt,m3prg002.lvt,m3prg003.lvt*.

```

1906 <*initex | package>

```

5.1 Primitive conditionals

`\if_bool:N` Those two primitive TeX conditionals are synonyms. They should not be used outside the kernel code.

```

1907 \tex_let:D \if_bool:N          \tex_ifodd:D
1908 \tex_let:D \if_predicate:w     \tex_ifodd:D

```

(End definition for `\if_bool:N`. This function is documented on page 41.)

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 36.)

5.3 The boolean data type

```

\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_new_protected_conditional:Npnn
\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
\prg_set_eq_conditional:Npnn
\prg_new_eq_conditional:Npnn
\prg_return_true:
\prg_return_false:

```

```

1909 <@@=bool>

```

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```

1910 \cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
1911 \cs_generate_variant:Nn \bool_new:N { c }

```

(End definition for `\bool_new:N` and `\bool_new:c`. These functions are documented on page ??.)

```

\bool_set_true:N Setting is already pretty easy.
\bool_set_true:c 1912 \cs_new_protected:Npn \bool_set_true:N #1
\bool_gset_true:N 1913 { \cs_set_eq:NN #1 \c_true_bool }
\bool_gset_true:c 1914 \cs_new_protected:Npn \bool_set_false:N #1
\bool_set_false:N 1915 { \cs_set_eq:NN #1 \c_false_bool }
\bool_set_false:c 1916 \cs_new_protected:Npn \bool_gset_true:N #1
\bool_gset_false:N 1917 { \cs_gset_eq:NN #1 \c_true_bool }
\bool_gset_false:c 1918 \cs_new_protected:Npn \bool_gset_false:N #1
1919 { \cs_gset_eq:NN #1 \c_false_bool }
1920 \cs_generate_variant:Nn \bool_set_true:N { c }
1921 \cs_generate_variant:Nn \bool_set_false:N { c }
1922 \cs_generate_variant:Nn \bool_gset_true:N { c }
1923 \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 1924 \cs_new_eq:NN \bool_set_eq:NN \cs_set_eq:NN
\bool_set_eq:Nc 1925 \cs_new_eq:NN \bool_set_eq:Nc \cs_set_eq:Nc
\bool_set_eq:cc 1926 \cs_new_eq:NN \bool_set_eq:cN \cs_set_eq:cN
\bool_gset_eq:NN 1927 \cs_new_eq:NN \bool_set_eq:cc \cs_set_eq:cc
\bool_gset_eq:cN 1928 \cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
\bool_gset_eq:Nc 1929 \cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
\bool_gset_eq:cN 1930 \cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
\bool_gset_eq:cc 1931 \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 1932 \cs_new_protected:Npn \bool_set:Nn #1#2
\bool_gset:cn 1933 { \tex_chardef:D #1 = \bool_if_p:n {#2} }
1934 \cs_new_protected:Npn \bool_gset:Nn #1#2
1935 { \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2} }
1936 \cs_generate_variant:Nn \bool_set:Nn { c }
1937 \cs_generate_variant:Nn \bool_gset:Nn { c }

```

(End definition for \bool_set:Nn and \bool_set:cn. These functions are documented on page ??.)

Booleans are not based on token lists but do need checking: this code complements similar material in l3tl.

```

1938 <*package>
1939 \tex_ifodd:D \l@expl@check@declarations@bool
1940 \cs_set_protected:Npn \bool_set_true:N #1
1941 {
1942   \__chk_if_exist_var:N #1
1943   \cs_set_eq:NN #1 \c_true_bool
1944 }
1945 \cs_set_protected:Npn \bool_set_false:N #1
1946 {
1947   \__chk_if_exist_var:N #1
1948   \cs_set_eq:NN #1 \c_false_bool

```

```

1949     }
1950 \cs_set_protected:Npn \bool_gset_true:N #1
1951 {
1952     \__chk_if_exist_var:N #1
1953     \cs_gset_eq:NN #1 \c_true_bool
1954 }
1955 \cs_set_protected:Npn \bool_gset_false:N #1
1956 {
1957     \__chk_if_exist_var:N #1
1958     \cs_gset_eq:NN #1 \c_false_bool
1959 }
1960 \cs_set_protected:Npn \bool_set_eq:NN #1
1961 {
1962     \__chk_if_exist_var:N #1
1963     \cs_set_eq:NN #1
1964 }
1965 \cs_set_protected:Npn \bool_gset_eq:NN #1
1966 {
1967     \__chk_if_exist_var:N #1
1968     \cs_gset_eq:NN #1
1969 }
1970 \cs_set_protected:Npn \bool_set:Nn #1#2
1971 {
1972     \__chk_if_exist_var:N #1
1973     \tex_chardef:D #1 = \bool_if_p:n {#2}
1974 }
1975 \cs_set_protected:Npn \bool_gset:Nn #1#2
1976 {
1977     \__chk_if_exist_var:N #1
1978     \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
1979 }
1980 \tex_fi:D
1981 \</package>

```

\bool_if_p:N Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

\bool_if_p:c

\bool_if:N~~TF~~

\bool_if:c~~TF~~

```

1982 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
1983 {
1984     \if_meaning:w \c_true_bool #1
1985     \prg_return_true:
1986     \else:
1987     \prg_return_false:
1988     \fi:
1989 }
1990 \cs_generate_variant:Nn \bool_if_p:N { c }
1991 \cs_generate_variant:Nn \bool_if:NT { c }
1992 \cs_generate_variant:Nn \bool_if:NF { c }
1993 \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` to get a better output; this function requires its argument to start with `>~`.

\bool_show:c

\bool_show:n

```

1994 \cs_new_protected:Npn \bool_show:N #1
1995 {
1996   \bool_if_exist:NTF #1
1997   { \bool_show:n {#1} }
1998   {
1999     \_msg_kernel_error:nmx { kernel } { variable-not-defined }
2000     { \token_to_str:N #1 }
2001   }
2002 }
2003 \cs_new_protected:Npn \bool_show:n #1
2004 {
2005   \bool_if:nTF {#1}
2006   { \_msg_show_variable:n { > ~ true } }
2007   { \_msg_show_variable:n { > ~ false } }
2008 }
2009 \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 37.)

\l_tmpa_bool A few booleans just if you need them.

\l_tmpb_bool

\g_tmpa_bool

\g_tmpb_bool

```

2010 \bool_new:N \l_tmpa_bool
2011 \bool_new:N \l_tmpb_bool
2012 \bool_new:N \g_tmpa_bool
2013 \bool_new:N \g_tmpb_bool

```

(End definition for `\l_tmpa_bool` and others. These variables are documented on page 37.)

\bool_if_exist_p:N Copies of the cs functions defined in l3basics.

\bool_if_exist_p:c

\bool_if_exist:N

\bool_if_exist:N

\bool_if_exist:c

```

2014 \prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N { TF , T , F , p }
2015 \prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c { TF , T , F , p }

```

(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 $\langle true \rangle$ or $\langle false \rangle$.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

$\langle true \rangle$ **And** Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

$\langle false \rangle$ **And** Current truth value is false, logical And seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return $\langle false \rangle$.

$\langle true \rangle$ **Or** Current truth value is true, logical Or seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return $\langle true \rangle$.

$\langle false \rangle$ **Or** Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

$\langle true \rangle$ **Close** Current truth value is true, Close seen, return $\langle true \rangle$.

$\langle false \rangle$ **Close** Current truth value is false, Close seen, return $\langle false \rangle$.

We introduce an additional Stop operation with the same semantics as the Close operation.

$\langle true \rangle$ **Stop** Current truth value is true, return $\langle true \rangle$.

$\langle false \rangle$ **Stop** Current truth value is false, return $\langle false \rangle$.

The reasons for this follow below.

```

2016 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
2017 {
2018   \if_predicate:w \bool_if_p:n {#1}
2019   \prg_return_true:
2020   \else:
2021   \prg_return_false:
2022   \fi:
2023 }
```

(End definition for `\bool_if:n`. These functions are documented on page 38.)

```

\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.

```

2024 \cs_new:Npn \bool_if_p:n #1
2025 {
2026   \group_align_safe_begin:
2027   \__bool_if_left_parentheses:wwwn \q_nil
2028   #1 \q_mark { }
2029   ( \q_mark { \__bool_if_right_parentheses:wwwn \q_nil }
2030   ) \q_mark { \__bool_if_or:wwwn \q_nil }
2031   || \q_mark \__bool_if_parse:NNNww
2032   \q_stop
2033 }
2034 \cs_new:Npn \__bool_if_left_parentheses:wwwn #1 \q_nil #2 ( #3 \q_mark #4
2035 { #4 \__bool_if_left_parentheses:wwwn #1 #2 (( \q_nil #3 \q_mark {#4} }
2036 \cs_new:Npn \__bool_if_right_parentheses:wwwn #1 \q_nil #2 ) #3 \q_mark #4
2037 { #4 \__bool_if_right_parentheses:wwwn #1 #2 )) \q_nil #3 \q_mark {#4} }
2038 \cs_new:Npn \__bool_if_or:wwwn #1 \q_nil #2 || #3 \q_mark #4
2039 { #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 38.)

`__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.

```

2040 \cs_new:Npn \__bool_if_parse:NNNww #1#2#3#4 \q_mark #5 \q_stop
2041 {
2042   \__bool_get_next:NN \use_i:nn (( #4 )) S
2043 }

```

(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.

```

2044 \cs_new:Npn \__bool_get_next:NN #1#2
2045 {
2046   \use:c
2047   {

```

```

2048     __bool_
2049     \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
2050     :Nw
2051   }
2052   #1 #2
2053 }

```

(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.

```

2054 \cs_new:cpn { __bool_!:Nw } #1#2
2055 { \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.

```

2056 \cs_new:cpn { __bool_(:Nw } #1#2
2057 {
2058   \exp_after:wN \__bool_choose:NNN \exp_after:wN #1
2059   \__int_value:w \__bool_get_next:NN \use_i:nn
2060 }

```

(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.

```

2061 \cs_new:cpn { __bool_p:Nw } #1
2062 { \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.

```

2063 \cs_new:Npn \__bool_choose:NNN #1#2#3
2064 {
2065   \use:c
2066   {
2067     __bool_ #3 _
2068     #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: }
2069     :w
2070   }
2071 }

```

(End definition for __bool_choose:NNN.)

`__bool_)_0:w` Closing a group is just about returning the result. The Stop operation is similar except
`__bool_)_1:w` it closes the special alignment group before returning the boolean.

`__bool_S_0:w` 2072 `\cs_new_nopar:cpn { __bool_)_0:w } { \c_false_bool }`
`__bool_S_1:w` 2073 `\cs_new_nopar:cpn { __bool_)_1:w } { \c_true_bool }`
2074 `\cs_new_nopar:cpn { __bool_S_0:w } { \group_align_safe_end: \c_false_bool }`
2075 `\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` 2076 `\cs_new_nopar:cpn { __bool_&_1:w } & { __bool_get_next:NN \use_i:nn }`
2077 `\cs_new_nopar:cpn { __bool_|_0:w } | { __bool_get_next:NN \use_i:nn }`
(End definition for __bool_&_1:w.)

`__bool_&_0:w` When the truth value has already been decided, we have to throw away the remainder
`__bool_|_1:w` of the current group as we are doing minimal evaluation. This is slightly tricky as there
`__bool_eval_skip_to_end_auxi:Nw` are no braces so we have to play match the () manually.
`__bool_eval_skip_to_end_auxii:Nw` 2078 `\cs_new_nopar:cpn { __bool_&_0:w } & { __bool_eval_skip_to_end_auxi:Nw \c_false_bool }`
`__bool_eval_skip_to_end_auxiii:Nw` 2079 `\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.

```
2080 %% (
2081 \cs_new:Npn \__bool_eval_skip_to_end_auxi:Nw #1#2 )
2082 {
2083   \__bool_eval_skip_to_end_auxii:Nw #1#2 ( % )
2084   \q_no_value \q_stop
2085   {#2}
2086 }
```

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

```
2087 \cs_new:Npn \__bool_eval_skip_to_end_auxii:Nw #1#2 ( #3#4 \q_stop #5 % )
2088 {
2089   \quark_if_no_value:NTF #3
2090   {#1}
2091   { \__bool_eval_skip_to_end_auxiii:Nw #1 #5 }
2092 }
```

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!

```
2093 \cs_new:Npn \__bool_eval_skip_to_end_auxiii:Nw #1#2 ( #3 )
2094 { % (
2095   \__bool_eval_skip_to_end_auxi:Nw #1#3 )
2096 }
```

(End definition for __bool_&_0:w.)

\bool_not_p:n The Not variant just reverses the outcome of \bool_if_p:n. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```
2097 \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 39.)

\bool_xor_p:nn Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```
2098 \cs_new:Npn \bool_xor_p:nn #1#2
2099 {
2100   \int_compare:nNnTF { \bool_if_p:n {#1} } = { \bool_if_p:n {#2} }
2101   \c_false_bool
2102   \c_true_bool
2103 }
```

(End definition for `\bool_xor_p:nn`. This function is documented on page 39.)

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`

```

2104 \cs_new:Npn \bool_while_do:Nn #1#2
2105   { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
2106 \cs_new:Npn \bool_until_do:Nn #1#2
2107   { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
2108 \cs_generate_variant:Nn \bool_while_do:Nn { c }
2109 \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`

```

2110 \cs_new:Npn \bool_do_while:Nn #1#2
2111   { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
2112 \cs_new:Npn \bool_do_until:Nn #1#2
2113   { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
2114 \cs_generate_variant:Nn \bool_do_while:Nn { c }
2115 \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`

```

2116 \cs_new:Npn \bool_while_do:nn #1#2
2117   {
2118     \bool_if:nT {#1}
2119     {
2120       #2
2121       \bool_while_do:nn {#1} {#2}
2122     }
2123   }
2124 \cs_new:Npn \bool_do_while:nn #1#2
2125   {
2126     #2
2127     \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
2128   }
2129 \cs_new:Npn \bool_until_do:nn #1#2
2130   {
2131     \bool_if:nF {#1}
2132     {
2133       #2
2134       \bool_until_do:nn {#1} {#2}
2135     }
2136   }

```

(End definition for `\bool while do:nn` and others. These functions are documented on page 39.)

5.6 Producing n copies

2142 $\langle @@=\text{prg} \rangle$

\prg_replicate:nn

```
\__prg_replicate:N
```

```
\__prg_replicate_first:N
```

```
\ prg replicate
```

```
\_ prg replicate 0:n
```

```
\ prg replicate 1:n
```

```
\ prg replicate 2:n
```

```
\ prg replicate 3:n
```

```
\ prg replicate 4:n
```

```
\ prg replicate 5:n
```

```
\ prg replicate 6:n
```

```
\ prg replicate 7:n
```

```
\ prg replicate 8:n
```

```
\ prg replicate 9:n
```

```
\ prg replicate first -:n
```

```
\ prg replicate first 0:n
```

```
\ prg replicate first 1:n
```

```
\ prg replicate first 2:n
```

```
\ prg replicate first 3:n
```

```
\ prg replicate first 4:n
```

```
\ prg replicate first 5:n
```

```
\ prg_replicate first 6:n
```

```
\ prg replicate first 7:n
```

```
\ prg replicate first 8:n
```

```
\ prg replicate first 9:n
```

— — — — —

This function uses a cascading 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.

```

2143 \cs_new:Npn \prg_replicate:nn #1
2144 {
2145   \__int_to_roman:w
2146   \exp_after:wN \__prg_replicate_first:N
2147   \__int_value:w \__int_eval:w #1 \__int_eval_end:
2148   \cs_end:
2149 }
2150 \cs_new:Npn \__prg_replicate:N #1
2151 { \cs:w \__prg_replicate_#1 :n \__prg_replicate:N }
2152 \cs_new:Npn \__prg_replicate_first:N #1
2153 { \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.

```
2154 \cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
2155 \cs_new:cpn { __prg_replicate_0:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
2156 \cs_new:cpn { __prg_replicate_1:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1 }
```



```

2157 \cs_new:cpn { __prg_replicate_2:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1#1 }
2158 \cs_new:cpn { __prg_replicate_3:n } #1
2159 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1#1#1 }
2160 \cs_new:cpn { __prg_replicate_4:n } #1
2161 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1#1#1#1 }
2162 \cs_new:cpn { __prg_replicate_5:n } #1
2163 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1#1#1#1#1 }
2164 \cs_new:cpn { __prg_replicate_6:n } #1
2165 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1#1#1#1#1#1 }
2166 \cs_new:cpn { __prg_replicate_7:n } #1
2167 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1#1#1#1#1#1#1 }
2168 \cs_new:cpn { __prg_replicate_8:n } #1
2169 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1#1#1#1#1#1#1#1 }
2170 \cs_new:cpn { __prg_replicate_9:n } #1
2171 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1#1#1#1#1#1#1#1#1 }

```

Users shouldn't ask for something to be replicated once or even not at all but...

```

2172 \cs_new:cpn { __prg_replicate_first_-:n } #1
2173 {
2174   \c_zero
2175   \_msg_kernel_expandable_error:nn { kernel } { negative-replication }
2176 }
2177 \cs_new:cpn { __prg_replicate_first_0:n } #1 { \c_zero }
2178 \cs_new:cpn { __prg_replicate_first_1:n } #1 { \c_zero #1 }
2179 \cs_new:cpn { __prg_replicate_first_2:n } #1 { \c_zero #1#1 }
2180 \cs_new:cpn { __prg_replicate_first_3:n } #1 { \c_zero #1#1#1 }
2181 \cs_new:cpn { __prg_replicate_first_4:n } #1 { \c_zero #1#1#1#1 }
2182 \cs_new:cpn { __prg_replicate_first_5:n } #1 { \c_zero #1#1#1#1#1 }
2183 \cs_new:cpn { __prg_replicate_first_6:n } #1 { \c_zero #1#1#1#1#1#1 }
2184 \cs_new:cpn { __prg_replicate_first_7:n } #1 { \c_zero #1#1#1#1#1#1#1 }
2185 \cs_new:cpn { __prg_replicate_first_8:n } #1 { \c_zero #1#1#1#1#1#1#1#1 }
2186 \cs_new:cpn { __prg_replicate_first_9:n } #1 { \c_zero #1#1#1#1#1#1#1#1#1 }

```

(End definition for `\prg_replicate:nn`. This function is documented on page 40.)

5.7 Detecting TeX's mode

`\mode_if_vertical_p:` For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the `\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.

```

2187 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
2188 { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_vertical:..` These functions are documented on page 40.)

`\mode_if_horizontal_p:` For testing horizontal mode.

```

\mode_if_horizontal:TF
2189 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
2190 { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_horizontal:..` These functions are documented on page 40.)

`\mode_if_inner_p:` For testing inner mode.
`\mode_if_inner:TF`

```

2191 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
2192 { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_inner:`. These functions are documented on page 40.)

`\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.

```

2193 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
2194 { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_math:`. These functions are documented on page 40.)

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.

```

2195 \cs_new_nopar:Npn \group_align_safe_begin:
2196 { \if_int_compare:w \if_false: { \fi: ' } = \c_zero \fi: }
2197 \cs_new_nopar:Npn \group_align_safe_end:
2198 { \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.

```

2199 \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:.`)

```
2200 <@@=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
2201 \group_begin:
2202 \tex_lccode:D '*' = 'g \scan_stop:
2203 \tex_catcode:D '*' = \c_twelve
2204 \tl_to_lowercase:n
2205 {
2206   \group_end:
2207   \cs_new:Npn __prg_variable_get_scope:N #1
2208   {
2209     \exp_after:wN \exp_after:wN
2210     \exp_after:wN __prg_variable_get_scope:w
2211     \cs_to_str:N #1 \exp_stop_f: \q_stop
2212   }
2213   \cs_new:Npn __prg_variable_get_scope:w #1#2 \q_stop
2214   { \token_if_eq_meaning:NNT * #1 { g } }
2215 }
2216 \group_begin:
2217 \tex_lccode:D '*' = ' _ \scan_stop:
2218 \tex_catcode:D '*' = \c_twelve
2219 \tl_to_lowercase:n
2220 {
2221   \group_end:
2222   \cs_new:Npn __prg_variable_get_type:N #1
```

```

2223     {
2224         \exp_after:wN \__prg_variable_get_type:w
2225         \token_to_str:N #1 * a \q_stop
2226     }
2227 \cs_new:Npn \__prg_variable_get_type:w #1 * #2#3 \q_stop
2228 {
2229     \token_if_eq_meaning:NNTF a #2
2230     {#1}
2231     { \__prg_variable_get_type:w #2#3 \q_stop }
2232 }
2233 }

```

(End definition for __prg_variable_get_scope:N.)

\g__prg_map_int A nesting counter for mapping.

```

2234 \int_new:N \g__prg_map_int

```

(End definition for \g__prg_map_int. This variable is documented on page 42.)

__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 42.)

__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 2235 </initex | package)

6 l3quark implementation

The following test files are used for this code: m3quark001.lvt.

```

2236 <*initex | package>

```

6.1 Quarks

\quark_new:N Allocate a new quark.

```

2237 \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 44.)

\q_nil Some “public” quarks. \q_stop is an “end of argument” marker, \q_nil is a empty value
\q_mark and \q_no_value marks an empty argument.
\q_no_value 2238 \quark_new:N \q_nil
\q_stop 2239 \quark_new:N \q_mark
2240 \quark_new:N \q_no_value
2241 \quark_new:N \q_stop

(End definition for \q_nil and others. These variables are documented on page 44.)

`\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.

```
2242 \quark_new:N \q_recursion_tail
2243 \quark_new:N \q_recursion_stop
```

(End definition for `\q_recursion_tail` and `\q_recursion_stop`. These variables are documented on page 45.)

`\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. `\quark_if_recursion_tail_stop_do:Nn` 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.

```
2244 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
2245 {
2246   \if_meaning:w \q_recursion_tail #1
2247   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2248   \fi:
2249 }
2250 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
2251 {
2252   \if_meaning:w \q_recursion_tail #1
2253   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2254   \else:
2255     \exp_after:wN \use_none:n
2256   \fi:
2257 }
```

(End definition for `\quark_if_recursion_tail_stop:N`. This function is documented on page 45.)

`\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`

```
2258 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
2259 {
2260   \if_int_compare:w \__str_if_eq_x:nn
2261     { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
2262   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2263   \fi:
2264 }
2265 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
2266 {
2267   \if_int_compare:w \__str_if_eq_x:nn
2268     { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
2269   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2270   \else:
2271     \exp_after:wN \use_none:n
2272   \fi:
```

```

2273 }
2274 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
2275 \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.

```

2276 \cs_new:Npn \_quark_if_recursion_tail_break:NN #1#2
2277 {
2278   \if_meaning:w \q_recursion_tail #1
2279   \exp_after:wN #2
2280   \fi:
2281 }
2282 \cs_new:Npn \_quark_if_recursion_tail_break:nN #1#2
2283 {
2284   \if_int_compare:w \_str_if_eq_x:nn
2285   { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
2286   \exp_after:wN #2
2287   \fi:
2288 }

```

(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
`\quark_if_nil:NTF` \q_no_value as the first argument since the whole thing may otherwise loop if #1 is
`\quark_if_no_value_p:N` wrongly given a string like aabc instead of a single token.⁴

```

\quark_if_no_value_p:c 2289 \prg_new_conditional:Nnn \quark_if_nil:N { p, T , F , TF }
\quark_if_no_value:NTF 2290 {
\quark_if_no_value:cTF 2291   \if_meaning:w \q_nil #1
2292   \prg_return_true:
2293   \else:
2294   \prg_return_false:
2295   \fi:
2296 }
2297 \prg_new_conditional:Nnn \quark_if_no_value:N { p, T , F , TF }
2298 {
2299   \if_meaning:w \q_no_value #1
2300   \prg_return_true:
2301   \else:
2302   \prg_return_false:
2303   \fi:
2304 }
2305 \cs_generate_variant:Nn \quark_if_no_value_p:N { c }
2306 \cs_generate_variant:Nn \quark_if_no_value:NT { c }
2307 \cs_generate_variant:Nn \quark_if_no_value:NF { c }
2308 \cs_generate_variant:Nn \quark_if_no_value:NTF { c }

```

(End definition for \quark_if_nil:N. These functions are documented on page ??.)

⁴It may still loop in special circumstances however!

`\quark_if_nil_p:n` These are essentially `\str_if_eq:nn` tests but done directly.

```

\quark_if_nil_p:V 2309 \prg_new_conditional:Nnn \quark_if_nil:n { p, T , F , TF }
\quark_if_nil_p:o 2310 {
\quark_if_nil:nTF 2311   \if_int_compare:w \__str_if_eq_x:nn
\quark_if_nil:VTF 2312   { \exp_not:N \q_nil } { \exp_not:n {#1} } = \c_zero
\quark_if_nil:oTF 2313   \prg_return_true:
2314   \else:
2315   \prg_return_false:
\quark_if_no_value_p:n 2316   \fi:
\quark_if_no_value:nTF 2317 }
2318 \prg_new_conditional:Nnn \quark_if_no_value:n { p, T , F , TF }
2319 {
2320   \if_int_compare:w \__str_if_eq_x:nn
2321   { \exp_not:N \q_no_value } { \exp_not:n {#1} } = \c_zero
2322   \prg_return_true:
2323   \else:
2324   \prg_return_false:
2325   \fi:
2326 }
2327 \cs_generate_variant:Nn \quark_if_nil_p:n { V , o }
2328 \cs_generate_variant:Nn \quark_if_nil:nTF { V , o }
2329 \cs_generate_variant:Nn \quark_if_nil:nT { V , o }
2330 \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 44.)

`\q__tl_act_mark` These private quarks are needed by `l3tl`, but that is loaded before the quark module, hence their definition is deferred.

```

2331 \quark_new:N \q__tl_act_mark
2332 \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

```

2333 <@@=scan>

```

`\g__scan_marks_tl` The list of all scan marks currently declared.

```

2334 \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.

```

2335 \cs_new_protected:Npn \__scan_new:N #1
2336 {
2337   \tl_if_in:NnTF \g__scan_marks_tl { #1 }
2338   {
2339     \_msg_kernel_error:nnx { kernel } { scanmark-already-defined }
2340     { \token_to_str:N #1 }

```

```

2341     }
2342     {
2343         \tl_gput_right:Nn \g__scan_marks_tl {#1}
2344         \cs_new_eq:NN #1 \scan_stop:
2345     }
2346 }

```

(End definition for __scan_new:N.)

\s__stop We only declare one scan mark here, more can be defined by specific modules.

```

2347 \__scan_new:N \s__stop

```

(End definition for \s__stop. This variable is documented on page 47.)

__use_none_delimit_by_s__stop:w Similar to \use_none_delimit_by_q_stop:w.

```

2348 \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.

```

2349 \__scan_new:N \s__seq

```

(End definition for \s__seq. This variable is documented on page 116.)

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.

```

2350 \cs_new:Npn \quark_if_recursion_tail_break:N #1
2351 { \__quark_if_recursion_tail_break:NN #1 \prg_break: }
2352 \cs_new:Npn \quark_if_recursion_tail_break:n #1
2353 { \__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 ??.)

```

2354 </initex | package>

```

7 l3token implementation

```

2355 <*initex | package>
2356 <@@=token>

```

7.1 Character tokens

\char_set_catcode:nn Category code changes.
\char_value_catcode:n
\char_show_value_catcode:n

```

2357 \cs_new_protected:Npn \char_set_catcode:nn #1#2
2358 { \tex_catcode:D #1 = \__int_eval:w #2 \__int_eval_end: }
2359 \cs_new:Npn \char_value_catcode:n #1
2360 { \tex_the:D \tex_catcode:D \__int_eval:w #1 \__int_eval_end: }
2361 \cs_new_protected:Npn \char_show_value_catcode:n #1
2362 { \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 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
2363 \cs_new_protected:Npn \char_set_catcode_escape:N #1
2364   { \char_set_catcode:nn { '#1 } \c_zero }
2365 \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
2366   { \char_set_catcode:nn { '#1 } \c_one }
2367 \cs_new_protected:Npn \char_set_catcode_group_end:N #1
2368   { \char_set_catcode:nn { '#1 } \c_two }
2369 \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
2370   { \char_set_catcode:nn { '#1 } \c_three }
2371 \cs_new_protected:Npn \char_set_catcode_alignment:N #1
2372   { \char_set_catcode:nn { '#1 } \c_four }
2373 \cs_new_protected:Npn \char_set_catcode_end_line:N #1
2374   { \char_set_catcode:nn { '#1 } \c_five }
2375 \cs_new_protected:Npn \char_set_catcode_parameter:N #1
2376   { \char_set_catcode:nn { '#1 } \c_six }
2377 \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
2378   { \char_set_catcode:nn { '#1 } \c_seven }
2379 \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
2380   { \char_set_catcode:nn { '#1 } \c_eight }
2381 \cs_new_protected:Npn \char_set_catcode_ignore:N #1
2382   { \char_set_catcode:nn { '#1 } \c_nine }
2383 \cs_new_protected:Npn \char_set_catcode_space:N #1
2384   { \char_set_catcode:nn { '#1 } \c_ten }
2385 \cs_new_protected:Npn \char_set_catcode_letter:N #1
2386   { \char_set_catcode:nn { '#1 } \c_eleven }
2387 \cs_new_protected:Npn \char_set_catcode_other:N #1
2388   { \char_set_catcode:nn { '#1 } \c_twelve }
2389 \cs_new_protected:Npn \char_set_catcode_active:N #1
2390   { \char_set_catcode:nn { '#1 } \c_thirteen }
2391 \cs_new_protected:Npn \char_set_catcode_comment:N #1
2392   { \char_set_catcode:nn { '#1 } \c_fourteen }
2393 \cs_new_protected:Npn \char_set_catcode_invalid:N #1
2394   { \char_set_catcode:nn { '#1 } \c_fifteen }

```

(End definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 49.)

```

\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
2395 \cs_new_protected:Npn \char_set_catcode_escape:n #1
2396   { \char_set_catcode:nn {#1} \c_zero }
2397 \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
2398   { \char_set_catcode:nn {#1} \c_one }
2399 \cs_new_protected:Npn \char_set_catcode_group_end:n #1
2400   { \char_set_catcode:nn {#1} \c_two }
2401 \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
2402   { \char_set_catcode:nn {#1} \c_three }
2403 \cs_new_protected:Npn \char_set_catcode_alignment:n #1
2404   { \char_set_catcode:nn {#1} \c_four }

```

```

2405 \cs_new_protected:Npn \char_set_catcode_end_line:n #1
2406 { \char_set_catcode:nn {#1} \c_five }
2407 \cs_new_protected:Npn \char_set_catcode_parameter:n #1
2408 { \char_set_catcode:nn {#1} \c_six }
2409 \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
2410 { \char_set_catcode:nn {#1} \c_seven }
2411 \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
2412 { \char_set_catcode:nn {#1} \c_eight }
2413 \cs_new_protected:Npn \char_set_catcode_ignore:n #1
2414 { \char_set_catcode:nn {#1} \c_nine }
2415 \cs_new_protected:Npn \char_set_catcode_space:n #1
2416 { \char_set_catcode:nn {#1} \c_ten }
2417 \cs_new_protected:Npn \char_set_catcode_letter:n #1
2418 { \char_set_catcode:nn {#1} \c_eleven }
2419 \cs_new_protected:Npn \char_set_catcode_other:n #1
2420 { \char_set_catcode:nn {#1} \c_twelve }
2421 \cs_new_protected:Npn \char_set_catcode_active:n #1
2422 { \char_set_catcode:nn {#1} \c_thirteen }
2423 \cs_new_protected:Npn \char_set_catcode_comment:n #1
2424 { \char_set_catcode:nn {#1} \c_fourteen }
2425 \cs_new_protected:Npn \char_set_catcode_invalid:n #1
2426 { \char_set_catcode:nn {#1} \c_fifteen }

```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 49.)

Pretty repetitive, but necessary!

```

\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
2427 \cs_new_protected:Npn \char_set_mathcode:nn #1#2
2428 { \tex_mathcode:D #1 = \__int_eval:w #2 \__int_eval_end: }
2429 \cs_new:Npn \char_value_mathcode:n #1
2430 { \tex_the:D \tex_mathcode:D \__int_eval:w #1 \__int_eval_end: }
2431 \cs_new_protected:Npn \char_show_value_mathcode:n #1
2432 { \tex_showthe:D \tex_mathcode:D \__int_eval:w #1 \__int_eval_end: }
2433 \cs_new_protected:Npn \char_set_lccode:nn #1#2
2434 { \tex_lccode:D #1 = \__int_eval:w #2 \__int_eval_end: }
2435 \cs_new:Npn \char_value_lccode:n #1
2436 { \tex_the:D \tex_lccode:D \__int_eval:w #1 \__int_eval_end: }
2437 \cs_new_protected:Npn \char_show_value_lccode:n #1
2438 { \tex_showthe:D \tex_lccode:D \__int_eval:w #1 \__int_eval_end: }
2439 \cs_new_protected:Npn \char_set_uccode:nn #1#2
2440 { \tex_uccode:D #1 = \__int_eval:w #2 \__int_eval_end: }
2441 \cs_new:Npn \char_value_uccode:n #1
2442 { \tex_the:D \tex_uccode:D \__int_eval:w #1 \__int_eval_end: }
2443 \cs_new_protected:Npn \char_show_value_uccode:n #1
2444 { \tex_showthe:D \tex_uccode:D \__int_eval:w #1 \__int_eval_end: }
2445 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
2446 { \tex_sfcode:D #1 = \__int_eval:w #2 \__int_eval_end: }
2447 \cs_new:Npn \char_value_sfcode:n #1
2448 { \tex_the:D \tex_sfcode:D \__int_eval:w #1 \__int_eval_end: }
2449 \cs_new_protected:Npn \char_show_value_sfcode:n #1
2450 { \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 52.)

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
`\token_to_str:N` on page ??.)
`\token_to_str:c`

`\token_new:Nn` Creates a new token.

```
2451 \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 52.)

`\c_group_begin_token` We define these useful tokens. We have to do it by hand with the brace tokens for obvious
`\c_group_end_token` reasons.

```
2452 \cs_new_eq:NN \c_group_begin_token {
2453 \cs_new_eq:NN \c_group_end_token }
2454 \group_begin:
2455 \char_set_catcode_math_toggle:N \*
2456 \token_new:Nn \c_math_toggle_token { * }
2457 \char_set_catcode_alignment:N \*
2458 \token_new:Nn \c_alignment_token { * }
2459 \token_new:Nn \c_parameter_token { # }
2460 \token_new:Nn \c_math_superscript_token { ^ }
2461 \char_set_catcode_math_subscript:N \*
2462 \token_new:Nn \c_math_subscript_token { * }
2463 \token_new:Nn \c_space_token { ~ }
2464 \token_new:Nn \c_catcode_letter_token { a }
2465 \token_new:Nn \c_catcode_other_token { 1 }
2466 \group_end:
```

(End definition for `\c_group_begin_token` and others. These functions are documented on page 52.)

`\c_catcode_active_tl` Not an implicit token!

```
2467 \group_begin:
2468 \char_set_catcode_active:N \*
2469 \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
2470 \group_end:
```

(End definition for `\c_catcode_active_tl`. This variable is documented on page 52.)

`\l_char_active_seq` Two sequences for dealing with special characters. The first is characters which may
`\l_char_special_seq` be active, and contains the active characters themselves to allow easy redefinition. The
second longer list is for “special” characters more generally, and these are escaped so
that for example bulk code assignments can be carried out. In both cases, the order is by
ASCII character code (as is done in for example `\ExplSyntaxOn`). The only complication
is dealing with `_`, which requires the use of `\use:n` and `\use:nn`.

```
2471 \seq_new:N \l_char_active_seq
2472 \use:n
2473 {
2474 \group_begin:
```

```

2475 \char_set_catcode_active:N \"
2476 \char_set_catcode_active:N \$
2477 \char_set_catcode_active:N &
2478 \char_set_catcode_active:N ^
2479 \char_set_catcode_active:N _
2480 \char_set_catcode_active:N ~
2481 \use:nn
2482 {
2483   \group_end:
2484   \seq_set_split:Nnn \l_char_active_seq { }
2485 }
2486 }
2487 { { " $ & ^ _ ~ } } %$
2488 \seq_new:N \l_char_special_seq
2489 \seq_set_split:Nnn \l_char_special_seq { }
2490 { \ \" \# \$ \% \& \\\ ^\_ \{ \} \~ }

```

(End definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 52.)

7.3 Token conditionals

`\token_if_group_begin_p:N` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.
`\token_if_group_begin:N \underline{TF}`

```

2491 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
2492 {
2493   \if_catcode:w \exp_not:N #1 \c_group_begin_token
2494   \prg_return_true: \else: \prg_return_false: \fi:
2495 }

```

(End definition for `\token_if_group_begin:N`. These functions are documented on page 53.)

`\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}`

```

2496 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
2497 {
2498   \if_catcode:w \exp_not:N #1 \c_group_end_token
2499   \prg_return_true: \else: \prg_return_false: \fi:
2500 }

```

(End definition for `\token_if_group_end:N`. These functions are documented on page 53.)

`\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}`

```

2501 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
2502 {
2503   \if_catcode:w \exp_not:N #1 \c_math_toggle_token
2504   \prg_return_true: \else: \prg_return_false: \fi:
2505 }

```

(End definition for `\token_if_math_toggle:N`. These functions are documented on page 53.)

`\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 \mathbf{TF}`

```

2506 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
2507 {
2508   \if_catcode:w \exp_not:N #1 \c_alignment_token
2509   \prg_return_true: \else: \prg_return_false: \fi:
2510 }

```

(End definition for `\token_if_alignment:N`. These functions are documented on page 53.)

`\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 \mathbf{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.

```

2511 \group_begin:
2512 \cs_set_eq:NN \c_parameter_token \scan_stop:
2513 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
2514 {
2515   \if_catcode:w \exp_not:N #1 \c_parameter_token
2516   \prg_return_true: \else: \prg_return_false: \fi:
2517 }
2518 \group_end:

```

(End definition for `\token_if_parameter:N`. These functions are documented on page 54.)

`\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 \mathbf{TF}`

```

2519 \prg_new_conditional:Npnn \token_if_math_superscript:N #1 { p , T , F , TF }
2520 {
2521   \if_catcode:w \exp_not:N #1 \c_math_superscript_token
2522   \prg_return_true: \else: \prg_return_false: \fi:
2523 }

```

(End definition for `\token_if_math_superscript:N`. These functions are documented on page 54.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token` for this.
`\token_if_math_subscript:N \mathbf{TF}`

```

2524 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
2525 {
2526   \if_catcode:w \exp_not:N #1 \c_math_subscript_token
2527   \prg_return_true: \else: \prg_return_false: \fi:
2528 }

```

(End definition for `\token_if_math_subscript:N`. These functions are documented on page 54.)

`\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 \mathbf{TF}`

```

2529 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
2530 {
2531   \if_catcode:w \exp_not:N #1 \c_space_token
2532   \prg_return_true: \else: \prg_return_false: \fi:
2533 }

```

(End definition for `\token_if_space:N`. These functions are documented on page 54.)

`\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}`

```

2534 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
2535 {
2536   \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
2537   \prg_return_true: \else: \prg_return_false: \fi:
2538 }

```

(End definition for `\token_if_letter:N`. These functions are documented on page 54.)

`\token_if_other_p:N` Check if token is an other char token. We use the constant `\c_catcode_other_token` for this.

`\token_if_other:N \underline{TF}`

```

2539 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
2540 {
2541   \if_catcode:w \exp_not:N #1 \c_catcode_other_token
2542   \prg_return_true: \else: \prg_return_false: \fi:
2543 }

```

(End definition for `\token_if_other:N`. These functions are documented on page 54.)

`\token_if_active_p:N` Check if token is an active char token. We use the constant `\c_catcode_active_tl` for this. A technical point is that `\c_catcode_active_tl` is in fact a macro expanding to `\exp_not:N *`, where `*` is active.

`\token_if_active:N \underline{TF}`

```

2544 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
2545 {
2546   \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
2547   \prg_return_true: \else: \prg_return_false: \fi:
2548 }

```

(End definition for `\token_if_active:N`. These functions are documented on page 54.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.

`\token_if_eq_meaning:NN \underline{TF}`

```

2549 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
2550 {
2551   \if_meaning:w #1 #2
2552   \prg_return_true: \else: \prg_return_false: \fi:
2553 }

```

(End definition for `\token_if_eq_meaning:NN`. These functions are documented on page 55.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.

`\token_if_eq_catcode:NN \underline{TF}`

```

2554 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
2555 {
2556   \if_catcode:w \exp_not:N #1 \exp_not:N #2
2557   \prg_return_true: \else: \prg_return_false: \fi:
2558 }

```

(End definition for `\token_if_eq_catcode:NN`. These functions are documented on page 54.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.

`\token_if_eq_charcode:NN \underline{TF}`

```

2559 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
2560 {
2561   \if_charcode:w \exp_not:N #1 \exp_not:N #2
2562   \prg_return_true: \else: \prg_return_false: \fi:
2563 }

```

(End definition for `\token_if_eq_charcode:NN`. These functions are documented on page 54.)

`\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.

```

2564 \group_begin:
2565 \char_set_catcode_other:N \M
2566 \char_set_catcode_other:N \A
2567 \char_set_lccode:nn { '\; } { '\: }
2568 \char_set_lccode:nn { '\T } { '\T }
2569 \char_set_lccode:nn { '\F } { '\F }
2570 \tl_to_lowercase:n
2571 {
2572   \group_end:
2573   \prg_new_conditional:Npnn \token_if_macro:N #1 { p , T , F , TF }
2574   {
2575     \exp_after:wN \__token_if_macro_p:w
2576     \token_to_meaning:N #1 MA; \q_stop
2577   }
2578   \cs_new:Npn \__token_if_macro_p:w #1 MA #2 ; #3 \q_stop
2579   {
2580     \if_int_compare:w \__str_if_eq_x:nn { #2 } { cro } = \c_zero
2581     \prg_return_true:
2582   \else:
2583     \prg_return_false:
2584   \fi:
2585   }
2586 }
```

(End definition for `\token_if_macro:N`. These functions are documented on page 55.)

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. This follows the same pattern as
`\token_if_cs:N` for `\token_if_letter:N` etc. We use `\scan_stop:` for this.

```

2587 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
2588 {
2589   \if_catcode:w \exp_not:N #1 \scan_stop:
2590   \prg_return_true: \else: \prg_return_false: \fi:
```

```
2591 }
```

(End definition for `\token_if_cs:N`. These functions are documented on page 55.)

```
\token_if_expandable_p:N
\token_if_expandable:N $\underline{TF}$ 
```

Check if token is expandable. We use the fact that T_EX will temporarily convert `\exp_not:N` $\langle token \rangle$ into `\scan_stop:` if $\langle token \rangle$ is expandable. An undefined token is not considered as expandable. No problem nesting the conditionals, since the third #1 is only skipped if it is non-expandable (hence not part of T_EX's conditional apparatus).

```
2592 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
2593 {
2594   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
2595   \prg_return_false:
2596   \else:
2597     \if_cs_exist:N #1
2598     \prg_return_true:
2599     \else:
2600       \prg_return_false:
2601     \fi:
2602   \fi:
2603 }
```

(End definition for `\token_if_expandable:N`. These functions are documented on page 55.)

```
\token_if_chardef_p:N
\token_if_mathchardef_p:N
\token_if_dim_register_p:N
\token_if_int_register_p:N
```

Most of these functions have to check the meaning of the token in question so we need to do some checkups on which characters are output by `\token_to_meaning:N`. As usual, these characters have catcode 12 so we must do some serious substitutions in the code below...

```
2604 \group_begin:
2605   \char_set_lccode:nn { 'T } { 'T }
2606   \char_set_lccode:nn { 'F } { 'F }
2607   \char_set_lccode:nn { 'X } { 'n }
2608   \char_set_lccode:nn { 'Y } { 't }
2609   \char_set_lccode:nn { 'Z } { 'd }
2610   \tl_map_inline:nn { A C E G H I K L M O P R S U X Y Z R " }
2611     { \char_set_catcode:nn { '#1 } \c_twelve }
```

We convert the token list to lower case and restore the catcode and lowercase code changes.

```
\token_if_muskip_register:N $\underline{TF}$ 
\token_if_skip_register:N $\underline{TF}$ 
\token_if_toks_register:N $\underline{TF}$ 
```

```
2612 \tl_to_lowercase:n
2613 {
2614   \group_end:
```

```
\token_if_long_macro:N $\underline{TF}$ 
\token_if_protected_macro:N $\underline{TF}$ 
```

First up is checking if something has been defined with `\chardef` or `\mathchardef`. This is easy since T_EX 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).

```
2615 \prg_new_conditional:Npnn \token_if_chardef:N #1 { p , T , F , TF }
2616 {
2617   \__str_if_eq_x_return:nn
```

```
\__token_if_chardef:w
\__token_if_dim_register:w
\__token_if_int_register:w
\__token_if_muskip_register:w
\__token_if_skip_register:w
\__token_if_toks_register:w
\__token_if_protected_macro:w
\__token_if_long_macro:w
```



```

2618         {
2619             \exp_after:wN \__token_if_chardef:w
2620             \token_to_meaning:N #1 CHAR" \q_stop
2621         }
2622         { \token_to_str:N \char }
2623     }
2624 \prg_new_conditional:Npnn \token_if_mathchardef:N #1 { p , T , F , TF }
2625 {
2626     \__str_if_eq_x_return:nn
2627     {
2628         \exp_after:wN \__token_if_chardef:w
2629         \token_to_meaning:N #1 CHAR" \q_stop
2630     }
2631     { \token_to_str:N \mathchar }
2632 }
2633 \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`.

```

2634 \prg_new_conditional:Npnn \token_if_dim_register:N #1 { p , T , F , TF }
2635 {
2636     \if_meaning:w \tex_dimen:D #1
2637     \prg_return_false:
2638     \else:
2639         \if_meaning:w \tex_dimendef:D #1
2640         \prg_return_false:
2641         \else:
2642             \__str_if_eq_x_return:nn
2643             {
2644                 \exp_after:wN \__token_if_dim_register:w
2645                 \token_to_meaning:N #1 ZIMEX \q_stop
2646             }
2647             { \token_to_str:N \ }
2648         \fi:
2649     \fi:
2650 }
2651 \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`.

```

2652 \prg_new_conditional:Npnn \token_if_int_register:N #1 { p , T , F , TF }
2653 {
2654     % \token_if_chardef:NTF #1 { \prg_return_true: }
2655     % {
2656     %     \token_if_mathchardef:NTF #1 { \prg_return_true: }
2657     %     {
2658     \if_meaning:w \tex_count:D #1
2659     \prg_return_false:
2660     \else:
2661         \if_meaning:w \tex_countdef:D #1

```

```

2662         \prg_return_false:
2663     \else:
2664         \__str_if_eq_x_return:nn
2665         {
2666             \exp_after:wN \__token_if_int_register:w
2667             \token_to_meaning:N #1 COUXY \q_stop
2668         }
2669         { \token_to_str:N \ }
2670     \fi:
2671 \fi:
2672 %     }
2673 % }
2674 }
2675 \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.

```

2676 \prg_new_conditional:Npnn \token_if_muskip_register:N #1 { p , T , F , TF }
2677 {
2678     \if_meaning:w \tex_muskip:D #1
2679     \prg_return_false:
2680 \else:
2681     \if_meaning:w \tex_muskipdef:D #1
2682     \prg_return_false:
2683 \else:
2684     \__str_if_eq_x_return:nn
2685     {
2686         \exp_after:wN \__token_if_muskip_register:w
2687         \token_to_meaning:N #1 MUSKIP \q_stop
2688     }
2689     { \token_to_str:N \ }
2690 \fi:
2691 \fi:
2692 }
2693 \cs_new:Npn \__token_if_muskip_register:w #1 MUSKIP #2 \q_stop { #1 ~ }

```

Skip registers.

```

2694 \prg_new_conditional:Npnn \token_if_skip_register:N #1 { p , T , F , TF }
2695 {
2696     \if_meaning:w \tex_skip:D #1
2697     \prg_return_false:
2698 \else:
2699     \if_meaning:w \tex_skipdef:D #1
2700     \prg_return_false:
2701 \else:
2702     \__str_if_eq_x_return:nn
2703     {
2704         \exp_after:wN \__token_if_skip_register:w
2705         \token_to_meaning:N #1 SKIP \q_stop
2706     }
2707     { \token_to_str:N \ }
2708 \fi:

```

```

2709         \fi:
2710     }
2711     \cs_new:Npn \__token_if_skip_register:w #1 SKIP #2 \q_stop { #1 ~ }

```

Toks registers.

```

2712     \prg_new_conditional:Npnn \token_if_toks_register:N #1 { p , T , F , TF }
2713     {
2714         \if_meaning:w \tex_toks:D #1
2715         \prg_return_false:
2716     \else:
2717         \if_meaning:w \tex_toksdef:D #1
2718         \prg_return_false:
2719     \else:
2720         \__str_if_eq_x_return:nn
2721         {
2722             \exp_after:wN \__token_if_toks_register:w
2723             \token_to_meaning:N #1 YOKS \q_stop
2724         }
2725         { \token_to_str:N \ }
2726     \fi:
2727     \fi:
2728 }
2729 \cs_new:Npn \__token_if_toks_register:w #1 YOKS #2 \q_stop { #1 ~ }

```

Protected macros.

```

2730     \prg_new_conditional:Npnn \token_if_protected_macro:N #1
2731     { p , T , F , TF }
2732     {
2733         \__str_if_eq_x_return:nn
2734         {
2735             \exp_after:wN \__token_if_protected_macro:w
2736             \token_to_meaning:N #1 PROVECYEZ~MACRO \q_stop
2737         }
2738         { \token_to_str:N \ }
2739     }
2740     \cs_new:Npn \__token_if_protected_macro:w
2741     #1 PROVECYEZ~MACRO #2 \q_stop { #1 ~ }

```

Long macros and protected long macros share an auxiliary.

```

2742     \prg_new_conditional:Npnn \token_if_long_macro:N #1 { p , T , F , TF }
2743     {
2744         \__str_if_eq_x_return:nn
2745         {
2746             \exp_after:wN \__token_if_long_macro:w
2747             \token_to_meaning:N #1 LOXG~MACRO \q_stop
2748         }
2749         { \token_to_str:N \ }
2750     }
2751     \prg_new_conditional:Npnn \token_if_protected_long_macro:N #1
2752     { p , T , F , TF }
2753     {

```

```

2754     \__str_if_eq_x_return:nn
2755     {
2756         \exp_after:wN \__token_if_long_macro:w
2757         \token_to_meaning:N #1 LOXG-MACRO \q_stop
2758     }
2759     { \token_to_str:N \protected \token_to_str:N \ }
2760 }
2761 \cs_new:Npn \__token_if_long_macro:w #1 LOXG-MACRO #2 \q_stop { #1 ~ }

```

Finally the `\tl_to_lowercase:n` ends!

```

2762 }

```

(End definition for `\token_if_chardef:N` and others. These functions are documented on page 55.)

```

\token_if_primitive_p:N

```

```

\token_if_primitive:N $\text{TF}$ 

```

```

\__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.

```

2763 \tex_chardef:D \c_token_A_int = 'A ~ %
2764 \group_begin:
2765 \char_set_catcode_other:N \;
2766 \char_set_lccode:nn { '\; } { '\: }
2767 \char_set_lccode:nn { '\T } { '\T }
2768 \char_set_lccode:nn { '\F } { '\F }
2769 \tl_to_lowercase:n {
2770   \group_end:
2771   \prg_new_conditional:Npnn \token_if_primitive:N #1 { p , T , F , TF }
2772   {
2773     \token_if_macro:N $\text{TF}$  #1

```

```

2774     \prg_return_false:
2775     {
2776         \exp_after:wN \__token_if_primitive:NNw
2777         \token_to_meaning:N #1 ; ; ; \q_stop #1
2778     }
2779 }
2780 \cs_new:Npn \__token_if_primitive:NNw #1#2 #3 ; #4 \q_stop
2781 {
2782     \tl_if_empty:oTF { \__token_if_primitive_space:w #3 ~ }
2783     { \__token_if_primitive_loop:N #3 ; \q_stop }
2784     { \__token_if_primitive_nullfont:N }
2785 }
2786 }
2787 \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
2788 \cs_new:Npn \__token_if_primitive_nullfont:N #1
2789 {
2790     \if_meaning:w \tex_nullfont:D #1
2791     \prg_return_true:
2792     \else:
2793     \prg_return_false:
2794     \fi:
2795 }
2796 \cs_new:Npn \__token_if_primitive_loop:N #1
2797 {
2798     \if_int_compare:w '#1 < \c_token_A_int %
2799     \exp_after:wN \__token_if_primitive:Nw
2800     \exp_after:wN #1
2801     \else:
2802     \exp_after:wN \__token_if_primitive_loop:N
2803     \fi:
2804 }
2805 \cs_new:Npn \__token_if_primitive:Nw #1 #2 \q_stop
2806 {
2807     \if:w : #1
2808     \exp_after:wN \__token_if_primitive_undefined:N
2809     \else:
2810     \prg_return_false:
2811     \exp_after:wN \use_none:n
2812     \fi:
2813 }
2814 \cs_new:Npn \__token_if_primitive_undefined:N #1
2815 {
2816     \if_cs_exist:N #1
2817     \prg_return_true:
2818     \else:
2819     \prg_return_false:
2820     \fi:
2821 }

```

(End definition for \token_if_primitive:N. These functions are documented on page 56.)

7.4 Peeking ahead at the next token

2822 `<@@=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` 2823 `\cs_new_eq:NN \l_peek_token ?`

2824 `\cs_new_eq:NN \g_peek_token ?`

(End definition for `\l_peek_token`. This variable is documented on page 57.)

`\l__peek_search_token` The token to search for as an implicit token: cf. `\l__peek_search_tl`.

2825 `\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`.

2826 `\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` 2827 `\cs_new_nopar:Npn __peek_true:w { }`

`__peek_false:w` 2828 `\cs_new_nopar:Npn __peek_true_aux:w { }`

`__peek_tmp:w` 2829 `\cs_new_nopar:Npn __peek_false:w { }`

2830 `\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` 2831 `\cs_new_protected_nopar:Npn \peek_after:Nw`

2832 `{ \tex_futurelet:D \l_peek_token }`

2833 `\cs_new_protected_nopar:Npn \peek_gafter:Nw`

2834 `{ \tex_global:D \tex_futurelet:D \g_peek_token }`

(End definition for `\peek_after:Nw`. This function is documented on page 57.)

`__peek_true_remove:w` A function to remove the next token and then regain control.

2835 `\cs_new_protected:Npn __peek_true_remove:w`

2836 `{`

2837 `\group_align_safe_end:`

2838 `\tex_afterassignment:D __peek_true_aux:w`

2839 `\cs_set_eq:NN __peek_tmp:w`

2840 `}`

(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.

```

2841 \cs_new_protected:Npn \__peek_token_generic:NNTF #1#2#3#4
2842 {
2843   \cs_set_eq:NN \l__peek_search_token #2
2844   \tl_set:Nn \l__peek_search_tl {#2}
2845   \cs_set_nopar:Npx \__peek_true:w
2846   {
2847     \exp_not:N \group_align_safe_end:
2848     \exp_not:n {#3}
2849   }
2850   \cs_set_nopar:Npx \__peek_false:w
2851   {
2852     \exp_not:N \group_align_safe_end:
2853     \exp_not:n {#4}
2854   }
2855   \group_align_safe_begin:
2856   \peek_after:Nw #1
2857 }
2858 \cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
2859 { \__peek_token_generic:NNTF #1 #2 {#3} { } }
2860 \cs_new_protected:Npn \__peek_token_generic:NNF #1#2#3
2861 { \__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.

```

2862 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF #1#2#3#4
2863 {
2864   \cs_set_eq:NN \l__peek_search_token #2
2865   \tl_set:Nn \l__peek_search_tl {#2}
2866   \cs_set_eq:NN \__peek_true:w \__peek_true_remove:w
2867   \cs_set_nopar:Npx \__peek_true_aux:w { \exp_not:n {#3} }
2868   \cs_set_nopar:Npx \__peek_false:w
2869   {
2870     \exp_not:N \group_align_safe_end:
2871     \exp_not:n {#4}
2872   }
2873   \group_align_safe_begin:
2874   \peek_after:Nw #1
2875 }
2876 \cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
2877 { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
2878 \cs_new_protected:Npn \__peek_token_remove_generic:NNF #1#2#3
2879 { \__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.

```

2880 \cs_new_nopar:Npn \__peek_execute_branches_meaning:
2881 {
2882   \if_meaning:w \l_peek_token \l__peek_search_token
2883   \exp_after:wN \__peek_true:w
2884   \else:
2885     \exp_after:wN \__peek_false:w
2886   \fi:
2887 }

```

(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-L^AT_EX3 code) from blowing up. In the third case, `\l_peek_token` is good enough for the test, and we compare it again with the explicit search token. Just like the peek token, the search token may be of any of the three types above, hence the need to use the explicit token that was given to the peek function.

```

2888 \cs_new_nopar:Npn \__peek_execute_branches_catcode:
2889 { \if_catcode:w \__peek_execute_branches_catcode_aux: }
2890 \cs_new_nopar:Npn \__peek_execute_branches_charcode:
2891 { \if_charcode:w \__peek_execute_branches_catcode_aux: }
2892 \cs_new_nopar:Npn \__peek_execute_branches_catcode_aux:
2893 {
2894   \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
2895   \exp_after:wN \exp_after:wN
2896   \exp_after:wN \__peek_execute_branches_catcode_auxii:N
2897   \exp_after:wN \exp_not:N
2898   \else:
2899     \exp_after:wN \__peek_execute_branches_catcode_auxiii:
2900   \fi:
2901 }
2902 \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
2903 {

```



```

2904         \exp_not:N #1
2905         \exp_after:wN \exp_not:N \l__peek_search_tl
2906         \exp_after:wN \__peek_true:w
2907     \else:
2908         \exp_after:wN \__peek_false:w
2909     \fi:
2910     #1
2911 }
2912 \cs_new_nopar:Npn \__peek_execute_branches_catcode_auxiii:
2913 {
2914     \exp_not:N \l_peek_token
2915     \exp_after:wN \exp_not:N \l__peek_search_tl
2916     \exp_after:wN \__peek_true:w
2917 \else:
2918     \exp_after:wN \__peek_false:w
2919 \fi:
2920 }

```

(End definition for __peek_execute_branches_catcode: and __peek_execute_branches_charcode:. These functions are documented on page ??.)

__peek_ignore_spaces_execute_branches: This function removes one space token at a time, and calls __peek_execute_branches: when encountering the first non-space token. We directly use the primitive meaning test rather than \token_if_eq_meaning:NNTF because \l_peek_token may be an outer macro (coming from non-L^AT_EX3 packages). Spaces are removed using a side-effect of f-expansion: \tex_romannumeral:D -‘0 removes one space.

```

2921 \cs_new_protected_nopar:Npn \__peek_ignore_spaces_execute_branches:
2922 {
2923     \if_meaning:w \l_peek_token \c_space_token
2924     \exp_after:wN \peek_after:Nw
2925     \exp_after:wN \__peek_ignore_spaces_execute_branches:
2926     \tex_romannumeral:D -‘0
2927 \else:
2928     \exp_after:wN \__peek_execute_branches:
2929 \fi:
2930 }

```

(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.

```

2931 \group_begin:
2932 \cs_set:Npn \__peek_def:nnnn #1#2#3#4
2933 {
2934     \__peek_def:nnnnn {#1} {#2} {#3} {#4} { TF }
2935     \__peek_def:nnnnn {#1} {#2} {#3} {#4} { T }
2936     \__peek_def:nnnnn {#1} {#2} {#3} {#4} { F }
2937 }
2938 \cs_set:Npn \__peek_def:nnnnn #1#2#3#4#5

```

```

2939 {
2940   \cs_new_protected_nopar:cpx { #1 #5 }
2941   {
2942     \tl_if_empty:nF {#2}
2943     { \exp_not:n { \cs_set_eq:NN \__peek_execute_branches: #2 } }
2944     \exp_not:c { #3 #5 }
2945     \exp_not:n {#4}
2946   }
2947 }

```

(End definition for __peek_def:nnnn.)

\peek_catcode:N \underline{TF} With everything in place the definitions can take place. First for category codes.
\peek_catcode_ignore_spaces:N \underline{TF}
\peek_catcode_remove:N \underline{TF}
\peek_catcode_remove_ignore_spaces:N \underline{TF}

```

2948 \__peek_def:nnnn { peek_catcode:N }
2949 { }
2950 { __peek_token_generic:NN }
2951 { \__peek_execute_branches_catcode: }
2952 \__peek_def:nnnn { peek_catcode_ignore_spaces:N }
2953 { \__peek_execute_branches_catcode: }
2954 { __peek_token_generic:NN }
2955 { \__peek_ignore_spaces_execute_branches: }
2956 \__peek_def:nnnn { peek_catcode_remove:N }
2957 { }
2958 { __peek_token_remove_generic:NN }
2959 { \__peek_execute_branches_catcode: }
2960 \__peek_def:nnnn { peek_catcode_remove_ignore_spaces:N }
2961 { \__peek_execute_branches_catcode: }
2962 { __peek_token_remove_generic:NN }
2963 { \__peek_ignore_spaces_execute_branches: }

```

(End definition for \peek_catcode:N \underline{TF} and others. These functions are documented on page 58.)

\peek_charcode:N \underline{TF} Then for character codes.
\peek_charcode_ignore_spaces:N \underline{TF}
\peek_charcode_remove:N \underline{TF}
\peek_charcode_remove_ignore_spaces:N \underline{TF}

```

2964 \__peek_def:nnnn { peek_charcode:N }
2965 { }
2966 { __peek_token_generic:NN }
2967 { \__peek_execute_branches_charcode: }
2968 \__peek_def:nnnn { peek_charcode_ignore_spaces:N }
2969 { \__peek_execute_branches_charcode: }
2970 { __peek_token_generic:NN }
2971 { \__peek_ignore_spaces_execute_branches: }
2972 \__peek_def:nnnn { peek_charcode_remove:N }
2973 { }
2974 { __peek_token_remove_generic:NN }
2975 { \__peek_execute_branches_charcode: }
2976 \__peek_def:nnnn { peek_charcode_remove_ignore_spaces:N }
2977 { \__peek_execute_branches_charcode: }
2978 { __peek_token_remove_generic:NN }
2979 { \__peek_ignore_spaces_execute_branches: }

```

(End definition for \peek_charcode:N \underline{TF} and others. These functions are documented on page 59.)

`\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.

```

2980 \__peek_def:nnnn { peek_meaning:N }
2981 { }
2982 { __peek_token_generic:NN }
2983 { \__peek_execute_branches_meaning: }
2984 \__peek_def:nnnn { peek_meaning_ignore_spaces:N }
2985 { \__peek_execute_branches_meaning: }
2986 { __peek_token_generic:NN }
2987 { \__peek_ignore_spaces_execute_branches: }
2988 \__peek_def:nnnn { peek_meaning_remove:N }
2989 { }
2990 { __peek_token_remove_generic:NN }
2991 { \__peek_execute_branches_meaning: }
2992 \__peek_def:nnnn { peek_meaning_remove_ignore_spaces:N }
2993 { \__peek_execute_branches_meaning: }
2994 { __peek_token_remove_generic:NN }
2995 { \__peek_ignore_spaces_execute_branches: }
2996 \group_end:

```

(End definition for `\peek_meaning:N`TF and others. These functions are documented on page 59.)

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.

```

2997 \exp_args:Nno \use:nn
2998 { \cs_new:Npn \__peek_get_prefix_arg_replacement:wN #1 }
2999 { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
3000 { #4 {#1} {#2} {#3} }
3001 \cs_new:Npn \token_get_prefix_spec:N #1
3002 {
3003   \token_if_macro:NTF #1
3004   {
3005     \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3006     \token_to_meaning:N #1 \q_stop \use_i:nnn
3007   }
3008   { \scan_stop: }
3009 }
3010 \cs_new:Npn \token_get_arg_spec:N #1
3011 {
3012   \token_if_macro:NTF #1
3013   {
3014     \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3015     \token_to_meaning:N #1 \q_stop \use_ii:nnn
3016   }

```

```

3017     { \scan_stop: }
3018   }
3019   \cs_new:Npn \token_get_replacement_spec:N #1
3020   {
3021     \token_if_macro:NTF #1
3022     {
3023       \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3024       \token_to_meaning:N #1 \q_stop \use_iii:nnn
3025     }
3026     { \scan_stop: }
3027   }

```

(End definition for \token_get_prefix_spec:N. This function is documented on page 60.)

```

3028 \</initex | package>

```

8 l3int implementation

```

3029 \<*initex | package>
3030 \<@@=int>

```

The following test files are used for this code: m3int001,m3int002,m3int03.

\c_max_register_int Done in l3basics.
(End definition for \c_max_register_int. This variable is documented on page 72.)

__int_to_roman:w Done in l3basics.
\if_int_compare:w (End definition for __int_to_roman:w. This function is documented on page 73.)

\or: Done in l3basics.
(End definition for \or:. This function is documented on page 73.)

__int_value:w Here are the remaining primitives for number comparisons and expressions.
__int_eval:w 3031 \cs_new_eq:NN __int_value:w \tex_number:D
__int_eval_end: 3032 \cs_new_eq:NN __int_eval:w \etex_numexpr:D
\if_int_odd:w 3033 \cs_new_eq:NN __int_eval_end: \tex_relax:D
\if_case:w 3034 \cs_new_eq:NN \if_int_odd:w \tex_ifodd:D
3035 \cs_new_eq:NN \if_case:w \tex_ifcase:D
(End definition for __int_value:w. This function is documented on page 73.)

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.

```

3036 \<*initex>
3037 \cs_set:Npn \int_eval:n #1 { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
3038 \</initex>
3039 \<*package>
3040 \cs_new:Npn \int_eval:n #1 { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
3041 \</package>

```

(End definition for `\int_eval:n`. This function is documented on page 61.)

```

\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
\int_min:nn
\__int_maxmin:wwN
3042 \cs_new:Npn \int_abs:n #1
3043 {
3044   \__int_value:w \exp_after:wN \__int_abs:N
3045   \int_use:N \__int_eval:w #1 \exp_after:wN \int_eval_end:
3046   \exp_stop_f:
3047 }
3048 \cs_new:Npn \__int_abs:N #1
3049 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
3050 \cs_set:Npn \int_max:nn #1#2
3051 {
3052   \__int_value:w \exp_after:wN \__int_maxmin:wwN
3053   \int_use:N \__int_eval:w #1 \exp_after:wN ;
3054   \int_use:N \__int_eval:w #2 ;
3055   >
3056   \exp_stop_f:
3057 }
3058 \cs_set:Npn \int_min:nn #1#2
3059 {
3060   \__int_value:w \exp_after:wN \__int_maxmin:wwN
3061   \int_use:N \__int_eval:w #1 \exp_after:wN ;
3062   \int_use:N \__int_eval:w #2 ;
3063   <
3064   \exp_stop_f:
3065 }
3066 \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
3067 {
3068   \if_int_compare:w #1 #3 #2 ~
3069   #1
3070   \else:
3071   #2
3072   \fi:
3073 }

```

(End definition for `\int_abs:n`. This function is documented on page 62.)

`\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.

```

3074 \cs_new:Npn \int_div_truncate:nn #1#2
3075 {

```

```

3076 \int_use:N \__int_eval:w
3077 \exp_after:wN \__int_div_truncate:NwNw
3078 \int_use:N \__int_eval:w #1 \exp_after:wN ;
3079 \int_use:N \__int_eval:w #2 ;
3080 \__int_eval_end:
3081 }
3082 \cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
3083 {
3084 \if_meaning:w 0 #1
3085 \c_zero
3086 \else:
3087 (
3088 #1#2
3089 \if_meaning:w - #1 + \else: - \fi:
3090 ( \if_meaning:w - #3 - \fi: #3#4 - \c_one ) / \c_two
3091 )
3092 \fi:
3093 / #3#4
3094 }

```

For the sake of completeness:

```

3095 \cs_new:Npn \int_div_round:nn #1#2
3096 { \__int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }

```

Finally there's the modulus operation.

```

3097 \cs_new:Npn \int_mod:nn #1#2
3098 {
3099 \__int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
3100 \__int_value:w \__int_eval:w #1 \exp_after:wN ;
3101 \__int_value:w \__int_eval:w #2 ;
3102 \__int_eval_end:
3103 }
3104 \cs_new:Npn \__int_mod:ww #1; #2;
3105 { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }

```

(End definition for `\int_div_truncate:nn`. This function is documented on page 62.)

8.2 Creating and initialising integers

`\int_new:N` Two ways to do this: one for the format and one for the L^AT_EX 2_ε package. In plain T_EX, `\int_new:c` `\newcount` (and other allocators) are `\outer:` to allow the code here to work in “generic” mode this is therefore accessed by name. (The same applies to `\newbox`, `\newdimen` and so on.)

```

3106 ⟨*package⟩
3107 \cs_new_protected:Npn \int_new:N #1
3108 {
3109 \__chk_if_free_cs:N #1
3110 \cs:w newcount \cs_end: #1
3111 }
3112 ⟨/package⟩
3113 \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`

`\c__max_constdef_int`

```

3114 \cs_new_protected:Npn \int_const:Nn #1#2
3115 {
3116   \int_compare:nNnTF {#2} > \c_minus_one
3117   {
3118     \int_compare:nNnTF {#2} > \c__max_constdef_int
3119     {
3120       \int_new:N #1
3121       \int_gset:Nn #1 {#2}
3122     }
3123     {
3124       \__chk_if_free_cs:N #1
3125       \tex_global:D \__int_constdef:Nw #1 =
3126       \__int_eval:w #2 \__int_eval_end:
3127     }
3128   }
3129   {
3130     \int_new:N #1
3131     \int_gset:Nn #1 {#2}
3132   }
3133 }
3134 \cs_generate_variant:Nn \int_const:Nn { c }
3135 \pdfTeX_if_engine:TF
3136 {
3137   \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
3138   \tex_mathchardef:D \c__max_constdef_int 32 767 ~
3139 }
3140 {
3141   \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D
3142   \tex_chardef:D \c__max_constdef_int 1 114 111 ~
3143 }

```

(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`

`\int_gzero:N`

`\int_gzero:c`

```

3144 \cs_new_protected:Npn \int_zero:N #1 { #1 = \c_zero }
3145 \cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero }
3146 \cs_generate_variant:Nn \int_zero:N { c }
3147 \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`

`\int_gzero_new:N`

`\int_gzero_new:c`

```

3148 \cs_new_protected:Npn \int_zero_new:N #1
3149 { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
3150 \cs_new_protected:Npn \int_gzero_new:N #1
3151 { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
3152 \cs_generate_variant:Nn \int_zero_new:N { c }
3153 \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` 3154 `\cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }`

`\int_set_eq:Nc` 3155 `\cs_generate_variant:Nn \int_set_eq:NN { c }`

`\int_set_eq:cc` 3156 `\cs_generate_variant:Nn \int_set_eq:NN { Nc , cc }`

`\int_gset_eq:NN` 3157 `\cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }`

`\int_gset_eq:cN` 3158 `\cs_generate_variant:Nn \int_gset_eq:NN { c }`

`\int_gset_eq:Nc` 3159 `\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` 3160 `\prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N { TF , T , F , p }`

`\int_if_exist:NTF` 3161 `\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` 3162 `\cs_new_protected:Npn \int_add:Nn #1#2`

`\int_gadd:Nn` 3163 `{ \tex_advance:D #1 by __int_eval:w #2 __int_eval_end: }`

`\int_gadd:cn` 3164 `\cs_new_protected:Npn \int_sub:Nn #1#2`

`\int_sub:Nn` 3165 `{ \tex_advance:D #1 by - __int_eval:w #2 __int_eval_end: }`

`\int_sub:cn` 3166 `\cs_new_protected_nopar:Npn \int_gadd:Nn`

`\int_gsub:Nn` 3167 `{ \tex_global:D \int_add:Nn }`

`\int_gsub:cn` 3168 `\cs_new_protected_nopar:Npn \int_gsub:Nn`

3169 `{ \tex_global:D \int_sub:Nn }`

3170 `\cs_generate_variant:Nn \int_add:Nn { c }`

3171 `\cs_generate_variant:Nn \int_gadd:Nn { c }`

3172 `\cs_generate_variant:Nn \int_sub:Nn { c }`

3173 `\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` 3174 `\cs_new_protected:Npn \int_incr:N #1`

`\int_gincr:N` 3175 `{ \tex_advance:D #1 \c_one }`

`\int_gincr:c` 3176 `\cs_new_protected:Npn \int_decr:N #1`

`\int_decr:N` 3177 `{ \tex_advance:D #1 \c_minus_one }`

`\int_decr:c` 3178 `\cs_new_protected_nopar:Npn \int_gincr:N`

`\int_gdecr:N` 3179 `{ \tex_global:D \int_incr:N }`

`\int_gdecr:c` 3180 `\cs_new_protected_nopar:Npn \int_gdecr:N`

3181 `{ \tex_global:D \int_decr:N }`

3182 `\cs_generate_variant:Nn \int_incr:N { c }`

3183 `\cs_generate_variant:Nn \int_decr:N { c }`

3184 `\cs_generate_variant:Nn \int_gincr:N { c }`

3185 `\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
\int_gset:Nn
\int_gset:cn
3186 \cs_new_protected:Npn \int_set:Nn #1#2
3187 { #1 ~ \__int_eval:w #2\__int_eval_end: }
3188 \cs_new_protected_nopar:Npn \int_gset:Nn { \tex_global:D \int_set:Nn }
3189 \cs_generate_variant:Nn \int_set:Nn { c }
3190 \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
3191 \cs_new_eq:NN \int_use:N \tex_the:D
3192 \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.

```

3193 \cs_new_protected_nopar:Npn \__prg_compare_error:
3194 {
3195   \if_int_compare:w \c_zero \c_zero \fi:
3196   =
3197   \__prg_compare_error:
3198 }
3199 \cs_new:Npn \__prg_compare_error:Nw
3200 #1#2 \q_stop
3201 {
3202   { }
3203   \c_zero \fi:
3204   \msg_kernel_expandable_error:nnn
3205     { kernel } { unknown-comparison } {#1}
3206   \prg_return_false:
3207 }

```

(End definition for `__prg_compare_error:` and `__prg_compare_error:Nw`.)

`\int_compare_p:n` 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

```

\__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

```

```

    <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.

```

3208 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
3209 {
3210     \exp_after:wN \__int_compare:w
3211     \int_use:N \__int_eval:w #1 \__prg_compare_error:
3212 }
3213 \cs_new:Npn \__int_compare:w #1 \__prg_compare_error:
3214 {
3215     \exp_after:wN \if_false: \__int_value:w
3216     \__int_compare:Nw #1 e { = nd_ } \q_stop
3217 }

```

The goal here is to find an *<operand>* and a *<comparison>*. The *<operand>* is already evaluated, but we cannot yet grab it as an argument. To access the following relation symbol, we remove the number by applying `__int_to_roman:w`, after making sure that the argument becomes non-positive: its roman numeral representation is then empty. Then probe the first two tokens with `__int_compare:NNw` to determine the relation symbol, building a control sequence from it (`\token_to_str:N` gives better errors if `#1` is not a character). All the extended forms have an extra `=` hence the test for that as a second token. If the relation symbol is unknown, then the control sequence is turned by \TeX into `\scan_stop:`, ignored thanks to `\unexpanded`, and `__prg_compare_error:Nw` raises an error.

```

3218 \cs_new:Npn \__int_compare:Nw #1#2 \q_stop
3219 {
3220     \exp_after:wN \__int_compare:NNw
3221     \__int_to_roman:w - 0 #2 \q_mark
3222     #1#2 \q_stop
3223 }
3224 \cs_new:Npn \__int_compare:NNw #1#2#3 \q_mark
3225 {
3226     \etex_unexpanded:D

```

```

3227 \use:c
3228 {
3229   __int_compare_ \token_to_str:N #1
3230   \if_meaning:w = #2 = \fi:
3231   :NNw
3232 }
3233 \__prg_compare_error:Nw #1
3234 }

```

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.

```

3235 \cs_new:cpn { __int_compare_end_=:NNw } #1#2#3 e #4 \q_stop
3236 {
3237   {#3} \exp_stop_f:
3238   \prg_return_false: \else: \prg_return_true: \fi:
3239 }
3240 \cs_new:Npn \__int_compare:nnN #1#2#3
3241 {
3242   {#2} \exp_stop_f:
3243   \prg_return_false: \exp_after:wN \use_none_delimit_by_q_stop:w
3244   \fi:
3245   #1 #2 #3 \exp_after:wN \__int_compare:Nw \__int_value:w \__int_eval:w
3246 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `__prg_compare_error:Nw` *<token>* responsible for error detection.

```

3247 \cs_new:cpn { __int_compare=:NNw } #1#2#3 =
3248 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
3249 \cs_new:cpn { __int_compare:<:NNw } #1#2#3 <
3250 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
3251 \cs_new:cpn { __int_compare:>:NNw } #1#2#3 >
3252 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
3253 \cs_new:cpn { __int_compare==:NNw } #1#2#3 ==
3254 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
3255 \cs_new:cpn { __int_compare!=:NNw } #1#2#3 !=
3256 { \__int_compare:nnN { \if_int_compare:w } {#3} = }
3257 \cs_new:cpn { __int_compare<=:NNw } #1#2#3 <=
3258 { \__int_compare:nnN { \if_int_compare:w } {#3} > }
3259 \cs_new:cpn { __int_compare>=:NNw } #1#2#3 >=
3260 { \__int_compare:nnN { \if_int_compare:w } {#3} < }

```

(End definition for \int_compare:n. These functions are documented on page 65.)

```

\int_compare_p:nNn
\int_compare:nNnTF
3261 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
3262 {
3263   \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
3264     \prg_return_true:
3265   \else:
3266     \prg_return_false:
3267   \fi:
3268 }

```

(End definition for \int_compare:nNn. These functions are documented on page 64.)

```

\int_case:nn
\int_case:nnTF
\__int_case:nnTF
\__int_case:nw
\__int_case_end:nw
3269 \cs_new:Npn \int_case:nnTF #1
3270 {
3271   \tex_romannumeral:D
3272   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
3273 }
3274 \cs_new:Npn \int_case:nnT #1#2#3
3275 {
3276   \tex_romannumeral:D
3277   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
3278 }
3279 \cs_new:Npn \int_case:nnF #1#2
3280 {
3281   \tex_romannumeral:D
3282   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
3283 }
3284 \cs_new:Npn \int_case:nn #1#2
3285 {
3286   \tex_romannumeral:D
3287   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
3288 }
3289 \cs_new:Npn \__int_case:nnTF #1#2#3#4
3290 { \__int_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
3291 \cs_new:Npn \__int_case:nw #1#2#3
3292 {
3293   \int_compare:nNnTF {#1} = {#2}
3294   { \__int_case_end:nw {#3} }
3295   { \__int_case:nw {#1} }
3296 }
3297 \cs_new_eq:NN \__int_case_end:nw \__prg_case_end:nw

```

(End definition for \int_case:nn. This function is documented on page 66.)

```

\int_if_odd_p:n
\int_if_odd:nTF
\int_if_even_p:n
\int_if_even:nTF
3298 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
3299 {

```

A predicate function.

```

3300     \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
3301     \prg_return_true:
3302     \else:
3303     \prg_return_false:
3304     \fi:
3305   }
3306 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
3307 {
3308   \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
3309   \prg_return_false:
3310   \else:
3311   \prg_return_true:
3312   \fi:
3313 }

```

(End definition for \int_if_odd:n. These functions are documented on page 66.)

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_while_do:nn
\int_until_do:nn
\int_do_while:nn
\int_do_until:nn
3314 \cs_new:Npn \int_while_do:nn #1#2
3315 {
3316   \int_compare:nT {#1}
3317   {
3318     #2
3319     \int_while_do:nn {#1} {#2}
3320   }
3321 }
3322 \cs_new:Npn \int_until_do:nn #1#2
3323 {
3324   \int_compare:nF {#1}
3325   {
3326     #2
3327     \int_until_do:nn {#1} {#2}
3328   }
3329 }
3330 \cs_new:Npn \int_do_while:nn #1#2
3331 {
3332   #2
3333   \int_compare:nT {#1}
3334   { \int_do_while:nn {#1} {#2} }
3335 }
3336 \cs_new:Npn \int_do_until:nn #1#2
3337 {
3338   #2
3339   \int_compare:nF {#1}
3340   { \int_do_until:nn {#1} {#2} }
3341 }

```

(End definition for \int_while_do:nn. This function is documented on page 67.)

```

\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.

```

3342 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
3343 {
3344   \int_compare:nNnT {#1} #2 {#3}
3345   {
3346     #4
3347     \int_while_do:nNnn {#1} #2 {#3} {#4}
3348   }
3349 }
3350 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
3351 {
3352   \int_compare:nNnF {#1} #2 {#3}
3353   {
3354     #4
3355     \int_until_do:nNnn {#1} #2 {#3} {#4}
3356   }
3357 }
3358 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
3359 {
3360   #4
3361   \int_compare:nNnT {#1} #2 {#3}
3362   { \int_do_while:nNnn {#1} #2 {#3} {#4} }
3363 }
3364 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
3365 {
3366   #4
3367   \int_compare:nNnF {#1} #2 {#3}
3368   { \int_do_until:nNnn {#1} #2 {#3} {#4} }
3369 }

```

(End definition for `\int_while_do:nNnn`. This function is documented on page 66.)

8.7 Integer step functions

```

\int_step_function:nnnN
  \__int_step:wwwN
  \__int_step:NnnnN

```

Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

3370 \cs_new:Npn \int_step_function:nnnN #1#2#3
3371 {
3372   \exp_after:wN \__int_step:wwwN
3373   \int_use:N \__int_eval:w #1 \exp_after:wN ;
3374   \int_use:N \__int_eval:w #2 \exp_after:wN ;
3375   \int_use:N \__int_eval:w #3 ;
3376 }
3377 \cs_new:Npn \__int_step:wwwN #1; #2; #3; #4
3378 {
3379   \int_compare:nNnTF {#2} > \c_zero

```

```

3380     { \__int_step:NnnnN > }
3381     {
3382         \int_compare:nNnTF {#2} = \c_zero
3383         {
3384             \__msg_kernel_expandable_error:nnn { kernel } { zero-step } {#4}
3385             \use_none:nnnn
3386         }
3387         { \__int_step:NnnnN < }
3388     }
3389     {#1} {#2} {#3} #4
3390 }
3391 \cs_new:Npn \__int_step:NnnnN #1#2#3#4#5
3392 {
3393     \int_compare:nNnF {#2} #1 {#4}
3394     {
3395         #5 {#2}
3396         \exp_args:Nnf \__int_step:NnnnN
3397             #1 { \int_eval:n { #2 + #3 } } {#3} {#4} #5
3398     }
3399 }

```

(End definition for `\int_step_function:nnnN`. This function is documented on page 68.)

```

\int_step_inline:nnnn
\int_step_variable:nnnNn
  \__int_step:NNnnnn

```

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\int_step_function:nnnN`. We put a `__prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so no breaking function will recognize this break point as its own.

```

3400 \cs_new_protected_nopar:Npn \int_step_inline:nnnn
3401 {
3402     \int_gincr:N \g__prg_map_int
3403     \exp_args:NNc \__int_step:NNnnnn
3404     \cs_gset_nopar:Npn
3405         { __prg_map_ \int_use:N \g__prg_map_int :w }
3406 }
3407 \cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
3408 {
3409     \int_gincr:N \g__prg_map_int
3410     \exp_args:NNc \__int_step:NNnnnn
3411     \cs_gset_nopar:Npx
3412         { __prg_map_ \int_use:N \g__prg_map_int :w }
3413         {#1}{#2}{#3}
3414         {
3415             \tl_set:Nn \exp_not:N #4 {##1}
3416             \exp_not:n {#5}
3417         }
3418 }
3419 \cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
3420 {

```

```

3421     #1 #2 ##1 {#6}
3422     \int_step_function:nnnN {#3} {#4} {#5} #2
3423     \__prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__prg_map_int }
3424 }

```

(End definition for \int_step_inline:nnnn. This function is documented on page 68.)

8.8 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```

3425 \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 68.)

`\int_to_symbols:nnn` For conversion of integers to arbitrary symbols the method is in general as follows. The
`__int_to_symbols:nnnn` 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.

```

3426 \cs_new:Npn \int_to_symbols:nnn #1#2#3
3427 {
3428   \int_compare:nNnTF {#1} > {#2}
3429   {
3430     \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
3431     {
3432       \int_case:nn
3433       { 1 + \int_mod:nn { #1 - 1 } {#2} }
3434       {#3}
3435     }
3436     {#1} {#2} {#3}
3437   }
3438   { \int_case:nn {#1} {#3} }
3439 }
3440 \cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
3441 {
3442   \exp_args:Nf \int_to_symbols:nnn
3443   { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
3444   #1
3445 }

```

(End definition for \int_to_symbols:nnn. This function is documented on page 69.)

`\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.

```

3446 \cs_new:Npn \int_to_alph:n #1
3447 {
3448   \int_to_symbols:nnn {#1} { 26 }
3449   {
3450     { 1 } { a }

```



```

3451         { 2 } { b }
3452         { 3 } { c }
3453         { 4 } { d }
3454         { 5 } { e }
3455         { 6 } { f }
3456         { 7 } { g }
3457         { 8 } { h }
3458         { 9 } { i }
3459         { 10 } { j }
3460         { 11 } { k }
3461         { 12 } { l }
3462         { 13 } { m }
3463         { 14 } { n }
3464         { 15 } { o }
3465         { 16 } { p }
3466         { 17 } { q }
3467         { 18 } { r }
3468         { 19 } { s }
3469         { 20 } { t }
3470         { 21 } { u }
3471         { 22 } { v }
3472         { 23 } { w }
3473         { 24 } { x }
3474         { 25 } { y }
3475         { 26 } { z }
3476     }
3477 }
3478 \cs_new:Npn \int_to_Alph:n #1
3479 {
3480     \int_to_symbols:nnn {#1} { 26 }
3481     {
3482         { 1 } { A }
3483         { 2 } { B }
3484         { 3 } { C }
3485         { 4 } { D }
3486         { 5 } { E }
3487         { 6 } { F }
3488         { 7 } { G }
3489         { 8 } { H }
3490         { 9 } { I }
3491         { 10 } { J }
3492         { 11 } { K }
3493         { 12 } { L }
3494         { 13 } { M }
3495         { 14 } { N }
3496         { 15 } { O }
3497         { 16 } { P }
3498         { 17 } { Q }
3499         { 18 } { R }
3500         { 19 } { S }

```

```

3501         { 20 } { T }
3502         { 21 } { U }
3503         { 22 } { V }
3504         { 23 } { W }
3505         { 24 } { X }
3506         { 25 } { Y }
3507         { 26 } { Z }
3508     }
3509 }

```

(End definition for `\int_to_alph:n` and `\int_to_Alph:n`. These functions are documented on page 69.)

```

\int_to_base:nn Converting from base ten (#1) to a second base (#2) starts with computing #1: if it is
\int_to_Base:nn a complicated calculation, we shouldn't perform it twice. Then check the sign, store it,
\__int_to_base:nn either - or \c_empty_tl, and feed the absolute value to the next auxiliary function.
\__int_to_Base:nn
\__int_to_base:nnN 3510 \cs_new:Npn \int_to_base:nn #1
\__int_to_Base:nnN 3511 { \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } }
\__int_to_base:nnN 3512 \cs_new:Npn \int_to_Base:nn #1
\__int_to_Base:nnN 3513 { \exp_args:Nf \__int_to_Base:nn { \int_eval:n {#1} } }
\__int_to_Base:nnN 3514 \cs_new:Npn \__int_to_base:nn #1#2
\__int_to_letter:n 3515 {
\__int_to_Letter:n 3516 \int_compare:nNnTF {#1} < \c_zero
3517 { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
3518 { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
3519 }
3520 \cs_new:Npn \__int_to_Base:nn #1#2
3521 {
3522 \int_compare:nNnTF {#1} < \c_zero
3523 { \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - }
3524 { \__int_to_Base:nnN {#1} {#2} \c_empty_tl }
3525 }

```

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.

```

3526 \cs_new:Npn \__int_to_base:nnN #1#2#3
3527 {
3528 \int_compare:nNnTF {#1} < {#2}
3529 { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
3530 {
3531 \exp_args:Nf \__int_to_base:nnnN
3532 { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
3533 {#1}
3534 {#2}
3535 #3
3536 }
3537 }

```

```

3538 \cs_new:Npn \__int_to_base:nnnN #1#2#3#4
3539 {
3540   \exp_args:Nf \__int_to_base:nnN
3541   { \int_div_truncate:nn {#2} {#3} }
3542   {#3}
3543   #4
3544   #1
3545 }
3546 \cs_new:Npn \__int_to_Base:nnN #1#2#3
3547 {
3548   \int_compare:nNnTF {#1} < {#2}
3549   { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
3550   {
3551     \exp_args:Nf \__int_to_Base:nnnN
3552     { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
3553     {#1}
3554     {#2}
3555     #3
3556   }
3557 }
3558 \cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
3559 {
3560   \exp_args:Nf \__int_to_Base:nnN
3561   { \int_div_truncate:nn {#2} {#3} }
3562   {#3}
3563   #4
3564   #1
3565 }

```

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:`.

```

3566 \cs_new:Npn \__int_to_letter:n #1
3567 {
3568   \exp_after:wN \exp_after:wN
3569   \if_case:w \__int_eval:w #1 - \c_ten \__int_eval_end:
3570   a
3571   \or: b
3572   \or: c
3573   \or: d
3574   \or: e
3575   \or: f
3576   \or: g
3577   \or: h
3578   \or: i
3579   \or: j
3580   \or: k

```

```

3581     \or: l
3582     \or: m
3583     \or: n
3584     \or: o
3585     \or: p
3586     \or: q
3587     \or: r
3588     \or: s
3589     \or: t
3590     \or: u
3591     \or: v
3592     \or: w
3593     \or: x
3594     \or: y
3595     \or: z
3596     \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
3597     \fi:
3598 }
3599 \cs_new:Npn \__int_to_Letter:n #1
3600 {
3601     \exp_after:wN \exp_after:wN
3602     \if_case:w \__int_eval:w #1 - \c_ten \__int_eval_end:
3603         A
3604     \or: B
3605     \or: C
3606     \or: D
3607     \or: E
3608     \or: F
3609     \or: G
3610     \or: H
3611     \or: I
3612     \or: J
3613     \or: K
3614     \or: L
3615     \or: M
3616     \or: N
3617     \or: O
3618     \or: P
3619     \or: Q
3620     \or: R
3621     \or: S
3622     \or: T
3623     \or: U
3624     \or: V
3625     \or: W
3626     \or: X
3627     \or: Y
3628     \or: Z
3629     \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
3630     \fi:

```

3631 }

(End definition for \int_to_base:nn and \int_to_Base:nn. These functions are documented on page 70.)

\int_to_bin:n Wrappers around the generic function.

```

\int_to_hex:n 3632 \cs_new:Npn \int_to_bin:n #1
\int_to_Hex:n 3633 { \int_to_base:nn {#1} { 2 } }
\int_to_oct:n 3634 \cs_new:Npn \int_to_hex:n #1
3635 { \int_to_base:nn {#1} { 16 } }
3636 \cs_new:Npn \int_to_Hex:n #1
3637 { \int_to_Base:nn {#1} { 16 } }
3638 \cs_new:Npn \int_to_oct:n #1
3639 { \int_to_base:nn {#1} { 8 } }
```

(End definition for \int_to_bin:n and others. These functions are documented on page 70.)

\int_to_roman:n The __int_to_roman:w primitive creates tokens of category code 12 (other). Usually, what is actually wanted is letters. The approach here is to convert the output of the primitive into letters using appropriate control sequence names. That keeps everything expandable. The loop will be terminated by the conversion of the Q.

```

\__int_to_roman:N 3640 \cs_new:Npn \__int_to_roman:n #1
\__int_to_roman_v:w 3641 {
\__int_to_roman_x:w 3642   \exp_after:wN \__int_to_roman:N
\__int_to_roman_l:w 3643   \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_roman_c:w 3644 }
\__int_to_roman_d:w 3645 \cs_new:Npn \__int_to_roman:N #1
\__int_to_roman_m:w 3646 {
\__int_to_roman_Q:w 3647   \use:c { __int_to_roman_ #1 :w }
\__int_to_Roman_i:w 3648   \__int_to_roman:N
\__int_to_Roman_v:w 3649 }
\__int_to_Roman_x:w 3650 \cs_new:Npn \int_to_Roman:n #1
\__int_to_Roman_l:w 3651 {
\__int_to_Roman_c:w 3652   \exp_after:wN \__int_to_Roman_aux:N
\__int_to_Roman_d:w 3653   \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_Roman_m:w 3654 }
\__int_to_Roman_Q:w 3655 \cs_new:Npn \__int_to_Roman_aux:N #1
3656 {
3657   \use:c { __int_to_Roman_ #1 :w }
3658   \__int_to_Roman_aux:N
3659 }
3660 \cs_new_nopar:Npn \__int_to_roman_i:w { i }
3661 \cs_new_nopar:Npn \__int_to_roman_v:w { v }
3662 \cs_new_nopar:Npn \__int_to_roman_x:w { x }
3663 \cs_new_nopar:Npn \__int_to_roman_l:w { l }
3664 \cs_new_nopar:Npn \__int_to_roman_c:w { c }
3665 \cs_new_nopar:Npn \__int_to_roman_d:w { d }
3666 \cs_new_nopar:Npn \__int_to_roman_m:w { m }
3667 \cs_new_nopar:Npn \__int_to_roman_Q:w #1 { }
3668 \cs_new_nopar:Npn \__int_to_Roman_i:w { I }
3669 \cs_new_nopar:Npn \__int_to_Roman_v:w { V }
```

```

3670 \cs_new_nopar:Npn \__int_to_Roman_x:w { X }
3671 \cs_new_nopar:Npn \__int_to_Roman_l:w { L }
3672 \cs_new_nopar:Npn \__int_to_Roman_c:w { C }
3673 \cs_new_nopar:Npn \__int_to_Roman_d:w { D }
3674 \cs_new_nopar:Npn \__int_to_Roman_m:w { M }
3675 \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 70.)

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.

```

3676 \cs_new:Npn \__int_get_sign:n #1
3677 {
3678   \__int_get_sign_and_digits:nNNN {#1}
3679   \c_true_bool \c_true_bool \c_false_bool
3680 }
3681 \cs_new:Npn \__int_get_digits:n #1
3682 {
3683   \__int_get_sign_and_digits:nNNN {#1}
3684   \c_true_bool \c_false_bool \c_true_bool
3685 }

```

The auxiliary loops through, finding sign tokens and removing them. The sign itself is carried through as a flag.

```

3686 \cs_new:Npn \__int_get_sign_and_digits:nNNN #1#2#3#4
3687 {
3688   \exp_args:Nf \tl_if_head_eq_charcode:nNTF {#1} -
3689   {
3690     \bool_if:NTF #2
3691     {
3692       \__int_get_sign_and_digits:oNNN
3693       { \use_none:n #1 } \c_false_bool #3#4
3694     }
3695     {
3696       \__int_get_sign_and_digits:oNNN
3697       { \use_none:n #1 } \c_true_bool #3#4
3698     }
3699   }
3700   {
3701     \exp_args:Nf \tl_if_head_eq_charcode:nNTF {#1} +
3702     { \__int_get_sign_and_digits:oNNN { \use_none:n #1 } #2#3#4 }
3703     {
3704       \bool_if:NT #3 { \bool_if:NF #2 - }
3705       \bool_if:NT #4 {#1}
3706     }

```

```

3707     }
3708   }
3709   \cs_generate_variant:Nn \__int_get_sign_and_digits:nNNN { o }
(End definition for \__int_get_sign:n.)

```

\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.

```

\__int_from_alph:n
\__int_from_alph:nN
\__int_from_alph:N
3710 \cs_new:Npn \int_from_alph:n #1
3711 {
3712   \int_eval:n
3713   {
3714     \__int_get_sign:n {#1}
3715     \exp_args:Nf \__int_from_alph:n { \__int_get_digits:n {#1} }
3716   }
3717 }
3718 \cs_new:Npn \__int_from_alph:n #1
3719 { \__int_from_alph:nN { 0 } #1 \q_nil }
3720 \cs_new:Npn \__int_from_alph:nN #1#2
3721 {
3722   \quark_if_nil:NTF #2
3723   {#1}
3724   {
3725     \exp_args:Nf \__int_from_alph:nN
3726     { \int_eval:n { #1 * 26 + \__int_from_alph:N #2 } }
3727   }
3728 }
3729 \cs_new:Npn \__int_from_alph:N #1
3730 { \int_eval:n { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } } }
(End definition for \int_from_alph:n. This function is documented on page 70.)

```

\int_from_base:nn 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.

```

\__int_from_base:nn
\__int_from_base:nnN
\__int_from_base:N
3731 \cs_new:Npn \int_from_base:nn #1#2
3732 {
3733   \int_eval:n
3734   {
3735     \__int_get_sign:n {#1}
3736     \exp_args:Nf \__int_from_base:nn
3737     { \__int_get_digits:n {#1} } {#2}
3738   }
3739 }
3740 \cs_new:Npn \__int_from_base:nn #1#2
3741 { \__int_from_base:nnN { 0 } { #2 } #1 \q_nil }
3742 \cs_new:Npn \__int_from_base:nnN #1#2#3
3743 {
3744   \quark_if_nil:NTF #3
3745   {#1}
3746   {

```

```

3747 \exp_args:Nf \__int_from_base:nnN
3748 { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
3749 {#2}
3750 }
3751 }

```

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.

```

3752 \cs_new:Npn \__int_from_base:N #1
3753 {
3754   \int_compare:nNnTF { '#1 } < { 58 }
3755   {#1}
3756   {
3757     \int_eval:n
3758     { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
3759   }
3760 }

```

(End definition for `\int_from_base:nn`. This function is documented on page 71.)

`\int_from_bin:n` Wrappers around the generic function.

```

\int_from_hex:n 3761 \cs_new:Npn \int_from_bin:n #1
\int_from_oct:n 3762 { \int_from_base:nn {#1} \c_two }
3763 \cs_new:Npn \int_from_hex:n #1
3764 { \int_from_base:nn {#1} \c_sixteen }
3765 \cs_new:Npn \int_from_oct:n #1
3766 { \int_from_base:nn {#1} \c_eight }

```

(End definition for `\int_from_bin:n`, `\int_from_hex:n`, and `\int_from_oct:n`. These functions are documented on page 71.)

`\c__int_from_roman_i_int` Constants used to convert from Roman numerals to integers.

```

\c__int_from_roman_v_int 3767 \int_const:cn { c__int_from_roman_i_int } { 1 }
\c__int_from_roman_x_int 3768 \int_const:cn { c__int_from_roman_v_int } { 5 }
\c__int_from_roman_l_int 3769 \int_const:cn { c__int_from_roman_x_int } { 10 }
\c__int_from_roman_c_int 3770 \int_const:cn { c__int_from_roman_l_int } { 50 }
\c__int_from_roman_d_int 3771 \int_const:cn { c__int_from_roman_c_int } { 100 }
\c__int_from_roman_m_int 3772 \int_const:cn { c__int_from_roman_d_int } { 500 }
\c__int_from_roman_I_int 3773 \int_const:cn { c__int_from_roman_m_int } { 1000 }
\c__int_from_roman_V_int 3774 \int_const:cn { c__int_from_roman_I_int } { 1 }
\c__int_from_roman_X_int 3775 \int_const:cn { c__int_from_roman_V_int } { 5 }
\c__int_from_roman_L_int 3776 \int_const:cn { c__int_from_roman_X_int } { 10 }
\c__int_from_roman_C_int 3777 \int_const:cn { c__int_from_roman_L_int } { 50 }
\c__int_from_roman_D_int 3778 \int_const:cn { c__int_from_roman_C_int } { 100 }
\c__int_from_roman_M_int 3779 \int_const:cn { c__int_from_roman_D_int } { 500 }
3780 \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

```



```

3781 \cs_new:Npn \int_from_roman:n #1
3782 {
3783   \tl_if_blank:nF {#1}
3784   {
3785     \exp_after:wN \__int_from_roman_end:w
3786     \__int_value:w \__int_eval:w
3787     \__int_from_roman:NN #1 Q \q_stop
3788   }
3789 }
3790 \cs_new:Npn \__int_from_roman:NN #1#2
3791 {
3792   \str_if_eq:nnTF {#1} { Q }
3793   {#1#2}
3794   {
3795     \str_if_eq:nnTF {#2} { Q }
3796     {
3797       \int_if_exist:cF { c__int_from_roman_ #1 _int }
3798       { \__int_from_roman_clean_up:w }
3799       +
3800       \use:c { c__int_from_roman_ #1 _int }
3801       #2
3802     }
3803     {
3804       \int_if_exist:cF { c__int_from_roman_ #1 _int }
3805       { \__int_from_roman_clean_up:w }
3806       \int_if_exist:cF { c__int_from_roman_ #2 _int }
3807       { \__int_from_roman_clean_up:w }
3808       \int_compare:nNnTF
3809       { \use:c { c__int_from_roman_ #1 _int } }
3810       <
3811       { \use:c { c__int_from_roman_ #2 _int } }
3812       {
3813         + \use:c { c__int_from_roman_ #2 _int }
3814         - \use:c { c__int_from_roman_ #1 _int }
3815         \__int_from_roman:NN
3816       }
3817       {
3818         + \use:c { c__int_from_roman_ #1 _int }
3819         \__int_from_roman:NN #2
3820       }
3821     }
3822   }
3823 }
3824 \cs_new:Npn \__int_from_roman_end:w #1 Q #2 \q_stop
3825 { \tl_if_empty:nTF {#2} {#1} {#2} }
3826 \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 71.)

8.10 Viewing integer

```
\int_show:N
\int_show:c 3827 \cs_new_eq:NN \int_show:N \__kernel_register_show:N
3828 \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.

```
3829 \cs_new_protected:Npn \int_show:n #1
3830 { \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 71.)
```

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 72.)

\c_zero Again, one in `l3basics` for obvious reasons.
(End definition for `\c_zero`. This variable is documented on page 72.)

\c_six Once again, in `l3basics`.
\c_seven (End definition for `\c_six` and `\c_seven`. These variables are documented on page 72.)

\c_twelve Low-number values not previously defined.
\c_one
\c_sixteen
\c_two

```
3831 \int_const:Nn \c_one { 1 }
3832 \int_const:Nn \c_two { 2 }
3833 \int_const:Nn \c_three { 3 }
3834 \int_const:Nn \c_four { 4 }
3835 \int_const:Nn \c_five { 5 }
3836 \int_const:Nn \c_eight { 8 }
3837 \int_const:Nn \c_nine { 9 }
3838 \int_const:Nn \c_ten { 10 }
3839 \int_const:Nn \c_eleven { 11 }
3840 \int_const:Nn \c_thirteen { 13 }
3841 \int_const:Nn \c_fourteen { 14 }
3842 \int_const:Nn \c_fifteen { 15 }
(End definition for \c_one and others. These variables are documented on page 72.)
```

\c_thirty_two One middling value.

```
3843 \int_const:Nn \c_thirty_two { 32 }
(End definition for \c_thirty_two. This variable is documented on page 72.)
```

\c_two_hundred_fifty_five Two classic mid-range integer constants.

```
\c_two_hundred_fifty_six 3844 \int_const:Nn \c_two_hundred_fifty_five { 255 }
3845 \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 72.)
```

`\c_one_hundred` Simple runs of powers of ten.
`\c_one_thousand` 3846 `\int_const:Nn \c_one_hundred { 100 }`
`\c_ten_thousand` 3847 `\int_const:Nn \c_one_thousand { 1000 }`
3848 `\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 72.)

`\c_max_int` The largest number allowed is $2^{31} - 1$
3849 `\int_const:Nn \c_max_int { 2 147 483 647 }`
(End definition for `\c_max_int`. This variable is documented on page 72.)

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` 3850 `\int_new:N \l_tmpa_int`
`\g_tmpa_int` 3851 `\int_new:N \l_tmpb_int`
`\g_tmpb_int` 3852 `\int_new:N \g_tmpa_int`
3853 `\int_new:N \g_tmpb_int`
(End definition for `\l_tmpa_int` and `\l_tmpb_int`. These variables are documented on page 72.)

8.13 Deprecated functions

`\int_case:nnn` Deprecated 2013-07-15.
3854 `\cs_new_eq:NN \int_case:nnn \int_case:nnF`
(End definition for `\int_case:nnn`. This function is documented on page ??.)

`\int_to_binary:n` Deprecated 2014-02-11.
`\int_from_binary:n` 3855 `\cs_new_eq:NN \int_to_binary:n \int_to_bin:n`
`\int_to_hexadecimal:n` 3856 `\cs_new_eq:NN \int_to_hexadecimal:n \int_to_Hex:n`
`\int_from_hexadecimal:n` 3857 `\cs_new_eq:NN \int_to_octal:n \int_to_oct:n`
`\int_to_octal:n` 3858 `\cs_new_eq:NN \int_from_binary:n \int_from_bin:n`
`\int_from_octal:n` 3859 `\cs_new_eq:NN \int_from_hexadecimal:n \int_from_hex:n`
3860 `\cs_new_eq:NN \int_from_octal:n \int_from_oct:n`
(End definition for `\int_to_binary:n` and `\int_from_binary:n`. These functions are documented on page ??.)
3861 `\</initex | package>`

9 l3skip implementation

3862 `\<*initex | package>`
3863 `\<@@=dim>`

9.1 Length primitives renamed

`\if_dim:w` Primitives renamed.
`__dim_eval:w` 3864 `\cs_new_eq:NN \if_dim:w \tex_ifdim:D`
`__dim_eval_end:` 3865 `\cs_new_eq:NN __dim_eval:w \etex_dimexpr:D`
3866 `\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 3867 <*package>
3868 \cs_new_protected:Npn \dim_new:N #1
3869 {
3870   \__chk_if_free_cs:N #1
3871   \cs:w newdimen \cs_end: #1
3872 }
3873 </package>
3874 \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 3875 \cs_new_protected:Npn \dim_const:Nn #1
3876 {
3877   \dim_new:N #1
3878   \dim_gset:Nn #1
3879 }
3880 \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 3881 \cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_dim }
\dim_gzero:N 3882 \cs_new_protected:Npn \dim_gzero:N { \tex_global:D \dim_zero:N }
\dim_gzero:c 3883 \cs_generate_variant:Nn \dim_zero:N { c }
3884 \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 3885 \cs_new_protected:Npn \dim_zero_new:N #1
\dim_gzero_new:N 3886 { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
\dim_gzero_new:c 3887 \cs_new_protected:Npn \dim_gzero_new:N #1
3888 { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
3889 \cs_generate_variant:Nn \dim_zero_new:N { c }
3890 \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 3891 \prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N { TF , T , F , p }
\dim_if_exist:NTF 3892 \prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c { TF , T , F , p }
\dim_if_exist:cTF (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      3893 \cs_new_protected:Npn \dim_set:Nn #1#2
\dim_set:cn      3894 { #1 ~ \__dim_eval:w #2 \__dim_eval_end: }
\dim_gset:Nn     3895 \cs_new_protected:Npn \dim_gset:Nn { \tex_global:D \dim_set:Nn }
\dim_gset:cn     3896 \cs_generate_variant:Nn \dim_set:Nn { c }
\dim_gset:cn     3897 \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   3898 \cs_new_protected:Npn \dim_set_eq:NN #1#2 { #1 = #2 }
\dim_set_eq:Nc   3899 \cs_generate_variant:Nn \dim_set_eq:NN { c }
\dim_set_eq:cc   3900 \cs_generate_variant:Nn \dim_set_eq:NN { Nc , cc }
\dim_gset_eq:NN  3901 \cs_new_protected:Npn \dim_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\dim_gset_eq:cN  3902 \cs_generate_variant:Nn \dim_gset_eq:NN { c }
\dim_gset_eq:Nc  3903 \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      3904 \cs_new_protected:Npn \dim_add:Nn #1#2
\dim_gadd:Nn     3905 { \tex_advance:D #1 by \__dim_eval:w #2 \__dim_eval_end: }
\dim_gadd:cn     3906 \cs_new_protected:Npn \dim_gadd:Nn { \tex_global:D \dim_add:Nn }
\dim_sub:Nn      3907 \cs_generate_variant:Nn \dim_add:Nn { c }
\dim_sub:cn      3908 \cs_generate_variant:Nn \dim_gadd:Nn { c }
\dim_gsub:Nn     3909 \cs_new_protected:Npn \dim_sub:Nn #1#2
\dim_gsub:cn     3910 { \tex_advance:D #1 by - \__dim_eval:w #2 \__dim_eval_end: }
\dim_gsub:cn     3911 \cs_new_protected:Npn \dim_gsub:Nn { \tex_global:D \dim_sub:Nn }
\dim_gsub:cn     3912 \cs_generate_variant:Nn \dim_sub:Nn { c }
\dim_gsub:cn     3913 \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     3914 \cs_new:Npn \dim_abs:n #1
\dim_max:nn      3915 {
\dim_min:nn      3916   \exp_after:wN \__dim_abs:N
\__dim_maxmin:wwN 3917   \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
\__dim_maxmin:wwN 3918 }
\dim_max:nn      3919 \cs_new:Npn \__dim_abs:N #1
\dim_max:nn      3920 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
\dim_max:nn      3921 \cs_set:Npn \dim_max:nn #1#2
\dim_max:nn      3922 {
\dim_max:nn      3923   \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
\dim_max:nn      3924   \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
\dim_max:nn      3925   \dim_use:N \__dim_eval:w #2 ;
```

```

3926     >
3927     \__dim_eval_end:
3928   }
3929   \cs_set:Npn \dim_min:nn #1#2
3930   {
3931     \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
3932     \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
3933     \dim_use:N \__dim_eval:w #2 ;
3934     <
3935     \__dim_eval_end:
3936   }
3937   \cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
3938   {
3939     \if_dim:w #1 #3 #2 ~
3940       #1
3941     \else:
3942       #2
3943     \fi:
3944   }

```

(End definition for `\dim_abs:n`. This function is documented on page 76.)

`\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.

```

3945   \cs_new:Npn \dim_ratio:nn #1#2
3946   { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
3947   \cs_new:Npn \__dim_ratio:n #1
3948   { \__int_value:w \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_ratio:nn`. This function is documented on page 77.)

9.5 Dimension expression conditionals

`\dim_compare_p:nNn` Simple comparison.

```

\dim_compare:nNnTF
3949   \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
3950   {
3951     \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
3952     \prg_return_true: \else: \prg_return_false: \fi:
3953   }

```

(End definition for `\dim_compare:nNn`. These functions are documented on page 77.)

`\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
3954   \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }

```

```

3955 {
3956   \exp_after:wN \__dim_compare:w
3957   \dim_use:N \__dim_eval:w #1 \__prg_compare_error:
3958 }
3959 \cs_new:Npn \__dim_compare:w #1 \__prg_compare_error:
3960 {
3961   \exp_after:wN \if_false: \tex_romannumeral:D -'0
3962   \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \q_stop
3963 }
3964 \exp_args:Nno \use:nn
3965 { \cs_new:Npn \__dim_compare:wNN #1 }
3966 { \tl_to_str:n {pt} }
3967 #2#3
3968 {
3969   \if_meaning:w = #3
3970   \use:c { __dim_compare_#2:w }
3971   \fi:
3972   #1 pt \exp_stop_f:
3973   \prg_return_false:
3974   \exp_after:wN \use_none_delimit_by_q_stop:w
3975   \fi:
3976   \reverse_if:N \if_dim:w #1 pt #2
3977   \exp_after:wN \__dim_compare:wNN
3978   \dim_use:N \__dim_eval:w #3
3979 }
3980 \cs_new:cpn { __dim_compare_ ! :w }
3981 #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
3982 \cs_new:cpn { __dim_compare_ = :w }
3983 #1 \__dim_eval:w = { #1 \__dim_eval:w }
3984 \cs_new:cpn { __dim_compare_ < :w }
3985 #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
3986 \cs_new:cpn { __dim_compare_ > :w }
3987 #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
3988 \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \q_stop
3989 { #1 \prg_return_false: \else: \prg_return_true: \fi: }

```

(End definition for \dim_compare:n. These functions are documented on page 78.)

\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 3990 \cs_new:Npn \dim_case:nnTF #1
\__dim_case:nw 3991 {
\__dim_case_end:nw 3992   \tex_romannumeral:D
3993   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} }
3994 }
3995 \cs_new:Npn \dim_case:nnT #1#2#3
3996 {
3997   \tex_romannumeral:D
3998   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
3999 }

```

```

4000 \cs_new:Npn \dim_case:nnF #1#2
4001 {
4002   \tex_romannumeral:D
4003   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
4004 }
4005 \cs_new:Npn \dim_case:nn #1#2
4006 {
4007   \tex_romannumeral:D
4008   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
4009 }
4010 \cs_new:Npn \__dim_case:nnTF #1#2#3#4
4011 { \__dim_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
4012 \cs_new:Npn \__dim_case:nw #1#2#3
4013 {
4014   \dim_compare:nNnTF {#1} = {#2}
4015   { \__dim_case_end:nw {#3} }
4016   { \__dim_case:nw {#1} }
4017 }
4018 \cs_new_eq:NN \__dim_case_end:nw \__prg_case_end:nw

```

(End definition for `\dim_case:nn`. This function is documented on page 79.)

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
4019 \cs_set:Npn \dim_while_do:nn #1#2
4020 {
4021   \dim_compare:nT {#1}
4022   {
4023     #2
4024     \dim_while_do:nn {#1} {#2}
4025   }
4026 }
4027 \cs_set:Npn \dim_until_do:nn #1#2
4028 {
4029   \dim_compare:nF {#1}
4030   {
4031     #2
4032     \dim_until_do:nn {#1} {#2}
4033   }
4034 }
4035 \cs_set:Npn \dim_do_while:nn #1#2
4036 {
4037   #2
4038   \dim_compare:nT {#1}
4039   { \dim_do_while:nn {#1} {#2} }
4040 }
4041 \cs_set:Npn \dim_do_until:nn #1#2
4042 {

```



```

4043     #2
4044     \dim_compare:nF {#1}
4045     { \dim_do_until:nn {#1} {#2} }
4046 }

```

(End definition for `\dim_while_do:nn`. This function is documented on page 80.)

`\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.

```

4047 \cs_set:Npn \dim_while_do:nNnn #1#2#3#4
4048 {
4049   \dim_compare:nNnT {#1} #2 {#3}
4050   {
4051     #4
4052     \dim_while_do:nNnn {#1} #2 {#3} {#4}
4053   }
4054 }
4055 \cs_set:Npn \dim_until_do:nNnn #1#2#3#4
4056 {
4057   \dim_compare:nNnF {#1} #2 {#3}
4058   {
4059     #4
4060     \dim_until_do:nNnn {#1} #2 {#3} {#4}
4061   }
4062 }
4063 \cs_set:Npn \dim_do_while:nNnn #1#2#3#4
4064 {
4065   #4
4066   \dim_compare:nNnT {#1} #2 {#3}
4067   { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
4068 }
4069 \cs_set:Npn \dim_do_until:nNnn #1#2#3#4
4070 {
4071   #4
4072   \dim_compare:nNnF {#1} #2 {#3}
4073   { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
4074 }

```

(End definition for `\dim_while_do:nNnn`. This function is documented on page 79.)

9.7 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

4075 \cs_new:Npn \dim_eval:n #1
4076 { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_eval:n`. This function is documented on page 80.)

`\dim_use:N` Accessing a $\langle dim \rangle$.

```

\dim_use:c
4077 \cs_new_eq:NN \dim_use:N \tex_the:D
4078 \cs_generate_variant:Nn \dim_use:N { c }

```

(End definition for `\dim_use:N` and `\dim_use:c`. These functions are documented on page ??.)

`\dim_to_decimal:n` A function which comes up often enough to deserve a place in the kernel. Evaluate the dimension expression `#1` then remove the trailing `pt`. The argument is put in parentheses as this prevents the dimension expression from terminating early and leaving extra tokens lying around. This is used a lot by low-level manipulations.

`__dim_to_decimal:w`

```

4079 \cs_new:Npn \dim_to_decimal:n #1
4080 {
4081   \exp_after:wN
4082   \__dim_to_decimal:w \dim_use:N \__dim_eval:w (#1) \__dim_eval_end:
4083 }
4084 \use:x
4085 {
4086   \cs_new:Npn \exp_not:N \__dim_to_decimal:w
4087   ##1 . ##2 \tl_to_str:n { pt }
4088 }
4089 {
4090   \int_compare:nNnTF {#2} > \c_zero
4091   { #1 . #2 }
4092   { #1 }
4093 }

```

(End definition for `\dim_to_decimal:n`. This function is documented on page 81.)

`\dim_to_decimal_in_bp:n` Conversion to big points is done using a scaling inside `__dim_eval:w` as ε -TeX does that using 64-bit precision. Here, 800/803 is the integer fraction for 72/72.27. This is a common case so is hand-coded for accuracy (and speed).

```

4094 \cs_new:Npn \dim_to_decimal_in_bp:n #1
4095 { \dim_to_decimal:n { ( #1 ) * 800 / 803 } }

```

(End definition for `\dim_to_decimal_in_bp:n`. This function is documented on page 81.)

`\dim_to_decimal_in_unit:nn` An analogue of `\dim_ratio:nn` that produces a decimal number as its result, rather than a rational fraction for use within dimension expressions.

```

4096 \cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
4097 {
4098   \dim_to_decimal:n
4099   {
4100     1pt *
4101     \dim_ratio:nn {#1} {#2}
4102   }
4103 }

```

(End definition for `\dim_to_decimal_in_unit:nn`. This function is documented on page 81.)

`\dim_to_fp:n` Defined in `l3fp-convert`, documented here.

(End definition for `\dim_to_fp:n`. This function is documented on page 82.)

9.8 Viewing dim variables

`\dim_show:N` Diagnostics.

```
\dim_show:c 4104 \cs_new_eq:NN \dim_show:N \__kernel_register_show:N
4105 \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.

```
4106 \cs_new_protected:Npn \dim_show:n #1
4107 { \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 4108 \dim_const:Nn \c_zero_dim { 0 pt }
4109 \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_tmpb_dim 4110 \dim_new:N \l_tmpa_dim
\g_tmpa_dim 4111 \dim_new:N \l_tmpb_dim
\g_tmpb_dim 4112 \dim_new:N \g_tmpa_dim
4113 \dim_new:N \g_tmpb_dim
(End definition for \l_tmpa_dim and \l_tmpb_dim. These variables are documented on page 82.)
```

9.11 Creating and initialising skip variables

`\skip_new:N` Allocation of a new internal registers.

```
\skip_new:c 4114 \<package>
4115 \cs_new_protected:Npn \skip_new:N #1
4116 {
4117   \__chk_if_free_cs:N #1
4118   \cs:w newskip \cs_end: #1
4119 }
4120 \</package>
4121 \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
4122 \cs_new_protected:Npn \skip_const:Nn #1
4123 {
4124   \skip_new:N #1
4125   \skip_gset:Nn #1
4126 }
4127 \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
4128 \cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }
\skip_gzero:N
4129 \cs_new_protected:Npn \skip_gzero:N { \tex_global:D \skip_zero:N }
\skip_gzero:c
4130 \cs_generate_variant:Nn \skip_zero:N { c }
4131 \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
4132 \cs_new_protected:Npn \skip_zero_new:N #1
\skip_gzero_new:N
4133 { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\skip_gzero_new:c
4134 \cs_new_protected:Npn \skip_gzero_new:N #1
4135 { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
4136 \cs_generate_variant:Nn \skip_zero_new:N { c }
4137 \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
4138 \prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N { TF , T , F , p }
\skip_if_exist:NTF
4139 \prg_new_eq_conditional:NNn \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
4140 \cs_new_protected:Npn \skip_set:Nn #1#2
\skip_gset:Nn
4141 { #1 ~ \etex_glueexpr:D #2 \scan_stop: }
\skip_gset:cn
4142 \cs_new_protected:Npn \skip_gset:Nn { \tex_global:D \skip_set:Nn }
4143 \cs_generate_variant:Nn \skip_set:Nn { c }
4144 \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
4145 \cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
\skip_set_eq:Nc
4146 \cs_generate_variant:Nn \skip_set_eq:NN { c }
\skip_set_eq:cc
4147 \cs_generate_variant:Nn \skip_set_eq:NN { Nc , cc }
\skip_gset_eq:NN
4148 \cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\skip_gset_eq:cN
4149 \cs_generate_variant:Nn \skip_gset_eq:NN { c }
\skip_gset_eq:Nc
4150 \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 4151 \cs_new_protected:Npn \skip_add:Nn #1#2
\skip_gadd:Nn 4152 { \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }
\skip_gadd:cn 4153 \cs_new_protected:Npn \skip_gadd:Nn { \tex_global:D \skip_add:Nn }
\skip_sub:Nn 4154 \cs_generate_variant:Nn \skip_add:Nn { c }
\skip_sub:cn 4155 \cs_generate_variant:Nn \skip_gadd:Nn { c }
\skip_gsub:Nn 4156 \cs_new_protected:Npn \skip_sub:Nn #1#2
\skip_gsub:cn 4157 { \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }
4158 \cs_new_protected:Npn \skip_gsub:Nn { \tex_global:D \skip_sub:Nn }
4159 \cs_generate_variant:Nn \skip_sub:Nn { c }
4160 \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.

```

4161 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
4162 {
4163   \if_int_compare:w
4164     \__str_if_eq_x:nn { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
4165     = \c_zero
4166     \prg_return_true:
4167   \else:
4168     \prg_return_false:
4169   \fi:
4170 }

```

(End definition for `\skip_if_eq:nn`. These functions are documented on page 84.)

`\skip_if_finite_p:n` With ε -TeX, we have an easy access to the order of infinities of the stretch and shrink
`\skip_if_finite:nTF` components of a skip. However, to access both, we either need to evaluate the expression
`__skip_if_finite:wwNw` 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.

```

4171 \cs_set_protected:Npn \__cs_tmp:w #1
4172 {
4173   \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
4174   {
4175     \exp_after:wN \__skip_if_finite:wwNw
4176     \skip_use:N \etex_glueexpr:D ##1 ; \prg_return_false:
4177     #1 ; \prg_return_true: \q_stop
4178   }
4179   \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \q_stop {##3}
4180 }
4181 \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.

```
4182 \cs_new:Npn \skip_eval:n #1
4183 { \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 4184 \cs_new_eq:NN \skip_use:N \tex_the:D
4185 \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 4186 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
\skip_horizontal:n 4187 \cs_new:Npn \skip_horizontal:n #1
\skip_vertical:N 4188 { \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop: }
\skip_vertical:c 4189 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
\skip_vertical:n 4190 \cs_new:Npn \skip_vertical:n #1
4191 { \skip_vertical:N \etex_glueexpr:D #1 \scan_stop: }
4192 \cs_generate_variant:Nn \skip_horizontal:N { c }
4193 \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 4194 \cs_new_eq:NN \skip_show:N \__kernel_register_show:N
4195 \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.

```
4196 \cs_new_protected:Npn \skip_show:n #1
4197 { \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 4198 \skip_const:Nn \c_zero_skip { \c_zero_dim }
4199 \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` 4200 `\skip_new:N \l_tmpa_skip`
`\g_tmpa_skip` 4201 `\skip_new:N \l_tmpb_skip`
`\g_tmpb_skip` 4202 `\skip_new:N \g_tmpa_skip`
4203 `\skip_new:N \g_tmpb_skip`

(End definition for `\l_tmpa_skip` and `\l_tmpb_skip`. These variables are documented on page 85.)

9.19 Creating and initialising muskip variables

`\muskip_new:N` And then we add muskips.

`\muskip_new:c` 4204 `*package`
4205 `\cs_new_protected:Npn \muskip_new:N #1`
4206 `{`
4207 `__chk_if_free_cs:N #1`
4208 `\cs:w newmuskip \cs_end: #1`
4209 `}`
4210 `*package`
4211 `\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` 4212 `\cs_new_protected:Npn \muskip_const:Nn #1`
4213 `{`
4214 `\muskip_new:N #1`
4215 `\muskip_gset:Nn #1`
4216 `}`
4217 `\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` 4218 `\cs_new_protected:Npn \muskip_zero:N #1`
`\muskip_gzero:N` 4219 `{ #1 \c_zero_muskip }`
`\muskip_gzero:c` 4220 `\cs_new_protected:Npn \muskip_gzero:N { \tex_global:D \muskip_zero:N }`
4221 `\cs_generate_variant:Nn \muskip_zero:N { c }`
4222 `\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` 4223 `\cs_new_protected:Npn \muskip_zero_new:N #1`
`\muskip_gzero_new:N` 4224 `{ \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }`
`\muskip_gzero_new:c` 4225 `\cs_new_protected:Npn \muskip_gzero_new:N #1`
4226 `{ \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }`
4227 `\cs_generate_variant:Nn \muskip_zero_new:N { c }`
4228 `\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` 4229 `\prg_new_eq_conditional:Nn \muskip_if_exist:N \cs_if_exist:N { TF , T , F , p }`
`\muskip_if_exist:N $\textcolor{red}{TF}$` 4230 `\prg_new_eq_conditional:Nn \muskip_if_exist:c \cs_if_exist:c { TF , T , F , p }`
`\muskip_if_exist:c $\textcolor{red}{TF}$` (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` 4231 `\cs_new_protected:Npn \muskip_set:Nn #1#2`
`\muskip_gset:Nn` 4232 `{ #1 ~ \etex_muexpr:D #2 \scan_stop: }`
`\muskip_gset:cn` 4233 `\cs_new_protected:Npn \muskip_gset:Nn { \tex_global:D \muskip_set:Nn }`
4234 `\cs_generate_variant:Nn \muskip_set:Nn { c }`
4235 `\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` 4236 `\cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }`
`\muskip_set_eq:Nc` 4237 `\cs_generate_variant:Nn \muskip_set_eq:NN { c }`
`\muskip_set_eq:cc` 4238 `\cs_generate_variant:Nn \muskip_set_eq:NN { Nc , cc }`
`\muskip_gset_eq:NN` 4239 `\cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }`
`\muskip_gset_eq:cN` 4240 `\cs_generate_variant:Nn \muskip_gset_eq:NN { c }`
`\muskip_gset_eq:Nc` 4241 `\cs_generate_variant:Nn \muskip_gset_eq:NN { Nc , cc }`
`\muskip_gset_eq:cc` (End definition for `\muskip_set_eq:NN` and others. These functions are documented on page ??.)

`\muskip_add:Nn` Using `by` here deals with the (incorrect) case `\muskip123`.
`\muskip_add:cn` 4242 `\cs_new_protected:Npn \muskip_add:Nn #1#2`
`\muskip_gadd:Nn` 4243 `{ \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }`
`\muskip_gadd:cn` 4244 `\cs_new_protected:Npn \muskip_gadd:Nn { \tex_global:D \muskip_add:Nn }`
`\muskip_sub:Nn` 4245 `\cs_generate_variant:Nn \muskip_add:Nn { c }`
`\muskip_sub:cn` 4246 `\cs_generate_variant:Nn \muskip_gadd:Nn { c }`
`\muskip_gsub:Nn` 4247 `\cs_new_protected:Npn \muskip_sub:Nn #1#2`
`\muskip_gsub:cn` 4248 `{ \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }`
4249 `\cs_new_protected:Npn \muskip_gsub:Nn { \tex_global:D \muskip_sub:Nn }`
4250 `\cs_generate_variant:Nn \muskip_sub:Nn { c }`
4251 `\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.
4252 `\cs_new:Npn \muskip_eval:n #1`
4253 `{ \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` 4254 `\cs_new_eq:NN \muskip_use:N \tex_the:D`
4255 `\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` 4256 `\cs_new_eq:NN \muskip_show:N __kernel_register_show:N`
4257 `\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.
4258 `\cs_new_protected:Npn \muskip_show:n #1`
4259 `{ \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` 4260 `\muskip_const:Nn \c_zero_muskip { 0 mu }`
4261 `\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

`\l_tmpa_muskip` We provide two local and two global scratch registers, maybe we need more or less.
`\l_tmpb_muskip` 4262 `\muskip_new:N \l_tmpa_muskip`
`\g_tmpa_muskip` 4263 `\muskip_new:N \l_tmpb_muskip`
`\g_tmpb_muskip` 4264 `\muskip_new:N \g_tmpa_muskip`
4265 `\muskip_new:N \g_tmpb_muskip`
(End definition for `\l_tmpa_muskip` and `\l_tmpb_muskip`. These variables are documented on page 88.)

9.25 Deprecated functions

`\dim_case:nnn` Deprecated 2013-07-15.
4266 `\cs_new_eq:NN \dim_case:nnn \dim_case:nnF`
(End definition for `\dim_case:nnn`. This function is documented on page ??.)
`__dim_strip_bp:n` Deprecated 2014-07-15.
`__dim_strip_pt:n` 4267 `\cs_new_eq:NN __dim_strip_bp:n \dim_to_decimal_in_bp:n`
4268 `\cs_new_eq:NN __dim_strip_pt:n \dim_to_decimal:n`
(End definition for `__dim_strip_bp:n` and `__dim_strip_pt:n`. These functions are documented on page ??.)
4269 `\</initex | package>`

10 l3tl implementation

```
4270 <*initex | package>
4271 <@@=tl>
```

A token list variable is a \TeX macro that holds tokens. By using the $\varepsilon\text{-TeX}$ primitive \unexpanded inside a \TeX \edef it is possible to store any tokens, including $\#$, in this way.

10.1 Functions

\tl_new:N Creating new token list variables is a case of checking for an existing definition and doing the definition.

```
\tl\_new:c
4272 \cs\_new\_protected:Npn \tl\_new:N #1
4273 {
4274   \__chk\_if\_free\_cs:N #1
4275   \cs\_gset\_eq:NN #1 \c\_empty\_tl
4276 }
4277 \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
\tl\_const:cn
\tl\_const:cx
4278 \cs\_new\_protected:Npn \tl\_const:Nn #1#2
4279 {
4280   \__chk\_if\_free\_cs:N #1
4281   \cs\_gset\_nopar:Npx #1 { \exp\_not:n {#2} }
4282 }
4283 \cs\_new\_protected:Npn \tl\_const:Nx #1#2
4284 {
4285   \__chk\_if\_free\_cs:N #1
4286   \cs\_gset\_nopar:Npx #1 {#2}
4287 }
4288 \cs\_generate\_variant:Nn \tl\_const:Nn { c }
4289 \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 sorted out by the parent function.

```
\tl\_clear:c
\tl\_gclear:N
\tl\_gclear:c
4290 \cs\_new\_protected:Npn \tl\_clear:N #1
4291 { \tl\_set\_eq:NN #1 \c\_empty\_tl }
4292 \cs\_new\_protected:Npn \tl\_gclear:N #1
4293 { \tl\_gset\_eq:NN #1 \c\_empty\_tl }
4294 \cs\_generate\_variant:Nn \tl\_clear:N { c }
4295 \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 sorted out by the parent function.

```
\tl\_clear\_new:c
\tl\_gclear\_new:N
\tl\_gclear\_new:c
4296 \cs\_new\_protected:Npn \tl\_clear\_new:N #1
```

```

4297 { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
4298 \cs_new_protected:Npn \tl_gclear_new:N #1
4299 { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
4300 \cs_generate_variant:Nn \tl_clear_new:N { c }
4301 \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 4302 \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN
\tl_set_eq:cN 4303 \cs_new_eq:NN \tl_set_eq:cN \cs_set_eq:cN
\tl_set_eq:cc 4304 \cs_new_eq:NN \tl_set_eq:Nc \cs_set_eq:Nc
\tl_gset_eq:NN 4305 \cs_new_eq:NN \tl_set_eq:cc \cs_set_eq:cc
\tl_gset_eq:Nc 4306 \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
\tl_gset_eq:cN 4307 \cs_new_eq:NN \tl_gset_eq:cN \cs_gset_eq:cN
\tl_gset_eq:Nc 4308 \cs_new_eq:NN \tl_gset_eq:Nc \cs_gset_eq:Nc
\tl_gset_eq:cc 4309 \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 4310 \cs_new_protected:Npn \tl_concat:NNN #1#2#3
\tl_gconcat:NNN 4311 { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
\tl_gconcat:ccc 4312 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
4313 { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
4314 \cs_generate_variant:Nn \tl_concat:NNN { ccc }
4315 \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 4316 \prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
\tl_if_exist:NTF 4317 \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`.

```

4318 \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. \LaTeX does not quote file names containing spaces, whereas $\text{pdf}\text{\TeX}$ and $\text{X}\text{\TeX}$ do. So there may be a correction to make in the \LaTeX case.

```

4319 <*initex>
4320 \luatex_if_engine:T
4321 {
4322   \tex_everyjob:D \exp_after:wN
4323   {

```

```

4324 \tex_the:D \tex_everyjob:D
4325 \lua_now_x:n
4326 { dofile ( assert ( kpse.find_file ("lualatexquotejobname.lua" ) ) ) }
4327 }
4328 }
4329 \tex_everyjob:D \exp_after:wN
4330 {
4331 \tex_the:D \tex_everyjob:D
4332 \tl_const:Nx \c_job_name_tl { \tex_jobname:D }
4333 }
4334 </initex>
4335 <*package>
4336 \tl_const:Nx \c_job_name_tl { \tex_jobname:D }
4337 </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).

```
4338 \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

`\tl_set:Nn` By using `\exp_not:n` token list variables can contain # tokens, which makes the token list registers provided by T_EX more or less redundant. The `\tl_set:No` version is done “by hand” as it is used quite a lot.

```

4339 \cs_new_protected:Npn \tl_set:Nn #1#2
4340 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
4341 \cs_new_protected:Npn \tl_set:No #1#2
4342 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
4343 \cs_new_protected:Npn \tl_set:Nx #1#2
4344 { \cs_set_nopar:Npx #1 {#2} }
4345 \cs_new_protected:Npn \tl_gset:Nn #1#2
4346 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
4347 \cs_new_protected:Npn \tl_gset:No #1#2
4348 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }
4349 \cs_new_protected:Npn \tl_gset:Nx #1#2
4350 { \cs_gset_nopar:Npx #1 {#2} }
4351 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
4352 \cs_generate_variant:Nn \tl_set:Nx { c }
4353 \cs_generate_variant:Nn \tl_set:Nn { c , co , cV , cv , cf }
4354 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
4355 \cs_generate_variant:Nn \tl_gset:Nx { c }
4356 \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 ??.)

`\tl_put_left:Nn` Adding to the left is done directly to gain a little performance.

```

4357 \cs_new_protected:Npn \tl_put_left:Nn #1#2
4358 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }

```

```

\tl_put_left:Nx
\tl_put_left:cn
\tl_put_left:cV
\tl_put_left:co
\tl_put_left:cx

```

```

\tl_gput_left:Nn
\tl_gput_left:NV
\tl_gput_left:No
\tl_gput_left:Nx
\tl_gput_left:cn

```

```

4359 \cs_new_protected:Npn \tl_put_left:NV #1#2
4360 { \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4361 \cs_new_protected:Npn \tl_put_left:No #1#2
4362 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4363 \cs_new_protected:Npn \tl_put_left:Nx #1#2
4364 { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
4365 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
4366 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
4367 \cs_new_protected:Npn \tl_gput_left:NV #1#2
4368 { \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4369 \cs_new_protected:Npn \tl_gput_left:No #1#2
4370 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4371 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
4372 { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
4373 \cs_generate_variant:Nn \tl_put_left:Nn { c }
4374 \cs_generate_variant:Nn \tl_put_left:NV { c }
4375 \cs_generate_variant:Nn \tl_put_left:No { c }
4376 \cs_generate_variant:Nn \tl_put_left:Nx { c }
4377 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
4378 \cs_generate_variant:Nn \tl_gput_left:NV { c }
4379 \cs_generate_variant:Nn \tl_gput_left:No { c }
4380 \cs_generate_variant:Nn \tl_gput_left:Nx { c }

```

(End definition for \tl_put_left:Nn and others. These functions are documented on page ??.)

\tl_put_right:Nn

The same on the right.

```

\tl_put_right:NV 4381 \cs_new_protected:Npn \tl_put_right:Nn #1#2
\tl_put_right:No 4382 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\tl_put_right:Nx 4383 \cs_new_protected:Npn \tl_put_right:NV #1#2
\tl_put_right:cn 4384 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\tl_put_right:cV 4385 \cs_new_protected:Npn \tl_put_right:No #1#2
\tl_put_right:co 4386 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\tl_put_right:cx 4387 \cs_new_protected:Npn \tl_put_right:Nx #1#2
\tl_gput_right:Nn 4388 { \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 } }
\tl_gput_right:NV 4389 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
\tl_gput_right:No 4390 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\tl_gput_right:Nx 4391 \cs_new_protected:Npn \tl_gput_right:NV #1#2
\tl_gput_right:cn 4392 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\tl_gput_right:cV 4393 \cs_new_protected:Npn \tl_gput_right:No #1#2
\tl_gput_right:co 4394 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\tl_gput_right:cx 4395 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
4396 { \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
4397 \cs_generate_variant:Nn \tl_put_right:Nn { c }
4398 \cs_generate_variant:Nn \tl_put_right:NV { c }
4399 \cs_generate_variant:Nn \tl_put_right:No { c }
4400 \cs_generate_variant:Nn \tl_put_right:Nx { c }
4401 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
4402 \cs_generate_variant:Nn \tl_gput_right:NV { c }
4403 \cs_generate_variant:Nn \tl_gput_right:No { c }
4404 \cs_generate_variant:Nn \tl_gput_right:Nx { c }

```

(End definition for `\tl_put_right:Nn` and others. These functions are documented on page ??.)

When used as a package, there is an option to be picky and to check definitions exist. This part of the process is done now, so that variable types based on `tl` (for example `clist`, `seq` and `prop`) will inherit the appropriate definitions. No `\tl_map_...` yet as the mechanisms are not fully in place. Thus instead do a more low level set up for a mapping, as in `l3basics`.

```

4405 <*package>
4406 \tex_ifodd:D \l@expl@check@declarations@bool
4407 \cs_set_protected:Npn \__cs_tmp:w #1
4408 {
4409   \if_meaning:w ? #1
4410   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
4411   \fi:
4412   \use:x
4413   {
4414     \cs_set_protected:Npn #1 \exp_not:n { ##1 ##2 }
4415     {
4416       \__chk_if_exist_var:N \exp_not:n {##1}
4417       \exp_not:o { #1 {##1} {##2} }
4418     }
4419   }
4420   \__cs_tmp:w
4421 }
4422 \__cs_tmp:w
4423 \tl_set:Nn \tl_set:No \tl_set:Nx
4424 \tl_gset:Nn \tl_gset:No \tl_gset:Nx
4425 \tl_put_left:Nn \tl_put_left:NV
4426 \tl_put_left:No \tl_put_left:Nx
4427 \tl_gput_left:Nn \tl_gput_left:NV
4428 \tl_gput_left:No \tl_gput_left:Nx
4429 \tl_put_right:Nn \tl_put_right:NV
4430 \tl_put_right:No \tl_put_right:Nx
4431 \tl_gput_right:Nn \tl_gput_right:NV
4432 \tl_gput_right:No \tl_gput_right:Nx
4433 ? \q_recursion_stop
4434 </package>

```

The two `set_eq` functions are done by hand as the internals there are a bit different.

```

4435 <*package>
4436 \cs_set_protected:Npn \tl_set_eq:NN #1#2
4437 {
4438   \__chk_if_exist_var:N #1
4439   \__chk_if_exist_var:N #2
4440   \cs_set_eq:NN #1 #2
4441 }
4442 \cs_set_protected:Npn \tl_gset_eq:NN #1#2
4443 {
4444   \__chk_if_exist_var:N #1
4445   \__chk_if_exist_var:N #2

```

```

4446     \cs_gset_eq:NN #1 #2
4447   }
4448 </package>

```

There is also a need to check all three arguments of the `concat` functions: a token list #2 or #3 equal to `\scan_stop:` would lead to problems later on.

```

4449 <*package>
4450   \cs_set_protected:Npn \tl_concat:NNN #1#2#3
4451   {
4452     \__chk_if_exist_var:N #1
4453     \__chk_if_exist_var:N #2
4454     \__chk_if_exist_var:N #3
4455     \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} }
4456   }
4457   \cs_set_protected:Npn \tl_gconcat:NNN #1#2#3
4458   {
4459     \__chk_if_exist_var:N #1
4460     \__chk_if_exist_var:N #2
4461     \__chk_if_exist_var:N #3
4462     \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} }
4463   }
4464   \tex_fi:D
4465 </package>

```

10.4 Reassigning token list category codes

`\c__tl_rescan_marker_tl` The rescanning code needs a special token list containing the same character with two different category codes. This is set up here, while the detail is described below. Note that we are sure that the colon has category letter at this stage.

```

4466 \tl_const:Nx \c__tl_rescan_marker_tl { : \token_to_str:N : }

```

(End definition for `\c__tl_rescan_marker_tl`. This variable is documented on page ??.)

```

\tl_set_rescan:Nnn
\tl_set_rescan:Nno
\tl_set_rescan:Nnx
\tl_set_rescan:cnn
\tl_set_rescan:cno
\tl_set_rescan:cnx
\tl_gset_rescan:Nnn
\tl_gset_rescan:Nno
\tl_gset_rescan:Nnx
\tl_gset_rescan:cnn
\tl_gset_rescan:cno
\tl_gset_rescan:cnx
\tl_rescan:nn
__tl_set_rescan:NNnn
__tl_rescan:w

```

The idea here is to deal cleanly with the problem that `\scantokens` treats the argument as a file, and without the correct settings a T_EX error occurs:

```
! File ended while scanning definition of ...
```

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”. To be safe, the `<setup>` #3 is followed by `\scan_stop:.`

```

4467 \cs_new_protected_nopar:Npn \tl_set_rescan:Nnn
4468 { \__tl_set_rescan:NNnn \tl_set:Nn }
4469 \cs_new_protected_nopar:Npn \tl_gset_rescan:NNnn
4470 { \__tl_set_rescan:NNnn \tl_gset:Nn }

```

```

4471 \cs_new_protected_nopar:Npn \tl_rescan:nn
4472 { \__tl_set_rescan:NNnn \prg_do_nothing: \use:n }
4473 \cs_new_protected:Npn \__tl_set_rescan:NNnn #1#2#3#4
4474 {
4475   \group_begin:
4476   \exp_args:No \etex_everyeof:D { \c__tl_rescan_marker_tl \exp_not:N }
4477   \tex_endlinechar:D \c_minus_one
4478   \tex_newlinechar:D \c_minus_one
4479   #3 \scan_stop:
4480   \use:x
4481   {
4482     \group_end:
4483     #1 \exp_not:N #2
4484     {
4485       \exp_after:wN \__tl_rescan:w
4486       \exp_after:wN \prg_do_nothing:
4487       \etex_scantokens:D {#4}
4488     }
4489   }
4490 }
4491 \use:x
4492 {
4493   \cs_new:Npn \exp_not:N \__tl_rescan:w ##1
4494   \c__tl_rescan_marker_tl
4495   { \exp_not:N \exp_not:o { ##1 } }
4496 }
4497 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno , Nnx }
4498 \cs_generate_variant:Nn \tl_set_rescan:Nnn { c , cno , cnx }
4499 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno , Nnx }
4500 \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`

```

4501 \cs_new_protected:Npn \tl_to_lowercase:n #1
4502 { \tex_lowercase:D {#1} }
4503 \cs_new_protected:Npn \tl_to_uppercase:n #1
4504 { \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` $\langle function \rangle$, $\langle tl_ (g) set:Nx \rangle$, $\langle tl\ var \rangle$, $\langle search\ tokens \rangle$, $\langle replacement\ tokens \rangle$.

```

\__tl_replace_all:Nnn
4505 \cs_new_protected_nopar:Npn \tl_replace_once:Nnn
4506 { \__tl_replace:NNNnn \__tl_replace_once: \tl_set:Nx }
4507 \cs_new_protected_nopar:Npn \tl_greplace_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`


```

4508 { \_tl\_replace:NNNnn \_tl\_replace\_once: \tl\_gset:Nx }
4509 \cs\_new\_protected\_nopar:Npn \tl\_replace\_all:Nnn
4510 { \_tl\_replace:NNNnn \_tl\_replace\_all: \tl\_set:Nx }
4511 \cs\_new\_protected\_nopar:Npn \tl\_greplace\_all:Nnn
4512 { \_tl\_replace:NNNnn \_tl\_replace\_all: \tl\_gset:Nx }
4513 \cs\_generate\_variant:Nn \tl\_replace\_once:Nnn { c }
4514 \cs\_generate\_variant:Nn \tl\_greplace\_once:Nnn { c }
4515 \cs\_generate\_variant:Nn \tl\_replace\_all:Nnn { c }
4516 \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 `\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.

```

4517 \cs\_new\_protected:Npn \_tl\_replace:NNNnn #1#2#3#4#5
4518 {
4519   \tl\_if\_empty:nTF {#4}
4520   {
4521     \_msg\_kernel\_error:nxx { kernel } { empty-search-pattern }
4522     { \tl\_to\_str:n {#5} }
4523   }
4524   {
4525     \group\_align\_safe\_begin:
4526     \cs\_set:Npx \_tl\_tmp:w ##1##2 #4
4527     {
4528       ##2
4529       \exp\_not:N \q\_mark
4530       \exp\_not:N \use\_none\_delimit\_by\_q\_stop:w
4531       \exp\_not:n { \exp\_not:n {#5} }
4532       ##1
4533     }
4534     \group\_align\_safe\_end:
4535     #2 #3
4536     {
4537       \exp\_after:wN #1
4538       #3 \q\_mark #4 \q\_stop
4539     }
4540   }
4541 }

```

```
4542 \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.

```
4543 \cs_new_nopar:Npn \__tl_replace_all:
4544 {
4545   \exp_after:wN \__tl_replace:w
4546   \__tl_tmp:w \__tl_replace_all: \prg_do_nothing:
4547 }
4548 \cs_new_nopar:Npn \__tl_replace_once:
4549 {
4550   \exp_after:wN \__tl_replace:w
4551   \__tl_tmp:w { \__tl_replace_once_end:w \prg_do_nothing: } \prg_do_nothing:
4552 }
4553 \cs_new:Npn \__tl_replace_once_end:w #1 \q_mark #2 \q_stop
4554 { \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.

```
4555 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
4556 { \tl_replace_once:Nnn #1 {#2} { } }
4557 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
4558 { \tl_greplace_once:Nnn #1 {#2} { } }
4559 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
4560 \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.

```
4561 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
4562 { \tl_replace_all:Nnn #1 {#2} { } }
4563 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
4564 { \tl_greplace_all:Nnn #1 {#2} { } }
4565 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
4566 \cs_generate_variant:Nn \tl_gremove_all:Nn { c }
```

10.7 Token list conditionals

```
\tl_if_blank_p:n
\tl_if_blank_p:V
\tl_if_blank_p:o
\tl_if_blank:nTF
\tl_if_blank:VTF
\tl_if_blank:oTF
\__tl_if_blank_p:NNw
```

TeX skips spaces when reading a non-delimited arguments. Thus, a *<token list>* is blank if and only if `\use_none:n <token list> ?` is empty after one expansion. The auxiliary

`_tl_if_empty_return:o` is a fast emptiness test, converting its argument to a string (after one expansion) and using the test `\if_meaning:w \q_nil ... \q_nil`.

```

4567 \prg_new_conditional:Npnn \tl\_if\_blank:n #1 { p , T , F , TF }
4568 { \_tl\_if\_empty\_return:o { \use\_none:n #1 ? } }
4569 \cs_generate_variant:Nn \tl\_if\_blank_p:n { V }
4570 \cs_generate_variant:Nn \tl\_if\_blank:nT { V }
4571 \cs_generate_variant:Nn \tl\_if\_blank:nF { V }
4572 \cs_generate_variant:Nn \tl\_if\_blank:nTF { V }
4573 \cs_generate_variant:Nn \tl\_if\_blank_p:n { o }
4574 \cs_generate_variant:Nn \tl\_if\_blank:nT { o }
4575 \cs_generate_variant:Nn \tl\_if\_blank:nF { o }
4576 \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
4577 \prg_new_conditional:Npnn \tl\_if\_empty:N #1 { p , T , F , TF }
4578 {
4579   \if\_meaning:w #1 \c\_empty\_tl
4580   \prg\_return\_true:
4581   \else:
4582     \prg\_return\_false:
4583   \fi:
4584 }
4585 \cs_generate_variant:Nn \tl\_if\_empty\_p:N { c }
4586 \cs_generate_variant:Nn \tl\_if\_empty:NT { c }
4587 \cs_generate_variant:Nn \tl\_if\_empty:NF { c }
4588 \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` Convert the argument to a string: this will be empty if and only if the argument is. Then
`\tl_if_empty_p:V` `\if_meaning:w \q_nil ... \q_nil` is true if and only if the string ... is empty. It
`\tl_if_empty:nTF` could be tempting to use `\if_meaning:w \q_nil #1 \q_nil` directly. This fails on a
`\tl_if_empty:VTF` token list starting with `\q_nil` of course but more troubling is the case where argument
is a complete conditional such as `\if_true: a \else: b \fi:` because then `\if_true:`
is used by `\if_meaning:w`, the test turns out false, the `\else:` executes the false
branch, the `\fi:` ends it and the `\q_nil` at the end starts executing...

```

4589 \prg_new_conditional:Npnn \tl\_if\_empty:n #1 { p , TF , T , F }
4590 {
4591   \exp\_after:wN \if\_meaning:w \exp\_after:wN \q\_nil \tl\_to\_str:n {#1} \q\_nil
4592   \prg\_return\_true:
4593   \else:
4594     \prg\_return\_false:
4595   \fi:
4596 }
4597 \cs_generate_variant:Nn \tl\_if\_empty\_p:n { V }
4598 \cs_generate_variant:Nn \tl\_if\_empty:nTF { V }

```

```

4599 \cs_generate_variant:Nn \tl_if_empty:nT { V }
4600 \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 The auxiliary function _tl_if_empty_return:o is for use in various token list conditionals which reduce to testing if a given token list is empty after applying a simple function to it. The test for emptiness is based on \tl_if_empty: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.

\tl_if_empty:oTF

_tl_if_empty_return:o

```

4601 \cs_new:Npn \_tl_if_empty_return:o #1
4602 {
4603   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
4604   \tl_to_str:n \exp_after:wN {#1} \q_nil
4605   \prg_return_true:
4606   \else:
4607     \prg_return_false:
4608   \fi:
4609 }
4610 \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
4611 { \_tl_if_empty_return:o {#1} }

```

(End definition for \tl_if_empty:o. These functions are documented on page ??.)

\tl_if_eq_p:NN Returns \c_true_bool if and only if the two token list variables are equal.

```

4612 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
4613 {
4614   \if_meaning:w #1 #2
4615   \prg_return_true:
4616   \else:
4617     \prg_return_false:
4618   \fi:
4619 }
4620 \cs_generate_variant:Nn \tl_if_eq_p:NN { Nc , c , cc }
4621 \cs_generate_variant:Nn \tl_if_eq:NNTF { Nc , c , cc }
4622 \cs_generate_variant:Nn \tl_if_eq:NNTF { Nc , c , cc }
4623 \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.

```

4624 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
4625 {
4626   \group_begin:
4627     \tl_set:Nn \l__tl_internal_a_tl {#1}
4628     \tl_set:Nn \l__tl_internal_b_tl {#2}
4629     \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
4630     \group_end:
4631     \prg_return_true:
4632   \else:

```

```

4633     \group_end:
4634     \prg_return_false:
4635     \fi:
4636   }
4637   \tl_new:N \l__tl_internal_a_tl
4638   \tl_new:N \l__tl_internal_b_tl

```

(End definition for `\tl_if_eq:nnTF`. 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)`.

```

4639 \cs_new_protected_nopar:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
4640 \cs_new_protected_nopar:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
4641 \cs_new_protected_nopar:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
4642 \cs_generate_variant:Nn \tl_if_in:NnT { c }
4643 \cs_generate_variant:Nn \tl_if_in:NnF { c }
4644 \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 the emptiness test for robustness. The function `__tl_-`
`\tl_if_in:VnTF` `tmp:w` removes tokens until the first occurrence of #2. If this does not appear in #1, then
`\tl_if_in:onTF` the final #2 is removed, leaving an empty token list. Otherwise some tokens remain, and
`\tl_if_in:noTF` the test is false. See `\tl_if_empty: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.

```

4645 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
4646 {
4647   \cs_set:Npn \__tl_tmp:w ##1 #2 { }
4648   \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
4649   { \prg_return_false: } { \prg_return_true: }
4650 }
4651 \cs_generate_variant:Nn \tl_if_in:nnT { V , o , no }
4652 \cs_generate_variant:Nn \tl_if_in:nnF { V , o , no }
4653 \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`.

```

4654 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
4655 \cs_new:Npn \tl_if_single:NT { \exp_args:No \tl_if_single:nT }
4656 \cs_new:Npn \tl_if_single:NF { \exp_args:No \tl_if_single:nF }
4657 \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` This test is similar to `\tl_if_empty:nTF`. Expanding `\use_none:nn #1 ??` once yields an empty result if `#1` is blank, a single `?` if `#1` has a single item, and otherwise yields some tokens ending with `??`. Then, `\tl_to_str:n` makes sure there are no odd category codes. An earlier version would compare the result to a single `?` using string comparison, but the Lua call is slow in LuaTeX. Instead, `__tl_if_single:nnw` picks the second token in front of it. If `#1` is empty, this token will be the trailing `?` and the catcode test yields `false`. If `#1` has a single item, the token will be `^` and the catcode test yields `true`. Otherwise, it will be one of the characters resulting from `\tl_to_str:n`, and the catcode test yields `false`. Note that `\if_catcode:w` takes care of the expansions, and that `\tl_to_str:n` (the `\detokenize` primitive) actually expands tokens until finding a begin-group token.

```

4658 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
4659 {
4660   \if_catcode:w ^ \exp_after:wN \__tl_if_single:nnw
4661     \tl_to_str:n \exp_after:wN { \use_none:nn #1 ?? } ^ ? \q_stop
4662     \prg_return_true:
4663   \else:
4664     \prg_return_false:
4665   \fi:
4666 }
4667 \cs_new:Npn \__tl_if_single:nnw #1#2#3 \q_stop {#2}

```

(End definition for `\tl_if_single:n`. These functions are documented on page 95.)

`\tl_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.
`\tl_case:cn` That is achieved by using the test input as the final case, as this will always be true. The
`\tl_case:NnTF` trick is then to tidy up the output such that the appropriate case code plus either the
`\tl_case:cnTF` true or false branch code is inserted.
`__tl_case:nnTF`
`__tl_case:Nw`

```

4668 \cs_new:Npn \tl_case:Nn #1#2
4669 {
4670   \tex_romannumeral:D
4671   \__tl_case:NnTF #1 {#2} { } { }
4672 }
4673 \cs_new:Npn \tl_case:NnT #1#2#3
4674 {
4675   \tex_romannumeral:D
4676   \__tl_case:NnTF #1 {#2} {#3} { }
4677 }
4678 \cs_new:Npn \tl_case:NnF #1#2#3
4679 {
4680   \tex_romannumeral:D
4681   \__tl_case:NnTF #1 {#2} { } {#3}
4682 }
4683 \cs_new:Npn \tl_case:NnTF #1#2
4684 {
4685   \tex_romannumeral:D
4686   \__tl_case:NnTF #1 {#2}
4687 }

```

```

4688 \cs_new:Npn \__tl_case:NnTF #1#2#3#4
4689 { \__tl_case:Nw #1 #2 #1 { } \q_mark {#3} \q_mark {#4} \q_stop }
4690 \cs_new:Npn \__tl_case:Nw #1#2#3
4691 {
4692   \tl_if_eq:NNTF #1 #2
4693   { \__tl_case_end:nw {#3} }
4694   { \__tl_case:Nw #1 }
4695 }
4696 \cs_generate_variant:Nn \tl_case:Nn { c }
4697 \cs_generate_variant:Nn \tl_case:NnT { c }
4698 \cs_generate_variant:Nn \tl_case:NnF { c }
4699 \cs_generate_variant:Nn \tl_case:NnTF { c }

```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 will be the code to insert, #2 will be the *next* case to check on and #3 will be all of the rest of the cases code. That means that #4 will be the **true** branch code, and #5 will be tidy up the spare \q_mark and the **false** branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with itself. That means that #1 will be empty, #2 will be the first \q_mark and so #4 will be the **false** code (the **true** code is mopped up by #3).

```

4700 \cs_new:Npn \__prg_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
4701 { \c_zero #1 #4 }
4702 \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 25.)

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
4703 \cs_new:Npn \tl_map_function:nN #1#2
4704 {
4705   \__tl_map_function:Nn #2 #1
4706   \q_recursion_tail
4707   \__prg_break_point:Nn \tl_map_break: { }
4708 }
4709 \cs_new_nopar:Npn \tl_map_function:NN
4710 { \exp_args:No \tl_map_function:nN }
4711 \cs_new:Npn \__tl_map_function:Nn #1#2
4712 {
4713   \__quark_if_recursion_tail_break:nN {#2} \tl_map_break:
4714   #1 {#2} \__tl_map_function:Nn #1
4715 }
4716 \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

```

4717 \cs_new_protected:Npn \tl_map_inline:nn #1#2
4718 {
4719   \int_gincr:N \g__prg_map_int
4720   \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
4721   \exp_args:Nc \__tl_map_function:Nn
4722     { __prg_map_ \int_use:N \g__prg_map_int :w }
4723     #1 \q_recursion_tail
4724   \__prg_break_point:Nn \tl_map_break: { \int_gdecr:N \g__prg_map_int }
4725 }
4726 \cs_new_protected:Npn \tl_map_inline:Nn
4727 { \exp_args:No \tl_map_inline:nn }
4728 \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 <token list> <temp> <action> assigns <temp> to each element and
\tl_map_variable:NNn executes <action>.
\tl_map_variable:cNn
\__tl_map_variable:Nnn
4729 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
4730 {
4731   \__tl_map_variable:Nnn #2 {#3} #1
4732   \q_recursion_tail
4733   \__prg_break_point:Nn \tl_map_break: { }
4734 }
4735 \cs_new_protected_nopar:Npn \tl_map_variable:NNn
4736 { \exp_args:No \tl_map_variable:nNn }
4737 \cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
4738 {
4739   \tl_set:Nn #1 {#3}
4740   \__quark_if_recursion_tail_break:NN #1 \tl_map_break:
4741   \use:n {#2}
4742   \__tl_map_variable:Nnn #1 {#2}
4743 }
4744 \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

```

4745 \cs_new_nopar:Npn \tl_map_break:
4746 { \__prg_map_break:Nn \tl_map_break: { } }
4747 \cs_new_nopar:Npn \tl_map_break:n
4748 { \__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.

```

4749 \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

```

4750 \cs_new:Npn \tl_to_str:N #1 { \etex_detokenize:D \exp_after:wN {#1} }
4751 \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.

```

4752 \cs_new:Npn \tl_use:N #1
4753 {
4754   \tl_if_exist:NTF #1 {#1}
4755   { \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#1} }
4756 }
4757 \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
\tl_count:V the list are read as a single element. Spaces are ignored. __tl_count:n grabs the
\tl_count:o element and replaces it by +1. The 0 ensures that it works on an empty list.

\tl_count:N

```

4758 \cs_new:Npn \tl_count:n #1
4759 {
4760   \int_eval:n
4761   { 0 \tl_map_function:nN {#1} \__tl_count:n }
4762 }
4763 \cs_new:Npn \tl_count:N #1
4764 {
4765   \int_eval:n
4766   { 0 \tl_map_function:NN #1 \__tl_count:n }
4767 }
4768 \cs_new:Npn \__tl_count:n #1 { + \c_one }
4769 \cs_generate_variant:Nn \tl_count:n { V , o }
4770 \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

```

4771 \cs_new:Npn \tl_reverse_items:n #1
4772 {
4773   \__tl_reverse_items:nwNwn #1 ?
4774   \q_mark \__tl_reverse_items:nwNwn
4775   \q_mark \__tl_reverse_items:wn
4776   \q_stop { }
4777 }
4778 \cs_new:Npn \__tl_reverse_items:nwNwn #1 #2 \q_mark #3 #4 \q_stop #5
4779 {
4780   #3 #2

```

```

4781 \q_mark \_tl_reverse_items:nwNwn
4782 \q_mark \_tl_reverse_items:wn
4783 \q_stop { {#1} #5 }
4784 }
4785 \cs_new:Npn \_tl_reverse_items:wn #1 \q_stop #2
4786 { \exp_not:o { \use_none:n #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.

```

4787 \cs_new:Npn \tl_trim_spaces:n #1
4788 { \_tl_trim_spaces:nn { \q_mark #1 } \exp_not:o }
4789 \cs_new_protected:Npn \tl_trim_spaces:N #1
4790 { \tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
4791 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
4792 { \tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
4793 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
4794 \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` 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)*.

```

4795 \cs_set:Npn \_tl_tmp:w #1
4796 {
4797   \cs_new:Npn \_tl_trim_spaces:nn ##1
4798   {
4799     \_tl_trim_spaces_auxi:w
4800     ##1
4801     \q_nil
4802     \q_mark #1 { }
4803     \q_mark \_tl_trim_spaces_auxii:w
4804     \_tl_trim_spaces_auxiii:w
4805     #1 \q_nil
4806     \_tl_trim_spaces_auxiv:w
4807     \q_stop
4808   }
4809   \cs_new:Npn \_tl_trim_spaces_auxi:w ##1 \q_mark #1 ##2 \q_mark ##3

```

```

4810     {
4811         ##3
4812         \_tl_trim_spaces_auxi:w
4813         \q_mark
4814         ##2
4815         \q_mark #1 {##1}
4816     }
4817     \cs_new:Npn \_tl_trim_spaces_auxii:w
4818         \_tl_trim_spaces_auxi:w \q_mark \q_mark ##1
4819     {
4820         \_tl_trim_spaces_auxiii:w
4821         ##1
4822     }
4823     \cs_new:Npn \_tl_trim_spaces_auxiii:w ##1 #1 \q_nil ##2
4824     {
4825         ##2
4826         ##1 \q_nil
4827         \_tl_trim_spaces_auxiii:w
4828     }
4829     \cs_new:Npn \_tl_trim_spaces_auxiv:w ##1 \q_nil ##2 \q_stop ##3
4830     { ##3 { \use_none:n ##1 } }
4831 }
4832 \_tl_tmp:w { ~ }
(End definition for \_tl_trim_spaces:nn.)

```

10.11 Token by token changes

`\q___tl_act_mark` The `\tl_act` functions may be applied to any token list. Hence, we use two private quarks, to allow any token, even quarks, in the token list. Only `\q___tl_act_mark` and `\q___tl_act_stop` may not appear in the token lists manipulated by `_tl_act:NNNnn` functions. The quarks are effectively defined in `l3quark`.
(End definition for `\q___tl_act_mark` and `\q___tl_act_stop`. These variables are documented on page ??.)

`_tl_act:NNNnn` To help control the expansion, `_tl_act:NNNnn` should always be preceded by `\romannumeral` and ends by producing `\c_zero` once the result has been obtained. Then `_tl_act_output:n` loop over tokens, groups, and spaces in #5. The marker `\q___tl_act_mark` is used both `_tl_act_reverse_output:n` to avoid losing outer braces and to detect the end of the token list more easily. The result `_tl_act_loop:w` is stored as an argument for the dummy function `_tl_act_result:n`.
`_tl_act_normal:NwnNNN`
`_tl_act_group:nwnNNN`
`_tl_act_space:wwnNNN`
`_tl_act_end:w`

```

4833 \cs_new:Npn \_tl_act:NNNnn #1#2#3#4#5
4834 {
4835     \group_align_safe_begin:
4836     \_tl_act_loop:w #5 \q___tl_act_mark \q___tl_act_stop
4837     {#4} #1 #2 #3
4838     \_tl_act_result:n { }
4839 }

```

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:wnnnn` gobble the space.

```

4840 \cs_new:Npn \_tl\_act\_loop:w #1 \q\_tl\_act\_stop
4841 {
4842   \tl\_if\_head\_is\_N\_type:nTF {#1}
4843   { \_tl\_act\_normal:Nwnnnn }
4844   {
4845     \tl\_if\_head\_is\_group:nTF {#1}
4846     { \_tl\_act\_group:wnnnnn }
4847     { \_tl\_act\_space:wnnnnn }
4848   }
4849   #1 \q\_tl\_act\_stop
4850 }
4851 \cs_new:Npn \_tl\_act\_normal:Nwnnnn #1 #2 \q\_tl\_act\_stop #3#4
4852 {
4853   \if\_meaning:w \q\_tl\_act\_mark #1
4854   \exp\_after:wN \_tl\_act\_end:wn
4855   \fi:
4856   #4 {#3} #1
4857   \_tl\_act\_loop:w #2 \q\_tl\_act\_stop
4858   {#3} #4
4859 }
4860 \cs_new:Npn \_tl\_act\_end:wn #1 \_tl\_act\_result:n #2
4861 { \group\_align\_safe\_end: \c\_zero #2 }
4862 \cs_new:Npn \_tl\_act\_group:wnnnnn #1 #2 \q\_tl\_act\_stop #3#4#5
4863 {
4864   #5 {#3} {#1}
4865   \_tl\_act\_loop:w #2 \q\_tl\_act\_stop
4866   {#3} #4 #5
4867 }
4868 \exp\_last\_unbraced:NNo
4869 \cs_new:Npn \_tl\_act\_space:wnnnnn \c\_space\_tl #1 \q\_tl\_act\_stop #2#3#4#5
4870 {
4871   #5 {#2}
4872   \_tl\_act\_loop:w #1 \q\_tl\_act\_stop
4873   {#2} #3 #4 #5
4874 }

```

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.

```

4875 \cs_new:Npn \_tl\_act\_output:n #1 #2 \_tl\_act\_result:n #3
4876 { #2 \_tl\_act\_result:n { #3 #1 } }
4877 \cs_new:Npn \_tl\_act\_reverse\_output:n #1 #2 \_tl\_act\_result:n #3
4878 { #2 \_tl\_act\_result:n { #1 #3 } }

```

(End definition for `_tl_act:NNNnn`.)

`\tl_reverse:n` The goal here is to reverse without losing spaces nor braces. This is done using the
`\tl_reverse:o` general internal function `__tl_act:NNNnn`. Spaces and “normal” tokens are output on
`\tl_reverse:V` the left of the current output. Grouped tokens are output to the left but without any
`__tl_reverse_normal:nN` reversal within the group. All of the internal functions here drop one argument: this is
`_tl_reverse_group_preserve:nn` needed by `__tl_act:NNNnn` when changing case (to record which direction the change
`__tl_reverse_space:n` is in), but not when reversing the tokens.

```

4879 \cs_new:Npn \tl_reverse:n #1
4880 {
4881   \etex_unexpanded:D \exp_after:wN
4882   {
4883     \tex_romannumeral:D
4884     \__tl_act:NNNnn
4885     \__tl_reverse_normal:nN
4886     \__tl_reverse_group_preserve:nn
4887     \__tl_reverse_space:n
4888     { }
4889     {#1}
4890   }
4891 }
4892 \cs_generate_variant:Nn \tl_reverse:n { o , V }
4893 \cs_new:Npn \__tl_reverse_normal:nN #1#2
4894 { \_tl_act_reverse_output:n {#2} }
4895 \cs_new:Npn \__tl_reverse_group_preserve:nn #1#2
4896 { \_tl_act_reverse_output:n { {#2} } }
4897 \cs_new:Npn \__tl_reverse_space:n #1
4898 { \_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.

```

4899 \cs_new_protected:Npn \tl_reverse:N #1
4900 { \tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
4901 \cs_new_protected:Npn \tl_greverse:N #1
4902 { \tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
4903 \cs_generate_variant:Nn \tl_reverse:N { c }
4904 \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
`\tl_head:n` this is consistent with for example mapping to a list. The empty brace groups in `\tl_-`
`\tl_head:V` `head:n` ensure that a blank argument gives an empty result. The result is returned
`\tl_head:v` within the `\unexpanded` primitive. The approach here is to use `\if_false:` to allow
`\tl_head:f` us to use `}` as the closing delimiter: this is the only safe choice, as any other token
`__tl_head_auxi:nw` would not be able to parse it’s own code. Using a marker, we can see if what we are
`__tl_head_auxii:n` grabbing is exactly the marker, or there is anything else to deal with. Is there is, there
`\tl_head:w`
`\tl_tail:N`
`\tl_tail:n`
`\tl_tail:V`
`\tl_tail:v`
`\tl_tail:f`

is a loop. If not, tidy up and leave the item in the output stream. More detail in <http://tex.stackexchange.com/a/70168>.

```

4905 \cs_new:Npn \tl_head:n #1
4906 {
4907   \etex_unexpanded:D
4908   \if_false: { \fi: \__tl_head_auxi:nw #1 { } \q_stop }
4909 }
4910 \cs_new:Npn \__tl_head_auxi:nw #1#2 \q_stop
4911 { \exp_after:wN \__tl_head_auxii:n \exp_after:wN { \if_false: } \fi: {#1} }
4912 \cs_new:Npn \__tl_head_auxii:n #1
4913 {
4914   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
4915   \tl_to_str:n \exp_after:wN { \use_none:n #1 } \q_nil
4916   \exp_after:wN \use_i:nn
4917   \else:
4918     \exp_after:wN \use_ii:nn
4919   \fi:
4920   {#1}
4921   { \if_false: { \fi: \__tl_head_auxi:nw #1 } }
4922 }
4923 \cs_generate_variant:Nn \tl_head:n { V , v , f }
4924 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
4925 \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.

```

4926 \cs_new:Npn \tl_tail:n #1
4927 {
4928   \etex_unexpanded:D
4929   \tl_if_blank:nTF {#1}
4930   { { } }
4931   { \exp_after:wN { \use_none:n #1 } }
4932 }
4933 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
4934 \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 ??.)

```

\tl_if_head_eq_meaning_p:nN
\tl_if_head_eq_meaning:nNTF
\tl_if_head_eq_charcode_p:nN
\tl_if_head_eq_charcode:nNTF
\tl_if_head_eq_charcode_p:fN
\tl_if_head_eq_charcode:fNTF
\tl_if_head_eq_catcode_p:nN
\tl_if_head_eq_catcode:nNTF

```

Accessing the first token of a token list is tricky in three cases: when it has category code 1 (begin-group token), when it is an explicit space, with category code 10 and character code 32, or when the token list is empty (obviously).

Forgetting temporarily about this issue we would use the following test in `\tl_if_head_eq_charcode:nN`. Here, `\tl_head:w` yields the first token of the token list, then passed to `\exp_not:N`.

```
\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`.

```
4935 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
4936 {
4937   \if_charcode:w
4938     \exp_not:N #2
4939     \tl_if_head_is_N_type:nTF { #1 ? }
4940     {
4941       \exp_after:wN \exp_not:N
4942       \tl_head:w #1 { ? \use_none:nn } \q_stop
4943     }
4944     { \str_head:n {#1} }
4945     \prg_return_true:
4946   \else:
4947     \prg_return_false:
4948   \fi:
4949 }
4950 \cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN { f }
4951 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF { f }
4952 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT { f }
4953 \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.

```
4954 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
4955 {
4956   \if_catcode:w
4957     \exp_not:N #2
4958     \tl_if_head_is_N_type:nTF { #1 ? }
4959     {
4960       \exp_after:wN \exp_not:N
4961       \tl_head:w #1 { ? \use_none:nn } \q_stop
```

```

4962     }
4963     {
4964         \tl_if_head_is_group:nTF {#1}
4965         { \c_group_begin_token }
4966         { \c_space_token }
4967     }
4968     \prg_return_true:
4969 \else:
4970     \prg_return_false:
4971 \fi:
4972 }

```

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.

```

4973 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
4974 {
4975     \tl_if_head_is_N_type:nTF { #1 ? }
4976     { \__tl_if_head_eq_meaning_normal:nN }
4977     { \__tl_if_head_eq_meaning_special:nN }
4978     {#1} #2
4979 }
4980 \cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
4981 {
4982     \exp_after:wN \if_meaning:w
4983     \tl_head:w #1 { ?? \use_none:nnn } \q_stop #2
4984     \prg_return_true:
4985 \else:
4986     \prg_return_false:
4987 \fi:
4988 }
4989 \cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
4990 {
4991     \if_charcode:w \str_head:n {#1} \exp_not:N #2
4992     \exp_after:wN \use:n
4993 \else:
4994     \prg_return_false:
4995     \exp_after:wN \use_none:n
4996 \fi:
4997 {
4998     \if_catcode:w \exp_not:N #2
4999         \tl_if_head_is_group:nTF {#1}
5000         { \c_group_begin_token }
5001         { \c_space_token }
5002     \prg_return_true:
5003 \else:

```



```

5004         \prg_return_false:
5005     \fi:
5006 }
5007 }

```

(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”.

```

5008 \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
5009 {
5010     \__str_if_eq_x_return:nn
5011     { \exp_not:o { \use:n #1 { } } }
5012     { \exp_not:n { #1 { } } }
5013 }

```

(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

```

5014 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
5015 {
5016     \if_catcode:w *
5017         \exp_after:wN \use_none:n
5018         \exp_after:wN {
5019             \exp_after:wN {
5020                 \token_to_str:N #1 ?
5021             }
5022         }
5023     *
5024     \prg_return_false:
5025 \else:
5026     \prg_return_true:
5027 \fi:
5028 }

```

(End definition for `\tl_if_head_is_group:n`. These functions are documented on page 101.)

`\tl_if_head_is_space_p:n`
`\tl_if_head_is_space:nTF`
`__tl_if_head_is_space:w`

The auxiliary’s argument is all that is before the first explicit space in `?#1?~`. If that is a single ? the test yields `true`. Otherwise, that is more than one token, and the test yields `false`. The work is done within braces (with an `\if_false: { \fi: ... }` construction) both to hide potential alignment tab characters from $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ in a table, and to allow for removing what remains of the token list after its first space. The `\tex_romannumeral:D` and `\c_zero` ensure that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).

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.

```

5029 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
5030 {
5031   \tex_romannumeral:D \if_false: { \fi:
5032     \__tl_if_head_is_space:w ? #1 ? ~ }
5033 }
5034 \cs_new:Npn \__tl_if_head_is_space:w #1 ~
5035 {
5036   \tl_if_empty:oTF { \use_none:n #1 }
5037   { \exp_after:wN \c_zero \exp_after:wN \prg_return_true: }
5038   { \exp_after:wN \c_zero \exp_after:wN \prg_return_false: }
5039   \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
5040 }

```

(End definition for `\tl_if_head_is_space:n`. These functions are documented on page 102.)

10.13 Using a single item

`\tl_item:nn` The idea here is to find the offset of the item from the left, then use a loop to grab
`\tl_item:Nn` the correct item. If the resulting offset is too large, then `\quark_if_recursion_tail-`
`\tl_item:cn` `stop:n` terminates the loop, and returns nothing at all.
`__tl_item:nn`

```

5041 \cs_new:Npn \tl_item:nn #1#2
5042 {
5043   \exp_args:Nf \__tl_item:nn
5044   {
5045     \int_eval:n
5046     {
5047       \int_compare:nNnT {#2} < \c_zero
5048       { \tl_count:n {#1} + \c_one + }
5049       #2
5050     }
5051   }
5052   #1
5053   \q_recursion_tail
5054   \__prg_break_point:
5055 }
5056 \cs_new:Npn \__tl_item:nn #1#2
5057 {
5058   \__quark_if_recursion_tail_break:nN {#2} \__prg_break:
5059   \int_compare:nNnTF {#1} = \c_one
5060   { \__prg_break:n { \exp_not:n {#2} } }
5061   { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
5062 }
5063 \cs_new_nopar:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
5064 \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 ??.)

10.14 Viewing token lists

\tl_show:N Showing token list variables is done after checking that the variable is defined (see `__-`
\tl_show:c `kernel_register_show:N`.

```

5065 \cs_new_protected:Npn \tl_show:N #1
5066 {
5067   \tl_if_exist:NTF #1
5068   { \cs_show:N #1 }
5069   {
5070     \__msg_kernel_error:nnx { kernel } { variable-not-defined }
5071     { \token_to_str:N #1 }
5072   }
5073 }
5074 \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.

```

5075 \cs_new_protected:Npn \tl_show:n #1
5076 { \__msg_show_variable:n { > ~ \tl_to_str:n {#1} } }

```

(End definition for \tl_show:n. This function is documented on page 103.)

10.15 Scratch token lists

\g_tmpa_tl Global temporary token list variables. They are supposed to be set and used immediately,
\g_tmpb_tl with no delay between the definition and the use because you can't count on other macros
 not to redefine them from under you.

```

5077 \tl_new:N \g_tmpa_tl
5078 \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 These are local temporary token list variables. Be sure not to assume that the value you
\l_tmpb_tl put into them will survive for long—see discussion above.

```

5079 \tl_new:N \l_tmpa_tl
5080 \tl_new:N \l_tmpb_tl

```

(End definition for \l_tmpa_tl and \l_tmpb_tl. These variables are documented on page 103.)

10.16 Deprecated functions

\tl_case:Nnn Deprecated 2013-07-15.

\tl_case:cnn

```

5081 \cs_new_eq:NN \tl_case:Nnn \tl_case:NnF
5082 \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 ??.)

```

5083 </initex | package>

```

11 l3str implementation

```

5084 <*initex | package>
5085 <@@=str>

\str_head:n After \tl_to_str:n, we have a list of character tokens, all with category code 12, except
\str_tail:n the space, which has category code 10. Directly using \tl_head:w would thus lose leading
\__str_head:w spaces. Instead, we take an argument delimited by an explicit space, and then only use
\__str_tail:w \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.

```

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`.

```

5086 \cs_new:Npn \str_head:n #1
5087 {
5088   \exp_after:wN \__str_head:w
5089   \tl_to_str:n {#1}
5090   { { } } ~ \q_stop
5091 }
5092 \cs_new:Npn \__str_head:w #1 ~ %
5093 { \tl_head:w #1 { ~ } }
5094 \cs_new:Npn \str_tail:n #1
5095 {
5096   \exp_after:wN \__str_tail:w
5097   \reverse_if:N \if_charcode:w
5098     \scan_stop: \tl_to_str:n {#1} X X \q_stop
5099 }
5100 \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 104.)

11.1 String comparisons

```

\__str_if_eq_x:nn
\__str_escape_x:n

```

String comparisons rely on the primitive `\(pdf)strcmp` if available: LuaTeX does not have it, so emulation is required. As the net result is that we do not *always* use the primitive, the correct approach is to wrap up in a function with defined behaviour. That's done by providing a wrapper and then redefining in the LuaTeX case. Note that the necessary Lua code is covered in `l3bootstrap`: long-term this may need to go into a separate Lua file, but at present it's somewhere that spaces are not skipped for ease-of-input. The need to detokenize and force expansion of input arises from the case where a `#` token is used in the input, *e.g.* `__str_if_eq_x:nn {#} { \tl_to_str:n {#} }`, which otherwise will fail as `\luaescapestring:D` does not double such tokens.

```

5101 \cs_new:Npn \__str_if_eq_x:nn #1#2 { \pdfstrcmp:D {#1} {#2} }

```

```

5102 \luatex_if_engine:T
5103 {
5104   \cs_set:Npn \__str_if_eq_x:nn #1#2
5105   {
5106     \luatex_directlua:D
5107     {
5108       l3kernel.strcmp
5109       (
5110         " \__str_escape_x:n {#1} " ,
5111         " \__str_escape_x:n {#2} "
5112       )
5113     }
5114   }
5115   \cs_new:Npn \__str_escape_x:n #1
5116   {
5117     \luatex_luaescapestring:D
5118     {
5119       \etex_detokenize:D \exp_after:wN { \luatex_expanded:D {#1} }
5120     }
5121   }
5122 }

```

(End definition for __str_if_eq_x:nn.)

`__str_if_eq_x_return:nn` It turns out that we often need to compare a token list with the result of applying some function to it, and return with `\prg_return_true/false:`. This test is similar to `\str_if_eq:nnTF` (see `l3str`), but is hard-coded for speed.

```

5123 \cs_new:Npn \__str_if_eq_x_return:nn #1 #2
5124 {
5125   \if_int_compare:w \__str_if_eq_x:nn {#1} {#2} = \c_zero
5126   \prg_return_true:
5127   \else:
5128     \prg_return_false:
5129   \fi:
5130 }

```

(End definition for __str_if_eq_x_return:nn.)

`\str_if_eq_p:nn` Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore make life a bit clearer. The `nn` and `xx` versions are created directly as this is most efficient.

```

5131 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
5132 {
5133   \if_int_compare:w \__str_if_eq_x:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
5134   = \c_zero
5135   \prg_return_true: \else: \prg_return_false: \fi:
5136 }
5137 \cs_generate_variant:Nn \str_if_eq_p:nn { V , o }
5138 \cs_generate_variant:Nn \str_if_eq_p:nn { nV , no , VV }
5139 \cs_generate_variant:Nn \str_if_eq:nnT { V , o }
5140 \cs_generate_variant:Nn \str_if_eq:nnT { nV , no , VV }

```

`\str_if_eq_p:nV`
`\str_if_eq_p:no`
`\str_if_eq_p:VV`
`\str_if_eq_x_p:nn`
`\str_if_eq:nnTF`
`\str_if_eq:nVTF`
`\str_if_eq:nnTF`
`\str_if_eq:nVTF`
`\str_if_eq:nnTF`
`\str_if_eq:nnTF`
`\str_if_eq_x:nnTF`

```

5141 \cs_generate_variant:Nn \str_if_eq:nnF { V , o }
5142 \cs_generate_variant:Nn \str_if_eq:nnF { nV , no , VV }
5143 \cs_generate_variant:Nn \str_if_eq:nnTF { V , o }
5144 \cs_generate_variant:Nn \str_if_eq:nnTF { nV , no , VV }
5145 \prg_new_conditional:Npnn \str_if_eq_x:nn #1#2 { p , T , F , TF }
5146 {
5147     \if_int_compare:w \__str_if_eq_x:nn {#1} {#2} = \c_zero
5148     \prg_return_true: \else: \prg_return_false: \fi:
5149 }

```

(End definition for `\str_if_eq:nn` and others. These functions are documented on page 105.)

`__str_if_eq_x_return:nn` (End definition for `__str_if_eq_x_return:nn`.)

`\str_case:nn` Much the same as `\tl_case:nn(TF)` here: just a change in the internal comparison.

```

\str_case:on 5150 \cs_new:Npn \str_case:nn #1#2
\str_case_x:nn 5151 {
\str_case:nnTF 5152     \tex_romannumeral:D
\str_case:onTF 5153     \__str_case:nnTF {#1} {#2} { } { }
\str_case_x:nnTF 5154 }
\__str_case:nnTF 5155 \cs_new:Npn \str_case:nnT #1#2#3
\__str_case_x:nnTF 5156 {
\__str_case:nw 5157     \tex_romannumeral:D
\__str_case_x:nw 5158     \__str_case:nnTF {#1} {#2} {#3} { }
\__str_case_end:nw 5159 }
5160 \cs_new:Npn \str_case:nnF #1#2
5161 {
5162     \tex_romannumeral:D
5163     \__str_case:nnTF {#1} {#2} { } { }
5164 }
5165 \cs_new:Npn \str_case:nnTF #1#2
5166 {
5167     \tex_romannumeral:D
5168     \__str_case:nnTF {#1} {#2}
5169 }
5170 \cs_new:Npn \__str_case:nnTF #1#2#3#4
5171 { \__str_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
5172 \cs_generate_variant:Nn \str_case:nn { o }
5173 \cs_generate_variant:Nn \str_case:nnT { o }
5174 \cs_generate_variant:Nn \str_case:nnF { o }
5175 \cs_generate_variant:Nn \str_case:nnTF { o }
5176 \cs_new:Npn \__str_case:nw #1#2#3
5177 {
5178     \str_if_eq:nnTF {#1} {#2}
5179     { \__str_case_end:nw {#3} }
5180     { \__str_case:nw {#1} }
5181 }
5182 \cs_new:Npn \str_case_x:nn #1#2
5183 {
5184     \tex_romannumeral:D
5185     \__str_case_x:nnTF {#1} {#2} { } { }

```

```

5186 }
5187 \cs_new:Npn \str_case_x:nnT #1#2#3
5188 {
5189   \tex_romannumeral:D
5190   \__str_case_x:nnTF {#1} {#2} {#3} { }
5191 }
5192 \cs_new:Npn \str_case_x:nnF #1#2
5193 {
5194   \tex_romannumeral:D
5195   \__str_case_x:nnTF {#1} {#2} { }
5196 }
5197 \cs_new:Npn \str_case_x:nnTF #1#2
5198 {
5199   \tex_romannumeral:D
5200   \__str_case_x:nnTF {#1} {#2}
5201 }
5202 \cs_new:Npn \__str_case_x:nnTF #1#2#3#4
5203 { \__str_case_x:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
5204 \cs_new:Npn \__str_case_x:nw #1#2#3
5205 {
5206   \str_if_eq_x:nnTF {#1} {#2}
5207   { \__str_case_end:nw {#3} }
5208   { \__str_case_x:nw {#1} }
5209 }
5210 \cs_new_eq:NN \__str_case_end:nw \__prg_case_end:nw

```

(End definition for `\str_case:nn`, `\str_case:on`, and `\str_case_x:nn`. These functions are documented on page 105.)

11.2 String manipulation

```

\str_fold_case:n
  \__str_fold_auxi:w
  \__str_fold_auxii:N
  \__str_fold_auxiii:NNNNNNNN
  \__str_fold_end:w

```

To convert a string to the “caseless” form, the first stage is to remove tokenization. Once that is done, provided the transformed chars are also detokenized then there is no need to worry about category codes. Spaces need to be retained as part of the mapping, so there is a little work to do in the set up. Data to support this process is loaded later in the `expl3` bundle.

```

5211 \cs_new:Npn \str_fold_case:n #1
5212 {
5213   \exp_after:wN \__str_fold_auxi:w \tl_to_str:n {#1}
5214   { ~ \c_empty_tl } \__str_fold_end:w ? ~
5215 }

```

A loop using spaces as delimiters: done in this way there is no issue with spaces in the input. Notice that there is a second inner loop with `__str_fold_auxii:N` for each “word”.

```

5216 \cs_new:Npn \__str_fold_auxi:w #1 ~
5217 {
5218   \__str_fold_auxii:N #1 { ~ \c_space_tl }
5219   \__str_fold_auxi:w
5220 }

```

The idea here is to take a single token and convert it to its decimal character code. This can then be used to split up the input into 100 separate manageable lists for comparison on a case-by-case basis.

```

5221 \cs_new:Npn \__str_fold_auxii:N #1
5222 {
5223   \exp_after:wN \__str_fold_auxiii:NNNNNNNN
5224   \int_use:N \__int_eval:w 1000000 + '#1 \__int_eval_end: #1
5225 }

```

At this stage, use a slow-but-expandable string case selection to look for a matching char. If one is not found then retain the input as-is. This also does some cleanup to allow a simple termination of the two loops.

```

5226 \cs_new:Npn \__str_fold_auxiii:NNNNNNNN #1#2#3#4#5#6#7#8
5227 {
5228   \exp_args:NNv \str_case_x:nnF #8
5229   { c__str_fold_ #6 _ #7 _t1 }
5230   {
5231     #8
5232     \exp_after:wN \use_none:n #8
5233   }
5234   \__str_fold_auxii:N
5235 }

```

When the end is reached, clean everything up leaving the converted string in the input stream.

```

5236 \cs_new:Npn \__str_fold_end:w ? #1 \__str_fold_auxi:w { }

```

(End definition for `\str_fold_case:n`. This function is documented on page 106.)

11.3 Deprecated functions

```

\str_case:nnn  Deprecated 2013-07-15.
\str_case:onnn 5237 \cs_new_eq:NN \str_case:nnn \str_case:nnF
\str_case_x:nnn 5238 \cs_new_eq:NN \str_case:onn \str_case:onF
5239 \cs_new_eq:NN \str_case_x:nnn \str_case_x:nnF

```

(End definition for `\str_case:nnn`, `\str_case:onn`, and `\str_case_x:nnn`. These functions are documented on page ??.)

```

5240 </initex | package>

```

12 l3seq implementation

The following test files are used for this code: `m3seq002,m3seq003`.

```

5241 <*initex | package>
5242 <@@=seq>

```


A sequence is a control sequence whose top-level expansion is of the form “`\s__seq __seq_item:n {⟨item1⟩} ... __seq_item:n {⟨itemn⟩}`”, with a leading scan mark followed by n items of the same form. An earlier implementation used the structure “`\seq_elt:w ⟨item1⟩ \seq_elt_end: ... \seq_elt:w ⟨itemn⟩ \seq_elt_end:`”. This allowed rapid searching using a delimited function, but was not suitable for items containing `{`, `}` and `#` tokens, and also lead to the loss of surrounding braces around items.

`\s__seq` The variable is defined in the `l3quark` module, loaded later.
(End definition for `\s__seq`. This variable is documented on page 116.)

`__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.

```
5243 \cs_new:Npn \__seq_item:n
5244 {
5245   \__msg_kernel_expandable_error:nn { kernel } { misused-sequence }
5246   \use_none:n
5247 }
```

(End definition for `__seq_item:n`.)

`\l__seq_internal_a_tl` Scratch space for various internal uses.
`\l__seq_internal_b_tl`

```
5248 \tl_new:N \l__seq_internal_a_tl
5249 \tl_new:N \l__seq_internal_b_tl
```

(End definition for `\l__seq_internal_a_tl` and `\l__seq_internal_b_tl`. These variables are documented on page ??.)

`__seq_tmp:w` Scratch function for internal use.

```
5250 \cs_new_eq:NN \__seq_tmp:w ?
```

(End definition for `__seq_tmp:w`.)

`\c_empty_seq` A sequence with no item, following the structure mentioned above.

```
5251 \tl_const:Nn \c_empty_seq { \s__seq }
```

(End definition for `\c_empty_seq`. This variable is documented on page 116.)

12.1 Allocation and initialisation

`\seq_new:N` Sequences are initialized to `\c_empty_seq`.
`\seq_new:c`

```
5252 \cs_new_protected:Npn \seq_new:N #1
5253 {
5254   \__chk_if_free_cs:N #1
5255   \cs_gset_eq:NN #1 \c_empty_seq
5256 }
5257 \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      5258 \cs_new_protected:Npn \seq_clear:N #1
\seq_gclear:N     5259 { \seq_set_eq:NN #1 \c_empty_seq }
\seq_gclear:c     5260 \cs_generate_variant:Nn \seq_clear:N { c }
                  5261 \cs_new_protected:Npn \seq_gclear:N #1
                  5262 { \seq_gset_eq:NN #1 \c_empty_seq }
                  5263 \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  5264 \cs_new_protected:Npn \seq_clear_new:N #1
\seq_gclear_new:N 5265 { \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }
\seq_gclear_new:c 5266 \cs_generate_variant:Nn \seq_clear_new:N { c }
                  5267 \cs_new_protected:Npn \seq_gclear_new:N #1
                  5268 { \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }
                  5269 \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    5270 \cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN
\seq_set_eq:Nc    5271 \cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc
\seq_set_eq:cc    5272 \cs_new_eq:NN \seq_set_eq:cN \tl_set_eq:cN
\seq_gset_eq:NN   5273 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
\seq_gset_eq:cN   5274 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
\seq_gset_eq:Nc   5275 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
\seq_gset_eq:cN   5276 \cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
\seq_gset_eq:cc   5277 \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_from_clist:NN` Setting a sequence from a comma-separated list is done using a simple mapping.

```

\seq_set_from_clist:cN 5278 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
\seq_set_from_clist:Nc 5279 {
\seq_set_from_clist:cc 5280   \tl_set:Nx #1
\seq_set_from_clist:Nn 5281   { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
\seq_set_from_clist:cn 5282 }
\seq_gset_from_clist:NN 5283 \cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
\seq_gset_from_clist:cN 5284 {
\seq_gset_from_clist:Nc 5285   \tl_set:Nx #1
\seq_gset_from_clist:cc 5286   { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
\seq_gset_from_clist:Nn 5287 }
\seq_gset_from_clist:cn 5288 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
\seq_gset_from_clist:cn 5289 {
                    5290   \tl_gset:Nx #1
                    5291   { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
                    5292 }
                    5293 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
                    5294 {
                    5295   \tl_gset:Nx #1

```

```

5296     { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
5297   }
5298   \cs_generate_variant:Nn \seq_set_from_clist:NN { Nc }
5299   \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
5300   \cs_generate_variant:Nn \seq_set_from_clist:Nn { c }
5301   \cs_generate_variant:Nn \seq_gset_from_clist:NN { Nc }
5302   \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
5303   \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_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.

```

5304   \cs_new_protected_nopar:Npn \seq_set_split:Nnn
5305     { \__seq_set_split:Nnnn \tl_set:Nx }
5306   \cs_new_protected_nopar:Npn \seq_gset_split:Nnn
5307     { \__seq_set_split:Nnnn \tl_gset:Nx }
5308   \cs_new_protected:Npn \__seq_set_split:Nnnn #1#2#3#4
5309     {
5310       \tl_if_empty:nTF {#3}
5311       {
5312         \tl_set:Nn \l__seq_internal_a_tl
5313           { \tl_map_function:nN {#4} \__seq_wrap_item:n }
5314       }
5315       {
5316         \tl_set:Nn \l__seq_internal_a_tl
5317           {
5318             \__seq_set_split_auxi:w \prg_do_nothing:
5319             #4
5320             \__seq_set_split_end:
5321           }
5322         \tl_replace_all:Nnn \l__seq_internal_a_tl { #3 }
5323         {
5324           \__seq_set_split_end:
5325           \__seq_set_split_auxi:w \prg_do_nothing:
5326         }
5327         \tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
5328       }
5329       #1 #2 { \s__seq \l__seq_internal_a_tl }
5330     }

```

```

5331 \cs_new:Npn \__seq_set_split_auxi:w #1 \__seq_set_split_end:
5332 {
5333   \exp_not:N \__seq_set_split_auxii:w
5334   \exp_args:No \tl_trim_spaces:n {#1}
5335   \exp_not:N \__seq_set_split_end:
5336 }
5337 \cs_new:Npn \__seq_set_split_auxii:w #1 \__seq_set_split_end:
5338 { \__seq_wrap_item:n {#1} }
5339 \cs_generate_variant:Nn \seq_set_split:Nnn { NnV }
5340 \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`
`\seq_concat:ccc`
`\seq_gconcat:NNN`
`\seq_gconcat:ccc`

When concatenating sequences, one must remove the leading `\s__seq` of the second sequence. The result starts with `\s__seq` (of the first sequence), which stops f-expansion.

```

5341 \cs_new_protected:Npn \seq_concat:NNN #1#2#3
5342 { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
5343 \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
5344 { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
5345 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
5346 \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`
`\seq_if_exist:p:c`
`\seq_if_exist:NTF`
`\seq_if_exist:cTF`

Copies of the cs functions defined in l3basics.

```

5347 \prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N { TF , T , F , p }
5348 \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 ??.)

12.2 Appending data to either end

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

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.

```

5349 \cs_new_protected:Npn \seq_put_left:Nn #1#2
5350 {
5351   \tl_set:Nx #1
5352   {
5353     \exp_not:n { \s__seq \__seq_item:n {#2} }
5354     \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
5355   }
5356 }
5357 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
5358 {
5359   \tl_gset:Nx #1
5360   {
5361     \exp_not:n { \s__seq \__seq_item:n {#2} }
5362     \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
5363   }
5364 }

```

`__seq_put_left_aux:w`

```

5365 \cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
5366 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
5367 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
5368 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
5369 \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
\seq_put_right:NV
\seq_put_right:Nv
\seq_put_right:No
\seq_put_right:Nx
\seq_put_right:cn
\seq_put_right:cV
\seq_put_right:cv
\seq_put_right:co
\seq_put_right:cx
\seq_gput_right:Nn
\seq_gput_right:NV
\seq_gput_right:Nv
\seq_gput_right:No
\seq_gput_right:Nx
\seq_gput_right:cn
\seq_gput_right:cV
\seq_gput_right:cv
\l__seq_remove_seq
\seq_gput_right:co
\seq_gput_right:cx

```

Since there is no trailing marker, adding an item to the right of a sequence simply means wrapping it in `__seq_item:n`.

```

5370 \cs_new_protected:Npn \seq_put_right:Nn #1#2
5371 { \tl_put_right:Nn #1 { \__seq_item:n {#2} } }
5372 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
5373 { \tl_gput_right:Nn #1 { \__seq_item:n {#2} } }
5374 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
5375 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
5376 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
5377 \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 ??.)

12.3 Modifying sequences

This function converts its argument to a proper sequence item in an x-expansion context.

```

5378 \cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }

```

(End definition for `__seq_wrap_item:n`.)

An internal sequence for the removal routines.

```

5379 \seq_new:N \l__seq_remove_seq

```

(End definition for `\l__seq_remove_seq`. This variable is documented on page ??.)

Removing duplicates means making a new list then copying it.

```

5380 \cs_new_protected:Npn \seq_remove_duplicates:N
5381 { \__seq_remove_duplicates:NN \seq_set_eq:NN }
5382 \cs_new_protected:Npn \seq_gremove_duplicates:N
5383 { \__seq_remove_duplicates:NN \seq_gset_eq:NN }
5384 \cs_new_protected:Npn \__seq_remove_duplicates:NN #1#2
5385 {
5386   \seq_clear:N \l__seq_remove_seq
5387   \seq_map_inline:Nn #2
5388   {
5389     \seq_if_in:NnF \l__seq_remove_seq {##1}
5390     { \seq_put_right:Nn \l__seq_remove_seq {##1} }
5391   }
5392   #1 #2 \l__seq_remove_seq
5393 }
5394 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
5395 \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_duplicates:N
\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
\__seq_remove_duplicates:NN

```

```

\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: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.

```

5396 \cs_new_protected:Npn \seq_remove_all:Nn
5397 { \__seq_remove_all_aux:NNn \tl_set:Nx }
5398 \cs_new_protected:Npn \seq_gremove_all:Nn
5399 { \__seq_remove_all_aux:NNn \tl_gset:Nx }
5400 \cs_new_protected:Npn \__seq_remove_all_aux:NNn #1#2#3
5401 {
5402   \__seq_push_item_def:n
5403   {
5404     \str_if_eq:nnT {##1} {#3}
5405     {
5406       \if_false: { \fi: }
5407       \tl_set:Nn \l__seq_internal_b_tl {##1}
5408       #1 #2
5409       { \if_false: } \fi:
5410       \exp_not:o {#2}
5411       \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
5412       { \use_none:nn }
5413     }
5414     \__seq_wrap_item:n {##1}
5415   }
5416   \tl_set:Nn \l__seq_internal_a_tl {#3}
5417   #1 #2 {#2}
5418   \__seq_pop_item_def:
5419 }
5420 \cs_generate_variant:Nn \seq_remove_all:Nn { c }
5421 \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 ??.)

```

\seq_reverse:N
\seq_reverse:c
\seq_greverse:N
\seq_greverse:c
__seq_reverse:NN
__seq_reverse_item:nwn

```

Previously, `\seq_reverse:N` was coded by collecting the items in reverse order after an `\exp_stop_f:` marker.

```

\cs_new_protected:Npn \seq_reverse:N #1
{
  \cs_set_eq:NN \@@_item:n \@@_reverse_item:nw
  \tl_set:Nf #2 { #2 \exp_stop_f: }
}
\cs_new:Npn \@@_reverse_item:nw #1 #2 \exp_stop_f:

```

```

{
  #2 \exp_stop_f:
  \@@_item:n {#1}
}

```

At first, this seems optimal, since we can forget about each item as soon as it is placed after `\exp_stop_f:`. Unfortunately, \TeX 's usual tail recursion does not take place in this case: since the following `_seq_reverse_item:nw` only reads tokens until `\exp_stop_f:`, and never reads the `\@@_item:n {#1}` left by the previous call, \TeX cannot remove that previous call from the stack, and in particular must retain the various macro parameters in memory, until the end of the replacement text is reached. The stack is thus only flushed after all the `_seq_reverse_item:nw` are expanded. Keeping track of the arguments of all those calls uses up a memory quadratic in the length of the sequence. \TeX can then not cope with more than a few thousand items.

Instead, we collect the items in the argument of `\exp_not:n`. The previous calls are cleanly removed from the stack, and the memory consumption becomes linear.

```

5422 \cs_new_protected_nopar:Npn \seq_reverse:N
5423 { \_seq_reverse:NN \tl_set:Nx }
5424 \cs_new_protected_nopar:Npn \seq_greverse:N
5425 { \_seq_reverse:NN \tl_gset:Nx }
5426 \cs_new_protected:Npn \_seq_reverse:NN #1 #2
5427 {
5428   \cs_set_eq:NN \_seq_tmp:w \_seq_item:n
5429   \cs_set_eq:NN \_seq_item:n \_seq_reverse_item:nwn
5430   #1 #2 { #2 \exp_not:n { } }
5431   \cs_set_eq:NN \_seq_item:n \_seq_tmp:w
5432 }
5433 \cs_new:Npn \_seq_reverse_item:nwn #1 #2 \exp_not:n #3
5434 {
5435   #2
5436   \exp_not:n { \_seq_item:n {#1} #3 }
5437 }
5438 \cs_generate_variant:Nn \seq_reverse:N { c }
5439 \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End definition for `\seq_reverse:N` and others. These functions are documented on page ??.)

12.4 Sequence conditionals

`\seq_if_empty_p:N`
`\seq_if_empty_p:c`
`\seq_if_empty:N $\text{\textit{TF}}$`
`\seq_if_empty:c $\text{\textit{TF}}$`

Similar to token lists, we compare with the empty sequence.

```

5440 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
5441 {
5442   \if_meaning:w #1 \c_empty_seq
5443   \prg_return_true:
5444   \else:
5445     \prg_return_false:
5446   \fi:
5447 }
5448 \cs_generate_variant:Nn \seq_if_empty_p:N { c }

```

```

5449 \cs_generate_variant:Nn \seq_if_empty:NT { c }
5450 \cs_generate_variant:Nn \seq_if_empty:NF { c }
5451 \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.

```

5452 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
5453 { T , F , TF }
5454 {
5455   \group_begin:
5456   \tl_set:Nn \l__seq_internal_a_tl {#2}
5457   \cs_set_protected:Npn \__seq_item:n ##1
5458   {
5459     \tl_set:Nn \l__seq_internal_b_tl {##1}
5460     \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
5461     \exp_after:wN \__seq_if_in:
5462     \fi:
5463   }
5464   #1
5465   \group_end:
5466   \prg_return_false:
5467   \__prg_break_point:
5468 }
5469 \cs_new_nopar:Npn \__seq_if_in:
5470 { \__prg_break:n { \group_end: \prg_return_true: } }
5471 \cs_generate_variant:Nn \seq_if_in:NnT { NV , Nv , No , Nx }
5472 \cs_generate_variant:Nn \seq_if_in:NnT { c , cV , cv , co , cx }
5473 \cs_generate_variant:Nn \seq_if_in:NnF { NV , Nv , No , Nx }
5474 \cs_generate_variant:Nn \seq_if_in:NnF { c , cV , cv , co , cx }
5475 \cs_generate_variant:Nn \seq_if_in:NnTF { NV , Nv , No , Nx }
5476 \cs_generate_variant:Nn \seq_if_in:NnTF { c , cV , cv , co , cx }

```

(End definition for `\seq_if_in:NnTF` and others. These functions are documented on page ??.)

12.5 Recovering data from sequences

`__seq_pop:NNNN` The two pop functions share their emptiness tests. We also use a common emptiness test for all branching get and pop functions.

```

5477 \cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
5478 {
5479   \if_meaning:w #3 \c_empty_seq
5480   \tl_set:Nn #4 { \q_no_value }
5481   \else:
5482     #1#2#3#4
5483   \fi:

```



```

5484 }
5485 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
5486 {
5487   \if_meaning:w #3 \c_empty_seq
5488     % \tl_set:Nn #4 { \q_no_value }
5489     \prg_return_false:
5490   \else:
5491     #1#2#3#4
5492     \prg_return_true:
5493   \fi:
5494 }

```

(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

```

5495 \cs_new_protected:Npn \seq_get_left:NN #1#2
5496 {
5497   \tl_set:Nx #2
5498   {
5499     \exp_after:wN \__seq_get_left:wnw
5500     #1 \__seq_item:n { \q_no_value } \q_stop
5501   }
5502 }
5503 \cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \q_stop
5504 { \exp_not:n {#2} }
5505 \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 The approach to popping an item is pretty similar to that to get an item, with the only
\seq_pop_left:cN difference being that the sequence itself has to be redefined. This makes it more sensible
\seq_gpop_left:NN to use an auxiliary function for the local and global cases.
\seq_gpop_left:cN

```

5506 \cs_new_protected_nopar:Npn \seq_pop_left:NN
5507   { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_set:Nn }
5508 \cs_new_protected_nopar:Npn \seq_gpop_left:NN
5509   { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_gset:Nn }
5510 \cs_new_protected:Npn \__seq_pop_left:NNN #1#2#3
5511   { \exp_after:wN \__seq_pop_left:wnwNNN #2 \q_stop #1#2#3 }
5512 \cs_new_protected:Npn \__seq_pop_left:wnwNNN
5513   #1 \__seq_item:n #2#3 \q_stop #4#5#6
5514   {
5515     #4 #5 { #1 #3 }
5516     \tl_set:Nn #6 {#2}
5517   }
5518 \cs_generate_variant:Nn \seq_pop_left:NN { c }
5519 \cs_generate_variant:Nn \seq_gpop_left:NN { c }

```

(End definition for \seq_pop_left:NN and \seq_gpop_left:cN. These functions are documented on page ??.)

`\seq_get_right:NN` First remove `\s__seq` and prepend `\q_no_value`, then take two arguments at a time.
`\seq_get_right:cN` Before the right-hand end of the sequence, this is a brace group followed by `__seq_`
`__seq_get_right_loop:nn` `item:n`, both removed by `\use_none:nn`. At the end of the sequence, the two question marks are taken by `\use_none:nn`, and the assignment is placed before the right-most item. In the next iteration, `__seq_get_right_loop:nn` receives two empty arguments, and `\use_none:nn` stops the loop.

```

5520 \cs_new_protected:Npn \seq_get_right:NN #1#2
5521 {
5522   \exp_after:wN \use_i_ii:nnn
5523   \exp_after:wN \__seq_get_right_loop:nn
5524   \exp_after:wN \q_no_value
5525   #1
5526   { ?? \tl_set:Nn #2 }
5527   { } { }
5528 }
5529 \cs_new_protected:Npn \__seq_get_right_loop:nn #1#2
5530 {
5531   \use_none:nn #2 {#1}
5532   \__seq_get_right_loop:nn
5533 }
5534 \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
`\seq_pop_right:cN` of the same ideas as getting from the right. What is needed is a “flexible length”
`\seq_gpop_right:NN` way to set a token list variable. This is supplied by the `{ \if_false: } \fi:`
`\seq_gpop_right:cN` `... \if_false: { \fi: }` construct. Using an x-type expansion and a “non-expanding”
`__seq_pop_right:NNN` definition for `__seq_item:n`, the left-most $n - 1$ entries in a sequence of n items will
`__seq_pop_right_loop:nn` be stored back in the sequence. That needs a loop of unknown length, hence using the
strange `\if_false:` way of including braces. When the last item of the sequence is reached, the closing brace for the assignment is inserted, and `\tl_set:Nn #3` is inserted in front of the final entry. This therefore does the pop assignment. One more iteration is performed, with an empty argument and `\use_none:nn`, which finally stops the loop.

```

5535 \cs_new_protected_nopar:Npn \seq_pop_right:NN
5536 { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_set:Nx }
5537 \cs_new_protected_nopar:Npn \seq_gpop_right:NN
5538 { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_gset:Nx }
5539 \cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3
5540 {
5541   \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
5542   \cs_set_eq:NN \__seq_item:n \scan_stop:
5543   #1 #2
5544   { \if_false: } \fi: \s__seq
5545   \exp_after:wN \use_i:nnn
5546   \exp_after:wN \__seq_pop_right_loop:nn
5547   #2
5548   {

```

```

5549         \if_false: { \fi: }
5550         \tl_set:Nx #3
5551     }
5552     { } \use_none:nn
5553     \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
5554 }
5555 \cs_new:Npn \__seq_pop_right_loop:nn #1#2
5556 {
5557     #2 { \exp_not:n {#1} }
5558     \__seq_pop_right_loop:nn
5559 }
5560 \cs_generate_variant:Nn \seq_pop_right:NN { c }
5561 \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
5562 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
5563 { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
5564 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
5565 { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
5566 \cs_generate_variant:Nn \seq_get_left:NNT { c }
5567 \cs_generate_variant:Nn \seq_get_left:NNF { c }
5568 \cs_generate_variant:Nn \seq_get_left:NNTF { c }
5569 \cs_generate_variant:Nn \seq_get_right:NNT { c }
5570 \cs_generate_variant:Nn \seq_get_right:NNF { c }
5571 \cs_generate_variant:Nn \seq_get_right:NNTF { c }

```

(End definition for `\seq_get_left:NNTF` and `\seq_get_left:cNTF`. These functions are documented on page ??.)

`\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
5572 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 { T , F , TF }
5573 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
5574 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 { T , F , TF }
5575 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
5576 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2 { T , F , TF }
5577 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_set:Nx #1 #2 }
5578 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2 { T , F , TF }
5579 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_gset:Nx #1 #2 }
5580 \cs_generate_variant:Nn \seq_pop_left:NNT { c }
5581 \cs_generate_variant:Nn \seq_pop_left:NNF { c }
5582 \cs_generate_variant:Nn \seq_pop_left:NNTF { c }
5583 \cs_generate_variant:Nn \seq_gpop_left:NNT { c }
5584 \cs_generate_variant:Nn \seq_gpop_left:NNF { c }
5585 \cs_generate_variant:Nn \seq_gpop_left:NNTF { c }
5586 \cs_generate_variant:Nn \seq_pop_right:NNT { c }
5587 \cs_generate_variant:Nn \seq_pop_right:NNF { c }
5588 \cs_generate_variant:Nn \seq_pop_right:NNTF { c }

```

```

5589 \cs_generate_variant:Nn \seq_gpop_right:NNT { c }
5590 \cs_generate_variant:Nn \seq_gpop_right:NNF { c }
5591 \cs_generate_variant:Nn \seq_gpop_right:NNTF { c }

```

(End definition for `\seq_pop_left:NNTF` and `\seq_pop_left:cNTF`. These functions are documented on page ??.)

`\seq_item:Nn` The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then the stop code `{ ? __prg_break: } { }` will be used by the auxiliary, terminating the loop and returning nothing at all.

`\seq_item:cn`

`__seq_item:wNn`

`__seq_item:nnn`

```

5592 \cs_new:Npn \seq_item:Nn #1
5593 { \exp_after:wN \__seq_item:wNn #1 \q_stop #1 }
5594 \cs_new:Npn \__seq_item:wNn \s__seq #1 \q_stop #2#3
5595 {
5596   \exp_args:Nf \__seq_item:nnn
5597   {
5598     \int_eval:n
5599     {
5600       \int_compare:nNnT {#3} < \c_zero
5601       { \seq_count:N #2 + \c_one + }
5602       #3
5603     }
5604   }
5605   #1
5606   { ? \__prg_break: } { }
5607   \__prg_break_point:
5608 }
5609 \cs_new:Npn \__seq_item:nnn #1#2#3
5610 {
5611   \use_none:n #2
5612   \int_compare:nNnTF {#1} = \c_one
5613   { \__prg_break:n { \exp_not:n {#3} } }
5614   { \exp_args:Nf \__seq_item:nnn { \int_eval:n { #1 - 1 } } }
5615 }
5616 \cs_generate_variant:Nn \seq_item:Nn { c }

```

(End definition for `\seq_item:Nn` and `\seq_item:cn`. These functions are documented on page ??.)

12.6 Mapping to sequences

`\seq_map_break:` To break a function, the special token `__prg_break_point:Nn` is used to find the end of the code. Any ending code is then inserted before the return value of `\seq_map_break:n` is inserted.

`\seq_map_break:n`

```

5617 \cs_new_nopar:Npn \seq_map_break:
5618 { \__prg_map_break:Nn \seq_map_break: { } }
5619 \cs_new_nopar:Npn \seq_map_break:n
5620 { \__prg_map_break:Nn \seq_map_break: }

```

(End definition for `\seq_map_break:.` This function is documented on page 113.)

`\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:NNn` sequence must be `__seq_item:n`, which can be gobbled by `\use_none:n`. At the end
of the loop, #2 is instead ? `\seq_map_break:`, which therefore breaks the loop without
needing to do a (relatively-expensive) quark test.

```

5621 \cs_new:Npn \seq_map_function:NN #1#2
5622 {
5623   \exp_after:wN \use_i_ii:nnn
5624   \exp_after:wN \__seq_map_function:NNn
5625   \exp_after:wN #2
5626   #1
5627   { ? \seq_map_break: } { }
5628   \__prg_break_point:Nn \seq_map_break: { }
5629 }
5630 \cs_new:Npn \__seq_map_function:NNn #1#2#3
5631 {
5632   \use_none:n #2
5633   #1 {#3}
5634   \__seq_map_function:NNn #1
5635 }
5636 \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` The definition of `__seq_item:n` needs to be saved and restored at various points within
`__seq_push_item_def:x` the mapping and manipulation code. That is handled here: as always, this approach uses
`__seq_push_item_def:` global assignments.

```

5637 \cs_new_protected:Npn \__seq_push_item_def:n
5638 {
5639   \__seq_push_item_def:
5640   \cs_gset:Npn \__seq_item:n ##1
5641 }
5642 \cs_new_protected:Npn \__seq_push_item_def:x
5643 {
5644   \__seq_push_item_def:
5645   \cs_gset:Npx \__seq_item:n ##1
5646 }
5647 \cs_new_protected:Npn \__seq_push_item_def:
5648 {
5649   \int_gincr:N \g__prg_map_int
5650   \cs_gset_eq:cN { __prg_map_ \int_use:N \g__prg_map_int :w }
5651   \__seq_item:n
5652 }
5653 \cs_new_protected_nopar:Npn \__seq_pop_item_def:
5654 {
5655   \cs_gset_eq:Nc \__seq_item:n
5656   { __prg_map_ \int_use:N \g__prg_map_int :w }
5657   \int_gdecr:N \g__prg_map_int
5658 }

```

(End definition for `_seq_push_item_def:n` and `_seq_push_item_def:x`.)

`\seq_map_inline:Nn` The idea here is that `_seq_item:n` is already “applied” to each item in a sequence,
`\seq_map_inline:cn` and so an in-line mapping is just a case of redefining `_seq_item:n`.

```
5659 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
5660 {
5661   \_seq_push_item_def:n {#2}
5662   #1
5663   \_prg_break_point:Nn \seq_map_break: { \_seq_pop_item_def: }
5664 }
5665 \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
`\seq_map_variable:Ncn` for the code set up so that the number of # tokens required is as expected.
`\seq_map_variable:cNn`
`\seq_map_variable:ccn`

```
5666 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
5667 {
5668   \_seq_push_item_def:x
5669   {
5670     \tl_set:Nn \exp_not:N #2 {##1}
5671     \exp_not:n {#3}
5672   }
5673   #1
5674   \_prg_break_point:Nn \seq_map_break: { \_seq_pop_item_def: }
5675 }
5676 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
5677 \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
`\seq_count:c` functions: turn each entry into a +1 then use integer evaluation to actually do the math-
`_seq_count:n` ematics.

```
5678 \cs_new:Npn \seq_count:N #1
5679 {
5680   \int_eval:n
5681   {
5682     0
5683     \seq_map_function:NN #1 \_seq_count:n
5684   }
5685 }
5686 \cs_new:Npn \_seq_count:n #1 { + \c_one }
5687 \cs_generate_variant:Nn \seq_count:N { c }
```

(End definition for `\seq_count:N` and `\seq_count:c`. These functions are documented on page ??.)

12.7 Using sequences

`\seq_use:Nnnn` See `\clist_use:Nnnn` for a general explanation. The main difference is that we use `_seq_item:n` as a delimiter rather than commas. We also need to add `_seq_item:n` at various places, and `\s__seq`.

```

\seq_use:cnnn
\_seq_use:NNnNnn
\_seq_use_setup:w
\_seq_use:nwwwwnwn
\_seq_use:nwnn
\seq_use:Nn
\seq_use:cn
5688 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
5689 {
5690   \seq_if_exist:NTF #1
5691   {
5692     \int_case:nnF { \seq_count:N #1 }
5693     {
5694       { 0 } { }
5695       { 1 } { \exp_after:wN \_seq_use:NNnNnn #1 ? { } { } }
5696       { 2 } { \exp_after:wN \_seq_use:NNnNnn #1 {#2} }
5697     }
5698     {
5699       \exp_after:wN \_seq_use_setup:w #1 \_seq_item:n
5700       \q_mark { \_seq_use:nwwwwnwn {#3} }
5701       \q_mark { \_seq_use:nwnn {#4} }
5702       \q_stop { }
5703     }
5704   }
5705   { \_msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#1} }
5706 }
5707 \cs_generate_variant:Nn \seq_use:Nnnn { c }
5708 \cs_new:Npn \_seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
5709 \cs_new:Npn \_seq_use_setup:w \s__seq { \_seq_use:nwwwwnwn { } }
5710 \cs_new:Npn \_seq_use:nwwwwnwn
5711   #1 \_seq_item:n #2 \_seq_item:n #3 \_seq_item:n #4#5
5712   \q_mark #6#7 \q_stop #8
5713   {
5714     #6 \_seq_item:n {#3} \_seq_item:n {#4} #5
5715     \q_mark {#6} #7 \q_stop { #8 #1 #2 }
5716   }
5717 \cs_new:Npn \_seq_use:nwnn #1 \_seq_item:n #2 #3 \q_stop #4
5718   { \exp_not:n { #4 #1 #2 } }
5719 \cs_new:Npn \seq_use:Nn #1#2
5720   { \seq_use:Nnnn #1 {#2} {#2} {#2} }
5721 \cs_generate_variant:Nn \seq_use:Nn { c }

```

(End definition for `\seq_use:Nnnn` and `\seq_use:cnnn`. These functions are documented on page ??.)

12.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

`\seq_push:Nn` Pushing to a sequence is the same as adding on the left.

```

\seq_push:NV
\seq_push:Nv
\seq_push:No
\seq_push:Nx
\seq_push:cn
\seq_push:cV
\seq_push:cV
\seq_push:co
\seq_push:cx
\seq_gpush:Nn
\seq_gpush:NV
\seq_gpush:Nv
\seq_gpush:No

```

```

5725 \cs_new_eq:NN \seq_push:No \seq_put_left:No
5726 \cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx
5727 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
5728 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
5729 \cs_new_eq:NN \seq_push:cv \seq_put_left:cv
5730 \cs_new_eq:NN \seq_push:co \seq_put_left:co
5731 \cs_new_eq:NN \seq_push:cx \seq_put_left:cx
5732 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
5733 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
5734 \cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
5735 \cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
5736 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
5737 \cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv
5738 \cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
5739 \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
`\seq_get:cN` left. So alias are provided.

```

\seq_pop:NN 5742 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
\seq_pop:cN 5743 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
\seq_gpop:NN 5744 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
\seq_gpop:cN 5745 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
5746 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
5747 \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 5748 \prg_new_eq_conditional:NNn \seq_get:NN \seq_get_left:NN { T , F , TF }
\seq_pop:NNTF 5749 \prg_new_eq_conditional:NNn \seq_get:cN \seq_get_left:cN { T , F , TF }
\seq_pop:cNTF 5750 \prg_new_eq_conditional:NNn \seq_pop:NN \seq_pop_left:NN { T , F , TF }
\seq_gpop:NNTF 5751 \prg_new_eq_conditional:NNn \seq_pop:cN \seq_pop_left:cN { T , F , TF }
\seq_gpop:cNTF 5752 \prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
5753 \prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }

```

(End definition for `\seq_get:NNTF` and `\seq_get:cNTF`. These functions are documented on page ??.)

12.9 Viewing sequences

`\seq_show:N` Apply the general `_msg_show_variable:Nnn`.

```

\seq_show:c 5754 \cs_new_protected:Npn \seq_show:N #1
5755 {
5756 \_msg_show_variable:Nnn #1 { seq }
5757 { \seq_map_function:NN #1 \_msg_show_item:n }
5758 }
5759 \cs_generate_variant:Nn \seq_show:N { c }

```

(End definition for `\seq_show:N` and `\seq_show:c`. These functions are documented on page ??.)

12.10 Scratch sequences

`\l_tmpa_seq` Temporary comma list variables.
`\l_tmpb_seq` 5760 `\seq_new:N \l_tmpa_seq`
`\g_tmpa_seq` 5761 `\seq_new:N \l_tmpb_seq`
`\g_tmpb_seq` 5762 `\seq_new:N \g_tmpa_seq`
5763 `\seq_new:N \g_tmpb_seq`
(End definition for \l_tmpa_seq and others. These variables are documented on page 116.)
5764 `\</initex | package>`

13 l3clist implementation

The following test files are used for this code: m3clist002.

5765 `<*initex | package>`
5766 `<@@=clist>`

`\c_empty_clist` An empty comma list is simply an empty token list.
5767 `\cs_new_eq:NN \c_empty_clist \c_empty_tl`
(End definition for \c_empty_clist. This variable is documented on page 125.)

`\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`
5768 `\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.
5769 `\cs_new_protected:Npn __clist_tmp:w { }`
(End definition for __clist_tmp:w.)

13.1 Allocation and initialisation

`\clist_new:N` Internally, comma lists are just token lists.
`\clist_new:c` 5770 `\cs_new_eq:NN \clist_new:N \tl_new:N`
5771 `\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_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` 5772 `\cs_new_protected:Npn \clist_const:Nn #1#2`
`\clist_const:cx` 5773 `{ \tl_const:Nx #1 { __clist_trim_spaces:n {#2} } }`
5774 `\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_clear:N` Clearing comma lists is just the same as clearing token lists.

```
\clist_clear:c 5775 \cs_new_eq:NN \clist_clear:N \tl_clear:N
\clist_gclear:N 5776 \cs_new_eq:NN \clist_clear:c \tl_clear:c
\clist_gclear:c 5777 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
5778 \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 5779 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
\clist_gclear_new:N 5780 \cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c
\clist_gclear_new:c 5781 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
5782 \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 5783 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
\clist_set_eq:Nc 5784 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
\clist_set_eq:cc 5785 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
\clist_gset_eq:NN 5786 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
\clist_gset_eq:cN 5787 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\clist_gset_eq:Nc 5788 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\clist_gset_eq:cN 5789 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
\clist_gset_eq:cc 5790 \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_set_from_seq:NN` Setting a comma list from a comma-separated list is done using a simple mapping. We wrap most items with `\exp_not:n`, and a comma. Items which contain a comma or a space are surrounded by an extra set of braces. The first comma must be removed, except in the case of an empty comma-list.

```
\clist_set_from_seq:cN 5791 \cs_new_protected:Npn \clist_set_from_seq:NN
\clist_set_from_seq:Nc 5792 { \__clist_set_from_seq:NNNN \clist_clear:N \tl_set:Nx }
\clist_set_from_seq:cc 5793 \cs_new_protected:Npn \clist_gset_from_seq:NN
5794 { \__clist_set_from_seq:NNNN \clist_gclear:N \tl_gset:Nx }
\__clist_set_from_seq:NNNN 5795 \cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4
5796 {
5797   \seq_if_empty:NTF #4
5798     { #1 #3 }
5799     {
5800       #2 #3
5801       {
5802         \exp_last_unbraced:Nf \use_none:n
5803         { \seq_map_function:NN #4 \__clist_wrap_item:n }
5804       }
5805     }
5806   }
5807   \cs_new:Npn \__clist_wrap_item:n #1
5808   {
```

```

5809 ,
5810 \tl_if_empty:oTF { \__clist_set_from_seq:w #1 ~ , #1 ~ }
5811 { \exp_not:n {#1} }
5812 { \exp_not:n { {#1} } }
5813 }
5814 \cs_new:Npn \__clist_set_from_seq:w #1 , #2 ~ { }
5815 \cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }
5816 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
5817 \cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }
5818 \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_concat:NNN`
`\clist_concat:ccc`
`\clist_gconcat:NNN`
`\clist_gconcat:ccc`
`__clist_concat:NNNN`

Concatenating comma lists is not quite as easy as it seems, as there needs to be the correct addition of a comma to the output. So a little work to do.

```

5819 \cs_new_protected_nopar:Npn \clist_concat:NNN
5820 { \__clist_concat:NNNN \tl_set:Nx }
5821 \cs_new_protected_nopar:Npn \clist_gconcat:NNN
5822 { \__clist_concat:NNNN \tl_gset:Nx }
5823 \cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
5824 {
5825   #1 #2
5826   {
5827     \exp_not:o #3
5828     \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
5829     \exp_not:o #4
5830   }
5831 }
5832 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
5833 \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`
`\clist_if_exist_p:c`
`\clist_if_exist:N \overline{TF}`
`\clist_if_exist:c \overline{TF}`

Copies of the cs functions defined in l3basics.

```

5834 \prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N { TF , T , F , p }
5835 \prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c { TF , T , F , p }

```

(End definition for \clist_if_exist:N and \clist_if_exist:c. These functions are documented on page ??.)

13.2 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 $\langle trimmed item \rangle$.

```

5836 \cs_new:Npn \__clist_trim_spaces_generic:nw #1#2 ,
5837 {

```

```

5838     \_tl_trim_spaces:nn {#2}
5839     { \exp_args:No \_clist_trim_spaces_generic:nn } {#1}
5840   }
5841 \cs_new:Npn \_clist_trim_spaces_generic:nn #1#2 { #2 {#1} }
(End definition for \_clist_trim_spaces_generic:nw.)

```

_clist_trim_spaces:n The first argument of _clist_trim_spaces:nn is initially empty, and later a comma, namely, as soon as we have added an item to the resulting list. The auxiliary tests for the end of the list, and also prevents empty arguments from finding their way into the output.

```

5842 \cs_new:Npn \_clist_trim_spaces:n #1
5843 {
5844   \_clist_trim_spaces_generic:nw
5845   { \_clist_trim_spaces:nn { } }
5846   \q_mark #1 ,
5847   \q_recursion_tail, \q_recursion_stop
5848 }
5849 \cs_new:Npn \_clist_trim_spaces:nn #1 #2
5850 {
5851   \quark_if_recursion_tail_stop:n {#2}
5852   \tl_if_empty:nTF {#2}
5853   {
5854     \_clist_trim_spaces_generic:nw
5855     { \_clist_trim_spaces:nn {#1} } \q_mark
5856   }
5857   {
5858     #1 \exp_not:n {#2}
5859     \_clist_trim_spaces_generic:nw
5860     { \_clist_trim_spaces:nn { , } } \q_mark
5861   }
5862 }
(End definition for \_clist_trim_spaces:n.)

```

13.3 Adding data to comma lists

```

\_clist_set:Nn
\_clist_set:NV 5863 \cs_new_protected:Npn \clist_set:Nn #1#2
\_clist_set:No 5864 { \tl_set:Nx #1 { \_clist_trim_spaces:n {#2} } }
\_clist_set:Nx 5865 \cs_new_protected:Npn \clist_gset:Nn #1#2
\_clist_set:cn 5866 { \tl_gset:Nx #1 { \_clist_trim_spaces:n {#2} } }
\_clist_set:cV 5867 \cs_generate_variant:Nn \clist_set:Nn { NV , No , Nx , c , cV , co , cx }
\_clist_set:co 5868 \cs_generate_variant:Nn \clist_gset:Nn { NV , No , Nx , c , cV , co , cx }
\_clist_set:cx
(End definition for \clist_set:Nn and others. These functions are documented on page ??.)

```

```

\_clist_gset:Nn
\_clist_put_left:Nn
\_clist_gset:NV
\_clist_put_left:NV
\_clist_gset:No
\_clist_put_left:No
\_clist_gset:Nx
\_clist_put_left:Nx
\_clist_gset:cn
\_clist_put_left:cn
\_clist_gset:cV
\_clist_put_left:cV
\_clist_gset:co
\_clist_put_left:co
\_clist_gset:cx
\_clist_put_left:cx

```

Comma lists cannot hold empty values: there are therefore a couple of sanity checks to avoid accumulating commas.

```

5869 \cs_new_protected_nopar:Npn \clist_put_left:Nn
5870 { \_clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }

```

```

5871 \cs_new_protected_nopar:Npn \clist_gput_left:Nn
5872 { \__clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
5873 \cs_new_protected:Npn \__clist_put_left:NNNn #1#2#3#4
5874 {
5875     #2 \l__clist_internal_clist {#4}
5876     #1 #3 \l__clist_internal_clist #3
5877 }
5878 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
5879 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
5880 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
5881 \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 ??.)

```

```

\clist_put_right:Nn
\clist_put_right:NV
\clist_put_right:No
\clist_put_right:Nx
\clist_put_right:cn
\clist_put_right:cV
\clist_put_right:co
\clist_put_right:cx
\clist_gput_right:Nn
\clist_gput_right:NV
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cn
\clist_gput_right:cV
\clist_gput_right:co
\clist_gput_right:cx
\__clist_put_right:NNNn
\clist_get:NN
\clist_get:cn
\__clist_get:wN
5882 \cs_new_protected_nopar:Npn \clist_put_right:Nn
5883 { \__clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
5884 \cs_new_protected_nopar:Npn \clist_gput_right:Nn
5885 { \__clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
5886 \cs_new_protected:Npn \__clist_put_right:NNNn #1#2#3#4
5887 {
5888     #2 \l__clist_internal_clist {#4}
5889     #1 #3 #3 \l__clist_internal_clist
5890 }
5891 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
5892 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
5893 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
5894 \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 ??.)

```

13.4 Comma lists as stacks

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma.

```

5895 \cs_new_protected:Npn \clist_get:NN #1#2
5896 {
5897     \if_meaning:w #1 \c_empty_clist
5898     \tl_set:Nn #2 { \q_no_value }
5899     \else:
5900     \exp_after:wN \__clist_get:wN #1 , \q_stop #2
5901     \fi:
5902 }
5903 \cs_new_protected:Npn \__clist_get:wN #1 , #2 \q_stop #3
5904 { \tl_set:Nn #3 {#1} }
5905 \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
\clist_pop:cn
\clist_gpop:NN
\clist_gpop:cn
\__clist_pop:NNN
\__clist_pop:wwNNN
\__clist_pop:wN

```

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.

```

5906 \cs_new_protected_nopar:Npn \clist_pop:NN
5907 { \__clist_pop:NNN \tl_set:Nx }
5908 \cs_new_protected_nopar:Npn \clist_gpop:NN
5909 { \__clist_pop:NNN \tl_gset:Nx }
5910 \cs_new_protected:Npn \__clist_pop:NNN #1#2#3
5911 {
5912   \if_meaning:w #2 \c_empty_clist
5913     \tl_set:Nn #3 { \q_no_value }
5914   \else:
5915     \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
5916   \fi:
5917 }
5918 \cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \q_stop #3#4#5
5919 {
5920   \tl_set:Nn #5 {#1}
5921   #3 #4
5922   {
5923     \__clist_pop:wN \prg_do_nothing:
5924     #2 \exp_not:o
5925     , \q_mark \use_none:n
5926     \q_stop
5927   }
5928 }
5929 \cs_new:Npn \__clist_pop:wN #1 , \q_mark #2 #3 \q_stop { #2 {#1} }
5930 \cs_generate_variant:Nn \clist_pop:NN { c }
5931 \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:NNTF The same, as branching code: very similar to the above.
\clist_get:cNTF 5932 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
\clist_pop:NNTF 5933 {
\clist_pop:cNTF 5934   \if_meaning:w #1 \c_empty_clist
\clist_gpop:NNTF 5935   \prg_return_false:
\clist_gpop:cNTF 5936   \else:
\__clist_pop_TF:NNN 5937   \exp_after:wN \__clist_get:wN #1 , \q_stop #2
5938   \prg_return_true:
5939   \fi:
5940 }
5941 \cs_generate_variant:Nn \clist_get:NN { c }
5942 \cs_generate_variant:Nn \clist_get:NNF { c }
5943 \cs_generate_variant:Nn \clist_get:NNTF { c }
5944 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
5945 { \__clist_pop_TF:NNN \tl_set:Nx #1 #2 }
5946 \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
5947 { \__clist_pop_TF:NNN \tl_gset:Nx #1 #2 }
5948 \cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
5949 {

```

```

5950 \if_meaning:w #2 \c_empty_clist
5951 \prg_return_false:
5952 \else:
5953 \exp_after:wN \_clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
5954 \prg_return_true:
5955 \fi:
5956 }
5957 \cs_generate_variant:Nn \clist_pop:NNT { c }
5958 \cs_generate_variant:Nn \clist_pop:NNF { c }
5959 \cs_generate_variant:Nn \clist_pop:NNTF { c }
5960 \cs_generate_variant:Nn \clist_gpop:NNT { c }
5961 \cs_generate_variant:Nn \clist_gpop:NNF { c }
5962 \cs_generate_variant:Nn \clist_gpop:NNTF { c }

```

(End definition for \clist_get:NNTF and \clist_get:CNTF. 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 5963 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
\clist_push:No 5964 \cs_new_eq:NN \clist_push:NV \clist_put_left:NV
\clist_push:Nx 5965 \cs_new_eq:NN \clist_push:No \clist_put_left:No
\clist_push:cn 5966 \cs_new_eq:NN \clist_push:Nx \clist_put_left:Nx
\clist_push:cV 5967 \cs_new_eq:NN \clist_push:cn \clist_put_left:cn
\clist_push:co 5968 \cs_new_eq:NN \clist_push:cV \clist_put_left:cV
\clist_push:cx 5969 \cs_new_eq:NN \clist_push:co \clist_put_left:co
\clist_gpush:Nn 5970 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
\clist_gpush:NV 5971 \cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
\clist_gpush:No 5972 \cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
\clist_gpush:Nx 5973 \cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx
\clist_gpush:cn 5974 \cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn
\clist_gpush:cV 5975 \cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV
\clist_gpush:co 5976 \cs_new_eq:NN \clist_gpush:co \clist_gput_left:co
\clist_gpush:cx 5977 \cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx
5978 \cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx

```

(End definition for \clist_push:Nn and others. These functions are documented on page ??.)

13.5 Modifying comma lists

\l_clist_internal_remove_clist An internal comma list for the removal routines.

```

5979 \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 5980 \cs_new_protected:Npn \clist_remove_duplicates:N
\clist_gremove_duplicates:N 5981 { \_clist_remove_duplicates:NN \clist_set_eq:NN }
\clist_gremove_duplicates:c 5982 \cs_new_protected:Npn \clist_gremove_duplicates:N
\_clist_remove_duplicates:NN 5983 { \_clist_remove_duplicates:NN \clist_gset_eq:NN }
5984 \cs_new_protected:Npn \_clist_remove_duplicates:NN #1#2
5985 {
5986 \clist_clear:N \l_clist_internal_remove_clist

```

```

5987 \clist_map_inline:Nn #2
5988 {
5989     \clist_if_in:NnF \l__clist_internal_remove_clist {##1}
5990     { \clist_put_right:Nn \l__clist_internal_remove_clist {##1} }
5991 }
5992 #1 #2 \l__clist_internal_remove_clist
5993 }
5994 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }
5995 \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 delimited by the $\langle item \rangle$ that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the $\langle item \rangle$. The loop is controlled by the argument grabbed by `__clist_remove_all:w`: when the item was found, the `\q_mark` delimiter used is the one inserted by `__clist_tmp:w`, and `\use_none_delimit_by_q_stop:w` is deleted. At the end, the final $\langle item \rangle$ is grabbed, and the argument of `__clist_tmp:w` contains `\q_mark`: in that case, `__clist_remove_all:w` removes the second `\q_mark` (inserted by `__clist_tmp:w`), and lets `\use_none_delimit_by_q_stop:w` act.

No brace is lost because items are always grabbed with a leading comma. The result of the first assignment has an extra leading comma, which we remove in a second assignment. Two exceptions: if the clist lost all of its elements, the result is empty, and we shouldn't remove anything; if the clist started up empty, the first step happens to turn it into a single comma, and the second step removes it.

```

5996 \cs_new_protected:Npn \clist_remove_all:Nn
5997 { \__clist_remove_all:NNn \tl_set:Nx }
5998 \cs_new_protected:Npn \clist_gremove_all:Nn
5999 { \__clist_remove_all:NNn \tl_gset:Nx }
6000 \cs_new_protected:Npn \__clist_remove_all:NNn #1#2#3
6001 {
6002     \cs_set:Npn \__clist_tmp:w ##1 , #3 ,
6003     {
6004         ##1
6005         , \q_mark , \use_none_delimit_by_q_stop:w ,
6006         \__clist_remove_all:
6007     }
6008     #1 #2
6009     {
6010         \exp_after:wN \__clist_remove_all:
6011         #2 , \q_mark , #3 , \q_stop
6012     }
6013     \clist_if_empty:NF #2
6014     {
6015         #1 #2
6016         {
6017             \exp_args:No \exp_not:o

```



```

6018             { \exp_after:wN \use_none:n #2 }
6019         }
6020     }
6021 }
6022 \cs_new:Npn \__clist_remove_all:
6023 { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
6024 \cs_new:Npn \__clist_remove_all:w #1 , \q_mark , #2 , { \exp_not:n {#1} }
6025 \cs_generate_variant:Nn \clist_remove_all:Nn { c }
6026 \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 ??.)

`\clist_reverse:N` Use `\clist_reverse:n` in an x-expanding assignment. The extra work that `\clist_reverse:n` does to preserve braces and spaces would not be needed for the well-controlled case of N-type comma lists, but the slow-down is not too bad.

```

\clist_reverse:c
\clist_greverse:N
\clist_greverse:c
6027 \cs_new_protected:Npn \clist_reverse:N #1
6028 { \tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
6029 \cs_new_protected:Npn \clist_greverse:N #1
6030 { \tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
6031 \cs_generate_variant:Nn \clist_reverse:N { c }
6032 \cs_generate_variant:Nn \clist_greverse:N { c }

```

(End definition for `\clist_reverse:N` and others. These functions are documented on page ??.)

`\clist_reverse:n` The reversed token list is built one item at a time, and stored between `\q_stop` and `\q_mark`, in the form of ? followed by zero or more instances of “`\langle item \rangle`,”. We start from a comma list “`\langle item_1 \rangle, \dots, \langle item_n \rangle`”. During the loop, the auxiliary `__clist_reverse:wwNww` receives “`? \langle item_i \rangle`” as #1, “`\langle item_{i+1} \rangle, \dots, \langle item_n \rangle`” as #2, `__clist_reverse:wwNww` as #3, what remains until `\q_stop` as #4, and “`\langle item_{i-1} \rangle, \dots, \langle item_1 \rangle`,” as #5. The auxiliary moves #1 just before #5, with a comma, and calls itself (#3). After the last item is moved, `__clist_reverse:wwNww` receives “`\q_mark __clist_reverse:wwNww !`” as its argument #1, thus `__clist_reverse_end:ww` as its argument #3. This second auxiliary cleans up until the marker !, removes the trailing comma (introduced when the first item was moved after `\q_stop`), and leaves its argument #1 within `\exp_not:n`. There is also a need to remove a leading comma, hence `\exp_not:o` and `\use_none:n`.

```

6033 \cs_new:Npn \clist_reverse:n #1
6034 {
6035     \__clist_reverse:wwNww ? #1 ,
6036     \q_mark \__clist_reverse:wwNww ! ,
6037     \q_mark \__clist_reverse_end:ww
6038     \q_stop ? \q_mark
6039 }
6040 \cs_new:Npn \__clist_reverse:wwNww
6041 #1 , #2 \q_mark #3 #4 \q_stop ? #5 \q_mark
6042 { #3 ? #2 \q_mark #3 #4 \q_stop #1 , #5 \q_mark }
6043 \cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \q_mark
6044 { \exp_not:o { \use_none:n #2 } }

```

(End definition for `\clist_reverse:n`. This function is documented on page 120.)

13.6 Comma list conditionals

`\clist_if_empty_p:N` Simple copies from the token list variable material.
`\clist_if_empty_p:c` 6045 `\prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N { p , T , F , TF }`
`\clist_if_empty:NTF` 6046 `\prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c { p , T , F , TF }`
`\clist_if_empty:cTF` (End definition for `\clist_if_empty:N` and `\clist_if_empty:c`. 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.

```
6047 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
6048 {
6049   \__clist_if_empty_n:w ? #1
6050   , \q_mark \prg_return_false:
6051   , \q_mark \prg_return_true:
6052   \q_stop
6053 }
6054 \cs_new:Npn \__clist_if_empty_n:w #1 ,
6055 {
6056   \tl_if_empty:oTF { \use_none:nn #1 ? }
6057   { \__clist_if_empty_n:w ? }
6058   { \__clist_if_empty_n:wNw }
6059 }
6060 \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 120.)
```

`\clist_if_in:NnTF` See description of the `\tl_if_in:Nn` function for details. We simply surround the comma
`\clist_if_in:NvTF` list, and the item, with commas.
`\clist_if_in:NoTF` 6061 `\prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }`
`\clist_if_in:cnTF` 6062 {
`\clist_if_in:cVTF` 6063 `\exp_args:No __clist_if_in_return:nn #1 {#2}`
`\clist_if_in:coTF` 6064 }
`\clist_if_in:nnTF` 6065 `\prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T , F , TF }`
`\clist_if_in:nvTF` 6066 {
`\clist_if_in:noTF` 6067 `\clist_set:Nn \l__clist_internal_clist {#1}`
`__clist_if_in_return:nn` 6068 `\exp_args:No __clist_if_in_return:nn \l__clist_internal_clist {#2}`
6069 }
6070 `\cs_new_protected:Npn __clist_if_in_return:nn #1#2`
6071 {
6072 `\cs_set:Npn __clist_tmp:w ##1 ,#2, { }`
6073 `\tl_if_empty:oTF`
6074 `{ __clist_tmp:w ,#1, {} {} ,#2, }`
6075 `{ \prg_return_false: } { \prg_return_true: }`
6076 }

```

6077 \cs_generate_variant:Nn \clist_if_in:NnT { NV , No }
6078 \cs_generate_variant:Nn \clist_if_in:NnT { c , cV , co }
6079 \cs_generate_variant:Nn \clist_if_in:NnF { NV , No }
6080 \cs_generate_variant:Nn \clist_if_in:NnF { c , cV , co }
6081 \cs_generate_variant:Nn \clist_if_in:NnTF { NV , No }
6082 \cs_generate_variant:Nn \clist_if_in:NnTF { c , cV , co }
6083 \cs_generate_variant:Nn \clist_if_in:nnT { nV , no }
6084 \cs_generate_variant:Nn \clist_if_in:nnF { nV , no }
6085 \cs_generate_variant:Nn \clist_if_in:nnTF { nV , no }

```

(End definition for \clist_if_in:NnTF and others. These functions are documented on page ??.)

13.7 Mapping to comma lists

\clist_map_function:NN If the variable is empty, the mapping is skipped (otherwise, that comma-list would be 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.

```

6086 \cs_new:Npn \clist_map_function:NN #1#2
6087 {
6088   \clist_if_empty:NF #1
6089   {
6090     \exp_last_unbraced:NNo \__clist_map_function:Nw #2 #1
6091     , \q_recursion_tail ,
6092     \__prg_break_point:Nn \clist_map_break: { }
6093   }
6094 }
6095 \cs_new:Npn \__clist_map_function:Nw #1#2 ,
6096 {
6097   \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
6098   #1 {#2}
6099   \__clist_map_function:Nw #1
6100 }
6101 \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. **__clist_map_function_n:Nn** 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.

```

6102 \cs_new:Npn \clist_map_function:nN #1#2
6103 {
6104   \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #2 }
6105   \q_mark #1, \q_recursion_tail,
6106   \__prg_break_point:Nn \clist_map_break: { }

```

```

6107 }
6108 \cs_new:Npn \__clist_map_function_n:Nn #1 #2
6109 {
6110   \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
6111   \tl_if_empty:nF {#2} { \__clist_map_unbrace:Nw #1 #2, }
6112   \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #1 }
6113   \q_mark
6114 }
6115 \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 TeX’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.

```

6116 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
6117 {
6118   \clist_if_empty:NF #1
6119   {
6120     \int_gincr:N \g__prg_map_int
6121     \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
6122     \exp_last_unbraced:Nco \__clist_map_function:Nw
6123     { __prg_map_ \int_use:N \g__prg_map_int :w }
6124     #1 , \q_recursion_tail ,
6125     \__prg_break_point:Nn \clist_map_break:
6126     { \int_gdecr:N \g__prg_map_int }
6127   }
6128 }
6129 \cs_new_protected:Npn \clist_map_inline:nn #1
6130 {
6131   \clist_set:Nn \l__clist_internal_clist {#1}
6132   \clist_map_inline:Nn \l__clist_internal_clist
6133 }
6134 \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.

```

6135 \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
6136 {
6137   \clist_if_empty:NF #1
6138   {
6139     \exp_args:Nno \use:nn
6140     { \__clist_map_variable:Nnw #2 {#3} }

```

```

6141         #1
6142         , \q_recursion_tail , \q_recursion_stop
6143         \__prg_break_point:Nn \clist_map_break: { }
6144     }
6145 }
6146 \cs_new_protected:Npn \clist_map_variable:nNn #1
6147 {
6148     \clist_set:Nn \l__clist_internal_clist {#1}
6149     \clist_map_variable:NNn \l__clist_internal_clist
6150 }
6151 \cs_new_protected:Npn \__clist_map_variable:Nnw #1#2#3,
6152 {
6153     \tl_set:Nn #1 {#3}
6154     \quark_if_recursion_tail_stop:N #1
6155     \use:n {#2}
6156     \__clist_map_variable:Nnw #1 {#2}
6157 }
6158 \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

```

6159 \cs_new_nopar:Npn \clist_map_break:
6160 { \__prg_map_break:Nn \clist_map_break: { } }
6161 \cs_new_nopar:Npn \clist_map_break:n
6162 { \__prg_map_break:Nn \clist_map_break: }

```

(End definition for \clist_map_break: and \clist_map_break:n. These functions are documented on page 122.)

\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).

```

6163 \cs_new:Npn \clist_count:N #1
6164 {
6165     \int_eval:n
6166     {
6167         0
6168         \clist_map_function:NN #1 \__clist_count:n
6169     }
6170 }
6171 \cs_generate_variant:Nn \clist_count:N { c }
6172 \cs_new:Npx \clist_count:n #1
6173 {
6174     \exp_not:N \int_eval:n
6175     {
6176         0
6177         \exp_not:N \__clist_count:w \c_space_tl

```

```

6178         #1 \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
6179     }
6180 }
6181 \cs_new:Npn \__clist_count:n #1 { + \c_one }
6182 \cs_new:Npx \__clist_count:w #1 ,
6183 {
6184     \exp_not:n { \exp_args:Nf \quark_if_recursion_tail_stop:n } {#1}
6185     \exp_not:N \tl_if_blank:nF {#1} { + \c_one }
6186     \exp_not:N \__clist_count:w \c_space_tl
6187 }

```

(End definition for `\clist_count:N`, `\clist_count:c`, and `\clist_count:n`. These functions are documented on page ??.)

13.8 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_ii`, switching to the other *continuation*, `use_iii`, which uses the *separator between final two*.

```

6188 \cs_new:Npn \clist_use:Nnnn #1#2#3#4
6189 {
6190     \clist_if_exist:NTF #1
6191     {
6192         \int_case:nnF { \clist_count:N #1 }
6193         {
6194             { 0 } { }
6195             { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
6196             { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
6197         }
6198         {
6199             \exp_after:wN \__clist_use:nwwwnwn
6200             \exp_after:wN { \exp_after:wN } #1 ,
6201             \q_mark , { \__clist_use:nwwwnwn {#3} }
6202             \q_mark , { \__clist_use:nwwn {#4} }
6203             \q_stop { }
6204         }
6205     }

```

```

6206      { \_msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#1} }
6207    }
6208    \cs_generate_variant:Nn \clist_use:Nnnn { c }
6209    \cs_new:Npn \_clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
6210    \cs_new:Npn \_clist_use:nwwwnwn
6211      #1#2 , #3 , #4 , #5 \q_mark , #6#7 \q_stop #8
6212      { #6 {#3} , {#4} , #5 \q_mark , {#6} #7 \q_stop { #8 #1 #2 } }
6213    \cs_new:Npn \_clist_use:nwwn #1#2 , #3 \q_stop #4
6214      { \exp_not:n { #4 #1 #2 } }
6215    \cs_new:Npn \clist_use:Nn #1#2
6216      { \clist_use:Nnnn #1 {#2} {#2} {#2} }
6217    \cs_generate_variant:Nn \clist_use:Nn { c }

```

(End definition for `\clist_use:Nnnn` and `\clist_use:cnnn`. These functions are documented on page ??.)

13.9 Using a single item

```

\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.

```

6218 \cs_new:Npn \clist_item:Nn #1#2
6219   {
6220     \exp_args:Nfo \_clist_item:nnNn
6221       { \clist_count:N #1 }
6222       #1
6223       \_clist_item_N_loop:nw
6224       {#2}
6225   }
6226 \cs_new:Npn \_clist_item:nnNn #1#2#3#4
6227   {
6228     \int_compare:nNnTF {#4} < \c_zero
6229     {
6230       \int_compare:nNnTF {#4} < { - #1 }
6231       { \use_none_delimit_by_q_stop:w }
6232       { \exp_args:Nf #3 { \int_eval:n { #4 + \c_one + #1 } } }
6233     }
6234     {
6235       \int_compare:nNnTF {#4} > {#1}
6236       { \use_none_delimit_by_q_stop:w }
6237       { #3 {#4} }
6238     }
6239     { } , #2 , \q_stop
6240   }
6241 \cs_new:Npn \_clist_item_N_loop:nw #1 #2,
6242   {
6243     \int_compare:nNnTF {#1} = \c_zero
6244     { \use_i_delimit_by_q_stop:nw { \exp_not:n {#2} } }

```

```

6245     { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
6246   }
6247   \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 This starts in the same way as \clist_item:Nn by counting the items of the comma list.
 __clist_item_n:nw The final item should be space-trimmed before being brace-stripped, hence we insert a
 __clist_item_n_loop:nw couple of odd-looking \prg_do_nothing: to avoid losing braces. Blank items are ignored.
 __clist_item_n_end:n
 __clist_item_n_strip:w

```

6248   \cs_new:Npn \clist_item:nn #1#2
6249   {
6250     \exp_args:Nf \__clist_item:nnNn
6251     { \clist_count:n {#1} }
6252     {#1}
6253     \__clist_item_n:nw
6254     {#2}
6255   }
6256   \cs_new:Npn \__clist_item_n:nw #1
6257   { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
6258   \cs_new:Npn \__clist_item_n_loop:nw #1 #2,
6259   {
6260     \exp_args:No \tl_if_blank:nTF {#2}
6261     { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
6262     {
6263       \int_compare:nNnTF {#1} = \c_zero
6264       { \exp_args:No \__clist_item_n_end:n {#2} }
6265       {
6266         \exp_args:Nf \__clist_item_n_loop:nw
6267         { \int_eval:n { #1 - 1 } }
6268         \prg_do_nothing:
6269       }
6270     }
6271   }
6272   \cs_new:Npn \__clist_item_n_end:n #1 #2 \q_stop
6273   {
6274     \__tl_trim_spaces:nn { \q_mark #1 }
6275     { \exp_last_unbraced:No \__clist_item_n_strip:w } ,
6276   }
6277   \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 ??.)

13.10 Viewing comma lists

\clist_show:N Apply the general __msg_show_variable:Nnn. In the case of an n-type comma-list,
 \clist_show:c first store it in a scratch variable, then show that variable: The message takes care of
 \clist_show:n omitting its name.

```

6278   \cs_new_protected:Npn \clist_show:N #1
6279   {
6280     \__msg_show_variable:Nnn #1 { clist }

```



```

6281     { \clist_map_function:NN #1 \__msg_show_item:n }
6282   }
6283   \cs_new_protected:Npn \clist_show:n #1
6284   {
6285     \clist_set:Nn \l__clist_internal_clist {#1}
6286     \clist_show:N \l__clist_internal_clist
6287   }
6288   \cs_generate_variant:Nn \clist_show:N { c }

```

(End definition for `\clist_show:N` and `\clist_show:c`. These functions are documented on page 125.)

13.11 Scratch comma lists

Temporary comma list variables.

```

\l_tmpa_clist \clist_new:N \l_tmpa_clist
\l_tmpb_clist \clist_new:N \l_tmpb_clist
\g_tmpa_clist \clist_new:N \g_tmpa_clist
\g_tmpb_clist \clist_new:N \g_tmpb_clist

```

(End definition for `\l_tmpa_clist` and `\l_tmpb_clist`. These variables are documented on page 126.)

```

6293 \</initex | package>

```

14 l3prop implementation

The following test files are used for this code: `m3prop001`, `m3prop002`, `m3prop003`, `m3prop004`, `m3show001`.

```

6294 \<*initex | package>
6295 \<@@=prop>

```

A property list is a macro whose top-level expansion is for the form

```

\s__prop \__prop_pair:wn <key_1> \s__prop {<value_1>}
...
\__prop_pair:wn <key_n> \s__prop {<value_n>}

```

where `\s__prop` is a scan mark (equal to `\scan_stop:`), and `__prop_pair:wn` can be used to map through the property list.

`\s__prop` A private scan mark is used as a marker after each key, and at the very beginning of the property list.

```

6296 \__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.

```

6297 \cs_new:Npn \__prop_pair:wn #1 \s__prop #2
6298 { \__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.

```
6299 \tl_new:N \l__prop_internal_tl
```

(End definition for `\l__prop_internal_tl`. This variable is documented on page 132.)

`\c_empty_prop` An empty prop.

```
6300 \tl_const:Nn \c_empty_prop { \s__prop }
```

(End definition for `\c_empty_prop`. This variable is documented on page 132.)

14.1 Allocation and initialisation

`\prop_new:N` Property lists are initialized with the value `\c_empty_prop`.

`\prop_new:c`

```
6301 \cs_new_protected:Npn \prop_new:N #1
6302 {
6303   \__chk_if_free_cs:N #1
6304   \cs_gset_eq:NN #1 \c_empty_prop
6305 }
6306 \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`

`\prop_gclear:N`

`\prop_gclear:c`

```
6307 \cs_new_protected:Npn \prop_clear:N #1
6308 { \prop_set_eq:NN #1 \c_empty_prop }
6309 \cs_generate_variant:Nn \prop_clear:N { c }
6310 \cs_new_protected:Npn \prop_gclear:N #1
6311 { \prop_gset_eq:NN #1 \c_empty_prop }
6312 \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`

`\prop_gclear_new:N`

`\prop_gclear_new:c`

```
6313 \cs_new_protected:Npn \prop_clear_new:N #1
6314 { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
6315 \cs_generate_variant:Nn \prop_clear_new:N { c }
6316 \cs_new_protected:Npn \prop_gclear_new:N #1
6317 { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
6318 \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`

`\prop_set_eq:Nc`

`\prop_set_eq:cc`

`\prop_gset_eq:NN`

`\prop_gset_eq:cN`

`\prop_gset_eq:Nc`

`\prop_gset_eq:cc`

```
6319 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
6320 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
6321 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
6322 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
6323 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
6324 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
6325 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
6326 \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 6327 \prop_new:N \l_tmpa_prop
\g_tmpa_prop 6328 \prop_new:N \l_tmpb_prop
\g_tmpb_prop 6329 \prop_new:N \g_tmpa_prop
6330 \prop_new:N \g_tmpb_prop

```

(End definition for `\l_tmpa_prop` and `\l_tmpb_prop`. These variables are documented on page 132.)

14.2 Accessing data in property lists

```

\__prop_split:NnTF
\__prop_split_aux:NnTF
\__prop_split_aux:w

```

This function is used by most of the module, and hence must be fast. It receives a $\langle\textit{property list}\rangle$, a $\langle\textit{key}\rangle$, a $\langle\textit{true code}\rangle$ and a $\langle\textit{false code}\rangle$. The aim is to split the $\langle\textit{property list}\rangle$ 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 \langle\textit{key}\rangle \s__prop #2
#3 \q_mark #4 #5 \q_stop
{ #4 {\langle\textit{true code}\rangle} {\langle\textit{false code}\rangle} }

```

If the $\langle\textit{key}\rangle$ is present in the property list, `__prop_split_aux:w`'s #1 is the part before the $\langle\textit{key}\rangle$, #2 is the $\langle\textit{value}\rangle$, #3 is the part after the $\langle\textit{key}\rangle$, #4 is `\use_i:nn`, and #5 is additional tokens that we do not care about. The $\langle\textit{true code}\rangle$ 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 `__prop_pair:wn \langle\textit{key}\rangle \s__prop {#2} #3`.

If the $\langle\textit{key}\rangle$ is not there, then the $\langle\textit{function}\rangle$ is `\use_ii:nn`, which keeps the $\langle\textit{false code}\rangle$.

```

6331 \cs_new_protected:Npn \__prop_split:NnTF #1#2
6332 { \exp_args:NNo \__prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
6333 \cs_new_protected:Npn \__prop_split_aux:NnTF #1#2#3#4
6334 {
6335   \cs_set:Npn \__prop_split_aux:w ##1
6336   \__prop_pair:wn #2 \s__prop ##2 ##3 \q_mark ##4 ##5 \q_stop
6337   { ##4 {#3} {#4} }
6338   \exp_after:wN \__prop_split_aux:w #1 \q_mark \use_i:nn
6339   \__prop_pair:wn #2 \s__prop { } \q_mark \use_ii:nn \q_stop
6340 }
6341 \cs_new:Npn \__prop_split_aux:w { }

```

(End definition for `__prop_split:NnTF`.)

```

\prop_remove:Nn \prop_remove:NV \prop_remove:cn \prop_remove:cV
\prop_gremove:Nn \prop_gremove:NV \prop_gremove:cn \prop_gremove:cV

```

Deleting from a property starts by splitting the list. If the key is present in the property list, the returned value is ignored. If the key is missing, nothing happens.

```

6342 \cs_new_protected:Npn \prop_remove:Nn #1#2
6343 {
6344   \__prop_split:NnTF #1 {#2}

```

```

6345     { \tl_set:Nn #1 { ##1 ##3 } }
6346     { }
6347   }
6348   \cs_new_protected:Npn \prop_gremove:Nn #1#2
6349   {
6350     \__prop_split:NnTF #1 {#2}
6351     { \tl_gset:Nn #1 { ##1 ##3 } }
6352     { }
6353   }
6354   \cs_generate_variant:Nn \prop_remove:Nn { NV }
6355   \cs_generate_variant:Nn \prop_remove:Nn { c , cV }
6356   \cs_generate_variant:Nn \prop_gremove:Nn { NV }
6357   \cs_generate_variant:Nn \prop_gremove:Nn { c , cV }

```

(End definition for \prop_remove:Nn and others. These functions are documented on page ??.)

\prop_get:NnN Getting an item from a list is very easy: after splitting, if the key is in the property list, just set the token list variable to the return value, otherwise to \q_no_value.

```

\prop_get:NVN
\prop_get:NoN
\prop_get:cnN
\prop_get:cVN
\prop_get:coN
6358   \cs_new_protected:Npn \prop_get:NnN #1#2#3
6359   {
6360     \__prop_split:NnTF #1 {#2}
6361     { \tl_set:Nn #3 {##2} }
6362     { \tl_set:Nn #3 { \q_no_value } }
6363   }
6364   \cs_generate_variant:Nn \prop_get:NnN { NV , No }
6365   \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
6366   \cs_new_protected:Npn \prop_pop:NnN #1#2#3
6367   {
6368     \__prop_split:NnTF #1 {#2}
6369     {
6370       \tl_set:Nn #3 {##2}
6371       \tl_set:Nn #1 { ##1 ##3 }
6372     }
6373     { \tl_set:Nn #3 { \q_no_value } }
6374   }
6375   \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
6376   {
6377     \__prop_split:NnTF #1 {#2}
6378     {
6379       \tl_set:Nn #3 {##2}
6380       \tl_gset:Nn #1 { ##1 ##3 }
6381     }
6382     { \tl_set:Nn #3 { \q_no_value } }
6383   }
6384   \cs_generate_variant:Nn \prop_pop:NnN { No }

```

```

6385 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
6386 \cs_generate_variant:Nn \prop_gpop:NnN { No }
6387 \cs_generate_variant:Nn \prop_gpop:NnN { c , co }

```

(End definition for \prop_pop:NnN and others. These functions are documented on page ??.)

\prop_item:Nn
\prop_item:cn
__prop_item:Nn:nwn

Getting the value corresponding to a key in a property list in an expandable fashion is similar to mapping some tokens. Go through the property list one $\langle key \rangle$ – $\langle value \rangle$ pair at a time: the arguments of **__prop_item:Nn:nwn** are the $\langle key \rangle$ we are looking for, a $\langle key \rangle$ of the property list, and its associated value. The $\langle keys \rangle$ are compared (as strings). If they match, the $\langle value \rangle$ is returned, within **\exp_not:n**. The loop terminates even if the $\langle key \rangle$ is missing, and yields an empty value, because we have appended the appropriate $\langle key \rangle$ – $\langle empty\ value \rangle$ pair to the property list.

```

6388 \cs_new:Npn \prop_item:Nn #1#2
6389 {
6390   \exp_last_unbraced:Noo \__prop_item:Nn:nwn { \tl_to_str:n {#2} } #1
6391   \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
6392   \__prg_break_point:
6393 }
6394 \cs_new:Npn \__prop_item:Nn:nwn #1#2 \__prop_pair:wn #3 \s__prop #4
6395 {
6396   \str_if_eq_x:nnTF {#1} {#3}
6397   { \__prg_break:n { \exp_not:n {#4} } }
6398   { \__prop_item:Nn:nwn {#1} }
6399 }
6400 \cs_generate_variant:Nn \prop_item:Nn { c }

```

(End definition for \prop_item:Nn and \prop_item:cn. These functions are documented on page ??.)

\prop_pop:NnTF
\prop_pop:cnTF
\prop_gpop:NnTF
\prop_gpop:cnTF

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.

```

6401 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
6402 {
6403   \__prop_split:NnTF #1 {#2}
6404   {
6405     \tl_set:Nn #3 {##2}
6406     \tl_set:Nn #1 { ##1 ##3 }
6407     \prg_return_true:
6408   }
6409   { \prg_return_false: }
6410 }
6411 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
6412 {
6413   \__prop_split:NnTF #1 {#2}
6414   {
6415     \tl_set:Nn #3 {##2}
6416     \tl_gset:Nn #1 { ##1 ##3 }
6417     \prg_return_true:
6418   }

```

```

6419     { \prg_return_false: }
6420   }
6421   \cs_generate_variant:Nn \prop_pop:NnNT { c }
6422   \cs_generate_variant:Nn \prop_pop:NnNF { c }
6423   \cs_generate_variant:Nn \prop_pop:NnTF { c }
6424   \cs_generate_variant:Nn \prop_gpop:NnNT { c }
6425   \cs_generate_variant:Nn \prop_gpop:NnNF { c }
6426   \cs_generate_variant:Nn \prop_gpop:NnTF { c }

```

(End definition for \prop_pop:NnTF 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 $\langle key \rangle$ and new $\langle value \rangle$ may contain arbitrary tokens, it is not safe to include them in the argument of __prop_split:NnTF. We thus start by storing in \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 $\langle key \rangle$ was absent, append the new key–value to the list. Otherwise concatenate the extracts ##1 and ##3 with the new key–value pair \l__prop_internal_tl. The updated entry is placed at the same spot as the original $\langle key \rangle$ in the property list, preserving the order of entries.

```

6427   \cs_new_protected_nopar:Npn \prop_put:Nnn { \__prop_put:NNnn \tl_set:Nx }
6428   \cs_new_protected_nopar:Npn \prop_gput:Nnn { \__prop_put:NNnn \tl_gset:Nx }
6429   \cs_new_protected:Npn \__prop_put:NNnn #1#2#3#4
6430   {
6431     \tl_set:Nn \l__prop_internal_tl
6432     {
6433       \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
6434       \s_prop { \exp_not:n {#4} }
6435     }
6436     \__prop_split:NnTF #2 {#3}
6437     { #1 #2 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } }
6438     { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
6439   }
6440   \cs_generate_variant:Nn \prop_put:Nnn
6441   { NnV , Nno , Nnx , NV , NVV , No , Noo }
6442   \cs_generate_variant:Nn \prop_put:Nnn
6443   { c , cnV , cno , cnx , cV , cVV , co , coo }
6444   \cs_generate_variant:Nn \prop_gput:Nnn
6445   { NnV , Nno , Nnx , NV , NVV , No , Noo }
6446   \cs_generate_variant:Nn \prop_gput:Nnn
6447   { 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 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.

```

6448   \cs_new_protected_nopar:Npn \prop_put_if_new:Nnn
6449   { \__prop_put_if_new:NNnn \tl_set:Nx }
6450   \cs_new_protected_nopar:Npn \prop_gput_if_new:Nnn
6451   { \__prop_put_if_new:NNnn \tl_gset:Nx }

```

```

6452 \cs_new_protected:Npn \__prop_put_if_new:Nnn #1#2#3#4
6453 {
6454   \tl_set:Nn \l__prop_internal_tl
6455   {
6456     \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
6457     \s_prop \exp_not:n { {#4} }
6458   }
6459   \__prop_split:NnTF #2 {#3}
6460   { }
6461   { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
6462 }
6463 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }
6464 \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 ??.)

14.3 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`.

```

6465 \prg_new_eq_conditional:Nnn \prop_if_exist:N \cs_if_exist:N { TF , T , F , p }
6466 \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.

```

6467 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
6468 {
6469   \tl_if_eq:NNTF #1 \c_empty_prop
6470   \prg_return_true: \prg_return_false:
6471 }
6472 \cs_generate_variant:Nn \prop_if_empty_p:N { c }
6473 \cs_generate_variant:Nn \prop_if_empty:NT { c }
6474 \cs_generate_variant:Nn \prop_if_empty:NF { c }
6475 \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`

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: }
}

```

`__prop_if_in:nwwn`
`__prop_if_in:N`

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:n` receives `__prop_pair:wn`, and if it is found as the extra item, the function receives `\q_recursion_tail`, easily recognizable.

Here, `\prop_map_function:NN` is not sufficient for the mapping, since it can only map a single token, and cannot carry the key that is searched for.

```

6476 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
6477 {
6478   \exp_last_unbraced:Noo \__prop_if_in:nwwn { \tl_to_str:n {#2} } #1
6479   \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
6480   \q_recursion_tail
6481   \__prg_break_point:
6482 }
6483 \cs_new:Npn \__prop_if_in:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
6484 {
6485   \str_if_eq_x:nnTF {#1} {#3}
6486   { \__prop_if_in:N }
6487   { \__prop_if_in:nwwn {#1} }
6488 }
6489 \cs_new:Npn \__prop_if_in:N #1
6490 {
6491   \if_meaning:w \q_recursion_tail #1
6492   \prg_return_false:
6493   \else:
6494     \prg_return_true:
6495   \fi:
6496   \__prg_break:
6497 }
6498 \cs_generate_variant:Nn \prop_if_in_p:Nn { NV , No }
6499 \cs_generate_variant:Nn \prop_if_in_p:Nn { c , cV , co }
6500 \cs_generate_variant:Nn \prop_if_in:NnT { NV , No }
6501 \cs_generate_variant:Nn \prop_if_in:NnT { c , cV , co }
6502 \cs_generate_variant:Nn \prop_if_in:NnF { NV , No }
6503 \cs_generate_variant:Nn \prop_if_in:NnF { c , cV , co }
6504 \cs_generate_variant:Nn \prop_if_in:NnTF { NV , No }
6505 \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 ??.)

14.4 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:NVNTF`

`\prop_get:NoNTF`

`\prop_get:cnNTF`

`\prop_get:cVNTF`

`\prop_get:coNTF`


```

6506 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
6507 {
6508   \__prop_split:NnTF #1 {#2}
6509   {
6510     \tl_set:Nn #3 {##2}
6511     \prg_return_true:
6512   }
6513   { \prg_return_false: }
6514 }
6515 \cs_generate_variant:Nn \prop_get:NnNT { NV , No }
6516 \cs_generate_variant:Nn \prop_get:NnNF { NV , No }
6517 \cs_generate_variant:Nn \prop_get:NnNTF { NV , No }
6518 \cs_generate_variant:Nn \prop_get:NnNT { c , cV , co }
6519 \cs_generate_variant:Nn \prop_get:NnNF { c , cV , co }
6520 \cs_generate_variant:Nn \prop_get:NnNTF { c , cV , co }

```

(End definition for `\prop_get:NnNTF` and others. These functions are documented on page ??.)

14.5 Mapping to property lists

`\prop_map_function:NN`
`\prop_map_function:Nc`
`\prop_map_function:cN`
`\prop_map_function:cc`
`__prop_map_function:Nwnn`

The fastest way to do a recursion here is to use an `\if_meaning:w` test: the keys are strings, and thus cannot match the marker `\q_recursion_tail`. A special case to note is when the key `#3` is empty: then `\q_recursion_tail` is compared to `\exp_after:wN`, also different. Note that `#2` is empty, except at the first iteration, where it is `\s__prop`.

```

6521 \cs_new:Npn \prop_map_function:NN #1#2
6522 {
6523   \exp_last_unbraced:NNo \__prop_map_function:Nwnn #2 #1
6524   \__prop_pair:wn \q_recursion_tail \s__prop { }
6525   \__prg_break_point:Nn \prop_map_break: { }
6526 }
6527 \cs_new:Npn \__prop_map_function:Nwnn #1#2 \__prop_pair:wn #3 \s__prop #4
6528 {
6529   \if_meaning:w \q_recursion_tail #3
6530   \exp_after:wN \prop_map_break:
6531   \fi:
6532   #1 {#3} {#4}
6533   \__prop_map_function:Nwnn #1
6534 }
6535 \cs_generate_variant:Nn \prop_map_function:NN { Nc }
6536 \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`
`\prop_map_inline:cn`

Mapping in line requires a nesting level counter. Store the current definition of `__prop_pair:wn`, and define it anew. At the end of the loop, revert to the earlier definition. Note that besides pairs of the form `__prop_pair:wn <key> \s__prop {<value>}`, there are a leading and a trailing tokens, but both are equal to `\scan_stop:`, hence have no effect in such inline mapping.

```

6537 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
6538 {

```

```

6539 \cs_gset_eq:cn
6540 { __prg_map_ \int_use:N \g__prg_map_int :wn } \__prop_pair:wn
6541 \int_gincr:N \g__prg_map_int
6542 \cs_gset:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
6543 #1
6544 \__prg_break_point:Nn \prop_map_break:
6545 {
6546   \int_gdecr:N \g__prg_map_int
6547   \cs_gset_eq:Nc \__prop_pair:wn
6548   { __prg_map_ \int_use:N \g__prg_map_int :wn }
6549 }
6550 }
6551 \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`

```

6552 \cs_new_nopar:Npn \prop_map_break:
6553 { \__prg_map_break:Nn \prop_map_break: { } }
6554 \cs_new_nopar:Npn \prop_map_break:n
6555 { \__prg_map_break:Nn \prop_map_break: }

```

(End definition for `\prop_map_break:.` This function is documented on page 131.)

14.6 Viewing property lists

`\prop_show:N` Apply the general `__msg_show_variable:Nnn`. Contrarily to sequences and comma
`\prop_show:c` lists, we use `__msg_show_item:nn` to format both the key and the value for each pair.

```

6556 \cs_new_protected:Npn \prop_show:N #1
6557 {
6558   \__msg_show_variable:Nnn #1 { prop }
6559   { \prop_map_function:NN #1 \__msg_show_item:nn }
6560 }
6561 \cs_generate_variant:Nn \prop_show:N { c }

```

(End definition for `\prop_show:N` and `\prop_show:c`. These functions are documented on page ??.)

14.7 Deprecated functions

`\prop_get:Nn` Deprecated 2014-07-17.

`\prop_get:cn`

```

6562 \cs_new_eq:NN \prop_get:Nn \prop_item:Nn
6563 \cs_new_eq:NN \prop_get:cn \prop_item:cn

```

(End definition for `\prop_get:Nn` and `\prop_get:cn`. These functions are documented on page ??.)

```

6564 </initex | package>

```

15 l3box implementation

```
6565 <*initex | package>
6566 <@@=box>
```

The code in this module is very straight forward so I'm not going to comment it very extensively.

15.1 Creating and initialising boxes

The following test files are used for this code: m3box001.lvt.

\box_new:N Defining a new $\langle box \rangle$ register: remember that box 255 is not generally available.

```
\box_new:c
6567 <*package>
6568 \cs_new_protected:Npn \box_new:N #1
6569 {
6570     \__chk_if_free_cs:N #1
6571     \cs:w newbox \cs_end: #1
6572 }
6573 </package>
6574 \cs_generate_variant:Nn \box_new:N { c }
```

\box_clear:N Clear a $\langle box \rangle$ register.

```
\box_clear:c
6575 \cs_new_protected:Npn \box_clear:N #1
\box_gclear:N
6576 { \box_set_eq:NN #1 \c_empty_box }
\box_gclear:c
6577 \cs_new_protected:Npn \box_gclear:N #1
6578 { \box_gset_eq:NN #1 \c_empty_box }
6579 \cs_generate_variant:Nn \box_clear:N { c }
6580 \cs_generate_variant:Nn \box_gclear:N { c }
```

\box_clear_new:N Clear or new.

```
\box_clear_new:c
6581 \cs_new_protected:Npn \box_clear_new:N #1
\box_gclear_new:N
6582 { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
\box_gclear_new:c
6583 \cs_new_protected:Npn \box_gclear_new:N #1
6584 { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
6585 \cs_generate_variant:Nn \box_clear_new:N { c }
6586 \cs_generate_variant:Nn \box_gclear_new:N { c }
```

\box_set_eq:NN Assigning the contents of a box to be another box.

```
\box_set_eq:cN
6587 \cs_new_protected:Npn \box_set_eq:NN #1#2
\box_set_eq:Nc
6588 { \tex_setbox:D #1 \tex_copy:D #2 }
\box_set_eq:cc
6589 \cs_new_protected:Npn \box_gset_eq:NN
\box_gset_eq:NN
6590 { \tex_global:D \box_set_eq:NN }
\box_gset_eq:cN
6591 \cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }
\box_gset_eq:Nc
6592 \cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }
```

\box_set_eq_clear:NN Assigning the contents of a box to be another box. This clears the second box globally (that's how \TeX does it).

```
\box_set_eq_clear:cN
6593 \cs_new_protected:Npn \box_set_eq_clear:NN #1#2
\box_set_eq_clear:Nc
6594 { \tex_setbox:D #1 \tex_box:D #2 }
```

\box_gset_eq_clear:NN

```
\box_gset_eq_clear:cN
\box_gset_eq_clear:Nc
\box_gset_eq_clear:cc
```

```

6595 \cs_new_protected:Npn \box_gset_eq_clear:NN
6596 { \tex_global:D \box_set_eq_clear:NN }
6597 \cs_generate_variant:Nn \box_set_eq_clear:NN { c , Nc , cc }
6598 \cs_generate_variant:Nn \box_gset_eq_clear:NN { c , Nc , cc }

```

`\box_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\box_if_exist_p:c 6599 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N { TF , T , F , p }
\box_if_exist:NTF 6600 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c { TF , T , F , p }
\box_if_exist:cTF

```

15.2 Measuring and setting box dimensions

`\box_ht:N` Accessing the height, depth, and width of a $\langle box \rangle$ register.

```

\box_ht:c 6601 \cs_new_eq:NN \box_ht:N \tex_ht:D
\box_dp:N 6602 \cs_new_eq:NN \box_dp:N \tex_dp:D
\box_dp:c 6603 \cs_new_eq:NN \box_wd:N \tex_wd:D
\box_wd:N 6604 \cs_generate_variant:Nn \box_ht:N { c }
\box_wd:c 6605 \cs_generate_variant:Nn \box_dp:N { c }
6606 \cs_generate_variant:Nn \box_wd:N { c }

```

`\box_set_ht:Nn` Measuring is easy: all primitive work. These primitives are not expandable, so the derived functions are not either.

```

\box_set_ht:cn 6607 \cs_new_protected:Npn \box_set_dp:Nn #1#2
\box_set_dp:cn 6608 { \box_dp:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\box_set_wd:Nn 6609 \cs_new_protected:Npn \box_set_ht:Nn #1#2
\box_set_wd:cn 6610 { \box_ht:N #1 \__dim_eval:w #2 \__dim_eval_end: }
6611 \cs_new_protected:Npn \box_set_wd:Nn #1#2
6612 { \box_wd:N #1 \__dim_eval:w #2 \__dim_eval_end: }
6613 \cs_generate_variant:Nn \box_set_ht:Nn { c }
6614 \cs_generate_variant:Nn \box_set_dp:Nn { c }
6615 \cs_generate_variant:Nn \box_set_wd:Nn { c }

```

15.3 Using boxes

`\box_use_clear:N` Using a $\langle box \rangle$. These are just \TeX primitives with meaningful names.

```

\box_use_clear:c 6616 \cs_new_eq:NN \box_use_clear:N \tex_box:D
\box_use:N 6617 \cs_new_eq:NN \box_use:N \tex_copy:D
\box_use:c 6618 \cs_generate_variant:Nn \box_use_clear:N { c }
6619 \cs_generate_variant:Nn \box_use:N { c }

```

`\box_move_left:nn` Move box material in different directions.

```

\box_move_right:nn 6620 \cs_new_protected:Npn \box_move_left:nn #1#2
\box_move_up:nn 6621 { \tex_moveleft:D \__dim_eval:w #1 \__dim_eval_end: #2 }
\box_move_down:nn 6622 \cs_new_protected:Npn \box_move_right:nn #1#2
6623 { \tex_moveright:D \__dim_eval:w #1 \__dim_eval_end: #2 }
6624 \cs_new_protected:Npn \box_move_up:nn #1#2
6625 { \tex_raise:D \__dim_eval:w #1 \__dim_eval_end: #2 }
6626 \cs_new_protected:Npn \box_move_down:nn #1#2
6627 { \tex_lower:D \__dim_eval:w #1 \__dim_eval_end: #2 }

```

15.4 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`
`\if_box_empty:N`

```

6628 \cs_new_eq:NN \if_hbox:N      \tex_ifhbox:D
6629 \cs_new_eq:NN \if_vbox:N      \tex_ifvbox:D
6630 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D

```

```

\box_if_horizontal_p:N
\box_if_horizontal_p:c  6631 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
\box_if_horizontal:N $\underline{TF}$  6632 { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_horizontal:c $\underline{TF}$  6633 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
\box_if_vertical_p:N 6634 { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_vertical_p:c 6635 \cs_generate_variant:Nn \box_if_horizontal_p:N { c }
\box_if_vertical:N $\underline{TF}$  6636 \cs_generate_variant:Nn \box_if_horizontal:NT { c }
\box_if_vertical:c $\underline{TF}$  6637 \cs_generate_variant:Nn \box_if_horizontal:NF { c }
6638 \cs_generate_variant:Nn \box_if_horizontal:NTF { c }
6639 \cs_generate_variant:Nn \box_if_vertical_p:N { c }
6640 \cs_generate_variant:Nn \box_if_vertical:NT { c }
6641 \cs_generate_variant:Nn \box_if_vertical:NF { c }
6642 \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`
`\box_if_empty:N \underline{TF}`
`\box_if_empty:c \underline{TF}`

```

6643 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
6644 { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
6645 \cs_generate_variant:Nn \box_if_empty_p:N { c }
6646 \cs_generate_variant:Nn \box_if_empty:NT { c }
6647 \cs_generate_variant:Nn \box_if_empty:NF { c }
6648 \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 ??.)

15.5 The last box inserted

`\box_set_to_last:N` Set a box to the previous box.
`\box_set_to_last:c`
`\box_gset_to_last:N`
`\box_gset_to_last:c`

```

6649 \cs_new_protected:Npn \box_set_to_last:N #1
6650 { \tex_setbox:D #1 \tex_lastbox:D }
6651 \cs_new_protected:Npn \box_gset_to_last:N
6652 { \tex_global:D \box_set_to_last:N }
6653 \cs_generate_variant:Nn \box_set_to_last:N { c }
6654 \cs_generate_variant:Nn \box_gset_to_last:N { c }

```

(End definition for `\box_set_to_last:N` and `\box_set_to_last:c`. These functions are documented on page ??.)

15.6 Constant boxes

`\c_empty_box` A box we never use.

```

6655 \box_new:N \c_empty_box

```

(End definition for `\c_empty_box`. This variable is documented on page 136.)

15.7 Scratch boxes

`\l_tmpa_box` Scratch boxes.

```
\l_tmpb_box 6656 \box_new:N \l_tmpa_box
\g_tmpa_box 6657 \box_new:N \l_tmpb_box
\g_tmpb_box 6658 \box_new:N \g_tmpa_box
6659 \box_new:N \g_tmpb_box
```

(End definition for `\l_tmpa_box` and others. These variables are documented on page 136.)

15.8 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 6660 \cs_new_protected:Npn \box_show:N #1
\box_show:Nnn 6661 { \box_show:Nnn #1 \c_max_int \c_max_int }
\box_show:cnn 6662 \cs_generate_variant:Nn \box_show:N { c }
6663 \cs_new_protected_nopar:Npn \box_show:Nnn
6664 { \__box_show:NNnn \c_one }
6665 \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 6666 \cs_new_protected:Npn \box_log:N #1
\box_log:Nnn 6667 { \box_log:Nnn #1 \c_max_int \c_max_int }
\box_log:cnn 6668 \cs_generate_variant:Nn \box_log:N { c }
6669 \cs_new_protected:Npn \box_log:Nnn #1#2#3
6670 {
6671   \use:x
6672   {
6673     \etex_interactionmode:D \c_zero
6674     \__box_show:NNnn \c_zero \exp_not:N #1
6675     { \int_eval:n {#2} } { \int_eval:n {#3} }
6676     \etex_interactionmode:D
6677     = \tex_the:D \etex_interactionmode:D \scan_stop:
6678   }
6679 }
6680 \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.

```
6681 \cs_new_protected:Npn \__box_show:NNnn #1#2#3#4
6682 {
```

```

6683 \group_begin:
6684 \int_set:Nn \tex_showboxbreadth:D {#3}
6685 \int_set:Nn \tex_showboxdepth:D {#4}
6686 \int_set_eq:NN \tex_tracingonline:D #1
6687 \box_if_exist:NTF #2
6688 { \tex_showbox:D \use:n {#2} }
6689 {
6690 \__msg_kernel_error:nnx { kernel } { variable-not-defined }
6691 { \token_to_str:N #2 }
6692 }
6693 \group_end:
6694 }

```

(End definition for `__box_show:NNnn`.)

15.9 Horizontal mode boxes

`\hbox:n` (The test suite for this command, and others in this file, is `m3box002.lvt`.)

Put a horizontal box directly into the input stream.

```
6695 \cs_new_protected:Npn \hbox:n { \tex_hbox:D \scan_stop: }
```

(End definition for `\hbox:n`. This function is documented on page 137.)

`\hbox_set:Nn`

`\hbox_set:cn`

`\hbox_gset:Nn`

`\hbox_gset:cn`

```

6696 \cs_new_protected:Npn \hbox_set:Nn #1#2 { \tex_setbox:D #1 \tex_hbox:D {#2} }
6697 \cs_new_protected:Npn \hbox_gset:Nn { \tex_global:D \hbox_set:Nn }
6698 \cs_generate_variant:Nn \hbox_set:Nn { c }
6699 \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`

`\hbox_set_to_wd:cnn`

`\hbox_gset_to_wd:Nnn`

`\hbox_gset_to_wd:cnn`

Storing material in a horizontal box with a specified width.

```

6700 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
6701 { \tex_setbox:D #1 \tex_hbox:D to \__dim_eval:w #2 \__dim_eval_end: {#3} }
6702 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn
6703 { \tex_global:D \hbox_set_to_wd:Nnn }
6704 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
6705 \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`

`\hbox_set:cw`

`\hbox_gset:Nw`

`\hbox_gset:cw`

`\hbox_set_end:`

`\hbox_gset_end:`

Storing material in a horizontal box. This type is useful in environment definitions.

```

6706 \cs_new_protected:Npn \hbox_set:Nw #1
6707 { \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token }
6708 \cs_new_protected:Npn \hbox_gset:Nw
6709 { \tex_global:D \hbox_set:Nw }
6710 \cs_generate_variant:Nn \hbox_set:Nw { c }
6711 \cs_generate_variant:Nn \hbox_gset:Nw { c }
6712 \cs_new_eq:NN \hbox_set_end: \c_group_end_token
6713 \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 138.)

`\hbox_set_inline_begin:N` Renamed September 2011.

```

\hbox_set_inline_begin:c 6714 \cs_new_eq:NN \hbox_set_inline_begin:N \hbox_set:Nw
\hbox_gset_inline_begin:N 6715 \cs_new_eq:NN \hbox_set_inline_begin:c \hbox_set:cw
\hbox_gset_inline_begin:c 6716 \cs_new_eq:NN \hbox_set_inline_end: \hbox_set_end:
\hbox_set_inline_end: 6717 \cs_new_eq:NN \hbox_gset_inline_begin:N \hbox_gset:Nw
\hbox_gset_inline_end: 6718 \cs_new_eq:NN \hbox_gset_inline_begin:c \hbox_gset:cw
6719 \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`

```

6720 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
6721 { \tex_hbox:D to \__dim_eval:w #1 \__dim_eval_end: {#2} }
6722 \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 137.)

`\hbox_overlap_left:n` Put a zero-sized box with the contents pushed against one side (which makes it stick out

`\hbox_overlap_right:n` on the other) directly into the input stream.

```

6723 \cs_new_protected:Npn \hbox_overlap_left:n #1
6724 { \hbox_to_zero:n { \tex_hss:D #1 } }
6725 \cs_new_protected:Npn \hbox_overlap_right:n #1
6726 { \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 137.)

`\hbox_unpack:N` Unpacking a box and if requested also clear it.

`\hbox_unpack:c`

`\hbox_unpack_clear:N`

`\hbox_unpack_clear:c`

```

6727 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
6728 \cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
6729 \cs_generate_variant:Nn \hbox_unpack:N { c }
6730 \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 ??.)

15.10 Vertical mode boxes

TeX ends these boxes directly with the internal `end_graf` routine. This means that there is no `\par` at the end of vertical boxes unless we insert one.

`\vbox:n` The following test files are used for this code: `m3box003.lvt`.

`\vbox_top:n` The following test files are used for this code: `m3box003.lvt`.

Put a vertical box directly into the input stream.

```

6731 \cs_new_protected:Npn \vbox:n #1 { \tex_vbox:D { #1 \par } }
6732 \cs_new_protected:Npn \vbox_top:n #1 { \tex_vtop:D { #1 \par } }

```

(End definition for `\vbox:n`. This function is documented on page 138.)

`\vbox_to_ht:nn` Put a vertical box directly into the input stream.
`\vbox_to_zero:n` 6733 `\cs_new_protected:Npn \vbox_to_ht:nn #1#2`
`\vbox_to_ht:nn` 6734 `{ \tex_vbox:D to __dim_eval:w #1 __dim_eval_end: { #2 \par } }`
`\vbox_to_zero:n` 6735 `\cs_new_protected:Npn \vbox_to_zero:n #1`
6736 `{ \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 139.)

`\vbox_set:Nn` Storing material in a vertical box with a natural height.
`\vbox_set:cn` 6737 `\cs_new_protected:Npn \vbox_set:Nn #1#2`
`\vbox_gset:Nn` 6738 `{ \tex_setbox:D #1 \tex_vbox:D { #2 \par } }`
`\vbox_gset:cn` 6739 `\cs_new_protected:Npn \vbox_gset:Nn { \tex_global:D \vbox_set:Nn }`
6740 `\cs_generate_variant:Nn \vbox_set:Nn { c }`
6741 `\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
`\vbox_set_top:cn` of the first object in the box.
`\vbox_gset_top:Nn` 6742 `\cs_new_protected:Npn \vbox_set_top:Nn #1#2`
`\vbox_gset_top:cn` 6743 `{ \tex_setbox:D #1 \tex_vtop:D { #2 \par } }`
6744 `\cs_new_protected:Npn \vbox_gset_top:Nn`
6745 `{ \tex_global:D \vbox_set_top:Nn }`
6746 `\cs_generate_variant:Nn \vbox_set_top:Nn { c }`
6747 `\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` 6748 `\cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3`
`\vbox_gset_to_ht:Nnn` 6749 `{ \tex_setbox:D #1 \tex_vbox:D to __dim_eval:w #2 __dim_eval_end: { #3 \par } }`
`\vbox_gset_to_ht:cnn` 6750 `\cs_new_protected:Npn \vbox_gset_to_ht:Nnn`
6751 `{ \tex_global:D \vbox_set_to_ht:Nnn }`
6752 `\cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }`
6753 `\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` 6754 `\cs_new_protected:Npn \vbox_set:Nw #1`
`\vbox_gset:Nw` 6755 `{ \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }`
`\vbox_gset:cw` 6756 `\cs_new_protected:Npn \vbox_gset:Nw`
`\vbox_set_end:` 6757 `{ \tex_global:D \vbox_set:Nw }`
`\vbox_gset_end:` 6758 `\cs_generate_variant:Nn \vbox_set:Nw { c }`
6759 `\cs_generate_variant:Nn \vbox_gset:Nw { c }`
6760 `\cs_new_protected:Npn \vbox_set_end:`
6761 `{`
6762 `\par`
6763 `\c_group_end_token`
6764 `}`
6765 `\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 139.)

```
\vbox_set_inline_begin:N Renamed September 2011.
\vbox_set_inline_begin:c 6766 \cs_new_eq:NN \vbox_set_inline_begin:N \vbox_set:Nw
\vbox_gset_inline_begin:N 6767 \cs_new_eq:NN \vbox_set_inline_begin:c \vbox_set:cw
\vbox_gset_inline_begin:c 6768 \cs_new_eq:NN \vbox_set_inline_end: \vbox_set_end:
\vbox_set_inline_end: 6769 \cs_new_eq:NN \vbox_gset_inline_begin:N \vbox_gset:Nw
\vbox_gset_inline_end: 6770 \cs_new_eq:NN \vbox_gset_inline_begin:c \vbox_gset:cw
6771 \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 6772 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
\vbox_unpack_clear:N 6773 \cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D
\vbox_unpack_clear:c 6774 \cs_generate_variant:Nn \vbox_unpack:N { c }
6775 \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.
6776 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
6777 { \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 139.)
6778 </initex | package>
```

16 l3coffins Implementation

```
6779 <*initex | package>
6780 <@@=coffin>
```

16.1 Coffins: data structures and general variables

```
\l__coffin_internal_box Scratch variables.
\l__coffin_internal_dim 6781 \box_new:N \l__coffin_internal_box
\l__coffin_internal_tl 6782 \dim_new:N \l__coffin_internal_dim
6783 \tl_new:N \l__coffin_internal_tl
(End definition for \l__coffin_internal_box. This variable is documented on page ??.)
```

`\c__coffin_corners_prop` The “corners”; of a coffin define the real content, as opposed to the T_EX bounding box. They all start off in the same place, of course.

```
6784 \prop_new:N \c__coffin_corners_prop
6785 \prop_put:Nnn \c__coffin_corners_prop { tl } { { 0 pt } { 0 pt } }
6786 \prop_put:Nnn \c__coffin_corners_prop { tr } { { 0 pt } { 0 pt } }
6787 \prop_put:Nnn \c__coffin_corners_prop { bl } { { 0 pt } { 0 pt } }
6788 \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.

```

6789 \prop_new:N \c__coffin_poles_prop
6790 \tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 0 pt } { 1000 pt } }
6791 \prop_put:Nno \c__coffin_poles_prop { l } { \l__coffin_internal_tl }
6792 \prop_put:Nno \c__coffin_poles_prop { hc } { \l__coffin_internal_tl }
6793 \prop_put:Nno \c__coffin_poles_prop { r } { \l__coffin_internal_tl }
6794 \tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 1000 pt } { 0 pt } }
6795 \prop_put:Nno \c__coffin_poles_prop { b } { \l__coffin_internal_tl }
6796 \prop_put:Nno \c__coffin_poles_prop { vc } { \l__coffin_internal_tl }
6797 \prop_put:Nno \c__coffin_poles_prop { t } { \l__coffin_internal_tl }
6798 \prop_put:Nno \c__coffin_poles_prop { B } { \l__coffin_internal_tl }
6799 \prop_put:Nno \c__coffin_poles_prop { H } { \l__coffin_internal_tl }
6800 \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`

```

6801 \fp_new:N \l__coffin_slope_x_fp
6802 \fp_new:N \l__coffin_slope_y_fp

```

(End definition for \l__coffin_slope_x_fp. This variable is documented on page ??.)

`\l__coffin_error_bool` For propagating errors so that parts of the code can work around them.

```

6803 \bool_new:N \l__coffin_error_bool

```

(End definition for \l__coffin_error_bool. This variable is documented on page ??.)

`\l__coffin_offset_x_dim` The offset between two sets of coffin handles when typesetting. These values are corrected

`\l__coffin_offset_y_dim` from those requested in an alignment for the positions of the handles.

```

6804 \dim_new:N \l__coffin_offset_x_dim
6805 \dim_new:N \l__coffin_offset_y_dim

```

(End definition for \l__coffin_offset_x_dim. This variable is documented on page ??.)

`\l__coffin_pole_a_tl` Needed for finding the intersection of two poles.

`\l__coffin_pole_b_tl`

```

6806 \tl_new:N \l__coffin_pole_a_tl
6807 \tl_new:N \l__coffin_pole_b_tl

```

(End definition for \l__coffin_pole_a_tl. This variable is documented on page ??.)

`\l__coffin_x_dim` For calculating intersections and so forth.

`\l__coffin_y_dim`

`\l__coffin_x_prime_dim`

`\l__coffin_y_prime_dim`

```

6808 \dim_new:N \l__coffin_x_dim
6809 \dim_new:N \l__coffin_y_dim
6810 \dim_new:N \l__coffin_x_prime_dim
6811 \dim_new:N \l__coffin_y_prime_dim

```

(End definition for \l__coffin_x_dim. This variable is documented on page ??.)

16.2 Basic coffin functions

There are a number of basic functions needed for creating coffins and placing material in them. This all relies on the following data structures.

`\coffin_if_exist_p:N` Several of the higher-level coffin functions will give multiple errors if the coffin does not exist. A cleaner way to handle this is provided here: both the box and the coffin structure are checked.

```
\coffin_if_exist_p:c
\coffin_if_exist:NTF
\coffin_if_exist:cTF
6812 \prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
6813 {
6814   \cs_if_exist:NTF #1
6815   {
6816     \cs_if_exist:cTF { l__coffin_poles_ \__int_value:w #1 _prop }
6817     { \prg_return_true: }
6818     { \prg_return_false: }
6819   }
6820   { \prg_return_false: }
6821 }
6822 \cs_generate_variant:Nn \coffin_if_exist_p:N { c }
6823 \cs_generate_variant:Nn \coffin_if_exist:NT { c }
6824 \cs_generate_variant:Nn \coffin_if_exist:NF { c }
6825 \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.

```
6826 \cs_new_protected:Npn \__coffin_if_exist:NT #1#2
6827 {
6828   \coffin_if_exist:NTF #1
6829   { #2 }
6830   {
6831     \__msg_kernel_error:nxx { kernel } { unknown-coffin }
6832     { \token_to_str:N #1 }
6833   }
6834 }
```

(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
6835 \cs_new_protected:Npn \coffin_clear:N #1
6836 {
6837   \__coffin_if_exist:NT #1
6838   {
6839     \box_clear:N #1
6840     \__coffin_reset_structure:N #1
6841   }
6842 }
6843 \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.

```

6844 \cs_new_protected:Npn \coffin_new:N #1
6845 {
6846   \box_new:N #1
6847   \prop_clear_new:c { l__coffin_corners_ } __int_value:w #1 _prop }
6848   \prop_clear_new:c { l__coffin_poles_ } __int_value:w #1 _prop }
6849   \prop_gset_eq:cN { l__coffin_corners_ } __int_value:w #1 _prop }
6850   \c__coffin_corners_prop
6851   \prop_gset_eq:cN { l__coffin_poles_ } __int_value:w #1 _prop }
6852   \c__coffin_poles_prop
6853 }
6854 \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.

```

6855 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
6856 {
6857   \__coffin_if_exist:NT #1
6858   {
6859     \hbox_set:Nn #1
6860     {
6861       \color_group_begin:
6862       \color_ensure_current:
6863       #2
6864       \color_group_end:
6865     }
6866     \__coffin_reset_structure:N #1
6867     \__coffin_update_poles:N #1
6868     \__coffin_update_corners:N #1
6869   }
6870 }
6871 \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:cn` 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).

```

6872 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
6873 {

```

```

6874     \_coffin_if_exist:NT #1
6875     {
6876         \vbox_set:Nn #1
6877         {
6878             \dim_set:Nn \tex_hsize:D {#2}
6879     <*package>
6880             \dim_set_eq:NN \linewidth \tex_hsize:D
6881             \dim_set_eq:NN \columnwidth \tex_hsize:D
6882     </package>
6883             \color_group_begin:
6884             #3
6885             \color_group_end:
6886         }
6887         \_coffin_reset_structure:N #1
6888         \_coffin_update_poles:N #1
6889         \_coffin_update_corners:N #1
6890         \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
6891         \_coffin_set_pole:Nnx #1 { T }
6892         {
6893             { 0 pt }
6894             { \dim_eval:n { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box } }
6895             { 1000 pt }
6896             { 0 pt }
6897         }
6898         \box_clear:N \l__coffin_internal_box
6899     }
6900 }
6901 \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
\hcoffin_set:cw
\hcoffin_set_end:

These are the “begin”/“end” versions of the above: watch the grouping!

```

6902 \cs_new_protected:Npn \hcoffin_set:Nw #1
6903 {
6904     \_coffin_if_exist:NT #1
6905     {
6906         \hbox_set:Nw #1 \color_group_begin: \color_ensure_current:
6907         \cs_set_protected_nopar:Npn \hcoffin_set_end:
6908         {
6909             \color_group_end:
6910             \hbox_set_end:
6911             \_coffin_reset_structure:N #1
6912             \_coffin_update_poles:N #1
6913             \_coffin_update_corners:N #1
6914         }
6915     }
6916 }
6917 \cs_new_protected_nopar:Npn \hcoffin_set_end: { }
6918 \cs_generate_variant:Nn \hcoffin_set:Nw { c }

```

(End definition for `\hcoffin_set:Nw` and `\hcoffin_set:cw`. These functions are documented on page 142.)

`\vcoffin_set:Nnw` The same for vertical coffins.

```

6919 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
6920 {
6921   \__coffin_if_exist:NT #1
6922   {
6923     \vbox_set:Nw #1
6924     \dim_set:Nn \tex_hsize:D {#2}
6925     <*package>
6926       \dim_set_eq:NN \linewidth \tex_hsize:D
6927       \dim_set_eq:NN \columnwidth \tex_hsize:D
6928     </package>
6929     \color_group_begin: \color_ensure_current:
6930     \cs_set_protected:Npn \vcoffin_set_end:
6931     {
6932       \color_group_end:
6933       \vbox_set_end:
6934       \__coffin_reset_structure:N #1
6935       \__coffin_update_poles:N #1
6936       \__coffin_update_corners:N #1
6937       \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
6938       \__coffin_set_pole:Nnx #1 { T }
6939       {
6940         { 0 pt }
6941         {
6942           \dim_eval:n { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
6943         }
6944         { 1000 pt }
6945         { 0 pt }
6946       }
6947       \box_clear:N \l__coffin_internal_box
6948     }
6949   }
6950 }
6951 \cs_new_protected_nopar:Npn \vcoffin_set_end: { }
6952 \cs_generate_variant:Nn \vcoffin_set:Nnw { c }

```

(End definition for `\vcoffin_set:Nnw` and `\vcoffin_set:cw`. These functions are documented on page 142.)

`\coffin_set_eq:NN` Setting two coffins equal is just a wrapper around other functions.

```

6953 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
6954 {
6955   \__coffin_if_exist:NT #1
6956   {
6957     \box_set_eq:NN #1 #2
6958     \__coffin_set_eq_structure:NN #1 #2
6959   }
6960 }

```

```

6961 \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 Special coffins: these cannot be set up earlier as they need \coffin_new:N. The empty coffin is set as a box as the full coffin-setting system needs some material which is not yet available.

\l__coffin_aligned_coffin

\l__coffin_aligned_internal_coffin

```

6962 \coffin_new:N \c_empty_coffin
6963 \hbox_set:Nn \c_empty_coffin { }
6964 \coffin_new:N \l__coffin_aligned_coffin
6965 \coffin_new:N \l__coffin_aligned_internal_coffin
(End definition for \c_empty_coffin. This variable is documented on page ??.)

```

\l_tmpa_coffin The usual scratch space.

\l_tmpb_coffin

```

6966 \coffin_new:N \l_tmpa_coffin
6967 \coffin_new:N \l_tmpb_coffin
(End definition for \l_tmpa_coffin and \l_tmpb_coffin. These variables are documented on page 144.)

```

16.3 Measuring coffins

\coffin_dp:N Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

\coffin_dp:c

\coffin_ht:N

\coffin_ht:c

\coffin_wd:N

\coffin_wd:c

```

6968 \cs_new_eq:NN \coffin_dp:N \box_dp:N
6969 \cs_new_eq:NN \coffin_dp:c \box_dp:c
6970 \cs_new_eq:NN \coffin_ht:N \box_ht:N
6971 \cs_new_eq:NN \coffin_ht:c \box_ht:c
6972 \cs_new_eq:NN \coffin_wd:N \box_wd:N
6973 \cs_new_eq:NN \coffin_wd:c \box_wd:c
(End definition for \coffin_dp:N and others. These functions are documented on page ??.)

```

16.4 Coffins: handle and pole management

__coffin_get_pole:NnN A simple wrapper around the recovery of a coffin pole, with some error checking and recovery built-in.

```

6974 \cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3
6975 {
6976   \prop_get:cnNF
6977     { l__coffin_poles_ \__int_value:w #1 _prop } {#2} #3
6978     {
6979       \_msg_kernel_error:nxxx { kernel } { unknown-coffin-pole }
6980       {#2} { \token_to_str:N #1 }
6981       \tl_set:Nn #3 { { 0 pt } { 0 pt } { 0 pt } { 0 pt } }
6982     }
6983 }
(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.

```

6984 \cs_new_protected:Npn \__coffin_reset_structure:N #1
6985 {
6986   \prop_set_eq:cN { l__coffin_corners_ \__int_value:w #1 _prop }
6987   \c__coffin_corners_prop
6988   \prop_set_eq:cN { l__coffin_poles_ \__int_value:w #1 _prop }
6989   \c__coffin_poles_prop
6990 }

```

(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`

```

6991 \cs_new_protected:Npn \__coffin_set_eq_structure:NN #1#2
6992 {
6993   \prop_set_eq:cc { l__coffin_corners_ \__int_value:w #1 _prop }
6994   { l__coffin_corners_ \__int_value:w #2 _prop }
6995   \prop_set_eq:cc { l__coffin_poles_ \__int_value:w #1 _prop }
6996   { l__coffin_poles_ \__int_value:w #2 _prop }
6997 }
6998 \cs_new_protected:Npn \__coffin_gset_eq_structure:NN #1#2
6999 {
7000   \prop_gset_eq:cc { l__coffin_corners_ \__int_value:w #1 _prop }
7001   { l__coffin_corners_ \__int_value:w #2 _prop }
7002   \prop_gset_eq:cc { l__coffin_poles_ \__int_value:w #1 _prop }
7003   { l__coffin_poles_ \__int_value:w #2 _prop }
7004 }

```

(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:cnm`

`\coffin_set_vertical_pole:Nnn`

`\coffin_set_vertical_pole:cnm`

`__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.

```

7005 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
7006 {
7007   \__coffin_if_exist:NT #1
7008   {
7009     \__coffin_set_pole:Nnx #1 {#2}
7010     {
7011       { 0 pt } { \dim_eval:n {#3} }
7012       { 1000 pt } { 0 pt }
7013     }
7014   }
7015 }
7016 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
7017 {
7018   \__coffin_if_exist:NT #1
7019   {
7020     \__coffin_set_pole:Nnx #1 {#2}
7021     {
7022       { \dim_eval:n {#3} } { 0 pt }

```

```

7023         { 0 pt } { 1000 pt }
7024     }
7025 }
7026 }
7027 \cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
7028 { \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } {#2} {#3} }
7029 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
7030 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
7031 \cs_generate_variant:Nn \__coffin_set_pole:Nnn { Nnx }

```

(End definition for `\coffin_set_horizontal_pole:Nnn` and `\coffin_set_horizontal_pole:cnx`. 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 \TeX box.

```

7032 \cs_new_protected:Npn \__coffin_update_corners:N #1
7033 {
7034     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tl }
7035     { { 0 pt } { \dim_use:N \box_ht:N #1 } }
7036     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tr }
7037     { { \dim_use:N \box_wd:N #1 } { \dim_use:N \box_ht:N #1 } }
7038     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { bl }
7039     { { 0 pt } { \dim_eval:n { - \box_dp:N #1 } } }
7040     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { br }
7041     { { \dim_use:N \box_wd:N #1 } { \dim_eval:n { - \box_dp:N #1 } } }
7042 }

```

(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.

```

7043 \cs_new_protected:Npn \__coffin_update_poles:N #1
7044 {
7045     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { hc }
7046     {
7047         { \dim_eval:n { 0.5 \box_wd:N #1 } }
7048         { 0 pt } { 0 pt } { 1000 pt }
7049     }
7050     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { r }
7051     {
7052         { \dim_use:N \box_wd:N #1 }
7053         { 0 pt } { 0 pt } { 1000 pt }
7054     }
7055     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { vc }
7056     {
7057         { 0 pt }
7058         { \dim_eval:n { ( \box_ht:N #1 - \box_dp:N #1 ) / 2 } }
7059         { 1000 pt }
7060         { 0 pt }

```

```

7061     }
7062     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { t }
7063     {
7064         { 0 pt }
7065         { \dim_use:N \box_ht:N #1 }
7066         { 1000 pt }
7067         { 0 pt }
7068     }
7069     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { b }
7070     {
7071         { 0 pt }
7072         { \dim_eval:n { - \box_dp:N #1 } }
7073         { 1000 pt }
7074         { 0 pt }
7075     }
7076 }

```

(End definition for `__coffin_update_poles:N`. This function is documented on page ??.)

16.5 Coffins: calculation of pole intersections

```

\__coffin_calculate_intersection:Nnn
\__coffin_calculate_intersection:nnnnnnnn
\__coffin_calculate_intersection_aux:nnnnnnN

```

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.

```

7077 \cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
7078 {
7079     \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
7080     \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
7081     \bool_set_false:N \l__coffin_error_bool
7082     \exp_last_two_unbraced:Noo
7083     \__coffin_calculate_intersection:nnnnnnnn
7084     \l__coffin_pole_a_tl \l__coffin_pole_b_tl
7085     \bool_if:NT \l__coffin_error_bool
7086     {
7087         \__msg_kernel_error:nn { kernel } { no-pole-intersection }
7088         \dim_zero:N \l__coffin_x_dim
7089         \dim_zero:N \l__coffin_y_dim
7090     }
7091 }

```

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.

```

7092 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
7093 #1#2#3#4#5#6#7#8
7094 {
7095     \dim_compare:nNnTF {#3} = { \c_zero_dim }

```

The case where the first pole is vertical. So the x -component of the interaction will be at a . There is then a test on the second pole: if it is also vertical then there is an error.

```

7096      {
7097          \dim_set:Nn \l__coffin_x_dim {#1}
7098          \dim_compare:nNnTF {#7} = \c_zero_dim
7099              { \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.

```

7100      {
7101          \dim_compare:nNnTF {#8} = \c_zero_dim
7102              { \dim_set:Nn \l__coffin_y_dim {#6} }
7103              {
7104                  \__coffin_calculate_intersection_aux:nnnnnN
7105                      {#1} {#5} {#6} {#7} {#8} \l__coffin_y_dim
7106              }
7107      }
7108  }

```

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.

```

7109      {
7110          \dim_compare:nNnTF {#4} = \c_zero_dim
7111              {
7112                  \dim_set:Nn \l__coffin_y_dim {#2}
7113                  \dim_compare:nNnTF {#8} = { \c_zero_dim }
7114                  { \bool_set_true:N \l__coffin_error_bool }
7115                  {
7116                      \dim_compare:nNnTF {#7} = \c_zero_dim
7117                      { \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.

```

7118      {
7119          \__coffin_calculate_intersection_aux:nnnnnN
7120              {#2} {#6} {#5} {#8} {#7} \l__coffin_x_dim
7121      }
7122  }
7123

```

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.

```

7124 {
7125     \dim_compare:nNnTF {#7} = \c_zero_dim
7126     {
7127         \dim_set:Nn \l__coffin_x_dim {#5}
7128         \__coffin_calculate_intersection_aux:nnnnnN
7129         {#5} {#1} {#2} {#3} {#4} \l__coffin_y_dim
7130     }
7131     {
7132         \dim_compare:nNnTF {#8} = \c_zero_dim
7133         {
7134             \dim_set:Nn \l__coffin_y_dim {#6}
7135             \__coffin_calculate_intersection_aux:nnnnnN
7136             {#6} {#2} {#1} {#4} {#3} \l__coffin_x_dim
7137         }

```

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.

```

7138 {
7139     \fp_set:Nn \l__coffin_slope_x_fp
7140     { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
7141     \fp_set:Nn \l__coffin_slope_y_fp
7142     { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
7143     \fp_compare:nNnTF
7144     \l__coffin_slope_x_fp = \l__coffin_slope_y_fp
7145     { \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.

```

7146 {
7147     \dim_set:Nn \l__coffin_x_dim
7148     {
7149         \fp_to_dim:n
7150         {
7151             (
7152                 \dim_to_fp:n {#1} * \l__coffin_slope_x_fp
7153                 - ( \dim_to_fp:n {#5} * \l__coffin_slope_y_fp )
7154                 - \dim_to_fp:n {#2}
7155                 + \dim_to_fp:n {#6}
7156             )
7157             /
7158             ( \l__coffin_slope_x_fp - \l__coffin_slope_y_fp )
7159         }
7160     }

```

```

7161         \__coffin_calculate_intersection_aux:nnnnnN
7162         { \l__coffin_x_dim }
7163         {#5} {#6} {#8} {#7} \l__coffin_y_dim
7164     }
7165 }
7166 }
7167 }
7168 }
7169 }

```

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.

```

7170 \cs_new_protected:Npn \__coffin_calculate_intersection_aux:nnnnnN #1#2#3#4#5#6
7171 {
7172     \dim_set:Nn #6
7173     {
7174         \fp_to_dim:n
7175         {
7176             \dim_to_fp:n {#4} *
7177             ( \dim_to_fp:n {#1} - \dim_to_fp:n {#2} ) /
7178             \dim_to_fp:n {#5}
7179             + \dim_to_fp:n {#3}
7180         }
7181     }
7182 }

```

(End definition for `__coffin_calculate_intersection:Nnn`. This function is documented on page ??.)

16.6 Aligning and typesetting of coffins

`\coffin_join:NnnNnnnn`
`\coffin_join:cnnNnnnn`
`\coffin_join:Nnncnnnn`
`\coffin_join:cnncnnnn`

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which will have all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```

7183 \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
7184 {
7185     \__coffin_align:NnnNnnnnN
7186     #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.

```

7187     \hbox_set:Nn \l__coffin_aligned_coffin
7188     {
7189         \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
7190         { \tex_kern:D -\l__coffin_offset_x_dim }

```

```

7191 \hbox_unpack:N \l__coffin_aligned_coffin
7192 \dim_set:Nn \l__coffin_internal_dim
7193 { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
7194 \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
7195 { \tex_kern:D -\l__coffin_internal_dim }
7196 }

```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```

7197 \__coffin_reset_structure:N \l__coffin_aligned_coffin
7198 \prop_clear:c
7199 { \l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _ prop }
7200 \__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.

```

7201 \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
7202 {
7203   \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
7204   \__coffin_offset_poles:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
7205   \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
7206   \__coffin_offset_corners:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
7207 }
7208 {
7209   \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
7210   \__coffin_offset_poles:Nnn #4
7211   { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
7212   \__coffin_offset_corners:Nnn #1 { 0 pt } { 0 pt }
7213   \__coffin_offset_corners:Nnn #4
7214   { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
7215 }
7216 \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
7217 \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
7218 }
7219 \cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }

```

(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.

`\coffin_attach:cnNnnnnn` The function used when marking a position is hear also as it is similar but without the

`\coffin_attach:Nnncnnnnn` structure updates.

```

\coffin_attach_mark:NnnNnnnn
7220 \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
7221 {
7222   \__coffin_align:NnnNnnnnN
7223   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
7224   \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
7225   \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
7226   \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
7227   \__coffin_reset_structure:N \l__coffin_aligned_coffin

```

```

7228 \prop_set_eq:cc
7229 { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
7230 { l__coffin_corners_ \__int_value:w #1 _prop }
7231 \__coffin_update_poles:N \l__coffin_aligned_coffin
7232 \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
7233 \__coffin_offset_poles:Nnn #4
7234 { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
7235 \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
7236 \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
7237 }
7238 \cs_new_protected:Npn \coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
7239 {
7240 \__coffin_align:NnnNnnnnN
7241 #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
7242 \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
7243 \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
7244 \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
7245 \box_set_eq:NN #1 \l__coffin_aligned_coffin
7246 }
7247 \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.

```

7248 \cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
7249 {
7250 \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
7251 \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
7252 \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
7253 \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
7254 \dim_set:Nn \l__coffin_offset_x_dim
7255 { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
7256 \dim_set:Nn \l__coffin_offset_y_dim
7257 { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
7258 \hbox_set:Nn \l__coffin_aligned_internal_coffin
7259 {
7260 \box_use:N #1
7261 \tex_kern:D -\box_wd:N #1
7262 \tex_kern:D \l__coffin_offset_x_dim
7263 \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
7264 }
7265 \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
7266 }

```

(End definition for __coffin_align:NnnNnnnnN. This function is documented on page ??.)

`_coffin_offset_poles:Nnn`
`_coffin_offset_pole:Nnnnnnn`

Transferring structures from one coffin to another requires that the positions are updated by the offset between the two coffins. This is done by mapping to the property list of the source coffins, moving as appropriate and saving to the new coffin data structures. The test for a - means that the structures from the parent coffins are uniquely labelled and do not depend on the order of alignment. The pay off for this is that - should not be used in coffin pole or handle names, and that multiple alignments do not result in a whole set of values.

```

7267 \cs_new_protected:Npn \_coffin_offset_poles:Nnn #1#2#3
7268 {
7269   \prop_map_inline:cn { l__coffin_poles_ \_int_value:w #1 _prop }
7270   { \_coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
7271 }
7272 \cs_new_protected:Npn \_coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
7273 {
7274   \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
7275   \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
7276   \tl_if_in:nnTF {#2} { - }
7277   { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
7278   { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
7279   \exp_last_unbraced:NNo \_coffin_set_pole:Nnx \l__coffin_aligned_coffin
7280   { \l__coffin_internal_tl }
7281   {
7282     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
7283     {#5} {#6}
7284   }
7285 }

```

(End definition for `_coffin_offset_poles:Nnn`. This function is documented on page ??.)

`_coffin_offset_corners:Nnn`
`_coffin_offset_corner:Nnnnn`

Saving the offset corners of a coffin is very similar, except that there is no need to worry about naming: every corner can be saved here as order is unimportant.

```

7286 \cs_new_protected:Npn \_coffin_offset_corners:Nnn #1#2#3
7287 {
7288   \prop_map_inline:cn { l__coffin_corners_ \_int_value:w #1 _prop }
7289   { \_coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
7290 }
7291 \cs_new_protected:Npn \_coffin_offset_corner:Nnnnn #1#2#3#4#5#6
7292 {
7293   \prop_put:cnx
7294   { l__coffin_corners_ \_int_value:w \l__coffin_aligned_coffin _prop }
7295   { #1 - #2 }
7296   {
7297     { \dim_eval:n { #3 + #5 } }
7298     { \dim_eval:n { #4 + #6 } }
7299   }
7300 }

```

(End definition for `_coffin_offset_corners:Nnn`. This function is documented on page ??.)

`_coffin_update_vertical_poles:NNN`
`_coffin_update_T:nnnnnnnnN`
`_coffin_update_B:nnnnnnnnN`

The T and B poles will need to be recalculated after alignment. These functions find the larger absolute value for the poles, but this is of course only logical when the poles are horizontal.

```

7301 \cs_new_protected:Npn \_coffin_update_vertical_poles:NNN #1#2#3
7302 {
7303   \_coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
7304   \_coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
7305   \exp_last_two_unbraced:Noo \_coffin_update_T:nnnnnnnnN
7306   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
7307   \_coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
7308   \_coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
7309   \exp_last_two_unbraced:Noo \_coffin_update_B:nnnnnnnnN
7310   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
7311 }
7312 \cs_new_protected:Npn \_coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
7313 {
7314   \dim_compare:nNnTF {#2} < {#6}
7315   {
7316     \_coffin_set_pole:Nnx #9 { T }
7317     { { 0 pt } {#6} { 1000 pt } { 0 pt } }
7318   }
7319   {
7320     \_coffin_set_pole:Nnx #9 { T }
7321     { { 0 pt } {#2} { 1000 pt } { 0 pt } }
7322   }
7323 }
7324 \cs_new_protected:Npn \_coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
7325 {
7326   \dim_compare:nNnTF {#2} < {#6}
7327   {
7328     \_coffin_set_pole:Nnx #9 { B }
7329     { { 0 pt } {#2} { 1000 pt } { 0 pt } }
7330   }
7331   {
7332     \_coffin_set_pole:Nnx #9 { B }
7333     { { 0 pt } {#6} { 1000 pt } { 0 pt } }
7334   }
7335 }

```

(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.

```

7336 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
7337 {
7338   \hbox_unpack:N \c_empty_box
7339   \_coffin_align:NnnNnnnnN \c_empty_coffin { H } { 1 }
7340   #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
7341   \box_use:N \l__coffin_aligned_coffin

```

```

7342 }
7343 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }

```

(End definition for \coffin_typeset:Nnnnn and \coffin_typeset:cnnnn. These functions are documented on page ??.)

16.7 Coffin diagnostics

\l__coffin_display_coffin Used for printing coffins with data structures attached.

```

\l__coffin_display_coord_coffin 7344 \coffin_new:N \l__coffin_display_coffin
\l__coffin_display_pole_coffin 7345 \coffin_new:N \l__coffin_display_coord_coffin
7346 \coffin_new:N \l__coffin_display_pole_coffin

```

(End definition for \l__coffin_display_coffin. This variable is documented on page ??.)

\l__coffin_display_handles_prop This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

```

7347 \prop_new:N \l__coffin_display_handles_prop
7348 \prop_put:Nnn \l__coffin_display_handles_prop { tl }
7349 { { b } { r } { -1 } { 1 } }
7350 \prop_put:Nnn \l__coffin_display_handles_prop { thc }
7351 { { b } { hc } { 0 } { 1 } }
7352 \prop_put:Nnn \l__coffin_display_handles_prop { tr }
7353 { { b } { l } { 1 } { 1 } }
7354 \prop_put:Nnn \l__coffin_display_handles_prop { vcl }
7355 { { vc } { r } { -1 } { 0 } }
7356 \prop_put:Nnn \l__coffin_display_handles_prop { vhc }
7357 { { vc } { hc } { 0 } { 0 } }
7358 \prop_put:Nnn \l__coffin_display_handles_prop { vcr }
7359 { { vc } { l } { 1 } { 0 } }
7360 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
7361 { { t } { r } { -1 } { -1 } }
7362 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
7363 { { t } { hc } { 0 } { -1 } }
7364 \prop_put:Nnn \l__coffin_display_handles_prop { br }
7365 { { t } { l } { 1 } { -1 } }
7366 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }
7367 { { t } { r } { -1 } { -1 } }
7368 \prop_put:Nnn \l__coffin_display_handles_prop { Thc }
7369 { { t } { hc } { 0 } { -1 } }
7370 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
7371 { { t } { l } { 1 } { -1 } }
7372 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
7373 { { vc } { r } { -1 } { 1 } }
7374 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
7375 { { vc } { hc } { 0 } { 1 } }
7376 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
7377 { { vc } { l } { 1 } { 1 } }
7378 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
7379 { { b } { r } { -1 } { -1 } }
7380 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }

```

```

7381 { { b } { hc } { 0 } { -1 } }
7382 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
7383 { { b } { 1 } { 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.

```

7384 \dim_new:N \l__coffin_display_offset_dim
7385 \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.

```

7386 \dim_new:N \l__coffin_display_x_dim
7387 \dim_new:N \l__coffin_display_y_dim
(End definition for \l__coffin_display_x_dim. This variable is documented on page ??.)

```

`\l__coffin_display_poles_prop` A property list for printing poles: various things need to be deleted from this to get a “nice” output.

```

7388 \prop_new:N \l__coffin_display_poles_prop
(End definition for \l__coffin_display_poles_prop. This variable is documented on page ??.)

```

`\l__coffin_display_font_tl` Stores the settings used to print coffin data: this keeps things flexible.

```

7389 \tl_new:N \l__coffin_display_font_tl
7390 <*initex>
7391 \tl_set:Nn \l__coffin_display_font_tl { } % TODO
7392 </initex>
7393 <*package>
7394 \tl_set:Nn \l__coffin_display_font_tl { \sfamily \tiny }
7395 </package>
(End definition for \l__coffin_display_font_tl. This variable is documented on page ??.)

```

`\coffin_mark_handle:Nnnn` Marking a single handle is relatively easy. The standard attachment function is used,
`\coffin_mark_handle:cnnn` meaning that there are two calculations for the location. However, this is likely to be
`__coffin_mark_handle_aux:nnnnNnn` okay given the load expected. Contrast with the more optimised version for showing all handles which comes next.

```

7396 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
7397 {
7398   \hcoffin_set:Nn \l__coffin_display_pole_coffin
7399   {
7400     <*initex>
7401     \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
7402     </initex>
7403     <*package>
7404     \color {#4}
7405     \rule { 1 pt } { 1 pt }
7406     </package>
7407   }

```

```

7408 \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
7409 \l__coffin_display_pole_coffin { hc } { vc } { 0 pt } { 0 pt }
7410 \hcoffin_set:Nn \l__coffin_display_coord_coffin
7411 {
7412 <*initex>
7413 % TODO
7414 </initex>
7415 <*package>
7416 \color {#4}
7417 </package>
7418 \l__coffin_display_font_tl
7419 ( \tl_to_str:n { #2 , #3 } )
7420 }
7421 \prop_get:NnN \l__coffin_display_handles_prop
7422 { #2 #3 } \l__coffin_internal_tl
7423 \quark_if_no_value:NTF \l__coffin_internal_tl
7424 {
7425 \prop_get:NnN \l__coffin_display_handles_prop
7426 { #3 #2 } \l__coffin_internal_tl
7427 \quark_if_no_value:NTF \l__coffin_internal_tl
7428 {
7429 \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
7430 \l__coffin_display_coord_coffin { l } { vc }
7431 { 1 pt } { 0 pt }
7432 }
7433 {
7434 \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
7435 \l__coffin_internal_tl #1 {#2} {#3}
7436 }
7437 }
7438 {
7439 \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
7440 \l__coffin_internal_tl #1 {#2} {#3}
7441 }
7442 }
7443 \cs_new_protected:Npn \__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
7444 {
7445 \coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
7446 \l__coffin_display_coord_coffin {#1} {#2}
7447 { #3 \l__coffin_display_offset_dim }
7448 { #4 \l__coffin_display_offset_dim }
7449 }
7450 \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.

```

7451 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
7452 {
7453   \hcoffin_set:Nn \l__coffin_display_pole_coffin
7454   {
7455     (*initex)
7456     \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
7457   }
7458   (*package)
7459   \color {#2}
7460   \rule { 1 pt } { 1 pt }
7461 }
7462 }
7463 \prop_set_eq:Nc \l__coffin_display_poles_prop
7464 { l__coffin_poles_ \__int_value:w #1 _prop }
7465 \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
7466 \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
7467 \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
7468 { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
7469 \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
7470 \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
7471 { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
7472 \coffin_set_eq:NN \l__coffin_display_coffin #1
7473 \prop_map_inline:Nn \l__coffin_display_poles_prop
7474 {
7475   \prop_remove:Nn \l__coffin_display_poles_prop {##1}
7476   \__coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
7477 }
7478 \box_use:N \l__coffin_display_coffin
7479 }

```

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.

```

7480 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
7481 {
7482   \prop_map_inline:Nn \l__coffin_display_poles_prop
7483   {
7484     \bool_set_false:N \l__coffin_error_bool
7485     \__coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2
7486     \bool_if:NF \l__coffin_error_bool
7487     {
7488       \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
7489       \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
7490       \__coffin_display_attach:Nnnnn
7491       \l__coffin_display_pole_coffin { hc } { vc }
7492       { 0 pt } { 0 pt }
7493       \hcoffin_set:Nn \l__coffin_display_coord_coffin
7494       {
7495         (*initex)

```

```

7496                                     % TODO
7497 </initex>
7498 <*package>
7499                                     \color {#6}
7500 </package>
7501                                     \l__coffin_display_font_tl
7502                                     ( \tl_to_str:n { #1 , ##1 } )
7503                                     }
7504 \prop_get:NnN \l__coffin_display_handles_prop
7505 { #1 ##1 } \l__coffin_internal_tl
7506 \quark_if_no_value:NTF \l__coffin_internal_tl
7507 {
7508     \prop_get:NnN \l__coffin_display_handles_prop
7509     { ##1 #1 } \l__coffin_internal_tl
7510     \quark_if_no_value:NTF \l__coffin_internal_tl
7511     {
7512         \__coffin_display_attach:Nnnnn
7513         \l__coffin_display_coord_coffin { 1 } { vc }
7514         { 1 pt } { 0 pt }
7515     }
7516     {
7517         \exp_last_unbraced:No
7518         \__coffin_display_handles_aux:nnnn
7519         \l__coffin_internal_tl
7520     }
7521 }
7522 {
7523     \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
7524     \l__coffin_internal_tl
7525 }
7526 }
7527 }
7528 }
7529 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
7530 {
7531     \__coffin_display_attach:Nnnnn
7532     \l__coffin_display_coord_coffin {#1} {#2}
7533     { #3 \l__coffin_display_offset_dim }
7534     { #4 \l__coffin_display_offset_dim }
7535 }
7536 \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.

```

7537 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
7538 {
7539     \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
7540     \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
7541     \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }

```

```

7542 \dim_set:Nn \l__coffin_offset_x_dim
7543   { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
7544 \dim_set:Nn \l__coffin_offset_y_dim
7545   { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
7546 \hbox_set:Nn \l__coffin_aligned_coffin
7547   {
7548     \box_use:N \l__coffin_display_coffin
7549     \tex_kern:D -\box_wd:N \l__coffin_display_coffin
7550     \tex_kern:D \l__coffin_offset_x_dim
7551     \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
7552   }
7553 \box_set_ht:Nn \l__coffin_aligned_coffin
7554   { \box_ht:N \l__coffin_display_coffin }
7555 \box_set_dp:Nn \l__coffin_aligned_coffin
7556   { \box_dp:N \l__coffin_display_coffin }
7557 \box_set_wd:Nn \l__coffin_aligned_coffin
7558   { \box_wd:N \l__coffin_display_coffin }
7559 \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
7560 }

```

(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.

```

7561 \cs_new_protected:Npn \coffin_show_structure:N #1
7562 {
7563   \__coffin_if_exist:NT #1
7564   {
7565     \__msg_show_variable:Nnn #1 { coffins }
7566     {
7567       \prop_map_function:cN
7568         { l__coffin_poles_ \__int_value:w #1 _prop }
7569       \__msg_show_item_unbraced:nn
7570     }
7571   }
7572 }
7573 \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 ??.)

16.8 Messages

```

7574 \__msg_kernel_new:nnnn { kernel } { no-pole-intersection }
7575 { No~intersection~between~coffin~poles. }
7576 {
7577   \c__msg_coding_error_text_tl
7578   LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
7579   but~they~do~not~have~a~unique~meeting~point:~
7580   the~value~(0~pt,~0~pt)~will~be~used.

```



```

7581 }
7582 \_msg_kernel_new:nnnn { kernel } { unknown-coffin }
7583 { Unknown~coffin~'#1'. }
7584 { The~coffin~'#1'~was~never~defined. }
7585 \_msg_kernel_new:nnnn { kernel } { unknown-coffin-pole }
7586 { Pole~'#1'~unknown~for~coffin~'#2'. }
7587 {
7588   \c_msg_coding_error_text_tl
7589   LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
7590   but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
7591 }
7592 \_msg_kernel_new:nnn { kernel } { show-coffins }
7593 {
7594   Size-of~coffin~\token_to_str:N #1 : \\
7595   > ~ ht~~\dim_use:N \box_ht:N #1 \\
7596   > ~ dp~~\dim_use:N \box_dp:N #1 \\
7597   > ~ wd~~\dim_use:N \box_wd:N #1 \\
7598   Poles~of~coffin~\token_to_str:N #1 :
7599 }
7600 </initex | package>

```

17 l3color Implementation

```

7601 <*initex | package>

```

\color_group_begin: Grouping for color is almost the same as using the basic `\group_begin:` and `\group_end:` functions. However, in vertical mode the end-of-group needs a `\par`, which in horizontal mode does nothing.

```

7602 \cs_new_eq:NN \color_group_begin: \group_begin:
7603 \cs_new_protected_nopar:Npn \color_group_end:
7604 {
7605   \tex_par:D
7606   \group_end:
7607 }

```

(End definition for \color_group_begin: and \color_group_end:.. These functions are documented on page 145.)

\color_ensure_current: A driver-independent wrapper for setting the foreground color to the current color “now”.

```

7608 <*initex>
7609 \cs_new_protected_nopar:Npn \color_ensure_current:
7610 { \_driver_color_ensure_current: }
7611 </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.

```

7612 <*package>
7613 \cs_new_protected_nopar:Npn \color_ensure_current: { }
7614 \AtBeginDocument

```

```

7615 {
7616   \cs_if_exist:NTF \__driver_color_ensure_current:
7617   {
7618     \cs_set_protected_nopar:Npn \color_ensure_current:
7619     { \__driver_color_ensure_current: }
7620   }
7621   {
7622     \cs_if_exist:NT \set@color
7623     { \cs_set_protected_nopar:Npn \color_ensure_current: { \set@color } }
7624   }
7625 }
7626 </package>

```

(End definition for `\color_ensure_current:`. This function is documented on page 145.)

```

7627 </initex | package>

```

18 l3msg implementation

```

7628 <*initex | package>
7629 <@@=msg>

```

`\l_msg_internal_tl` A general scratch for the module.

```

7630 \tl_new:N \l_msg_internal_tl

```

(End definition for `\l_msg_internal_tl`. This variable is documented on page ??.)

18.1 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

```

\c_msg_text_prefix_tl \c_msg_more_text_prefix_tl

```

Locations for the text of messages.

```

7631 \tl_const:Nn \c_msg_text_prefix_tl { msg~text~>~ }
7632 \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
7633 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
7634 {
7635   \cs_if_exist:cTF { \c_msg_text_prefix_tl #1 / #2 }
7636   { \prg_return_true: } { \prg_return_false: }
7637 }

```

(End definition for `\msg_if_exist:nn`. These functions are documented on page 147.)

`__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.

```

7638 \cs_new_protected:Npn \__chk_if_free_msg:nn #1#2
7639 {
7640   \msg_if_exist:nnT {#1} {#2}
7641   {
7642     \__msg_kernel_error:nxxx { kernel } { message-already-defined }
7643     {#1} {#2}
7644   }
7645 }
7646 <*package>
7647 \tex_ifodd:D \l@expl@log@functions@bool
7648 \cs_gset_protected:Npn \__chk_if_free_msg:nn #1#2
7649 {
7650   \msg_if_exist:nnT {#1} {#2}
7651   {
7652     \__msg_kernel_error:nxxx { kernel } { message-already-defined }
7653     {#1} {#2}
7654   }
7655   \iow_log:x { Defining-message~ #1 / #2 ~\msg_line_context: }
7656 }
7657 \fi:
7658 </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:nnnn
\msg_new:nnnn
\msg_gset:nnnn
\msg_set:nnnn
\msg_set:nnnn
7659 \cs_new_protected:Npn \msg_new:nnnn #1#2
7660 {
7661   \__chk_if_free_msg:nn {#1} {#2}
7662   \msg_gset:nnnn {#1} {#2}
7663 }
7664 \cs_new_protected:Npn \msg_new:nnn #1#2#3
7665 { \msg_new:nnnn {#1} {#2} {#3} { } }
7666 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
7667 {
7668   \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
7669   ##1##2##3##4 {#3}
7670   \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
7671   ##1##2##3##4 {#4}
7672 }
7673 \cs_new_protected:Npn \msg_set:nnn #1#2#3
7674 { \msg_set:nnnn {#1} {#2} {#3} { } }
7675 \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
7676 {
7677   \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
7678   ##1##2##3##4 {#3}

```

```

7679 \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
7680      ##1##2##3##4 {#4}
7681 }
7682 \cs_new_protected:Npn \msg_gset:nnn #1#2#3
7683 { \msg_gset:nnnn {#1} {#2} {#3} { } }

```

(End definition for \msg_new:nnnn and \msg_new:nnn. These functions are documented on page ??.)

18.2 Messages: support functions and text

`\c__msg_coding_error_text_tl` Simple pieces of text for messages.

```

\c__msg_continue_text_tl 7684 \tl_const:Nn \c__msg_coding_error_text_tl
\c__msg_critical_text_tl 7685 {
\c__msg_fatal_text_tl    7686   This-is-a-coding-error.
\c__msg_help_text_tl    7687   \\ \\
\c__msg_no_info_text_tl 7688 }
\c__msg_on_line_text_tl  7689 \tl_const:Nn \c__msg_continue_text_tl
\c__msg_return_text_tl   7690 { Type-<return>-to-continue }
\c__msg_trouble_text_tl  7691 \tl_const:Nn \c__msg_critical_text_tl
                          7692 { Reading-the-current-file~'\g_file_current_name_tl'~will-stop. }
                          7693 \tl_const:Nn \c__msg_fatal_text_tl
                          7694 { This-is-a-fatal-error:~LaTeX-will-abort. }
                          7695 \tl_const:Nn \c__msg_help_text_tl
                          7696 { For~immediate-help-type-H~<return> }
                          7697 \tl_const:Nn \c__msg_no_info_text_tl
                          7698 {
                          7699   LaTeX~does~not~know~anything~more~about~this~error,~sorry.
                          7700   \c__msg_return_text_tl
                          7701 }
                          7702 \tl_const:Nn \c__msg_on_line_text_tl { on-line }
                          7703 \tl_const:Nn \c__msg_return_text_tl
                          7704 {
                          7705   \\ \\
                          7706   Try~typing~<return>~to-proceed.
                          7707   \\
                          7708   If~that~doesn't~work,~type-X~<return>~to-quit.
                          7709 }
                          7710 \tl_const:Nn \c__msg_trouble_text_tl
                          7711 {
                          7712   \\ \\
                          7713   More-errors~will~almost~certainly~follow: \\
                          7714   the~LaTeX-run-should-be-aborted.
                          7715 }

```

(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.

```

7716 \cs_new_nopar:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
7717 \cs_gset_nopar:Npn \msg_line_context:

```

```

7718 {
7719   \c__msg_on_line_text_tl
7720   \c_space_tl
7721   \msg_line_number:
7722 }

```

(End definition for `\msg_line_number:` and `\msg_line_context:`. These functions are documented on page 147.)

18.3 Showing messages: low level mechanism

`\msg_interrupt:nnn` The low-level interruption macro is rather opaque, unfortunately. Depending on the availability of more information there is a choice of how to set up the further help. We feed the extra help text and the message itself to a wrapping auxiliary, in this order because we must first setup TeX's `\errhelp` register before issuing an `\errmessage`.

```

7723 \cs_new_protected:Npn \msg_interrupt:nnn #1#2#3
7724 {
7725   \tl_if_empty:nTF {#3}
7726   {
7727     \__msg_interrupt_wrap:nn { \ \ \c__msg_no_info_text_tl }
7728     {#1 \ \ \ \ #2 \ \ \ \ \c__msg_continue_text_tl }
7729   }
7730   {
7731     \__msg_interrupt_wrap:nn { \ \ #3 }
7732     {#1 \ \ \ \ #2 \ \ \ \ \c__msg_help_text_tl }
7733   }
7734 }

```

(End definition for `\msg_interrupt:nnn`. This function is documented on page 151.)

`__msg_interrupt_wrap:nn` First setup TeX's `\errhelp` register with the extra help #1, then build a nice-looking error message with #2. Everything is done using x-type expansion as the new line markers are different for the two type of text and need to be correctly set up. The auxiliary `__msg_interrupt_more_text:n` receives its argument as a line-wrapped string, which is thus unaffected by expansion.

```

7735 \cs_new_protected:Npn \__msg_interrupt_wrap:nn #1#2
7736 {
7737   \iow_wrap:nnnN {#1} { | ~ } { } \__msg_interrupt_more_text:n
7738   \iow_wrap:nnnN {#2} { ! ~ } { } \__msg_interrupt_text:n
7739 }
7740 \cs_new_protected:Npn \__msg_interrupt_more_text:n #1
7741 {
7742   \exp_args:Nx \tex_errhelp:D
7743   {
7744     |,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
7745     #1 \iow_newline:
7746     |.....
7747   }
7748 }

```

(End definition for `__msg_interrupt_wrap:nn`.)

`_msg_interrupt_text:n` The business end of the process starts by producing some visual separation of the message from the main part of the log. The error message needs to be printed with everything made “invisible”: T_EX’s own information involves the macro in which `\errmessage` is called, and the end of the argument of the `\errmessage`, including the closing brace. We use an active `!` to call the `\errmessage` primitive, and end its argument with `\use_none:n {⟨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.

```

7749 \group_begin:
7750   \char_set_lccode:nn {'\} {'\ }
7751   \char_set_lccode:nn {'\} {'\ }
7752   \char_set_lccode:nn {'&} {'\!}
7753   \char_set_catcode_active:N \&
7754   \tl_to_lowercase:n
7755   {
7756     \group_end:
7757     \cs_new_protected:Npn \_msg_interrupt_text:n #1
7758     {
7759       \iow_term:x
7760       {
7761         \iow_newline:
7762         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
7763         \iow_newline:
7764         !
7765       }
7766       \group_begin:
7767       \cs_set_protected_nopar:Npn &
7768       {
7769         \tex_errmessage:D
7770         {
7771           #1
7772           \use_none:n
7773           { ..... }
7774         }
7775       }
7776       \exp_after:wN
7777       \group_end:
7778       &
7779     }
7780   }

```

(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.

```

7781 \cs_new_protected:Npn \msg_log:n #1
7782 {
7783   \iow_log:n { ..... }

```

```

7784 \iow_wrap:nnnN { . ~ #1} { . ~ } { } \iow_log:n
7785 \iow_log:n { ..... }
7786 }
7787 \cs_new_protected:Npn \msg_term:n #1
7788 {
7789 \iow_term:n { ***** }
7790 \iow_wrap:nnnN { * ~ #1} { * ~ } { } \iow_term:n
7791 \iow_term:n { ***** }
7792 }

```

(End definition for `\msg_log:n`. This function is documented on page 152.)

18.4 Displaying messages

L^AT_EX is handling error messages and so the T_EX ones are disabled. This is already done by the L^AT_EX 2_ε kernel, so to avoid messing up any deliberate change by a user this is only set in format mode.

```

7793 <*initex>
7794 \int_gset_eq:NN \tex_errorcontextlines:D \c_minus_one
7795 </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

```

```

7796 \cs_new:Npn \msg_fatal_text:n #1 { Fatal~#1~error }
7797 \cs_new:Npn \msg_critical_text:n #1 { Critical~#1~error }
7798 \cs_new:Npn \msg_error_text:n #1 { #1~error }
7799 \cs_new:Npn \msg_warning_text:n #1 { #1~warning }
7800 \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 148.)

```

\msg_see_documentation_text:n Contextual footer information. The LATEX module only comprises LATEX3 code, so we
refer to the LATEX3 documentation rather than simply “LATEX”.

```

```

7801 \cs_new:Npn \msg_see_documentation_text:n #1
7802 {
7803 \\\ \ See~the~
7804 \str_if_eq:nnTF {#1} { LaTeX } { LaTeX3 } {#1} ~
7805 documentation~for~further~information.
7806 }

```

(End definition for `\msg_see_documentation_text:n`. This function is documented on page 148.)

```

\__msg_class_new:nn

```

```

7807 \group_begin:
7808 \cs_set_protected:Npn \__msg_class_new:nn #1#2
7809 {
7810 \prop_new:c { l__msg_redirect_ #1 _prop }
7811 \cs_new_protected:cpn { __msg_ #1 _code:nnnnnn } ##1##2##3##4##5##6 {#2}
7812 \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
7813 {
7814 \use:x
7815 {

```

```

7816         \exp_not:n { \_msg_use:nnnnnn {#1} {##1} {##2} }
7817         { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
7818         { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
7819     }
7820 }
7821 \cs_new_protected:cpx { msg_ #1 :nnnnn } ##1##2##3##4##5
7822 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
7823 \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4
7824 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
7825 \cs_new_protected:cpx { msg_ #1 :nnn } ##1##2##3
7826 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
7827 \cs_new_protected:cpx { msg_ #1 :nn } ##1##2
7828 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
7829 \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5##6
7830 {
7831     \use:x
7832     {
7833         \exp_not:N \exp_not:n
7834         { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
7835         {##3} {##4} {##5} {##6}
7836     }
7837 }
7838 \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
7839 { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
7840 \cs_new_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
7841 { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
7842 \cs_new_protected:cpx { msg_ #1 :nnx } ##1##2##3
7843 { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
7844 }

```

(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 7845 \_msg_class_new:nn { fatal }
\msg_fatal:nnnn 7846 {
\msg_fatal:nnnn 7847     \msg_interrupt:nnn
\msg_fatal:nn 7848     { \msg_fatal_text:n {#1} : ~ "#2" }
\msg_fatal:nnxxxx 7849     {
\msg_fatal:nnxxx 7850         \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
\msg_fatal:nnxx 7851         \msg_see_documentation_text:n {#1}
\msg_fatal:nnx 7852     }
7853     { \c__msg_fatal_text_tl }
7854     \tex_end:D
7855 }

```

(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 7856 \_msg_class_new:nn { critical }
\msg_critical:nnnn 7857 {
\msg_critical:nnnn 7858     \msg_interrupt:nnn
\msg_critical:nn
\msg_critical:nnxxxx
\msg_critical:nnxxx
\msg_critical:nnxx
\msg_critical:nnx

```



```

7859     { \msg_critical_text:n {#1} : ~ "#2" }
7860     {
7861       \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
7862       \msg_see_documentation_text:n {#1}
7863     }
7864     { \c__msg_critical_text_tl }
7865   \tex_endinput:D
7866 }

```

(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
\msg_error:nnnnnn comparing that control sequence with a permanently empty text.
\msg_error:nnnnn
\msg_error:nnnn 7867 \__msg_class_new:nn { error }
\msg_error:nnnn 7868 {
\msg_error:nnnn 7869   \__msg_error:cnnnnn
\msg_error:nnxxxx 7870   { \c__msg_more_text_prefix_tl #1 / #2 }
\msg_error:nnxxx 7871   {#3} {#4} {#5} {#6}
\msg_error:nnxx 7872   {
\msg_error:nnx 7873     \msg_interrupt:nnn
7874     { \msg_error_text:n {#1} : ~ "#2" }
7875     {
7876       \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
7877       \msg_see_documentation_text:n {#1}
7878     }
7879   }
7880 }
7881 \cs_new_protected:Npn \__msg_error:cnnnnn #1#2#3#4#5#6
7882 {
7883   \cs_if_eq:cNTF {#1} \__msg_no_more_text:nnnn
7884   { #6 { } }
7885   { #6 { \use:c {#1} {#2} {#3} {#4} {#5} } }
7886 }
7887 \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:nnnnn 7888 \__msg_class_new:nn { warning }
\msg_warning:nnnn 7889 {
\msg_warning:nnnn 7890   \msg_term:n
\msg_warning:nnnn 7891   {
\msg_warning:nnxxxx 7892     \msg_warning_text:n {#1} : ~ "#2" \\ \\
\msg_warning:nnxxx 7893     \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
\msg_warning:nnxx 7894     }
\msg_warning:nnx 7895   }

```

(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 7896 \__msg_class_new:nn { info }
\msg_info:nnnnnn 7897 {
\msg_info:nnnn
\msg_info:nnnn
\msg_info:nnnn
\msg_info:nnxxxx
\msg_info:nnxxx
\msg_info:nnxx
\msg_info:nnx

```

```

7898     \msg_log:n
7899     {
7900         \msg_info_text:n {#1} : ~ "#2" \\ \\
7901         \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
7902     }
7903 }

```

(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 7904 \__msg_class_new:nn { log }
\msg_log:nnnn 7905 {
\msg_log:nnnn 7906     \iow_wrap:nnnN
\msg_log:nn 7907     { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
\msg_log:nnxxxx 7908     { } { } \iow_log:n
\msg_log:nnxxx 7909 }

```

(End definition for \msg_log:nnnnnn and others. These functions are documented on page ??.)

\msg_log:nnx
\msg_none:nnnnnn The none message type is needed so that input can be gobbled.

```

\msg_none:nnnnnn 7910 \__msg_class_new:nn { none } { }
\msg_none:nnnn 7911 (End definition for \msg_none:nnnnnn and others. These functions are documented on page ??.)
\msg_none:nnnn
\msg_none:nnn
\msg_none:nn
7911 \group_end:
\msg_none:nnxxxx

```

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.

```

7912 \cs_new:Npn \__msg_class_chk_exist:nT #1
7913 {
7914     \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
7915     { \__msg_kernel_error:nnx { kernel } { message-class-unknown } {#1} }
7916 }

```

(End definition for __msg_class_chk_exist:nT.)

\l__msg_class_tl
\l__msg_current_class_tl Support variables needed for the redirection system.

```

7917 \tl_new:N \l__msg_class_tl
7918 \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

```

7919 \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 {}.

```

7920 \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.

```
7921 \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.

```
7922 \cs_new_protected:Npn \__msg_use:nnnnnnn #1#2#3#4#5#6#7
7923 {
7924   \msg_if_exist:nnTF {#2} {#3}
7925   {
7926     \__msg_class_chk_exist:nT {#1}
7927     {
7928       \tl_set:Nn \l__msg_current_class_tl {#1}
7929       \cs_set_protected_nopar:Npx \__msg_use_code:
7930       {
7931         \exp_not:n
7932         {
7933           \use:c { __msg_ \l__msg_class_tl _code:nnnnnnn }
7934           {#2} {#3} {#4} {#5} {#6} {#7}
7935         }
7936       }
7937       \__msg_use_redirect_name:n { #2 / #3 }
7938     }
7939   }
7940   { \__msg_kernel_error:nnxx { kernel } { message-unknown } {#2} {#3} }
7941 }
7942 \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.

```
7943 \cs_new_protected:Npn \__msg_use_redirect_name:n #1
7944 {
7945   \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
7946   { \__msg_use_code: }
7947   {
7948     \seq_clear:N \l__msg_hierarchy_seq
7949     \__msg_use_hierarchy:nwN { }
7950     #1 \q_mark \__msg_use_hierarchy:nwN
7951     / \q_mark \use_none_delimit_by_q_stop:w
7952     \q_stop
7953     \__msg_use_redirect_module:n { }
7954   }
7955 }
```

```

7956 \cs_new_protected:Npn \__msg_use_hierarchy:nwN #1#2 / #3 \q_mark #4
7957 {
7958   \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
7959   #4 { #1 / #2 } #3 \q_mark #4
7960 }

```

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`.

```

7961 \cs_new_protected:Npn \__msg_use_redirect_module:n #1
7962 {
7963   \seq_map_inline:Nn \l__msg_hierarchy_seq
7964   {
7965     \prop_get:cnTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
7966     {##1} \l__msg_class_tl
7967     {
7968       \seq_map_break:n
7969       {
7970         \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
7971         { \__msg_use_code: }
7972         {
7973           \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
7974           \__msg_use_redirect_module:n {##1}
7975         }
7976       }
7977     }
7978     {
7979       \str_if_eq:nnT {##1} {#1}
7980       {
7981         \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
7982         \seq_map_break:n { \__msg_use_code: }
7983       }
7984     }
7985   }
7986 }

```

(End definition for `__msg_use:nnnnnnn`.)

`\msg_redirect_name:nnn` Named message will always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```

7987 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
7988 {
7989   \tl_if_empty:nTF {#3}

```

```

7990     { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
7991     {
7992         \__msg_class_chk_exist:nT {#3}
7993         { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
7994     }
7995 }

```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page 151.)

```

\msg_redirect_class:nn If the target class is empty, eliminate the corresponding redirection. Otherwise, add the
\msg_redirect_module:nn redirection. We must then check for a loop: as an initialization, we start by storing the
\__msg_redirect:nnn initial class in \l__msg_current_class_tl.
\__msg_redirect_loop_chk:nnn
\__msg_redirect_loop_list:n
7996 \cs_new_protected_nopar:Npn \msg_redirect_class:nn
7997 { \__msg_redirect:nnn { } }
7998 \cs_new_protected:Npn \msg_redirect_module:nnn #1
7999 { \__msg_redirect:nnn { / #1 } }
8000 \cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
8001 {
8002     \__msg_class_chk_exist:nT {#2}
8003     {
8004         \tl_if_empty:nTF {#3}
8005         { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
8006         {
8007             \__msg_class_chk_exist:nT {#3}
8008             {
8009                 \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
8010                 \tl_set:Nn \l__msg_current_class_tl {#2}
8011                 \seq_clear:N \l__msg_class_loop_seq
8012                 \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
8013             }
8014         }
8015     }
8016 }

```

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.

```

8017 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
8018 {
8019     \seq_put_right:Nn \l__msg_class_loop_seq {#1}
8020     \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
8021     {

```

```

8022 \str_if_eq_x:nnF { \l__msg_class_tl } {#1}
8023 {
8024 \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
8025 {
8026 \prop_put:cnn { l__msg_redirect_ #2_prop } {#3} {#2}
8027 \__msg_kernel_warning:nnxxxx { kernel } { message-redirect-loop }
8028 { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
8029 { \seq_item:Nn \l__msg_class_loop_seq { \c_two } }
8030 {#3}
8031 {
8032 \seq_map_function:NN \l__msg_class_loop_seq
8033 \__msg_redirect_loop_list:n
8034 { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
8035 }
8036 }
8037 { \__msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
8038 }
8039 }
8040 }
8041 \cs_generate_variant:Nn \__msg_redirect_loop_chk:nnn { o }
8042 \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 151.)

18.5 Kernel-specific functions

```

\__msg_kernel_new:nnnn
\__msg_kernel_new:nnn
\__msg_kernel_set:nnnn
\__msg_kernel_set:nnn

```

The kernel needs some messages of its own. These are created using pre-built functions. Two functions are provided: one more general and one which only has the short text part.

```

8043 \cs_new_protected:Npn \__msg_kernel_new:nnnn #1#2
8044 { \msg_new:nnnn { LaTeX } { #1 / #2 } }
8045 \cs_new_protected:Npn \__msg_kernel_new:nnn #1#2
8046 { \msg_new:nnn { LaTeX } { #1 / #2 } }
8047 \cs_new_protected:Npn \__msg_kernel_set:nnnn #1#2
8048 { \msg_set:nnnn { LaTeX } { #1 / #2 } }
8049 \cs_new_protected:Npn \__msg_kernel_set:nnn #1#2
8050 { \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
\__msg_kernel_class_new_aux:nN

```

All the functions for kernel messages come in variants ranging from 0 to 4 arguments. Those with less than 4 arguments are defined in terms of the 4-argument variant, in a way very similar to `__msg_class_new:nn`. This auxiliary is destroyed at the end of the group.

```

8051 \group_begin:
8052 \cs_set_protected:Npn \__msg_kernel_class_new:nN #1
8053 { \__msg_kernel_class_new_aux:nN { kernel_ #1 } }
8054 \cs_set_protected:Npn \__msg_kernel_class_new_aux:nN #1#2
8055 {

```

```

8056 \cs_new_protected:cpn { __msg_ #1 :nnnnnn } ##1##2##3##4##5##6
8057 {
8058   \use:x
8059   {
8060     \exp_not:n { #2 { LaTeX } { ##1 / ##2 } }
8061     { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
8062     { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
8063   }
8064 }
8065 \cs_new_protected:cpx { __msg_ #1 :nnnnn } ##1##2##3##4##5
8066 { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
8067 \cs_new_protected:cpx { __msg_ #1 :nnnn } ##1##2##3##4
8068 { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
8069 \cs_new_protected:cpx { __msg_ #1 :nnn } ##1##2##3
8070 { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
8071 \cs_new_protected:cpx { __msg_ #1 :nn } ##1##2
8072 { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
8073 \cs_new_protected:cpx { __msg_ #1 :nnxxxx } ##1##2##3##4##5##6
8074 {
8075   \use:x
8076   {
8077     \exp_not:N \exp_not:n
8078     { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} }
8079     {##3} {##4} {##5} {##6}
8080   }
8081 }
8082 \cs_new_protected:cpx { __msg_ #1 :nnxxx } ##1##2##3##4##5
8083 { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
8084 \cs_new_protected:cpx { __msg_ #1 :nnxx } ##1##2##3##4
8085 { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
8086 \cs_new_protected:cpx { __msg_ #1 :nnx } ##1##2##3
8087 { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
8088 }

```

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.

(End definition for `_msg_kernel_fatal:nnnnnn` and others. These functions are documented on page ??.)

```
8094 \_msg_kernel_class_new:nN { warning } \msg_warning:nnxxxxx
8095 \_msg_kernel_class_new:nN { info } \msg_info:nnxxxxx
```

(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`.

8096 `\group_end:`

Error messages needed to actually implement the message system itself.

```
8097 \_msg\_kernel\_new:nnnn { kernel } { message-already-defined }
8098 { Message~'#2'~for~module~'#1'~already-defined. }
8099 {
8100   \c\_msg\_coding\_error\_text\_tl
8101   LaTeX~was~asked~to~define~a~new~message~called~'#2'\
8102   by~the~module~'#1':~this~message~already~exists.
8103   \c\_msg\_return\_text\_tl
8104 }
8105 \_msg\_kernel\_new:nnnn { kernel } { message-unknown }
8106 { Unknown~message~'#2'~for~module~'#1'. }
8107 {
8108   \c\_msg\_coding\_error\_text\_tl
8109   LaTeX~was~asked~to~display~a~message~called~'#2'\
8110   by~the~module~'#1':~this~message~does~not~exist.
8111   \c\_msg\_return\_text\_tl
8112 }
8113 \_msg\_kernel\_new:nnnn { kernel } { message-class-unknown }
8114 { Unknown~message~class~'#1'. }
8115 {
8116   LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\
8117   this~was~never~defined.
8118   \c\_msg\_return\_text\_tl
8119 }
8120 \_msg\_kernel\_new:nnnn { kernel } { message-redirect-loop }
8121 {
8122   Message~redirection~loop~caused~by~ {#1} ~>~ {#2}
8123   \tl\_if\_empty:nF {#3} { ~for~module~' \use\_none:n #3 ' } .
8124 }
8125 {
8126   Adding~the~message~redirection~ {#1} ~>~ {#2}
8127   \tl\_if\_empty:nF {#3} { ~for~the~module~' \use\_none:n #3 ' } ~
8128   created~an~infinite~loop\\\
8129   \iow\_indent:n { #4 \\\ }
8130 }
```

Messages for earlier kernel modules.

```
8131 \_msg\_kernel\_new:nnnn { kernel } { bad-number-of-arguments }
8132 { Function~'#1'~cannot~be~defined~with~#2~arguments. }
8133 {
8134   \c\_msg\_coding\_error\_text\_tl
8135   LaTeX~has~been~asked~to~define~a~function~'#1'~with~
8136   #2~arguments.~
8137   TeX~allows~between~0~and~9~arguments~for~a~single~function.
8138 }
```



```

8139 \_msg_kernel_new:nnnn { kernel } { command-already-defined }
8140 { Control~sequence~#1~already~defined. }
8141 {
8142   \c__msg_coding_error_text_tl
8143   LaTeX~has~been~asked~to~create~a~new~control~sequence~'~#1'~
8144   but~this~name~has~already~been~used~elsewhere. \\ \\
8145   The~current~meaning~is:\\
8146   \ \ #2
8147 }
8148 \_msg_kernel_new:nnnn { kernel } { command-not-defined }
8149 { Control~sequence~#1~undefined. }
8150 {
8151   \c__msg_coding_error_text_tl
8152   LaTeX~has~been~asked~to~use~a~command~#1,~but~this~has~not~
8153   been~defined~yet.
8154 }
8155 \_msg_kernel_new:nnnn { kernel } { empty-search-pattern }
8156 { Empty~search~pattern. }
8157 {
8158   \c__msg_coding_error_text_tl
8159   LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'~#1':~that~
8160   would~lead~to~an~infinite~loop!
8161 }
8162 \_msg_kernel_new:nnnn { kernel } { out-of-registers }
8163 { No~room~for~a~new~#1. }
8164 {
8165   TeX~only~supports~\int~use:N \c_max_register_int \
8166   of~each~type.~All~the~#1~registers~have~been~used.~
8167   This~run~will~be~aborted~now.
8168 }
8169 \_msg_kernel_new:nnnn { kernel } { missing-colon }
8170 { Function~'~#1'~contains~no~':'. }
8171 {
8172   \c__msg_coding_error_text_tl
8173   Code~level~functions~must~contain~':~'~to~separate~the~
8174   argument~specification~from~the~function~name.~This~is~
8175   needed~when~defining~conditionals~or~variants,~or~when~building~a~
8176   parameter~text~from~the~number~of~arguments~of~the~function.
8177 }
8178 \_msg_kernel_new:nnnn { kernel } { protected-predicate }
8179 { Predicate~'~#1'~must~be~expandable. }
8180 {
8181   \c__msg_coding_error_text_tl
8182   LaTeX~has~been~asked~to~define~'~#1'~as~a~protected~predicate.~
8183   Only~expandable~tests~can~have~a~predicate~version.
8184 }
8185 \_msg_kernel_new:nnnn { kernel } { conditional-form-unknown }
8186 { Conditional~form~'~#1'~for~function~'~#2'~unknown. }
8187 {
8188   \c__msg_coding_error_text_tl

```

```

8189 LaTeX-has-been-asked-to-define-the-conditional-form-~'~#1'~of~
8190 the~function~'~#2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.
8191 }
8192 \*package>
8193 \bool_if:NT \l@expl@check@declarations@bool
8194 {
8195   \__msg_kernel_new:nnnn { check } { non-declared-variable }
8196   { The~variable~#1~has~not~been~declared~\msg_line_context:. }
8197   {
8198     Checking-is-active,~and-you-have-tried-do-so-something-like: \\\
8199     \ \ \tl_set:Nn ~ #1 ~ \{ ~ ... ~ \} \\\
8200     without~first~having: \\\
8201     \ \ \tl_new:N ~ #1 \\\
8202     \\\
8203     LaTeX~will~create~the~variable~and~continue.
8204   }
8205 }
8206 \</package>
8207 \__msg_kernel_new:nnnn { kernel } { scanmark-already-defined }
8208 { Scan-mark~#1~already~defined. }
8209 {
8210   \c__msg_coding_error_text_tl
8211   LaTeX-has-been-asked-to-create-a-new-scan-mark-~'~#1'~
8212   but~this~name~has~already~been~used~for~a~scan~mark.
8213 }
8214 \__msg_kernel_new:nnnn { kernel } { variable-not-defined }
8215 { Variable~#1~undefined. }
8216 {
8217   \c__msg_coding_error_text_tl
8218   LaTeX-has-been-asked-to-show-a-variable~#1,~but~this~has~not~
8219   been~defined~yet.
8220 }
8221 \__msg_kernel_new:nnnn { kernel } { variant-too-long }
8222 { Variant~form~'~#1'~longer~than~base~signature~of~'~#2'. }
8223 {
8224   \c__msg_coding_error_text_tl
8225   LaTeX-has-been-asked-to-create-a-variant-of-the-function~'~#2'~
8226   with~a~signature~starting~with~'~#1',~but~that~is~longer~than~
8227   the~signature~(part~after~the~colon)~of~'~#2'.
8228 }
8229 \__msg_kernel_new:nnnn { kernel } { invalid-variant }
8230 { Variant~form~'~#1'~invalid~for~base~form~'~#2'. }
8231 {
8232   \c__msg_coding_error_text_tl
8233   LaTeX-has-been-asked-to-create-a-variant-of-the-function~'~#2'~
8234   with~a~signature~starting~with~'~#1',~but~cannot~change~an~argument~
8235   from~type~'~#3'~to~type~'~#4'.
8236 }

```

Some errors only appear in expandable settings, hence don't need a "more-text"

argument.

```

8237 \_msg_kernel_new:nnn { kernel } { bad-variable }
8238 { Erroneous~variable~#1 used! }
8239 \_msg_kernel_new:nnn { kernel } { misused-sequence }
8240 { A~sequence~was~misused. }
8241 \_msg_kernel_new:nnn { kernel } { misused-prop }
8242 { A~property~list~was~misused. }
8243 \_msg_kernel_new:nnn { kernel } { negative-replication }
8244 { Negative~argument~for~\prg_replicate:nn. }
8245 \_msg_kernel_new:nnn { kernel } { unknown-comparison }
8246 { Relation~'~#1'~unknown:~use~=,~<,~>,~==,~!=,~<=,~>=. }
8247 \_msg_kernel_new:nnn { kernel } { zero-step }
8248 { Zero~step~size~for~step~function~#1. }

```

Messages used by the “show” functions.

```

8249 \_msg_kernel_new:nnn { kernel } { show-clist }
8250 {
8251   The~comma~list~
8252   \str_if_eq:nnF {#1} { \l__clist_internal_clist } { \token_to_str:N #1~}
8253   \clist_if_empty:NTF #1
8254   { is~empty }
8255   { contains~the~items~(without~outer~braces): }
8256 }
8257 \_msg_kernel_new:nnn { kernel } { show-prop }
8258 {
8259   The~property~list~\token_to_str:N #1~
8260   \prop_if_empty:NTF #1
8261   { is~empty }
8262   { contains~the~pairs~(without~outer~braces): }
8263 }
8264 \_msg_kernel_new:nnn { kernel } { show-seq }
8265 {
8266   The~sequence~\token_to_str:N #1~
8267   \seq_if_empty:NTF #1
8268   { is~empty }
8269   { contains~the~items~(without~outer~braces): }
8270 }
8271 \_msg_kernel_new:nnn { kernel } { show-no-stream }
8272 { No~ #1 ~streams~are~open }
8273 \_msg_kernel_new:nnn { kernel } { show-open-streams }
8274 { The~following~ #1 ~streams~are~in~use: }

```

18.6 Expandable errors

`_msg_expandable_error:n` In expansion only context, we cannot use the normal means of reporting errors. Instead, we feed T_EX 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`.

```

8275 \group_begin:
8276 \char_set_catcode_math_superscript:N \^
8277 \char_set_lccode:nn { '^ } { '\ }
8278 \char_set_lccode:nn { 'L } { 'L }
8279 \char_set_lccode:nn { 'T } { 'T }
8280 \char_set_lccode:nn { 'X } { 'X }
8281 \tl_to_lowercase:n
8282 {
8283   \cs_new:Npx \__msg_expandable_error:n #1
8284   {
8285     \exp_not:n
8286     {
8287       \tex_romannumeral:D
8288       \exp_after:wN \exp_after:wN
8289       \exp_after:wN \__msg_expandable_error:w
8290       \exp_after:wN \exp_after:wN
8291       \exp_after:wN \c_zero
8292     }
8293     \exp_not:N \use:n { \exp_not:c { LaTeX3~error: } ^ #1 } ^
8294   }
8295   \cs_new:Npn \__msg_expandable_error:w #1 ^ #2 ^ { #1 }
8296 }
8297 \group_end:
(End definition for \__msg_expandable_error:n.)

```

```

\__msg_kernel_expandable_error:nnnnnn
\__msg_kernel_expandable_error:nnnnn
\__msg_kernel_expandable_error:nnnn
\__msg_kernel_expandable_error:nnn
\__msg_kernel_expandable_error:nn

```

The command built from the csname `\c_@@_text_prefix_tl LaTeX / #1 / #2` takes four arguments and builds the error text, which is fed to `__msg_expandable_error:n`.

```

8298 \cs_new:Npn \__msg_kernel_expandable_error:nnnnnn #1#2#3#4#5#6
8299 {
8300   \exp_args:Nf \__msg_expandable_error:n
8301   {
8302     \exp_args:NNc \exp_after:wN \exp_stop_f:
8303     { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
8304     {#3} {#4} {#5} {#6}
8305   }
8306 }
8307 \cs_new:Npn \__msg_kernel_expandable_error:nnnnn #1#2#3#4#5
8308 {
8309   \__msg_kernel_expandable_error:nnnnnn
8310   {#1} {#2} {#3} {#4} {#5} { }
8311 }
8312 \cs_new:Npn \__msg_kernel_expandable_error:nnnn #1#2#3#4
8313 {

```

```

8314     \_msg_kernel_expandable_error:nnnnnn
8315     {#1} {#2} {#3} {#4} { } { }
8316 }
8317 \cs_new:Npn \_msg_kernel_expandable_error:nnn #1#2#3
8318 {
8319     \_msg_kernel_expandable_error:nnnnnn
8320     {#1} {#2} {#3} { } { } { }
8321 }
8322 \cs_new:Npn \_msg_kernel_expandable_error:nn #1#2
8323 {
8324     \_msg_kernel_expandable_error:nnnnnn
8325     {#1} {#2} { } { } { } { }
8326 }

```

(End definition for `_msg_kernel_expandable_error:nnnnnn` and others. These functions are documented on page ??.)

18.7 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_wrap:nnnN`.

```

\_msg_term:nnnnnnV
\_msg_term:nnnnnn
\_msg_term:nnnn
\_msg_term:nnnn
\_msg_term:nnnn

```

```

8327 \cs_new_protected:Npn \_msg_term:nnnnnn #1#2#3#4#5#6
8328 {
8329     \iow_wrap:nnnN
8330     { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
8331     { } { } \iow_term:n
8332 }
8333 \cs_generate_variant:Nn \_msg_term:nnnnnn { nnnnnV }
8334 \cs_new_protected:Npn \_msg_term:nnnnnn #1#2#3#4#5
8335 { \_msg_term:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
8336 \cs_new_protected:Npn \_msg_term:nnnn #1#2#3
8337 { \_msg_term:nnnnnn {#1} {#2} {#3} { } { } { } }
8338 \cs_new_protected:Npn \_msg_term:nn #1#2
8339 { \_msg_term:nnnnnn {#1} {#2} { } { } { } { } }

```

(End definition for `_msg_term:nnnnnn` and `_msg_term:nnnnnV`.)

```

\_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.

```

8340 \cs_new_protected:Npn \__msg_show_variable:Nnn #1#2#3
8341 {
8342   \cs_if_exist:NTF #1
8343   {
8344     \__msg_term:nnn { LaTeX / kernel } { show- #2 } {#1}
8345     \__msg_show_variable:n {#3}
8346   }
8347   {
8348     \__msg_kernel_error:nnx { kernel } { variable-not-defined }
8349     { \token_to_str:N #1 }
8350   }
8351 }
8352 \cs_new_protected:Npn \__msg_show_variable:n #1
8353 { \iow_wrap:nnnN {#1} { } { } \__msg_show_variable_aux:n }
8354 \cs_new_protected:Npn \__msg_show_variable_aux:n #1
8355 {
8356   \tl_if_empty:nTF {#1}
8357   { \tl_clear:N \l__msg_internal_tl }
8358   { \tl_set:Nf \l__msg_internal_tl { \__msg_show_variable_aux:w #1 } }
8359   \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
8360   { \exp_after:wN \l__msg_internal_tl }
8361 }
8362 \cs_new:Npn \__msg_show_variable_aux:w #1 > { }
(End definition for \__msg_show_variable:Nnn.)

```

`__msg_show_item:n` Each item in the variable is formatted using one of the following functions.

`__msg_show_item:nn`

`__msg_show_item_unbraced:nn`

```

8363 \cs_new:Npn \__msg_show_item:n #1
8364 {
8365   \> \ \ \{ \tl_to_str:n {#1} \}
8366 }
8367 \cs_new:Npn \__msg_show_item:nn #1#2
8368 {
8369   \> \ \ \{ \tl_to_str:n {#1} \}
8370   \ \ => \ \ \{ \tl_to_str:n {#2} \}
8371 }
8372 \cs_new:Npn \__msg_show_item_unbraced:nn #1#2
8373 {
8374   \> \ \ \tl_to_str:n {#1}
8375   \ \ => \ \ \tl_to_str:n {#2}
8376 }

```

(End definition for _msg_show_item:n.)

8377 </initex | package>

19 l3keys Implementation

8378 <*initex | package>

19.1 Low-level interface

8379 <@@=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.

8380 \int_new:N \g__keyval_level_int

(End definition for \g__keyval_level_int. This variable is documented on page ??.)

\l__keyval_key_tl The current key name and value.

\l__keyval_value_tl 8381 \tl_new:N \l__keyval_key_tl

8382 \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 8383 \tl_new:N \l__keyval_sanitise_tl

8384 \tl_new:N \l__keyval_parse_tl

(End definition for \l__keyval_sanitise_tl. This variable 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.

8385 \group_begin:

8386 \char_set_catcode_active:n { ‘\= }

8387 \char_set_catcode_active:n { ‘\, }

8388 \char_set_lccode:nn { ‘\8 } { ‘\= }

8389 \char_set_lccode:nn { ‘\9 } { ‘\, }

8390 \tl_to_lowercase:n

8391 {

8392 \group_end:

8393 \cs_new_protected:Npn __keyval_parse:n #1

8394 {

8395 \group_begin:

8396 \tl_set:Nn \l__keyval_sanitise_tl {#1}

8397 \tl_replace_all:Nnn \l__keyval_sanitise_tl { = } { 8 }

8398 \tl_replace_all:Nnn \l__keyval_sanitise_tl { , } { 9 }

8399 \tl_clear:N \l__keyval_parse_tl

8400 \exp_after:wN __keyval_parse_elt:w \exp_after:wN

8401 \q_nil \l__keyval_sanitise_tl 9 \q_recursion_tail 9 \q_recursion_stop

8402 \exp_after:wN \group_end:

8403 \l__keyval_parse_tl

8404 }

8405 }

(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. To allow rapid matching for an `=` while not stripping any braces, another `\q_nil` needed before the next phase of the parser. Finally, loop around for the next item, adding in the `\q_nil`: this happens whatever the nature of the current argument as the end-of-recursion will clear up in all cases.

```

8406 \cs_new_protected:Npn \__keyval_parse_elt:w #1 ,
8407 {
8408   \tl_if_blank:oF { \use_none:n #1 }
8409   {
8410     \quark_if_recursion_tail_stop:o { \use_none:n #1 }
8411     \__keyval_split_key_value:w #1 \q_nil = = \q_stop
8412   }
8413   \__keyval_parse_elt:w \q_nil
8414 }

```

(End definition for `__keyval_parse_elt:w`. This function is documented on page ??.)

`__keyval_split_key_value:w` Split the key and value using a delimited argument. The `\q_nil` values added earlier ensure that no braces will be stripped as part of this process. A blank test can then be used on `#3`: it is only empty if there was no `=` in the original input. In that case, strip a `\q_nil` from the end of the key name then hand on to remove other things and store as `\l__keyval_key_tl` before adding to the output token list. In the case where there is an `=`, first tidy up the key, this time without a trailing `\q_nil`, then do a check to ensure that `#3` is exactly one token (`=`). With that done, the final stage is to hand off to tidy up the value.

```

8415 \cs_new_protected:Npn \__keyval_split_key_value:w #1 = #2 = #3 \q_stop
8416 {
8417   \tl_if_blank:nTF {#3}
8418   {
8419     \__keyval_split_key:w #1 \q_stop
8420     \tl_put_right:Nx \l__keyval_parse_tl
8421     {
8422       \exp_not:c
8423       { __keyval_key_no_value_elt_ \int_use:N \g__keyval_level_int :n }
8424       { \exp_not:o \l__keyval_key_tl }
8425     }
8426   }
8427   {
8428     \__keyval_split:Nn \l__keyval_key_tl {#1}
8429     \tl_if_blank:oTF { \use_none:n #3 }
8430     { \__keyval_split_value:w \q_nil #2 \q_stop }
8431     { \__msg_kernel_error:nn { kernel } { misplaced-equals-sign } }
8432   }
8433 }
8434 \cs_new_protected:Npn \__keyval_split_key:w #1 \q_nil \q_stop
8435 { \__keyval_split:Nn \l__keyval_key_tl {#1} }

```

(End definition for `__keyval_split_key_value:w`. This function is documented on page ??.)

`__keyval_split:Nn` There are two possible cases here. The first case is that `#1` is surrounded by braces, in which case the `\use_none:nnn #1 \q_nil \q_nil` will yield `\q_nil`. There, we can remove the leading `\q_nil`, the braces and any spaces around the outside with `\use_ii:nnn`. On the other hand, if there are no braces then the second branch removes the leading `\q_nil` and any surrounding spaces.

```

8436 \cs_new_protected:Npn \__keyval_split:Nn #1#2
8437 {
8438   \quark_if_nil:oTF { \use_none:nnn #2 \q_nil \q_nil }
8439   { \tl_set:Nx #1 { \exp_not:o { \use_ii:nnn #2 \q_nil } } }
8440   { \__keyval_split:Nw #1 #2 \q_stop }
8441 }
8442 \cs_new_protected:Npn \__keyval_split:Nw #1 \q_nil #2 \q_stop
8443 { \tl_set:Nx #1 { \tl_trim_spaces:n {#2} } }
(End definition for \__keyval_split:Nn. This function is documented on page ??.)

```

`__keyval_split_value:w` As this stage there is just the value to deal with. The leading and trailing `\q_nil` tokens are removed in two steps before storing the value with spaces stripped (see `__keyval_split:Nn`). Doing the storage of key and value in one shot will put exactly the right number of brace groups into the output.

```

8444 \cs_new_protected:Npn \__keyval_split_value:w #1 \q_nil \q_stop
8445 {
8446   \__keyval_split:Nn \l__keyval_value_tl {#1}
8447   \tl_put_right:Nx \l__keyval_parse_tl
8448   {
8449     \exp_not:c
8450     { __keyval_key_value_elt_ \int_use:N \g__keyval_level_int :nn }
8451     { \exp_not:o \l__keyval_key_tl }
8452     { \exp_not:o \l__keyval_value_tl }
8453   }
8454 }
(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.

```

8455 \cs_new_protected:Npn \keyval_parse:NNn #1#2#3
8456 {
8457   \int_gincr:N \g__keyval_level_int
8458   \cs_gset_eq:cN
8459   { __keyval_key_no_value_elt_ \int_use:N \g__keyval_level_int :n } #1
8460   \cs_gset_eq:cN
8461   { __keyval_key_value_elt_ \int_use:N \g__keyval_level_int :nn } #2
8462   \__keyval_parse:n {#3}
8463   \int_gdecr:N \g__keyval_level_int
8464 }
(End definition for \keyval_parse:NNn. This function is documented on page 167.)

```

One message for the low level parsing system.

```

8465 \__msg_kernel_new:nnnn { kernel } { misplaced-equals-sign }
8466 { Misplaced-equals-sign-in-key-value-input~\msg_line_number: }
8467 {

```

```

8468 LaTeX-is-attempting-to-parse-some-key-value-input-but-found~
8469 two~equals~signs~not~separated~by~a~comma.
8470 }

```

19.2 Constants and variables

```

8471 <@@=keys>

```

`\c__keys_code_root_tl` The prefixes for the code and variables of the keys themselves.

```

8472 \tl_const:Nn \c__keys_code_root_tl { key~code~>~ }

```

```

8473 \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.

```

8474 \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`

```

8475 \int_new:N \l_keys_choice_int

```

```

8476 \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 162.)

`\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.

```

8477 \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.

```

8478 \tl_new:N \l_keys_key_tl

```

(End definition for `\l_keys_key_tl`. This variable is documented on page 164.)

`\l__keys_module_tl` The module for an entire set of keys.

```

8479 \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.

```

8480 \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.

```

8481 \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>8482 \tl_new:N \l_keys_path_tl</code> <i>(End definition for \l_keys_path_tl. This variable is documented on page 164.)</i>
<code>\l__keys_property_tl</code>	The “property” begin set for a key at definition time is stored here. <code>8483 \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>8484 \bool_new:N \l_keys_selective_bool</code> <code>8485 \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>8486 \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>8487 \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>8488 \tl_new:N \l_keys_value_tl</code> <i>(End definition for \l_keys_value_tl. This variable is documented on page 164.)</i>
<code>\l__keys_tmp_bool</code>	Scratch space. <code>8489 \bool_new:N \l__keys_tmp_bool</code> <i>(End definition for \l__keys_tmp_bool. This variable is documented on page ??.)</i>

19.3 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>8490 \cs_new_protected:Npn \keys_define:nn</code> <code>8491 { __keys_define:onn \l__keys_module_tl }</code> <code>8492 \cs_new_protected:Npn __keys_define:nnn #1#2#3</code> <code>8493 {</code> <code>8494 \tl_set:Nx \l__keys_module_tl { \tl_to_str:n {#2} }</code> <code>8495 \keyval_parse:NNn __keys_define_elt:n __keys_define_elt:nn {#3}</code> <code>8496 \tl_set:Nn \l__keys_module_tl {#1}</code> <code>8497 }</code> <code>8498 \cs_generate_variant:Nn __keys_define:nnn { o }</code> <i>(End definition for \keys_define:nn. This function is documented on page 157.)</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.

```

8499 \cs_new_protected:Npn __keys_define_elt:n #1
8500 {
8501   \bool_set_true:N \l__keys_no_value_bool
8502   __keys_define_elt_aux:nn {#1} { }
8503 }
8504 \cs_new_protected:Npn __keys_define_elt:nn #1#2
8505 {
8506   \bool_set_false:N \l__keys_no_value_bool
8507   __keys_define_elt_aux:nn {#1} {#2}
8508 }
8509 \cs_new_protected:Npn __keys_define_elt_aux:nn #1#2
8510 {
8511   __keys_property_find:n {#1}
8512   \cs_if_exist:cTF { \c__keys_props_root_tl \l__keys_property_tl }
8513     { __keys_define_key:n {#2} }
8514     {
8515       \str_if_eq:x:nnF { \l__keys_property_tl } { .abort: }
8516       {
8517         __msg_kernel_error:nnxx { kernel } { property-unknown }
8518         { \l__keys_property_tl } { \l_keys_path_tl }
8519       }
8520     }
8521 }

```

(End definition for `__keys_define_elt:n`.)

`__keys_property_find:n` Searching for a property means finding the last . in the input, and storing the text before
`__keys_property_find:w` and after it. Everything is turned into strings, so there is no problem using an x-type
 expansion.

```

8522 \cs_new_protected:Npn __keys_property_find:n #1
8523 {
8524   \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / }
8525   \tl_if_in:nnTF {#1} { . }
8526     { __keys_property_find:w #1 \q_stop }
8527     {
8528       __msg_kernel_error:nnx { kernel } { key-no-property } {#1}
8529       \tl_set:Nn \l__keys_property_tl { .abort: }
8530     }
8531 }
8532 \cs_new_protected:Npn __keys_property_find:w #1 . #2 \q_stop
8533 {
8534   \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl \tl_to_str:n {#1} }
8535   \tl_if_in:nnTF {#2} { . }
8536     {
8537       \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl . }
8538       __keys_property_find:w #2 \q_stop
8539     }

```

```

8540         { \tl_set:Nn \l__keys_property_tl { . #2 } }
8541     }

```

(End definition for __keys_property_find:n.)

__keys_define_key:n
__keys_define_key:w

Two possible cases. If there is a value for the key, then just use the function. If not, then a check to make sure there is no need for a value with the property. If there should be one then complain, otherwise execute it. There is no need to check for a : as if it is missing the earlier tests will have failed.

```

8542 \cs_new_protected:Npn \__keys_define_key:n #1
8543 {
8544     \bool_if:NTF \l__keys_no_value_bool
8545     {
8546         \exp_after:wN \__keys_define_key:w
8547         \l__keys_property_tl \q_stop
8548         { \use:c { \c__keys_props_root_tl \l__keys_property_tl } }
8549         {
8550             \__msg_kernel_error:nnxx { kernel }
8551             { property-requires-value } { \l__keys_property_tl }
8552             { \l_keys_path_tl }
8553         }
8554     }
8555     { \use:c { \c__keys_props_root_tl \l__keys_property_tl } {#1} }
8556 }
8557 \cs_new_protected:Npn \__keys_define_key:w #1 : #2 \q_stop
8558 { \tl_if_empty:NTF {#2} }

```

(End definition for __keys_define_key:n.)

19.4 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.

```

8559 \cs_new_protected:Npn \__keys_bool_set:Nn #1#2
8560 {
8561     \bool_if_exist:NF #1 { \bool_new:N #1 }
8562     \__keys_choice_make:
8563     \__keys_cmd_set:nx { \l_keys_path_tl / true }
8564     { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
8565     \__keys_cmd_set:nx { \l_keys_path_tl / false }
8566     { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
8567     \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
8568     {
8569         \__msg_kernel_error:nnx { kernel } { boolean-values-only }
8570         { \l_keys_key_tl }
8571     }
8572     \__keys_default_set:n { true }
8573 }
8574 \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

```

8575 \\cs_new_protected:Npn \\_keys_bool_set_inverse:Nn #1#2
8576 {
8577   \\bool_if_exist:NF #1 { \\bool_new:N #1 }
8578   \\_keys_choice_make:
8579   \\_keys_cmd_set:nx { \\l_keys_path_tl / true }
8580   { \\exp_not:c { bool_ #2 set_false:N } \\exp_not:N #1 }
8581   \\_keys_cmd_set:nx { \\l_keys_path_tl / false }
8582   { \\exp_not:c { bool_ #2 set_true:N } \\exp_not:N #1 }
8583   \\_keys_cmd_set:nn { \\l_keys_path_tl / unknown }
8584   {
8585     \\_msg_kernel_error:nxx { kernel } { boolean-values-only }
8586     { \\l_keys_key_tl }
8587   }
8588   \\_keys_default_set:n { true }
8589 }
8590 \\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:
 _keys_multichoice_make:
 _keys_choice_make:N
 _keys_choice_make_aux:N
 _keys_parent:n
 _keys_parent:o
 _keys_parent:wn

To make a choice from a key, two steps: set the code, and set the unknown key. There is one point to watch here: choice keys cannot be nested! As multichoices and choices are essentially the same bar one function, the code is given together.

```

8591 \\cs_new_protected_nopar:Npn \\_keys_choice_make:
8592 { \\_keys_choice_make:N \\_keys_choice_find:n }
8593 \\cs_new_protected_nopar:Npn \\_keys_multichoice_make:
8594 { \\_keys_choice_make:N \\_keys_multichoice_find:n }
8595 \\cs_new_protected_nopar:Npn \\_keys_choice_make:N #1
8596 {
8597   \\prop_if_exist:cTF { \\c__keys_info_root_tl \\_keys_parent:o \\l_keys_path_tl }
8598   {
8599     \\prop_get:cnNTF { \\c__keys_info_root_tl \\_keys_parent:o \\l_keys_path_tl }
8600     { choice } \\l_keys_value_tl
8601     {
8602       \\_msg_kernel_error:nxxx { kernel } { nested-choice-key }
8603       { \\l_keys_path_tl } { \\_keys_parent:o \\l_keys_path_tl }
8604     }
8605     { \\_keys_choice_make_aux:N #1 }
8606   }
8607   { \\_keys_choice_make_aux:N #1 }
8608 }
8609 \\cs_new_protected_nopar:Npn \\_keys_choice_make_aux:N #1
8610 {
8611   \\_keys_cmd_set:nn { \\l_keys_path_tl } { #1 {##1} }
8612   \\prop_put:cn { \\c__keys_info_root_tl \\l_keys_path_tl } { choice }
8613   { true }
8614   \\_keys_cmd_set:nn { \\l_keys_path_tl / unknown }
8615   {
8616     \\_msg_kernel_error:nxxx { kernel } { key-choice-unknown }
8617     { \\l_keys_path_tl } {##1}

```

```

8618     }
8619   }
8620   \cs_new:Npn \__keys_parent:n #1
8621   { \__keys_parent:wn #1 / / \q_stop { } }
8622   \cs_generate_variant:Nn \__keys_parent:n { o }
8623   \cs_new:Npn \__keys_parent:wn #1 / #2 / #3 \q_stop #4
8624   {
8625     \tl_if_blank:nTF {#2}
8626     { \use_none:n #4 }
8627     {
8628       \__keys_parent:wn #2 / #3 \q_stop { #4 / #1 }
8629     }
8630   }

```

(End definition for __keys_choice_make: and __keys_multichoice_make:.)

__keys_choices_make:nn Auto-generating choices means setting up the root key as a choice, then defining each
 __keys_multichoices_make:nn choice in turn.

```

\__keys_choices_make:Nnn
8631 \cs_new_protected_nopar:Npn \__keys_choices_make:nn
8632 { \__keys_choices_make:Nnn \__keys_choice_make: }
8633 \cs_new_protected_nopar:Npn \__keys_multichoices_make:nn
8634 { \__keys_choices_make:Nnn \__keys_multichoice_make: }
8635 \cs_new_protected:Npn \__keys_choices_make:Nnn #1#2#3
8636 {
8637   #1
8638   \int_zero:N \l_keys_choice_int
8639   \clist_map_inline:nn {#2}
8640   {
8641     \int_incr:N \l_keys_choice_int
8642     \__keys_cmd_set:nx { \l_keys_path_tl / ##1 }
8643     {
8644       \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
8645       \int_set:Nn \exp_not:N \l_keys_choice_int
8646       { \int_use:N \l_keys_choice_int }
8647       \exp_not:n {#3}
8648     }
8649   }
8650 }

```

(End definition for __keys_choices_make:nn and __keys_multichoices_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
 __keys_cmd_set:Vo

```

8651 \cs_new_protected:Npn \__keys_cmd_set:nn #1#2
8652 {
8653   \prop_clear_new:c { \c__keys_info_root_tl #1 }
8654   \cs_set:cpn { \c__keys_code_root_tl #1 } ##1 {#2}
8655 }
8656 \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.

```
8657 \cs_new_protected:Npn \__keys_default_set:n #1
8658 {
8659   \prop_if_exist:cT { \c__keys_info_root_tl \l_keys_path_tl }
8660   { \prop_put:cnn { \c__keys_info_root_tl \l_keys_path_tl } { default } {#1} }
8661 }
```

(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.

```
8662 \cs_new_protected:Npn \__keys_groups_set:n #1
8663 {
8664   \prop_if_exist:cT { \c__keys_info_root_tl \l_keys_path_tl }
8665   {
8666     \clist_set:Nn \l__keys_groups_clist {#1}
8667     \prop_put:cnV { \c__keys_info_root_tl \l_keys_path_tl }
8668     { groups } \l__keys_groups_clist
8669   }
8670 }
```

(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.

```
8671 \cs_new_protected:Npn \__keys_initialise:n #1
8672 { \exp_after:wN \__keys_initialise:wn \l_keys_path_tl \q_stop {#1} }
8673 \cs_new_protected:Npn \__keys_initialise:wn #1 / #2 \q_stop #3
8674 { \keys_set:nn {#1} { #2 = {#3} } }
```

(End definition for __keys_initialise:n.)

`__keys_meta_make:n` To create a meta-key, simply set up to pass data through.

```
\__keys_meta_make:nn
8675 \cs_new_protected:Npn \__keys_meta_make:n #1
8676 {
8677   \__keys_cmd_set:Vo \l_keys_path_tl
8678   { \exp_after:wN \keys_set:nn \exp_after:wN { \l__keys_module_tl } {#1} }
8679 }
8680 \cs_new_protected:Npn \__keys_meta_make:nn #1#2
8681 { \__keys_cmd_set:Vn \l_keys_path_tl { \keys_set:nn {#1} {#2} } }
```

(End definition for __keys_meta_make:n.)

`__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!

```
8682 \cs_new_protected:Npn \__keys_value_requirement:n #1
8683 {
8684   \prop_if_exist:cT { \c__keys_info_root_tl \l_keys_path_tl }
8685   {
8686     \prop_remove:cn { \c__keys_info_root_tl \l_keys_path_tl } { required }
8687     \prop_remove:cn { \c__keys_info_root_tl \l_keys_path_tl } { forbidden }
```



```

8688         \prop_put:cnn { \c__keys_info_root_tl \l_keys_path_tl } {#1} { true }
8689     }
8690 }
(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 variable if needed.

```

8691 \cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
8692 {
8693     \use:c { #2_if_exist:Nf } #1 { \use:c { #2_new:N } #1 }
8694     \__keys_cmd_set:nx { \l_keys_path_tl }
8695     { \exp_not:c { #2 _ #3 set:N #4 } \exp_not:N #1 \exp_not:n { {##1} } }
8696 }
8697 \cs_generate_variant:Nn \__keys_variable_set:NnnN { c }
(End definition for \__keys_variable_set:NnnN and \__keys_variable_set:cnnN.)

```

19.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

Importantly, while key properties have “normal” argument specs, the underlying code always supplies one braced argument to these. As such, argument expansion is handled by hand rather than using the standard tools. This shows up particularly for the two-argument properties, where things would otherwise go badly wrong.

`.bool_set:N` One function for this.

```

8698 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:N } #1
8699 { \__keys_bool_set:Nn #1 { } }
8700 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:c } #1
8701 { \__keys_bool_set:cn {#1} { } }
8702 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:N } #1
8703 { \__keys_bool_set:Nn #1 { g } }
8704 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:c } #1
8705 { \__keys_bool_set:cn {#1} { g } }
(End definition for .bool_set:N and .bool_set:c. These functions are documented on page 158.)

```

`.bool_set_inverse:N` One function for this.

```

8706 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:N } #1
8707 { \__keys_bool_set_inverse:Nn #1 { } }
8708 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:c } #1
8709 { \__keys_bool_set_inverse:cn {#1} { } }
8710 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:N } #1
8711 { \__keys_bool_set_inverse:Nn #1 { g } }
8712 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:c } #1
8713 { \__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 158.)

```

.choice: Making a choice is handled internally, as it is also needed by **.generate_choices:n**.

```

8714 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .choice: }
8715 { \__keys_choice_make: }

```

(End definition for **.choice:**. This function is documented on page 158.)

.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

```

8716 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:nn } #1
8717 { \__keys_choices_make:nn #1 }
8718 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:Vn } #1
8719 { \exp_args:NV \__keys_choices_make:nn #1 }
8720 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:on } #1
8721 { \exp_args:No \__keys_choices_make:nn #1 }
8722 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:xn } #1
8723 { \exp_args:Nx \__keys_choices_make:nn #1 }

```

(End definition for **.choices:nn** and others. These functions are documented on page 158.)

.code:n Creating code is simply a case of passing through to the underlying set function.

```

8724 \cs_new_protected:cpn { \c__keys_props_root_tl .code:n } #1
8725 { \__keys_cmd_set:nn { \l_keys_path_tl } {#1} }

```

(End definition for **.code:n**. This function is documented on page 158.)

.clist_set:N

.clist_set:c

.clist_gset:N

.clist_gset:c

```

8726 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:N } #1
8727 { \__keys_variable_set:NnnN #1 { clist } { } n }
8728 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:c } #1
8729 { \__keys_variable_set:cnnN {#1} { clist } { } n }
8730 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:N } #1
8731 { \__keys_variable_set:NnnN #1 { clist } { g } n }
8732 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:c } #1
8733 { \__keys_variable_set:cnnN {#1} { clist } { g } n }

```

(End definition for **.clist_set:N** and **.clist_set:c**. These functions are documented on page 158.)

.default:n Expansion is left to the internal functions.

.default:V

.default:o

.default:x

```

8734 \cs_new_protected:cpn { \c__keys_props_root_tl .default:n } #1
8735 { \__keys_default_set:n {#1} }
8736 \cs_new_protected:cpn { \c__keys_props_root_tl .default:V } #1
8737 { \exp_args:NV \__keys_default_set:n #1 }
8738 \cs_new_protected:cpn { \c__keys_props_root_tl .default:o } #1
8739 { \exp_args:No \__keys_default_set:n {#1} }
8740 \cs_new_protected:cpn { \c__keys_props_root_tl .default:x } #1
8741 { \exp_args:Nx \__keys_default_set:n {#1} }

```

(End definition for **.default:n** and others. These functions are documented on page 159.)

```

.dim_set:N Setting a variable is very easy: just pass the data along.
.dim_set:c 8742 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:N } #1
            { \__keys_variable_set:NnnN #1 { dim } { } n }
.dim_gset:N 8743 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:c } #1
            { \__keys_variable_set:cnnN {#1} { dim } { } n }
.dim_gset:c 8744 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:N } #1
            { \__keys_variable_set:NnnN #1 { dim } { g } n }
            8745 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
            { \__keys_variable_set:cnnN {#1} { dim } { g } n }
            8746 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
            { \__keys_variable_set:cnnN {#1} { dim } { g } n }
            8747 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
            { \__keys_variable_set:cnnN {#1} { dim } { g } n }
            8748 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
            { \__keys_variable_set:cnnN {#1} { dim } { g } n }
            8749 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
            { \__keys_variable_set:cnnN {#1} { dim } { g } n }
(End definition for .dim_set:N and .dim_set:c. These functions are documented on page 159.)

.fp_set:N Setting a variable is very easy: just pass the data along.
.fp_set:c 8750 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:N } #1
.fp_gset:N 8751 { \__keys_variable_set:NnnN #1 { fp } { } n }
.fp_gset:c 8752 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:c } #1
            { \__keys_variable_set:cnnN {#1} { fp } { } n }
            8753 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:N } #1
            { \__keys_variable_set:NnnN #1 { fp } { g } n }
            8754 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
            { \__keys_variable_set:cnnN {#1} { fp } { g } n }
            8755 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
            { \__keys_variable_set:cnnN {#1} { fp } { g } n }
            8756 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
            { \__keys_variable_set:cnnN {#1} { fp } { g } n }
            8757 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
            { \__keys_variable_set:cnnN {#1} { fp } { g } n }
(End definition for .fp_set:N and .fp_set:c. These functions are documented on page 159.)

.groups:n A single property to create groups of keys.
8758 \cs_new_protected:cpn { \c__keys_props_root_tl .groups:n } #1
8759 { \__keys_groups_set:n {#1} }
(End definition for .groups:n. This function is documented on page 159.)

.initial:n The standard hand-off approach.
.initial:N 8760 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:n } #1
.initial:V 8761 { \__keys_initialise:n {#1} }
.initial:o 8762 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:V } #1
.initial:x 8763 { \exp_args:NV \__keys_initialise:n #1 }
            8764 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:o } #1
            { \exp_args:No \__keys_initialise:n {#1} }
            8765 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
            { \exp_args:Nx \__keys_initialise:n {#1} }
            8766 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
            { \exp_args:Nx \__keys_initialise:n {#1} }
            8767 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
            { \exp_args:Nx \__keys_initialise:n {#1} }
(End definition for .initial:n and others. These functions are documented on page 159.)

.int_set:N Setting a variable is very easy: just pass the data along.
.int_set:c 8768 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:N } #1
.int_gset:N 8769 { \__keys_variable_set:NnnN #1 { int } { } n }
.int_gset:c 8770 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:c } #1
            { \__keys_variable_set:cnnN {#1} { int } { } n }
            8771 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:N } #1
            { \__keys_variable_set:NnnN #1 { int } { g } n }
            8772 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:N } #1
            { \__keys_variable_set:NnnN #1 { int } { g } n }
            8773 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
            { \__keys_variable_set:cnnN {#1} { int } { g } n }
            8774 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
            { \__keys_variable_set:cnnN {#1} { int } { g } n }
            8775 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
            { \__keys_variable_set:cnnN {#1} { int } { g } n }
(End definition for .int_set:N and .int_set:c. These functions are documented on page 159.)

```

.meta:n Making a meta is handled internally.

```
8776 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:n } #1
8777 { \__keys_meta_make:n {#1} }
```

(End definition for .meta:n. This function is documented on page 160.)

.meta:nn Meta with path: potentially lots of variants, but for the moment no so many defined.

```
8778 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:nn } #1
8779 { \__keys_meta_make:nn #1 }
```

(End definition for .meta:nn. This function is documented on page 160.)

.multichoice: The same idea as .choice: and .choices:nn, but where more than one choice is allowed.

```
8780 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .multichoice: }
8781 { \__keys_multichoice_make: }
8782 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:nn } #1
8783 { \__keys_multichoices_make:nn #1 }
8784 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:Vn } #1
8785 { \exp_args:NV \__keys_multichoices_make:nn #1 }
8786 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:on } #1
8787 { \exp_args:No \__keys_multichoices_make:nn #1 }
8788 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:xn } #1
8789 { \exp_args:Nx \__keys_multichoices_make:nn #1 }
```

(End definition for .multichoice:. This function is documented on page 160.)

.skip_set:N Setting a variable is very easy: just pass the data along.

```
8790 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:N } #1
8791 { \__keys_variable_set:NnnN #1 { skip } { } n }
8792 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:c } #1
8793 { \__keys_variable_set:cnnN {#1} { skip } { } n }
8794 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:N } #1
8795 { \__keys_variable_set:NnnN #1 { skip } { g } n }
8796 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:c } #1
8797 { \__keys_variable_set:cnnN {#1} { skip } { g } n }
```

(End definition for .skip_set:N and .skip_set:c. These functions are documented on page 160.)

.tl_set:N Setting a variable is very easy: just pass the data along.

```
8798 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:N } #1
8799 { \__keys_variable_set:NnnN #1 { tl } { } n }
8800 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:c } #1
8801 { \__keys_variable_set:cnnN {#1} { tl } { } n }
8802 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:N } #1
8803 { \__keys_variable_set:NnnN #1 { tl } { } x }
8804 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:c } #1
8805 { \__keys_variable_set:cnnN {#1} { tl } { } x }
8806 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:N } #1
8807 { \__keys_variable_set:NnnN #1 { tl } { g } n }
8808 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:c } #1
8809 { \__keys_variable_set:cnnN {#1} { tl } { g } n }
8810 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:N } #1
```

```

8811 { \__keys_variable_set:NnnN #1 { t1 } { g } x }
8812 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:c } #1
8813 { \__keys_variable_set:cnnN {#1} { t1 } { g } x }

```

(End definition for .tl_set:N and .tl_set:c. These functions are documented on page 160.)

.value_forbidden: These are very similar, so both call the same function.

```

.value_required: 8814 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .value_forbidden: }
8815 { \__keys_value_requirement:n { forbidden } }
8816 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .value_required: }
8817 { \__keys_value_requirement:n { required } }

```

(End definition for .value_forbidden:. This function is documented on page 160.)

19.6 Setting keys

\keys_set:nn A simple wrapper again.

```

\keys_set:nV 8818 \cs_new_protected_nopar:Npn \keys_set:nn
\keys_set:nv 8819 { \__keys_set:onn { \l__keys_module_tl } }
\keys_set:no 8820 \cs_new_protected:Npn \__keys_set:nnn #1#2#3
\__keys_set:nnn 8821 {
\__keys_set:onn 8822   \tl_set:Nx \l__keys_module_tl { \tl_to_str:n {#2} }
8823   \keyval_parse:Nnn \__keys_set_elt:n \__keys_set_elt:nn {#3}
8824   \tl_set:Nn \l__keys_module_tl {#1}
8825 }
8826 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
8827 \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. To allow for nested setting, any existing value of \l__keys_unused_clist is saved on the stack and reset afterwards. Note that for speed/simplicity reasons we use a tl operation to set the clist here!

```

\keys_set_known:nVN 8828 \cs_new_protected_nopar:Npn \keys_set_known:nnN
\keys_set_known:nvN 8829 { \__keys_set_known:onnN \l__keys_unused_clist }
\keys_set_known:noN 8830 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
\__keys_set_known:nnnN 8831 \cs_new_protected:Npn \__keys_set_known:nnnN #1#2#3#4
\__keys_set_known:onnN 8832 {
\keys_set_known:nn 8833   \clist_clear:N \l__keys_unused_clist
\keys_set_known:nV 8834   \keys_set_known:nn {#2} {#3}
\keys_set_known:nv 8835   \tl_set:Nx #4 { \exp_not:o { \l__keys_unused_clist } }
\keys_set_known:no 8836   \tl_set:Nn \l__keys_unused_clist {#1}
8837 }
8838 \cs_generate_variant:Nn \__keys_set_known:nnnN { o }
8839 \cs_new_protected:Npn \keys_set_known:nn #1#2
8840 {
8841   \bool_set_true:N \l__keys_only_known_bool
8842   \keys_set:nn {#1} {#2}
8843   \bool_set_false:N \l__keys_only_known_bool
8844 }
8845 \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
\keys_set_filter:nnVN
\keys_set_filter:nnvN\keys_set_filter:nnnN
  \__keys_set_filter:nnnnN
  \__keys_set_filter:nnnnN
    \keys_set_filter:nnn
    \keys_set_filter:nnV
\keys_set_filter:nnv\keys_set_filter:nnn
  \keys_set_groups:nnn
  \keys_set_groups:nnV
\keys_set_groups:nnv\keys_set_groups:nnn
8846 \cs_new_protected_nopar:Npn \keys_set_filter:nnnN
8847 { \__keys_set_filter:nnnnN \l__keys_unused_clist }
8848 \cs_generate_variant:Nn \keys_set_filter:nnnN { nnV , nnv , nno }
8849 \cs_new_protected:Npn \__keys_set_filter:nnnnN #1#2#3#4#5
8850 {
8851   \clist_clear:N \l__keys_unused_clist
8852   \keys_set_filter:nnn {#2} {#3} {#4}
8853   \tl_set:Nx #5 { \exp_not:o { \l__keys_unused_clist } }
8854   \tl_set:Nn \l__keys_unused_clist {#1}
8855 }
8856 \cs_generate_variant:Nn \__keys_set_filter:nnnnN { o }
8857 \cs_new_protected:Npn \keys_set_filter:nnn #1#2#3
8858 {
8859   \bool_set_true:N \l__keys_selective_bool
8860   \bool_set_true:N \l__keys_filtered_bool
8861   \seq_set_from_clist:Nn \l__keys_selective_seq {#2}
8862   \keys_set:nn {#1} {#3}
8863   \bool_set_false:N \l__keys_selective_bool
8864 }
8865 \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
8866 \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
8867 {
8868   \bool_set_true:N \l__keys_selective_bool
8869   \bool_set_false:N \l__keys_filtered_bool
8870   \seq_set_from_clist:Nn \l__keys_selective_seq {#2}
8871   \keys_set:nn {#1} {#3}
8872   \bool_set_false:N \l__keys_selective_bool
8873 }
8874 \cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }

```

(End definition for \keys_set_filter:nnnN, \keys_set_filter:nnVN, and \keys_set_filter:nnvN\keys_set_filter:nnnN. These functions are documented on page ??.)

```

  \__keys_set_elt:n
  \__keys_set_elt:nn
  \__keys_set_elt_aux:nn
  \__keys_set_elt_aux:
  \__keys_set_elt_selective:
2875 \cs_new_protected:Npn \__keys_set_elt:n #1
2876 {
2877   \bool_set_true:N \l__keys_no_value_bool
2878   \__keys_set_elt_aux:nn {#1} { }
2879 }
2880 \cs_new_protected:Npn \__keys_set_elt:nn #1#2
2881 {
2882   \bool_set_false:N \l__keys_no_value_bool
2883   \__keys_set_elt_aux:nn {#1} {#2}
2884 }

```

A shared system once again. First, set the current path and add a default if needed. There are then checks to see if the a value is required or forbidden. If everything passes, move on to execute the code.

```

8885 \cs_new_protected:Npn \__keys_set_elt_aux:nn #1#2
8886 {
8887   \tl_set:Nx \l_keys_key_tl { \tl_to_str:n {#1} }
8888   \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / \l_keys_key_tl }
8889   \__keys_value_or_default:n {#2}
8890   \bool_if:NTF \l__keys_selective_bool
8891     { \__keys_set_elt_selective: }
8892     { \__keys_set_elt_aux: }
8893 }
8894 \cs_new_protected_nopar:Npn \__keys_set_elt_aux:
8895 {
8896   \bool_if:nTF
8897     {
8898       \__keys_if_value_p:n { required } &&
8899       \l__keys_no_value_bool
8900     }
8901     {
8902       \__msg_kernel_error:nxx { kernel } { value-required }
8903       { \l_keys_path_tl }
8904     }
8905     {
8906       \bool_if:nTF
8907         {
8908           \__keys_if_value_p:n { forbidden } &&
8909           ! \l__keys_no_value_bool
8910         }
8911         {
8912           \__msg_kernel_error:nxxx { kernel } { value-forbidden }
8913           { \l_keys_path_tl } { \l_keys_value_tl }
8914         }
8915         { \__keys_execute: }
8916     }
8917 }

```

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.

```

8918 \cs_new_protected_nopar:Npn \__keys_set_elt_selective:
8919 {
8920   \prop_if_exist:cTF { \c__keys_info_root_tl \l_keys_path_tl }
8921   {
8922     \prop_get:cnNTF { \c__keys_info_root_tl \l_keys_path_tl }
8923     { groups } \l__keys_groups_clist
8924     { \__keys_check_groups: }
8925     {
8926       \bool_if:NTF \l__keys_filtered_bool
8927         { \__keys_set_elt_aux: }
8928         { \__keys_store_unused: }
8929     }
8930   }

```

```

8931     {
8932         \bool_if:NTF \l__keys_filtered_bool
8933         { \__keys_set_elt_aux: }
8934         { \__keys_store_unused: }
8935     }
8936 }

```

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.

```

8937 \cs_new_protected_nopar:Npn \__keys_check_groups:
8938 {
8939     \bool_set_false:N \l__keys_tmp_bool
8940     \seq_map_inline:Nn \l__keys_selective_seq
8941     {
8942         \clist_map_inline:Nn \l__keys_groups_clist
8943         {
8944             \str_if_eq:nnT {##1} {####1}
8945             {
8946                 \bool_set_true:N \l__keys_tmp_bool
8947                 \clist_map_break:n { \seq_map_break: }
8948             }
8949         }
8950     }
8951     \bool_if:NTF \l__keys_tmp_bool
8952     {
8953         \bool_if:NTF \l__keys_filtered_bool
8954         { \__keys_store_unused: }
8955         { \__keys_set_elt_aux: }
8956     }
8957     {
8958         \bool_if:NTF \l__keys_filtered_bool
8959         { \__keys_set_elt_aux: }
8960         { \__keys_store_unused: }
8961     }
8962 }

```

(End definition for __keys_set_elt:n and __keys_set_elt:nn.)

__keys_value_or_default:n If a value is given, return it as #1, otherwise send a default if available.

```

8963 \cs_new_protected:Npn \__keys_value_or_default:n #1
8964 {
8965     \bool_if:NTF \l__keys_no_value_bool
8966     {
8967         \prop_get:cnNF { \c__keys_info_root_tl \l_keys_path_tl }
8968         { default } \l_keys_value_tl
8969         { \tl_clear:N \l_keys_value_tl }
8970     }
8971     { \tl_set:Nn \l_keys_value_tl {#1} }
8972 }

```

(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.

```

8973 \prg_new_conditional:Npnn \__keys_if_value:n #1 { p }
8974 {
8975   \prop_if_exist:cTF { \c__keys_info_root_tl \l_keys_path_tl }
8976   {
8977     \prop_if_in:cnTF { \c__keys_info_root_tl \l_keys_path_tl } {#1}
8978     { \prg_return_true: }
8979     { \prg_return_false: }
8980   }
8981   { \prg_return_false: }
8982 }

```

(End definition for `__keys_if_value_p:n`.)

`__keys_execute:` Actually executing a key is done in two parts. First, look for the key itself, then look for the `unknown` key with the same path. If both of these fail, complain. What exactly happens if a key is unknown depends on whether unknown keys are being skipped or if an error should be raised.

```

8983 \cs_new_protected_nopar:Npn \__keys_execute:
8984 { \__keys_execute:nn { \l_keys_path_tl } { \__keys_execute_unknown: } }
8985 \cs_new_protected_nopar:Npn \__keys_execute_unknown:
8986 {
8987   \bool_if:NTF \l__keys_only_known_bool
8988   { \__keys_store_unused: }
8989   {
8990     \__keys_execute:nn { \l__keys_module_tl / unknown }
8991     {
8992       \__msg_kernel_error:nxxx { kernel } { key-unknown }
8993       { \l_keys_path_tl } { \l__keys_module_tl }
8994     }
8995   }
8996 }
8997 \cs_new:Npn \__keys_execute:nn #1#2
8998 {
8999   \cs_if_exist:cTF { \c__keys_code_root_tl #1 }
9000   {
9001     \exp_args:Nc \exp_args:No { \c__keys_code_root_tl #1 }
9002     \l_keys_value_tl
9003   }
9004   {#2}
9005 }
9006 \cs_new_protected_nopar:Npn \__keys_store_unused:
9007 {
9008   \clist_put_right:Nx \l__keys_unused_clist
9009   {
9010     \exp_not:o \l_keys_key_tl
9011     \bool_if:NF \l__keys_no_value_bool
9012     { = { \exp_not:o \l_keys_value_tl } }
9013   }

```

```

9014 }
(End definition for \__keys_execute:.)

```

`__keys_choice_find:n` Executing a choice has two parts. First, try the choice given, then if that fails call the
`__keys_multichoice_find:n` unknown key. That will exist, as it is created when a choice is first made. So there is no
need for any escape code. For multiple choices, the same code ends up used in a mapping.

```

9015 \cs_new:Npn \__keys_choice_find:n #1
9016 {
9017   \__keys_execute:nn { \l_keys_path_tl / \tl_to_str:n {#1} }
9018   { \__keys_execute:nn { \l_keys_path_tl / unknown } { } }
9019 }
9020 \cs_new:Npn \__keys_multichoice_find:n #1
9021 { \clist_map_function:nN {#1} \__keys_choice_find:n }
(End definition for \__keys_choice_find:n.)

```

19.7 Utilities

`\keys_if_exist:p:nn` A utility for others to see if a key exists.

```

\keys_if_exist:nnTF
9022 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
9023 {
9024   \cs_if_exist:cTF { \c__keys_code_root_tl #1 / #2 }
9025   { \prg_return_true: }
9026   { \prg_return_false: }
9027 }
(End definition for \keys_if_exist:nn. These functions are documented on page 166.)

```

`\keys_if_choice_exist:p:nnn` Just an alternative view on `\keys_if_exist:nn(TF)`.

```

\keys_if_choice_exist:nnnTF
9028 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3 { p , T , F , TF }
9029 {
9030   \cs_if_exist:cTF { \c__keys_code_root_tl #1 / #2 / #3 }
9031   { \prg_return_true: }
9032   { \prg_return_false: }
9033 }
(End definition for \keys_if_choice_exist:nnn. These functions are documented on page 166.)

```

`\keys_show:nn` Showing a key is just a question of using the correct name.

```

9034 \cs_new_protected:Npn \keys_show:nn #1#2
9035 { \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 166.)

```

19.8 Messages

For when there is a need to complain.

```

9036 \_msg_kernel_new:nnnn { kernel } { boolean-values-only }
9037 { Key~'#1'~accepts~boolean~values~only. }
9038 { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
9039 \_msg_kernel_new:nnnn { kernel } { choice-unknown }
9040 { Choice~'#2'~unknown~for~key~'#1'. }
9041 {
9042   The~key~'#1'~takes~a~limited~number~of~values.\\
9043   The~input~given,~'#2',~is~not~on~the~list~accepted.
9044 }
9045 \_msg_kernel_new:nnnn { kernel } { key-choice-unknown }
9046 { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
9047 {
9048   The~key~'#1'~only~accepts~predefined~values,~and~'#2'~is~not~one~of~these.
9049 }
9050 \_msg_kernel_new:nnnn { kernel } { key-no-property }
9051 { No~property~given~in~definition~of~key~'#1'. }
9052 {
9053   \c__msg_coding_error_text_tl
9054   Inside~\keys_define:nn  each~key~name~
9055   needs~a~property:  \\ \\
9056   \iow_indent:n { #1 .<property> } \\ \\
9057   LaTeX~did~not~find~a~'. ' ~to~indicate~the~start~of~a~property.
9058 }
9059 \_msg_kernel_new:nnnn { kernel } { key-unknown }
9060 { The~key~'#1'~is~unknown~and~is~being~ignored. }
9061 {
9062   The~module~'#2'~does~not~have~a~key~called~'#1'.\\
9063   Check~that~you~have~spelled~the~key~name~correctly.
9064 }
9065 \_msg_kernel_new:nnnn { kernel } { nested-choice-key }
9066 { Attempt~to~define~'#1'~as~a~nested~choice~key. }
9067 {
9068   The~key~'#1'~cannot~be~defined~as~a~choice~as~the~parent~key~'#2'~is~
9069   itself~a~choice.
9070 }
9071 \_msg_kernel_new:nnnn { kernel } { property-requires-value }
9072 { The~property~'#1'~requires~a~value. }
9073 {
9074   \c__msg_coding_error_text_tl
9075   LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'.\\
9076   No~value~was~given~for~the~property,~and~one~is~required.
9077 }
9078 \_msg_kernel_new:nnnn { kernel } { property-unknown }
9079 { The~key~property~'#1'~is~unknown. }
9080 {
9081   \c__msg_coding_error_text_tl
9082   LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~

```

```

9083     this~property~is~not~defined.
9084   }
9085   \__msg_kernel_new:nnnn { kernel } { value-forbidden }
9086   { The~key~'#1'~does~not~taken~a~value. }
9087   {
9088     The~key~'#1'~should~be~given~without~a~value.\\
9089     LaTeX~will~ignore~the~given~value~'#2'.
9090   }
9091   \__msg_kernel_new:nnnn { kernel } { value-required }
9092   { The~key~'#1'~requires~a~value. }
9093   {
9094     The~key~'#1'~must~have~a~value.\\
9095     No~value~was~present:~the~key~will~be~ignored.
9096   }

```

19.9 Deprecated functions

Deprecated on 2013-07-09.

```

\__keys_choice_code_store:n
\__keys_choice_code_store:x
    .choice_code:n
    .choice_code:x
\__keys_choices_generate:n
    \__keys_choices_generate_aux:n
    .generate_choices:n

9097 \cs_new_protected:Npn \__keys_choice_code_store:n #1
9098 {
9099   \cs_if_exist:cF
9100   { \c__keys_info_root_tl \l_keys_path_tl .choice-code }
9101   {
9102     \tl_new:c
9103     { \c__keys_info_root_tl \l_keys_path_tl .choice-code }
9104   }
9105   \tl_set:cn { \c__keys_info_root_tl \l_keys_path_tl .choice-code }
9106   {#1}
9107 }
9108 \cs_generate_variant:Nn \__keys_choice_code_store:n { x }
9109 \cs_new_protected:cpn { \c__keys_props_root_tl .choice_code:n } #1
9110 { \__keys_choice_code_store:n {#1} }
9111 \cs_new_protected:cpn { \c__keys_props_root_tl .choice_code:x } #1
9112 { \__keys_choice_code_store:x {#1} }
9113 \cs_new_protected:Npn \__keys_choices_generate:n #1
9114 {
9115   \cs_if_exist:cTF
9116   { \c__keys_info_root_tl \l_keys_path_tl .choice-code }
9117   {
9118     \__keys_choice_make:
9119     \int_zero:N \l_keys_choice_int
9120     \clist_map_function:nN {#1} \__keys_choices_generate_aux:n
9121   }
9122   {
9123     \__msg_kernel_error:nnx { kernel }
9124     { generate-choices-before-code } { \l_keys_path_tl }
9125   }
9126 }
9127 \cs_new_protected:Npn \__keys_choices_generate_aux:n #1

```

```

9128 {
9129   \int_incr:N \l_keys_choice_int
9130   \__keys_cmd_set:nx { \l_keys_path_tl / #1 }
9131   {
9132     \tl_set:Nn \exp_not:N \l_keys_choice_tl {#1}
9133     \int_set:Nn \exp_not:N \l_keys_choice_int
9134       { \int_use:N \l_keys_choice_int }
9135     \exp_not:v
9136       { \c__keys_info_root_tl \l_keys_path_tl .choice~code }
9137   }
9138 }
9139 \__msg_kernel_new:nnnn { kernel } { generate-choices-before-code }
9140 { No~code~available~to~generate~choices~for~key~'~#1'. }
9141 {
9142   \c__msg_coding_error_text_tl
9143   Before~using~.generate_choices:n~the~code~should~be~defined~
9144   with~'.choice_code:n'~or~'.choice_code:x'.
9145 }
9146 \cs_new_protected:cpn { \c__keys_props_root_tl .generate_choices:n } #1
9147 { \__keys_choices_generate:n {#1} }
(End definition for \__keys_choice_code_store:n and \__keys_choice_code_store:x.)
9148 </initex | package>

```

20 l3file implementation

The following test files are used for this code: *m3file001*.

```

9149 <*initex | package>
9150 <@@=file>

```

20.1 File operations

\g_file_current_name_tl The name of the current file should be available at all times. For the format the file name needs to be picked up at the start of the file. In package mode the current file name is collected from L^AT_EX 2_ε.

```

9151 \tl_new:N \g_file_current_name_tl
9152 <*initex>
9153 \tex_everyjob:D \exp_after:wN
9154 {
9155   \tex_the:D \tex_everyjob:D
9156   \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
9157 }
9158 </initex>
9159 <*package>
9160 \tl_gset_eq:NN \g_file_current_name_tl \@currname
9161 </package>
(End definition for \g_file_current_name_tl. This variable is documented on page 168.)

```

`\g__file_stack_seq` The input list of files is stored as a sequence stack.

```

9162 \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`.

```

9163 \seq_new:N \g__file_record_seq
9164 \*initex
9165 \tex_everyjob:D \exp_after:wN
9166 {
9167   \tex_the:D \tex_everyjob:D
9168   \seq_gput_right:NV \g__file_record_seq \g_file_current_name_tl
9169 }
9170 \*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.

```

9171 \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.

```

9172 \tl_new:N \l__file_internal_name_tl
(End definition for \l__file_internal_name_tl. This variable is documented on page 174.)

```

`\l__file_search_path_seq` The current search path.

```

9173 \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.

```

9174 \*package
9175 \seq_new:N \l_file_saved_search_path_seq
9176 \*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.

```

9177 \*package
9178 \seq_new:N \l__file_internal_seq
9179 \*package
(End definition for \l__file_internal_seq. This variable is documented on page ??.)

```

`__file_name_sanitiz:nn` For converting a token list to a string where active characters are treated as strings from the start. The logic to the quoting normalisation is the same as used by `lualatexquotejobname`: check for balanced `"`, and assuming they balance strip all of them out before quoting the entire name if it contains spaces.

```

9180 \cs_new_protected:Npn \__file_name_sanitiz:nn #1#2
9181 {
9182   \group_begin:
9183     \seq_map_inline:Nn \l_char_active_seq
9184       { \cs_set_nopar:Npx ##1 { \token_to_str:N ##1 } }
9185     \tl_set:Nx \l__file_internal_name_tl {#1}
9186     \tl_set:Nx \l__file_internal_name_tl
9187       { \tl_to_str:N \l__file_internal_name_tl }
9188     \int_compare:nNnTF
9189       {
9190         \int_mod:nn
9191         {
9192           0 \tl_map_function:NN \l__file_internal_name_tl
9193             \__file_name_sanitiz_aux:n
9194         }
9195         \c_two
9196       }
9197     = \c_zero
9198     {
9199       \tl_remove_all:Nn \l__file_internal_name_tl { " }
9200       \tl_if_in:NnT \l__file_internal_name_tl { ~ }
9201       {
9202         \tl_set:Nx \l__file_internal_name_tl
9203           { " \exp_not:V \l__file_internal_name_tl " }
9204       }
9205     }
9206     {
9207       \__msg_kernel_error:nnx { kernel } { unbalanced-quote-in-filename }
9208       { \l__file_internal_name_tl }
9209     }
9210   \use:x
9211   {
9212     \group_end:
9213     \exp_not:n {#2} { \l__file_internal_name_tl }
9214   }
9215 }
9216 \cs_new:Npn \__file_name_sanitiz_aux:n #1
9217 {
9218   \str_if_eq:nnT {#1} { " }
9219   { + \c_one }
9220 }

```

(End definition for `__file_name_sanitiz:nn`.)

`\file_add_path: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.

`__file_add_path:nN`

`__file_add_path_search:nN`

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.

```

9221 \cs_new_protected:Npn \file_add_path:nN #1
9222 { \__file_name_sanitiz:n {#1} { \__file_add_path:nN } }
9223 \cs_new_protected:Npn \__file_add_path:nN #1#2
9224 {
9225   \__ior_open:Nn \g__file_internal_ior {#1}
9226   \ior_if_eof:NTF \g__file_internal_ior
9227     { \__file_add_path_search:nN {#1} #2 }
9228     { \tl_set:Nn #2 {#1} }
9229   \ior_close:N \g__file_internal_ior
9230 }
9231 \cs_new_protected:Npn \__file_add_path_search:nN #1#2
9232 {
9233   \tl_set:Nn #2 { \q_no_value }
9234   <*package>
9235   \cs_if_exist:NT \input@path
9236   {
9237     \seq_set_eq:NN \l__file_saved_search_path_seq \l__file_search_path_seq
9238     \seq_set_split:NnV \l__file_internal_seq { , } \input@path
9239     \seq_concat:NNN \l__file_search_path_seq
9240       \l__file_search_path_seq \l__file_internal_seq
9241   }
9242   </package>
9243   \seq_map_inline:Nn \l__file_search_path_seq
9244   {
9245     \__ior_open:Nn \g__file_internal_ior { ##1 #1 }
9246     \ior_if_eof:NF \g__file_internal_ior
9247     {
9248       \tl_set:Nx #2 { ##1 #1 }
9249       \seq_map_break:
9250     }
9251   }
9252   <*package>
9253   \cs_if_exist:NT \input@path
9254   { \seq_set_eq:NN \l__file_search_path_seq \l__file_saved_search_path_seq }
9255   </package>
9256 }

```

(End definition for \file_add_path:nN. This function is documented on page 168.)

\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.

```

9257 \prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
9258 {
9259   \file_add_path:nN {#1} \l__file_internal_name_tl
9260   \quark_if_no_value:NTF \l__file_internal_name_tl
9261   { \prg_return_false: }

```



```

9262     { \prg_return_true: }
9263   }

```

(End definition for \file_if_exist:nTF. This function is documented on page 168.)

```

\file_input:n
\__file_if_exist:nT
\__file_input:n\__file_input:V
\__file_input_aux:n
\__file_input_aux:o

```

Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the \g__file_stack_seq, and add it to the file list, either \g__file_record_seq, or \@filelist in package mode.

```

9264 \cs_new_protected:Npn \file_input:n #1
9265 {
9266   \__file_if_exist:nT {#1}
9267   { \__file_input:V \l__file_internal_name_tl }
9268 }

```

This code is spun out as a separate function to encapsulate the error message into a easy-to-reuse form.

```

9269 \cs_new_protected:Npn \__file_if_exist:nT #1#2
9270 {
9271   \file_if_exist:nTF {#1}
9272   {#2}
9273   {
9274     \__file_name_sanitiz:n {#1}
9275     { \__msg_kernel_error:nx { kernel } { file-not-found } }
9276   }
9277 }
9278 \cs_new_protected:Npn \__file_input:n #1
9279 {
9280   \tl_if_in:nnTF {#1} { . }
9281   { \__file_input_aux:n {#1} }
9282   { \__file_input_aux:o { \tl_to_str:n { #1 . tex } } }
9283 }
9284 \cs_generate_variant:Nn \__file_input:n { V }
9285 \cs_new_protected:Npn \__file_input_aux:n #1
9286 {
9287   <*initex>
9288   \seq_gput_right:Nn \g__file_record_seq {#1}
9289   </initex>
9290   <*package>
9291   \clist_if_exist:NTF \@filelist
9292   { \@addtofilelist {#1} }
9293   { \seq_gput_right:Nn \g__file_record_seq {#1} }
9294   </package>
9295   \seq_gpush:Nn \g__file_stack_seq \g_file_current_name_tl
9296   \tl_gset:Nn \g_file_current_name_tl {#1}
9297   \tex_input:D #1 \c_space_tl
9298   \seq_gpop:NN \g__file_stack_seq \l__file_internal_tl
9299   \tl_gset_eq:NN \g_file_current_name_tl \l__file_internal_tl
9300 }
9301 \cs_generate_variant:Nn \__file_input_aux:n { o }

```

(End definition for \file_input:n. This function is documented on page ??.)

`\file_path_include:n` Wrapper functions to manage the search path.

`\file_path_remove:n`

`__file_path_include:n`

```

9302 \cs_new_protected:Npn \file_path_include:n #1
9303 { \__file_name_sanitize:nn {#1} { \__file_path_include:n } }
9304 \cs_new_protected:Npn \__file_path_include:n #1
9305 {
9306   \seq_if_in:NnF \l__file_search_path_seq {#1}
9307   { \seq_put_right:Nn \l__file_search_path_seq {#1} }
9308 }
9309 \cs_new_protected:Npn \file_path_remove:n #1
9310 {
9311   \__file_name_sanitize:nn {#1}
9312   { \seq_remove_all:Nn \l__file_search_path_seq }
9313 }

```

(End definition for `\file_path_include:n`. This function is documented on page 169.)

`\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.

```

9314 \cs_new_protected_nopar:Npn \file_list:
9315 {
9316   \seq_set_eq:NN \l__file_internal_seq \g__file_record_seq
9317   <*package>
9318   \clist_if_exist:NT \@filelist
9319   {
9320     \clist_map_inline:Nn \@filelist
9321     {
9322       \seq_put_right:No \l__file_internal_seq
9323       { \tl_to_str:n {##1} }
9324     }
9325   }
9326   </package>
9327   \seq_remove_duplicates:N \l__file_internal_seq
9328   \iow_log:n { *-File-List-* }
9329   \seq_map_inline:Nn \l__file_internal_seq { \iow_log:n {##1} }
9330   \iow_log:n { ***** }
9331 }

```

(End definition for `\file_list:`. This function is documented on page 169.)

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`.

```

9332 <*package>
9333 \AtBeginDocument
9334 {
9335   \clist_map_inline:Nn \@filelist
9336   { \seq_gput_right:No \g__file_record_seq { \tl_to_str:n {#1} } }
9337 }
9338 </package>

```

20.2 Input operations

9339 `<@@=ior>`

20.2.1 Variables and constants

`\c_term_ior` Reading from the terminal (with a prompt) is done using a positive but non-existent stream number. Unlike writing, there is no concept of reading from the log.

9340 `\cs_new_eq:NN \c_term_ior \c_sixteen`

(End definition for `\c_term_ior`. This variable is documented on page 174.)

`\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!

9341 `\seq_new:N \g__ior_streams_seq`

9342 `<*initex>`

9343 `\seq_gset_split:Nnn \g__ior_streams_seq { , }`

9344 `{ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 }`

9345 `</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.

9346 `\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.

9347 `\prop_new:N \g__ior_streams_prop`

9348 `<*package>`

9349 `\prop_gput:Nnn \g__ior_streams_prop { 0 } { LaTeX2e-reserved }`

9350 `</package>`

(End definition for `\g__ior_streams_prop`. This variable is documented on page ??.)

20.2.2 Stream management

`\ior_new:N` Reserving a new stream is done by defining the name as equal to using the terminal.

`\ior_new:c` 9351 `\cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c_term_ior }`

9352 `\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`

9353 `\cs_new_protected:Npn \ior_open:Nn #1#2`

9354 `{ __file_name_sanitize:nn {#2} { __ior_open_aux:Nn #1 } }`

9355 `\cs_generate_variant:Nn \ior_open:Nn { c }`

9356 `\cs_new_protected:Npn __ior_open_aux:Nn #1#2`

9357 `{`

9358 `\file_add_path:nN {#2} \l__file_internal_name_tl`

```

9359 \quark_if_no_value:NTF \l__file_internal_name_tl
9360 { \_msg_kernel_error:nx { kernel } { file-not-found } {#2} }
9361 { \_ior_open:No #1 \l__file_internal_name_tl }
9362 }

```

(End definition for \ior_open:Nn and \ior_open:cn. These functions are documented on page ??.)

\ior_open:NnTF

\ior_open:cnTF

Much the same idea for opening a read with a conditional, except the auxiliary function does not issue an error if the file is not found.

_ior_open_aux:NnTF

```

9363 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
9364 { \_file_name_sanitize:nn {#2} { \_ior_open_aux:NnTF #1 } }
9365 \cs_generate_variant:Nn \ior_open:NnT { c }
9366 \cs_generate_variant:Nn \ior_open:NnF { c }
9367 \cs_generate_variant:Nn \ior_open:NnTF { c }
9368 \cs_new_protected:Npn \_ior_open_aux:NnTF #1#2
9369 {
9370   \file_add_path:nN {#2} \l__file_internal_name_tl
9371   \quark_if_no_value:NTF \l__file_internal_name_tl
9372   { \prg_return_false: }
9373   {
9374     \_ior_open:No #1 \l__file_internal_name_tl
9375     \prg_return_true:
9376   }
9377 }

```

(End definition for \ior_open:NnTF and \ior_open:cnTF. These functions are documented on page ??.)

_ior_open:Nn

_ior_open:No

_ior_open_stream:Nn

The stream allocation itself uses the fact that there is a list of all of those available, so allocation is simply a question of using the number at the top of the list. In package mode, life gets more complex as it's important to keep things in sync. That is done using a two-part approach: any streams that have already been taken up by ior but are now free are tracked, so we first try those. If that fails, ask L^AT_EX 2_ε for a new stream and use that number (after a bit of conversion).

```

9378 \cs_new_protected:Npn \_ior_open:Nn #1#2
9379 {
9380   \ior_close:N #1
9381   \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
9382   { \_ior_open_stream:Nn #1 {#2} }
9383   <*initex>
9384   { \_msg_kernel_fatal:nn { kernel } { input-streams-exhausted } }
9385   </initex>
9386   <*package>
9387   {
9388     \cs:w newread \cs_end: #1
9389     \tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
9390     \_ior_open_stream:Nn #1 {#2}
9391   }
9392   </package>
9393   }
9394   \cs_generate_variant:Nn \_ior_open:Nn { No }
9395   \cs_new_protected:Npn \_ior_open_stream:Nn #1#2

```

```

9396 {
9397   \tex_global:D \tex_chardef:D #1 = \l__ior_stream_tl \scan_stop:
9398   \prop_gput:NvN \g__ior_streams_prop #1 {#2}
9399   \tex_openin:D #1 #2 \scan_stop:
9400 }

```

(End definition for `__ior_open:Nn` and `__ior_open:No`.)

`\ior_close:N` Closing a stream means getting rid of it at the T_EX level and removing from the various 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`

```

9401 \cs_new_protected:Npn \ior_close:N #1
9402 {
9403   \int_compare:nT { \c_minus_one < #1 < \c_sixteen }
9404   {
9405     \tex_closein:D #1
9406     \prop_gremove:Nv \g__ior_streams_prop #1
9407     \seq_if_in:NvF \g__ior_streams_seq #1
9408     { \seq_gpush:Nv \g__ior_streams_seq #1 }
9409     \cs_gset_eq:NN #1 \c_term_ior
9410   }
9411 }
9412 \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`

```

9413 \cs_new_protected_nopar:Npn \ior_list_streams:
9414 { \__ior_list_streams:Nn \g__ior_streams_prop { input } }
9415 \cs_new_protected:Npn \__ior_list_streams:Nn #1#2
9416 {
9417   \__msg_term:nnn { LaTeX / kernel }
9418   { \prop_if_empty:NTF #1 { show-no-stream } { show-open-streams } }
9419   {#2}
9420   \__msg_show_variable:n
9421   { \prop_map_function:NN #1 \__msg_show_item_unbraced:nn }
9422 }

```

(End definition for `\ior_list_streams:`. This function is documented on page 170.)

20.2.3 Reading input

`\if_eof:w` The primitive conditional

```

9423 \cs_new_eq:NN \if_eof:w \tex_ifeof:D

```

(End definition for `\if_eof:w`.)

`\ior_if_eof_p:N` To test if some particular input stream is exhausted the following conditional is provided.

```
\ior_if_eof:NTF
9424 \prg_new_conditional:Nnn \ior_if_eof:N { p , T , F , TF }
9425 {
9426   \cs_if_exist:NTF #1
9427   {
9428     \if_int_compare:w #1 = \c_sixteen
9429     \prg_return_true:
9430   \else:
9431     \if_eof:w #1
9432     \prg_return_true:
9433   \else:
9434     \prg_return_false:
9435   \fi:
9436 \fi:
9437 }
9438 { \prg_return_true: }
9439 }
```

(End definition for `\ior_if_eof:N`. These functions are documented on page 171.)

`\ior_get:NN` And here we read from files.

```
9440 \cs_new_protected:Npn \ior_get:NN #1#2
9441 { \tex_read:D #1 to #2 }
```

(End definition for `\ior_get:NN`. This function is documented on page 170.)

`\ior_get_str:NN` Reading as strings is a more complicated wrapper, as we wish to remove the endline character.

```
9442 \cs_new_protected:Npn \ior_get_str:NN #1#2
9443 {
9444   \use:x
9445   {
9446     \int_set_eq:NN \tex_endlinechar:D \c_minus_one
9447     \exp_not:n { \etex_readline:D #1 to #2 }
9448     \int_set:Nn \tex_endlinechar:D { \int_use:N \tex_endlinechar:D }
9449   }
9450 }
```

(End definition for `\ior_get_str:NN`. This function is documented on page 171.)

`\g_file_internal_ior` Needed by the higher-level code, but cannot be created until here.

```
9451 \ior_new:N \g_file_internal_ior
```

(End definition for `\g_file_internal_ior`. This variable is documented on page 174.)

20.3 Output operations

```
9452 <@@=iow>
```

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

20.3.1 Variables and constants

`\c_log_iow` Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`)
`\c_term_iow` and to both the terminal and transcript file (`\c_term_iow`).

```
9453 \cs_new_eq:NN \c_log_iow \c_minus_one
9454 \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 174.)

`\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!

```
9455 \seq_new:N \g__iow_streams_seq
9456 \*initex
9457 \seq_gset_eq:NN \g__iow_streams_seq \g__ior_streams_seq
9458 \*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.

```
9459 \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.

```
9460 \prop_new:N \g__iow_streams_prop
9461 \*package
9462 \prop_put:Nnn \g__iow_streams_prop { 0 } { LaTeX2e-reserved }
9463 \prop_put:Nnn \g__iow_streams_prop { 1 } { LaTeX2e-reserved }
9464 \prop_put:Nnn \g__iow_streams_prop { 2 } { LaTeX2e-reserved }
9465 \*package
```

(End definition for `\g__iow_streams_prop`. This variable is documented on page ??.)

20.4 Stream management

`\iow_new:N` Reserving a new stream is done by defining the name as equal to writing to the terminal:
`\iow_new:c` odd but at least consistent.

```
9466 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
9467 \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`

```
9468 \cs_new_protected:Npn \iow_open:Nn #1#2
9469 { \__file_name_sanitize:nn {#2} { \__iow_open:Nn #1 } }
9470 \cs_generate_variant:Nn \iow_open:Nn { c }
9471 \cs_new_protected:Npn \__iow_open:Nn #1#2
9472 {
9473   \iow_close:N #1
9474   \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
```

```

9475 { \_iow\_open\_stream:Nn #1 {#2} }
9476 <*initex>
9477 { \_msg\_kernel\_fatal:nn { kernel } { output-streams-exhausted } }
9478 </initex>
9479 <*package>
9480 {
9481   \cs:w newwrite \cs_end: #1
9482   \tl_set:Nx \l\_iow\_stream\_tl { \int_eval:n {#1} }
9483   \_iow\_open\_stream:Nn #1 {#2}
9484 }
9485 </package>
9486 }
9487 \cs_generate_variant:Nn \_iow\_open:Nn { No }
9488 \cs_new_protected:Npn \_iow\_open\_stream:Nn #1#2
9489 {
9490   \tex_global:D \tex_chardef:D #1 = \l\_iow\_stream\_tl \scan_stop:
9491   \prop_gput:Nv \g\_iow\_streams\_prop #1 {#2}
9492   \tex_immediate:D \tex_openout:D #1 #2 \scan_stop:
9493 }

```

(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.

\iow_close:c

```

9494 \cs_new_protected:Npn \iow\_close:N #1
9495 {
9496   \int_compare:nT { \c_minus_one < #1 < \c_sixteen }
9497   {
9498     \tex_immediate:D \tex_closeout:D #1
9499     \prop_gremove:Nv \g\_iow\_streams\_prop #1
9500     \seq_if_in:NVF \g\_iow\_streams\_seq #1
9501     { \seq_gpush:Nv \g\_iow\_streams\_seq #1 }
9502     \cs_gset_eq:NN #1 \c_term\_ior
9503   }
9504 }
9505 \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:
_iow_list_streams:Nn

Done as for input, but with a copy of the auxiliary so the name is correct.

```

9506 \cs_new_protected_nopar:Npn \iow\_list\_streams:
9507 { \_iow\_list\_streams:Nn \g\_iow\_streams\_prop { output } }
9508 \cs_new_eq:NN \_iow\_list\_streams:Nn \_ior\_list\_streams:Nn

```

(End definition for \iow_list_streams:. This function is documented on page ??.)

20.4.1 Deferred writing

\iow_shipout_x:Nn
\iow_shipout_x:Nx
\iow_shipout_x:cn
\iow_shipout_x:cx

First the easy part, this is the primitive, which expects its argument to be braced.

```

9509 \cs_new_protected:Npn \iow\_shipout\_x:Nn #1#2
9510 { \tex_write:D #1 {#2} }

```



```
9511 \cs_generate_variant:Nn \iow_shipout_x:Nn { c, Nx, cx }
(End definition for \iow_shipout_x:Nn and others. These functions are documented on page ??.)
```

\iow_shipout:Nn With ϵ -TeX available deferred writing without expansion is easy.

```
\iow_shipout:Nx 9512 \cs_new_protected:Npn \iow_shipout:Nn #1#2
\iow_shipout:cn 9513 { \tex_write:D #1 { \exp_not:n {#2} } }
\iow_shipout:cx 9514 \cs_generate_variant:Nn \iow_shipout:Nn { c, Nx, cx }
(End definition for \iow_shipout:Nn and others. These functions are documented on page ??.)
```

20.4.2 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
\iow_now:cn no output stream at all, we get an internal error. We don't use the expansion done by
\iow_now:cx `\write` to get the Nx variant, because it differs in subtle ways from x-expansion, namely, macro parameter characters would not need to be doubled.

```
9515 \cs_new_protected:Npn \iow_now:Nn #1#2
9516 { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
9517 \cs_generate_variant:Nn \iow_now:Nn { c, Nx, cx }
(End definition for \iow_now:Nn and others. These functions are documented on page ??.)
```

\iow_log:n Writing to the log and the terminal directly are relatively easy.

```
\iow_log:x 9518 \cs_set_protected_nopar:Npn \iow_log:x { \iow_now:Nx \c_log_iow }
\iow_term:n 9519 \cs_new_protected_nopar:Npn \iow_log:n { \iow_now:Nn \c_log_iow }
\iow_term:x 9520 \cs_set_protected_nopar:Npn \iow_term:x { \iow_now:Nx \c_term_iow }
9521 \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 ??.)
```

20.4.3 Special characters for writing

\iow_newline: Global variable holding the character that forces a new line when something is written to an output stream.

```
9522 \cs_new_nopar:Npn \iow_newline: { ^~J }
(End definition for \iow_newline:. This function is documented on page 172.)
```

\iow_char:N Function to write any escaped char to an output stream.

```
9523 \cs_new_eq:NN \iow_char:N \cs_to_str:N
(End definition for \iow_char:N. This function is documented on page 172.)
```

20.4.4 Hard-wrapping lines to a character count

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

<code>\l_iow_line_count_int</code>	<p>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.</p> <pre>9524 \int_new:N \l_iow_line_count_int 9525 \int_set:Nn \l_iow_line_count_int { 78 }</pre> <p>(End definition for <code>\l_iow_line_count_int</code>. This variable is documented on page 173.)</p>
<code>\l__iow_target_count_int</code>	<p>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.</p> <pre>9526 \int_new:N \l__iow_target_count_int</pre> <p>(End definition for <code>\l__iow_target_count_int</code>.)</p>
<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>9527 \int_new:N \l__iow_current_line_int 9528 \int_new:N \l__iow_current_word_int 9529 \int_new:N \l__iow_current_indentation_int</pre> <p>(End definition for <code>\l__iow_current_line_int</code>, <code>\l__iow_current_word_int</code>, and <code>\l__iow_current_indentation_int</code>.)</p>
<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>9530 \tl_new:N \l__iow_current_line_tl 9531 \tl_new:N \l__iow_current_word_tl 9532 \tl_new:N \l__iow_current_indentation_tl</pre> <p>(End definition for <code>\l__iow_current_line_tl</code>, <code>\l__iow_current_word_tl</code>, and <code>\l__iow_current_indentation_tl</code>.)</p>
<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>9533 \tl_new:N \l__iow_wrap_tl</pre> <p>(End definition for <code>\l__iow_wrap_tl</code>.)</p>
<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>9534 \tl_new:N \l__iow_newline_tl</pre> <p>(End definition for <code>\l__iow_newline_tl</code>.)</p>
<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>9535 \bool_new:N \l__iow_line_start_bool</pre> <p>(End definition for <code>\l__iow_line_start_bool</code>.)</p>

`\c_catcode_other_space_tl` Lowercase a character with category code 12 to produce an “other” space. We can do everything within the group, because `\tl_const:Nn` defines its argument globally.

```

9536 \group_begin:
9537   \char_set_catcode_other:N \*
9538   \char_set_lccode:nn {'\*} {'\ }
9539   \tl_to_lowercase:n { \tl_const:Nn \c_catcode_other_space_tl { * } }
9540 \group_end:

```

(End definition for `\c_catcode_other_space_tl`. This function is documented on page 174.)

`\c__iow_wrap_marker_tl` Every special action of the wrapping code is preceded by the same recognizable string, `\c__iow_wrap_marker_tl`. Upon seeing that “word”, the wrapping code reads one space-delimited argument to know what operation to perform. The setting of `\escapechar` here is not very important, but makes `\c__iow_wrap_marker_tl` look nicer.

```

\c__iow_wrap_end_marker_tl
\c__iow_wrap_newline_marker_tl
\c__iow_wrap_indent_marker_tl
\c__iow_wrap_unindent_marker_tl
9541 \group_begin:
9542   \int_set_eq:NN \tex_escapechar:D \c_minus_one
9543   \tl_const:Nx \c__iow_wrap_marker_tl
9544     { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }
9545 \group_end:
9546 \tl_map_inline:nn
9547   { { end } { newline } { indent } { unindent } }
9548   {
9549     \tl_const:cx { c__iow_wrap_ #1 _marker_tl }
9550     {
9551       \c_catcode_other_space_tl
9552       \c__iow_wrap_marker_tl
9553       \c_catcode_other_space_tl
9554       #1
9555       \c_catcode_other_space_tl
9556     }
9557   }

```

(End definition for `\c__iow_wrap_marker_tl`.)

`\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.

```

9558 \cs_new_protected:Npn \iow_indent:n #1 { }
9559 \cs_new:Npx \__iow_indent:n #1
9560 {
9561   \c__iow_wrap_indent_marker_tl
9562   #1
9563   \c__iow_wrap_unindent_marker_tl
9564 }

```

(End definition for `\iow_indent:n`. This function is documented on page 173.)

`\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.

```

9565 \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
9566 {
9567   \group_begin:
9568     \int_set_eq:NN \tex_escapechar:D \c_minus_one
9569     \cs_set_nopar:Npx \{ { \token_to_str:N \{ }
9570     \cs_set_nopar:Npx \# { \token_to_str:N \# }
9571     \cs_set_nopar:Npx \} { \token_to_str:N \} }
9572     \cs_set_nopar:Npx \% { \token_to_str:N \% }
9573     \cs_set_nopar:Npx \~ { \token_to_str:N \~ }
9574     \int_set:Nn \tex_escapechar:D { 92 }
9575     \cs_set_eq:NN \ \c__iow_wrap_newline_marker_tl
9576     \cs_set_eq:NN \ \c_catcode_other_space_tl
9577     \cs_set_eq:NN \iow_indent:n \__iow_indent:n
9578     #3
9579   <*initex>
9580     \tl_set:Nx \l__iow_wrap_tl {#1}
9581   </initex>
9582   <*package>
9583     \__iow_wrap_set:Nx \l__iow_wrap_tl {#1}
9584   </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.

```

9585   \tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
9586   \tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
9587   \tl_replace_all:Nnn \l__iow_newline_tl { ~ } { \c_space_tl }
9588   \int_set:Nn \l__iow_target_count_int
9589     { \l__iow_line_count_int - \tl_count:N \l__iow_newline_tl + \c_one }
9590   \int_zero:N \l__iow_current_indentation_int
9591   \tl_clear:N \l__iow_current_indentation_tl
9592   \int_zero:N \l__iow_current_line_int
9593   \tl_clear:N \l__iow_current_line_tl
9594   \bool_set_true:N \l__iow_line_start_bool
9595   \use:x
9596   {
9597     \exp_not:n { \tl_clear:N \l__iow_wrap_tl }
9598     \__iow_wrap_loop:w
9599     \tl_to_str:N \l__iow_wrap_tl
9600     \tl_to_str:N \c__iow_wrap_end_marker_tl
9601     \c_space_tl \c_space_tl

```

```

9602         \exp_not:N \q_stop
9603     }
9604     \exp_args:NNo \group_end:
9605     #4 \l__iow_wrap_tl
9606 }

```

As using the generic loader will mean that `\protected@edef` is not available, it's not placed directly in the wrap function but is set up as an auxiliary. In the generic loader this can then be redefined.

```

9607 <*package>
9608 \cs_new_eq:NN \__iow_wrap_set:Nx \protected@edef
9609 </package>

```

(End definition for `\iow_wrap:nnnN`. This function is documented on page 173.)

`__iow_wrap_loop:w`

The loop grabs one word in the input, and checks whether it is the special marker, or a normal word.

```

9610 \cs_new_protected:Npn \__iow_wrap_loop:w #1 ~ %
9611 {
9612     \tl_set:Nn \l__iow_current_word_tl {#1}
9613     \tl_if_eq:NNTF \l__iow_current_word_tl \c__iow_wrap_marker_tl
9614     { \__iow_wrap_special:w }
9615     { \__iow_wrap_word: }
9616 }

```

(End definition for `__iow_wrap_loop:w`.)

`__iow_wrap_word:`
`__iow_wrap_word_fits:`
`__iow_wrap_word_newline:`

For a normal word, update the line count, then test if the current word would fit in the current line, and call the appropriate function. If the word fits in the current line, add it to the line, preceded by a space unless it is the first word of the line. Otherwise, the current line is added to the result, with the run-on text. The current word (and its character count) are then put in the new line.

```

9617 \cs_new_protected_nopar:Npn \__iow_wrap_word:
9618 {
9619     \int_set:Nn \l__iow_current_word_int
9620     { \__str_count_ignore_spaces:N \l__iow_current_word_tl }
9621     \int_add:Nn \l__iow_current_line_int { \l__iow_current_word_int }
9622     \int_compare:nNnTF \l__iow_current_line_int < \l__iow_target_count_int
9623     { \__iow_wrap_word_fits: }
9624     { \__iow_wrap_word_newline: }
9625     \__iow_wrap_loop:w
9626 }
9627 \cs_new_protected_nopar:Npn \__iow_wrap_word_fits:
9628 {
9629     \bool_if:NNTF \l__iow_line_start_bool
9630     {
9631         \bool_set_false:N \l__iow_line_start_bool
9632         \tl_put_right:Nx \l__iow_current_line_tl
9633         { \l__iow_current_indentation_tl \l__iow_current_word_tl }
9634         \int_add:Nn \l__iow_current_line_int
9635         { \l__iow_current_indentation_int }

```

```

9636     }
9637     {
9638         \tl_put_right:Nx \l__iow_current_line_tl
9639         { ~ \l__iow_current_word_tl }
9640         \int_incr:N \l__iow_current_line_int
9641     }
9642 }
9643 \cs_new_protected_nopar:Npn \__iow_wrap_word_newline:
9644 {
9645     \tl_put_right:Nx \l__iow_wrap_tl
9646     { \l__iow_current_line_tl \l__iow_newline_tl }
9647     \int_set:Nn \l__iow_current_line_int
9648     {
9649         \l__iow_current_word_int
9650         + \l__iow_current_indentation_int
9651     }
9652     \tl_set:Nx \l__iow_current_line_tl
9653     { \l__iow_current_indentation_tl \l__iow_current_word_tl }
9654 }

```

(End definition for __iow_wrap_word:.)

`__iow_wrap_special:w` When the “special” marker is encountered, read what operation to perform, as a space-delimited argument, perform it, and remember to loop. In fact, to avoid spurious spaces when two special actions follow each other, we look ahead for another copy of the marker.

`__iow_wrap_newline:w` 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.

`__iow_wrap_indent:w` 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.

`__iow_wrap_unindent:w`

`__iow_wrap_end:w`

```

9655 \cs_new_protected:Npn \__iow_wrap_special:w #1 ~ #2 ~ #3 ~ %
9656 {
9657     \use:c { __iow_wrap_#1: }
9658     \str_if_eq_x:nnTF { #2~#3 } { ~ \c__iow_wrap_marker_tl }
9659     { \__iow_wrap_special:w }
9660     { \__iow_wrap_loop:w #2 ~ #3 ~ }
9661 }
9662 \cs_new_protected_nopar:Npn \__iow_wrap_newline:
9663 {
9664     \tl_put_right:Nx \l__iow_wrap_tl
9665     { \l__iow_current_line_tl \l__iow_newline_tl }
9666     \int_zero:N \l__iow_current_line_int
9667     \tl_clear:N \l__iow_current_line_tl
9668     \bool_set_true:N \l__iow_line_start_bool
9669 }
9670 \cs_new_protected_nopar:Npx \__iow_wrap_indent:
9671 {
9672     \int_add:Nn \l__iow_current_indentation_int \c_four
9673     \tl_put_right:Nx \exp_not:N \l__iow_current_indentation_tl
9674     { \c_space_tl \c_space_tl \c_space_tl \c_space_tl }
9675 }

```

```

9676 \cs_new_protected_nopar:Npn \__iow_wrap_unindent:
9677 {
9678   \int_sub:Nn \l__iow_current_indentation_int \c_four
9679   \tl_set:Nx \l__iow_current_indentation_tl
9680     { \prg_replicate:nn \l__iow_current_indentation_int { ~ } }
9681 }
9682 \cs_new_protected_nopar:Npn \__iow_wrap_end:
9683 {
9684   \tl_put_right:Nx \l__iow_wrap_tl
9685     { \l__iow_current_line_tl }
9686   \use_none_delimit_by_q_stop:w
9687 }
(End definition for \__iow_wrap_special:w.)

```

```

\__str_count_ignore_spaces:N
\__str_count_ignore_spaces:n
\__str_count_loop:NNNNNNNNN

```

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.

```

9688 \cs_new_nopar:Npn \__str_count_ignore_spaces:N
9689 { \exp_args:No \__str_count_ignore_spaces:n }
9690 \cs_new:Npn \__str_count_ignore_spaces:n #1
9691 {
9692   \__int_value:w \__int_eval:w
9693   \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
9694     { X8 } { X7 } { X6 } { X5 } { X4 } { X3 } { X2 } { X1 } { X0 } \q_stop
9695   \__int_eval_end:
9696 }
9697 \cs_new:Npn \__str_count_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
9698 {
9699   \if_catcode:w X #9
9700     \exp_after:wN \use_none_delimit_by_q_stop:w
9701   \else:
9702     9 +
9703     \exp_after:wN \__str_count_loop:NNNNNNNNN
9704   \fi:
9705 }
(End definition for \__str_count_ignore_spaces:N.)

```

20.5 Messages

```

9706 \__msg_kernel_new:nnnn { kernel } { file-not-found }
9707 { File~'#1'~not-found. }
9708 {
9709   The~requested~file~could~not~be~found~in~the~current~directory,~
9710   in~the~TeX~search~path~or~in~the~LaTeX~search~path.
9711 }
9712 \__msg_kernel_new:nnnn { kernel } { input-streams-exhausted }
9713 { Input~streams~exhausted }
9714 {
9715   TeX~can~only~open~up~to~16~input~streams~at~one~time.\\
9716   All~16~are~currently~in~use,~and~something~wanted~to~open~

```

```

9717     another~one.
9718   }
9719   \_msg_kernel_new:nnnn { kernel } { output-streams-exhausted }
9720   { Output~streams~exhausted }
9721   {
9722     TeX~can~only~open~up~to~16~output~streams~at~one~time.\\
9723     All~16~are~currently~in~use,~and~something~wanted~to~open~
9724     another~one.
9725   }
9726   \_msg_kernel_new:nnnn { kernel } { unbalanced-quote-in-filename }
9727   { Unbalanced~quotes~in~file~name~'#1'. }
9728   {
9729     File~names~must~contain~balanced~numbers~of~quotes~(").
9730   }
9731 </initex | package>

```

21 l3fp implementation

Nothing to see here: everything is in the subfiles!

22 l3fp-aux implementation

```

9732 <*initex | package>
9733 <@@=fp>

```

23 Internal representation

Internally, a floating point number $\langle X \rangle$ is a token list containing

```
\s__fp \__fp_chk:w  $\langle case \rangle$   $\langle sign \rangle$   $\langle body \rangle$  ;
```

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,

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.

- 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 \langle case \rangle \langle sign \rangle \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 \langle sign \rangle {\langle exponent \rangle} {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} ;`

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 $1000 \leq \langle X_1 \rangle \leq 9999$.

24 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.

24.1 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.

```
9734 \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:n 9735 \cs_new:Npn \_fp\_use\_s:n #1 { #1; }
\_fp\_use\_s:nn 9736 \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`

```
9737 \cs_new:Npn \_fp\_use\_none\_until\_s:w #1; { }
9738 \cs_new:Npn \_fp\_use\_i\_until\_s:nw #1#2; {#1}
9739 \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.

```
9740 \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.

```
9741 \cs_new:Npn \_fp\_rrot:www #1; #2; #3; { #2; #3; #1; }
(End definition for \_fp\_rrot:www.)
```

`_fp_use_i:ww` Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.
`_fp_use_i:www`

```
9742 \cs_new:Npn \_fp\_use\_i:ww #1; #2; { #1; }
9743 \cs_new:Npn \_fp\_use\_i:www #1; #2; #3; { #1; }
(End definition for \_fp\_use\_i:ww and \_fp\_use\_i:www.)
```

24.2 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 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.

```

9744 \__scan_new:N \s__fp
9745 \cs_new_protected:Npn \__fp_chk:w #1 ;
9746 {
9747   \msg_kernel_error:nnx { kernel } { misused-fp }
9748   { \fp_to_tl:n { \s__fp \__fp_chk:w #1 ; } }
9749 }
(End definition for \s__fp and \__fp_chk:w.)

```

`\s__fp_mark` Aliases of `\tex_relax:D`, used to terminate expressions.

```

\s__fp_stop
9750 \__scan_new:N \s__fp_mark
9751 \__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
\s__fp_overflow
\s__fp_division
\s__fp_exact
9752 \__scan_new:N \s__fp_invalid
9753 \__scan_new:N \s__fp_underflow
9754 \__scan_new:N \s__fp_overflow
9755 \__scan_new:N \s__fp_division
9756 \__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` where the dots in `\s__fp...` are one of `invalid`, `underflow`, `overflow`, `division`,
`\c_minus_inf_fp` `exact`, describing how the floating point was created. We define the floating points here
`\c_nan_fp` as “exact”.

```

9757 \tl_const:Nn \c_zero_fp { \s__fp \__fp_chk:w 0 0 \s__fp_exact ; }
9758 \tl_const:Nn \c_minus_zero_fp { \s__fp \__fp_chk:w 0 2 \s__fp_exact ; }
9759 \tl_const:Nn \c_inf_fp { \s__fp \__fp_chk:w 2 0 \s__fp_exact ; }
9760 \tl_const:Nn \c_minus_inf_fp { \s__fp \__fp_chk:w 2 2 \s__fp_exact ; }
9761 \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 ??.)

```

`\c__fp_max_exponent_int` Normal floating point numbers have an exponent at most `max_exponent` in absolute value. Larger numbers are rounded to $\pm\infty$. Smaller numbers are subnormal (not implemented yet), and digits beyond $10^{-\text{max_exponent}}$ are rounded away, hence the true minimum exponent is $-\text{max_exponent} - 16$; beyond this, numbers are rounded to zero. Why this choice of limits? When computing $(a \cdot 10^n)^{(b \cdot 10^p)}$, we need to evaluate $\log(a \cdot 10^n) = \log(a) + n \log(10)$ as a fixed point number, which we manipulate as blocks

of 4 digits. Multiplying such a fixed point number by $n < 10000$ is much cheaper than larger n , because we can multiply n with each block safely.

```
9762 \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`

```
9763 \cs_new:Npn \__fp_zero_fp:N #1 { \s__fp \__fp_chk:w 0 #1 \s__fp_underflow ; }
9764 \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
`__fp_min_fp:N` numbers.

```
9765 \cs_new:Npn \__fp_min_fp:N #1
9766 {
9767   \s__fp \__fp_chk:w 1 #1
9768   { \int_eval:n { - \c__fp_max_exponent_int } }
9769   {1000} {0000} {0000} {0000} ;
9770 }
9771 \cs_new:Npn \__fp_max_fp:N #1
9772 {
9773   \s__fp \__fp_chk:w 1 #1
9774   { \int_use:N \c__fp_max_exponent_int }
9775   {9999} {9999} {9999} {9999} ;
9776 }
(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.

```
9777 \cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
9778 {
9779   \if_meaning:w 1 #1
9780   \exp_after:wN \__fp_use_ii_until_s:nnw
9781   \else:
9782   \exp_after:wN \__fp_use_i_until_s:nw
9783   \exp_after:wN 0
9784   \fi:
9785 }
(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.

```
9786 \cs_new:Npn \__fp_neg_sign:N #1
9787 { \__int_eval:w \c_two - #1 \__int_eval_end: }
(End definition for \__fp_neg_sign:N.)
```

24.3 Overflow, underflow, and exact zero

`__fp_sanitize:Nw` Expects the sign and the exponent in some order, then the significand (which we don't touch). Outputs the corresponding floating point number, possibly underflowed to ± 0 or overflowed to $\pm\infty$. The functions `__fp_underflow:w` and `__fp_overflow:w` are defined in `l3fp-traps`.

```

9788 \cs_new:Npn \__fp_sanitize:Nw #1 #2;
9789 {
9790   \if_case:w \if_int_compare:w #2 > \c__fp_max_exponent_int \c_one \else:
9791     \if_int_compare:w #2 < - \c__fp_max_exponent_int \c_two \else:
9792     \if_meaning:w 1 #1 \c_three \else: \c_zero \fi: \fi: \fi:
9793   \or: \exp_after:wN \__fp_overflow:w
9794   \or: \exp_after:wN \__fp_underflow:w
9795   \or: \exp_after:wN \__fp_sanitize_zero:w
9796   \fi:
9797   \s__fp \__fp_chk:w 1 #1 {#2}
9798 }
9799 \cs_new:Npn \__fp_sanitize:wN #1; #2 { \__fp_sanitize:Nw #2 #1; }
9800 \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.)

```

24.4 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.

```

9801 \cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
9802 {
9803   \if_meaning:w 1 #1
9804     \exp_after:wN \__fp_exp_after_normal:nNNw
9805   \else:
9806     \exp_after:wN \__fp_exp_after_special:nNNw
9807   \fi:
9808   { }
9809   #1
9810 }
9811 \cs_new:Npn \__fp_exp_after_o:nw #1 \s__fp \__fp_chk:w #2
9812 {
9813   \if_meaning:w 1 #2
9814     \exp_after:wN \__fp_exp_after_normal:nNNw
9815   \else:
9816     \exp_after:wN \__fp_exp_after_special:nNNw
9817   \fi:
9818   { #1 }
9819   #2
9820 }

```

```

9821 \cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
9822 {
9823   \if_meaning:w 1 #2
9824     \exp_after:wN \__fp_exp_after_normal:nNNw
9825   \else:
9826     \exp_after:wN \__fp_exp_after_special:nNNw
9827   \fi:
9828   { \tex_romannumeral:D -'0 #1 }
9829   #2
9830 }

```

(End definition for __fp_exp_after_o:w.)

__fp_exp_after_special:nNNw Special floating point numbers are easy to jump over since they contain few tokens.

```

9831 \cs_new:Npn \__fp_exp_after_special:nNNw #1#2#3#4;
9832 {
9833   \exp_after:wN \s__fp
9834   \exp_after:wN \__fp_chk:w
9835   \exp_after:wN #2
9836   \exp_after:wN #3
9837   \exp_after:wN #4
9838   \exp_after:wN ;
9839   #1
9840 }

```

(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.

```

9841 \cs_new:Npn \__fp_exp_after_normal:nNNw #1 1 #2 #3 #4#5#6#7;
9842 {
9843   \exp_after:wN \__fp_exp_after_normal:Nwwwww
9844   \exp_after:wN #2
9845   \__int_value:w #3 \exp_after:wN ;
9846   \__int_value:w 1 #4 \exp_after:wN ;
9847   \__int_value:w 1 #5 \exp_after:wN ;
9848   \__int_value:w 1 #6 \exp_after:wN ;
9849   \__int_value:w 1 #7 \exp_after:wN ; #1
9850 }
9851 \cs_new:Npn \__fp_exp_after_normal:Nwwwww
9852   #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
9853   { \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

```

9854 \cs_new:Npn \__fp_exp_after_array_f:w #1
9855 {
9856   \cs:w __fp_exp_after \__fp_type_from_scan:N #1 _f:nw \cs_end:
9857   { \__fp_exp_after_array_f:w }
9858   #1

```

```

9859 }
9860 \cs_new_eq:NN \__fp_exp_after_stop_f:nw \use_none:nn
(End definition for \__fp_exp_after_array_f:w.)

```

24.5 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:NNNNnw #1 #2#3#4#5 #6; { {#2#3#4#5} {#6} }
\exp_after:wN \pack:NNNNnw
\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 $-5\,0000$ (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 $4\,9995\,0000$ (the “middle shift”). That, in turn, expands `\exp_after:wN` which triggers the third computation. The third computation's value is $5\,0000\,0000 + 12345 \times 8899$, which has 9 digits. Adding $5 \cdot 10^8$ to the product allowed us to know how many digits to expect as long as the numbers to multiply are not too big; it 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 $5\,0000\,0000/10^4 + 4\,9995\,0000 = 5\,0000\,0000$. As long as the operands are in some range, the result of this second computation 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 $\text{T}_{\text{E}}\text{X}$ 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` This set of shifts allows for computations involving results in the range $[-4 \cdot 10^8, 5 \cdot 10^8 - 1]$.
`\c__fp_trailing_shift_int` Shifted values all have exactly 9 digits.

```

\c__fp_middle_shift_int 9861 \int_const:Nn \c__fp_leading_shift_int { - 5 0000 }
\c__fp_leading_shift_int 9862 \int_const:Nn \c__fp_middle_shift_int { 5 0000 * 9999 }
9863 \int_const:Nn \c__fp_trailing_shift_int { 5 0000 * 10000 }
9864 \cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }

```

(End definition for `__fp_pack:NNNNNw`.)

`__fp_pack_big:NNNNNNw` This set of shifts allows for computations involving results in the range $[-5 \cdot 10^8, 6 \cdot 10^8 - 1]$
`\c__fp_big_trailing_shift_int` (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper
`\c__fp_big_middle_shift_int` bound is due to TeX's limit of $2^{31} - 1$ on integers. The shifts are chosen to be roughly
`\c__fp_big_leading_shift_int` the mid-point of 10^9 and 2^{31} , the two bounds on 10-digit integers in TeX.

```

9865 \int_const:Nn \c__fp_big_leading_shift_int { - 15 2374 }
9866 \int_const:Nn \c__fp_big_middle_shift_int { 15 2374 * 9999 }
9867 \int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 }
9868 \cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
9869 { + #1#2#3#4#5#6 ; {#7} }

```

(End definition for `__fp_pack_big:NNNNNNw`.)

`__fp_pack_Bigg:NNNNNNw` This set of shifts allows for computations involving results in the range $[-1 \cdot 10^9, 147483647]$;
`\c__fp_Bigg_trailing_shift_int` the end-point is $2^{31} - 1 - 2 \cdot 10^9 \simeq 1.47 \cdot 10^8$. Shifted values all have exactly 10 digits.

```

\c__fp_Bigg_middle_shift_int 9870 \int_const:Nn \c__fp_Bigg_leading_shift_int { - 20 0000 }
\c__fp_Bigg_leading_shift_int 9871 \int_const:Nn \c__fp_Bigg_middle_shift_int { 20 0000 * 9999 }
9872 \int_const:Nn \c__fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
9873 \cs_new:Npn \__fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
9874 { + #1#2#3#4#5#6 ; {#7} }

```

(End definition for `__fp_pack_Bigg:NNNNNNw`.)

`_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.

```

9875 \cs_new:Npn \_fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
9876 { #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.

```

9877 \cs_new:Npn \__fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
9878 { #1 {#2#3#4#5#6#7#8#9} ; }

```

(End definition for `__fp_pack_eight:wNNNNNNNN`.)

24.6 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 $\langle rounding \rangle$ digit to be placed after the $\langle X_i \rangle$, but the choice we make involves less reshuffling.

Note that this function fails for negative $\langle shift \rangle$.

```

9879 \cs_new:Npn \_fp_decimate:nNnnnn #1
9880 {
9881   \cs:w
9882     \_fp_decimate_
9883     \if_int_compare:w \_int_eval:w #1 > \c_sixteen
9884       tiny
9885     \else:
9886       \tex_romannumeral:D \_int_eval:w #1
9887     \fi:
9888     :Nnnnn
9889   \cs_end:
9890 }
```

Each of the auxiliaries see the function $\langle f_1 \rangle$, followed by 4 blocks of 4 digits.
(End definition for `_fp_decimate:nNnnnn`.)

`_fp_decimate_:Nnnnn`
`_fp_decimate_tiny:Nnnnn`

If the $\langle shift \rangle$ is zero, or too big, life is very easy.

```

9891 \cs_new:Npn \_fp_decimate_:Nnnnn #1 #2#3#4#5
9892 { #1 0 {#2#3} {#4#5} ; }
9893 \cs_new:Npn \_fp_decimate_tiny:Nnnnn #1 #2#3#4#5
9894 { #1 1 { 0000 0000 } { 0000 0000 } 0 #2#3#4#5 ; }
```

(End definition for `_fp_decimate_:Nnnnn` and `_fp_decimate_tiny:Nnnnn`.)

`_fp_decimate_auxi:Nnnnn`
`_fp_decimate_auxii:Nnnnn`
`_fp_decimate_auxiii:Nnnnn`
`_fp_decimate_auxiv:Nnnnn`
`_fp_decimate_auxv:Nnnnn`
`_fp_decimate_auxvi:Nnnnn`
`_fp_decimate_auxvii:Nnnnn`
`_fp_decimate_auxviii:Nnnnn`
`_fp_decimate_auxix:Nnnnn`
`_fp_decimate_auxx:Nnnnn`
`_fp_decimate_auxxi:Nnnnn`
`_fp_decimate_auxxii:Nnnnn`
`_fp_decimate_auxxiii:Nnnnn`
`_fp_decimate_auxxiv:Nnnnn`
`_fp_decimate_auxxv:Nnnnn`
`_fp_decimate_auxxvi:Nnnnn`

Shifting happens in two steps: compute the $\langle rounding \rangle$ digit, and repack digits into two blocks of 8. The sixteen functions are very similar, and defined through `_fp_tmp:w`. The arguments are as follows: **#1** indicates which function is being defined; after one step of expansion, **#2** yields the “extra digits” which are then converted by `_fp_round_digit:Nw` to the $\langle rounding \rangle$ digit. This triggers the f-expansion of `_fp_decimate_pack:nnnnnnnnnw`,⁷ responsible for building two blocks of 8 digits, and removing the

⁷No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

rest. For this to work, #3 alternates between braced and unbraced blocks of 4 digits, in such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.

```

9895 \cs_new:Npn \__fp_tmp:w #1 #2 #3
9896 {
9897   \cs_new:cpn { __fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
9898   {
9899     \exp_after:wN ##1
9900     \__int_value:w
9901     \exp_after:wN \__fp_round_digit:Nw #2 ;
9902     \__fp_decimate_pack:nnnnnnnnnw #3 ;
9903   }
9904 }
9905 \__fp_tmp:w {i} {\use_none:nnn #50} { 0{#2}#3{#4}#5 }
9906 \__fp_tmp:w {ii} {\use_none:nn #5 } { 00{#2}#3{#4}#5 }
9907 \__fp_tmp:w {iii} {\use_none:n #5 } { 000{#2}#3{#4}#5 }
9908 \__fp_tmp:w {iv} { #5 } { {0000}#2{#3}#4 #5 }
9909 \__fp_tmp:w {v} {\use_none:nnn #4#5 } { 0{0000}#2{#3}#4 #5 }
9910 \__fp_tmp:w {vi} {\use_none:nn #4#5 } { 00{0000}#2{#3}#4 #5 }
9911 \__fp_tmp:w {vii} {\use_none:n #4#5 } { 000{0000}#2{#3}#4 #5 }
9912 \__fp_tmp:w {viii}{ #4#5 } { {0000}0000{#2}#3 #4 #5 }
9913 \__fp_tmp:w {ix} {\use_none:nnn #3#4+#5} { 0{0000}0000{#2}#3 #4 #5 }
9914 \__fp_tmp:w {x} {\use_none:nn #3#4+#5} { 00{0000}0000{#2}#3 #4 #5 }
9915 \__fp_tmp:w {xi} {\use_none:n #3#4+#5} { 000{0000}0000{#2}#3 #4 #5 }
9916 \__fp_tmp:w {xii} { #3#4+#5} { {0000}0000{0000}#2 #3 #4 #5 }
9917 \__fp_tmp:w {xiii}{\use_none:nnn#2#3+#4#5} { 0{0000}0000{0000}#2 #3 #4 #5 }
9918 \__fp_tmp:w {xiv} {\use_none:nn #2#3+#4#5} { 00{0000}0000{0000}#2 #3 #4 #5 }
9919 \__fp_tmp:w {xv} {\use_none:n #2#3+#4#5} { 000{0000}0000{0000}#2 #3 #4 #5 }
9920 \__fp_tmp:w {xvi} { #2#3+#4#5} { {0000}0000{0000}0000 #2 #3 #4 #5 }
(End definition for \__fp_decimate_auxi:Nnnnn and others.)

```

__fp_round_digit:Nw __fp_round_digit:Nw will receive the “extra digits” as its argument, and its expansion is triggered by __int_value:w. If the first digit is neither 0 nor 5, then it is the *rounding* digit. Otherwise, if the remaining digits are not all zero, we need to add 1 to that leading digit to get the rounding digit. Some caution is required, though, because there may be more than 10 “extra digits”, and this may overflow T_EX’s integers. Instead of feeding the digits directly to __fp_round_digit:Nw, they come split into several blocks, separated by +. Hence the first __int_eval:w here.

The computation of the *rounding* digit leaves an unfinished __int_value:w, which expands the following functions. This allows us to repack nicely the digits we keep. Those digits come as an alternation of unbraced and braced blocks of 4 digits, such that the first 5 groups of token consist in 4 single digits, and one brace group (in some order), and the next 5 have the same structure. This is followed by some digits and a semicolon.

```

9921 \cs_new:Npn \__fp_decimate_pack:nnnnnnnnnw #1#2#3#4#5
9922 { \__fp_decimate_pack:nnnnnnnw { #1#2#3#4#5 } }
9923 \cs_new:Npn \__fp_decimate_pack:nnnnnnnw #1 #2#3#4#5#6
9924 { {#1} {#2#3#4#5#6} }
(End definition for \__fp_round_digit:Nw and \__fp_decimate_pack:nnnnnnnnnw.)

```

24.7 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 $\langle fp\ var \rangle$, expanding once after that floating point. Case 1 will do $\langle some\ computation \rangle$ using the $\langle floating\ point \rangle$ (presumably compute the operation requested by the user in that non-trivial case). Case 2 will return the $\langle floating\ point \rangle$ without modifying it, removing the $\langle junk \rangle$ and expanding once after. Case 3 will close the conditional, remove the $\langle junk \rangle$ and the $\langle floating\ point \rangle$, and expand $\langle something \rangle$ next. In other cases, the “ $\langle junk \rangle$ ” is expanded, performing some other operation on the $\langle floating\ point \rangle$. We provide similar functions with two trailing $\langle floating\ points \rangle$.

`__fp_case_use:nw` This function ends a `TEX` conditional, removes junk until the next floating point, and places its first argument before that floating point, to perform some operation on the floating point.

```
9925 \cs_new:Npn \__fp_case_use:nw #1#2 \fi: #3 \s__fp { \fi: #1 \s__fp }
(End definition for \__fp_case_use:nw.)
```

`__fp_case_return:nw` This function ends a `TEX` conditional, removes junk and a floating point, and places its first argument in the input stream. A quirk is that we don’t define this function requiring a floating point to follow, simply anything ending in a semicolon. This, in turn, means that the $\langle junk \rangle$ may not contain semicolons.

```
9926 \cs_new:Npn \__fp_case_return:nw #1#2 \fi: #3 ; { \fi: #1 }
(End definition for \__fp_case_return:nw.)
```

`__fp_case_return_o:Nw` This function ends a `TEX` conditional, removes junk and a floating point, and returns its first argument (an $\langle fp\ var \rangle$) then expands once after it.

```
9927 \cs_new:Npn \__fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 ;
9928 { \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.

```
9929 \cs_new:Npn \__fp_case_return_same_o:w #1 \fi: #2 \s__fp
9930 { \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.

```
9931 \cs_new:Npn \__fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 ; #5 ;
9932 { \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.

```

9933 \cs_new:Npn \__fp_case_return_i_o:ww #1 \fi: #2 \s__fp #3 ; \s__fp #4 ;
9934 { \fi: \__fp_exp_after_o:w \s__fp #3 ; }
9935 \cs_new:Npn \__fp_case_return_ii_o:ww #1 \fi: #2 \s__fp #3 ;
9936 { \fi: \__fp_exp_after_o:w }

```

(End definition for `__fp_case_return_i_o:ww` and `__fp_case_return_ii_o:ww`.)

24.8 Small integer floating points

`__fp_small_int:wTF` Tests if the floating point argument is an integer or $\pm\infty$. If so, it is converted to an integer in the range $[-10^8, 10^8]$ and fed as a braced argument to the *⟨true code⟩*. Otherwise, the *⟨false code⟩* is performed. First filter special cases: neither `nan` nor infinities are integers. `__fp_small_int_normal:NnwTF` 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.

```

9937 \cs_new:Npn \__fp_small_int:wTF \s__fp \__fp_chk:w #1#2
9938 {
9939   \if_case:w #1 \exp_stop_f:
9940     \__fp_case_return:nw { \__fp_small_int_true:wTF 0 ; }
9941   \or: \exp_after:wN \__fp_small_int_normal:NnwTF
9942   \or:
9943     \__fp_case_return:nw
9944     {
9945       \exp_after:wN \__fp_small_int_true:wTF \__int_value:w
9946       \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
9947     }
9948   \else: \__fp_case_return:nw \use_ii:nn
9949   \fi:
9950   #2
9951 }
9952 \cs_new:Npn \__fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
9953 \cs_new:Npn \__fp_small_int_normal:NnwTF #1#2#3;
9954 {
9955   \if_int_compare:w #2 > \c_zero
9956     \__fp_decimate:nNnnnn { \c_sixteen - #2 }
9957     \__fp_small_int_test:NnnwNTF
9958     #3 #1 {#2}
9959   \else:
9960     \exp_after:wN \use_iii:nnn
9961   \fi:
9962 ;

```

```

9963 }
9964 \cs_new:Npn \__fp_small_int_test:NnnwNnw #1#2#3#4; #5#6
9965 {
9966   \if_meaning:w 0 #1
9967     \exp_after:wN \__fp_small_int_true:wTF
9968     \__int_value:w \if_meaning:w 2 #5 - \fi:
9969     \if_int_compare:w #6 > \c_eight
9970       1 0000 0000
9971     \else:
9972       #3
9973     \fi:
9974   \else:
9975     \use_i:nn
9976   \fi:
9977 }

```

(End definition for __fp_small_int:wTF.)

24.9 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.

```

9978 \cs_new:Npn \__fp_array_count:n #1
9979 {
9980   \int_use:N \__int_eval:w \c_zero
9981   \__fp_array_count_loop:Nw #1 { ? \__prg_break: } ;
9982   \__prg_break_point:
9983   \__int_eval_end:
9984 }
9985 \cs_new:Npn \__fp_array_count_loop:Nw #1#2;
9986 { \use_none:n #1 + \c_one \__fp_array_count_loop:Nw }

```

(End definition for __fp_array_count:n.)

24.10 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.

```

9987 \cs_new:Npn \__fp_expand:n #1
9988 {
9989   \__fp_expand_loop:nwnN { }
9990   #1 \prg_do_nothing:
9991   \s__fp_mark { } \__fp_expand_loop:nwnN
9992   \s__fp_mark { } \__fp_use_i_until:s:nw ;

```

```

9993 }
9994 \cs_new:Npn \__fp_expand_loop:nwnN #1#2 \s__fp_mark #3 #4
9995 {
9996   \exp_after:wN #4 \tex_romannumeral:D -‘0
9997   #2
9998   \s__fp_mark { #3 #1 } #4
9999 }

```

(End definition for __fp_expand:n.)

24.11 Messages

Using a floating point directly is an error.

```

10000 \__msg_kernel_new:nnnn { kernel } { misused-fp }
10001 { A~floating~point~with~value~‘#1’~was~misused. }
10002 {
10003   To~obtain~the~value~of~a~floating~point~variable,~use~
10004   ‘\token_to_str:N \fp_to_decimal:N’,~
10005   ‘\token_to_str:N \fp_to_scientific:N’,~or~other~
10006   conversion~functions.
10007 }
10008 </initex | package>

```

25 l3fp-traps Implementation

```

10009 <*initex | package>
10010 <@@=fp>

```

Exceptions should be accessed by an n-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

25.1 Flags

`\fp_flag_off:n` Function to turn a flag off. Simply undefine it.

```

10011 \cs_new_protected:Npn \fp_flag_off:n #1
10012 { \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 183.)

`\fp_flag_on:n` Function to turn a flag on expandably: use TeX’s automatic assignment to `\scan_stop:.`

```

10013 \cs_new:Npn \fp_flag_on:n #1
10014 { \exp_args:Nc \use_none:n { l__fp_ #1 _flag_token } }

```

(End definition for `\fp_flag_on:n`. This function is documented on page 183.)

```

\fp_if_flag_on_p:n Returns true if the flag is on, false otherwise.
\fp_if_flag_on:nTF
10015 \prg_new_conditional:Npnn \fp_if_flag_on:n #1 { p , T , F , TF }
10016 {
10017   \if_cs_exist:w l__fp_ #1 _flag_token \cs_end:
10018   \prg_return_true:
10019   \else:
10020   \prg_return_false:
10021   \fi:
10022 }

```

(End definition for `\fp_if_flag_on:n`. These functions are documented on page 183.)

```

\l_fp_invalid_operation_flag_token The IEEE standard defines five exceptions. We currently don't support the "inexact"
\l_fp_division_by_zero_flag_token exception.
\l_fp_overflow_flag_token
\l__fp_underflow_flag_token
10023 \cs_new_eq:NN \l__fp_invalid_operation_flag_token \tex_undefined:D
10024 \cs_new_eq:NN \l__fp_division_by_zero_flag_token \tex_undefined:D
10025 \cs_new_eq:NN \l__fp_overflow_flag_token \tex_undefined:D
10026 \cs_new_eq:NN \l__fp_underflow_flag_token \tex_undefined:D

```

(End definition for `\l__fp_invalid_operation_flag_token` and others.)

25.2 Traps

Exceptions can be trapped to obtain custom behaviour. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an N -type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives $+\infty$. Currently, the inexact exception is entirely ignored.

The behaviour when an exception occurs is controlled by the definitions of the functions

- `__fp_invalid_operation:nnw`,
- `__fp_invalid_operation_o:Nww`,
- `__fp_invalid_operation_tl_o:ff`,
- `__fp_division_by_zero_o:Nnw`,
- `__fp_division_by_zero_o:NNww`,
- `__fp_overflow:w`,
- `__fp_underflow:w`.

Rather than changing them directly, we provide a user interface as `\fp_trap:nn` $\{\langle exception \rangle\} \{\langle way of trapping \rangle\}$, where the $\langle way of trapping \rangle$ is one of `error`, `flag`, or `none`.

We also provide `__fp_invalid_operation_o:nw`, defined in terms of `__fp_invalid_operation:nnw`.

`\fp_trap:nn`

```

10027 \cs_new_protected:Npn \fp_trap:nn #1#2
10028 {
10029   \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
10030   {
10031     \clist_if_in:nnTF
10032     { invalid_operation , division_by_zero , overflow , underflow }
10033     {#1}
10034     {
10035       \__msg_kernel_error:nnxx { kernel }
10036       { unknown-fpu-trap-type } {#1} {#2}
10037     }
10038     { \__msg_kernel_error:nnx { kernel } { unknown-fpu-exception } {#1} }
10039   }
10040 }
```

(End definition for `\fp_trap:nn`. This function is documented on page 184.)

`__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`

```

10041 \cs_new_protected_nopar:Npn \__fp_trap_invalid_operation_set_error:
10042 { \__fp_trap_invalid_operation_set:N \prg_do_nothing: }
10043 \cs_new_protected_nopar:Npn \__fp_trap_invalid_operation_set_flag:
10044 { \__fp_trap_invalid_operation_set:N \use_none:nnnnn }
10045 \cs_new_protected_nopar:Npn \__fp_trap_invalid_operation_set_none:
10046 { \__fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
10047 \cs_new_protected:Npn \__fp_trap_invalid_operation_set:N #1
10048 {
10049   \exp_args:Nno \use:n
10050   { \cs_set:Npn \__fp_invalid_operation:nnw ##1##2##3; }
10051   {
10052     #1
10053     \__fp_error:nnfn { invalid } {##2} { \fp_to_tl:n { ##3; } } { }
10054     \fp_flag_on:n { invalid_operation }
10055     ##1
10056   }
10057   \exp_args:Nno \use:n
10058   { \cs_set:Npn \__fp_invalid_operation_o:Nww ##1##2; ##3; }
10059   {
10060     #1
10061     \__fp_error:nffn { invalid-ii }
10062     { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
10063     \fp_flag_on:n { invalid_operation }

```



```

10064         \exp_after:wN \c_nan_fp
10065     }
10066     \exp_args:Nno \use:n
10067     { \cs_set:Npn \__fp_invalid_operation_tl_o:ff ##1##2 }
10068     {
10069         #1
10070         \__fp_error:nffn { invalid } {##1} {##2} { }
10071         \fp_flag_on:n { invalid_operation }
10072         \exp_after:wN \c_nan_fp
10073     }
10074 }

```

(End definition for __fp_trap_invalid_operation_set_error: and others.)

```

\__fp_trap_division_by_zero_set_error:
\__fp_trap_division_by_zero_set_flag:
\__fp_trap_division_by_zero_set_none:
\__fp_trap_division_by_zero_set:N

```

We provide three types of trapping for invalid operations and division by zero: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In all cases, the function must produce a result, namely its first argument, $\pm\infty$ or NaN.

```

10075 \cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_error:
10076 { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
10077 \cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_flag:
10078 { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
10079 \cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_none:
10080 { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
10081 \cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
10082 {
10083     \exp_args:Nno \use:n
10084     { \cs_set:Npn \__fp_division_by_zero_o:NNw ##1##2##3; }
10085     {
10086         #1
10087         \__fp_error:nnfn { zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
10088         \fp_flag_on:n { division_by_zero }
10089         \exp_after:wN ##1
10090     }
10091     \exp_args:Nno \use:n
10092     { \cs_set:Npn \__fp_division_by_zero_o:NNww ##1##2##3; ##4; }
10093     {
10094         #1
10095         \__fp_error:nffn { zero-div-ii }
10096         { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
10097         \fp_flag_on:n { division_by_zero }
10098         \exp_after:wN ##1
10099     }
10100 }

```

(End definition for __fp_trap_division_by_zero_set_error: and others.)

```

\__fp_trap_overflow_set_error:
\__fp_trap_overflow_set_flag:
\__fp_trap_overflow_set_none:
\__fp_trap_overflow_set:N
\__fp_trap_underflow_set_error:
\__fp_trap_underflow_set_flag:
\__fp_trap_underflow_set_none:
\__fp_trap_underflow_set:N
\__fp_trap_overflow_set:NnNn

```

Just as for invalid operations and division by zero, the three different behaviours are obtained by feeding \prg_do_nothing:, \use_none:nnnnn or \use_none:nnnnnnn to an auxiliary, with a further auxiliary common to overflow and underflow functions. In most cases, the argument of the __fp_overflow:w and __fp_underflow:w functions will

be an (almost) normal number (with an exponent outside the allowed range), and the error message thus displays that number together with the result to which it overflowed or underflowed. For extreme cases such as `10 ** 1e9999`, the exponent would be too large for `TEX`, and `__fp_overflow:w` receives $\pm\infty$ (`__fp_underflow:w` would receive ± 0); then we cannot do better than simply say an overflow or underflow occurred.

```

10101 \cs_new_protected_nopar:Npn \__fp_trap_overflow_set_error:
10102 { \__fp_trap_overflow_set:N \prg_do_nothing: }
10103 \cs_new_protected_nopar:Npn \__fp_trap_overflow_set_flag:
10104 { \__fp_trap_overflow_set:N \use_none:nnnnn }
10105 \cs_new_protected_nopar:Npn \__fp_trap_overflow_set_none:
10106 { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
10107 \cs_new_protected:Npn \__fp_trap_overflow_set:N #1
10108 { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
10109 \cs_new_protected_nopar:Npn \__fp_trap_underflow_set_error:
10110 { \__fp_trap_underflow_set:N \prg_do_nothing: }
10111 \cs_new_protected_nopar:Npn \__fp_trap_underflow_set_flag:
10112 { \__fp_trap_underflow_set:N \use_none:nnnnn }
10113 \cs_new_protected_nopar:Npn \__fp_trap_underflow_set_none:
10114 { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
10115 \cs_new_protected:Npn \__fp_trap_underflow_set:N #1
10116 { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }
10117 \cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4
10118 {
10119   \exp_args:Nno \use:n
10120   { \cs_set:cpn { __fp_ #2 :w } \s__fp \__fp_chk:w ##1##2##3; }
10121   {
10122     #1
10123     \__fp_error:nffn
10124     { flow \if_meaning:w 1 ##1 -to \fi: }
10125     { \fp_to_tl:n { \s__fp \__fp_chk:w ##1##2##3; } }
10126     { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
10127     {#2}
10128     \fp_flag_on:n {#2}
10129     #3 ##2
10130   }
10131 }

```

(End definition for __fp_trap_overflow_set_error: and others.)

<pre> __fp_invalid_operation:nnw __fp_invalid_operation_o:Nww __fp_invalid_operation_tl_o:ff __fp_division_by_zero_o:Nnw __fp_division_by_zero_o:NNww __fp_overflow:w __fp_underflow:w </pre>	<pre> 10132 \cs_new:Npn __fp_invalid_operation:nnw #1#2#3; { } 10133 \cs_new:Npn __fp_invalid_operation_o:Nww #1#2; #3; { } 10134 \cs_new:Npn __fp_invalid_operation_tl_o:ff #1 #2 { } 10135 \cs_new:Npn __fp_division_by_zero_o:Nnw #1#2#3; { } 10136 \cs_new:Npn __fp_division_by_zero_o:NNww #1#2#3; #4; { } 10137 \cs_new:Npn __fp_overflow:w { } 10138 \cs_new:Npn __fp_underflow:w { } 10139 \fp_trap:nn { invalid_operation } { error } </pre>
--	--

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.

```

10140 \fp_trap:nn { division_by_zero } { flag }
10141 \fp_trap:nn { overflow } { flag }
10142 \fp_trap:nn { underflow } { flag }
(End definition for \__fp_invalid_operation:nw 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.

```

10143 \cs_new_nopar:Npn \__fp_invalid_operation_o:nw
10144 { \__fp_invalid_operation:nw { \exp_after:wN \c_nan_fp } }
10145 \cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }
(End definition for \__fp_invalid_operation_o:nw and \__fp_invalid_operation_o:fw.)

```

25.3 Errors

```

\__fp_error:nnnn
\__fp_error:nnfn 10146 \cs_new:Npn \__fp_error:nnnn #1
\__fp_error:nffn 10147 { \__msg_kernel_expandable_error:nnnnn { kernel } { fp - #1 } }
10148 \cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff }
(End definition for \__fp_error:nnnn, \__fp_error:nnfn, and \__fp_error:nffn.)

```

25.4 Messages

Some messages.

```

10149 \__msg_kernel_new:nnnn { kernel } { unknown-fpu-exception }
10150 { The~FPU~exception~'#1'~is~not~known:~that~trap~will~never~be~triggered. }
10151 {
10152   The~only~exceptions~to~which~traps~can~be~attached~are \\
10153   \iow_indent:n
10154   {
10155     * ~ invalid_operation \\
10156     * ~ division_by_zero \\
10157     * ~ overflow \\
10158     * ~ underflow
10159   }
10160 }
10161 \__msg_kernel_new:nnnn { kernel } { unknown-fpu-trap-type }
10162 { The~FPU~trap~type~'#2'~is~not~known. }
10163 {
10164   The~trap~type~must~be~one~of \\
10165   \iow_indent:n
10166   {
10167     * ~ error \\
10168     * ~ flag \\
10169     * ~ none
10170   }
10171 }
10172 \__msg_kernel_new:nnn { kernel } { fp-flow }
10173 { An ~ #3 ~ occurred. }

```

```

10174 \_msg_kernel_new:nnn { kernel } { fp-flow-to }
10175 { #1 ~ #3 ed ~ to ~ #2 . }
10176 \_msg_kernel_new:nnn { kernel } { fp-zero-div }
10177 { Division-by-zero-in~ #1 (#2) }
10178 \_msg_kernel_new:nnn { kernel } { fp-zero-div-ii }
10179 { Division-by-zero-in~ (#1) #3 (#2) }
10180 \_msg_kernel_new:nnn { kernel } { fp-invalid }
10181 { Invalid-operation~ #1 (#2) }
10182 \_msg_kernel_new:nnn { kernel } { fp-invalid-ii }
10183 { Invalid-operation~ (#1) #3 (#2) }
10184 </initex | package>

```

26 l3fp-round implementation

```

10185 <*initex | package>
10186 <@@=fp>

```

26.1 Rounding tools

Floating point operations often yield a result that cannot be exactly represented in a significand with 16 digits. In that case, we need to round the exact result to a representable number. The IEEE standard defines four rounding modes:

- Round to nearest: round to the representable floating point number whose absolute difference with the exact result is the smallest. If the exact result lies exactly at the mid-point between two consecutive representable floating point numbers, round to the floating point number whose last digit is even.
- Round towards negative infinity: round to the greatest floating point number not larger than the exact result.
- Round towards zero: round to a floating point number with the same sign as the exact result, with the largest absolute value not larger than the absolute value of the exact result.
- Round towards positive infinity: round to the least floating point number not smaller than the exact result.

This is not fully implemented in `l3fp` yet, and transcendental functions fall back on the “round to nearest” mode. All rounding for basic algebra is done through the functions defined in this module, which can be redefined to change their rounding behaviour (but there is not interface for that yet).

The rounding tools available in this module are many variations on a base function `_fp_round:NNN`, which expands to `\c_zero` or `\c_one` depending on whether the final result should be rounded up or down.

- `_fp_round:NNN <sign> <digit1> <digit2>` can expand to `\c_zero` or `\c_one`.

- `__fp_round_s:NNNw` $\langle sign \rangle \langle digit_1 \rangle \langle digit_2 \rangle \langle more\ digits \rangle$; 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.

```

10187 \cs_new:Npn \__fp_round_return_one:
10188 { \exp_after:wN \c_one \tex_romannumeral:D }
10189 \cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
10190 {
10191   \if_meaning:w 2 #1
10192   \if_int_compare:w #3 > \c_zero
10193     \__fp_round_return_one:
10194   \fi:
10195   \fi:
10196   \c_zero
10197 }
10198 \cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { \c_zero }
10199 \cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
10200 {
10201   \if_meaning:w 0 #1
10202   \if_int_compare:w #3 > \c_zero
10203     \__fp_round_return_one:
10204   \fi:
10205   \fi:
10206   \c_zero
10207 }
10208 \cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
10209 {
10210   \if_int_compare:w #3 > \c_five
10211     \__fp_round_return_one:
10212   \else:
10213     \if_meaning:w 5 #3
10214       \if_int_odd:w #2 \exp_stop_f:

```

```

10215         \__fp_round_return_one:
10216         \fi:
10217         \fi:
10218         \fi:
10219         \c_zero
10220     }
10221 \cs_new_eq:NN \__fp_round:NNN \__fp_round_to_nearest:NNN
(End definition for \__fp_round:NNN.)

```

`__fp_round_s:NNNw` Similar to `__fp_round:NNN`, but with an extra semicolon, this function expands to `\c_zero` ; if rounding $\langle final\ sign \rangle \langle digit \rangle . \langle more\ digits \rangle$ to an integer truncates, and to `\c_one` ; otherwise. The $\langle more\ digits \rangle$ part must be a digit, followed by something that does not overflow a `\int_use:N __int_eval:w` construction. The only relevant information about this piece is whether it is zero or not.

```

10222 \cs_new:Npn \__fp_round_s:NNNw #1 #2 #3 #4;
10223 {
10224     \exp_after:wN \__fp_round:NNN
10225     \exp_after:wN #1
10226     \exp_after:wN #2
10227     \int_use:N \__int_eval:w
10228     \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
10229         \if_meaning:w 5 #3 1 \fi:
10230         \exp_stop_f:
10231         \if_int_compare:w \__int_eval:w #4 > \c_zero
10232             1 +
10233         \fi:
10234     \fi:
10235     #3
10236 ;
10237 }
(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.

```

10238 \cs_new:Npn \__fp_round_digit:Nw #1 #2;
10239 {
10240     \if_int_odd:w \if_meaning:w 0 #1 \c_one \else:
10241         \if_meaning:w 5 #1 \c_one \else:
10242         \c_zero \fi: \fi:
10243     \if_int_compare:w \__int_eval:w #2 > \c_zero
10244         \__int_eval:w \c_one +
10245     \fi:
10246     \fi:
10247     #1
10248 }
(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.

```

10249 \cs_new:Npn \__fp_round_to_ninf_neg:NNN #1 #2 #3
10250 {
10251   \if_meaning:w 0 #1
10252     \if_int_compare:w #3 > \c_zero
10253       \__fp_round_return_one:
10254     \fi:
10255   \fi:
10256   \c_zero
10257 }
10258 \cs_new:Npn \__fp_round_to_zero_neg:NNN #1 #2 #3
10259 {
10260   \if_int_compare:w #3 > \c_zero
10261     \__fp_round_return_one:
10262   \fi:
10263   \c_zero
10264 }
10265 \cs_new:Npn \__fp_round_to_pinf_neg:NNN #1 #2 #3
10266 {
10267   \if_meaning:w 2 #1
10268     \if_int_compare:w #3 > \c_zero
10269       \__fp_round_return_one:
10270     \fi:
10271   \fi:
10272   \c_zero
10273 }
10274 \cs_new_eq:NN \__fp_round_to_nearest_neg:NNN \__fp_round_to_nearest:NNN
10275 \cs_new_eq:NN \__fp_round_neg:NNN \__fp_round_to_nearest_neg:NNN
(End definition for \__fp_round_neg:NNN.)

```

26.2 The round function

`__fp_round_o:Nw` This function expects one or two arguments.

```

10276 \cs_new:Npn \__fp_round_o:Nw #1#2 @
10277 {
10278   \if_case:w
10279     \__int_eval:w \__fp_array_count:n {#2} - \c_one \__int_eval_end:
10280     \__fp_round:Nwn #1 #2 {0} \tex_romannumeral:D
10281   \or: \__fp_round:Nww #1 #2 \tex_romannumeral:D
10282   \else:
10283     \__fp_error:nffn { num-args }
10284     { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }

```

```

10285 \exp_after:wN \c_nan_fp \tex_romannumeral:D
10286 \fi:
10287 -'0
10288 }

```

(End definition for _fp_round_o:Nw.)

_fp_round_name_from_cs:N

```

10289 \cs_new:Npn \_fp_round_name_from_cs:N #1
10290 {
10291   \cs_if_eq:NNTF #1 \_fp_round_to_zero:NNN { trunc }
10292   {
10293     \cs_if_eq:NNTF #1 \_fp_round_to_ninf:NNN { floor }
10294     {
10295       \cs_if_eq:NNTF #1 \_fp_round_to_pinf:NNN { ceil }
10296       { round }
10297     }
10298   }
10299 }

```

(End definition for _fp_round_name_from_cs:N.)

_fp_round:Nww

_fp_round:Nwn

_fp_round_normal:NwNNnw

_fp_round_normal:NnnwNNnn

_fp_round_pack:Nw

_fp_round_normal:NNwNnn

_fp_round_normal_end:wwNnn

_fp_round_special:NwwNnn

_fp_round_special_aux:Nw

```

10300 \cs_new:Npn \_fp_round:Nww #1#2 ; #3 ;
10301 {
10302   \_fp_small_int:wTF #3; { \_fp_round:Nwn #1#2; }
10303   {
10304     \_fp_invalid_operation_tl_o:ff
10305     { \_fp_round_name_from_cs:N #1 }
10306     { \_fp_array_to_clist:n { #2; #3; } }
10307   }
10308 }
10309 \cs_new:Npn \_fp_round:Nwn #1 \s__fp \_fp_chk:w #2#3#4; #5
10310 {
10311   \if_meaning:w 1 #2
10312     \exp_after:wN \_fp_round_normal:NwNNnw
10313     \exp_after:wN #1
10314     \_int_value:w #5
10315   \else:
10316     \exp_after:wN \_fp_exp_after_o:w
10317     \fi:
10318     \s__fp \_fp_chk:w #2#3#4;
10319   }
10320 \cs_new:Npn \_fp_round_normal:NwNNnw #1#2 \s__fp \_fp_chk:w 1#3#4#5;
10321 {
10322   \_fp_decimate:nNnnnn { \c_sixteen - #4 - #2 }
10323   \_fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
10324 }
10325 \cs_new:Npn \_fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
10326 {
10327   \exp_after:wN \_fp_round_normal:NNwNnn

```



```

10328 \int_use:N \__int_eval:w
10329 \if_int_compare:w #2 > \c_zero
10330   1 \__int_value:w #2
10331   \exp_after:wN \__fp_round_pack:Nw
10332   \int_use:N \__int_eval:w 1#3 +
10333 \else:
10334   \if_int_compare:w #3 > \c_zero
10335   1 \__int_value:w #3 +
10336 \fi:
10337 \fi:
10338 \exp_after:wN #5
10339 \exp_after:wN #6
10340 \use_none:n nnnnnnn #3
10341 #1
10342 \__int_eval_end:
10343 0000 0000 0000 0000 ; #6
10344 }
10345 \cs_new:Npn \__fp_round_pack:Nw #1
10346 { \if_meaning:w 2 #1 + \c_one \fi: \__int_eval_end: }
10347 \cs_new:Npn \__fp_round_normal:NNwNnn #1 #2
10348 {
10349   \if_meaning:w 0 #2
10350   \exp_after:wN \__fp_round_special:NwwNnn
10351   \exp_after:wN #1
10352   \fi:
10353   \__fp_pack_twice_four:wNNNNNNNN
10354   \__fp_pack_twice_four:wNNNNNNNN
10355   \__fp_round_normal_end:wwNnn
10356   ; #2
10357 }
10358 \cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
10359 {
10360   \exp_after:wN \__fp_exp_after_o:w \tex_romannumeral:D -'0
10361   \__fp_sanitiz:Nw #3 #4 ; #1 ;
10362 }
10363 \cs_new:Npn \__fp_round_special:NwwNnn #1#2;#3;#4#5#6
10364 {
10365   \if_meaning:w 0 #1
10366   \__fp_case_return:nw
10367   { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
10368 \else:
10369   \exp_after:wN \__fp_round_special_aux:Nw
10370   \exp_after:wN #4
10371   \int_use:N \__int_eval:w \c_one
10372   \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
10373 \fi:
10374 ;
10375 }
10376 \cs_new:Npn \__fp_round_special_aux:Nw #1#2;
10377 {

```

```

10378 \exp_after:wN \_fp_exp_after_o:w \tex_romannumeral:D -‘0
10379 \_fp_sanitize:Nw #1#2; {1000}{0000}{0000}{0000};
10380 }
(End definition for \_fp_round:Nww and \_fp_round:Nwn.)
10381 </initex | package>

```

27 l3fp-parse implementation

```

10382 <*initex | package>
10383 <@@=fp>

```

27.1 Work plan

The task at hand is non-trivial, and some previous failed attempts show that the code leads to unreadable logs, so we had better get it (almost) right the first time. Let us first describe our goal, then discuss the design precisely before writing any code.

`_fp_parse:n` Evaluates the *<floating point expression>* and leaves the result in the input stream as an internal floating point number. This function forms the basis of almost all public `l3fp` functions. During evaluation, each token is fully `f`-expanded.

T_EXhackers note: Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as `\int_use:N`. Invalid tokens remaining after `f`-expansion will lead to unrecoverable low-level T_EX errors.

(End definition for `_fp_parse:n`.)

Floating point expressions are composed of numbers, given in various forms, infix operators, such as `+`, `**`, or `,` (which joins two numbers into a list), and prefix operators, such as the unary `-`, functions, or opening parentheses. Here is a list of precedences which control the order of evaluation (some distinctions are irrelevant for the order of evaluation, but serve as signals), from the tightest binding to the loosest binding.

- 16 Function calls with multiple arguments.
- 15 Function calls expecting exactly one argument.
- 14 Binary `**` and `^` (right to left).
- 12 Unary `+`, `-`, `!` (right to left).
- 10 Binary `*`, `/`, and juxtaposition (implicit `*`).
- 9 Binary `+` and `-`.
- 7 Comparisons.
- 5 Logical `and`, denoted by `&&`.
- 4 Logical `or`, denoted by `||`.

- 3 Ternary operator `?:`, piece `?`.
- 2 Ternary operator `?:`, piece `:`.
- 1 Commas, and parentheses accepting commas.
- 0 Parentheses expecting exactly one argument.
- 1 Start and end of the expression.

27.1.1 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.

27.1.2 Precedence and infix operators

The various operators we will encounter have different precedences, which influence the order of calculations: $1 + 2 \times 3 = 1 + (2 \times 3)$ because \times has a higher precedence than $+$. The true analog of our macro `\operand:w` must thus take care of that. When looking for an operand, it needs to perform calculations until reaching an operator which has lower precedence than the one which called `\operand:w`. This means that `\operand:w` must know what the previous binary operator is, or rather, its precedence: we thus rename it `\operand:Nw`. Let us describe as an example how the calculation $41 - 2^3 * 4 + 5$ will be done. Here, we abuse notations: the first argument of `\operand:Nw` should be an integer constant (`\c_three`, `\c_nine`, ...) equal to the precedence of the given operator, not directly the operator itself.

- Clean up 41 and find $-$. We call `\operand:Nw -` to find the second operand.
- Clean up 2 and find \wedge .
- Compare the precedences of $-$ and \wedge . Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to `\operand:Nw \wedge`.
- Clean up 3 and find $*$.
- Compare the precedences of \wedge and $*$. Since the former is higher, `\operand:Nw \wedge` has found the second operand of the exponentiation, which is computed: $2^3 = 8$.
- We now have $41 + 8 * 4 + 5$, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 8?

- Compare the precedences of $-$ and $*$. Since the latter is higher, we are not done with 8. Call `\operand:Nw *` to find the second operand of the multiplication.
- Clean up 4, and find $-$.
- Compare the precedences of $*$ and $-$. Since the former is higher, `\operand:Nw *` has found the second operand of the multiplication, which is computed: $8 * 4 = 32$.
- We now have $41+32+5$, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 32?
- Compare the precedences of $-$ and $+$. Since they are equal, `\operand:Nw -` has found the second operand for the subtraction, which is computed: $41 - 32 = 9$.
- We now have $9+5$.

The procedure above stops short of performing all computations, but adding a surrounding call to `\operand:Nw` with a very low precedence ensures that all computations will be performed before `\operand:Nw` is done. Adding a trailing marker with the same very low precedence prevents the surrounding `\operand:Nw` from going beyond the marker.

The pattern above to find an operand for a given operator, is to find one number and the next operator, then compare precedences to know if the next computation should be done. If it should, then perform it after finding its second operand, and look at the next operator, then compare precedences to know if the next computation should be done. This continues until we find that the next computation should not be done. Then, we stop.

We are now ready to get a bit more technical and describe which of the `l3fp-parse` functions correspond to each step above.

First, `__fp_parse_operand:Nw` is the `\operand:Nw` function above, with small modifications due to expansion issues discussed later. We denote by $\langle precedence \rangle$ the argument of `__fp_parse_operand:Nw`, that is, the precedence of the binary operator whose operand we are trying to find. The basic action is to read numbers from the input stream. This is done by `__fp_parse_one:Nw`. A first approximation of this function is that it reads one $\langle number \rangle$, performing no computation, and finds the following binary $\langle operator \rangle$. Then it expands to

$$\langle number \rangle \\ __fp_parse_infix_ \langle operator \rangle :N \langle precedence \rangle$$

expanding the `infix` auxiliary before leaving the above in the input stream.

We now explain the `infix` auxiliaries. We need some flexibility in how we treat the case of equal precedences: most often, the first operation encountered should be performed, such as $1-2-3$ being computed as $(1-2)-3$, but 2^3^4 should be evaluated as $2^{(3^4)}$ instead. For this reason, and to support the equivalence between `**` and `^` more easily, each binary operator is converted to a control sequence `__fp_parse_infix_ \langle operator \rangle :N` when it is encountered for the first time. Instead of passing both precedences to a test function to do the comparison steps above, we pass the $\langle precedence \rangle$ (of the earlier operator) to the `infix` auxiliary for the following $\langle operator \rangle$, to know whether to perform the computation of the $\langle operator \rangle$. If it should not be performed, the `infix` auxiliary expands to

```
@ \use_none:n \__fp_parse_infix_⟨operator⟩:N
```

and otherwise it calls `__fp_parse_operand:Nw` with the precedence of the $\langle operator \rangle$ to find its second operand $\langle number_2 \rangle$ and the next $\langle operator_2 \rangle$, and expands to

```
@ \__fp_parse_apply_binary:NwNwN
⟨operator⟩ ⟨number_2⟩
@ \__fp_parse_infix_⟨operator_2⟩:N
```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand $\langle number \rangle$ is before the `infix` function in the input stream. This is why we stop the expansion here and give control to another function to close the loop.

A definition of `__fp_parse_operand:Nw` $\langle precedence \rangle$ with some of the expansion control removed is

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN ⟨precedence⟩
\tex_romannumeral:D -‘0
\__fp_parse_one:Nw ⟨precedence⟩
```

This expands `__fp_parse_one:Nw` $\langle precedence \rangle$ completely, which finds a number, wraps the next $\langle operator \rangle$ into an `infix` function, feeds this function the $\langle precedence \rangle$, and expands it, yielding either

```
\__fp_parse_continue:NwN ⟨precedence⟩
⟨number⟩ @
\use_none:n \__fp_parse_infix_⟨operator⟩:N
```

or

```
\__fp_parse_continue:NwN ⟨precedence⟩
⟨number⟩ @
\__fp_parse_apply_binary:NwNwN
⟨operator⟩ ⟨number_2⟩
@ \__fp_parse_infix_⟨operator_2⟩: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 $\langle number \rangle @ _ _ \text{fp_parse_infix_} \langle operator \rangle :N$. In the second case, `#3` is `__fp_parse_apply_binary:NwNwN`, whose role is to compute $\langle number \rangle \langle operator \rangle \langle number_2 \rangle$ and to prepare for the next comparison of precedences: first we get

```

\__fp_parse_apply_binary:NwNwN
<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.

27.1.3 Prefix operators, parentheses, and functions

Prefix operators (unary `-`, `+`, `!`) and parentheses are taken care of by the same mechanism, and functions (`sin`, `exp`, etc.) as well. Finding the argument of the unary `-`, for instance, is very similar to grabbing the second operand of a binary infix operator, with a subtle precedence explained below. Once that operand is found, the operator can be applied to it (for the unary `-`, this simply flips the sign). A left parenthesis is just a prefix operator with a very low precedence equal to that of the closing parenthesis (which is treated as an infix operator, since it normally appears just after numbers), so that all computations are performed until the closing parenthesis. The prefix operator associated to the left parenthesis does not alter its argument, but it removes the closing parenthesis (with some checks).

Prefix operators are the reason why we only summarily described the function `__fp_parse_one:Nw` earlier. This function is responsible for reading in the input stream the first possible `<number>` and the next infix `<operator>`. If what follows `__fp_parse_one:Nw <precedence>` is a prefix operator, then we must find the operand of this prefix operator through a nested call to `__fp_parse_operand:Nw` with the appropriate precedence, then apply the operator to the operand found to yield the result of `__fp_parse_one:Nw`. So far, all is simple.

The unary operators `+`, `-`, `!` complicate things a little bit: `-3**2` should be $-(3^2) = -9$, and not $(-3)^2 = 9$. This would easily be done by giving `-` a lower precedence, equal to that of the infix `+` and `-`. Unfortunately, this fails in cases such as `3**-2*4`, yielding $3^{-2 \times 4}$ instead of the correct $3^{-2} \times 4$. A second attempt would be to call `__fp_parse_operand:Nw` with the `<precedence>` of the previous operator, but `0>-2+3` is then parsed as `0>-(2+3)`: the addition is performed because it binds more tightly than the comparison which precedes `-`. The correct approach is for a unary `-` to perform operations whose precedence is greater than both that of the previous operation, and

that of the unary `-` itself. The unary `-` is given a precedence higher than multiplication and division. This does not lead to any surprising result, since $-(x/y) = (-x)/y$ and similarly for multiplication, and it reduces the number of nested calls to `_fp_parse_operand:Nw`.

Functions are implemented as prefix operators with very high precedence, so that their argument is the first number that can possibly be built.

Note that contrarily to the `infix` functions discussed earlier, the `prefix` functions do perform tests on the previous *precedence* to decide whether to find an argument or not, since we know that we need a number, and must never stop there.

27.1.4 Numbers and reading tokens one by one

So far, we have glossed over one important point: what is a “number”? A number is typically given in the form $\langle \textit{significand} \rangle \mathbf{e} \langle \textit{exponent} \rangle$, where the $\langle \textit{significand} \rangle$ is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent “ $\mathbf{e} \langle \textit{exponent} \rangle$ ” is optional and is composed of an exponent mark `e` followed by a possibly empty string of signs `+` or `-` and a non-empty string of decimal digits. The $\langle \textit{significand} \rangle$ can also be an integer, dimension, skip, or muskip variable, in which case dimensions are converted from points (or mu units) to floating points, and the $\langle \textit{exponent} \rangle$ can also be an integer variable. Numbers can also be given as floating point variables, or as named constants such as `nan`, `inf` or `pi`. We may add more types in the future.

When `_fp_parse_one:Nw` is looking for a “number”, here is what happens.

- If the next token is a control sequence with the meaning of `\scan_stop:`, it can be: `\s_fp`, in which case our job is done, as what follows is an internal floating point number, or `\s_fp_mark`, in which case the expression has come to an early end, as we are still looking for a number here, or something else, in which case we consider the control sequence to be a bad variable resulting from `c`-expansion.
- If the next token is a control sequence with a different meaning, we assume that it is a register, unpack it with `\tex_the:D`, and use its value (in `pt` for dimensions and skips, `mu` for muskips) as the $\langle \textit{significand} \rangle$ of a number: we look for an exponent.
- If the next token is a digit, we remove any leading zeros, then read a significand larger than 1 if the next character is a digit, read a significand smaller than 1 if the next character is a period, or we have found a significand equal to 0 otherwise, and look for an exponent.
- If the next token is a letter, we collect more letters until the first non-letter: the resulting word may denote a function such as `asin`, a constant such as `pi` or be unknown. In the first case, we call `_fp_parse_operand:Nw` to find the argument of the function, then apply the function, before declaring that we are done. Otherwise, we are done, either with the value of the constant, or with the value `nan` for unknown words.
- If the next token is anything else, we check whether it is a known prefix operator, in which case `_fp_parse_operand:Nw` finds its operand. If it is not known, then

either a number is missing (if the token is a known infix operator) or the token is simply invalid in floating point expressions.

Once a number is found, `_fp_parse_one:Nw` also finds an infix operator. This goes as follows.

- If the next token is a control sequence, it could be the special marker `\s_fp_mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_three`, with an implied multiplication.
- If the next token is a letter, it is also a case of juxtaposition, as letters cannot be proper infix operators.
- Otherwise (including in the case of digits), if the token is a known infix operator, the appropriate `_fp_infix_⟨operator⟩:N` function is built, and if it does not exist, we complain. In particular, the juxtaposition `\c_three 2` is disallowed.

In the above, we need to test whether a character token `#1` is a digit:

```
\if_int_compare:w \c\_nine < 1 \token_to_str:N #1 \exp_stop_f:
  is a digit
\else:
  not a digit
\fi:
```

To exclude 0, replace `\c_nine` by `\c_ten`. The use of `\token_to_str:N` ensures that a digit with any catcode is detected. To test if a character token is a letter, we need to work with its character code, testing if `'#1` lies in `[65,90]` (uppercase letters) or `[97,112]` (lowercase letters)

```
\if_int_compare:w \_int_eval:w
  ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26 = \c\_three
  is a letter
\else:
  not a letter
\fi:
```

At all steps, we try to accept all category codes: when `#1` is kept to be used later, it is almost always converted to category code other through `\token_to_str:N`. More precisely, catcodes `{3,6,7,8,11,12}` should work without trouble, but `{1,2,4,10,13}` will not work, and of course `{0,5,9}` cannot become tokens.

Floating point expressions should behave as much as possible like ε -TeX-based integer expressions and dimension expressions. In particular, **f**-expansion should be performed as the expression is read, token by token, forcing the expansion of protected macros, and ignoring spaces. One advantage of expanding at every step is that restricted expandable functions can then be used in floating point expressions just as they can be in other kinds of expressions. Problematically, spaces stop **f**-expansion: for instance, the macro `\X` below will not be expanded if we simply perform **f**-expansion.

```

\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }

```

Of course, spaces will not appear in a code setting, but may very easily come in document-level input, from which some expressions may come. To avoid this problem, at every step, we do essentially what `\use:f` would do: take an argument, put it back in the input stream, then `f`-expand it. This is not a complete solution, since a macro's expansion could contain leading spaces which will stop the `f`-expansion before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers will correctly be expanded to the underlying `\s__fp ...` structure. The `f`-expansion is performed by `__fp_parse_expand:w`.

27.2 Main auxiliary functions

`__fp_parse_operand:Nw` Reads the "...", performing every computation with a precedence higher than $\langle precedence \rangle$, then expands to where the $\langle operation \rangle$ is the first operation with a lower precedence, possibly `end`, and the "..." start just after the $\langle operation \rangle$.
(End definition for `__fp_parse_operand:Nw`.)

`__fp_parse_infix_+:N` If `+` has a precedence higher than the $\langle precedence \rangle$, cleans up a second $\langle operand \rangle$ and finds the $\langle operation_2 \rangle$ which follows, and expands to Otherwise expands to A similar function exists for each infix operator.
(End definition for `__fp_parse_infix_+:N`.)

`__fp_parse_one:Nw` Cleans up one or two operands depending on how the precedence of the next operation compares to the $\langle precedence \rangle$. If the following $\langle operation \rangle$ has a precedence higher than $\langle precedence \rangle$, expands to and otherwise expands to
(End definition for `__fp_parse_one:Nw`.)

27.3 Helpers

`__fp_parse_expand:w` This function must always come within a `\romannumeral` 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.
10384 `\cs_new:Npn __fp_parse_expand:w #1 { -'0 #1 }`
(End definition for `__fp_parse_expand:w`.)

`__fp_parse_return_semicolon:w` This very odd function swaps its position with the following `\fi:` and removes `__fp_parse_expand:w` normally responsible for expansion. That turns out to be useful.
10385 `\cs_new:Npn __fp_parse_return_semicolon:w`
10386 `#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 `_?`.

```

10387 \group_begin:
10388 \char_set_catcode_other:N \S
10389 \char_set_catcode_other:N \F
10390 \char_set_catcode_other:N \P
10391 \char_set_lccode:nm { '\- } { '\_ }
10392 \tl_to_lowercase:n
10393 {
10394   \group_end:
10395   \cs_new:Npn \__fp_type_from_scan:N #1
10396   {
10397     \exp_after:wN \__fp_type_from_scan:w
10398     \token_to_str:N #1 \q_mark S--FP-? \q_mark \q_stop
10399   }
10400   \cs_new:Npn \__fp_type_from_scan:w #1 S--FP #2 \q_mark #3 \q_stop {#2}
10401 }

```

(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 meeting a non-digit, or up to a number of digits equal to their index. The full expansion is `__fp_parse_digits_v:N` `__fp_parse_digits_iv:N` `__fp_parse_digits_iii:N` `__fp_parse_digits_ii:N` `__fp_parse_digits_i:N`

$\langle digits \rangle ; \langle filling\ 0 \rangle ; \langle length \rangle$

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.

```

10402 \cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
10403 {
10404   \cs_new:cpn { __fp_parse_digits_ #1 :N } ##1
10405   {
10406     \if_int_compare:w \c_nine < 1 \token_to_str:N ##1 \exp_stop_f:
10407     \token_to_str:N ##1 \exp_after:wN #2 \tex_romannumeral:D
10408   } else:
10409     \__fp_parse_return_semicolon:w #3 ##1
10410   \fi:
10411   \__fp_parse_expand:w
10412 }
10413 }
10414 \__fp_tmp:w {vii} \__fp_parse_digits_vi:N { 0000000 ; 7 }
10415 \__fp_tmp:w {vi} \__fp_parse_digits_v:N { 000000 ; 6 }
10416 \__fp_tmp:w {v} \__fp_parse_digits_iv:N { 00000 ; 5 }
10417 \__fp_tmp:w {iv} \__fp_parse_digits_iii:N { 0000 ; 4 }
10418 \__fp_tmp:w {iii} \__fp_parse_digits_ii:N { 000 ; 3 }

```

```

10419 \__fp_tmp:w {ii} \__fp_parse_digits_i:N { 00 ; 2 }
10420 \__fp_tmp:w {i} \__fp_parse_digits_:N { 0 ; 1 }
10421 \cs_new_nopar:Npn \__fp_parse_digits_:N { ; ; 0 }
(End definition for \__fp_parse_digits_vii:N and others.)

```

27.4 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 $\text{\LaTeX} 2_{\epsilon}$ command `\protect`. Testing if #2 is a control sequence thus includes `\exp_not:N`.

```

10422 \cs_new:Npn \__fp_parse_one:Nw #1 #2
10423 {
10424   \if_catcode:w \scan_stop: \exp_not:N #2
10425     \if_meaning:w \scan_stop: #2
10426       \exp_after:wN \exp_after:wN
10427       \exp_after:wN \__fp_parse_one_fp:NN
10428     \else:
10429       \exp_after:wN \exp_after:wN
10430       \exp_after:wN \__fp_parse_one_register:NN
10431     \fi:
10432   \else:
10433     \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
10434       \exp_after:wN \exp_after:wN
10435       \exp_after:wN \__fp_parse_one_digit:NN
10436     \else:
10437       \exp_after:wN \exp_after:wN
10438       \exp_after:wN \__fp_parse_one_other:NN
10439     \fi:
10440   \fi:
10441   #1 #2
10442 }
(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.

```

10443 \cs_new:Npn \__fp_parse_one_fp:NN #1#2
10444 {
10445   \cs:w __fp_exp_after \__fp_type_from_scan:N #2 _f:nw \cs_end:
10446   {
10447     \exp_after:wN \__fp_parse_infix:NN
10448     \exp_after:wN #1 \tex_romannumeral:D \__fp_parse_expand:w
10449   }
10450   #2
10451 }
10452 \cs_new:Npn \__fp_exp_after_mark_f:nw #1
10453 {
10454   \__msg_kernel_expandable_error:nn { kernel } { fp-early-end }
10455   \exp_after:wN \c_nan_fp \tex_romannumeral:D -'0 #1
10456 }
10457 \cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
10458 {
10459   \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
10460   \exp_after:wN \c_nan_fp \tex_romannumeral:D -'0 #1
10461 }
(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.

```

10462 \cs_new:Npn \__fp_parse_one_register:NN #1#2
10463 {
10464   \exp_after:wN \__fp_parse_infix_after_operand:NwN
10465   \exp_after:wN #1
10466   \tex_romannumeral:D -'0
10467   \exp_after:wN \__fp_parse_one_register_aux:Nw
10468   \exp_after:wN #2
10469   \__int_value:w
10470   \exp_after:wN \__fp_parse_exponent:N
10471   \tex_romannumeral:D \__fp_parse_expand:w
10472 }
10473 \group_begin:

```

```

10474 \char_set_catcode_other:N \P
10475 \char_set_catcode_other:N \T
10476 \char_set_catcode_other:N \M
10477 \char_set_catcode_other:N \U
10478 \tl_to_lowercase:n
10479 {
10480   \group_end:
10481   \cs_new:Npn \__fp_parse_one_register_aux:Nw #1
10482   {
10483     \exp_after:wN \use:nn
10484     \exp_after:wN \__fp_parse_one_register_auxii:wwwNw
10485     \exp_after:wN { \tex_the:D \exp_not:N #1 }
10486     ; \__fp_parse_one_register_dim:ww
10487     PT ; \__fp_parse_one_register_mu:www
10488     . PT ; \__fp_parse_one_register_int:www
10489     \q_stop
10490   }
10491   \cs_new:Npn \__fp_parse_one_register_auxii:wwwNw
10492   #1 . #2 PT #3 ; #4#5 \q_stop { #4 #1.#2; }
10493   \cs_new:Npn \__fp_parse_one_register_mu:www #1 MU; #2;
10494   { \__fp_parse_one_register_dim:ww #1; }
10495 }
10496 \cs_new:Npn \__fp_parse_one_register_int:www #1; #2.; #3;
10497 { \__fp_parse:n { #1 e #3 } }
10498 \cs_new:Npn \__fp_parse_one_register_dim:ww #1; #2;
10499 {
10500   \exp_after:wN \__fp_from_dim_test:ww
10501   \__int_value:w #2 \exp_after:wN ,
10502   \__int_value:w \__dim_eval:w #1 pt ;
10503 }

```

(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.

```

10504 \cs_new:Npn \__fp_parse_one_digit:NN #1
10505 {
10506   \exp_after:wN \__fp_parse_infix_after_operand:NwN
10507   \exp_after:wN #1
10508   \tex_romannumeral:D -‘0
10509   \exp_after:wN \__fp_sanitize:wN
10510   \int_use:N \__int_eval:w \c_zero \__fp_parse_trim_zeros:N
10511 }

```

(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`.

```

10512 \cs_new:Npn \__fp_parse_one_other:NN #1 #2
10513 {
10514   \if_int_compare:w
10515     \__int_eval:w
10516     ( '#2 \if_int_compare:w '#2 > 'Z - \c_thirty_two \fi: ) / 26
10517     = \c_three
10518     \exp_after:wN \__fp_parse_word:Nw
10519     \exp_after:wN #1
10520     \exp_after:wN #2
10521     \tex_romannumeral:D \exp_after:wN \__fp_parse_letters:N
10522     \tex_romannumeral:D
10523   \else:
10524     \exp_after:wN \__fp_parse_prefix:NNN
10525     \exp_after:wN #1
10526     \exp_after:wN #2
10527     \cs:w \__fp_parse_prefix_#2:Nw \exp_after:wN \cs_end:
10528     \tex_romannumeral:D
10529   \fi:
10530   \__fp_parse_expand:w
10531 }

```

(End definition for `__fp_parse_one_other:NN`.)

`__fp_parse_word:Nw`
`__fp_parse_letters:N`

Finding letters is a simple recursion. Once `__fp_parse_letters:N` has done its job, we try to build a control sequence from the word #2. If it is a known word, then the corresponding action is taken, and otherwise, we complain about an unknown word, yield `\c_nan_fp`, and look for the following infix operator. Note that the unknown word could be a mistyped function as well as a mistyped constant, so there is no way to tell whether to look for arguments; we do not.

```

10532 \cs_new:Npn \__fp_parse_word:Nw #1#2;
10533 {
10534   \cs_if_exist_use:cF { \__fp_parse_word_#2:N }
10535   {
10536     \__msg_kernel_expandable_error:nnn
10537     { kernel } { unknown-fp-word } {#2}
10538     \exp_after:wN \c_nan_fp \tex_romannumeral:D -'0
10539     \__fp_parse_infix:NN
10540   }
10541   #1
10542 }
10543 \cs_new:Npn \__fp_parse_letters:N #1
10544 {
10545   -'0
10546   \if_int_compare:w
10547     \if_catcode:w \scan_stop: \exp_not:N #1
10548     \c_zero
10549   \else:
10550     \__int_eval:w

```

```

10551      ( '#1 \if_int_compare:w '#1 > 'Z - \c_thirty_two \fi: )
10552      / 26
10553      \fi:
10554      = \c_three
10555      \exp_after:wN #1
10556      \tex_romannumeral:D \exp_after:wN \__fp_parse_letters:N
10557      \tex_romannumeral:D
10558      \else:
10559      \__fp_parse_return_semicolon:w #1
10560      \fi:
10561      \__fp_parse_expand:w
10562      }

```

(End definition for __fp_parse_word:Nw.)

__fp_parse_prefix:NNN
 __fp_parse_prefix_unknown:NNN

For this function, #1 is the previous *<precedence>*, #2 is the operator just seen, and #3 is a control sequence which implements the operator if it is a known operator. If this control sequence is \scan_stop:, then the operator is in fact unknown. Either the expression is missing a number there (if the operator is valid as an infix operator), and we put nan, wrapping the infix operator in a csname as appropriate, or the character is simply invalid in floating point expressions, and we continue looking for a number, starting again from __fp_parse_one:Nw.

```

10563 \cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
10564 {
10565   \if_meaning:w \scan_stop: #3
10566   \exp_after:wN \__fp_parse_prefix_unknown:NNN
10567   \exp_after:wN #2
10568   \fi:
10569   #3 #1
10570 }
10571 \cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
10572 {
10573   \cs_if_exist:cTF { \__fp_parse_infix_#1:N }
10574   {
10575     \__msg_kernel_expandable_error:nnn
10576     { kernel } { fp-missing-number } {#1}
10577     \exp_after:wN \c_nan_fp \tex_romannumeral:D -'0
10578     \__fp_parse_infix:NN #3 #1
10579   }
10580   {
10581     \__msg_kernel_expandable_error:nnn
10582     { kernel } { fp-unknown-symbol } {#1}
10583     \__fp_parse_one:Nw #3
10584   }
10585 }

```

(End definition for __fp_parse_prefix:NNN and __fp_parse_prefix_unknown:NNN.)

27.4.1 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` 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.

```

10586 \cs_new:Npn \__fp_parse_trim_zeros:N #1
10587 {
10588   \if:w 0 \exp_not:N #1
10589     \exp_after:wN \__fp_parse_trim_zeros:N
10590     \tex_romannumeral:D
10591   \else:
10592     \if:w . \exp_not:N #1
10593       \exp_after:wN \__fp_parse_strim_zeros:N
10594       \tex_romannumeral:D
10595     \else:
10596       \__fp_parse_trim_end:w #1
10597     \fi:
10598   \fi:
10599   \__fp_parse_expand:w
10600 }
10601 \cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
10602 {
10603   \fi:
10604   \fi:
10605   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10606     \exp_after:wN \__fp_parse_large:N
10607   \else:
10608     \exp_after:wN \__fp_parse_zero:
10609   \fi:
10610   #1
10611 }
```

(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”.

```

10612 \cs_new:Npn \__fp_parse_strim_zeros:N #1
```

```

10613 {
10614   \if:w 0 \exp_not:N #1
10615     - \c_one
10616     \exp_after:wN \__fp_parse_strim_zeros:N \tex_romannumeral:D
10617   \else:
10618     \__fp_parse_strim_end:w #1
10619   \fi:
10620   \__fp_parse_expand:w
10621 }
10622 \cs_new:Npn \__fp_parse_strim_end:w #1 \fi: \__fp_parse_expand:w
10623 {
10624   \fi:
10625   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10626     \exp_after:wN \__fp_parse_small:N
10627   \else:
10628     \exp_after:wN \__fp_parse_zero:
10629   \fi:
10630   #1
10631 }

```

(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).

```

10632 \cs_new:Npn \__fp_parse_zero:
10633 {
10634   \exp_after:wN ; \exp_after:wN 1
10635   \__int_value:w \__fp_parse_exponent:N
10636 }

```

(End definition for __fp_parse_zero:.)

27.4.2 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.

```

10637 \cs_new:Npn \__fp_parse_small:N #1
10638 {
10639   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
10640   \int_use:N \__int_eval:w 1 \token_to_str:N #1
10641   \exp_after:wN \__fp_parse_small_leading:wwNN
10642   \__int_value:w 1
10643   \exp_after:wN \__fp_parse_digits_vii:N

```

```

10644         \tex_romannumeral:D \__fp_parse_expand:w
10645     }

```

(End definition for __fp_parse_small:N.)

__fp_parse_small_leading:wwNN

We leave $\langle \textit{digits} \rangle$ $\langle \textit{zeros} \rangle$ 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.

```

10646 \cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
10647 {
10648     #1 #2
10649     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
10650     \exp_after:wN \c_zero
10651     \int_use:N \__int_eval:w 1
10652     \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
10653         \token_to_str:N #4
10654         \exp_after:wN \__fp_parse_small_trailing:wwNN
10655         \__int_value:w 1
10656         \exp_after:wN \__fp_parse_digits_vi:N
10657         \tex_romannumeral:D
10658     \else:
10659         0000 0000 \__fp_parse_exponent:Nw #4
10660     \fi:
10661     \__fp_parse_expand:w
10662 }

```

(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 $\langle \textit{next token} \rangle$ 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.

```

10663 \cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
10664 {
10665     #1 #2
10666     \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
10667         \token_to_str:N #4
10668         \exp_after:wN \__fp_parse_small_round:NN
10669         \exp_after:wN #4
10670         \tex_romannumeral:D
10671     \else:
10672         0 \__fp_parse_exponent:Nw #4
10673     \fi:
10674     \__fp_parse_expand:w
10675 }

```

(End definition for __fp_parse_small_trailing:wwNN.)

```

\__fp_parse_pack_trailing:NNNNNww
\__fp_parse_pack_leading:NNNNNww
\__fp_parse_pack_carry:w

```

Those functions are expanded after all the digits are found, we took care of the rounding, as well as the exponent. The last argument is the exponent. The previous five arguments are 8 digits which we pack in groups of 4, and the argument before that is 1, except in the rare case where rounding lead to a carry, in which case the argument is 2. The **trailing** function has an exponent shift as its first argument, which we add to the exponent found in the **e...** syntax. If the trailing digits cause a carry, the integer expression for the leading digits is incremented (+ \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.

```

10676 \cs_new:Npn \__fp_parse_pack_trailing:NNNNNww #1 #2 #3#4#5#6 #7; #8 ;
10677 {
10678   \if_meaning:w 2 #2 + \c_one \fi:
10679   ; #8 + #1 ; {#3#4#5#6} {#7};
10680 }
10681 \cs_new:Npn \__fp_parse_pack_leading:NNNNNww #1 #2#3#4#5 #6; #7;
10682 {
10683   + #7
10684   \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
10685   ; 0 {#2#3#4#5} {#6}
10686 }
10687 \cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
10688 { \fi: + \c_one ; 0 {1000} }
(End definition for \__fp_parse_pack_trailing:NNNNNww, \__fp_parse_pack_leading:NNNNNww, and
\__fp_parse_pack_carry:w.)

```

27.4.3 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.

```

10689 \cs_new:Npn \__fp_parse_large:N #1
10690 {
10691   \exp_after:wN \__fp_parse_large_leading:wwNN
10692   \__int_value:w 1 \token_to_str:N #1
10693   \exp_after:wN \__fp_parse_digits_vii:N
10694   \tex_romannumeral:D \__fp_parse_expand:w
10695 }
(End definition for \__fp_parse_large:N.)

```

```

\__fp_parse_large_leading:wwNN

```

We shift the exponent by the number of digits in #1, namely the target number, 8, minus the $\langle \text{number of zeros} \rangle$ (number of digits missing). Then prepare to pack the 8 first digits. If the $\langle \text{next token} \rangle$ is a digit, read up to 6 more digits (digits 10 to 15). If it is a period, try

to grab the end of our 8 first digits, branching to the `small` functions since the number of digit does not affect the exponent anymore. Finally, if this is the end of the significand, insert the `<zeros>` to complete the 8 first digits, insert 8 more, and look for an exponent.

```

10696 \cs_new:Npn \__fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
10697 {
10698   + \c_eight - #3
10699   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
10700   \int_use:N \__int_eval:w 1 #1
10701   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
10702     \exp_after:wN \__fp_parse_large_trailing:wwNN
10703     \__int_value:w 1 \token_to_str:N #4
10704     \exp_after:wN \__fp_parse_digits_vi:N
10705     \tex_romannumeral:D
10706   \else:
10707     \if:w . \exp_not:N #4
10708       \exp_after:wN \__fp_parse_small_leading:wwNN
10709       \__int_value:w 1
10710       \cs:w
10711         \__fp_parse_digits_
10712         \tex_romannumeral:D #3
10713         :N \exp_after:wN
10714         \cs_end:
10715         \tex_romannumeral:D
10716     \else:
10717       #2
10718       \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
10719       \exp_after:wN \c_zero
10720       \__int_value:w 1 0000 0000
10721       \__fp_parse_exponent:Nw #4
10722     \fi:
10723   \fi:
10724   \__fp_parse_expand:w
10725 }

```

(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.

```

10726 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
10727 {
10728   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
10729     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww

```

```

10730 \exp_after:wN \c_eight
10731 \int_use:N \__int_eval:w 1 #1 \token_to_str:N #4
10732 \exp_after:wN \__fp_parse_large_round:NN
10733 \exp_after:wN #4
10734 \tex_romannumeral:D
10735 \else:
10736 \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
10737 \int_use:N \__int_eval:w \c_seven - #3 \exp_stop_f:
10738 \int_use:N \__int_eval:w 1 #1
10739 \if:w . \exp_not:N #4
10740 \exp_after:wN \__fp_parse_small_trailing:wwNN
10741 \__int_value:w 1
10742 \cs:w
10743 \__fp_parse_digits_
10744 \tex_romannumeral:D #3
10745 :N \exp_after:wN
10746 \cs_end:
10747 \tex_romannumeral:D
10748 \else:
10749 #2 0 \__fp_parse_exponent:Nw #4
10750 \fi:
10751 \fi:
10752 \__fp_parse_expand:w
10753 }

```

(End definition for __fp_parse_large_trailing:wwNN.)

27.4.4 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.

```

10754 \cs_new:Npn \__fp_parse_round_loop:N #1
10755 {
10756 \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10757 + \c_one
10758 \if:w 0 \token_to_str:N #1
10759 \exp_after:wN \__fp_parse_round_loop:N
10760 \tex_romannumeral:D
10761 \else:
10762 \exp_after:wN \__fp_parse_round_up:N
10763 \tex_romannumeral:D
10764 \fi:
10765 \else:
10766 \__fp_parse_return_semicolon:w \c_zero #1
10767 \fi:
10768 \__fp_parse_expand:w

```

```

10769 }
10770 \cs_new:Npn \__fp_parse_round_up:N #1
10771 {
10772   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10773     + \c_one
10774     \exp_after:wN \__fp_parse_round_up:N
10775     \tex_romannumeral:D
10776   \else:
10777     \__fp_parse_return_semicolon:w \c_one #1
10778   \fi:
10779   \__fp_parse_expand:w
10780 }

```

(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.

```

10781 \cs_new:Npn \__fp_parse_round_after:wN #1; #2
10782 {
10783   + #2 \exp_after:wN ;
10784   \int_use:N \__int_eval:w #1 + \__fp_parse_exponent:N
10785 }

```

(End definition for __fp_parse_round_after:wN.)

__fp_parse_small_round:NN Here, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If #2 is not a digit, then fetch an exponent and expand to ;*<exponent>* only. Otherwise, we will expand to +\c_zero or +\c_one, then ;*<exponent>*. To decide which, call __fp_round_s:NNNw to know whether to round up, giving it as arguments a sign 0 (all explicit numbers are positive), the digit #1 to round, the first following digit #2, and either +\c_zero or +\c_one depending on whether the following digits are all zero or not. This last argument is obtained by __fp_parse_round_loop:N, whose number of digits we discard by multiplying it by 0. The exponent which follows the number is also fetched by __fp_parse_round_after:wN.

```

10786 \cs_new:Npn \__fp_parse_small_round:NN #1#2
10787 {
10788   \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
10789     +
10790     \exp_after:wN \__fp_round_s:NNNw
10791     \exp_after:wN 0
10792     \exp_after:wN #1
10793     \exp_after:wN #2
10794     \int_use:N \__int_eval:w
10795     \exp_after:wN \__fp_parse_round_after:wN
10796     \int_use:N \__int_eval:w \c_zero * \__int_eval:w \c_zero
10797     \exp_after:wN \__fp_parse_round_loop:N
10798     \tex_romannumeral:D
10799   \else:

```

```

10800     \__fp_parse_exponent:Nw #2
10801     \fi:
10802     \__fp_parse_expand:w
10803 }
(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.

```

10804 \cs_new:Npn \__fp_parse_large_round:NN #1#2
10805 {
10806   \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
10807   +
10808   \exp_after:wN \__fp_round_s:NNNw
10809   \exp_after:wN 0
10810   \exp_after:wN #1
10811   \exp_after:wN #2
10812   \int_use:N \__int_eval:w
10813   \exp_after:wN \__fp_parse_large_round_aux:wNN
10814   \int_use:N \__int_eval:w \c_one
10815   \exp_after:wN \__fp_parse_round_loop:N
10816   \else: %^^A could be dot, or e, or other
10817   \exp_after:wN \__fp_parse_large_round_test:NN
10818   \exp_after:wN #1
10819   \exp_after:wN #2
10820   \fi:
10821 }
10822 \cs_new:Npn \__fp_parse_large_round_test:NN #1#2
10823 {
10824   \if:w . \exp_not:N #2
10825   \exp_after:wN \__fp_parse_small_round:NN
10826   \exp_after:wN #1
10827   \tex_romannumeral:D
10828   \else:
10829   \__fp_parse_exponent:Nw #2
10830   \fi:
10831   \__fp_parse_expand:w
10832 }
10833 \cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
10834 {
10835   + #2
10836   \exp_after:wN \__fp_parse_round_after:wN
10837   \int_use:N \__int_eval:w #1

```



```

10838 \if:w . \exp_not:N #3
10839 + \c_zero * \__int_eval:w \c_zero
10840 \exp_after:wN \__fp_parse_round_loop:N
10841 \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
10842 \else:
10843 \exp_after:wN ;
10844 \exp_after:wN \c_zero
10845 \exp_after:wN #3
10846 \fi:
10847 }
(End definition for \__fp_parse_large_round:NN, \__fp_parse_large_round_test:NN, and \__fp_parse_large_round_aux:w)

```

27.4.5 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.

```

10848 \cs_new:Npn \__fp_parse_exponent:Nw #1 #2 \__fp_parse_expand:w
10849 {
10850 \exp_after:wN ;
10851 \__int_value:w #2 \__fp_parse_exponent:N #1
10852 }
(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.

```

10853 \cs_new:Npn \__fp_parse_exponent:N #1
10854 {
10855   \if:w e \exp_not:N #1
10856     \exp_after:wN \__fp_parse_exponent_aux:N
10857     \tex_romannumeral:D
10858   \else:
10859     0 \__fp_parse_return_semicolon:w #1
10860   \fi:
10861   \__fp_parse_expand:w
10862 }
10863 \cs_new:Npn \__fp_parse_exponent_aux:N #1
10864 {
10865   \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #1
10866     \c_zero \else: ' #1 \fi: > '9 \exp_stop_f:
10867   0 \exp_after:wN ; \exp_after:wN e
10868   \else:
10869     \exp_after:wN \__fp_parse_exponent_sign:N
10870   \fi:
10871   #1
10872 }

```

(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).

```

10873 \cs_new:Npn \__fp_parse_exponent_sign:N #1
10874 {
10875   \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
10876   \exp_after:wN \__fp_parse_exponent_sign:N
10877   \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
10878   \else:
10879     \exp_after:wN \__fp_parse_exponent_body:N
10880     \exp_after:wN #1
10881   \fi:
10882 }

```

(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).

```

10883 \cs_new:Npn \__fp_parse_exponent_body:N #1
10884 {
10885   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10886   \token_to_str:N #1
10887   \exp_after:wN \__fp_parse_exponent_digits:N
10888   \tex_romannumeral:D

```

```

10889 \else:
10890 \__fp_parse_exponent_keep:NTF #1
10891 { \__fp_parse_return_semicolon:w #1 }
10892 {
10893 \exp_after:wN ;
10894 \tex_romannumeral:D
10895 }
10896 \fi:
10897 \__fp_parse_expand:w
10898 }

```

(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 T_EX error. It is mostly harmless, except when parsing 0e9876543210, which should be a valid representation of 0, but is not.

```

10899 \cs_new:Npn \__fp_parse_exponent_digits:N #1
10900 {
10901 \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10902 \token_to_str:N #1
10903 \exp_after:wN \__fp_parse_exponent_digits:N
10904 \tex_romannumeral:D
10905 \else:
10906 \__fp_parse_return_semicolon:w #1
10907 \fi:
10908 \__fp_parse_expand:w
10909 }

```

(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.

```

10910 \prg_new_conditional:Npnn \__fp_parse_exponent_keep:N #1 { TF }
10911 {
10912 \if_catcode:w \scan_stop: \exp_not:N #1
10913 \if_meaning:w \scan_stop: #1
10914 \if_int_compare:w
10915 \__str_if_eq_x:nn { \s__fp } { \exp_not:N #1 } = \c_zero
10916 0
10917 \__msg_kernel_expandable_error:nnn
10918 { kernel } { fp-after-e } { floating~point~ }
10919 \prg_return_true:

```

```

10920         \else:
10921             0
10922             \_msg_kernel_expandable_error:nnn
10923             { kernel } { bad-variable } {#1}
10924             \prg_return_false:
10925         \fi:
10926     \else:
10927         \if_int_compare:w
10928             \_str_if_eq_x:nn { \_int_value:w #1 } { \tex_the:D #1 }
10929             = \c_zero
10930             \_int_value:w #1
10931         \else:
10932             0
10933             \_msg_kernel_expandable_error:nnn
10934             { kernel } { fp-after-e } { dimension~#1 }
10935         \fi:
10936         \prg_return_false:
10937     \fi:
10938 \else:
10939     0
10940     \_msg_kernel_expandable_error:nnn
10941     { kernel } { fp-missing } { exponent }
10942     \prg_return_true:
10943 \fi:
10944 }

```

(End definition for _fp_parse_exponent_keep:NTF.)

27.5 Constants, functions and prefix operators

27.5.1 Prefix operators

_fp_parse_prefix+:Nw A unary + does nothing: we should continue looking for a number.

```

10945 \cs_new_eq:cN { \_fp_parse_prefix+:Nw } \_fp_parse_one:Nw

```

(End definition for _fp_parse_prefix+:Nw.)

_fp_parse_apply_unary:NNwN 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.

```

10946 \cs_new:Npn \_fp_parse_apply_unary:NNwN #1#2#3#4@#5
10947 {
10948     #3 #2 #4 @
10949     \tex_romannumeral:D -'0 #5 #1
10950 }

```

(End definition for _fp_parse_apply_unary:NNwN.)

_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

_fp_parse_prefix!:Nw

unary operator, then call the appropriate `__fp_⟨operation⟩_o:w` function, where the `⟨operation⟩` is `set_sign` or `not`.

```

10951 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
10952 {
10953   \cs_new:cpn { __fp_parse_prefix_ #1 :Nw } ##1
10954   {
10955     \exp_after:wN \__fp_parse_apply_unary:NNNwN
10956     \exp_after:wN ##1
10957     \exp_after:wN #4
10958     \exp_after:wN #3
10959     \tex_romannumeral:D
10960     \if_int_compare:w #2 < ##1
10961       \__fp_parse_operand:Nw ##1
10962     \else:
10963       \__fp_parse_operand:Nw #2
10964     \fi:
10965     \__fp_parse_expand:w
10966   }
10967 }
10968 \__fp_tmp:w - \c_twelve \__fp_set_sign_o:w 2
10969 \__fp_tmp:w ! \c_twelve \__fp_not_o:w ?
(End definition for \__fp_parse_prefix_-:Nw and \__fp_parse_prefix_!:Nw.)

```

`__fp_parse_prefix_:Nw` Numbers which start with a decimal separator (a period) end up here. Of course, we do not look for an operand, but for the rest of the number. This function is very similar to `__fp_parse_one_digit:NN` but calls `__fp_parse_strim_zeros:N` to trim zeros after the decimal point, rather than the `trim_zeros` function for zeros before the decimal point.

```

10970 \cs_new:cpn { __fp_parse_prefix_:Nw } #1
10971 {
10972   \exp_after:wN \__fp_parse_infix_after_operand:NwN
10973   \exp_after:wN #1
10974   \tex_romannumeral:D -‘0
10975   \exp_after:wN \__fp_sanitize:wN
10976   \int_use:N \__int_eval:w \c_zero \__fp_parse_strim_zeros:N
10977 }
(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.

```

10978 \group_begin:
10979   \char_set_catcode_letter:N (
10980   \char_set_catcode_letter:N )

```

```

10981 \cs_new:Npn \__fp_parse_prefix_(:Nw #1
10982 {
10983   \exp_after:wN \__fp_parse_lparen_after:NwN
10984   \exp_after:wN #1
10985   \tex_romannumeral:D
10986   \if_int_compare:w #1 = \c_sixteen
10987     \__fp_parse_operand:Nw \c_one
10988   \else:
10989     \__fp_parse_operand:Nw \c_zero
10990   \fi:
10991   \__fp_parse_expand:w
10992 }
10993 \cs_new:Npn \__fp_parse_lparen_after:NwN #1#2 @ #3
10994 {
10995   \token_if_eq_meaning:NNTF #3 \__fp_parse_infix_):N
10996   {
10997     \__fp_exp_after_array_f:w #2 \s__fp_stop
10998     \exp_after:wN \__fp_parse_infix:NN
10999     \exp_after:wN #1
11000     \tex_romannumeral:D \__fp_parse_expand:w
11001   }
11002   {
11003     \__msg_kernel_expandable_error:nnn
11004     { kernel } { fp-missing } { } }
11005     #2 @ \use_none:n #3
11006   }
11007 }
11008 \group_end:

```

(End definition for __fp_parse_prefix_(:Nw and __fp_parse_lparen_after:NwN.)

27.5.2 Constants

__fp_parse_word_inf: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.

```

\__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
11009 \cs_set_protected:Npn \__fp_tmp:w #1 #2
11010 {
11011   \cs_new_nopar:cpn { __fp_parse_word_#1:N }
11012   { \exp_after:wN #2 \tex_romannumeral:D -'0 \__fp_parse_infix:NN }
11013 }
11014 \__fp_tmp:w { inf } \c_inf_fp
11015 \__fp_tmp:w { nan } \c_nan_fp
11016 \__fp_tmp:w { pi } \c_pi_fp
11017 \__fp_tmp:w { deg } \c_one_degree_fp
11018 \__fp_tmp:w { true } \c_one_fp
11019 \__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

```

```

11020 \cs_set_protected:Npn \__fp_tmp:w #1 #2
11021 {
11022   \cs_new_nopar:cpn { __fp_parse_word_#1:N }
11023   {
11024     \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
11025     \s__fp \__fp_chk:w 10 #2 ;
11026   }
11027 }
11028 \__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
11029 \__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
11030 \__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
11031 \__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
11032 \__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
11033 \__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
11034 \__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
11035 \__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
11036 \__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
11037 \__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
11038 \__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`.

```

11039 \tl_map_inline:nn { {em} {ex} }
11040 {
11041   \cs_new_nopar:cpn { __fp_parse_word_#1:N }
11042   {
11043     \exp_after:wN \__fp_from_dim_test:ww
11044     \exp_after:wN 0 \exp_after:wN ,
11045     \__int_value:w \__dim_eval:w 1 #1 \exp_after:wN ;
11046     \tex_romannumeral:D -'0 \__fp_parse_infix:NN
11047   }
11048 }
(End definition for \__fp_parse_word_em:N and \__fp_parse_word_ex:N.)

```

27.5.3 Functions

```

\__fp_parse_unary_function:nNN
\__fp_parse_function:NNN
11049 \cs_new:Npn \__fp_parse_unary_function:nNN #1#2#3
11050 {
11051   \exp_after:wN \__fp_parse_apply_unary:NNNwN
11052   \exp_after:wN #3
11053   \exp_after:wN #2
11054   \cs:w __fp_#1_o:w \exp_after:wN \cs_end:
11055   \tex_romannumeral:D
11056   \__fp_parse_operand:Nw \c_fifteen \__fp_parse_expand:w
11057 }
11058 \cs_new:Npn \__fp_parse_function:NNN #1#2#3
11059 {

```

```

11060 \exp_after:wN \__fp_parse_apply_unary:NNWwN
11061 \exp_after:wN #3
11062 \exp_after:wN #2
11063 \exp_after:wN #1
11064 \tex_romannumeral:D
11065 \__fp_parse_operand:Nw \c_sixteen \__fp_parse_expand:w
11066 }

```

(End definition for __fp_parse_unary_function:nNN and __fp_parse_function:NNN.)

```

\__fp_parse_word_acot:N
\__fp_parse_word_acotd:N
\__fp_parse_word_atan:N
\__fp_parse_word_atand:N
\__fp_parse_word_max:N
\__fp_parse_word_min:N

```

Those functions are also unary (not binary), but may receive a variable number of arguments.

```

11067 \cs_new_nopar:Npn \__fp_parse_word_acot:N
11068 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_i:nn }
11069 \cs_new_nopar:Npn \__fp_parse_word_acotd:N
11070 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_ii:nn }
11071 \cs_new_nopar:Npn \__fp_parse_word_atan:N
11072 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_i:nn }
11073 \cs_new_nopar:Npn \__fp_parse_word_atand:N
11074 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_ii:nn }
11075 \cs_new_nopar:Npn \__fp_parse_word_max:N
11076 { \__fp_parse_function:NNN \__fp_minmax_o:Nw 2 }
11077 \cs_new_nopar:Npn \__fp_parse_word_min:N
11078 { \__fp_parse_function:NNN \__fp_minmax_o:Nw 0 }

```

(End definition for __fp_parse_word_acot:N and others.)

```

\__fp_parse_word_abs:N
\__fp_parse_word_exp:N
\__fp_parse_word_ln:N
\__fp_parse_word_sqrt:N

```

Unary functions.

```

11079 \cs_new:Npn \__fp_parse_word_abs:N
11080 { \__fp_parse_unary_function:nNN { set_sign } 0 }
11081 \cs_new_nopar:Npn \__fp_parse_word_exp:N
11082 { \__fp_parse_unary_function:nNN {exp} ? }
11083 \cs_new_nopar:Npn \__fp_parse_word_ln:N
11084 { \__fp_parse_unary_function:nNN {ln} ? }
11085 \cs_new_nopar:Npn \__fp_parse_word_sqrt:N
11086 { \__fp_parse_unary_function:nNN {sqrt} ? }

```

(End definition for __fp_parse_word_abs:N and others.)

```

\__fp_parse_word_acos:N
\__fp_parse_word_acosd:N
\__fp_parse_word_acsc:N
\__fp_parse_word_acscd:N
\__fp_parse_word_asec:N
\__fp_parse_word_asecd:N
\__fp_parse_word_asin:N
\__fp_parse_word_asind:N
\__fp_parse_word_cos:N
\__fp_parse_word_cosd:N
\__fp_parse_word_cot:N
\__fp_parse_word_cotd:N
\__fp_parse_word_csc:N
\__fp_parse_word_cscd:N
\__fp_parse_word_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.

```

11087 \tl_map_inline:nn
11088 {
11089   {acos} {acsc} {asec} {asin}
11090   {cos} {cot} {csc} {sec} {sin} {tan}
11091 }
11092 {
11093   \cs_new_nopar:cpn { \__fp_parse_word_#1:N }
11094   { \__fp_parse_unary_function:nNN {#1} \use_i:nn }
11095   \cs_new_nopar:cpn { \__fp_parse_word_#1d:N }
11096   { \__fp_parse_unary_function:nNN {#1} \use_ii:nn }
11097 }

```

(End definition for __fp_parse_word_acos:N and others.)


```

\__fp_parse_word_trunc:N
\__fp_parse_word_floor:N
\__fp_parse_word_ceil:N
11098 \cs_new_nopar:Npn \__fp_parse_word_trunc:N
11099 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_zero:NNN }
11100 \cs_new_nopar:Npn \__fp_parse_word_floor:N
11101 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_ninf:NNN }
11102 \cs_new_nopar:Npn \__fp_parse_word_ceil:N
11103 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_pinf:NNN }
(End definition for \__fp_parse_word_trunc:N, \__fp_parse_word_floor:N, and \__fp_parse_word_ceil:N.)

\__fp_parse_word_round:N
\__fp_parse_round:Nw
11104 \cs_new:Npn \__fp_parse_word_round:N #1#2
11105 {
11106   \if_meaning:w + #2
11107     \__fp_parse_round:Nw \__fp_round_to_pinf:NNN
11108   \else:
11109     \if_meaning:w 0 #2
11110       \__fp_parse_round:Nw \__fp_round_to_zero:NNN
11111     \else:
11112       \if_meaning:w - #2
11113         \__fp_parse_round:Nw \__fp_round_to_ninf:NNN
11114       \fi:
11115     \fi:
11116   \fi:
11117   \__fp_parse_function:NNN
11118   \__fp_round_o:Nw \__fp_round_to_nearest:NNN #1
11119   #2
11120 }
11121 \cs_new:Npn \__fp_parse_round:Nw
11122   #1 #2 \__fp_round_to_nearest:NNN #3#4 { #2 #1 #3 }
(End definition for \__fp_parse_word_round:N and \__fp_parse_round:Nw.)

```

27.6 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, namely, the end of the expression. The marker `\s__fp_mark` indicates that the next token is an already parsed version of an infix operator, and `__fp_parse_infix_end:N` has infinitely negative precedence. Finally, clean up a (well-defined) set of extra tokens and stop the initial expansion with `\c_zero`.

```

11123 \cs_new:Npn \__fp_parse:n #1
11124 {
11125   \tex_romannumeral:D
11126   \exp_after:wN \__fp_parse_after:ww
11127   \tex_romannumeral:D
11128   \__fp_parse_operand:Nw \c_minus_one
11129   \__fp_parse_expand:w #1
11130   \s__fp_mark \__fp_parse_infix_end:N
11131   \s__fp_stop

```

```

11132 }
11133 \cs_new:Npn \__fp_parse_after:ww
11134   #1@ \__fp_parse_infix_end:N \s__fp_stop
11135   { \c_zero #1 }
(End definition for \__fp_parse:n.)

```

__fp_parse_operand:Nw
 __fp_parse_continue:NwN

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.

```

11136 \cs_new:Npn \__fp_parse_operand:Nw #1
11137   {
11138     -'0
11139     \exp_after:wN \__fp_parse_continue:NwN
11140     \exp_after:wN #1
11141     \tex_romannumeral:D -'0
11142     \exp_after:wN \__fp_parse_one:Nw
11143     \exp_after:wN #1
11144     \tex_romannumeral:D
11145   }
11146 \cs_new:Npn \__fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }
(End definition for \__fp_parse_operand:Nw.)

```

__fp_parse_apply_binary:NwNwN

Receives $\langle precedence \rangle \langle operand_1 \rangle @ \langle operation \rangle \langle operand_2 \rangle @ \langle infix command \rangle$. Builds the appropriate call to the $\langle operation \rangle$ #3.

```

11147 \cs_new:Npn \__fp_parse_apply_binary:NwNwN #1 #2@ #3 #4@ #5
11148   {
11149     \exp_after:wN \__fp_parse_continue:NwN
11150     \exp_after:wN #1
11151     \tex_romannumeral:D -'0 \cs:w __fp_#3_o:ww \cs_end: #2 #4
11152     \tex_romannumeral:D -'0 #5 #1
11153   }
(End definition for \__fp_parse_apply_binary:NwNwN.)

```

27.7 Infix operators

__fp_parse_infix_after_operand:NwN

```

11154 \cs_new:Npn \__fp_parse_infix_after_operand:NwN #1 #2;
11155   {
11156     \__fp_exp_after_f:nw { \__fp_parse_infix:NN #1 }
11157     #2;
11158   }
11159 \group_begin:
11160   \char_set_catcode_letter:N \*
11161   \cs_new:Npn \__fp_parse_infix:NN #1 #2
11162     {
11163       \if_catcode:w \scan_stop: \exp_not:N #2
11164       \if_int_compare:w
11165         \__str_if_eq_x:nn { \s__fp_mark } { \exp_not:N #2 }

```

```

11166         = \c_zero
11167         \exp_after:wN \exp_after:wN
11168         \exp_after:wN \_fp_parse_infix_mark:NNN
11169     \else:
11170         \exp_after:wN \exp_after:wN
11171         \exp_after:wN \_fp_parse_infix_juxtapose:N
11172     \fi:
11173 \else:
11174     \if_int_compare:w
11175         \_int_eval:w
11176         ( '#2 \if_int_compare:w '#2 > 'Z - \c_thirty_two \fi: )
11177         / 26
11178         = \c_three
11179         \exp_after:wN \exp_after:wN
11180         \exp_after:wN \_fp_parse_infix_juxtapose:N
11181     \else:
11182         \exp_after:wN \_fp_parse_infix_check:NNN
11183         \cs:w
11184         \_fp_parse_infix_#2:N
11185         \exp_after:wN \exp_after:wN \exp_after:wN
11186         \cs_end:
11187     \fi:
11188 \fi:
11189 #1
11190 #2
11191 }
11192 \cs_new:Npn \_fp_parse_infix_check:NNN #1#2#3
11193 {
11194     \if_meaning:w \scan_stop: #1
11195         \_msg_kernel_expandable_error:nnn
11196         { kernel } { fp-missing } { * }
11197         \exp_after:wN \_fp_parse_infix_*:N
11198         \exp_after:wN #2
11199         \exp_after:wN #3
11200     \else:
11201         \exp_after:wN #1
11202         \exp_after:wN #2
11203         \tex_romannumeral:D \exp_after:wN \_fp_parse_expand:w
11204     \fi:
11205 }
11206 \group_end:

```

(End definition for _fp_parse_infix_after_operand:NwN.)

27.7.1 Closing parentheses and commas

_fp_parse_infix_mark:NNN As an infix operator, \s_fp_mark means that the next token (#3) has already gone through _fp_parse_infix:NN and should be provided the precedence #1. The scan mark #2 is discarded.

```

11207 \cs_new:Npn \_fp_parse_infix_mark:NNN #1#2#3 { #3 #1 }

```

(End definition for `__fp_parse_infix_mark:NNN`.)

`__fp_parse_infix_end:N` This one is a little bit odd: force every previous operator to end, regardless of the precedence.

```
11208 \cs_new:Npn \__fp_parse_infix_end:N #1
11209 { @ \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.

```
11210 \group_begin:
11211 \char_set_catcode_letter:N \)
11212 \cs_new:Npn \__fp_parse_infix_):N #1
11213 {
11214   \if_int_compare:w #1 < \c_zero
11215     \__msg_kernel_expandable_error:nnn { kernel } { fp-extra } { ) }
11216     \exp_after:wN \__fp_parse_infix:NN
11217     \exp_after:wN #1
11218     \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
11219   \else:
11220     \exp_after:wN @
11221     \exp_after:wN \use_none:n
11222     \exp_after:wN \__fp_parse_infix_):N
11223   \fi:
11224 }
11225 \group_end:
```

(End definition for `__fp_parse_infix_):N`.)

`__fp_parse_infix_`

:N

```
11226 \group_begin:
11227 \char_set_catcode_letter:N \,
11228 \cs_new:Npn \__fp_parse_infix_,:N #1
11229 {
11230   \if_int_compare:w #1 > \c_one
11231     \exp_after:wN @
11232     \exp_after:wN \use_none:n
11233     \exp_after:wN \__fp_parse_infix_,:N
11234   \else:
11235     \if_int_compare:w #1 = \c_one
11236       \exp_after:wN \__fp_parse_infix_comma:w
11237       \tex_romannumeral:D
11238     \else:
11239       \exp_after:wN \__fp_parse_infix_comma_gobble:w
11240       \tex_romannumeral:D
11241     \fi:
11242     \__fp_parse_operand:Nw \c_one
11243     \exp_after:wN \__fp_parse_expand:w
11244   \fi:
11245 }
```

```

11246 \cs_new:Npn \__fp_parse_infix_comma:w #1 @
11247 { #1 @ \use_none:n }
11248 \cs_new:Npn \__fp_parse_infix_comma_gobble:w #1 @
11249 {
11250   \__msg_kernel_expandable_error:nn { kernel } { fp-extra-comma }
11251   @ \use_none:n
11252 }
11253 \group_end:
(End definition for \__fp_parse_infix_ and :N.)

```

27.7.2 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.

The odd requirement to set `\+` here is to cover the case where `expl3` is loaded by plain TeX: `\+` is an `\outer` macro there, and so the following code would otherwise give an error in that case.

```

11254 \group_begin:
11255 <*package>
11256 \cs_set_nopar:Npn \+ { }
11257 </package>
11258 \char_set_catcode_other:N \&
11259 \char_set_catcode_letter:N \^
11260 \char_set_catcode_letter:N \/
11261 \char_set_catcode_letter:N \-
11262 \char_set_catcode_letter:N \+
11263 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
11264 {
11265   \cs_new:Npn #1 ##1
11266   {
11267     \if_int_compare:w ##1 < #3
11268       \exp_after:wN @
11269       \exp_after:wN \__fp_parse_apply_binary:NwNwN
11270       \exp_after:wN #2
11271       \tex_romannumeral:D
11272       \__fp_parse_operand:Nw #4
11273       \exp_after:wN \__fp_parse_expand:w
11274     \else:
11275       \exp_after:wN @
11276       \exp_after:wN \use_none:n
11277       \exp_after:wN #1
11278     \fi:
11279   }
11280 }
11281 \__fp_tmp:w \__fp_parse_infix^:N ^ \c_fifteen \c_fourteen
11282 \__fp_tmp:w \__fp_parse_infix/:N / \c_ten \c_ten
11283 \__fp_tmp:w \__fp_parse_infix_mul:N * \c_ten \c_ten

```

```

11284 \__fp_tmp:w \__fp_parse_infix -:N - \c_nine \c_nine
11285 \__fp_tmp:w \__fp_parse_infix +:N + \c_nine \c_nine
11286 \__fp_tmp:w \__fp_parse_infix_and:N & \c_five \c_five
11287 \__fp_tmp:w \__fp_parse_infix_or:N | \c_four \c_four
11288 \group_end:
(End definition for \__fp_parse_infix +:N and others.)

```

27.7.3 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`.

```

11289 \cs_new:cpn { \__fp_parse_infix_(:N } #1
11290 { \__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 `... (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.

```

11291 \cs_new:Npn \__fp_parse_infix_juxtapose:N #1
11292 {
11293   \if_int_compare:w #1 < \c_ten
11294     \exp_after:wN @
11295     \exp_after:wN \__fp_parse_apply_juxtapose:NwwN
11296     \tex_romannumeral:D
11297     \__fp_parse_operand:Nw \c_ten
11298     \exp_after:wN \__fp_parse_expand:w
11299   \else:
11300     \exp_after:wN @
11301     \exp_after:wN \use_none:n
11302     \exp_after:wN \__fp_parse_infix_juxtapose:N
11303   \fi:
11304 }
11305 \cs_new:Npn \__fp_parse_apply_juxtapose:NwwN #1 #2;#3@ #4;#5@
11306 {
11307   \if_catcode:w ^ \tl_to_str:n { #3 #5 } ^
11308   \else:
11309     \__fp_error:nffn { invalid-ii }
11310     { \__fp_array_to_clist:n { #2; #3 } }
11311     { \__fp_array_to_clist:n { #4; #5 } }
11312     { }
11313   \fi:
11314   \__fp_parse_apply_binary:NwNwN #1 #2;@ * #4;@
11315 }

```

(End definition for `__fp_parse_infix_juxtapose:N` and `__fp_parse_apply_juxtapose:NwwN`.)

27.7.4 Multi-character cases

`__fp_parse_infix_*:N`

```

11316 \group_begin:
11317   \char_set_catcode_letter:N ^
11318   \cs_new:cpn { __fp_parse_infix_*:N } #1#2
11319   {
11320     \if:w * \exp_not:N #2
11321       \exp_after:wN \__fp_parse_infix_^:N
11322       \exp_after:wN #1
11323     \else:
11324       \exp_after:wN \__fp_parse_infix_mul:N
11325       \exp_after:wN #1
11326       \exp_after:wN #2
11327     \fi:
11328   }
11329 \group_end:

```

(End definition for __fp_parse_infix_:N.)*

`__fp_parse_infix_|:Nw`

`__fp_parse_infix_&:Nw`

```

11330 \group_begin:
11331   \char_set_catcode_letter:N \
11332   \char_set_catcode_letter:N &
11333   \cs_new:Npn \__fp_parse_infix_|:N #1#2
11334   {
11335     \if:w | \exp_not:N #2
11336       \exp_after:wN \__fp_parse_infix_|:N
11337       \exp_after:wN #1
11338       \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
11339     \else:
11340       \exp_after:wN \__fp_parse_infix_or:N
11341       \exp_after:wN #1
11342       \exp_after:wN #2
11343     \fi:
11344   }
11345   \cs_new:Npn \__fp_parse_infix_&:N #1#2
11346   {
11347     \if:w & \exp_not:N #2
11348       \exp_after:wN \__fp_parse_infix_&:N
11349       \exp_after:wN #1
11350       \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
11351     \else:
11352       \exp_after:wN \__fp_parse_infix_and:N
11353       \exp_after:wN #1
11354       \exp_after:wN #2
11355     \fi:
11356   }
11357 \group_end:

```

(End definition for __fp_parse_infix_|:Nw.)

27.7.5 Ternary operator

```

\__fp_parse_infix_?:N
\__fp_parse_infix_::N
11358 \group_begin:
11359 \char_set_catcode_letter:N \?
11360 \cs_new:Npn \__fp_parse_infix_?:N #1
11361 {
11362   \if_int_compare:w #1 < \c_three
11363     \exp_after:wN @
11364     \exp_after:wN \__fp_ternary:NwwN
11365     \tex_romannumeral:D
11366     \__fp_parse_operand:Nw \c_three
11367     \exp_after:wN \__fp_parse_expand:w
11368   \else:
11369     \exp_after:wN @
11370     \exp_after:wN \use_none:n
11371     \exp_after:wN \__fp_parse_infix_?:N
11372   \fi:
11373 }
11374 \cs_new:Npn \__fp_parse_infix_::N #1
11375 {
11376   \if_int_compare:w #1 < \c_three
11377     \__msg_kernel_expandable_error:nnnn
11378     { kernel } { fp-missing } { ? } { ~for~?: }
11379     \exp_after:wN @
11380     \exp_after:wN \__fp_ternary_auxii:NwwN
11381     \tex_romannumeral:D
11382     \__fp_parse_operand:Nw \c_two
11383     \exp_after:wN \__fp_parse_expand:w
11384   \else:
11385     \exp_after:wN @
11386     \exp_after:wN \use_none:n
11387     \exp_after:wN \__fp_parse_infix_::N
11388   \fi:
11389 }
11390 \group_end:
(End definition for \__fp_parse_infix_?:N and \__fp_parse_infix_::N.)

```

27.7.6 Comparisons

```

\__fp_parse_infix_<:N
\__fp_parse_infix_=:N
\__fp_parse_infix_>:N
\__fp_parse_infix_!:N
\__fp_parse_excl_error:
\__fp_parse_compare:NNNNNNN
\__fp_parse_compare_auxi:NNNNNNN
\__fp_parse_compare_auxii:NNNNN
\__fp_parse_compare_end:NNNNw
\__fp_compare:wNNNNw
11391 \cs_new:cpn { __fp_parse_infix_<:N } #1
11392 {
11393   \__fp_parse_compare:NNNNNNN #1 \c_one
11394   \c_zero \c_zero \c_zero \c_zero <
11395 }
11396 \cs_new:cpn { __fp_parse_infix_=:N } #1
11397 {
11398   \__fp_parse_compare:NNNNNNN #1 \c_one

```



```

11399         \c_zero \c_zero \c_zero \c_zero =
11400     }
11401 \cs_new:cpn { __fp_parse_infix_>:N } #1
11402 {
11403     \__fp_parse_compare:NNNNNNN #1 \c_one
11404     \c_zero \c_zero \c_zero \c_zero >
11405 }
11406 \cs_new:cpn { __fp_parse_infix_!:N } #1
11407 {
11408     \exp_after:wN \__fp_parse_compare:NNNNNNN
11409     \exp_after:wN #1
11410     \exp_after:wN \c_zero
11411     \exp_after:wN \c_one
11412     \exp_after:wN \c_one
11413     \exp_after:wN \c_one
11414     \exp_after:wN \c_one
11415 }
11416 \cs_new:Npn \__fp_parse_excl_error:
11417 {
11418     \_msg_kernel_expandable_error:nnnn
11419     { kernel } { fp-missing } { = } { ~after~!. }
11420 }
11421 \cs_new:Npn \__fp_parse_compare:NNNNNNN #1
11422 {
11423     \if_int_compare:w #1 < \c_seven
11424         \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
11425         \exp_after:wN \__fp_parse_excl_error:
11426     \else:
11427         \exp_after:wN @
11428         \exp_after:wN \use_none:n
11429         \exp_after:wN \__fp_parse_compare:NNNNNNN
11430     \fi:
11431 }
11432 \cs_new:Npn \__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
11433 {
11434     \if_case:w
11435         \if_catcode:w \scan_stop: \exp_not:N #7
11436         \c_minus_one
11437     \else:
11438         \__int_eval:w '#7 - '< \__int_eval_end:
11439     \fi:
11440     \__fp_parse_compare_auxii:NNNNN #2#2#4#5#6
11441     \or: \__fp_parse_compare_auxii:NNNNN #2#3#2#5#6
11442     \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#2#6
11443     \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#5#2
11444     \else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
11445     \fi:
11446 }
11447 \cs_new:Npn \__fp_parse_compare_auxii:NNNNN #1#2#3#4#5
11448 {

```

```

11449 \exp_after:wN \_fp_parse_compare_auxi:NNNNNNN
11450 \exp_after:wN \prg_do_nothing:
11451 \exp_after:wN #1
11452 \exp_after:wN #2
11453 \exp_after:wN #3
11454 \exp_after:wN #4
11455 \exp_after:wN #5
11456 \tex_romannumeral:D \exp_after:wN \_fp_parse_expand:w
11457 }
11458 \cs_new:Npn \_fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
11459 {
11460 \fi:
11461 \exp_after:wN @
11462 \exp_after:wN \_fp_parse_apply_compare:NwNNNNNwN
11463 \exp_after:wN \c_one_fp
11464 \exp_after:wN #1
11465 \exp_after:wN #2
11466 \exp_after:wN #3
11467 \exp_after:wN #4
11468 \tex_romannumeral:D
11469 \_fp_parse_operand:Nw \c_seven \_fp_parse_expand:w #5
11470 }
11471 \cs_new:Npn \_fp_parse_apply_compare:NwNNNNNwN
11472 #1 #2@ #3 #4#5#6#7 #8@ #9
11473 {
11474 \if_int_odd:w
11475 \if_meaning:w \c_zero_fp #3
11476 \c_zero
11477 \else:
11478 \if_case:w \_fp_compare_back:ww #8 #2 \exp_stop_f:
11479 #5 \or: #6 \or: #7 \else: #4
11480 \fi:
11481 \fi:
11482 \exp_after:wN \_fp_parse_apply_compare_aux:NNwN
11483 \exp_after:wN \c_one_fp
11484 \else:
11485 \exp_after:wN \_fp_parse_apply_compare_aux:NNwN
11486 \exp_after:wN \c_zero_fp
11487 \fi:
11488 #1 #8 #9
11489 }
11490 \cs_new:Npn \_fp_parse_apply_compare_aux:NNwN #1 #2 #3; #4
11491 {
11492 \if_meaning:w \_fp_parse_compare:NNNNNNN #4
11493 \exp_after:wN \_fp_parse_continue_compare:NNwNN
11494 \exp_after:wN #1
11495 \exp_after:wN #2
11496 \tex_romannumeral:D -'0
11497 \_fp_exp_after_o:w #3;
11498 \tex_romannumeral:D -'0

```

```

11499 \else:
11500 \exp_after:wN \__fp_parse_continue:NwN
11501 \exp_after:wN #2
11502 \tex_romannumeral:D -‘0
11503 \exp_after:wN #1
11504 \tex_romannumeral:D -‘0
11505 \fi:
11506 #4 #2
11507 }
11508 \cs_new:Npn \__fp_parse_continue_compare:NNwNN #1#2 #3@ #4#5
11509 { #4 #2 #3@ #1 }
(End definition for \__fp_parse_infix_<:N and others.)

```

27.8 Candidate: defining new l3fp functions

`\fp_function:Nw` Parse the argument of the function #1 using `__fp_parse_operand:Nw` with a precedence of 16, and pass the function and argument to `__fp_function_apply:nw`.

```

11510 \cs_new:Npn \fp_function:Nw #1
11511 {
11512 \exp_after:wN \__fp_function_apply:nw
11513 \exp_after:wN #1
11514 \tex_romannumeral:D
11515 \__fp_parse_operand:Nw \c_sixteen \__fp_parse_expand:w
11516 }
(End definition for \fp_function:Nw. This function is documented on page ??.)

```

`\fp_new_function:Npn` Save the code provided by the user in the control sequence `__fp_user_#1`. Define `__fp_new_function:NNnnn` #1 to call `__fp_function_apply:nw` after parsing one operand using `__fp_parse_operand:Nw` with precedence 16. The auxiliary `__fp_function_args:Nwn` receives the user function and the number of arguments (half of the number of tokens in the parameter text #2), followed by the operand (as a token list of floating points). It checks the number of arguments, and applies the user function to the arguments (without the outer brace group).

```

11517 \cs_new_protected:Npn \fp_new_function:Npn #1#2#
11518 {
11519 \__fp_new_function:Ncfnn #1
11520 { __fp_user_ \cs_to_str:N #1 }
11521 { \int_eval:n { \tl_count:n {#2} / \c_two } }
11522 {#2}
11523 }
11524 \cs_new_protected:Npn \__fp_new_function:NNnnn #1#2#3#4#5
11525 {
11526 \cs_new_nopar:Npn #1
11527 {
11528 \exp_after:wN \__fp_function_apply:nw \exp_after:wN
11529 {
11530 \exp_after:wN \__fp_function_args:Nwn
11531 \exp_after:wN #2

```

```

11532         \__int_value:w #3 \exp_after:wN ; \exp_after:wN
11533     }
11534     \tex_romannumeral:D
11535     \__fp_parse_operand:Nw \c_sixteen \__fp_parse_expand:w
11536 }
11537 \cs_new:Npn #2 #4 {#5}
11538 }
11539 \cs_generate_variant:Nn \__fp_new_function:NNnnn { Ncf }
11540 \cs_new:Npn \__fp_function_args:Nwn #1#2; #3
11541 {
11542     \int_compare:nNnTF { \tl_count:n {#3} } = {#2}
11543     { #1 #3 }
11544     {
11545         \__msg_kernel_expandable_error:nnnnn
11546         { kernel } { fp-num-args } { #1() } {#2} {#2}
11547         \c_nan_fp
11548     }
11549 }

```

(End definition for `\fp_new_function:Npn`. This function is documented on page ??.)

```

\__fp_function_apply:nw
\__fp_function_store:wwNwnn
\__fp_function_store_end:wnnn

```

The auxiliary `__fp_function_apply:nw` is called after parsing an operand, so it receives some code `#1`, then the operand ending with `@`, then a function such as `__fp_parse_infix_+:N` (but not always of this form, see comparisons for instance). Package the operand (an array) into a token list with floating point items: this is the role of `__fp_function_store:wwNwnn` and `__fp_function_store_end:wnnn`. Then apply `__fp_parse:n` to the code `#1` followed by a brace group with this token list. This results in a floating point result, which will correctly be parsed as the next operand of whatever was looking for one. The trailing `\s__fp_mark` is used as a special infix operator to indicate that the next token has already gone through `__fp_parse_infix:NN`.

```

11550 \cs_new:Npn \__fp_function_apply:nw #1#2 @
11551 {
11552     \__fp_parse:n
11553     {
11554         \__fp_function_store:wwNwnn #2
11555         \s__fp_mark \__fp_function_store:wwNwnn ;
11556         \s__fp_mark \__fp_function_store_end:wnnn
11557         \s__fp_stop { } { } {#1}
11558     }
11559     \s__fp_mark
11560 }
11561 \cs_new:Npn \__fp_function_store:wwNwnn
11562     #1; #2 \s__fp_mark #3#4 \s__fp_stop #5#6
11563     { #3 #2 \s__fp_mark #3#4 \s__fp_stop { #5 #6 } { { #1; } } }
11564 \cs_new:Npn \__fp_function_store_end:wnnn
11565     #1 \s__fp_stop #2#3#4
11566     { #4 {#2} }

```

(End definition for `__fp_function_apply:nw`, `__fp_function_store:wwNwnn`, and `__fp_function_store_end:wnnn`.)

27.9 Messages

```

11567 \_msg_kernel_new:nnn { kernel } { unknown-fp-word }
11568 { Unknown~fp~word~#1. }
11569 \_msg_kernel_new:nnn { kernel } { fp-missing }
11570 { Missing~#1~inserted #2. }
11571 \_msg_kernel_new:nnn { kernel } { fp-extra }
11572 { Extra~#1~ignored. }
11573 \_msg_kernel_new:nnn { kernel } { fp-early-end }
11574 { Premature~end~in~fp~expression. }
11575 \_msg_kernel_new:nnn { kernel } { fp-after-e }
11576 { Cannot~use~#1 after~'e'. }
11577 \_msg_kernel_new:nnn { kernel } { fp-missing-number }
11578 { Missing~number~before~'#1'. }
11579 \_msg_kernel_new:nnn { kernel } { fp-unknown-symbol }
11580 { Unknown~symbol~#1~ignored. }
11581 \_msg_kernel_new:nnn { kernel } { fp-extra-comma }
11582 { Unexpected~comma:~extra~arguments~ignored. }
11583 \_msg_kernel_new:nnn { kernel } { fp-num-args }
11584 { #1~expects~between~#2~and~#3~arguments. }
11585 </initex | package>

```

28 l3fp-logic Implementation

```

11586 <*initex | package>
11587 <@@=fp>

```

28.1 Syntax of internal functions

- `_fp_compare_npos:nwnw` { $\langle expo_1 \rangle$ } $\langle body_1 \rangle$; { $\langle expo_2 \rangle$ } $\langle body_2 \rangle$;
- `_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.

28.2 Existence test

`\fp_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\fp_if_exist_p:c 11588 \prg_new_eq_conditional:NNn \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }
\fp_if_exist:NTF 11589 \prg_new_eq_conditional:NNn \fp_if_exist:c \cs_if_exist:c { TF , T , F , p }
\fp_if_exist:cTF (End definition for \fp_if_exist:N and \fp_if_exist:c. These functions are documented on page ??.)

```

28.3 Comparison

`\fp_compare_p:n` Within floating point expressions, comparison operators are treated as operations, so we evaluate `#1`, then compare with 0.

`\fp_compare:nTF`

```
\__fp_compare_return:w 11590 \prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
11591 {
11592   \exp_after:wN \__fp_compare_return:w
11593   \tex_romannumeral:D -'0 \__fp_parse:n {#1}
11594 }
11595 \cs_new:Npn \__fp_compare_return:w \s_fp \__fp_chk:w #1#2;
11596 {
11597   \if_meaning:w 0 #1
11598   \prg_return_false:
11599   \else:
11600   \prg_return_true:
11601   \fi:
11602 }
```

(End definition for `\fp_compare:n`. These functions are documented on page 180.)

`\fp_compare_p:nNn`

`\fp_compare:nNnTF`

`__fp_compare_aux:wn`

Evaluate `#1` and `#3`, using an auxiliary to expand both, and feed the two floating point numbers swapped to `__fp_compare_back:ww`, defined below. Compare the result with `'#2-'=`, which is -1 for $<$, 0 for $=$, 1 for $>$ and 2 for $?$.

```
11603 \prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
11604 {
11605   \if_int_compare:w
11606     \exp_after:wN \__fp_compare_aux:wn
11607     \tex_romannumeral:D -'0 \__fp_parse:n {#1} {#3}
11608     = \__int_eval:w '#2 - '= \__int_eval_end:
11609     \prg_return_true:
11610   \else:
11611     \prg_return_false:
11612   \fi:
11613 }
11614 \cs_new:Npn \__fp_compare_aux:wn #1; #2
11615 {
11616   \exp_after:wN \__fp_compare_back:ww
11617   \tex_romannumeral:D -'0 \__fp_parse:n {#2} #1;
11618 }
```

(End definition for `\fp_compare:nNn`. These functions are documented on page 180.)

`__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.

```

11619 \cs_new:Npn \__fp_compare_back:ww
11620   \s__fp \__fp_chk:w #1 #2 #3;
11621   \s__fp \__fp_chk:w #4 #5 #6;
11622   {
11623     \__int_value:w
11624     \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
11625     \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
11626     \if_meaning:w 2 #5 - \fi:
11627     \if_meaning:w #2 #5
11628     \if_meaning:w #1 #4
11629     \if_meaning:w 1 #1
11630     \__fp_compare_npos:nwnw #6; #3;
11631     \else:
11632       0
11633     \fi:
11634     \else:
11635     \if_int_compare:w #4 < #1 - \fi: 1
11636     \fi:
11637     \else:
11638     \if_int_compare:w #1#4 = \c_zero
11639       0
11640     \else:
11641       1
11642     \fi:
11643     \fi:
11644     \exp_stop_f:
11645   }
11646 \cs_new:Npn \__fp_compare_nan:w #1 \exp_stop_f: { \c_two }
(End definition for \__fp_compare_back:ww and \__fp_compare_nan:w.)

```

__fp_compare_npos:nwnw __fp_compare_npos:nwnw { $\langle expo_1 \rangle$ } $\langle body_1 \rangle$; { $\langle expo_2 \rangle$ } $\langle body_2 \rangle$;

__fp_compare_significand:nnnnnnnn Within an __int_value:w ... \exp_stop_f: construction, this expands to 0 if the two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```

11647 \cs_new:Npn \__fp_compare_npos:nwnw #1#2; #3#4;
11648   {
11649     \if_int_compare:w #1 = #3 \exp_stop_f:
11650     \__fp_compare_significand:nnnnnnnn #2 #4
11651     \else:
11652     \if_int_compare:w #1 < #3 - \fi: 1
11653     \fi:
11654   }
11655 \cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
11656   {
11657     \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
11658     \if_int_compare:w #3#4 = #7#8 \exp_stop_f:

```

```

11659         0
11660         \else:
11661             \if_int_compare:w #3#4 < #7#8 - \fi: 1
11662         \fi:
11663     \else:
11664         \if_int_compare:w #1#2 < #5#6 - \fi: 1
11665     \fi:
11666 }

```

(End definition for _fp_compare_npos:nwnw.)

28.4 Floating point expression loops

\fp_do_until:nn These are quite easy given the above functions. The `do_until` and `do_while` versions execute the body, then test. The `until_do` and `while_do` do it the other way round.

```

\fp_do_while:nn
\fp_until_do:nn
\fp_while_do:nn
11667 \cs_new:Npn \fp_do_until:nn #1#2
11668 {
11669     #2
11670     \fp_compare:nF {#1}
11671     { \fp_do_until:nn {#1} {#2} }
11672 }
11673 \cs_new:Npn \fp_do_while:nn #1#2
11674 {
11675     #2
11676     \fp_compare:nT {#1}
11677     { \fp_do_while:nn {#1} {#2} }
11678 }
11679 \cs_new:Npn \fp_until_do:nn #1#2
11680 {
11681     \fp_compare:nF {#1}
11682     {
11683         #2
11684         \fp_until_do:nn {#1} {#2}
11685     }
11686 }
11687 \cs_new:Npn \fp_while_do:nn #1#2
11688 {
11689     \fp_compare:nT {#1}
11690     {
11691         #2
11692         \fp_while_do:nn {#1} {#2}
11693     }
11694 }

```

(End definition for `\fp_do_until:nn` and others. These functions are documented on page 182.)

\fp_do_until:nNnn As above but not using the `nNn` syntax.

```

\fp_do_while:nNnn
\fp_until_do:nNnn
\fp_while_do:nNnn
11695 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
11696 {
11697     #4
11698     \fp_compare:nNnF {#1} #2 {#3}

```



```

11699     { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
11700   }
11701   \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
11702   {
11703     #4
11704     \fp_compare:nNnT {#1} #2 {#3}
11705     { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
11706   }
11707   \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
11708   {
11709     \fp_compare:nNnF {#1} #2 {#3}
11710     {
11711       #4
11712       \fp_until_do:nNnn {#1} #2 {#3} {#4}
11713     }
11714   }
11715   \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
11716   {
11717     \fp_compare:nNnT {#1} #2 {#3}
11718     {
11719       #4
11720       \fp_while_do:nNnn {#1} #2 {#3} {#4}
11721     }
11722   }

```

(End definition for `\fp_do_until:nNnn` and others. These functions are documented on page [181](#).)

28.5 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`.

```

11723   \cs_new:Npn \__fp_minmax_o:Nw #1#2 @
11724   {
11725     \if_meaning:w 0 #1
11726     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN \c_one
11727     \else:
11728     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN \c_minus_one
11729     \fi:
11730     #2
11731     \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
11732     \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
11733   }

```

(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.

```

11734 \cs_new:Npn \__fp_minmax_loop:Nww
11735   #1 \s__fp \__fp_chk:w #2#3; \s__fp \__fp_chk:w #4#5;
11736   {
11737     \if_meaning:w 3 #4
11738     \if_meaning:w 3 #2
11739       \__fp_minmax_auxi:ww
11740     \else:
11741       \__fp_minmax_auxii:ww
11742     \fi:
11743   \else:
11744     \if_int_compare:w
11745       \__fp_compare_back:ww
11746       \s__fp \__fp_chk:w #4#5;
11747       \s__fp \__fp_chk:w #2#3;
11748       = #1
11749       \__fp_minmax_auxii:ww
11750     \else:
11751       \__fp_minmax_auxi:ww
11752     \fi:
11753   \fi:
11754   \__fp_minmax_loop:Nww #1
11755   \s__fp \__fp_chk:w #2#3;
11756   \s__fp \__fp_chk:w #4#5;
11757 }
(End definition for \__fp_minmax_loop:Nww.)

```

`__fp_minmax_auxi:ww` Keep the first/second number, and remove the other.
`__fp_minmax_auxii:ww`

```

11758 \cs_new:Npn \__fp_minmax_auxi:ww #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
11759 { \fi: \fi: #2 \s__fp #3 ; }
11760 \cs_new:Npn \__fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
11761 { \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.

```

11762 \cs_new:Npn \__fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3; #4;
11763 { \fi: \__fp_exp_after_o:w \s__fp #3; }
(End definition for \__fp_minmax_break_o:w.)

```

28.6 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.

```

11764 \cs_new:cpn { __fp_not_o:w } #1 \s__fp \__fp_chk:w #2#3; @
11765 {
11766     \if_meaning:w 0 #2
11767     \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
11768     \else:
11769     \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
11770     \fi:
11771 }

```

(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.

```

11772 \group_begin:
11773 \char_set_catcode_letter:N &
11774 \char_set_catcode_letter:N |
11775 \cs_new:Npn \__fp_&o:ww #1 \s__fp \__fp_chk:w #2#3;
11776 {
11777     \if_meaning:w 0 #2 #1
11778     \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
11779     \fi:
11780     \__fp_exp_after_o:w
11781 }
11782 \cs_new_nopar:Npn \__fp_|o:ww { \__fp_&o:ww \else: }
11783 \group_end:
11784 \cs_new:Npn \__fp_and_return:wNw #1; \fi: #2#3; { \fi: #2 #1; }

```

(End definition for `__fp_&o:ww`.)

28.7 Ternary operator

The first function receives the test and the true branch of the `?:` ternary operator. It returns the true branch, unless the test branch is zero. In that case, the function returns a very specific `nan`. The second function receives the output of the first function, and the false branch. It returns the previous input, unless that is the special `nan`, in which case we return the false branch.

```

\__fp_ternary:NwwN
\__fp_ternary_auxi:NwwN
\__fp_ternary_auxii:NwwN
\__fp_ternary_loop_break:w
\__fp_ternary_loop:Nw
\__fp_ternary_map_break:
\__fp_ternary_break_point:n
11785 \cs_new:Npn \__fp_ternary:NwwN #1 #2@ #3@ #4
11786 {
11787     \if_meaning:w \__fp_parse_infix_:N #4
11788     \__fp_ternary_loop:Nw
11789     #2
11790     \s__fp \__fp_chk:w { \__fp_ternary_loop_break:w } ;
11791     \__fp_ternary_break_point:n { \exp_after:wN \__fp_ternary_auxi:NwwN }

```

```

11792 \exp_after:wN #1
11793 \tex_romannumeral:D -'0
11794 \__fp_exp_after_array_f:w #3 \s__fp_stop
11795 \exp_after:wN @
11796 \tex_romannumeral:D
11797 \__fp_parse_operand:Nw \c_two
11798 \__fp_parse_expand:w
11799 \else:
11800 \__msg_kernel_expandable_error:nnnn
11801 { kernel } { fp-missing } { : } { ~for~?: }
11802 \exp_after:wN \__fp_parse_continue:NwN
11803 \exp_after:wN #1
11804 \tex_romannumeral:D -'0
11805 \__fp_exp_after_array_f:w #3 \s__fp_stop
11806 \exp_after:wN #4
11807 \exp_after:wN #1
11808 \fi:
11809 }
11810 \cs_new:Npn \__fp_ternary_loop_break:w #1 \fi: #2 \__fp_ternary_break_point:n #3
11811 {
11812 \c_zero = \c_zero \fi:
11813 \exp_after:wN \__fp_ternary_auxii:NwwN
11814 }
11815 \cs_new:Npn \__fp_ternary_loop:Nw \s__fp \__fp_chk:w #1#2;
11816 {
11817 \if_int_compare:w #1 > \c_zero
11818 \exp_after:wN \__fp_ternary_map_break:
11819 \fi:
11820 \__fp_ternary_loop:Nw
11821 }
11822 \cs_new:Npn \__fp_ternary_map_break: #1 \__fp_ternary_break_point:n #2 {#2}
11823 \cs_new:Npn \__fp_ternary_auxi:NwwN #1#2@#3@#4
11824 {
11825 \exp_after:wN \__fp_parse_continue:NwN
11826 \exp_after:wN #1
11827 \tex_romannumeral:D -'0
11828 \__fp_exp_after_array_f:w #2 \s__fp_stop
11829 #4 #1
11830 }
11831 \cs_new:Npn \__fp_ternary_auxii:NwwN #1#2@#3@#4
11832 {
11833 \exp_after:wN \__fp_parse_continue:NwN
11834 \exp_after:wN #1
11835 \tex_romannumeral:D -'0
11836 \__fp_exp_after_array_f:w #3 \s__fp_stop
11837 #4 #1
11838 }

```

(End definition for __fp_ternary:NwwN, __fp_ternary_auxi:NwwN, and __fp_ternary_auxii:NwwN.)

```

11839 </initex | package>

```

29 l3fp-basics Implementation

```
11840 <*initex | package>
11841 <@@=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>.

29.1 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}.

```
11842 \cs_new:Npn \__fp_basics_pack_low:NNNNNw #1 #2#3#4#5 #6;
11843 { + #1 - \c_one ; {#2#3#4#5} {#6} ; }
11844 \cs_new:Npn \__fp_basics_pack_high:NNNNNw #1 #2#3#4#5 #6;
11845 {
11846   \if_meaning:w 2 #1
11847     \__fp_basics_pack_high_carry:w
11848   \fi:
11849   ; {#2#3#4#5} {#6}
11850 }
11851 \cs_new:Npn \__fp_basics_pack_high_carry:w \fi: ; #1
11852 { \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.

```
11853 \cs_new:Npn \__fp_basics_pack_weird_low:NNNNw #1 #2#3#4 #5;
11854 {
11855   \if_meaning:w 2 #1
11856     + \c_one
11857   \fi:
11858   \__int_eval_end:
11859   #2#3#4; {#5} ;
11860 }
11861 \cs_new:Npn \__fp_basics_pack_weird_high:NNNNNNNNw
11862 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.)

29.2 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.

29.2.1 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.

```

11863 \cs_new_nopar:cpx { __fp_-_o:ww } \s__fp
11864 {
11865   \exp_not:c { __fp+_o:ww }
11866   \exp_not:n { \s__fp \__fp_neg_sign:N }
11867 }

```

(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.

```

11868 \cs_new:cpn { __fp+_o:ww }
11869   \s__fp #1 \__fp_chk:w #2 #3 ; \s__fp \__fp_chk:w #4 #5
11870   {
11871     \if_case:w
11872       \if_meaning:w #2 #4
11873         #2 \exp_stop_f:
11874     \else:
11875       \if_int_compare:w #2 > #4 \exp_stop_f:
11876       \c_three
11877     \else:
11878       \c_minus_one
11879     \fi:
11880   \fi:
11881     \exp_after:wN \__fp_add_zeros_o:Nww \__int_value:w
11882 \or:   \exp_after:wN \__fp_add_normal_o:Nww \__int_value:w
11883 \or:   \exp_after:wN \__fp_add_inf_o:Nww \__int_value:w
11884 \or:   \__fp_case_return_i_o:ww
11885 \else: \exp_after:wN \__fp_add_return_ii_o:Nww \__int_value:w
11886 \fi:
11887 #1 #5
11888   \s__fp \__fp_chk:w #2 #3 ;
11889   \s__fp \__fp_chk:w #4 #5
11890 }
(End definition for \__fp+_o:ww.)

```

$\backslash_fp_add_return_ii_o:Nww$ Ignore the first operand, and return the second, but using the sign #1 rather than #4. As usual, expand after the floating point.

```

11891 \cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
11892 { \__fp_exp_after_o:w \s__fp \__fp_chk:w #3 #1 }
(End definition for \__fp_add_return_ii_o:Nww.)

```

$\backslash_fp_add_zeros_o:Nww$ Adding two zeros yields $\backslash c_zero_fp$, except if both zeros were -0 .

```

11893 \cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
11894 {
11895   \if_int_compare:w #2 #1 = 20 \exp_stop_f:
11896   \exp_after:wN \__fp_add_return_ii_o:Nww
11897 \else:
11898   \__fp_case_return_i_o:ww
11899 \fi:
11900 #1
11901 \s__fp \__fp_chk:w 0 #2
11902 }
(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.

```

11903 \cs_new:Npn \__fp_add_inf_o:Nww
11904   #1 \s__fp \__fp_chk:w 2 #2 #3; \s__fp \__fp_chk:w 2 #4
11905   {
11906     \if_meaning:w #1 #2
11907       \__fp_case_return_i_o:ww
11908     \else:
11909       \__fp_case_use:nw
11910       {
11911         \if_meaning:w #1 #4
11912           \exp_after:wN \__fp_invalid_operation_o:Nww
11913           \exp_after:wN +
11914         \else:
11915           \exp_after:wN \__fp_invalid_operation_o:Nww
11916           \exp_after:wN -
11917         \fi:
11918       }
11919     \fi:
11920     \s__fp \__fp_chk:w 2 #2 #3;
11921     \s__fp \__fp_chk:w 2 #4
11922   }

```

(End definition for `__fp_add_inf_o:Nww`.)

`__fp_add_normal_o:Nww` $__fp_add_normal_o:Nww \langle sign_2 \rangle \s__fp __fp_chk:w 1 \langle sign_1 \rangle \langle exp_1 \rangle$
 $\langle body_1 \rangle ; \s__fp __fp_chk:w 1 \langle initial\ sign_2 \rangle \langle exp_2 \rangle \langle body_2 \rangle ;$

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```

11923 \cs_new:Npn \__fp_add_normal_o:Nww #1 \s__fp \__fp_chk:w 1 #2
11924   {
11925     \if_meaning:w #1#2
11926       \exp_after:wN \__fp_add_npos_o:NnwNnw
11927     \else:
11928       \exp_after:wN \__fp_sub_npos_o:NnwNnw
11929     \fi:
11930     #2
11931   }

```

(End definition for `__fp_add_normal_o:Nww`.)

29.2.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

`__fp_add_npos_o:NnwNnw` $__fp_add_npos_o:NnwNnw \langle sign_1 \rangle \langle exp_1 \rangle \langle body_1 \rangle ; \s__fp __fp_chk:w 1$
 $\langle initial\ sign_2 \rangle \langle exp_2 \rangle \langle body_2 \rangle ;$

Since we are doing an addition, the final sign is $\langle sign_1 \rangle$. Start an `__int_eval:w`, responsible for computing the exponent: the result, and the $\langle final\ sign \rangle$ are then given to

_fp_sanitize:Nw which checks for overflow. The exponent is computed as the largest exponent #2 or #5, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by _fp_add_big_i:wNww or _fp_add_big_ii:wNww. We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```

11932 \cs_new:Npn \_fp\_add\_npos_o:NnwNnw #1#2#3 ; \s\_fp \_fp\_chk:w 1 #4 #5
11933 {
11934   \exp\_after:wN \_fp\_sanitize:Nw
11935   \exp\_after:wN #1
11936   \int\_use:N \_int\_eval:w
11937   \if\_int\_compare:w #2 > #5 \exp\_stop\_f:
11938     #2
11939     \exp\_after:wN \_fp\_add\_big\_i_o:wNww \_int\_value:w -
11940   \else:
11941     #5
11942     \exp\_after:wN \_fp\_add\_big\_ii_o:wNww \_int\_value:w
11943   \fi:
11944   \_int\_eval:w #5 - #2 ; #1 #3;
11945 }

```

(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.

```

11946 \cs_new:Npn \_fp\_add\_big\_i_o:wNww #1; #2 #3; #4;
11947 {
11948   \_fp\_decimate:nNnnnn {#1}
11949   \_fp\_add\_significand_o:NnnwnnnnN
11950   #4
11951   #3
11952   #2
11953 }
11954 \cs_new:Npn \_fp\_add\_big\_ii_o:wNww #1; #2 #3; #4;
11955 {
11956   \_fp\_decimate:nNnnnn {#1}
11957   \_fp\_add\_significand_o:NnnwnnnnN
11958   #3
11959   #4
11960   #2
11961 }

```

(End definition for _fp_add_big_i_o:wNww.)

_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 $0.99\cdots95 \rightarrow 1.00\cdots0$, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```

11962 \cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
11963 {
11964   \exp_after:wN \__fp_add_significand_test_o:N
11965   \int_use:N \__int_eval:w 1#5#6 + #2
11966   \exp_after:wN \__fp_add_significand_pack:NNNNNNN
11967   \int_use:N \__int_eval:w 1#7#8 + #3 ; #1
11968 }
11969 \cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
11970 {
11971   \if_meaning:w 2 #1
11972     + \c_one
11973   \fi:
11974   ; #2 #3 #4 #5 #6 #7 ;
11975 }
11976 \cs_new:Npn \__fp_add_significand_test_o:N #1
11977 {
11978   \if_meaning:w 2 #1
11979     \exp_after:wN \__fp_add_significand_carry_o:wwwNN
11980   \else:
11981     \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
11982   \fi:
11983 }
(End definition for \__fp_add_significand_o:NnnwnnnnN.)

```

$\backslash_fp_add_significand_no_carry_o:wwwNN$ $\backslash_fp_add_significand_no_carry_o:wwwNN \langle 8d \rangle ; \langle 6d \rangle ; \langle 2d \rangle ; \langle rounding digit \rangle \langle sign \rangle$

If there's no carry, grab all the digits again and round. The packing function $\backslash_fp_basics_pack_high:NNNNNw$ takes care of the case where rounding brings a carry.

```

11984 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
11985 #1; #2; #3#4 ; #5#6
11986 {
11987   \exp_after:wN \__fp_basics_pack_high:NNNNNw
11988   \int_use:N \__int_eval:w 1 #1
11989   \exp_after:wN \__fp_basics_pack_low:NNNNNw
11990   \int_use:N \__int_eval:w 1 #2 #3#4
11991   + \__fp_round:NNN #6 #4 #5
11992   \exp_after:wN ;
11993 }
(End definition for \__fp_add_significand_no_carry_o:wwwNN.)

```

$\backslash_fp_add_significand_carry_o:wwwNN$ $\backslash_fp_add_significand_carry_o:wwwNN \langle 8d \rangle ; \langle 6d \rangle ; \langle 2d \rangle ; \langle rounding digit \rangle \langle sign \rangle$

The case where there is a carry is very similar. Rounding can even raise the first digit from 1 to 2, but we don't care.

```

11994 \cs_new:Npn \__fp_add_significand_carry_o:wwwNN
11995 #1; #2; #3#4; #5#6

```

```

11996 {
11997   + \c_one
11998   \exp_after:wN \_fp_basics_pack_weird_high:NNNNNNNNw
11999   \int_use:N \_int_eval:w 1 1 #1
12000   \exp_after:wN \_fp_basics_pack_weird_low:NNNNw
12001   \int_use:N \_int_eval:w 1 #2#3 +
12002   \exp_after:wN \_fp_round:NNN
12003   \exp_after:wN #6
12004   \exp_after:wN #3
12005   \_int_value:w \_fp_round_digit:Nw #4 #5 ;
12006   \exp_after:wN ;
12007 }
(End definition for \_fp_add_significand_carry_o:wwwNN.)

```

29.2.3 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 $\langle sign_1 \rangle$.

```

12008 \cs_new:Npn \_fp_sub_npos_o:NnwNnw #1#2#3; \s\_fp \_fp_chk:w 1 #4#5#6;
12009 {
12010   \if_case:w \_fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
12011   \exp_after:wN \_fp_sub_eq_o:Nnwnw
12012   \or:
12013   \exp_after:wN \_fp_sub_npos_i_o:Nnwnw
12014   \else:
12015   \exp_after:wN \_fp_sub_npos_ii_o:Nnwnw
12016   \fi:
12017   #1 {#2} #3; {#5} #6;
12018 }
12019 \cs_new:Npn \_fp_sub_eq_o:Nnwnw #1#2; #3; { \exp_after:wN \c_zero_fp }
12020 \cs_new:Npn \_fp_sub_npos_ii_o:Nnwnw #1 #2; #3;
12021 {
12022   \exp_after:wN \_fp_sub_npos_i_o:Nnwnw
12023   \int_use:N \_int_eval:w \c_two - #1 \_int_eval_end:
12024   #3; #2;
12025 }
(End definition for \_fp_sub_npos_o:NnwNnw.)

```

`_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.

```

12026 \cs_new:Npn \__fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
12027 {
12028   \exp_after:wN \__fp_sanitize:Nw
12029   \exp_after:wN #1
12030   \int_use:N \__int_eval:w
12031   #2
12032   \if_int_compare:w #2 = #4 \exp_stop_f:
12033     \exp_after:wN \__fp_sub_back_near_o:nnnnnnnnN
12034   \else:
12035     \exp_after:wN \__fp_decimate:nNnnnn \exp_after:wN
12036     { \int_use:N \__int_eval:w #2 - #4 - \c_one \exp_after:wN }
12037     \exp_after:wN \__fp_sub_back_far_o:NnnwnnnnnN
12038   \fi:
12039   #5
12040   #3
12041   #1
12042 }

```

(End definition for __fp_sub_npos_i_o:Nnwnw.)

```

\__fp_sub_back_near_o:nnnnnnnnN \__fp_sub_back_near_o:nnnnnnnnN {\langle Y_1 \rangle} {\langle Y_2 \rangle} {\langle Y_3 \rangle} {\langle Y_4 \rangle} {\langle X_1 \rangle}
\__fp_sub_back_near_pack:NNNNNNw {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} \langle final sign \rangle
\__fp_sub_back_near_after:wNNNNw

```

In this case, the subtraction is exact, so we discard the *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.

```

12043 \cs_new:Npn \__fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
12044 {
12045   \exp_after:wN \__fp_sub_back_near_after:wNNNNw
12046   \int_use:N \__int_eval:w 10#5#6 - #1#2 - \c_eleven
12047   \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
12048   \int_use:N \__int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
12049 }
12050 \cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
12051 { + #1#2 ; {#3#4#5#6} {#7} ; }
12052 \cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
12053 {
12054   \if_meaning:w 0 #1
12055     \exp_after:wN \__fp_sub_back_shift:wnnnn
12056   \fi:
12057   ; {#1#2#3#4} {#5}
12058 }

```

(End definition for __fp_sub_back_near_o:nnnnnnnnN.)

```

\__fp_sub_back_shift:wnnnn \__fp_sub_back_shift:wnnnn ; {\langle Z_1 \rangle} {\langle Z_2 \rangle} {\langle Z_3 \rangle} {\langle Z_4 \rangle} ;
\__fp_sub_back_shift_ii:ww This function is called with  $\langle Z_1 \rangle \leq 999$ . Act with \number to trim leading zeros from
\__fp_sub_back_shift_iii:NNNNNNNNw \langle Z_1 \rangle \langle Z_2 \rangle (we don't do all four blocks at once, since non-zero blocks would then overflow
\__fp_sub_back_shift_iv:nnnnw

```

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.

```

12059 \cs_new:Npn \__fp_sub_back_shift:wnnnn ; #1#2
12060 {
12061   \exp_after:wN \__fp_sub_back_shift_ii:ww
12062   \__int_value:w #1 #2 0 ;
12063 }
12064 \cs_new:Npn \__fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
12065 {
12066   \if_meaning:w @ #1 @
12067   - \c_seven
12068   - \exp_after:wN \use_i:nnn
12069   \exp_after:wN \__fp_sub_back_shift_iii:NNNNNNNNw
12070   \__int_value:w #2#3 0 ~ 123456789;
12071   \else:
12072   - \__fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
12073   \fi:
12074   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
12075   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
12076   \exp_after:wN \__fp_sub_back_shift_iv:nnnnw
12077   \exp_after:wN ;
12078   \__int_value:w
12079   #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
12080 }
12081 \cs_new:Npn \__fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
12082 \cs_new:Npn \__fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }
(End definition for \__fp_sub_back_shift:wnnnn.)

```

$\backslash_fp_sub_back_far_o:NnnwnnnnN$ $\backslash_fp_sub_back_far_o:NnnwnnnnN \langle rounding \rangle \{ \langle Y'_1 \rangle \} \{ \langle Y'_2 \rangle \} \langle extra-digits \rangle$
 $; \{ \langle X_1 \rangle \} \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \} \langle final sign \rangle$

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`).

```

12083 \cs_new:Npn \__fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
12084 {
12085   \if_case:w
12086   \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
12087   \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
12088   \c_zero
12089   \else:
12090   \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: \c_one

```

```

12091         \fi:
12092     \else:
12093         \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: \c_one
12094     \fi:
12095     \exp_after:wN \__fp_sub_back_quite_far_o:wwNN
12096 \or:   \exp_after:wN \__fp_sub_back_very_far_o:wwwNN
12097 \else: \exp_after:wN \__fp_sub_back_not_far_o:wwwNN
12098 \fi:
12099     #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
12100 }
(End definition for \__fp_sub_back_far_o:NnnwnnnN.)

```

$\backslash_fp_sub_back_quite_far_o:wwNN$
 $\backslash_fp_sub_back_quite_far_ii:NN$

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).

```

12101 \cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
12102 {
12103     \exp_after:wN \__fp_sub_back_quite_far_ii:NN
12104     \exp_after:wN #3
12105     \exp_after:wN #4
12106 }
12107 \cs_new:Npn \__fp_sub_back_quite_far_ii:NN #1#2
12108 {
12109     \if_case:w \__fp_round_neg:NNN #2 0 #1
12110     \exp_after:wN \use_i:nn
12111 \else:
12112     \exp_after:wN \use_ii:nn
12113 \fi:
12114     { ; {1000} {0000} {0000} {0000} ; }
12115     { - \c_one ; {9999} {9999} {9999} {9999} ; }
12116 }
(End definition for \__fp_sub_back_quite_far_o:wwNN.)

```

$\backslash_fp_sub_back_not_far_o:wwwNN$

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 $-\backslash c_one$). 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 $\backslash_fp_round_neg:NNN$ 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 $\backslash_fp_round_neg:NNN$ only cares about its parity, which is identical to that of the last digit of #2.

```

12117 \cs_new:Npn \__fp_sub_back_not_far_o:wwwNN #1 ~ #2; #3 ~ #4; #5#6
12118 {
12119     - \c_one
12120     \exp_after:wN \__fp_sub_back_near_after:wNNNNw

```

```

12121 \int_use:N \__int_eval:w 1#30 - #1 - \c_eleven
12122 \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
12123 \int_use:N \__int_eval:w 11 0000 0000 + #40 - #2
12124 - \exp_after:wN \__fp_round_neg:NNN
12125 \exp_after:wN #6
12126 \use_none:nnnnnn #2 #5
12127 \exp_after:wN ;
12128 }

```

(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.

```

12129 \cs_new:Npn \__fp_sub_back_very_far_o:wwwNN #1#2#3#4#5#6#7
12130 {
12131   \__fp_pack_eight:wNNNNNNNN
12132   \__fp_sub_back_very_far_ii_o:nnNwwNN
12133   { 0 #1#2#3 #4#5#6#7 }
12134   ;
12135 }
12136 \cs_new:Npn \__fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
12137 {
12138   \exp_after:wN \__fp_basics_pack_high:NNNNNw
12139   \int_use:N \__int_eval:w 1#4 - #1 - \c_one
12140   \exp_after:wN \__fp_basics_pack_low:NNNNNw
12141   \int_use:N \__int_eval:w 2#5 - #2
12142   - \exp_after:wN \__fp_round_neg:NNN
12143   \exp_after:wN #7
12144   \__int_value:w
12145   \if_int_odd:w \__int_eval:w #5 - #2 \__int_eval_end:
12146     1 \else: 2 \fi:
12147   \__int_value:w \__fp_round_digit:Nw #3 #6 ;
12148   \exp_after:wN ;
12149 }

```

(End definition for __fp_sub_back_very_far_o:wwwNN.)

29.3 Multiplication

29.3.1 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`.

```

12150 \cs_new_nopar:cpn { \_fp\_/_o:ww }
12151 {
12152   \_fp_mul_cases_o:NnNnw
12153   *
12154   { - \c_two + }
12155   \_fp_mul_npos_o:Nww
12156   { }
12157 }

```

(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.

```

12158 \cs_new:Npn \_fp_mul_cases_o:NnNnw
12159   #1#2#3#4 \s_fp \_fp_chk:w #5#6#7; \s_fp \_fp_chk:w #8#9
12160 {
12161   \if_case:w \_int_eval:w
12162     \if_int_compare:w #5 #8 = \c_eleven
12163     \c_one
12164   \else:
12165     \if_meaning:w 3 #8
12166     \c_three
12167   \else:
12168     \if_meaning:w 3 #5
12169     \c_two
12170   \else:
12171     \if_int_compare:w #5 #8 = \c_ten
12172     \c_nine #2 - \c_two
12173   \else:
12174     (#5 #2 #8) / \c_two * \c_two + \c_seven
12175   \fi:
12176   \fi:
12177   \fi:
12178   \if_meaning:w #6 #9 - \c_one \fi:
12179   \_int_eval_end:
12180   \_fp_case_use:nw { #3 0 }
12181   \or: \_fp_case_use:nw { #3 2 }
12182 }

```



```

12183 \or: \_fp_case_return_i_o:ww
12184 \or: \_fp_case_return_ii_o:ww
12185 \or: \_fp_case_return_o:Nww \c_zero_fp
12186 \or: \_fp_case_return_o:Nww \c_minus_zero_fp
12187 \or: \_fp_case_use:nw { \_fp_invalid_operation_o:Nww #1 }
12188 \or: \_fp_case_use:nw { \_fp_invalid_operation_o:Nww #1 }
12189 \or: \_fp_case_return_o:Nww \c_inf_fp
12190 \or: \_fp_case_return_o:Nww \c_minus_inf_fp
12191 #4
12192 \fi:
12193 \s__fp \_fp_chk:w #5 #6 #7;
12194 \s__fp \_fp_chk:w #8 #9
12195 }
(End definition for \_fp_mul_cases_o:nNnnww.)

```

29.3.2 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`.

```

12196 \cs_new:Npn \_fp_mul_npos_o:Nww
12197 #1 \s__fp \_fp_chk:w #2 #3 #4 #5 ; \s__fp \_fp_chk:w #6 #7 #8 #9 ;
12198 {
12199 \exp_after:wN \_fp_sanitize:Nw
12200 \exp_after:wN #1
12201 \int_use:N \_int_eval:w
12202 #4 + #8
12203 \_fp_mul_significand_o:nnnnNnnnn #5 #1 #9
12204 }
(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`.

```

12205 \cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
12206 {
12207   \exp_after:wN \__fp_mul_significand_test_f:NNN
12208   \exp_after:wN #5
12209   \int_use:N \__int_eval:w 99990000 + #1*#6 +
12210   \exp_after:wN \__fp_mul_significand_keep:NNNNNw
12211   \int_use:N \__int_eval:w 99990000 + #1*#7 + #2*#6 +
12212   \exp_after:wN \__fp_mul_significand_keep:NNNNNw
12213   \int_use:N \__int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
12214   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
12215   \int_use:N \__int_eval:w 99990000 + #1*#9 + #2*#8 + #3*#7 + #4*#6 +
12216   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
12217   \int_use:N \__int_eval:w 99990000 + #2*#9 + #3*#8 + #4*#7 +
12218   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
12219   \int_use:N \__int_eval:w 99990000 + #3*#9 + #4*#8 +
12220   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
12221   \int_use:N \__int_eval:w 100000000 + #4*#9 ;
12222   ; \exp_after:wN ;
12223 }
12224 \cs_new:Npn \__fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
12225 { #1#2#3#4#5 ; + #6 }
12226 \cs_new:Npn \__fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
12227 { #1#2#3#4#5 ; #6 ; }
(End definition for \__fp_mul_significand_o:nnnnNnnnn.)

```

```

\__fp_mul_significand_test_f:NNN \__fp_mul_significand_test_f:NNN <sign> 1 <digits 1–8> ; <digits 9–12> ;
<digits 13–16> ; + <digits 17–20> + <digits 21–24> + <digits 25–28> + <digits
29–32> ; \exp_after:wN ;

```

If the *<digit 1>* is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if *<digit 1>* is zero, we care about digits 17 and 18, and whether further digits are zero.

```

12228 \cs_new:Npn \__fp_mul_significand_test_f:NNN #1 #2 #3
12229 {
12230   \if_meaning:w 0 #3
12231   \exp_after:wN \__fp_mul_significand_small_f:NNwwwN
12232   \else:
12233   \exp_after:wN \__fp_mul_significand_large_f:NwwNNNN
12234   \fi:
12235   #1 #3
12236 }
(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`.

```

12237 \cs_new:Npn \__fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
12238 {
12239   \exp_after:wN \__fp_basics_pack_high:NNNNw
12240   \int_use:N \__int_eval:w 1#2
12241   \exp_after:wN \__fp_basics_pack_low:NNNNw
12242   \int_use:N \__int_eval:w 1#3#4#5#6#7
12243   + \exp_after:wN \__fp_round:NNN
12244   \exp_after:wN #1
12245   \exp_after:wN #7
12246   \__int_value:w \__fp_round_digit:Nw
12247 }
(End definition for \__fp_mul_significand_large_f:NwwNNNN.)

```

`__fp_mul_significand_small_f:NNwwN` In this branch, $\langle \text{digit } 1 \rangle$ is zero. Our result will thus be $\langle \text{digits } 2\text{--}17 \rangle$, 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.

```

12248 \cs_new:Npn \__fp_mul_significand_small_f:NNwwN #1 #2#3; #4#5; #6; + #7
12249 {
12250   - \c_one
12251   \exp_after:wN \__fp_basics_pack_high:NNNNw
12252   \int_use:N \__int_eval:w 1#3#4
12253   \exp_after:wN \__fp_basics_pack_low:NNNNw
12254   \int_use:N \__int_eval:w 1#5#6#7
12255   + \exp_after:wN \__fp_round:NNN
12256   \exp_after:wN #1
12257   \exp_after:wN #7
12258   \__int_value:w \__fp_round_digit:Nw
12259 }
(End definition for \__fp_mul_significand_small_f:NNwwN.)

```

29.4 Division

29.4.1 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.

```

12260 \cs_new_nopar:cpn { __fp/_o:ww }
12261 {
12262   \__fp_mul_cases_o:NnNww
12263   /

```

```

12264 { - }
12265 \__fp_div_npos_o:Nww
12266 {
12267   \or:
12268     \__fp_case_use:nw
12269     { \__fp_division_by_zero_o:NNww \c_inf_fp / }
12270   \or:
12271     \__fp_case_use:nw
12272     { \__fp_division_by_zero_o:NNww \c_minus_inf_fp / }
12273 }
12274 }

```

(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.

```

12275 \cs_new:Npn \__fp_div_npos_o:Nww
12276   #1 \s__fp \__fp_chk:w 1 #2 #3 #4 ; \s__fp \__fp_chk:w 1 #5 #6 #7#8#9;
12277   {
12278     \exp_after:wN \__fp_sanitize:Nw
12279     \exp_after:wN #1
12280     \int_use:N \__int_eval:w
12281     #3 - #6
12282     \exp_after:wN \__fp_div_significand_i_o:wnnw
12283     \int_use:N \__int_eval:w #7 \use_i:n #8 + \c_one ;
12284     #4
12285     {#7}{#8}#9 ;
12286     #1
12287   }

```

(End definition for __fp_div_npos_o:Nww.)

29.4.2 Work plan

In this subsection, we explain how to avoid overflowing $\text{T}_{\text{E}}\text{X}$'s integers when performing the division of two positive normal numbers.

We are given two numbers, $A = 0.A_1A_2A_3A_4$ and $Z = 0.Z_1Z_2Z_3Z_4$, in blocks of 4 digits, and we know that the first digits of A_1 and of Z_1 are non-zero. To compute A/Z , we proceed as follows.

- Find an integer $Q_A \simeq 10^4 A/Z$.
- Replace A by $B = 10^4 A - Q_A Z$.

- Find an integer $Q_B \simeq 10^4 B/Z$.
- Replace B by $C = 10^4 B - Q_B Z$.
- Find an integer $Q_C \simeq 10^4 C/Z$.
- Replace C by $D = 10^4 C - Q_C Z$.
- Find an integer $Q_D \simeq 10^4 D/Z$.
- Consider $E = 10^4 D - Q_D Z$, and ensure correct rounding.

The result is then $Q = 10^{-4}Q_A + 10^{-8}Q_B + 10^{-12}Q_C + 10^{-16}Q_D + \text{rounding}$. Since the Q_i are integers, B , C , D , and E are all exact multiples of 10^{-16} , in other words, computing with 16 digits after the decimal separator yields exact results. The problem will be overflow: in general B , C , D , and E may be greater than 1.

Unfortunately, things are not as easy as they seem. In particular, we want all intermediate steps to be positive, since negative results would require extra calculations at the end. This requires that $Q_A \leq 10^4 A/Z$ etc. A reasonable attempt would be to define Q_A as

$$\backslash\text{int_eval:n}\left\{\frac{A_1 A_2}{Z_1 + 1} - 1\right\} \leq 10^4 \frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that $\varepsilon\text{-TeX}$'s $\backslash_int_eval:w$ rounds divisions instead of truncating (really, $1/2$ would be sufficient, but we work with integers). We add 1 to Z_1 because $Z_1 \leq 10^4 Z < Z_1 + 1$ and we need Q_A to be an underestimate. However, we are now underestimating Q_A too much: it can be wrong by up to 100, for instance when $Z = 0.1$ and $A \simeq 1$. Then B could take values up to 10 (maybe more), and a few steps down the line, we would run into arithmetic overflow, since TeX can only handle integers less than roughly $2 \cdot 10^9$.

A better formula is to take

$$Q_A = \backslash\text{int_eval:n}\left\{\frac{10 \cdot A_1 A_2}{\lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1} - 1\right\}.$$

This is always less than $10^9 A/(10^5 Z)$, as we wanted. In words, we take the 5 first digits of Z into account, and the 8 first digits of A , using 0 as a 9-th digit rather than the true digit for efficiency reasons. We shall prove that using this formula to define all the Q_i avoids any overflow. For convenience, let us denote

$$y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1,$$

so that, taking into account the fact that $\varepsilon\text{-TeX}$ rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1 A_2 0}{y} - \frac{3}{2}. \end{aligned}$$

Note that $10^4 < y \leq 10^5$, and $999 \leq Q_A \leq 99989$. Also note that this formula does not cause an overflow as long as $A < (2^{31} - 1)/10^9 \simeq 2.147 \dots$, since the numerator involves an integer slightly smaller than $10^9 A$.

Let us bound B :

$$\begin{aligned}
10^5 B &= A_1 A_2 0 + 10 \cdot 0 \cdot A_3 A_4 - 10 \cdot Z_1 \cdot Z_2 Z_3 Z_4 \cdot Q_A \\
&< A_1 A_2 0 \cdot \left(1 - 10 \cdot \frac{Z_1 \cdot Z_2 Z_3 Z_4}{y}\right) + \frac{3}{2} \cdot 10 \cdot Z_1 \cdot Z_2 Z_3 Z_4 + 10 \\
&\leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1 \cdot Z_2 Z_3 Z_4)}{y} + \frac{3}{2} y + 10 \\
&\leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2} y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y.
\end{aligned}$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$\begin{aligned}
10^5 B &< 10^9 A/y + 1.6y, \\
10^5 C &< 10^9 B/y + 1.6y, \\
10^5 D &< 10^9 C/y + 1.6y, \\
10^5 E &< 10^9 D/y + 1.6y.
\end{aligned}$$

The goal is now to prove that none of B , C , D , and E can go beyond $(2^{31} - 1)/10^9 = 2.147 \dots$.

Combining the various inequalities together with $A < 1$, we get

$$\begin{aligned}
10^5 B &< 10^9/y + 1.6y, \\
10^5 C &< 10^{13}/y^2 + 1.6(y + 10^4), \\
10^5 D &< 10^{17}/y^3 + 1.6(y + 10^4 + 10^8/y), \\
10^5 E &< 10^{21}/y^4 + 1.6(y + 10^4 + 10^8/y + 10^{12}/y^2).
\end{aligned}$$

All of those bounds are convex functions of y (since every power of y involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range $10^4 < y \leq 10^5$. Thus,

$$\begin{aligned}
10^5 B &< \max(1.16 \cdot 10^5, 1.7 \cdot 10^5), \\
10^5 C &< \max(1.32 \cdot 10^5, 1.77 \cdot 10^5), \\
10^5 D &< \max(1.48 \cdot 10^5, 1.777 \cdot 10^5), \\
10^5 E &< \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5).
\end{aligned}$$

All of those bounds are less than $2.147 \cdot 10^5$, and we are thus within $\text{T}_{\text{E}}\text{X}$'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

$$P = \text{\texttt{_int_eval:n}} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

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.

29.4.3 Implementing the significand division

```
\_fp_div_significand_i_o:wnnw <y> ; {\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} ; <sign>
```

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace $\langle A_1 \rangle$ and $\langle A_2 \rangle$, and prepare the $\langle continuation \rangle$ 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.

```
12288 \cs_new:Npn \_fp_div_significand_i_o:wnnw #1 ; #2#3 #4 ;
12289 {
12290   \exp_after:wN \_fp_div_significand_test_o:w
12291   \int_use:N \_int_eval:w
12292   \exp_after:wN \_fp_div_significand_calc:wnnnnnnnn
12293   \int_use:N \_int_eval:w 999999 + #2 #3 0 / #1 ;
```

```

12294      #2 #3 ;
12295      #4
12296      { \exp_after:wN \__fp_div_significand_ii:wwn \__int_value:w #1 }
12297      { \exp_after:wN \__fp_div_significand_ii:wwn \__int_value:w #1 }
12298      { \exp_after:wN \__fp_div_significand_ii:wwn \__int_value:w #1 }
12299      { \exp_after:wN \__fp_div_significand_iii:wwnnnnn \__int_value:w #1 }
12300    }

```

(End definition for __fp_div_significand_i_o:wnnw.)

```

\__fp_div_significand_calc:wwnnnnnnn \__fp_div_significand_calc:wwnnnnnnn  $\langle 10^6 + Q_A \rangle$  ;  $\langle A_1 \rangle \langle A_2 \rangle$  ;  $\{\langle A_3 \rangle\}$ 
\__fp_div_significand_calc_i:wwnnnnnnn  $\{\langle A_4 \rangle\} \{\langle Z_1 \rangle\} \{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\} \{\langle Z_4 \rangle\} \{\langle continuation \rangle\}$ 
\__fp_div_significand_calc_ii:wwnnnnnnn 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 worst $-8 \cdot 10^8 4$, while the other negative term is very small $< 10^6$ (except in the first expression, where we don't care about the number of digits); for the auxiliary with $\langle i \rangle = 0$, $\#1$ can go up to 99999, but there is no other negative term. Hence, a good choice is $2 \cdot 10^9$, which produces totals in the range $[10^9, 2.1 \cdot 10^9]$. We are flirting with $\text{T}_{\text{E}}\text{X}$'s limits once more.

```

12301 \cs_new:Npn \__fp_div_significand_calc:wwnnnnnnn 1#1
12302 {
12303   \if_meaning:w 1 #1
12304     \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnn
12305   \else:

```



```

12306      \exp_after:wN \_fp_div_significand_calc_ii:wwnnnnnnn
12307      \fi:
12308    }
12309    \cs_new:Npn \_fp_div_significand_calc_i:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
12310    {
12311      1 1 #1
12312      #9 \exp_after:wN ;
12313      \int_use:N \_int_eval:w \c__fp_Bigg_leading_shift_int
12314      + #2 - #1 * #5 - #5#60
12315      \exp_after:wN \_fp_pack_Bigg:NNNNNNw
12316      \int_use:N \_int_eval:w \c__fp_Bigg_middle_shift_int
12317      + #3 - #1 * #6 - #70
12318      \exp_after:wN \_fp_pack_Bigg:NNNNNNw
12319      \int_use:N \_int_eval:w \c__fp_Bigg_middle_shift_int
12320      + #4 - #1 * #7 - #80
12321      \exp_after:wN \_fp_pack_Bigg:NNNNNNw
12322      \int_use:N \_int_eval:w \c__fp_Bigg_trailing_shift_int
12323      - #1 * #8 ;
12324      {#5}{#6}{#7}{#8}
12325    }
12326    \cs_new:Npn \_fp_div_significand_calc_ii:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
12327    {
12328      1 0 #1
12329      #9 \exp_after:wN ;
12330      \int_use:N \_int_eval:w \c__fp_Bigg_leading_shift_int
12331      + #2 - #1 * #5
12332      \exp_after:wN \_fp_pack_Bigg:NNNNNNw
12333      \int_use:N \_int_eval:w \c__fp_Bigg_middle_shift_int
12334      + #3 - #1 * #6
12335      \exp_after:wN \_fp_pack_Bigg:NNNNNNw
12336      \int_use:N \_int_eval:w \c__fp_Bigg_middle_shift_int
12337      + #4 - #1 * #7
12338      \exp_after:wN \_fp_pack_Bigg:NNNNNNw
12339      \int_use:N \_int_eval:w \c__fp_Bigg_trailing_shift_int
12340      - #1 * #8 ;
12341      {#5}{#6}{#7}{#8}
12342    }

```

(End definition for _fp_div_significand_calc:wwnnnnnnn.)

```

\_fp_div_significand_ii:wwn      \_fp_div_significand_ii:wwn  $\langle y \rangle$  ;  $\langle B_1 \rangle$  ;  $\{\langle B_2 \rangle\}$   $\{\langle B_3 \rangle\}$   $\{\langle B_4 \rangle\}$   $\{\langle Z_1 \rangle\}$ 
                                 $\{\langle Z_2 \rangle\}$   $\{\langle Z_3 \rangle\}$   $\{\langle Z_4 \rangle\}$   $\langle continuations \rangle$   $\langle sign \rangle$ 

```

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 `_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 .

```

12343 \cs_new:Npn \_fp_div_significand_ii:wwn #1; #2;#3
12344 {
12345   \exp_after:wN \_fp_div_significand_pack:NNN

```

```

12346      \int_use:N \__int_eval:w
12347      \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
12348      \int_use:N \__int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
12349    }
(End definition for \__fp_div_significand_ii:wnn.)

```

```

\__fp_div_significand_iii:wwnnnnn  \__fp_div_significand_iii:wwnnnnn <y> ; <E1> ; {<E2>} {<E3>} {<E4>}
{<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>

```

We compute $P \simeq 2E/Z$ by rounding $2E_1E_2/Z_1Z_2$. Note the first 0, which multiplies Q_D by 10: we 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 .

```

12350 \cs_new:Npn \__fp_div_significand_iii:wwnnnnn #1; #2;#3#4#5 #6#7
12351 {
12352   0
12353   \exp_after:wN \__fp_div_significand_iv:wwnnnnnnn
12354   \int_use:N \__int_eval:w (\c_two * #2 #3) / #6 #7 ; % <- P
12355   #2 ; {#3} {#4} {#5}
12356   {#6} {#7}
12357 }
(End definition for \__fp_div_significand_iii:wwnnnnn.)

```

```

\__fp_div_significand_iv:wwnnnnnnn  \__fp_div_significand_iv:wwnnnnnnn <P> ; <E1> ; {<E2>} {<E3>} {<E4>}
\__fp_div_significand_v:NNw      {<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>
\__fp_div_significand_vi:Nw

```

This adds to the current expression ($10^7 + 10 \cdot Q_D$) a contribution of $5 \cdot P + \text{sign}(T)$ with $T = 2E - PZ$. This amounts to adding $P/2$ to Q_D , with an extra *rounding* digit. This *rounding* digit is 0 or 5 if T does not contribute, *i.e.*, if $0 = T = 2E - PZ$, in other words if $10^{16}A/Z$ is an integer or half-integer. Otherwise it is in the appropriate range, $[1, 4]$ or $[6, 9]$. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute T exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation $\#1 \cdot \#6\#7$ below does not cause an overflow: naively, P can be up to 35, and $\#6\#7$ up to 10^8 , but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For $P < 10$, the product obeys $P \cdot \#6\#7 < 10^8 \cdot P < 10^9$.
- For large $P \geq 3$, the rounding error on P , which is at most 1, is less than a factor of 2, hence $P \leq 4E/Z$. Also, $\#6\#7 \leq 10^8 \cdot Z$, hence $P \cdot \#6\#7 \leq 4E \cdot 10^8 < 10^9$.

Both inequalities could be made tighter if needed.

Note however that $P \cdot \#8\#9$ may overflow, since the two factors are now independent, and the result may reach $3.5 \cdot 10^9$. Thus we compute the two lower levels separately. The rest is standard, except that we use $+$ as a separator (ending integer expressions explicitly). T is negative if the first character is $-$, it is positive if the first character is neither 0 nor $-$. It is also positive if the first character is 0 and second argument of $__fp_div_significand_vi:Nw$, a sum of several terms, is also zero. Otherwise, there was an exact agreement: $T = 0$.

```

12358 \cs_new:Npn \__fp_div_significand_iv:wwnnnnnnn #1; #2;#3#4#5 #6#7#8#9
12359 {
12360   + \c_five * #1
12361   \exp_after:wN \__fp_div_significand_vi:Nw
12362   \int_use:N \__int_eval:w -20 + 2*#2#3 - #1*#6#7 +
12363   \exp_after:wN \__fp_div_significand_v:NN
12364   \int_use:N \__int_eval:w 199980 + 2*#4 - #1*#8 +
12365   \exp_after:wN \__fp_div_significand_v:NN
12366   \int_use:N \__int_eval:w 200000 + 2*#5 - #1*#9 ;
12367 }
12368 \cs_new:Npn \__fp_div_significand_v:NN #1#2 { #1#2 \__int_eval_end: + }
12369 \cs_new:Npn \__fp_div_significand_vi:Nw #1#2;
12370 {
12371   \if_meaning:w 0 #1
12372   \if_int_compare:w \__int_eval:w #2 > \c_zero + \c_one \fi:
12373   \else:
12374   \if_meaning:w - #1 - \else: + \fi: \c_one
12375   \fi:
12376   ;
12377 }

```

(End definition for $\backslash_fp_div_significand_iv:wwnnnnnnn$, $\backslash_fp_div_significand_v:NNw$, and $\backslash_fp_div_significand_vi:$

$\backslash_fp_div_significand_pack:NNN$ 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.

$$\backslash_fp_div_significand_test_o:w 10^6 + Q_A \backslash_fp_div_significand_pack:NNN 10^6 + Q_B \backslash_fp_div_significand_pack:NNN 10^6 + Q_C \backslash_fp_div_significand_pack:NNN 10^7 + 10 \cdot Q_D + 5 \cdot P + \varepsilon ; \langle sign \rangle$$

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).

```

12378 \cs_new:Npn \__fp_div_significand_pack:NNN 1 #1 #2 { + #1 #2 ; }

```

(End definition for $\backslash_fp_div_significand_pack:NNN$.)

$\backslash_fp_div_significand_test_o:w$ $\backslash_fp_div_significand_test_o:w 1 0 \langle 5d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 5d \rangle ; \langle sign \rangle$

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.

```

12379 \cs_new:Npn \__fp_div_significand_test_o:w 10 #1
12380 {
12381   \if_meaning:w 0 #1
12382   \exp_after:wN \__fp_div_significand_small_o:wwwNNNNwN
12383   \else:
12384   \exp_after:wN \__fp_div_significand_large_o:wwwNNNNwN
12385   \fi:
12386   #1
12387 }

```

(End definition for `_fp_div_significand_test_o:w`.)

```
\_fp_div_significand_small_o:wwwNNNNwN \_fp_div_significand_small_o:wwwNNNNwN 0 <4d> ; <4d> ; <4d> ; <5d>
; <final sign>
```

Standard use of `_fp_basics_pack_low:NNNNw` and `_fp_basics_pack_high:NNNNw`. We finally get to use the *<final sign>* which has been sitting there for a while.

```
12388 \cs_new:Npn \_fp_div_significand_small_o:wwwNNNNwN
12389 0 #1; #2; #3; #4#5#6#7#8; #9
12390 {
12391   \exp_after:wN \_fp_basics_pack_high:NNNNw
12392   \int_use:N \_int_eval:w 1 #1#2
12393   \exp_after:wN \_fp_basics_pack_low:NNNNw
12394   \int_use:N \_int_eval:w 1 #3#4#5#6#7
12395   + \_fp_round:NNN #9 #7 #8
12396   \exp_after:wN ;
12397 }
```

(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.

```
12398 \cs_new:Npn \_fp_div_significand_large_o:wwwNNNNwN
12399 #1; #2; #3; #4#5#6#7#8; #9
12400 {
12401   + \c_one
12402   \exp_after:wN \_fp_basics_pack_weird_high:NNNNNNwN
12403   \int_use:N \_int_eval:w 1 #1 #2
12404   \exp_after:wN \_fp_basics_pack_weird_low:NNNNw
12405   \int_use:N \_int_eval:w 1 #3 #4 #5 #6 +
12406   \exp_after:wN \_fp_round:NNN
12407   \exp_after:wN #9
12408   \exp_after:wN #6
12409   \_int_value:w \_fp_round_digit:Nw #7 #8 ;
12410   \exp_after:wN ;
12411 }
```

(End definition for `_fp_div_significand_large_o:wwwNNNNwN`.)

29.5 Square root

`_fp_sqrt_o:w` Zeros are unchanged: $\sqrt{-0} = -0$ and $\sqrt{+0} = +0$. Negative numbers (other than -0) have no real square root. Positive infinity, and `nan`, are unchanged. Finally, for normal positive numbers, there is some work to do.

```
12412 \cs_new:Npn \_fp_sqrt_o:w #1 \s__fp \_fp_chk:w #2#3#4; @
12413 {
12414   \if_meaning:w 0 #2 \_fp_case_return_same_o:w \fi:
12415   \if_meaning:w 2 #3
```

```

12416     \__fp_case_use:nw { \__fp_invalid_operation_o:nw { sqrt } }
12417     \fi:
12418     \if_meaning:w 1 #2 \else: \__fp_case_return_same_o:w \fi:
12419     \__fp_sqrt_npos_o:w
12420     \s__fp \__fp_chk:w #2 #3 #4;
12421 }

```

(End definition for __fp_sqrt_o:w.)

```

\__fp_sqrt_npos_o:w
\__fp_sqrt_npos_auxi_o:wNnnN
\__fp_sqrt_npos_auxii_o:wNnnNnnN

```

Prepare __fp_sanitize:Nw to receive the final sign 0 (the result is always positive) and the exponent, equal to half of the exponent #1 of the argument. If the exponent #1 is even, find a first approximation of the square root of the significand $10^8 a_1 + a_2 = 10^8 \#2\#3 + \#4\#5$ through Newton's method, starting at $x = 57234133 \simeq 10^{7.75}$. Otherwise, first shift the significand of of the argument by one digit, getting $a'_1 \in [10^6, 10^7)$ instead of $[10^7, 10^8)$, then use Newton's method starting at $17782794 \simeq 10^{7.25}$.

```

12422 \cs_new:Npn \__fp_sqrt_npos_o:w \s__fp \__fp_chk:w 1 0 #1#2#3#4#5;
12423 {
12424   \exp_after:wN \__fp_sanitize:Nw
12425   \exp_after:wN 0
12426   \int_use:N \__int_eval:w
12427   \if_int_odd:w #1 \exp_stop_f:
12428     \exp_after:wN \__fp_sqrt_npos_auxi_o:wNnnN
12429     \fi:
12430     #1 / \c_two
12431     \__fp_sqrt_Newton_o:wN 56234133; 0; {#2#3} {#4#5} 0
12432 }
12433 \cs_new:Npn \__fp_sqrt_npos_auxi_o:wNnnN #1 / \c_two #2; 0; #3#4#5
12434 {
12435   ( #1 + \c_one ) / \c_two
12436   \__fp_pack_eight:wNnnNnnN
12437   \__fp_sqrt_npos_auxii_o:wNnnNnnN
12438   ;
12439   0 #3 #4
12440 }
12441 \cs_new:Npn \__fp_sqrt_npos_auxii_o:wNnnNnnN #1; #2#3#4#5#6#7#8#9
12442 { \__fp_sqrt_Newton_o:wN 17782794; 0; {#1} {#2#3#4#5#6#7#8#9} }

```

(End definition for __fp_sqrt_npos_o:w.)

```

\__fp_sqrt_Newton_o:wN

```

Newton's method maps $x \mapsto [(x + [10^8 a_1/x])/2]$ in each iteration, where $[b/c]$ denotes ε -TeX's division. This division rounds the real number b/c to the closest integer, rounding ties away from zero, hence when c is even, $b/c - 1/2 + 1/c \leq [b/c] \leq b/c + 1/2$ and when c is odd, $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2 - 1/(2c)$. For all c , $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2$.

Let us prove that the method converges when implemented with ε -TeX integer division, for any $10^6 \leq a_1 < 10^8$ and starting value $10^6 \leq x < 10^8$. Using the inequalities above and the arithmetic-geometric inequality $(x + t)/2 \geq \sqrt{xt}$ for $t = 10^8 a_1/x$, we find

$$x' = \left\lceil \frac{x + [10^8 a_1/x]}{2} \right\rceil \geq \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}.$$

After any step of iteration, we thus have $\delta = x - \sqrt{10^8 a_1} \geq -0.25 + 0.25 \cdot 10^{-8}$. The new difference $\delta' = x' - \sqrt{10^8 a_1}$ after one step is bounded above as

$$x' - \sqrt{10^8 a_1} \leq \frac{x + 10^8 a_1/x + 1/2}{2} + \frac{1}{2} - \sqrt{10^8 a_1} \leq \frac{\delta}{2} \frac{\delta}{\sqrt{10^8 a_1} + \delta} + \frac{3}{4}.$$

For $\delta > 3/2$, this last expression is $\leq \delta/2 + 3/4 < \delta$, hence δ decreases at each step: since all x are integers, δ must reach a value $-1/4 < \delta \leq 3/2$. In this range of values, we get $\delta' \leq \frac{3}{4} \frac{3}{2\sqrt{10^8 a_1}} + \frac{3}{4} \leq 0.75 + 1.125 \cdot 10^{-7}$. We deduce that the difference $\delta = x - \sqrt{10^8 a_1}$ eventually reaches a value in the interval $[-0.25 + 0.25 \cdot 10^{-8}, 0.75 + 11.25 \cdot 10^{-8}]$, whose width is $1 + 11 \cdot 10^{-8}$. The corresponding interval for x may contain two integers, hence x might oscillate between those two values.

However, the fact that $x \mapsto x - 1$ and $x - 1 \mapsto x$ puts stronger constraints, which are not compatible: the first implies

$$x + [10^8 a_1/x] \leq 2x - 2$$

hence $10^8 a_1/x \leq x - 3/2$, while the second implies

$$x - 1 + [10^8 a_1/(x - 1)] \geq 2x - 1$$

hence $10^8 a_1/(x - 1) \geq x - 1/2$. Combining the two inequalities yields $x^2 - 3x/2 \geq 10^8 a_1 \geq x - 3x/2 + 1/2$, which cannot hold. Therefore, the iteration always converges to a single integer x . To stop the iteration when two consecutive results are equal, the function `_fp_sqrt_Newton_o:wnn` receives the newly computed result as `#1`, the previous result as `#2`, and a_1 as `#3`. Note that ε -TeX combines the computation of a multiplication and a following division, thus avoiding overflow in `#3 * 100000000 / #1`. In any case, the result is within $[10^7, 10^8]$.

```

12443 \cs_new:Npn \_fp_sqrt_Newton_o:wnn #1; #2; #3
12444 {
12445   \if_int_compare:w #1 = #2 \exp_stop_f:
12446     \exp_after:wN \_fp_sqrt_auxi_o:NNNNwnnN
12447     \int_use:N \_int_eval:w 9999 9999 +
12448     \exp_after:wN \_fp_use_none_until_s:w
12449   \fi:
12450   \exp_after:wN \_fp_sqrt_Newton_o:wnn
12451   \int_use:N \_int_eval:w (#1 + #3 * 1 0000 0000 / #1) / \c_two ;
12452   #1; {#3}
12453 }

```

(End definition for `_fp_sqrt_Newton_o:wnn`.)

`_fp_sqrt_auxi_o:NNNNwnnN` This function is followed by $10^8 + x - 1$, which has 9 digits starting with 1, then ; $\{ \langle a_1 \rangle \} \{ \langle a_2 \rangle \} \langle a' \rangle$. Here, $x \simeq \sqrt{10^8 a_1}$ and we want to estimate the square root of $a = 10^{-8} a_1 + 10^{-16} a_2 + 10^{-17} a'$. We set up an initial underestimate

$$y = (x - 1)10^{-8} + 0.2499998875 \cdot 10^{-8} \lesssim \sqrt{a}.$$

From the inequalities shown earlier, we know that $y \leq \sqrt{10^{-8}a_1} \leq \sqrt{a}$ and that $\sqrt{10^{-8}a_1} \leq y + 10^{-8} + 11 \cdot 10^{-16}$ hence (using $0.1 \leq y \leq \sqrt{a} \leq 1$)

$$a - y^2 \leq 10^{-8}a_1 + 10^{-8} - y^2 \leq (y + 10^{-8} + 11 \cdot 10^{-16})^2 - y^2 + 10^{-8} < 3.2 \cdot 10^{-8},$$

and $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y) \leq 16 \cdot 10^{-8}$. Next, `_fp_sqrt_auxii_o:NnnnnnnnnN` will be called several times to get closer and closer underestimates of \sqrt{a} . By construction, the underestimates y are always increasing, $a - y^2 < 3.2 \cdot 10^{-8}$ for all. Also, $y < 1$.

```

12454 \cs_new:Npn \_fp_sqrt_auxi_o:NNNNwnnnN 1 #1#2#3#4#5;
12455 {
12456   \_fp_sqrt_auxii_o:NnnnnnnnnN
12457   \_fp_sqrt_auxiii_o:wnnnnnnnnn
12458   {#1#2#3#4} {#5} {2499} {9988} {7500}
12459 }

```

(End definition for `_fp_sqrt_auxi_o:NNNNwnnnN`.)

`_fp_sqrt_auxii_o:NnnnnnnnnN`

This receives a continuation function `#1`, then five blocks of 4 digits for y , then two 8-digit blocks and a single digit for a . A common estimate of $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y)$ is $(a - y^2)/(2y)$, which leads to alternating overestimates and underestimates. We tweak this, to only work with underestimates (no need then to worry about signs in the computation). Each step finds the largest integer $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then computes the integer (with ε -TeX's rounding division)

$$10^{4j}z = \left[([10^{4j}(a - y^2)] - 257) \cdot (0.5 \cdot 10^8) \right] / [10^8y + 1].$$

The choice of j ensures that $10^{4j}z < 2 \cdot 10^8 \cdot 0.5 \cdot 10^8/10^7 = 10^9$, thus $10^9 + 10^{4j}z$ has exactly 10 digits, does not overflow TeX's integer range, and starts with 1. Incidentally, since all $a - y^2 \leq 3.2 \cdot 10^{-8}$, we know that $j \geq 3$.

Let us show that z is an underestimate of $\sqrt{a} - y$. On the one hand, $\sqrt{a} - y \leq 16 \cdot 10^{-8}$ because this holds for the initial y and values of y can only increase. On the other hand, the choice of j implies that $\sqrt{a} - y \leq 5(\sqrt{a} + y)(\sqrt{a} - y) = 5(a - y^2) < 10^{9-4j}$. For $j = 3$, the first bound is better, while for larger j , the second bound is better. For all $j \in [3, 6]$, we find $\sqrt{a} - y < 16 \cdot 10^{-2j}$. From this, we deduce that

$$10^{4j}(\sqrt{a} - y) = \frac{10^{4j}(a - y^2 - (\sqrt{a} - y)^2)}{2y} \geq \frac{[10^{4j}(a - y^2)] - 257}{2 \cdot 10^{-8}[10^8y + 1]} + \frac{1}{2}$$

where we have replaced the bound $10^{4j}(16 \cdot 10^{-2j}) = 256$ by 257 and extracted the corresponding term $1/(2 \cdot 10^{-8}[10^8y + 1]) \geq 1/2$. Given that ε -TeX's integer division obeys $[b/c] \leq b/c + 1/2$, we deduce that $10^{4j}z \leq 10^{4j}(\sqrt{a} - y)$, hence $y + z \leq \sqrt{a}$ is an underestimate of \sqrt{a} , as claimed. One implementation detail: because the computation involves `-#4*#4 - 2*#3*#5 - 2*#2*#6` which may be as low as $-5 \cdot 10^8$, we need to use the `pack_big` functions, and the big shifts.

```

12460 \cs_new:Npn \_fp_sqrt_auxii_o:NnnnnnnnnN #1 #2#3#4#5#6 #7#8#9
12461 {
12462   \exp_after:wN #1
12463   \int_use:N \_int_eval:w \c\_fp_big_leading_shift_int

```

```

12464 + #7 - #2 * #2
12465 \exp_after:wN \__fp_pack_big:NNNNNNw
12466 \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
12467 - 2 * #2 * #3
12468 \exp_after:wN \__fp_pack_big:NNNNNNw
12469 \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
12470 + #8 - #3 * #3 - 2 * #2 * #4
12471 \exp_after:wN \__fp_pack_big:NNNNNNw
12472 \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
12473 - 2 * #3 * #4 - 2 * #2 * #5
12474 \exp_after:wN \__fp_pack_big:NNNNNNw
12475 \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
12476 + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
12477 \exp_after:wN \__fp_pack_big:NNNNNNw
12478 \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
12479 - 2 * #4 * #5 - 2 * #3 * #6
12480 \exp_after:wN \__fp_pack_big:NNNNNNw
12481 \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
12482 - #5 * #5 - 2 * #4 * #6
12483 \exp_after:wN \__fp_pack_big:NNNNNNw
12484 \int_use:N \__int_eval:w
12485 \c__fp_big_middle_shift_int
12486 - 2 * #5 * #6
12487 \exp_after:wN \__fp_pack_big:NNNNNNw
12488 \int_use:N \__int_eval:w
12489 \c__fp_big_trailing_shift_int
12490 - #6 * #6 ;
12491 % (
12492 - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
12493 {#2}{#3}{#4}{#5}{#6} {#7}{#8}#9
12494 }
(End definition for \__fp_sqrt_auxii_o:NnnnnnnN.)

```

```

\__fp_sqrt_auxiii_o:wnnnnnnnN
\__fp_sqrt_auxiv_o:NNNNNw
\__fp_sqrt_auxv_o:NNNNNw
\__fp_sqrt_auxvi_o:NNNNNw
\__fp_sqrt_auxvii_o:NNNNNw

```

We receive here the difference $a - y^2 = d = \sum_i d_i \cdot 10^{-4i}$, as $\langle d_2 \rangle$; $\{\langle d_3 \rangle\} \dots \{\langle d_{10} \rangle\}$, where each block has 4 digits, except $\langle d_2 \rangle$. This function finds the largest $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then leaves an open parenthesis and the integer $\lfloor 10^{4j}(a - y^2) \rfloor$ in an integer expression. The closing parenthesis is provided by the caller `__fp_sqrt_auxii_o:NnnnnnnN`, which completes the expression

$$10^{4j}z = \left[(\lfloor 10^{4j}(a - y^2) \rfloor - 257) \cdot (0.5 \cdot 10^8) \right] / \lfloor 10^8 y + 1 \rfloor$$

for an estimate of $10^{4j}(\sqrt{a} - y)$. If $d_2 \geq 2$, $j = 3$ and the `auxiv` auxiliary receives $10^{12}z$. If $d_2 \leq 1$ but $10^4 d_2 + d_3 \geq 2$, $j = 4$ and the `auxv` auxiliary is called, and receives $10^{16}z$, and so on. In all those cases, the `auxviii` auxiliary is set up to add z to y , then go back to the `auxii` step with continuation `auxiii` (the function we are currently describing). The maximum value of j is 6, regardless of whether $10^{12}d_2 + 10^8 d_3 + 10^4 d_4 + d_5 \geq 1$. In this last case, we detect when $10^{24}z < 10^7$, which essentially means $\sqrt{a} - y \lesssim 10^{-17}$: once this threshold is reached, there is enough information to find the correctly rounded \sqrt{a} with

only one more call to `__fp_sqrt_auxii_o:NnnnnnnnN`. Note that the iteration cannot be stuck before reaching $j = 6$, because for $j < 6$, one has $2 \cdot 10^8 \leq 10^{4(j+1)}(a - y^2)$, hence

$$10^{4j}z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{[10^8y + 1]} \geq (20000 - 257) \cdot 0.5 > 0.$$

```

12495 \cs_new:Npn \__fp_sqrt_auxiii_o:wnnnnnnnn
12496   #1; #2#3#4#5#6#7#8#9
12497   {
12498     \if_int_compare:w #1 > \c_one
12499       \exp_after:wN \__fp_sqrt_auxiv_o:NNNNNw
12500       \int_use:N \__int_eval:w (#1#2 %)
12501     \else:
12502       \if_int_compare:w #1#2 > \c_one
12503         \exp_after:wN \__fp_sqrt_auxv_o:NNNNNw
12504         \int_use:N \__int_eval:w (#1#2#3 %)
12505       \else:
12506         \if_int_compare:w #1#2#3 > \c_one
12507           \exp_after:wN \__fp_sqrt_auxvi_o:NNNNNw
12508           \int_use:N \__int_eval:w (#1#2#3#4 %)
12509         \else:
12510           \exp_after:wN \__fp_sqrt_auxvii_o:NNNNNw
12511           \int_use:N \__int_eval:w (#1#2#3#4#5 %)
12512         \fi:
12513       \fi:
12514     \fi:
12515   }
12516 \cs_new:Npn \__fp_sqrt_auxiv_o:NNNNNw #1#2#3#4#5#6;
12517   { \__fp_sqrt_auxviii_o:nnnnnnnn {#1#2#3#4#5#6} {00000000} }
12518 \cs_new:Npn \__fp_sqrt_auxv_o:NNNNNw #1#2#3#4#5#6;
12519   { \__fp_sqrt_auxviii_o:nnnnnnnn {000#1#2#3#4#5} {#60000} }
12520 \cs_new:Npn \__fp_sqrt_auxvi_o:NNNNNw #1#2#3#4#5#6;
12521   { \__fp_sqrt_auxviii_o:nnnnnnnn {0000000#1} {#2#3#4#5#6} }
12522 \cs_new:Npn \__fp_sqrt_auxvii_o:NNNNNw #1#2#3#4#5#6;
12523   {
12524     \if_int_compare:w #1#2 = \c_zero
12525       \exp_after:wN \__fp_sqrt_auxx_o:Nnnnnnnn
12526     \fi:
12527     \__fp_sqrt_auxviii_o:nnnnnnnn {00000000} {000#1#2#3#4#5}
12528   }

```

(End definition for `__fp_sqrt_auxiii_o:wnnnnnnnn` and others.)

`__fp_sqrt_auxviii_o:nnnnnnnn`
`__fp_sqrt_auxix_o:wnwnw`

Simply add the two 8-digit blocks of z , aligned to the last four of the five 4-digit blocks of y , then call the `auxii` auxiliary to evaluate $y'^2 = (y + z)^2$.

```

12529 \cs_new:Npn \__fp_sqrt_auxviii_o:nnnnnnnn #1#2 #3#4#5#6#7
12530   {
12531     \exp_after:wN \__fp_sqrt_auxix_o:wnwnw
12532     \int_use:N \__int_eval:w #3
12533     \exp_after:wN \__fp_basics_pack_low:NNNNNw
12534     \int_use:N \__int_eval:w #1 + 1#4#5

```

```

12535         \exp_after:wN \_fp_basics_pack_low:NNNNNw
12536         \int_use:N \_int_eval:w #2 + 1#6#7 ;
12537     }
12538 \cs_new:Npn \_fp_sqrt_auxix_o:wnwnw #1; #2#3; #4#5;
12539 {
12540     \_fp_sqrt_auxii_o:NnnnnnnnN
12541     \_fp_sqrt_auxiii_o:wnnnnnnnn {#1}{#2}{#3}{#4}{#5}
12542 }
(End definition for \_fp_sqrt_auxviii_o:nnnnnnn and \_fp_sqrt_auxix_o:wnwnw.)

```

```

\_fp_sqrt_auxx_o:Nnnnnnnn
\_fp_sqrt_auxxi_o:wnnnN

```

At this stage, $j = 6$ and $10^{24}z < 10^7$, hence

$$10^7 + 1/2 > 10^{24}z + 1/2 \geq (10^{24}(a - y^2) - 258) \cdot (0.5 \cdot 10^8) / (10^8y + 1),$$

then $10^{24}(a - y^2) - 258 < 2(10^7 + 1/2)(y + 10^{-8})$, and

$$10^{24}(a - y^2) < (10^7 + 1290.5)(1 + 10^{-8}/y)(2y) < (10^7 + 1290.5)(1 + 10^{-7})(y + \sqrt{a}),$$

which finally implies $0 \leq \sqrt{a} - y < 0.2 \cdot 10^{-16}$. In particular, y is an underestimate of \sqrt{a} and $y + 0.5 \cdot 10^{-16}$ is a (strict) overestimate. There is at exactly one multiple m of $0.5 \cdot 10^{-16}$ in the interval $[y, y + 0.5 \cdot 10^{-16})$. If $m^2 > a$, then the square root is inexact and is obtained by rounding $m - \epsilon$ to a multiple of 10^{-16} (the precise shift $0 < \epsilon < 0.5 \cdot 10^{-16}$ is irrelevant for rounding). If $m^2 = a$ then the square root is exactly m , and there is no rounding. If $m^2 < a$ then we round $m + \epsilon$. For now, discard a few irrelevant arguments #1, #2, #3, and find the multiple of $0.5 \cdot 10^{-16}$ within $[y, y + 0.5 \cdot 10^{-16})$; rather, only the last 4 digits #8 of y are considered, and we do not perform any carry yet. The `auxxi` auxiliary sets up `auxii` with a continuation function `auxxii` instead of `auxiii` as before. To prevent `auxii` from giving a negative results $a - m^2$, we compute $a + 10^{-16} - m^2$ instead, always positive since $m < \sqrt{a} + 0.5 \cdot 10^{-16}$ and $a \leq 1 - 10^{-16}$.

```

12543 \cs_new:Npn \_fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
12544 {
12545     \exp_after:wN \_fp_sqrt_auxxi_o:wnnnN
12546     \int_use:N \_int_eval:w
12547     (#8 + 2499) / 5000 * 5000 ;
12548     {#4} {#5} {#6} {#7} ;
12549 }
12550 \cs_new:Npn \_fp_sqrt_auxxi_o:wnnnN #1; #2; #3#4#5
12551 {
12552     \_fp_sqrt_auxii_o:NnnnnnnnN
12553     \_fp_sqrt_auxxii_o:nnnnnnnnw
12554     #2 {#1}
12555     {#3} { #4 + \c_one } #5
12556 }
(End definition for \_fp_sqrt_auxx_o:Nnnnnnnn and \_fp_sqrt_auxxi_o:wnnnN.)

```

```

\_fp_sqrt_auxxii_o:nnnnnnnnw
\_fp_sqrt_auxxiii_o:w

```

The difference $0 \leq a + 10^{-16} - m^2 \leq 10^{-16} + (\sqrt{a} - m)(\sqrt{a} + m) \leq 2 \cdot 10^{-16}$ was just computed: its first 8 digits vanish, as do the next four, #1, and most of the following four, #2. The guess m is an overestimate if $a + 10^{-16} - m^2 < 10^{-16}$, that is, #1#2

vanishes. Otherwise it is an underestimate, unless $a + 10^{-16} - m^2 = 10^{-16}$ exactly. For an underestimate, call the `auxxiv` function with argument 9998. For an exact result call it with 9999, and for an overestimate call it with 10000.

```

12557 \cs_new:Npn \__fp_sqrt_auxxii_o:nnnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
12558 {
12559   \if_int_compare:w #1#2 > \c_zero
12560     \if_int_compare:w #1#2 = \c_one
12561       \if_int_compare:w #3#4 = \c_zero
12562         \if_int_compare:w #5#6 = \c_zero
12563           \if_int_compare:w #7#8 = \c_zero
12564             \__fp_sqrt_auxxiii_o:w
12565           \fi:
12566         \fi:
12567       \fi:
12568     \fi:
12569     \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
12570     \__int_value:w 9998
12571   \else:
12572     \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
12573     \__int_value:w 10000
12574   \fi:
12575 ;
12576 }
12577 \cs_new:Npn \__fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: #1 \fi: ;
12578 {
12579   \fi: \fi: \fi: \fi: \fi:
12580   \__fp_sqrt_auxxiv_o:wnnnnnnnN 9999 ;
12581 }

```

(End definition for `__fp_sqrt_auxxii_o:nnnnnnnnw` and `__fp_sqrt_auxxiii_o:w`.)

`__fp_sqrt_auxxiv_o:wnnnnnnnN`

This receives 9998, 9999 or 10000 as #1 when m is an underestimate, exact, or an overestimate, respectively. Then comes m as five blocks of 4 digits, but where the last block #6 may be 0, 5000, or 10000. In the latter case, we need to add a carry, unless m is an overestimate (#1 is then 10000). Then comes a as three arguments. Rounding is done by `__fp_round:NNN`, whose first argument is the final sign 0 (square roots are positive). We fake its second argument. It should be the last digit kept, but this is only used when ties are “rounded to even”, and only when the result is exactly half-way between two representable numbers rational square roots of numbers with 16 significant digits have: this situation never arises for the square root, as any exact square root of a 16 digit number has at most 8 significant digits. Finally, the last argument is the next digit, possibly shifted by 1 when there are further nonzero digits. This is achieved by `__fp_round_digit:Nw`, which receives (after removal of the 10000’s digit) one of 0000, 0001, 4999, 5000, 5001, or 9999, which it converts to 0, 1, 4, 5, 6, and 9, respectively.

```

12582 \cs_new:Npn \__fp_sqrt_auxxiv_o:wnnnnnnnN #1; #2#3#4#5#6 #7#8#9
12583 {
12584   \exp_after:wN \__fp_basics_pack_high:NNNNNw
12585   \int_use:N \__int_eval:w 1 0000 0000 + #2#3
12586   \exp_after:wN \__fp_basics_pack_low:NNNNNw

```

```

12587 \int_use:N \__int_eval:w 1 0000 0000
12588 + #4#5
12589 \if_int_compare:w #6 > #1 \exp_stop_f: + \c_one \fi:
12590 + \exp_after:wN \__fp_round:NNN
12591 \exp_after:wN 0
12592 \exp_after:wN 0
12593 \__int_value:w
12594 \exp_after:wN \use_i:nn
12595 \exp_after:wN \__fp_round_digit:Nw
12596 \int_use:N \__int_eval:w #6 + 19999 - #1 ;
12597 \exp_after:wN ;
12598 }
(End definition for \__fp_sqrt_auxxiv_o:wnnnnnnN.)

```

29.6 Setting the sign

`__fp_set_sign_o:w` This function is used for the unary minus and for `abs`. It leaves the sign of `nan` invariant, turns negative numbers (sign 2) to positive numbers (sign 0) and positive numbers (sign 0) to positive or negative numbers depending on `#1`. It also expands after itself in the input stream, just like `__fp+_o:ww`.

```

12599 \cs_new:Npn \__fp_set_sign_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
12600 {
12601   \exp_after:wN \__fp_exp_after_o:w
12602   \exp_after:wN \s__fp
12603   \exp_after:wN \__fp_chk:w
12604   \exp_after:wN #2
12605   \__int_value:w
12606   \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
12607   #4;
12608 }
(End definition for \__fp_set_sign_o:w.)
12609 </initex | package>

```

30 l3fp-extended implementation

```

12610 <*initex | package>
12611 <@@=fp>

```

30.1 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 `__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.

30.2 Helpers for numbers with extended precision

`\c__fp_one_fixed_tl` The fixed-point number 1, used in `l3fp-expo`.

```
12612 \tl_const:Nn \c__fp_one_fixed_tl
12613 { {10000} {0000} {0000} {0000} {0000} {0000} }
(End definition for \c__fp_one_fixed_tl.)
```

`__fp_fixed_continue:wn` This function does nothing. Of course, there is no bound on a_1 (except T_EX’s own $2^{31} - 1$).

```
12614 \cs_new:Npn \__fp_fixed_continue:wn #1; #2 { #2 #1; }
(End definition for \__fp_fixed_continue:wn.)
```

`__fp_fixed_add_one:wN` This function adds 1 to the fixed point $\langle a \rangle$, by changing a_1 to $10000 + a_1$, then calls the $\langle continuation \rangle$. This requires $a_1 \leq 2^{31} - 10001$.

```
12615 \cs_new:Npn \__fp_fixed_add_one:wN #1#2; #3
12616 {
12617   \exp_after:wN #3 \exp_after:wN
12618   { \int_use:N \__int_eval:w \c_ten_thousand + #1 } #2 ;
12619 }
(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 $[0, 5 \cdot 10^8 - 1]$.

```

12620 \cs_new:Npn \_fp_fixed_div_myriad:wn #1#2#3#4#5#6;
12621 {
12622   \exp_after:wN \_fp_fixed_mul_after:wnn
12623   \int_use:N \_int_eval:w \c\_fp_leading_shift_int
12624   \exp_after:wN \_fp_pack:NNNNNw
12625   \int_use:N \_int_eval:w \c\_fp_trailing_shift_int
12626   + #1 ; {#2}{#3}{#4}{#5};
12627 }

```

(End definition for `_fp_fixed_div_myriad:wn`.)

`_fp_fixed_mul_after:wnn` The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the $\langle continuation \rangle$ #2 in front. The $\langle continuation \rangle$ was brought up through the expansions by the packing functions.

```

12628 \cs_new:Npn \_fp_fixed_mul_after:wnn #1; #2; #3 { #3 {#1} #2; }

```

(End definition for `_fp_fixed_mul_after:wnn`.)

30.3 Multiplying a fixed point number by a short one

`_fp_fixed_mul_short:wnn` Computes the product $c = ab$ of $a = \sum_i \langle a_i \rangle 10^{-4i}$ and $b = \sum_i \langle b_i \rangle 10^{-4i}$, rounds it to the closest multiple of 10^{-24} , and leaves $\langle continuation \rangle \{ \langle c_1 \rangle \} \dots \{ \langle c_6 \rangle \}$; in the input stream, where each of the $\langle c_i \rangle$ are blocks of 4 digits, except $\langle c_1 \rangle$, which is any TeX integer. Note that indices for $\langle b \rangle$ start at 0: a second operand of $\{0001\}\{0000\}\{0000\}$ will leave the first operand unchanged (rather than dividing it by 10^4 , as `_fp_fixed_mul:wnn` would).

```

12629 \cs_new:Npn \_fp_fixed_mul_short:wnn #1#2#3#4#5#6; #7#8#9;
12630 {
12631   \exp_after:wN \_fp_fixed_mul_after:wnn
12632   \int_use:N \_int_eval:w \c\_fp_leading_shift_int
12633   + #1*#7
12634   \exp_after:wN \_fp_pack:NNNNNw
12635   \int_use:N \_int_eval:w \c\_fp_middle_shift_int
12636   + #1*#8 + #2*#7
12637   \exp_after:wN \_fp_pack:NNNNNw
12638   \int_use:N \_int_eval:w \c\_fp_middle_shift_int
12639   + #1*#9 + #2*#8 + #3*#7
12640   \exp_after:wN \_fp_pack:NNNNNw
12641   \int_use:N \_int_eval:w \c\_fp_middle_shift_int
12642   + #2*#9 + #3*#8 + #4*#7
12643   \exp_after:wN \_fp_pack:NNNNNw
12644   \int_use:N \_int_eval:w \c\_fp_middle_shift_int
12645   + #3*#9 + #4*#8 + #5*#7
12646   \exp_after:wN \_fp_pack:NNNNNw
12647   \int_use:N \_int_eval:w \c\_fp_trailing_shift_int
12648   + #4*#9 + #5*#8 + #6*#7

```

```

12649             + ( #5*#9 + #6*#8 + #6*#9 / \c_ten_thousand )
12650             / \c_ten_thousand ; ;
12651     }
(End definition for \_fp_fixed_mul_short:wnn.)

```

30.4 Dividing a fixed point number by a small integer

```

\_fp_fixed_div_int:wnN
\_fp_fixed_div_int:wnN
\_fp_fixed_div_int_auxi:wnn
  \_fp_fixed_div_int_auxii:wnn
\_fp_fixed_div_int_pack:Nw
\_fp_fixed_div_int_after:Nw

```

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.

```

12652 \cs_new:Npn \_fp_fixed_div_int:wnN #1#2#3#4#5#6 ; #7 ; #8
12653 {
12654   \exp_after:wN \_fp_fixed_div_int_after:Nw
12655   \exp_after:wN #8
12656   \int_use:N \_int_eval:w \c_minus_one
12657   \_fp_fixed_div_int:wnN
12658   #1; {\#7} \_fp_fixed_div_int_auxi:wnn
12659   #2; {\#7} \_fp_fixed_div_int_auxi:wnn
12660   #3; {\#7} \_fp_fixed_div_int_auxi:wnn
12661   #4; {\#7} \_fp_fixed_div_int_auxi:wnn
12662   #5; {\#7} \_fp_fixed_div_int_auxi:wnn
12663   #6; {\#7} \_fp_fixed_div_int_auxii:wnn ;
12664 }
12665 \cs_new:Npn \_fp_fixed_div_int:wnN #1; #2 #3
12666 {

```

```

12667 \exp_after:wN #3
12668 \int_use:N \__int_eval:w #1 / #2 - \c_one ;
12669 {#2}
12670 {#1}
12671 }
12672 \cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1; #2 #3
12673 {
12674 + #1
12675 \exp_after:wN \__fp_fixed_div_int_pack:Nw
12676 \int_use:N \__int_eval:w 9999
12677 \exp_after:wN \__fp_fixed_div_int:wN
12678 \int_use:N \__int_eval:w #3 - #1*#2 \__int_eval_end:
12679 }
12680 \cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + \c_two ; }
12681 \cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
12682 \cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }
(End definition for \__fp_fixed_div_int:wN.)

```

30.5 Adding and subtracting fixed points

`__fp_fixed_add:wN` Computes $a + b$ (resp. $a - b$) and feeds the result to the *continuation*. This function requires $0 \leq a_1, b_1 \leq 114748$, its result must be positive (this happens automatically for addition) and its first group must have at most 5 digits: $(a \pm b)_1 < 100000$. The two functions only differ by a sign, hence use a common auxiliary. It would be nice to grab the 12 brace groups in one go; only 9 parameters are allowed. Start by grabbing the sign, a_1, \dots, a_4 , the rest of a , and b_1 and b_2 . The second auxiliary receives the rest of a , the sign multiplying b , the rest of b , and the *continuation* as arguments. After going down through the various level, we go back up, packing digits and bringing the *continuation* (#8, then #7) from the end of the argument list to its start.

```

12683 \cs_new_nopar:Npn \__fp_fixed_add:wN { \__fp_fixed_add:Nnnnnwnn + }
12684 \cs_new_nopar:Npn \__fp_fixed_sub:wN { \__fp_fixed_add:Nnnnnwnn - }
12685 \cs_new:Npn \__fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
12686 {
12687 \exp_after:wN \__fp_fixed_add_after:NNNNNwn
12688 \int_use:N \__int_eval:w 9 9999 9998 + #2#3 #1 #7#8
12689 \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
12690 \int_use:N \__int_eval:w 1 9999 9998 + #4#5
12691 \__fp_fixed_add:nnNnnwn #6 #1
12692 }
12693 \cs_new:Npn \__fp_fixed_add:nnNnnwn #1#2 #3 #4#5 #6#7 ; #8
12694 {
12695 #3 #4#5
12696 \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
12697 \int_use:N \__int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
12698 }
12699 \cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
12700 { + #1 ; {#7} {#2#3#4#5} {#6} }
12701 \cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7

```


12702 { #7 {#1#2#3#4#5} {#6} }

(End definition for _fp_fixed_add:wnn and _fp_fixed_sub:wnn.)

30.6 Multiplying fixed points

_fp_fixed_mul:wnn
_fp_fixed_mul:nnnnnnnw

Computes $a \times b$ and feeds the result to $\langle continuation \rangle$. This function requires $0 \leq a_1, b_1 < 10000$. Once more, we need to play around the limit of 9 arguments for \TeX macros. Note that we don't need to obtain an exact rounding, contrarily to the $*$ operator, so things could be harder. We wish to perform carries in

$$\begin{aligned} a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\ & + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\ & + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\ & + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\ & + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} + a_1 \cdot b_5 + a_5 \cdot b_1 \right) \end{aligned}$$

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 $\langle continuation \rangle$ is finally placed in front of the 6 brace groups by `_fp_fixed_mul_after:wnn`.

```
12703 \cs_new:Npn \_fp_fixed_mul:wnn #1#2#3#4 #5; #6#7#8#9
12704 {
12705   \exp_after:wN \_fp_fixed_mul_after:wnn
12706   \int_use:N \__int_eval:w \c__fp_leading_shift_int
12707   \exp_after:wN \_fp_pack:NNNNNw
12708   \int_use:N \__int_eval:w \c__fp_middle_shift_int
12709   + #1*#6
12710   \exp_after:wN \_fp_pack:NNNNNw
12711   \int_use:N \__int_eval:w \c__fp_middle_shift_int
12712   + #1*#7 + #2*#6
12713   \exp_after:wN \_fp_pack:NNNNNw
12714   \int_use:N \__int_eval:w \c__fp_middle_shift_int
12715   + #1*#8 + #2*#7 + #3*#6
12716   \exp_after:wN \_fp_pack:NNNNNw
12717   \int_use:N \__int_eval:w \c__fp_middle_shift_int
12718   + #1*#9 + #2*#8 + #3*#7 + #4*#6
12719   \exp_after:wN \_fp_pack:NNNNNw
12720   \int_use:N \__int_eval:w \c__fp_trailing_shift_int
12721   + #2*#9 + #3*#8 + #4*#7
12722   + ( #3*#9 + #4*#8
```

```

12723         + \_fp_fixed_mul:nnnnnnnw #5 {#6}{#7} {#1}{#2}
12724     }
12725 \cs_new:Npn \_fp_fixed_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
12726 {
12727     #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c_ten_thousand
12728     + #1*#3 + #5*#7 ; ;
12729 }
(End definition for \_fp_fixed_mul:wwn.)

```

30.7 Combining product and sum of fixed points

_fp_fixed_mul_add:wwwn 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.

```

12730 \cs_new:Npn \_fp_fixed_mul_add:wwwn #1; #2; #3#4#5#6#7#8;
12731 {
12732     \exp_after:wN \_fp_fixed_mul_after:wwn
12733     \int_use:N \_int_eval:w \c\_fp_big_leading_shift_int
12734     \exp_after:wN \_fp_pack_big:NNNNNNw
12735     \int_use:N \_int_eval:w \c\_fp_big_middle_shift_int + #3 #4
12736     \_fp_fixed_mul_add:Nwnnnwnnn +
12737     + #5 #6 ; #2 ; #1 ; #2 ; +
12738     + #7 #8 ; ;
12739 }
12740 \cs_new:Npn \_fp_fixed_mul_sub_back:wwwn #1; #2; #3#4#5#6#7#8;
12741 {
12742     \exp_after:wN \_fp_fixed_mul_after:wwn
12743     \int_use:N \_int_eval:w \c\_fp_big_leading_shift_int
12744     \exp_after:wN \_fp_pack_big:NNNNNNw

```

```

12745     \int_use:N \__int_eval:w \c__fp_big_middle_shift_int + #3 #4
12746     \__fp_fixed_mul_add:Nwnnnwnnn -
12747     + #5 #6 ; #2 ; #1 ; #2 ; -
12748     + #7 #8 ; ;
12749   }
12750 \cs_new:Npn \__fp_fixed_one_minus_mul:wnn #1; #2;
12751 {
12752   \exp_after:wN \__fp_fixed_mul_after:wnn
12753   \int_use:N \__int_eval:w \c__fp_big_leading_shift_int
12754   \exp_after:wN \__fp_pack_big:NNNNNNw
12755   \int_use:N \__int_eval:w \c__fp_big_middle_shift_int + 1 0000 0000
12756   \__fp_fixed_mul_add:Nwnnnwnnn -
12757   ; #2 ; #1 ; #2 ; -
12758   ; ;
12759 }

```

(End definition for `__fp_fixed_mul_add:wnnn`, `__fp_fixed_mul_sub_back:wnnn`, and `__fp_fixed_mul_one_minus_mul:wnnn`)

`__fp_fixed_mul_add:Nwnnnwnnn` Here, $\langle op \rangle$ is either + or -. Arguments #3, #4, #5 are $\langle b_1 \rangle$, $\langle b_2 \rangle$, $\langle b_3 \rangle$; arguments #7, #8, #9 are $\langle a_1 \rangle$, $\langle a_2 \rangle$, $\langle a_3 \rangle$. We can build three levels: $a_1 \cdot b_1$ for 10^{-8} , $(a_1 \cdot b_2 + a_2 \cdot b_1)$ for 10^{-12} , and $(a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4)$ for 10^{-16} . The a - b products use the sign #1. Note that #2 is empty for `__fp_fixed_one_minus_mul:wnn`. We call the `ii` auxiliary for levels 10^{-20} and 10^{-24} , keeping the pieces of $\langle a \rangle$ we've read, but not $\langle b \rangle$, since there is another copy later in the input stream.

```

12760 \cs_new:Npn \__fp_fixed_mul_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
12761 {
12762   #1 #7*#3
12763   \exp_after:wN \__fp_pack_big:NNNNNNw
12764   \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
12765   #1 #7*#4 #1 #8*#3
12766   \exp_after:wN \__fp_pack_big:NNNNNNw
12767   \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
12768   #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
12769   \exp_after:wN \__fp_pack_big:NNNNNNw
12770   \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
12771   #1 \__fp_fixed_mul_add:nnnnwnnnn {#7}{#8}{#9}
12772 }

```

(End definition for `__fp_fixed_mul_add:Nwnnnwnnn`.)

`__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

$$\begin{aligned}
 & b_1 + a_4 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1 \\
 & b_2 + a_4 \cdot b_3 + a_3 \cdot b_4 + a_2.
 \end{aligned}$$

Obviously, those expressions make no mathematical sense: we will complete them with $a_5 \cdot$ and $\cdot b_5$, and with $a_6 \cdot b_1 + a_5 \cdot$ and $\cdot b_5 + a_1 \cdot b_6$, and of course with the trailing $+ c_5 c_6$. To do all this, we keep a_1 , a_5 , a_6 , and the corresponding pieces of $\langle b \rangle$.

```

12773 \cs_new:Npn \__fp_fixed_mul_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
12774 {
12775   ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
12776   \exp_after:wN \__fp_pack_big:NNNNNNw
12777   \int_use:N \__int_eval:w \c__fp_big_trailing_shift_int
12778   \__fp_fixed_mul_add:nnnnwnnwN
12779   { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
12780   { #7 + #4*#8 + #3*#9 + #2 }
12781   {#1} #5;
12782   {#6}
12783 }

```

(End definition for __fp_fixed_mul_add:nnnnwnnnn.)

__fp_fixed_mul_add:nnnnwnnwN

Complete the $\langle partial_1 \rangle$ and $\langle partial_2 \rangle$ expressions as explained for the `ii` auxiliary. The second one is divided by 10000: this is the carry from level 10^{-28} . The trailing $+c_5c_6$ 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.

```

12784 \cs_new:Npn \__fp_fixed_mul_add:nnnnwnnwN #1#2 #3#4#5; #6#7#8; #9
12785 {
12786   #9 (#4* #1 *#7)
12787   #9 (#5*#6+#4* #2 *#7+#3*#8) / \c_ten_thousand
12788 }

```

(End definition for __fp_fixed_mul_add:nnnnwnnwN.)

30.8 Extended-precision floating point numbers

In this section we manipulate floating point numbers with roughly 24 significant figures (“extended-precision” numbers, in short, “ep”), which take the form of an integer exponent, followed by a comma, then six groups of digits, ending with a semicolon. The first group of digit may be any non-negative integer, while other groups of digits have 4 digits. In other words, an extended-precision number is an exponent ending in a comma, then a fixed point number.

__fp_ep_to_fixed:wwn
 __fp_ep_to_fixed_auxi:www
 __fp_ep_to_fixed_auxii:nnnnnnnnwn

Converts an extended-precision number with an exponent at most 4 to a fixed point number whose first block will have 12 digits, most often starting with many zeros.

```

12789 \cs_new:Npn \__fp_ep_to_fixed:wwn #1,#2
12790 {
12791   \exp_after:wN \__fp_ep_to_fixed_auxi:www
12792   \int_use:N \__int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
12793   \tex_romannumeral:D -‘0
12794   \prg_replicate:nn { \c_four - \int_max:nn {#1} { -32 } } { 0 } ;
12795 }
12796 \cs_new:Npn \__fp_ep_to_fixed_auxi:www 1#1; #2; #3#4#5#6#7;
12797 {
12798   \__fp_pack_eight:wNNNNNNNN
12799   \__fp_pack_twice_four:wNNNNNNNN

```

```

12800     \__fp_pack_twice_four:wNNNNNNNN
12801     \__fp_pack_twice_four:wNNNNNNNN
12802     \__fp_ep_to_fixed_auxii:nnnnnnwn ;
12803     #2 #1#3#4#5#6#7 0000 !
12804 }
12805 \cs_new:Npn \__fp_ep_to_fixed_auxii:nnnnnnwn #1#2#3#4#5#6#7; #8! #9
12806 { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }
(End definition for \__fp_ep_to_fixed:wnn.)

```

```

\__fp_ep_to_ep:wwN
\__fp_ep_to_ep_loop:N
\__fp_ep_to_ep_end:www
\__fp_ep_to_ep_zero:ww

```

Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation #8 is placed before the resulting exponent–mantissa pair. The input exponent may in fact be given as an integer expression. The `loop` auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the `end` auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by some 0, then packs them up neatly in $3 \times 2 = 6$ blocks of four. At the end of the day, remove with `__fp_use_i:ww` any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).

```

12807 \cs_new:Npn \__fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
12808 {
12809     \exp_after:wN #8
12810     \int_use:N \__int_eval:w #1 + \c_four
12811     \exp_after:wN \use_i:nn
12812     \exp_after:wN \__fp_ep_to_ep_loop:N
12813     \int_use:N \__int_eval:w 1 0000 0000 + #2 \__int_eval_end:
12814     #3#4#5#6#7 ; ; !
12815 }
12816 \cs_new:Npn \__fp_ep_to_ep_loop:N #1
12817 {
12818     \if_meaning:w 0 #1
12819     - \c_one
12820     \else:
12821     \__fp_ep_to_ep_end:www #1
12822     \fi:
12823     \__fp_ep_to_ep_loop:N
12824 }
12825 \cs_new:Npn \__fp_ep_to_ep_end:www
12826 #1 \fi: \__fp_ep_to_ep_loop:N #2; #3!
12827 {
12828     \fi:
12829     \if_meaning:w ; #1
12830     - \c_two * \c__fp_max_exponent_int
12831     \__fp_ep_to_ep_zero:ww
12832     \fi:
12833     \__fp_pack_twice_four:wNNNNNNNN
12834     \__fp_pack_twice_four:wNNNNNNNN
12835     \__fp_pack_twice_four:wNNNNNNNN

```

```

12836     \__fp_use_i:ww , ;
12837     #1 #2 0000 0000 0000 0000 0000 0000 ;
12838 }
12839 \cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
12840 { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }
(End definition for \__fp_ep_to_ep:wwN.)

```

`__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]`.

```

12841 \cs_new:Npn \__fp_ep_compare:www #1,#2#3#4#5#6#7;
12842 { \__fp_ep_compare_aux:www {#1}{#2}{#3}{#4}{#5}; #6#7; }
12843 \cs_new:Npn \__fp_ep_compare_aux:www #1;#2;#3,#4#5#6#7#8#9;
12844 {
12845     \if_case:w
12846         \__fp_compare_npos:nwn #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
12847         \if_int_compare:w #2 = #8#9 \exp_stop_f:
12848             0
12849         \else:
12850             \if_int_compare:w #2 < #8#9 - \fi: 1
12851             \fi:
12852         \or: 1
12853         \else: -1
12854         \fi:
12855     }
(End definition for \__fp_ep_compare:www.)

```

`__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]`.

```

12856 \cs_new:Npn \__fp_ep_mul:wwwN #1,#2; #3,#4;
12857 {
12858     \__fp_ep_to_ep:wwN #3,#4;
12859     \__fp_fixed_continue:wn
12860     {
12861         \__fp_ep_to_ep:wwN #1,#2;
12862         \__fp_ep_mul_raw:wwwN
12863     }
12864     \__fp_fixed_continue:wn
12865 }
12866 \cs_new:Npn \__fp_ep_mul_raw:wwwN #1,#2; #3,#4; #5
12867 {
12868     \__fp_fixed_mul:wn #2; #4;
12869     { \exp_after:wN #5 \int_use:N \__int_eval:w #1 + #3 , }
12870 }
(End definition for \__fp_ep_mul:wwwN and \__fp_ep_mul_raw:wwwN.)

```

30.9 Dividing extended-precision numbers

Divisions of extended-precision numbers are difficult to perform with exact rounding: the technique used in `l3fp-basics` for 16-digit floating point numbers does not generalize easily to 24-digit numbers. Thankfully, there is no need for exact rounding.

Let us call $\langle n \rangle$ the numerator and $\langle d \rangle$ the denominator. After a simple normalization step, we can assume that $\langle n \rangle \in [0.1, 1)$ and $\langle d \rangle \in [0.1, 1)$, and compute $\langle n \rangle / (10 \langle d \rangle) \in (0.01, 1)$. In terms of the 6 blocks of digits $\langle n_1 \rangle \cdots \langle n_6 \rangle$ and the 6 blocks $\langle d_1 \rangle \cdots \langle d_6 \rangle$, the condition translates to $\langle n_1 \rangle, \langle d_1 \rangle \in [1000, 9999]$.

We will first find an integer estimate $a \simeq 10^8 / \langle d \rangle$ by computing

$$\begin{aligned}\alpha &= \left\lceil \frac{10^9}{\langle d_1 \rangle + 1} \right\rceil \\ \beta &= \left\lfloor \frac{10^9}{\langle d_1 \rangle} \right\rfloor \\ a &= 10^3 \alpha + (\beta - \alpha) \cdot \left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) - 1250,\end{aligned}$$

where $\left\lceil \cdot \right\rceil$ denotes ε -TeX's rounding division, which rounds ties away from zero. The idea is to interpolate between $10^3 \alpha$ and $10^3 \beta$ with a parameter $\langle d_2 \rangle / 10^4$, so that when $\langle d_2 \rangle = 0$ one gets $a = 10^3 \beta - 1250 \simeq 10^{12} / \langle d_1 \rangle \simeq 10^8 / \langle d \rangle$, while when $\langle d_2 \rangle = 9999$ one gets $a = 10^3 \alpha - 1250 \simeq 10^{12} / (\langle d_1 \rangle + 1) \simeq 10^8 / \langle d \rangle$. The shift by 1250 helps to ensure that a is an underestimate of the correct value. We will prove that

$$1 - 1.755 \cdot 10^{-5} < \frac{\langle d \rangle a}{10^8} < 1.$$

We can then compute the inverse of $\langle d \rangle a / 10^8 = 1 - \epsilon$ using the relation $1/(1 - \epsilon) \simeq (1 + \epsilon)(1 + \epsilon^2) + \epsilon^4$, which is correct up to a relative error of $\epsilon^5 < 1.6 \cdot 10^{-24}$. This allows us to find the desired ratio as

$$\frac{\langle n \rangle}{\langle d \rangle} = \frac{\langle n \rangle a}{10^8} ((1 + \epsilon)(1 + \epsilon^2) + \epsilon^4).$$

Let us prove the upper bound first (multiplied by 10^{15}). Note that $10^7 \langle d \rangle < 10^3 \langle d_1 \rangle + 10^{-1}(\langle d_2 \rangle + 1)$, and that ε -TeX's division $\left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor$ will at most underestimate $10^{-1}(\langle d_2 \rangle + 1)$ by 0.5, as can be checked for each possible last digit of $\langle d_2 \rangle$. Then,

$$\begin{aligned}10^7 \langle d \rangle a &< \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \beta + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \alpha - 1250 \right) \quad (1) \\ &< \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \left(\frac{10^9}{\langle d_1 \rangle} + \frac{1}{2} \right) + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \left(\frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2} \right) - 1250 \right) \quad (2) \\ &< \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 750 \right) \quad (3)\end{aligned}$$

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` *<denominator>* *<numerator>*, responsible for estimating the inverse of the denominator.

```

12871 \cs_new:Npn \__fp_ep_div:wwwn #1,#2; #3,#4;
12872 {
12873   \__fp_ep_to_ep:wwN #1,#2;
12874   \__fp_fixed_continue:wn
12875   {
12876     \__fp_ep_to_ep:wwN #3,#4;
12877     \__fp_ep_div_esti:wwwn
12878   }
12879 }
```

(End definition for `__fp_ep_div:wwwn`.)

`__fp_ep_div_esti:wwwn` The `esti` function evaluates $\alpha = 10^9 / (\langle d_1 \rangle + 1)$, which is used twice in the expression for a , and combines the exponents `#1` and `#4` (with a shift by 1 because we will compute $\langle n \rangle / (10 \langle d \rangle)$). Then the `estii` function evaluates $10^9 + a$, and puts the exponent `#2` after the continuation `#7`: from there on we can forget exponents and focus on the mantissa. The `estiii` function multiplies the denominator `#7` by $10^{-8}a$ (obtained as a

split into the single digit #1 and two blocks of 4 digits, #2#3#4#5 and #6). The result $10^{-8}a\langle d \rangle = (1 - \epsilon)$, and a partially packed $10^{-9}a$ (as a block of four digits, and five individual digits, not packed by lack of available macro parameters here) are passed to `__fp_ep_div_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.

```

12880 \cs_new:Npn \__fp_ep_div_esti:wwwn #1,#2#3; #4,
12881 {
12882   \exp_after:wN \__fp_ep_div_estii:wnnnwn
12883   \int_use:N \__int_eval:w 10 0000 0000 / ( #2 + \c_one )
12884   \exp_after:wN ;
12885   \int_use:N \__int_eval:w #4 - #1 + \c_one ,
12886   {#2} #3;
12887 }
12888 \cs_new:Npn \__fp_ep_div_estii:wnnnwn #1; #2,#3#4#5; #6; #7
12889 {
12890   \exp_after:wN \__fp_ep_div_estiii:NNNNNwwwn
12891   \int_use:N \__int_eval:w 10 0000 0000 - 1750
12892   + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
12893   {#3}{#4}#5; #6; { #7 #2, }
12894 }
12895 \cs_new:Npn \__fp_ep_div_estiii:NNNNNwwwn 1#1#2#3#4#5#6; #7;
12896 {
12897   \__fp_fixed_mul_short:wnn #7; {#1}{#2#3#4#5}{#6};
12898   \__fp_ep_div_epsilon:wnNNNNn {#1#2#3#4}#5#6
12899   \__fp_fixed_mul:wnn
12900 }

```

(End definition for `__fp_ep_div_esti:wwwn`, `__fp_ep_div_estii:wnnnwn`, and `__fp_ep_div_estiii:NNNNNwwwn`.)

```

\__fp_ep_div_epsilon:wnNNNNn
\__fp_ep_div_eps_pack:NNNNNw
\__fp_ep_div_epsii:wnNNNNn

```

The bounds shown above imply that the `epsi` function's first operand is $(1 - \epsilon)$ with $\epsilon \in [0, 1.755 \cdot 10^{-5}]$. The `epsi` function computes ϵ as $1 - (1 - \epsilon)$. Since $\epsilon < 10^{-4}$, its first block vanishes and there is no need to explicitly use #1 (which is 9999). Then `epsii` evaluates $10^{-9}a/(1 - \epsilon)$ as $(1 + \epsilon^2)(1 + \epsilon)(10^{-9}a\epsilon) + 10^{-9}a$. Importantly, we compute $10^{-9}a\epsilon$ before multiplying it with the rest, rather than multiplying by ϵ and then $10^{-9}a$, as this second option loses more precision. Also, the combination of `short_mul` and `div_myriad` is both faster and more precise than a simple `mul`.

```

12901 \cs_new:Npn \__fp_ep_div_epsilon:wnNNNNn #1#2#3#4#5#6;
12902 {
12903   \exp_after:wN \__fp_ep_div_epsii:wnNNNNn
12904   \int_use:N \__int_eval:w 1 9998 - #2
12905   \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
12906   \int_use:N \__int_eval:w 1 9999 9998 - #3#4
12907   \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
12908   \int_use:N \__int_eval:w 2 0000 0000 - #5#6 ; ;
12909 }
12910 \cs_new:Npn \__fp_ep_div_eps_pack:NNNNNw #1#2#3#4#5#6;
12911 { + #1 ; {#2#3#4#5} {#6} }
12912 \cs_new:Npn \__fp_ep_div_epsii:wnNNNNn 1#1; #2; #3#4#5#6#7#8
12913 {

```

```

12914 \__fp_fixed_mul:wnn {0000}{#1}#2; {0000}{#1}#2;
12915 \__fp_fixed_add_one:wN
12916 \__fp_fixed_mul:wnn {10000} {#1} #2 ;
12917 {
12918   \__fp_fixed_mul_short:wnn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
12919   \__fp_fixed_div_myriad:wn
12920   \__fp_fixed_mul:wnn
12921 }
12922 \__fp_fixed_add:wnn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
12923 }
(End definition for \__fp_ep_div_epsilon:wnNNNNNn, \__fp_ep_div_eps_pack:NNNNNw, and \__fp_ep_div_epsilonii:wnNNNNNn.)

```

30.10 Inverse square root of extended precision numbers

The idea here is similar to division. Normalize the input, multiplying by powers of 100 until we have $x \in [0.01, 1)$. Then find an integer approximation $r \in [101, 1003]$ of $10^2/\sqrt{x}$, as the fixed point of iterations of the Newton method: essentially $r \mapsto (r + 10^8/(x_1 r))/2$, starting from a guess that optimizes the number of steps before convergence. In fact, just as there is a slight shift when computing divisions to ensure that some inequalities hold, we will replace 10^8 by a slightly larger number which will ensure that $r^2 x \geq 10^4$. This also causes $r \in [101, 1003]$. Another correction to the above is that the input is actually normalized to $[0.1, 1)$, and we use either 10^8 or 10^9 in the Newton method, depending on the parity of the exponent. Skipping those technical hurdles, once we have the approximation r , we set $y = 10^{-4}r^2x$ (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.

```

12924 \cs_new:Npn \__fp_ep_isqrt:wnn #1,#2;
12925 {
12926   \__fp_ep_to_ep:wnn #1,#2;
12927   \__fp_ep_isqrt_auxi:wnn
12928 }
12929 \cs_new:Npn \__fp_ep_isqrt_auxi:wnn #1,
12930 {
12931   \exp_after:wN \__fp_ep_isqrt_auxii:wnnnwn

```

```

12932 \int_use:N \__int_eval:w
12933 \int_if_odd:nTF {#1}
12934 { (\c_one - #1) / \c_two , 535 , { 0 } { } }
12935 { \c_one - #1 / \c_two , 168 , { } { 0 } }
12936 }
12937 \cs_new:Npn \__fp_ep_isqrt_auxii:wwnnwn #1, #2, #3#4 #5#6; #7
12938 {
12939 \__fp_ep_isqrt_esti:wwnnwn #2, 0, #5, {#3} {#4}
12940 {#5} #6 ; { #7 #1 , }
12941 }
(End definition for \__fp_ep_isqrt:wnn.)

```

```

\__fp_ep_isqrt_esti:wwnnwn
\__fp_ep_isqrt_estii:wwnnwn
\__fp_ep_isqrt_estiii:NNNNwwnn

```

If the last two approximations gave the same result, we are done: call the `esti` 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 .

```

12942 \cs_new:Npn \__fp_ep_isqrt_esti:wwnnwn #1, #2, #3, #4
12943 {
12944 \if_int_compare:w #1 = #2 \exp_stop_f:
12945 \exp_after:wN \__fp_ep_isqrt_estii:wwnnwn
12946 \fi:
12947 \exp_after:wN \__fp_ep_isqrt_esti:wwnnwn
12948 \int_use:N \__int_eval:w
12949 (#1 + 1 0050 0000 #4 / (#1 * #3)) / \c_two ,
12950 #1, #3, {#4}
12951 }
12952 \cs_new:Npn \__fp_ep_isqrt_estii:wwnnwn #1, #2, #3, #4#5
12953 {
12954 \exp_after:wN \__fp_ep_isqrt_estiii:NNNNwwnn
12955 \int_use:N \__int_eval:w 1000 0000 + #2 * #2 #5 * \c_five
12956 \exp_after:wN , \int_use:N \__int_eval:w 10000 + #2 ;
12957 }
12958 \cs_new:Npn \__fp_ep_isqrt_estiii:NNNNwwnn 1#1#2#3#4#5#6, 1#7#8; #9;
12959 {
12960 \__fp_fixed_mul_short:wnn #9; {#1} {#2#3#4#5} {#600} ;
12961 \__fp_ep_isqrt_epsilon:wN
12962 \__fp_fixed_mul_short:wnn {#7} {#80} {0000} ;
12963 }
(End definition for \__fp_ep_isqrt_esti:wwnnwn, \__fp_ep_isqrt_estii:wwnnwn, and \__fp_ep_isqrt_estiii:NNNNwwnn)

```

```

\__fp_ep_isqrt_epsilon:wN
\__fp_ep_isqrt_epsii:wnN

```

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.`

```

12964 \cs_new:Npn \__fp_ep_isqrt_epsilon:wN #1;
12965 {
12966   \__fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}{0000}; #1;
12967   \__fp_ep_isqrt_epsilon:wwN #1;
12968   \__fp_ep_isqrt_epsilon:wwN #1;
12969   \__fp_ep_isqrt_epsilon:wwN #1;
12970 }
12971 \cs_new:Npn \__fp_ep_isqrt_epsilon:wwN #1; #2;
12972 {
12973   \__fp_fixed_mul:wwn #1; #1;
12974   \__fp_fixed_mul_sub_back:wwwN #2;
12975   {15000}{0000}{0000}{0000}{0000}{0000};
12976   \__fp_fixed_mul:wwn #1;
12977 }

```

(End definition for `__fp_ep_isqrt_epsilon:wN` and `__fp_ep_isqrt_epsilon:wwN`.)

30.11 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` 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.

```

12978 \cs_new:Npn \__fp_ep_to_float:wwN #1,
12979 { + \__int_eval:w #1 \__fp_fixed_to_float:wN }
12980 \cs_new:Npn \__fp_ep_inv_to_float:wwN #1,#2;
12981 {
12982   \__fp_ep_div:wwwN 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1,#2;
12983   \__fp_ep_to_float:wwN
12984 }

```

(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.

```

12985 \cs_new:Npn \__fp_fixed_inv_to_float:wN
12986 { \__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`.

```

12987 \cs_new:Npn \__fp_fixed_to_float_rad:wN #1;
12988 {
12989   \__fp_fixed_mul:wwn #1; {5729}{5779}{5130}{8232}{0876}{7981};
12990   { \__fp_ep_to_float:wwN 2, }
12991 }

```

(End definition for `__fp_fixed_to_float_rad:wN`.)

```
\__fp_fixed_to_float:wN yields
\__fp_fixed_to_float:Nw
    <exponent'> ; {\langle a'_1 \rangle} {\langle a'_2 \rangle} {\langle a'_3 \rangle} {\langle a'_4 \rangle} ;
```

And the `to_fixed` version gives six brace groups instead of 4, ensuring that $1000 \leq \langle a'_1 \rangle \leq 9999$. At this stage, we know that $\langle a'_1 \rangle$ is positive (otherwise, it is sign of an error before), and we assume that it is less than 10^8 .⁸

```
12992 \cs_new:Npn \__fp_fixed_to_float:Nw #1#2; { \__fp_fixed_to_float:wN #2; #1 }
12993 \cs_new:Npn \__fp_fixed_to_float:wN #1#2#3#4#5#6; #7
12994 {
12995   + \__int_eval:w \c_four % for the 8-digit-at-the-start thing.
12996   \exp_after:wN \exp_after:wN
12997   \exp_after:wN \__fp_fixed_to_loop:N
12998   \exp_after:wN \use_none:n
12999   \int_use:N \__int_eval:w
13000   1 0000 0000 + #1 \exp_after:wN \__fp_use_none_stop_f:n
13001   \__int_value:w 1#2 \exp_after:wN \__fp_use_none_stop_f:n
13002   \__int_value:w 1#3#4 \exp_after:wN \__fp_use_none_stop_f:n
13003   \__int_value:w 1#5#6
13004   \exp_after:wN ;
13005   \exp_after:wN ;
13006 }
13007 \cs_new:Npn \__fp_fixed_to_loop:N #1
13008 {
13009   \if_meaning:w 0 #1
13010   - \c_one
13011   \exp_after:wN \__fp_fixed_to_loop:N
13012   \else:
13013   \exp_after:wN \__fp_fixed_to_loop_end:w
13014   \exp_after:wN #1
13015   \fi:
13016 }
13017 \cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
13018 {
13019   \if_meaning:w ; #1
13020   \exp_after:wN \__fp_fixed_to_float_zero:w
13021   \else:
13022   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
13023   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
13024   \exp_after:wN \__fp_fixed_to_float_pack:ww
13025   \exp_after:wN ;
13026   \fi:
13027   #1 #2 0000 0000 0000 0000 ;
13028 }
13029 \cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
13030 {
```

⁸Bruno: I must double check this assumption.

```

13031     - \c_two * \c__fp_max_exponent_int ;
13032     {0000} {0000} {0000} {0000} ;
13033   }
13034   \cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
13035   {
13036     \if_int_compare:w #2 > \c_four
13037       \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
13038     \fi:
13039     ; #1 ;
13040   }
13041   \cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
13042   {
13043     \exp_after:wN \__fp_basics_pack_high:NNNNNw
13044     \int_use:N \__int_eval:w 1 #1#2
13045     \exp_after:wN \__fp_basics_pack_low:NNNNNw
13046     \int_use:N \__int_eval:w 1 #3#4 + \c_one ;
13047   }

```

(End definition for __fp_fixed_to_float:wN and __fp_fixed_to_float:Nw.)

```

13048   </initex | package>

```

31 l3fp-expo implementation

```

13049   <*initex | package>
13050   <@@=fp>

```

31.1 Logarithm

31.1.1 Work plan

As for many other functions, we filter out special cases in __fp_ln_o:w. Then __fp_ln_npos_o:w receives a positive normal number, which we write in the form $a \cdot 10^b$ with $a \in [0.1, 1)$.

The rest of this section is actually not in sync with the code. Or is the code not in sync with the section? In the current code, $c \in [1, 10]$ will be such that $0.7 \leq ac < 1.4$.

We are given a positive normal number, of the form $a \cdot 10^b$ with $a \in [0.1, 1)$. To compute its logarithm, we find a small integer $5 \leq c < 50$ such that $0.91 \leq ac/5 < 1.1$, and use the relation

$$\ln(a \cdot 10^b) = b \cdot \ln(10) - \ln(c/5) + \ln(ac/5).$$

The logarithms $\ln(10)$ and $\ln(c/5)$ are looked up in a table. The last term is computed using the following Taylor series of \ln near 1:

$$\ln\left(\frac{ac}{5}\right) = \ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + t^2 \left(\frac{1}{3} + t^2 \left(\frac{1}{5} + t^2 \left(\frac{1}{7} + t^2 \left(\frac{1}{9} + \dots\right)\right)\right)\right)\right)$$

where $t = 1 - 10/(ac + 5)$. We can now see one reason for the choice of $ac \sim 5$: then $ac + 5 = 10(1 - \epsilon)$ with $-0.05 < \epsilon \leq 0.045$, hence

$$t = \frac{\epsilon}{1 - \epsilon} = \epsilon(1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4) \dots,$$

is not too difficult to compute.

31.1.2 Some constants

A few values of the logarithm as extended fixed point numbers. Those are needed in the implementation. It turns out that we don't need the value of $\ln(5)$.

```

\c__fp_ln_i_fixed_t1
\c__fp_ln_ii_fixed_t1
\c__fp_ln_iii_fixed_t1
\c__fp_ln_iv_fixed_t1
\c__fp_ln_vi_fixed_t1
\c__fp_ln_vii_fixed_t1
\c__fp_ln_viii_fixed_t1
\c__fp_ln_ix_fixed_t1
\c__fp_ln_x_fixed_t1
13051 \tl_const:Nn \c__fp_ln_i_fixed_t1 { {0000}{0000}{0000}{0000}{0000} }
13052 \tl_const:Nn \c__fp_ln_ii_fixed_t1 { {6931}{4718}{0559}{9453}{0941}{7232} }
13053 \tl_const:Nn \c__fp_ln_iii_fixed_t1 { {10986}{1228}{8668}{1096}{9139}{5245} }
13054 \tl_const:Nn \c__fp_ln_iv_fixed_t1 { {13862}{9436}{1119}{8906}{1883}{4464} }
13055 \tl_const:Nn \c__fp_ln_vi_fixed_t1 { {17917}{5946}{9228}{0550}{0081}{2477} }
13056 \tl_const:Nn \c__fp_ln_vii_fixed_t1 { {19459}{1014}{9055}{3133}{0510}{5353} }
13057 \tl_const:Nn \c__fp_ln_viii_fixed_t1 { {20794}{4154}{1679}{8359}{2825}{1696} }
13058 \tl_const:Nn \c__fp_ln_ix_fixed_t1 { {21972}{2457}{7336}{2193}{8279}{0490} }
13059 \tl_const:Nn \c__fp_ln_x_fixed_t1 { {23025}{8509}{2994}{0456}{8401}{7991} }

```

(End definition for `\c__fp_ln_i_fixed_t1` and others.)

31.1.3 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`.

```

13060 \cs_new:Npn \__fp_ln_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
13061 {
13062   \if_meaning:w 2 #3
13063     \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
13064   \fi:
13065   \if_case:w #2 \exp_stop_f:
13066     \__fp_case_use:nw
13067     { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
13068   \or:
13069   \else:
13070     \__fp_case_return_same_o:w
13071   \fi:
13072   \__fp_ln_npos_o:w \s__fp \__fp_chk:w #2#3#4;
13073 }

```

(End definition for `__fp_ln_o:w`.)

31.1.4 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.

```

13074 \cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3;

```

```

13075 { %^A todo: ln(1) should be "exact zero", not "underflow"
13076 \exp_after:wN \__fp_sanitize:Nw
13077 \__int_value:w % for the overall sign
13078 \if_int_compare:w #1 < \c_one
13079 2
13080 \else:
13081 0
13082 \fi:
13083 \exp_after:wN \exp_stop_f:
13084 \int_use:N \__int_eval:w % for the exponent
13085 \__fp_ln_significand:NNNNnnnnN #2#3
13086 \__fp_ln_exponent:wn {#1}
13087 }
(End definition for \__fp_ln_npos_o:w.)

```

$\backslash_fp_ln_significand:NNNNnnnnN$ $\backslash_fp_ln_significand:NNNNnnnnN \langle X_1 \rangle \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \} \langle continuation \rangle$
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.

```

13088 \cs_new:Npn \__fp_ln_significand:NNNNnnnnN #1#2#3#4
13089 {
13090 \exp_after:wN \__fp_ln_x_ii:wnnnnn
13091 \__int_value:w
13092 \if_case:w #1 \exp_stop_f:
13093 \or:
13094 \if_int_compare:w #2 < \c_four
13095 \__int_eval:w \c_ten - #2
13096 \else:
13097 6
13098 \fi:
13099 \or: 4
13100 \or: 3
13101 \or: 2
13102 \or: 2
13103 \or: 2
13104 \else: 1
13105 \fi:
13106 ; { #1 #2 #3 #4 }
13107 }

```

(End definition for $\backslash_fp_ln_significand:NNNNnnnnN$.)

$\backslash_fp_ln_x_ii:wnnnnn$ We have thus found $c \in [1, 10]$ such that $0.7 \leq ac < 1.4$ in all cases. Compute $1 + x = 1 + ac \in [1.7, 2.4)$.

```

13108 \cs_new:Npn \__fp_ln_x_ii:wnnnnn #1; #2#3#4#5
13109 {
13110 \exp_after:wN \__fp_ln_div_after:Nw
13111 \cs:w c__fp_ln_ \tex_romannumeral:D #1 _fixed_tl \exp_after:wN \cs_end:

```



```

13112 \__int_value:w
13113 \exp_after:wN \__fp_ln_x_iv:wnnnnnnnnn
13114 \int_use:N \__int_eval:w
13115 \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
13116 \int_use:N \__int_eval:w 9999 9990 + #1*#2#3 +
13117 \exp_after:wN \__fp_ln_x_iii:NNNNNNw
13118 \int_use:N \__int_eval:w 10 0000 0000 + #1*#4#5 ;
13119 {20000} {0000} {0000} {0000}
13120 } %^A todo: reoptimize (a generalization attempt failed).
13121 \cs_new:Npn \__fp_ln_x_iii:NNNNNNw #1#2 #3#4#5#6 #7;
13122 { #1#2; {#3#4#5#6} {#7} }
13123 \cs_new:Npn \__fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
13124 {
13125 #1#2#3#4#5 + \c_one ;
13126 {#1#2#3#4#5} {#6}
13127 }

```

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.

```

13128 \cs_new:Npn \__fp_ln_x_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
13129 {
13130   \exp_after:wN \__fp_div_significand_pack:NNN
13131   \int_use:N \__int_eval:w
13132   \__fp_ln_div_i:w #1 ;
13133   #6 #7 ; {#8} {#9}
13134   {#2} {#3} {#4} {#5}
13135   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
13136   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
13137   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
13138   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
13139   { \exp_after:wN \__fp_ln_div_vi:wnn \__int_value:w #1 }
13140 }
13141 \cs_new:Npn \__fp_ln_div_i:w #1;
13142 {
13143   \exp_after:wN \__fp_div_significand_calc:wnnnnnnnn
13144   \int_use:N \__int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
13145 }
13146 \cs_new:Npn \__fp_ln_div_ii:wnn #1; #2;#3 % y; B1;B2 <- for k=1
13147 {
13148   \exp_after:wN \__fp_div_significand_pack:NNN
13149   \int_use:N \__int_eval:w
13150   \exp_after:wN \__fp_div_significand_calc:wnnnnnnnn
13151   \int_use:N \__int_eval:w 999999 + #2 #3 / #1 ; % Q2

```

⁹Bruno: to be completed.

```

13152         #2 #3 ;
13153     }
13154 \cs_new:Npn \__fp_ln_div_vi:wwn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
13155 {
13156     \exp_after:wN \__fp_div_significand_pack:NNN
13157     \int_use:N \__int_eval:w 1000000 + #2 #3 / #1 ; % Q6
13158 }

```

We now have essentially¹⁰

$$\begin{aligned} & \backslash_fp_ln_div_after:Nw \langle fixed\ tl \rangle \backslash_fp_div_significand_pack:NNN 10^6 + \\ & Q_1 \backslash_fp_div_significand_pack:NNN 10^6 + Q_2 \backslash_fp_div_significand_ \\ & pack:NNN 10^6 + Q_3 \backslash_fp_div_significand_pack:NNN 10^6 + Q_4 \backslash_fp_ \\ & div_significand_pack:NNN 10^6 + Q_5 \backslash_fp_div_significand_pack:NNN \\ & 10^6 + Q_6 ; \langle exponent \rangle ; \langle continuation \rangle \end{aligned}$$

where $\langle fixed\ tl \rangle$ holds the logarithm of a number in $[1, 10]$, and $\langle exponent \rangle$ is the exponent. Also, the expansion is done backwards. Then $\backslash_fp_div_significand_pack:NNN$ 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

$$\begin{aligned} & \backslash_fp_ln_div_after:Nw \langle fixed\ tl \rangle \langle 1d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \\ & \langle 4d \rangle ; \langle exponent \rangle ; \end{aligned}$$

Just as with division, we know that the first two digits are 1 and 0 because of bounds on the final result of the division $2/(x+1)$, which is between roughly 0.8 and 1.2. We then compute $1 - 2/(x+1)$, after testing whether $2/(x+1)$ is greater than or smaller than 1.

```

13159 \cs_new:Npn \__fp_ln_div_after:Nw #1#2;
13160 {
13161     \if_meaning:w 0 #2
13162         \exp_after:wN \__fp_ln_t_small:Nw
13163     \else:
13164         \exp_after:wN \__fp_ln_t_large:NNw
13165         \exp_after:wN -
13166     \fi:
13167     #1
13168 }
13169 \cs_new:Npn \__fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
13170 {
13171     \exp_after:wN \__fp_ln_t_large:NNw
13172     \exp_after:wN + % <sign>
13173     \exp_after:wN #1
13174     \int_use:N \__int_eval:w 9999 - #2 \exp_after:wN ;
13175     \int_use:N \__int_eval:w 9999 - #3 \exp_after:wN ;
13176     \int_use:N \__int_eval:w 9999 - #4 \exp_after:wN ;
13177     \int_use:N \__int_eval:w 9999 - #5 \exp_after:wN ;
13178     \int_use:N \__int_eval:w 9999 - #6 \exp_after:wN ;
13179     \int_use:N \__int_eval:w 1 0000 - #7 ;
13180 }

```

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.

$\backslash_fp_ln_t_large:NNw \langle sign \rangle \langle fixed\ tl \rangle \langle t_1 \rangle ; \langle t_2 \rangle ; \langle t_3 \rangle ; \langle t_4 \rangle ; \langle t_5 \rangle ; \langle t_6 \rangle ;$
 $\langle exponent \rangle ; \langle continuation \rangle$

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 $\backslash_fp_ln_t_small:w$, they can have less than 4 digits.

```

13181 \cs_new:Npn \__fp_ln_t_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
13182 {
13183   \exp_after:wN \__fp_ln_square_t_after:w
13184   \int_use:N \__int_eval:w 9999 0000 + #3*#3
13185   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
13186   \int_use:N \__int_eval:w 9999 0000 + 2*#3*#4
13187   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
13188   \int_use:N \__int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
13189   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
13190   \int_use:N \__int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
13191   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
13192   \int_use:N \__int_eval:w 1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
13193   + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
13194   % ; ; ;
13195   \exp_after:wN \__fp_ln_twice_t_after:w
13196   \int_use:N \__int_eval:w -1 + 2*#3
13197   \exp_after:wN \__fp_ln_twice_t_pack:Nw
13198   \int_use:N \__int_eval:w 9999 + 2*#4
13199   \exp_after:wN \__fp_ln_twice_t_pack:Nw
13200   \int_use:N \__int_eval:w 9999 + 2*#5
13201   \exp_after:wN \__fp_ln_twice_t_pack:Nw
13202   \int_use:N \__int_eval:w 9999 + 2*#6
13203   \exp_after:wN \__fp_ln_twice_t_pack:Nw
13204   \int_use:N \__int_eval:w 9999 + 2*#7
13205   \exp_after:wN \__fp_ln_twice_t_pack:Nw
13206   \int_use:N \__int_eval:w 10000 + 2*#8 ; ;
13207   { \__fp_ln_c:NwNw #1 }
13208   #2
13209 }
13210 \cs_new:Npn \__fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
13211 \cs_new:Npn \__fp_ln_twice_t_after:w #1; { ; ; ; {#1} }
13212 \cs_new:Npn \__fp_ln_square_t_pack:NNNNNw #1 #2#3#4#5 #6;
13213 { + #1#2#3#4#5 ; {#6} }
13214 \cs_new:Npn \__fp_ln_square_t_after:w 1 0 #1#2#3 #4;
13215 { \__fp_ln_Taylor:wwNw {0#1#2#3} {#4} }
(End definition for \__fp_ln_x_ii:wnnnn.)

```

$\backslash_fp_ln_Taylor:wwNw$ Denoting $T = t^2$, we get

$\backslash_fp_ln_Taylor:wwNw \{ \langle T_1 \rangle \} \{ \langle T_2 \rangle \} \{ \langle T_3 \rangle \} \{ \langle T_4 \rangle \} \{ \langle T_5 \rangle \} \{ \langle T_6 \rangle \} ; ;$
 $\{ \langle (2t)_1 \rangle \} \{ \langle (2t)_2 \rangle \} \{ \langle (2t)_3 \rangle \} \{ \langle (2t)_4 \rangle \} \{ \langle (2t)_5 \rangle \} \{ \langle (2t)_6 \rangle \} ; \{ \backslash_fp_ln_c:NwNn \langle sign \rangle \} \langle fixed\ tl \rangle \langle exponent \rangle ; \langle continuation \rangle$

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 ($< 10^4$).

```
13216 \cs_new:Npn \__fp_ln_Taylor:wwNw
13217 { \__fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000}{0000} ; }
13218 \cs_new:Npn \__fp_ln_Taylor_loop:www #1; #2; #3;
13219 {
13220   \if_int_compare:w #1 = \c_one
13221     \__fp_ln_Taylor_break:w
13222   \fi:
13223   \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl ; #1;
13224   \__fp_fixed_add:wwN #2;
13225   \__fp_fixed_mul:wwN #3;
13226   {
13227     \exp_after:wN \__fp_ln_Taylor_loop:www
13228     \int_use:N \__int_eval:w #1 - \c_two ;
13229   }
13230   #3;
13231 }
13232 \cs_new:Npn \__fp_ln_Taylor_break:w \fi: #1 \__fp_fixed_add:wwN #2#3; #4 ;;
13233 {
13234   \fi:
13235   \exp_after:wN \__fp_fixed_mul:wwN
13236   \exp_after:wN { \int_use:N \__int_eval:w 10000 + #2 } #3;
13237 }
```

(End definition for __fp_ln_Taylor:wwNw. This function is documented on page ??.)

```
\__fp_ln_c:NwNw \__fp_ln_c:NwNw <sign> {\langle r_1 \rangle} {\langle r_2 \rangle} {\langle r_3 \rangle} {\langle r_4 \rangle} {\langle r_5 \rangle} {\langle r_6 \rangle} ; <fixed tl>
<exponent> ; <continuation>
```

We are now reduced to finding $\ln(c)$ and $\langle exponent \rangle \ln(10)$ in a table, and adding it to the mixture. The first step is to get $\ln(c) - \ln(x) = -\ln(a)$, then we get $\mathbf{b} \ln(10)$ and add or subtract.

For now, $\ln(x)$ is given as $\cdot 10^0$. Unless both the exponent is 1 and $c = 1$, we shift to working in units of $\cdot 10^4$, since the final result will be at least $\ln(10/7) \simeq 0.35$.¹²

¹¹Bruno: add explanations.

¹²Bruno: that was wrong at some point, I must check.

```

13238 \cs_new:Npn \__fp_ln_c:NwNw #1 #2; #3
13239 {
13240   \if_meaning:w + #1
13241     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_sub:wnn
13242   \else:
13243     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_add:wnn
13244   \fi:
13245   #3 ; #2 ;
13246 }

```

13

(End definition for `__fp_ln_c:NwNw`. This function is documented on page ??.)

`__fp_ln_exponent:wn` `__fp_ln_exponent:wn {⟨ s_1 ⟩} {⟨ s_2 ⟩} {⟨ s_3 ⟩} {⟨ s_4 ⟩} {⟨ s_5 ⟩} {⟨ s_6 ⟩} ; {⟨ $exponent$ ⟩}`
 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.

```

13247 \cs_new:Npn \__fp_ln_exponent:wn #1; #2
13248 {
13249   \if_case:w #2 \exp_stop_f:
13250     \c_zero \__fp_case_return:nw { \__fp_fixed_to_float:Nw 2 }
13251   \or:
13252     \exp_after:wN \__fp_ln_exponent_one:ww \__int_value:w
13253   \else:
13254     \if_int_compare:w #2 > \c_zero
13255       \exp_after:wN \__fp_ln_exponent_small:NNww
13256       \exp_after:wN 0
13257       \exp_after:wN \__fp_fixed_sub:wnn \__int_value:w
13258     \else:
13259       \exp_after:wN \__fp_ln_exponent_small:NNww
13260       \exp_after:wN 2
13261       \exp_after:wN \__fp_fixed_add:wnn \__int_value:w -
13262     \fi:
13263   \fi:
13264   #2; #1;
13265 }

```

Now we painfully write all the cases.¹⁴ No overflow nor underflow can happen, except when computing $\ln(1)$.

```

13266 \cs_new:Npn \__fp_ln_exponent_one:ww 1; #1;
13267 {
13268   \c_zero
13269   \exp_after:wN \__fp_fixed_sub:wnn \c__fp_ln_x_fixed_t1 ; #1;

```

¹³Bruno: this *must* be updated with correct values!

¹⁴Bruno: do rounding.

```

13270     \__fp_fixed_to_float:wN 0
13271 }

```

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.

```

13272 \cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
13273 {
13274     \c_four
13275     \exp_after:wN \__fp_fixed_mul:wwn
13276     \c__fp_ln_x_fixed_tl ;
13277     {#3}{0000}{0000}{0000}{0000}{0000} ;
13278     #2
13279     {0000}{#4}{#5}{#6}{#7}{#8};
13280     \__fp_fixed_to_float:wN #1
13281 }

```

(End definition for __fp_ln_exponent:wn. This function is documented on page ??.)

31.2 Exponential

31.2.1 Sign, exponent, and special numbers

__fp_exp_o:w

```

13282 \cs_new:Npn \__fp_exp_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
13283 {
13284     \if_case:w #2 \exp_stop_f:
13285     \__fp_case_return_o:Nw \c_one_fp
13286     \or:
13287     \exp_after:wN \__fp_exp_normal:w
13288     \or:
13289     \if_meaning:w 0 #3
13290     \exp_after:wN \__fp_case_return_o:Nw
13291     \exp_after:wN \c_inf_fp
13292     \else:
13293     \exp_after:wN \__fp_case_return_o:Nw
13294     \exp_after:wN \c_zero_fp
13295     \fi:
13296     \or:
13297     \__fp_case_return_same_o:w
13298     \fi:
13299     \s__fp \__fp_chk:w #2#3#4;
13300 }

```

(End definition for __fp_exp_o:w.)

__fp_exp_normal:w

__fp_exp_pos:Nnwnw

```

13301 \cs_new:Npn \__fp_exp_normal:w \s__fp \__fp_chk:w 1#1
13302 {
13303     \if_meaning:w 0 #1
13304     \__fp_exp_pos:Nnwnw + \__fp_fixed_to_float:wN

```

```

13305     \else:
13306         \__fp_exp_pos:Nnwnw - \__fp_fixed_inv_to_float:wN
13307     \fi:
13308 }
13309 \cs_new:Npn \__fp_exp_pos:Nnwnw #1#2#3 \fi: #4#5;
13310 {
13311     \fi:
13312     \exp_after:wN \__fp_sanitize:Nw
13313     \exp_after:wN 0
13314     \__int_value:w #1 \__int_eval:w
13315     \if_int_compare:w #4 < - \c_eight
13316         \c_one
13317         \exp_after:wN \__fp_add_big_i_o:wNww
13318         \int_use:N \__int_eval:w \c_one - #4 ;
13319         0 {1000}{0000}{0000}{0000} ; #5;
13320         \tex_romannumeral:D
13321     \else:
13322         \if_int_compare:w #4 > \c_five % cf \c__fp_max_exponent_int
13323         \exp_after:wN \__fp_exp_overflow:
13324         \tex_romannumeral:D
13325     \else:
13326         \if_int_compare:w #4 < \c_zero
13327             \exp_after:wN \use_i:nn
13328         \else:
13329             \exp_after:wN \use_ii:nn
13330         \fi:
13331         {
13332             \c_zero
13333             \__fp_decimate:nNnnnn { - #4 }
13334             \__fp_exp_Taylor:Nnnwn
13335         }
13336         {
13337             \__fp_decimate:nNnnnn { \c_sixteen - #4 }
13338             \__fp_exp_pos_large:NnnNwn
13339         }
13340         #5
13341         {#4}
13342         #1 #2 0
13343         \tex_romannumeral:D
13344     \fi:
13345     \fi:
13346     \exp_after:wN \c_zero
13347 }
13348 \cs_new:Npn \__fp_exp_overflow:
13349 { + \c_two * \c__fp_max_exponent_int ; {1000} {0000} {0000} {0000} ; }
(End definition for \__fp_exp_normal:w and \__fp_exp_pos:Nnwnw.)

```

`__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.

```

13350 \cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
13351 {
13352     #6
13353     \__fp_pack_twice_four:wNNNNNNNN
13354     \__fp_pack_twice_four:wNNNNNNNN
13355     \__fp_pack_twice_four:wNNNNNNNN
13356     \__fp_exp_Taylor_ii:ww
13357     ; #2#3#4 0000 0000 ;
13358 }
13359 \cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;
13360 { \__fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s_stop }
13361 \cs_new:Npn \__fp_exp_Taylor_loop:www #1; #2; #3;
13362 {
13363     \if_int_compare:w #1 = \c_one
13364     \exp_after:wN \__fp_exp_Taylor_break:Nww
13365     \fi:
13366     \__fp_fixed_div_int:wwN #3 ; #1 ;
13367     \__fp_fixed_add_one:wN
13368     \__fp_fixed_mul:wwN #2 ;
13369     {
13370         \exp_after:wN \__fp_exp_Taylor_loop:www
13371         \int_use:N \__int_eval:w #1 - 1 ;
13372         #2 ;
13373     }
13374 }
13375 \cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2; #3 \s_stop
13376 { \__fp_fixed_add_one:wN #2 ; }
(End definition for \__fp_exp_Taylor:Nnnwn.)

```

```

\__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.

```

13377 \cs_new:Npn \__fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
13378 {
13379     \exp_after:wN \exp_after:wN
13380     \cs:w \__fp_exp_large\tex_romannumeral:D #6:wN \exp_after:wN \cs_end:
13381     \exp_after:wN \c__fp_one_fixed_tl
13382     \exp_after:wN ;

```

```

13383     \__int_value:w #3 #4 \exp_stop_f:
13384     #5 00000 ;
13385 }
13386 \cs_new:Npn \__fp_exp_large:w #1 \or: #2 \fi:
13387 { \fi: \__fp_fixed_mul:wn #1; }
13388 \cs_new:Npn \__fp_exp_large_v:wN #1; #2
13389 {
13390     \if_case:w #2 ~           \exp_after:wN \__fp_fixed_continue:wn \or:
13391     + 4343 \__fp_exp_large:w {8806}{8182}{2566}{2921}{5872}{6150} \or:
13392     + 8686 \__fp_exp_large:w {7756}{0047}{2598}{6861}{0458}{3204} \or:
13393     + 13029 \__fp_exp_large:w {6830}{5723}{7791}{4884}{1932}{7351} \or:
13394     + 17372 \__fp_exp_large:w {6015}{5609}{3095}{3052}{3494}{7574} \or:
13395     + 21715 \__fp_exp_large:w {5297}{7951}{6443}{0315}{3251}{3576} \or:
13396     + 26058 \__fp_exp_large:w {4665}{6719}{0099}{3379}{5527}{2929} \or:
13397     + 30401 \__fp_exp_large:w {4108}{9724}{3326}{3186}{5271}{5665} \or:
13398     + 34744 \__fp_exp_large:w {3618}{6973}{3140}{0875}{3856}{4102} \or:
13399     + 39087 \__fp_exp_large:w {3186}{9209}{6113}{3900}{6705}{9685} \or:
13400     \fi:
13401     #1;
13402     \__fp_exp_large_iv:wN
13403 }
13404 \cs_new:Npn \__fp_exp_large_iv:wN #1; #2
13405 {
13406     \if_case:w #2 ~           \exp_after:wN \__fp_fixed_continue:wn \or:
13407     + 435 \__fp_exp_large:w {1970}{0711}{1401}{7046}{9938}{8888} \or:
13408     + 869 \__fp_exp_large:w {3881}{1801}{9428}{4368}{5764}{8232} \or:
13409     + 1303 \__fp_exp_large:w {7646}{2009}{8905}{4704}{8893}{1073} \or:
13410     + 1738 \__fp_exp_large:w {1506}{3559}{7005}{0524}{9009}{7592} \or:
13411     + 2172 \__fp_exp_large:w {2967}{6283}{8402}{3667}{0689}{6630} \or:
13412     + 2606 \__fp_exp_large:w {5846}{4389}{5650}{2114}{7278}{5046} \or:
13413     + 3041 \__fp_exp_large:w {1151}{7900}{5080}{6878}{2914}{4154} \or:
13414     + 3475 \__fp_exp_large:w {2269}{1083}{0850}{6857}{8724}{4002} \or:
13415     + 3909 \__fp_exp_large:w {4470}{3047}{3316}{5442}{6408}{6591} \or:
13416     \fi:
13417     #1;
13418     \__fp_exp_large_iii:wN
13419 }
13420 \cs_new:Npn \__fp_exp_large_iii:wN #1; #2
13421 {
13422     \if_case:w #2 ~           \exp_after:wN \__fp_fixed_continue:wn \or:
13423     + 44 \__fp_exp_large:w {2688}{1171}{4181}{6135}{4484}{1263} \or:
13424     + 87 \__fp_exp_large:w {7225}{9737}{6812}{5749}{2581}{7748} \or:
13425     + 131 \__fp_exp_large:w {1942}{4263}{9524}{1255}{9365}{8421} \or:
13426     + 174 \__fp_exp_large:w {5221}{4696}{8976}{4143}{9505}{8876} \or:
13427     + 218 \__fp_exp_large:w {1403}{5922}{1785}{2837}{4107}{3977} \or:
13428     + 261 \__fp_exp_large:w {3773}{0203}{0092}{9939}{8234}{0143} \or:
13429     + 305 \__fp_exp_large:w {1014}{2320}{5473}{5004}{5094}{5533} \or:
13430     + 348 \__fp_exp_large:w {2726}{3745}{7211}{2566}{5673}{6478} \or:
13431     + 391 \__fp_exp_large:w {7328}{8142}{2230}{7421}{7051}{8866} \or:
13432     \fi:

```

```

13433     #1;
13434     \__fp_exp_large_ii:wN
13435 }
13436 \cs_new:Npn \__fp_exp_large_ii:wN #1; #2
13437 {
13438     \if_case:w #2 ~          \exp_after:wN \__fp_fixed_continue:wn \or:
13439     + 5 \__fp_exp_large:w {2202}{6465}{7948}{0671}{6516}{9579} \or:
13440     + 9 \__fp_exp_large:w {4851}{6519}{5409}{7902}{7796}{9107} \or:
13441     + 14 \__fp_exp_large:w {1068}{6474}{5815}{2446}{2146}{9905} \or:
13442     + 18 \__fp_exp_large:w {2353}{8526}{6837}{0199}{8540}{7900} \or:
13443     + 22 \__fp_exp_large:w {5184}{7055}{2858}{7072}{4640}{8745} \or:
13444     + 27 \__fp_exp_large:w {1142}{0073}{8981}{5684}{2836}{6296} \or:
13445     + 31 \__fp_exp_large:w {2515}{4386}{7091}{9167}{0062}{6578} \or:
13446     + 35 \__fp_exp_large:w {5540}{6223}{8439}{3510}{0525}{7117} \or:
13447     + 40 \__fp_exp_large:w {1220}{4032}{9431}{7840}{8020}{0271} \or:
13448     \fi:
13449     #1;
13450     \__fp_exp_large_i:wN
13451 }
13452 \cs_new:Npn \__fp_exp_large_i:wN #1; #2
13453 {
13454     \if_case:w #2 ~          \exp_after:wN \__fp_fixed_continue:wn \or:
13455     + 1 \__fp_exp_large:w {2718}{2818}{2845}{9045}{2353}{6029} \or:
13456     + 1 \__fp_exp_large:w {7389}{0560}{9893}{0650}{2272}{3043} \or:
13457     + 2 \__fp_exp_large:w {2008}{5536}{9231}{8766}{7740}{9285} \or:
13458     + 2 \__fp_exp_large:w {5459}{8150}{0331}{4423}{9078}{1103} \or:
13459     + 3 \__fp_exp_large:w {1484}{1315}{9102}{5766}{0342}{1116} \or:
13460     + 3 \__fp_exp_large:w {4034}{2879}{3492}{7351}{2260}{8387} \or:
13461     + 4 \__fp_exp_large:w {1096}{6331}{5842}{8458}{5992}{6372} \or:
13462     + 4 \__fp_exp_large:w {2980}{9579}{8704}{1728}{2747}{4359} \or:
13463     + 4 \__fp_exp_large:w {8103}{0839}{2757}{5384}{0077}{1000} \or:
13464     \fi:
13465     #1;
13466     \__fp_exp_large_:wN
13467 }
13468 \cs_new:Npn \__fp_exp_large_:wN #1; #2
13469 {
13470     \if_case:w #2 ~          \exp_after:wN \__fp_fixed_continue:wn \or:
13471     + 1 \__fp_exp_large:w {1105}{1709}{1807}{5647}{6248}{1171} \or:
13472     + 1 \__fp_exp_large:w {1221}{4027}{5816}{0169}{8339}{2107} \or:
13473     + 1 \__fp_exp_large:w {1349}{8588}{0757}{6003}{1039}{8374} \or:
13474     + 1 \__fp_exp_large:w {1491}{8246}{9764}{1270}{3178}{2485} \or:
13475     + 1 \__fp_exp_large:w {1648}{7212}{7070}{0128}{1468}{4865} \or:
13476     + 1 \__fp_exp_large:w {1822}{1188}{0039}{0508}{9748}{7537} \or:
13477     + 1 \__fp_exp_large:w {2013}{7527}{0747}{0476}{5216}{2455} \or:
13478     + 1 \__fp_exp_large:w {2225}{5409}{2849}{2467}{6045}{7954} \or:
13479     + 1 \__fp_exp_large:w {2459}{6031}{1115}{6949}{6638}{0013} \or:
13480     \fi:
13481     #1;
13482     \__fp_exp_large_after:wwn

```

```

13483 }
13484 \cs_new:Npn \__fp_exp_large_after:wwn #1; #2; #3
13485 {
13486   \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
13487   \__fp_fixed_mul:wwn #1;
13488 }

```

(End definition for `__fp_exp_pos_large:NnnNwn` and others.)

31.3 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.

```

13489 \cs_new:cpn { __fp_ \iow_char:N \^ _o:ww }
13490   \s__fp \__fp_chk:w #1#2#3; \s__fp \__fp_chk:w #4#5#6;
13491   {
13492     \if_meaning:w 0 #4
13493       \__fp_case_return_o:Nw \c_one_fp
13494     \fi:

```

```

13495 \if_case:w #2 \exp_stop_f:
13496 \exp_after:wN \use_i:nn
13497 \or:
13498 \__fp_case_return_o:Nw \c_nan_fp
13499 \else:
13500 \exp_after:wN \__fp_pow_neg:www
13501 \tex_romannumeral:D -'0 \exp_after:wN \use:nn
13502 \fi:
13503 {
13504 \if_meaning:w 1 #1
13505 \exp_after:wN \__fp_pow_normal:ww
13506 \else:
13507 \exp_after:wN \__fp_pow_zero_or_inf:ww
13508 \fi:
13509 \s__fp \__fp_chk:w #1#2#3;
13510 }
13511 { \s__fp \__fp_chk:w #4#5#6; \s__fp \__fp_chk:w #1#2#3; }
13512 \s__fp \__fp_chk:w #4#5#6;
13513 }

```

(End definition for $\backslash_fp_^o:ww$.)

$\backslash_fp_pow_zero_or_inf:ww$

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.

```

13514 \cs_new:Npn \__fp_pow_zero_or_inf:ww \s__fp \__fp_chk:w #1#2; \s__fp \__fp_chk:w #3#4
13515 {
13516 \if_meaning:w 1 #4
13517 \__fp_case_return_same_o:w
13518 \fi:
13519 \if_meaning:w #1 #4
13520 \__fp_case_return_o:Nw \c_zero_fp
13521 \fi:
13522 \if_meaning:w 0 #1
13523 \__fp_case_use:nw
13524 {
13525 \__fp_division_by_zero_o:NNww \c_inf_fp ^
13526 \s__fp \__fp_chk:w #1 #2 ;
13527 }
13528 \else:
13529 \__fp_case_return_o:Nw \c_inf_fp
13530 \fi:
13531 \s__fp \__fp_chk:w #3#4
13532 }

```

(End definition for $\backslash_fp_pow_zero_or_inf:ww$.)

$\backslash_fp_pow_normal:ww$

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 .

```

13533 \cs_new:Npn \__fp_pow_normal:ww \s__fp \__fp_chk:w 1 #1#2#3; \s__fp \__fp_chk:w #4#5
13534 {
13535   \if_int_compare:w \__str_if_eq_x:nn { #2 #3 }
13536     { 1 {1000} {0000} {0000} {0000} } = \c_zero
13537     \if_int_compare:w #4 #1 = 32 \exp_stop_f:
13538       \exp_after:wN \__fp_case_return_ii_o:ww
13539     \fi:
13540     \__fp_case_return_o:Nww \c_one_fp
13541   \fi:
13542   \if_case:w #4 \exp_stop_f:
13543   \or:
13544     \exp_after:wN \__fp_pow_npos:Nww
13545     \exp_after:wN #5
13546   \or:
13547     \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
13548     \if_int_compare:w #2 > \c_zero
13549       \exp_after:wN \__fp_case_return_o:Nww
13550       \exp_after:wN \c_inf_fp
13551     \else:
13552       \exp_after:wN \__fp_case_return_o:Nww
13553       \exp_after:wN \c_zero_fp
13554     \fi:
13555   \or:
13556     \__fp_case_return_ii_o:ww
13557   \fi:
13558   \s__fp \__fp_chk:w 1 #1 {#2} #3 ;
13559   \s__fp \__fp_chk:w #4 #5
13560 }

```

(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.

```

13561 \cs_new:Npn \__fp_pow_npos:Nww #1 \s__fp \__fp_chk:w 1#2#3

```

```

13562 {
13563   \exp_after:wN \__fp_sanitize:Nw
13564   \exp_after:wN 0
13565   \__int_value:w
13566   \if:w #1 \if_int_compare:w #3 > \c_zero 0 \else: 2 \fi:
13567     \exp_after:wN \__fp_pow_npos_aux:NNnw
13568     \exp_after:wN +
13569     \exp_after:wN \__fp_fixed_to_float:wN
13570   \else:
13571     \exp_after:wN \__fp_pow_npos_aux:NNnw
13572     \exp_after:wN -
13573     \exp_after:wN \__fp_fixed_inv_to_float:wN
13574   \fi:
13575   {#3}
13576 }

```

(End definition for __fp_pow_npos:Nnw.)

__fp_pow_npos_aux:NNnw The first argument is the conversion function from fixed point to float. Then comes an exponent and the 4 brace groups of x , followed by b . Compute $-\ln(x)$.

```

13577 \cs_new:Npn \__fp_pow_npos_aux:NNnw #1#2#3#4#5; \s_fp \__fp_chk:w 1#6#7#8;
13578 {
13579   #1
13580   \__int_eval:w
13581   \__fp_ln_significand:NNNNnnN #4#5
13582   \__fp_pow_exponent:wnN {#3}
13583   \__fp_fixed_mul:wwn #8 {0000}{0000} ;
13584   \__fp_pow_B:wwN #7;
13585   #1 #2 0 % fixed_to_float:wN
13586 }
13587 \cs_new:Npn \__fp_pow_exponent:wnN #1; #2
13588 {
13589   \if_int_compare:w #2 > \c_zero
13590     \exp_after:wN \__fp_pow_exponent:Nwnnnnw % n\ln(10) - (-\ln(x))
13591     \exp_after:wN +
13592   \else:
13593     \exp_after:wN \__fp_pow_exponent:Nwnnnnw % -( |n|\ln(10) + (-\ln(x)) )
13594     \exp_after:wN -
13595   \fi:
13596   #2; #1;
13597 }
13598 \cs_new:Npn \__fp_pow_exponent:Nwnnnnw #1#2; #3#4#5#6#7#8;
13599 { %^A todo: use that in ln.
13600   \exp_after:wN \__fp_fixed_mul_after:wwn
13601   \int_use:N \__int_eval:w \c__fp_leading_shift_int
13602   \exp_after:wN \__fp_pack:NNNNNw
13603   \int_use:N \__int_eval:w \c__fp_middle_shift_int
13604   #1#2*23025 - #1 #3
13605   \exp_after:wN \__fp_pack:NNNNNw
13606   \int_use:N \__int_eval:w \c__fp_middle_shift_int

```

```

13607         #1 #2*8509 - #1 #4
13608         \exp_after:wN \__fp_pack:NNNNw
13609         \int_use:N \__int_eval:w \c__fp_middle_shift_int
13610         #1 #2*2994 - #1 #5
13611         \exp_after:wN \__fp_pack:NNNNw
13612         \int_use:N \__int_eval:w \c__fp_middle_shift_int
13613         #1 #2*0456 - #1 #6
13614         \exp_after:wN \__fp_pack:NNNNw
13615         \int_use:N \__int_eval:w \c__fp_trailing_shift_int
13616         #1 #2*8401 - #1 #7
13617         #1 ( #2*7991 - #8 ) / 1 0000 ; ;
13618     }
13619     \cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
13620     {
13621         \if_int_compare:w #7 < \c_zero
13622         \exp_after:wN \__fp_pow_C_neg:w \__int_value:w -
13623         \else:
13624         \if_int_compare:w #7 < 22 \exp_stop_f:
13625         \exp_after:wN \__fp_pow_C_pos:w \__int_value:w
13626         \else:
13627         \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
13628         \fi:
13629         \fi:
13630         #7 \exp_after:wN ;
13631         \int_use:N \__int_eval:w 10 0000 + #1 \__int_eval_end:
13632         #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^A todo: how many 0?
13633     }
13634     \cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
13635     {
13636         + \c_two * \c__fp_max_exponent_int
13637         \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl ;
13638     }
13639     \cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
13640     {
13641         \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
13642         \prg_replicate:nn {#1} {0}
13643     }
13644     \cs_new:Npn \__fp_pow_C_pos:w #1; 1
13645     { \__fp_pow_C_pos_loop:wN #1; }
13646     \cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
13647     {
13648         \if_meaning:w 0 #1
13649         \exp_after:wN \__fp_pow_C_pack:w
13650         \exp_after:wN #2
13651         \else:
13652         \if_meaning:w 0 #2
13653         \exp_after:wN \__fp_pow_C_pos_loop:wN \__int_value:w
13654         \else:
13655         \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
13656         \fi:

```



```

13657     \__int_eval:w #1 - \c_one \exp_after:wN ;
13658     \fi:
13659   }
13660   \cs_new:Npn \__fp_pow_C_pack:w
13661     { \exp_after:wN \__fp_exp_large_v:wN \c__fp_one_fixed_tl ; }
(End definition for \__fp_pow_npos_aux:NNnw.)

```

`__fp_pow_neg:www`
`__fp_pow_neg_aux:wNN`

This function is followed by three floating point numbers: a^b , $a \in [-\infty, -0]$, and b . If b is an even integer (case -1), $a^b = a^b$. If b is an odd integer (case 0), $a^b = -a^b$, obtained by a call to `__fp_pow_neg_aux:wNN`. Otherwise, the sign is undefined. This is invalid, unless a^b turns out to be $+0$ or `nan`, in which case we return that as a^b . In particular, since the underflow detection occurs before `__fp_pow_neg:www` is called, $(-0.1)**(12345.6)$ will give $+0$ rather than complaining that the sign is not defined.

```

13662   \cs_new:Npn \__fp_pow_neg:www \s__fp \__fp_chk:w #1#2; #3; #4;
13663   {
13664     \if_case:w \__fp_pow_neg_case:w #4 ;
13665     \exp_after:wN \__fp_pow_neg_aux:wNN
13666     \or:
13667       \if_int_compare:w \__int_eval:w #1 / \c_two = \c_one
13668         \__fp_invalid_operation_o:Nww ^ #3; #4;
13669         \tex_romannumeral:D -'0
13670         \exp_after:wN \exp_after:wN
13671         \exp_after:wN \__fp_use_none_until_s:w
13672       \fi:
13673     \fi:
13674     \__fp_exp_after_o:w
13675     \s__fp \__fp_chk:w #1#2;
13676   }
13677   \cs_new:Npn \__fp_pow_neg_aux:wNN #1 \s__fp \__fp_chk:w #2#3
13678   {
13679     \exp_after:wN \__fp_exp_after_o:w
13680     \exp_after:wN \s__fp
13681     \exp_after:wN \__fp_chk:w
13682     \exp_after:wN #2
13683     \int_use:N \__int_eval:w \c_two - #3 \__int_eval_end:
13684   }

```

(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:NNNNNNnw`

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`.

```

13685 \cs_new:Npn \__fp_pow_neg_case:w \s__fp \__fp_chk:w #1#2#3;
13686 {
13687   \if_case:w #1 \exp_stop_f:
13688     \c_minus_one
13689   \or:   \__fp_pow_neg_case_aux:nnnnn #3
13690   \else: \c_one
13691   \fi:
13692 }
13693 \cs_new:Npn \__fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
13694 {
13695   \if_int_compare:w #1 > \c_eight
13696   \if_int_compare:w #1 > \c_sixteen
13697     \c_minus_one
13698   \else:
13699     \exp_after:wN \exp_after:wN
13700     \exp_after:wN \__fp_pow_neg_case_aux:NNNNNNNNw
13701     \prg_replicate:nn { \c_sixteen - #1 } { 0 } #4#5 ;
13702   \fi:
13703   \else:
13704     \if_int_compare:w #1 > \c_zero
13705     \if_int_compare:w #4#5 = \c_zero
13706       \exp_after:wN \exp_after:wN
13707       \exp_after:wN \__fp_pow_neg_case_aux:NNNNNNNNw
13708       \prg_replicate:nn { \c_eight - #1 } { 0 } #2#3 ;
13709     \else:
13710       \c_one
13711     \fi:
13712   \else:
13713     \c_one
13714   \fi:
13715   \fi:
13716 }
13717 \cs_new:Npn \__fp_pow_neg_case_aux:NNNNNNNNw #1#2#3#4#5#6#7#8#9;
13718 {
13719   \if_int_compare:w 0 #9 = \c_zero
13720   \if_int_odd:w #8 \exp_stop_f:
13721     \c_zero
13722   \else:
13723     \c_minus_one
13724   \fi:
13725   \else:
13726     \c_one
13727   \fi:
13728 }
(End definition for \__fp_pow_neg_case:w, \__fp_pow_neg_case_aux:nnnnn, and \__fp_pow_neg_case_aux:NNNNNNNNw.)
13729 </initex | package>

```

32 l3fp-trig Implementation

13730 $\langle *initex | package \rangle$

13731 $\langle @@=fp \rangle$

32.1 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.

32.1.1 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 $\sin(x) = \#3 \sin|x|$.

```

13732 \cs_new:Npn \__fp_sin_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
13733 {
13734   \if_case:w #2 \exp_stop_f:
13735     \__fp_case_return_same_o:w
13736   \or: \__fp_case_use:nw
13737     {
13738       \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
13739       \__fp_ep_to_float:wwN #3 \c_zero
13740     }
13741   \or: \__fp_case_use:nw
13742     { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
13743   \else: \__fp_case_return_same_o:w
13744   \fi:

```

```

13745     \s__fp __fp_chk:w #2 #3 #4;
13746 }

```

(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 $\cos(x) = +\sin(\pi/2 + |x|)$.

```

13747 \cs_new:Npn __fp_cos_o:w #1 \s__fp __fp_chk:w #2#3; @
13748 {
13749   \if_case:w #2 \exp_stop_f:
13750     __fp_case_return_o:Nw \c_one_fp
13751   \or:   __fp_case_use:nw
13752     {
13753       __fp_trig:NNNNNwn #1 __fp_sin_series_o:NNwww
13754       __fp_ep_to_float:wwN 0 \c_two
13755     }
13756   \or:   __fp_case_use:nw
13757     { __fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
13758   \else: __fp_case_return_same_o:w
13759   \fi:
13760   \s__fp __fp_chk:w #2 #3;
13761 }

```

(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 $\csc(x) = \#3(\sin|x|)^{-1}$.

```

13762 \cs_new:Npn __fp_csc_o:w #1 \s__fp __fp_chk:w #2#3#4; @
13763 {
13764   \if_case:w #2 \exp_stop_f:
13765     __fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
13766   \or:   __fp_case_use:nw
13767     {
13768       __fp_trig:NNNNNwn #1 __fp_sin_series_o:NNwww
13769       __fp_ep_inv_to_float:wwN #3 \c_zero
13770     }
13771   \or:   __fp_case_use:nw
13772     { __fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
13773   \else: __fp_case_return_same_o:w
13774   \fi:
13775   \s__fp __fp_chk:w #2 #3 #4;
13776 }

```

(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 $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```

13777 \cs_new:Npn \_fp_sec_o:w #1 \s\_fp \_fp_chk:w #2#3; @
13778 {
13779   \if_case:w #2 \exp_stop_f:
13780     \_fp_case_return_o:Nw \c_one_fp
13781   \or: \_fp_case_use:nw
13782     {
13783       \_fp_trig:NNNNNwn #1 \_fp_sin_series_o:NNwww
13784       \_fp_ep_inv_to_float:wwN 0 \c_two
13785     }
13786   \or: \_fp_case_use:nw
13787     { \_fp_invalid_operation_o:fw { #1 { sec } { secd } } }
13788   \else: \_fp_case_return_same_o:w
13789   \fi:
13790   \s\_fp \_fp_chk:w #2 #3;
13791 }

```

(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.

```

13792 \cs_new:Npn \_fp_tan_o:w #1 \s\_fp \_fp_chk:w #2#3#4; @
13793 {
13794   \if_case:w #2 \exp_stop_f:
13795     \_fp_case_return_same_o:w
13796   \or: \_fp_case_use:nw
13797     {
13798       \_fp_trig:NNNNNwn #1
13799       \_fp_tan_series_o:NNwww 0 #3 \c_one
13800     }
13801   \or: \_fp_case_use:nw
13802     { \_fp_invalid_operation_o:fw { #1 { tan } { tand } } }
13803   \else: \_fp_case_return_same_o:w
13804   \fi:
13805   \s\_fp \_fp_chk:w #2 #3 #4;
13806 }

```

(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.

```

13807 \cs_new:Npn \__fp_cot_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
13808 {
13809   \if_case:w #2 \exp_stop_f:
13810     \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
13811   \or:   \__fp_case_use:nw
13812     {
13813       \__fp_trig:NNNNwn #1
13814       \__fp_tan_series_o:NNwww 2 #3 \c_three
13815     }
13816   \or:   \__fp_case_use:nw
13817     { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
13818   \else: \__fp_case_return_same_o:w
13819   \fi:
13820   \s__fp \__fp_chk:w #2 #3 #4;
13821 }
13822 \cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:
13823 {
13824   \fi:
13825   \token_if_eq_meaning:NNTF 0 #1
13826   { \exp_args:Nnf \__fp_division_by_zero_o:Nnw \c_inf_fp }
13827   { \exp_args:Nnf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
13828   {#2}
13829 }
(End definition for \__fp_cot_o:w.)

```

32.1.2 Distinguishing small and large arguments

`__fp_trig:NNNNwn` 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.

```

13830 \cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
13831 {
13832   \exp_after:wN #2
13833   \exp_after:wN #3
13834   \exp_after:wN #4
13835   \int_use:N \__int_eval:w #5
13836   \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
13837   \if_int_compare:w #7 > #1 \c_zero \c_one
13838   #1 \__fp_trig_large:ww \__fp_trigd_large:ww
13839   \else:
13840   #1 \__fp_trig_small:ww \__fp_trigd_small:ww
13841   \fi:
13842   #7,#8{0000}{0000};
13843 }

```

(End definition for __fp_trig:NNNNNwn.)

32.1.3 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.

```

13844 \cs_new:Npn \__fp_trig_small:ww #1,#2;
13845 { \__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.

```

13846 \cs_new:Npn \__fp_trigd_small:ww #1,#2;
13847 {
13848   \__fp_ep_mul_raw:wwwn
13849   -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
13850   \__fp_trig_small:ww
13851 }

```

(End definition for __fp_trigd_small:ww.)

32.1.4 Argument reduction in degrees

`__fp_trigd_large:ww` Note that $25 \times 360 = 9000$, so $10^{k+1} \equiv 10^k \pmod{360}$ for $k \geq 3$. When the exponent #1 is very large, we can thus safely replace it by 22 (or even 19). We turn the floating point number into a fixed point number with two blocks of 8 digits followed by five blocks of 4 digits. The original float is $100 \times \langle block_1 \rangle \cdots \langle block_3 \rangle . \langle block_4 \rangle \cdots \langle block_7 \rangle$, or is equal to it modulo 360 if the exponent #1 is very large. The first auxiliary finds $\langle block_1 \rangle + \langle block_2 \rangle \pmod{9}$, a single digit, and prepends it to the 4 digits of $\langle block_3 \rangle$. It also unpacks $\langle block_4 \rangle$ and grabs the 4 digits of $\langle block_7 \rangle$. The second auxiliary grabs the $\langle block_3 \rangle$ plus any contribution from the first two blocks as #1, the first digit of $\langle block_4 \rangle$ (just after the

decimal point in hundreds of degrees) as #2, and the three other digits as #3. It finds the quotient and remainder of #1#2 modulo 9, adds twice the quotient to the integer expression for the octant, and places the remainder (between 0 and 8) before #3 to form a new $\langle block_4 \rangle$. The resulting fixed point number is $x \in [0, 0.9]$. If $x \geq 0.45$, we add 1 to the octant and feed $0.9 - x$ with an exponent of 2 (to compensate the fact that we are working in units of hundreds of degrees rather than degrees) to `_fp_trigd_small:ww`. Otherwise, we feed it x with an exponent of 2. The third auxiliary also discards digits which were not packed into the various $\langle blocks \rangle$. Since the original exponent #1 is at least 2, those are all 0 and no precision is lost (#6 and #7 are four 0 each).

```

13852 \cs_new:Npn \_fp_trigd_large:ww #1, #2#3#4#5#6#7;
13853 {
13854   \exp_after:wN \_fp_pack_eight:wNNNNNNNN
13855   \exp_after:wN \_fp_pack_eight:wNNNNNNNN
13856   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
13857   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
13858   \exp_after:wN \_fp_trigd_large_auxi:nnnnwNNNN
13859   \exp_after:wN ;
13860   \tex_romannumeral:D -'0
13861   \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
13862   #2#3#4#5#6#7 0000 0000 0000 !
13863 }
13864 \cs_new:Npn \_fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
13865 {
13866   \exp_after:wN \_fp_trigd_large_auxii:wNw
13867   \int_use:N \_int_eval:w #1 + #2
13868   - (#1 + #2 - \c_four) / \c_nine * \c_nine \_int_eval_end:
13869   #3;
13870   #4; #5{#6#7#8#9};
13871 }
13872 \cs_new:Npn \_fp_trigd_large_auxii:wNw #1; #2#3;
13873 {
13874   + (#1#2 - \c_four) / \c_nine * \c_two
13875   \exp_after:wN \_fp_trigd_large_auxiii:www
13876   \int_use:N \_int_eval:w #1#2
13877   - (#1#2 - \c_four) / \c_nine * \c_nine \_int_eval_end: #3 ;
13878 }
13879 \cs_new:Npn \_fp_trigd_large_auxiii:www #1; #2; #3!
13880 {
13881   \if_int_compare:w #1 < 4500 \exp_stop_f:
13882     \exp_after:wN \_fp_use_i_until_s:nw
13883     \exp_after:wN \_fp_fixed_continue:wn
13884   \else:
13885     + \c_one
13886   \fi:
13887   \_fp_fixed_sub:wwn {9000}{0000}{0000}{0000}{0000}{0000};
13888   {#1}#2{0000}{0000};
13889   { \_fp_trigd_small:ww 2, }
13890 }

```

(End definition for `_fp_trigd_large:ww` and others.)

32.1.5 Argument reduction in radians

Arguments greater or equal to 1 need to be reduced to a range where we only need a few terms of the Taylor series. We reduce to the range $[0, 2\pi]$ by subtracting multiples of 2π , then to the smaller range $[0, \pi/2]$ by subtracting multiples of $\pi/2$ (keeping track of how many times $\pi/2$ is subtracted), then to $[0, \pi/4]$ by mapping $x \rightarrow \pi/2 - x$ if appropriate. When the argument is very large, say, 10^{100} , an equally large multiple of 2π must be subtracted, hence we must work with a very good approximation of 2π in order to get a sensible remainder modulo 2π .

Specifically, we multiply the argument by an approximation of $1/(2\pi)$ with 10048 digits, then discard the integer part of the result, keeping 52 digits of the fractional part. From the fractional part of $x/(2\pi)$ we deduce the octant (quotient of the first three digits by 125). We then multiply by 8 or -8 (the latter when the octant is odd), ignore any integer part (related to the octant), and convert the fractional part to an extended precision number, before multiplying by $\pi/4$ to convert back to a value in radians in $[0, \pi/4]$.

It is possible to prove that given the precision of floating points and their range of exponents, the 52 digits may start at most with 24 zeros. The 5 last digits are affected by carries from computations which are not done, hence we are left with at least $52 - 24 - 5 = 23$ significant digits, enough to round correctly up to $0.6 \cdot \text{ulp}$ in all cases.

`_fp_trig_inverse_two_pi:` This macro expands to `,,!` or `,!` followed by 10112 decimals of $10^{-16}/(2\pi)$. The number of decimals we really need is the maximum exponent plus the number of digits we will need later, 52, plus 12 (4 – 1 groups of 4 digits). We store the decimals as a control sequence name, and convert it to a token list when required: strings take up less memory than their token list representation.

```

13891 \cs_new_nopar:Npx \_fp_trig_inverse_two_pi:
13892 {
13893   \exp_not:n { \exp_after:wN \use_none:n \token_to_str:N }
13894   \cs:w , , !
13895   0000000000000000159154943091895335768883763372514362034459645740 ~
13896   4564487476673440588967976342265350901138027662530859560728427267 ~
13897   5795803689291184611457865287796741073169983922923996693740907757 ~
13898   3077746396925307688717392896217397661693362390241723629011832380 ~
13899   1142226997557159404618900869026739561204894109369378440855287230 ~
13900   9994644340024867234773945961089832309678307490616698646280469944 ~
13901   8652187881574786566964241038995874139348609983868099199962442875 ~
13902   5851711788584311175187671605465475369880097394603647593337680593 ~
13903   0249449663530532715677550322032477781639716602294674811959816584 ~
13904   0606016803035998133911987498832786654435279755070016240677564388 ~
13905   8495713108801221993761476813777647378906330680464579784817613124 ~
13906   2731406996077502450029775985708905690279678513152521001631774602 ~
13907   0924811606240561456203146484089248459191435211575407556200871526 ~
13908   6068022171591407574745827225977462853998751553293908139817724093 ~
13909   5825479707332871904069997590765770784934703935898280871734256403 ~
13910   6689511662545705943327631268650026122717971153211259950438667945 ~
13911   0376255608363171169525975812822494162333431451061235368785631136 ~
13912   3669216714206974696012925057833605311960859450983955671870995474 ~

```

13913 6510431623815517580839442979970999505254387566129445883306846050 ~
13914 7852915151410404892988506388160776196993073410389995786918905980 ~
13915 9373777206187543222718930136625526123878038753888110681406765434 ~
13916 0828278526933426799556070790386060352738996245125995749276297023 ~
13917 5940955843011648296411855777124057544494570217897697924094903272 ~
13918 9477021664960356531815354400384068987471769158876319096650696440 ~
13919 4776970687683656778104779795450353395758301881838687937766124814 ~
13920 9530599655802190835987510351271290432315804987196868777594656634 ~
13921 6221034204440855497850379273869429353661937782928735937843470323 ~
13922 0237145837923557118636341929460183182291964165008783079331353497 ~
13923 7909974586492902674506098936890945883050337030538054731232158094 ~
13924 3197676032283131418980974982243833517435698984750103950068388003 ~
13925 9786723599608024002739010874954854787923568261139948903268997427 ~
13926 0834961149208289037767847430355045684560836714793084567233270354 ~
13927 8539255620208683932409956221175331839402097079357077496549880868 ~
13928 6066360968661967037474542102831219251846224834991161149566556037 ~
13929 9696761399312829960776082779901007830360023382729879085402387615 ~
13930 5744543092601191005433799838904654921248295160707285300522721023 ~
13931 6017523313173179759311050328155109373913639645305792607180083617 ~
13932 9548767246459804739772924481092009371257869183328958862839904358 ~
13933 6866663975673445140950363732719174311388066383072592302759734506 ~
13934 0548212778037065337783032170987734966568490800326988506741791464 ~
13935 6835082816168533143361607309951498531198197337584442098416559541 ~
13936 5225064339431286444038388356150879771645017064706751877456059160 ~
13937 8716857857939226234756331711132998655941596890719850688744230057 ~
13938 5191977056900382183925622033874235362568083541565172971088117217 ~
13939 9593683256488518749974870855311659830610139214454460161488452770 ~
13940 2511411070248521739745103866736403872860099674893173561812071174 ~
13941 0478899368886556923078485023057057144063638632023685201074100574 ~
13942 8592281115721968003978247595300166958522123034641877365043546764 ~
13943 6456565971901123084767099309708591283646669191776938791433315566 ~
13944 5066981321641521008957117286238426070678451760111345080069947684 ~
13945 2235698962488051577598095339708085475059753626564903439445420581 ~
13946 7886435683042000315095594743439252544850674914290864751442303321 ~
13947 3324569511634945677539394240360905438335528292434220349484366151 ~
13948 4663228602477666660495314065734357553014090827988091478669343492 ~
13949 2737602634997829957018161964321233140475762897484082891174097478 ~
13950 2637899181699939487497715198981872666294601830539583275209236350 ~
13951 6853889228468247259972528300766856937583659722919824429747406163 ~
13952 8183113958306744348516928597383237392662402434501997809940402189 ~
13953 6134834273613676449913827154166063424829363741850612261086132119 ~
13954 9863346284709941839942742955915628333990480382117501161211667205 ~
13955 1912579303552929241134403116134112495318385926958490443846807849 ~
13956 0973982808855297045153053991400988698840883654836652224668624087 ~
13957 2540140400911787421220452307533473972538149403884190586842311594 ~
13958 6322744339066125162393106283195323883392131534556381511752035108 ~
13959 7459558201123754359768155340187407394340363397803881721004531691 ~
13960 8295194879591767395417787924352761740724605939160273228287946819 ~
13961 3649128949714953432552723591659298072479985806126900733218844526 ~
13962 7943350455801952492566306204876616134365339920287545208555344144 ~

13963 0990512982727454659118132223284051166615650709837557433729548631 ~
13964 2041121716380915606161165732000083306114606181280326258695951602 ~
13965 4632166138576614804719932707771316441201594960110632830520759583 ~
13966 4850305079095584982982186740289838551383239570208076397550429225 ~
13967 9847647071016426974384504309165864528360324933604354657237557916 ~
13968 1366324120457809969715663402215880545794313282780055246132088901 ~
13969 8742121092448910410052154968097113720754005710963406643135745439 ~
13970 9159769435788920793425617783022237011486424925239248728713132021 ~
13971 7667360756645598272609574156602343787436291321097485897150713073 ~
13972 9104072643541417970572226547980381512759579124002534468048220261 ~
13973 7342299001020483062463033796474678190501811830375153802879523433 ~
13974 4195502135689770912905614317878792086205744999257897569018492103 ~
13975 2420647138519113881475640209760554895793785141404145305151583964 ~
13976 2823265406020603311891586570272086250269916393751527887360608114 ~
13977 5569484210322407772727421651364234366992716340309405307480652685 ~
13978 0930165892136921414312937134106157153714062039784761842650297807 ~
13979 8606266969960809184223476335047746719017450451446166382846208240 ~
13980 8673595102371302904443779408535034454426334130626307459513830310 ~
13981 2293146934466832851766328241515210179422644395718121717021756492 ~
13982 1964449396532222187658488244511909401340504432139858628621083179 ~
13983 3939608443898019147873897723310286310131486955212620518278063494 ~
13984 5711866277825659883100535155231665984394090221806314454521212978 ~
13985 9734471488741258268223860236027109981191520568823472398358013366 ~
13986 0683786328867928619732367253606685216856320119489780733958419190 ~
13987 6659583867852941241871821727987506103946064819585745620060892122 ~
13988 8416394373846549589932028481236433466119707324309545859073361878 ~
13989 6290631850165106267576851216357588696307451999220010776676830946 ~
13990 9814975622682434793671310841210219520899481912444048751171059184 ~
13991 4139907889455775184621619041530934543802808938628073237578615267 ~
13992 7971143323241969857805637630180884386640607175368321362629671224 ~
13993 2609428540110963218262765120117022552929289655594608204938409069 ~
13994 0760692003954646191640021567336017909631872891998634341086903200 ~
13995 5796637103128612356988817640364252540837098108148351903121318624 ~
13996 7228181050845123690190646632235938872454630737272808789830041018 ~
13997 9485913673742589418124056729191238003306344998219631580386381054 ~
13998 2457893450084553280313511884341007373060595654437362488771292628 ~
13999 9807423539074061786905784443105274262641767830058221486462289361 ~
14000 9296692992033046693328438158053564864073184440599549689353773183 ~
14001 6726613130108623588021288043289344562140479789454233736058506327 ~
14002 0439981932635916687341943656783901281912202816229500333012236091 ~
14003 8587559201959081224153679499095448881099758919890811581163538891 ~
14004 6339402923722049848375224236209100834097566791710084167957022331 ~
14005 7897107102928884897013099533995424415335060625843921452433864640 ~
14006 3432440657317477553405404481006177612569084746461432976543900008 ~
14007 3826521145210162366431119798731902751191441213616962045693602633 ~
14008 6102355962140467029012156796418735746835873172331004745963339773 ~
14009 2477044918885134415363760091537564267438450166221393719306748706 ~
14010 2881595464819775192207710236743289062690709117919412776212245117 ~
14011 2354677115640433357720616661564674474627305622913332030953340551 ~
14012 3841718194605321501426328000879551813296754972846701883657425342 ~

```

14013 5016994231069156343106626043412205213831587971115075454063290657 ~
14014 0248488648697402872037259869281149360627403842332874942332178578 ~
14015 7750735571857043787379693402336902911446961448649769719434527467 ~
14016 4429603089437192540526658890710662062575509930379976658367936112 ~
14017 8137451104971506153783743579555867972129358764463093757203221320 ~
14018 2460565661129971310275869112846043251843432691552928458573495971 ~
14019 5042565399302112184947232132380516549802909919676815118022483192 ~
14020 5127372199792134331067642187484426215985121676396779352982985195 ~
14021 8545392106957880586853123277545433229161989053189053725391582222 ~
14022 9232597278133427818256064882333760719681014481453198336237910767 ~
14023 1255017528826351836492103572587410356573894694875444694018175923 ~
14024 0609370828146501857425324969212764624247832210765473750568198834 ~
14025 5641035458027261252285503154325039591848918982630498759115406321 ~
14026 0354263890012837426155187877318375862355175378506956599570028011 ~
14027 5841258870150030170259167463020842412449128392380525772514737141 ~
14028 2310230172563968305553583262840383638157686828464330456805994018 ~
14029 7001071952092970177990583216417579868116586547147748964716547948 ~
14030 8312140431836079844314055731179349677763739898930227765607058530 ~
14031 4083747752640947435070395214524701683884070908706147194437225650 ~
14032 2823145872995869738316897126851939042297110721350756978037262545 ~
14033 8141095038270388987364516284820180468288205829135339013835649144 ~
14034 3004015706509887926715417450706686888783438055583501196745862340 ~
14035 8059532724727843829259395771584036885940989939255241688378793572 ~
14036 7967951654076673927031256418760962190243046993485989199060012977 ~
14037 7469214532970421677817261517850653008552559997940209969455431545 ~
14038 2745856704403686680428648404512881182309793496962721836492935516 ~
14039 2029872469583299481932978335803459023227052612542114437084359584 ~
14040 9443383638388317751841160881711251279233374577219339820819005406 ~
14041 3292937775306906607415304997682647124407768817248673421685881509 ~
14042 9133422075930947173855159340808957124410634720893194912880783576 ~
14043 3115829400549708918023366596077070927599010527028150868897828549 ~
14044 4340372642729262103487013992868853550062061514343078665396085995 ~
14045 0058714939141652065302070085265624074703660736605333805263766757 ~
14046 2018839497277047222153633851135483463624619855425993871933367482 ~
14047 0422097449956672702505446423243957506869591330193746919142980999 ~
14048 3424230550172665212092414559625960554427590951996824313084279693 ~
14049 7113207021049823238195747175985519501864630940297594363194450091 ~
14050 9150616049228764323192129703446093584259267276386814363309856853 ~
14051 2786024332141052330760658841495858718197071242995959226781172796 ~
14052 4438853796763139274314227953114500064922126500133268623021550837 ~
14053 \cs_end:
14054 }

```

(End definition for _fp_trig_inverse_two_pi:.)

```

\_fp_trig_large:ww
\_fp_trig_large_auxi:wwwww
\_fp_trig_large_auxii:ww
  \_fp_trig_large_auxiii:wnnnnnnnn
\_fp_trig_large_auxiv:wN

```

The exponent #1 is between 1 and 10000. We discard the integer part of $10^{\#1-16}/(2\pi)$, that is, the first #1 digits of $10^{-16}/(2\pi)$, because it yields an integer contribution to $x/(2\pi)$. The auxii auxiliary discards 64 digits at a time thanks to spaces inserted in the result of _fp_trig_inverse_two_pi:, while auxiii discards 8 digits at a time, and auxiv discards digits one at a time. Then 64 digits are packed into groups of 4 and the

auxv auxiliary is called.

```

14055 \cs_new:Npn \__fp_trig_large:ww #1, #2#3#4#5#6;
14056 {
14057   \exp_after:wN \__fp_trig_large_auxi:wwwww
14058   \int_use:N \__int_eval:w (#1 - 32) / 64 \exp_after:wN ,
14059   \int_use:N \__int_eval:w (#1 - 4) / 8 \exp_after:wN ,
14060   \__int_value:w #1 \__fp_trig_inverse_two_pi: ;
14061   {#2}{#3}{#4}{#5} ;
14062 }
14063 \cs_new:Npn \__fp_trig_large_auxi:wwwww #1, #2, #3, #4!
14064 {
14065   \prg_replicate:nn {#1} { \__fp_trig_large_auxii:ww }
14066   \prg_replicate:nn { #2 - #1 * \c_eight }
14067   { \__fp_trig_large_auxiii:wNNNNNNNN }
14068   \prg_replicate:nn { #3 - #2 * \c_eight }
14069   { \__fp_trig_large_auxiv:wN }
14070   \prg_replicate:nn { \c_eight } { \__fp_pack_twice_four:wNNNNNNNN }
14071   \__fp_trig_large_auxv:www
14072   ;
14073 }
14074 \cs_new:Npn \__fp_trig_large_auxii:ww #1; #2 ~ { #1; }
14075 \cs_new:Npn \__fp_trig_large_auxiii:wNNNNNNNN
14076   #1; #2#3#4#5#6#7#8#9 { #1; }
14077 \cs_new:Npn \__fp_trig_large_auxiv:wN #1; #2 { #1; }
(End definition for \__fp_trig_large:ww and others.)

```

```

\__fp_trig_large_auxv:www
\__fp_trig_large_auxvi:wNNNNNNNN
\__fp_trig_large_pack:NNNNw

```

First come the first 64 digits of the fractional part of $10^{1-16}/(2\pi)$, arranged in 16 blocks of 4, and ending with a semicolon. Then some more digits of the same fractional part, ending with a semicolon, then 4 blocks of 4 digits holding the significand of the original argument. Multiply the 16-digit significand with the 64-digit fractional part: the auxvi auxiliary receives the significand as #2#3#4#5 and 16 digits of the fractional part as #6#7#8#9, and computes one step of the usual ladder of pack functions we use for multiplication (see *e.g.*, __fp_fixed_mul:wwn), then discards one block of the fractional part to set things up for the next step of the ladder. We perform 13 such steps, replacing the last middle shift by the appropriate trailing shift, then discard the significand and remaining 3 blocks from the fractional part, as there are not enough digits to compute any more step in the ladder. The last semicolon closes the ladder, and we return control to the auxvii auxiliary.

```

14078 \cs_new:Npn \__fp_trig_large_auxv:www #1; #2; #3;
14079 {
14080   \exp_after:wN \__fp_use_i_until_s:nw
14081   \exp_after:wN \__fp_trig_large_auxvii:w
14082   \int_use:N \__int_eval:w \c__fp_leading_shift_int
14083   \prg_replicate:nn { \c_thirteen }
14084   { \__fp_trig_large_auxvi:wNNNNNNNN }
14085   + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
14086   \__fp_use_i_until_s:nw
14087   ; #3 #1 ; ;

```

```

14088 }
14089 \cs_new:Npn \__fp_trig_large_auxvi:wnnnnnnnn #1; #2#3#4#5#6#7#8#9
14090 {
14091   \exp_after:wN \__fp_trig_large_pack:NNNNNw
14092   \int_use:N \__int_eval:w \c__fp_middle_shift_int
14093     + #2*#9 + #3*#8 + #4*#7 + #5*#6
14094     #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
14095 }
14096 \cs_new:Npn \__fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
14097 { + #1#2#3#4#5 ; #6 }
(End definition for \__fp_trig_large_auxv:www, \__fp_trig_large_auxvi:wnnnnnnnn, and \__fp_trig_large_pack:NNNNNw.)

```

```

\__fp_trig_large_auxvii:w
\__fp_trig_large_auxviii:w
\__fp_trig_large_auxix:Nw
\__fp_trig_large_auxx:wNNNNN
\__fp_trig_large_auxxi:w

```

The `auxvii` auxiliary is followed by 52 digits and a semicolon. We find the octant as the integer part of 8 times what follows, or equivalently as the integer part of $\#1\#2\#3/125$, and add it to the surrounding integer expression for the octant. We then compute 8 times the 52-digit number, with a minus sign if the octant is odd. Again, the last `middle` shift is converted to a `trailing` shift. Any integer part (including negative values which come up when the octant is odd) is discarded by `__fp_use_i_until_s:nw`. The resulting fractional part should then be converted to radians by multiplying by $2\pi/8$, but first, build an extended precision number by abusing `__fp_ep_to_ep_loop:N` with the appropriate trailing markers. Finally, `__fp_trig_small:ww` sets up the argument for the functions which compute the Taylor series.

```

14098 \cs_new:Npn \__fp_trig_large_auxvii:w #1#2#3
14099 {
14100   \exp_after:wN \__fp_trig_large_auxviii:ww
14101   \int_use:N \__int_eval:w (#1#2#3 - 62) / 125 ;
14102   #1#2#3
14103 }
14104 \cs_new:Npn \__fp_trig_large_auxviii:ww #1;
14105 {
14106   + #1
14107   \if_int_odd:w #1 \exp_stop_f:
14108     \exp_after:wN \__fp_trig_large_auxix:Nw
14109     \exp_after:wN -
14110   \else:
14111     \exp_after:wN \__fp_trig_large_auxix:Nw
14112     \exp_after:wN +
14113   \fi:
14114 }
14115 \cs_new_nopar:Npn \__fp_trig_large_auxix:Nw
14116 {
14117   \exp_after:wN \__fp_use_i_until_s:nw
14118   \exp_after:wN \__fp_trig_large_auxxi:w
14119   \int_use:N \__int_eval:w \c__fp_leading_shift_int
14120   \prg_replicate:nn { \c_thirteen }
14121   { \__fp_trig_large_auxx:wNNNNN }
14122   + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
14123   ;
14124 }

```

```

14125 \cs_new:Npn \__fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
14126 {
14127   \exp_after:wN \__fp_trig_large_pack:NNNNNw
14128   \int_use:N \__int_eval:w \c__fp_middle_shift_int
14129     #2 \c_eight * #3#4#5#6
14130     #1; #2
14131 }
14132 \cs_new:Npn \__fp_trig_large_auxxi:w #1;
14133 {
14134   \exp_after:wN \__fp_ep_mul_raw:wwwN
14135   \int_use:N \__int_eval:w \c_zero \__fp_ep_to_ep_loop:N #1 ; ; !
14136   0,{7853}{9816}{3397}{4483}{0961}{5661};
14137   \__fp_trig_small:ww
14138 }
(End definition for \__fp_trig_large_auxvii:w and \__fp_trig_large_auxviii:w.)

```

32.1.6 Computing the power series

```

\__fp_sin_series_o:NNwww
\__fp_sin_series_aux_o:NNwww

```

Here we receive a conversion function `__fp_ep_to_float:wwN` or `__fp_ep_inv_to_float:wwN`, a *sign* (0 or 2), a (non-negative) *octant* delimited by a dot, a *fixed point* number delimited by a semicolon, and an extended-precision number. The auxiliary receives:

- the conversion function `#1`;
- the final sign, which depends on the octant `#3` and the sign `#2`;
- the octant `#3`, which will control the series we use;
- the square `#4 * #4` of the argument as a fixed point number, computed with `__fp_fixed_mul:wwn`;
- the number itself as an extended-precision number.

If the octant is in $\{1, 2, 5, 6, \dots\}$, 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.

```

14139 \cs_new:Npn \__fp_sin_series_o:NNwww #1#2#3. #4;
14140 {
14141   \__fp_fixed_mul:wwn #4; #4;
14142   {
14143     \exp_after:wN \__fp_sin_series_aux_o:NNwww

```

```

14144 \exp_after:wN #1
14145 \__int_value:w
14146 \if_int_odd:w \__int_eval:w ( #3 + \c_two ) / \c_four \__int_eval_end:
14147 #2
14148 \else:
14149 \if_meaning:w #2 0 2 \else: 0 \fi:
14150 \fi:
14151 {#3}
14152 }
14153 }
14154 \cs_new:Npn \__fp_sin_series_aux_o:NNnwww #1#2#3 #4; #5,#6;
14155 {
14156 \if_int_odd:w \__int_eval:w #3 / \c_two \__int_eval_end:
14157 \exp_after:wN \use_i:nn
14158 \else:
14159 \exp_after:wN \use_ii:nn
14160 \fi:
14161 { % 1/18!
14162 \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0001}{5619}{2070};
14163 #4; {0000}{0000}{0000}{0477}{9477}{3324};
14164 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0000}{0011}{4707}{4559}{7730};
14165 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0000}{2087}{6756}{9878}{6810};
14166 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0027}{5573}{1922}{3985}{8907};
14167 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{2480}{1587}{3015}{8730}{1587};
14168 \__fp_fixed_mul_sub_back:wwwn #4; {0013}{8888}{8888}{8888}{8888}{8889};
14169 \__fp_fixed_mul_sub_back:wwwn #4; {0416}{6666}{6666}{6666}{6666}{6667};
14170 \__fp_fixed_mul_sub_back:wwwn #4; {5000}{0000}{0000}{0000}{0000}{0000};
14171 \__fp_fixed_mul_sub_back:wwwn #4; {10000}{0000}{0000}{0000}{0000}{0000};
14172 { \__fp_fixed_continue:wn 0, }
14173 }
14174 { % 1/17!
14175 \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0028}{1145}{7254};
14176 #4; {0000}{0000}{0000}{7647}{1637}{3182};
14177 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0000}{0160}{5904}{3836}{8216};
14178 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0002}{5052}{1083}{8544}{1719};
14179 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0275}{5731}{9223}{9858}{9065};
14180 \__fp_fixed_mul_sub_back:wwwn #4; {0001}{9841}{2698}{4126}{9841}{2698};
14181 \__fp_fixed_mul_sub_back:wwwn #4; {0083}{3333}{3333}{3333}{3333}{3333};
14182 \__fp_fixed_mul_sub_back:wwwn #4; {1666}{6666}{6666}{6666}{6666}{6667};
14183 \__fp_fixed_mul_sub_back:wwwn #4; {10000}{0000}{0000}{0000}{0000}{0000};
14184 { \__fp_ep_mul:wwwn 0, } #5,#6;
14185 }
14186 {
14187 \exp_after:wN \__fp_sanitize:Nw
14188 \exp_after:wN #2
14189 \int_use:N \__int_eval:w #1
14190 }
14191 #2
14192 }

```

(End definition for __fp_sin_series_o:NNwww and __fp_sin_series_aux_o:NNnwww.)

`_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.

```

14193 \cs_new:Npn \_fp_tan_series_o:NNwww #1#2#3. #4;
14194 {
14195   \_fp_fixed_mul:wwn #4; #4;
14196   {
14197     \exp_after:wN \_fp_tan_series_aux_o:Nnwww
14198     \_int_value:w
14199     \if_int_odd:w \_int_eval:w #3 / \c_two \_int_eval_end:
14200     \exp_after:wN \reverse_if:N
14201     \fi:
14202     \if_meaning:w #1#2 2 \else: 0 \fi:
14203     {#3}
14204   }
14205 }
14206 \cs_new:Npn \_fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
14207 {
14208   \_fp_fixed_mul_sub_back:wwwn {0000}{0000}{1527}{3493}{0856}{7059};
14209   #3; {0000}{0159}{6080}{0274}{5257}{6472};
14210   \_fp_fixed_mul_sub_back:wwwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
14211   \_fp_fixed_mul_sub_back:wwwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
14212   \_fp_fixed_mul_sub_back:wwwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
14213   \_fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
14214   { \_fp_ep_mul:wwwn 0, } #4,#5;
14215   {
14216     \_fp_fixed_mul_sub_back:wwwn {0000}{0007}{0258}{0681}{9408}{4706};
14217     #3; {0000}{2343}{7175}{1399}{6151}{7670};
14218     \_fp_fixed_mul_sub_back:wwwn #3; {0019}{2638}{4588}{9232}{8861}{3691};
14219     \_fp_fixed_mul_sub_back:wwwn #3; {0536}{6357}{0691}{4344}{6852}{4252};
14220     \_fp_fixed_mul_sub_back:wwwn #3; {5263}{1578}{9473}{6842}{1052}{6315};
14221     \_fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
14222     {

```

```

14223         \reverse_if:N \if_int_odd:w
14224         \__int_eval:w (#2 - \c_one) / \c_two \__int_eval_end:
14225         \exp_after:wN \__fp_reverse_args:Nww
14226         \fi:
14227         \__fp_ep_div:wwwn 0,
14228     }
14229 }
14230 {
14231     \exp_after:wN \__fp_sanitize:Nw
14232     \exp_after:wN #1
14233     \int_use:N \__int_eval:w \__fp_ep_to_float:wwN
14234 }
14235 #1
14236 }

```

(End definition for `__fp_tan_series_o:NNwww` and `__fp_tan_series_aux_o:Nnwww`.)

32.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arcsecant, and arcsecant) are based on a function often denoted `atan2`. This function is accessed directly by feeding two arguments to arctangent, and is defined by $\text{atan}(y, x) = \text{atan}(y/x)$ for generic y and x . Its advantages over the conventional arctangent is that it takes values in $[-\pi, \pi]$ rather than $[-\pi/2, \pi/2]$, and that it is better behaved in boundary cases. Other inverse trigonometric functions are expressed in terms of `atan` as

$$\arccos x = \text{atan}(\sqrt{1 - x^2}, x) \quad (4)$$

$$\arcsin x = \text{atan}(x, \sqrt{1 - x^2}) \quad (5)$$

$$\text{arcsec } x = \text{atan}(\sqrt{x^2 - 1}, 1) \quad (6)$$

$$\text{acsc } x = \text{atan}(1, \sqrt{x^2 - 1}) \quad (7)$$

$$\text{atan } x = \text{atan}(x, 1) \quad (8)$$

$$\text{acot } x = \text{atan}(1, x). \quad (9)$$

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 $\text{atan}(y, x) = \text{atan}(y/x)$ is the angular coordinate of the point (x, y) .

As for direct trigonometric functions, the first step in computing $\text{atan}(y, x)$ is argument reduction. The sign of y will give that of the result. We distinguish eight regions where the point $(x, |y|)$ can lie, of angular size roughly $\pi/8$, characterized by their “octant”, between 0 and 7 included. In each region, we compute an arctangent as a Taylor series, then shift this arctangent by the appropriate multiple of $\pi/4$ and sign to get the result. Here is a list of octants, and how we compute the arctangent (we assume $y > 0$: otherwise replace y by $-y$ below):

0 $0 < |y| < 0.41421x$, then $\text{atan } \frac{|y|}{x}$ is given by a nicely convergent Taylor series;

- 1 $0 < 0.41421x < |y| < x$, then $\operatorname{atan} \frac{|y|}{x} = \frac{\pi}{4} - \operatorname{atan} \frac{x-|y|}{x+|y|}$;
- 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.

32.2.1 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: $\operatorname{atan}(y) = \operatorname{atan}(y, 1) = \operatorname{acot}(1, y)$ and $\operatorname{acot}(x) = \operatorname{atan}(1, x) =$
 $\operatorname{acot}(x, 1)$.

```

14237 \cs_new_nopar:Npn \_fp_atan_o:Nw
14238 {
14239   \_fp_atan_dispatch_o:NNnNw
14240   \_fp_acotii_o:Nww \_fp_atanii_o:Nww { atan }
14241 }
14242 \cs_new_nopar:Npn \_fp_acot_o:Nw
14243 {
14244   \_fp_atan_dispatch_o:NNnNw
14245   \_fp_atanii_o:Nww \_fp_acotii_o:Nww { acot }
14246 }
14247 \cs_new:Npn \_fp_atan_dispatch_o:NNnNw #1#2#3#4#5@
14248 {
14249   \if_case:w
14250     \__int_eval:w \_fp_array_count:n {#5} - \c_one \__int_eval_end:
14251     \exp_after:wN #1 \exp_after:wN #4 \c_one_fp #5
14252     \tex_romannumeral:D
14253   \or: #2 #4 #5 \tex_romannumeral:D
14254   \else:
14255     \_msg_kernel_expandable_error:nnnnn
14256     { kernel } { fp-num-args } { #3() } { 1 } { 2 }
14257     \exp_after:wN \c_nan_fp \tex_romannumeral:D
14258   \fi:
14259   \exp_after:wN \c_zero
14260 }
```

(End definition for `_fp_atan_o:Nw` and `_fp_acot_o:Nw`.)

`_fp_atanii_o:Nww` If either operand is `nan`, we return it. If both are normal, we call `_fp_atan_normal_o:NNnwNnw`. If both are zero or both infinity, we call `_fp_atan_inf_o:NNNw` with argument 2, leading to a result among $\{\pm\pi/4, \pm3\pi/4\}$ (in degrees, $\{\pm45, \pm135\}$). Otherwise, one is much bigger than the other, and we call `_fp_atan_inf_o:NNNw` with either an argument of 4, leading to the values $\pm\pi/2$ (in degrees, ±90), or 0, leading to $\{\pm0, \pm\pi\}$ (in degrees, $\{\pm0, \pm180\}$). Since $\text{acot}(x, y) = \text{atan}(y, x)$, `_fp_acotii_o:ww` simply reverses its two arguments.

```

14261 \cs_new:Npn \_fp_atanii_o:Nww
14262   #1 \s_fp \_fp_chk:w #2#3#4; \s_fp \_fp_chk:w #5
14263   {
14264     \if_meaning:w 3 #2 \_fp_case_return_i_o:ww \fi:
14265     \if_meaning:w 3 #5 \_fp_case_return_ii_o:ww \fi:
14266     \if_case:w
14267       \if_meaning:w #2 #5
14268       \if_meaning:w 1 #2 \c_ten \else: \c_zero \fi:
14269       \else:
14270       \if_int_compare:w #2 > #5 \c_one \else: \c_two \fi:
14271       \fi:
14272       \_fp_case_return:nw { \_fp_atan_inf_o:NNNw #1 #3 \c_two }
14273       \or: \_fp_case_return:nw { \_fp_atan_inf_o:NNNw #1 #3 \c_four }
14274       \or: \_fp_case_return:nw { \_fp_atan_inf_o:NNNw #1 #3 \c_zero }
14275       \fi:
14276       \_fp_atan_normal_o:NNnwNnw #1
14277       \s_fp \_fp_chk:w #2#3#4;
14278       \s_fp \_fp_chk:w #5
14279     }
14280 \cs_new:Npn \_fp_acotii_o:Nww #1#2; #3;
14281   { \_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.

```

14282 \cs_new:Npn \_fp_atan_inf_o:NNNw #1#2#3 \s_fp \_fp_chk:w #4#5#6;
14283   {
14284     \exp_after:wN \_fp_atan_combine_o:NwwwwwN
14285     \exp_after:wN #2
14286     \int_use:N \_int_eval:w
14287     \if_meaning:w 2 #5 \c_seven - \fi: #3 \exp_after:wN ;
14288     \c__fp_one_fixed_tl ;
14289     {0000}{0000}{0000}{0000}{0000}{0000};
14290     0,{0000}{0000}{0000}{0000}{0000}{0000}; #1
14291   }

```

(End definition for `_fp_atan_inf_o:NNNw`.)

`_fp_atan_normal_o:NNwNnw`

Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a semi-colon. This extended precision is required by other inverse trigonometric functions, to compute things like $\text{atan}(x, \sqrt{1-x^2})$ without intermediate rounding errors.

```

14292 \cs_new_protected:Npn \_fp_atan_normal_o:NNwNnw
14293   #1 \s\_fp \_fp_chk:w 1#2#3#4; \s\_fp \_fp_chk:w 1#5#6#7;
14294   {
14295     \_fp_atan_test_o:NwwNwN
14296     #2 #3, #4{0000}{0000};
14297     #5 #6, #7{0000}{0000}; #1
14298   }

```

(End definition for `_fp_atan_normal_o:NNwNnw`.)

`_fp_atan_test_o:NwwNwN`

This receives: the sign `#1` of y , its exponent `#2`, its 24 digits `#3` in groups of 4, and similarly for x . We prepare to call `_fp_atan_combine_o:NwwwwwN` which expects the sign `#1`, the octant, the ratio $(\text{atan } z)/z = 1 - \dots$, and the value of z , both as a fixed point number and as an extended-precision floating point number with a mantissa in $[0.01, 1)$. For now, we place `#1` as a first argument, and start an integer expression for the octant. The sign of x does not affect what z will be, so we simply leave a contribution to the octant: $\langle \text{octant} \rangle \rightarrow 7 - \langle \text{octant} \rangle$ for negative x . Then we order $|y|$ and $|x|$ in a non-decreasing order: if $|y| > |x|$, insert 3– in the expression for the octant, and swap the two numbers. The finer test with 0.41421 is done by `_fp_atan_div:wnwnw` after the operands have been ordered.

```

14299 \cs_new:Npn \_fp_atan_test_o:NwwNwN #1#2,#3; #4#5,#6;
14300   {
14301     \exp_after:wN \_fp_atan_combine_o:NwwwwwN
14302     \exp_after:wN #1
14303     \int_use:N \_int_eval:w
14304     \if_meaning:w 2 #4
14305       \c_seven - \_int_eval:w
14306     \fi:
14307     \if_int_compare:w
14308       \_fp_ep_compare:www #2,#3; #5,#6; > \c_zero
14309       \c_three -
14310       \exp_after:wN \_fp_reverse_args:Nw
14311     \fi:
14312     \_fp_atan_div:wnwnw #2,#3; #5,#6;
14313   }

```

(End definition for `_fp_atan_test_o:NwwNwN`.)

`_fp_atan_div:wnwnw`
`_fp_atan_near:wwn`
`_fp_atan_near_aux:wnw`

This receives two positive numbers a and b (equal to $|x|$ and $|y|$ in some order), each as an exponent and 6 blocks of 4 digits, such that $0 < a < b$. If $0.41421b < a$, the two numbers are “near”, hence the point (y, x) that we started with is closer to the diagonals $\{|y| = |x|\}$ than to the axes $\{xy = 0\}$. In that case, the octant is 1 (possibly combined with the 7– and 3– inserted earlier) and we wish to compute $\text{atan } \frac{b-a}{a+b}$. Otherwise, the

octant is 0 (again, combined with earlier terms) and we wish to compute $\text{atan } \frac{a}{b}$. In any case, call `__fp_atan_auxi:ww` followed by z , as a comma-delimited exponent and a fixed point number.

```

14314 \cs_new:Npn \__fp_atan_div:wnwnw #1,#2#3; #4,#5#6;
14315 {
14316   \if_int_compare:w 41421 * #5 < #2 000
14317     \__int_eval:w \__int_eval:w #4 - #1 \__int_eval_end: 00 \or: 0 \fi:
14318     \if_case:w \__int_eval:w #4 - #1 \__int_eval_end: 00 \or: 0 \fi:
14319     \exp_stop_f:
14320     \exp_after:wN \__fp_atan_near:wwwn
14321     \fi:
14322     \c_zero
14323     \__fp_ep_div:wwwn #1,{#2}#3; #4,{#5}#6;
14324     \__fp_atan_auxi:ww
14325   }
14326 \cs_new:Npn \__fp_atan_near:wwwn
14327   \c_zero \__fp_ep_div:wwwn #1,#2; #3,
14328   {
14329     \c_one
14330     \__fp_ep_to_fixed:wn #1 - #3, #2;
14331     \__fp_atan_near_aux:wn
14332   }
14333 \cs_new:Npn \__fp_atan_near_aux:wn #1; #2;
14334 {
14335   \__fp_fixed_add:wn #1; #2;
14336   { \__fp_fixed_sub:wn #2; #1; { \__fp_ep_div:wwwn 0, } 0, }
14337 }
(End definition for \__fp_atan_div:wnwnw and \__fp_atan_near:wwwn.)

```

`__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`

```

14338 \cs_new:Npn \__fp_atan_auxi:ww #1,#2;
14339 { \__fp_ep_to_fixed:wn #1,#2; \__fp_atan_auxii:w #1,#2; }
14340 \cs_new:Npn \__fp_atan_auxii:w #1;
14341 {
14342   \__fp_fixed_mul:wn #1; #1;
14343   {
14344     \__fp_atan_Taylor_loop:www 39 ;
14345     {0000}{0000}{0000}{0000}{0000}{0000} ;
14346   }
14347   ! #1;
14348 }
(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 $(\text{atan } z)/z$. 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.

```

14349 \cs_new:Npn \__fp_atan_Taylor_loop:www #1; #2; #3;
14350 {
14351   \if_int_compare:w #1 = \c_minus_one
14352     \__fp_atan_Taylor_break:w
14353   \fi:
14354   \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl ; #1;
14355   \__fp_rrot:www \__fp_fixed_mul_sub_back:wwwn #2; #3;
14356   {
14357     \exp_after:wN \__fp_atan_Taylor_loop:www
14358     \int_use:N \__int_eval:w #1 - \c_two ;
14359   }
14360   #3;
14361 }
14362 \cs_new:Npn \__fp_atan_Taylor_break:w
14363   \fi: #1 \__fp_fixed_mul_sub_back:wwwn #2; #3 !
14364   { \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 $(\operatorname{atan} z)/z$, 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

$$\langle sign \rangle \left(\left\lceil \frac{\langle octant \rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle octant \rangle} \frac{\operatorname{atan} z}{z} \cdot z \right), \quad (10)$$

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 = $\frac{\operatorname{atan} z}{z}$ with #6, the adjusted z . Otherwise, multiply #3 = $\frac{\operatorname{atan} z}{z}$ with #4 = z , then compute the appropriate multiple of $\frac{\pi}{4}$ and add or subtract the product #3 · #4. In both cases, convert to a floating point with `__fp_fixed_to_float:wN`.

```

14365 \cs_new:Npn \__fp_atan_combine_o:NwwwwN #1 #2; #3; #4; #5,#6; #7
14366 {
14367   \exp_after:wN \__fp_sanitize:Nw
14368   \exp_after:wN #1
14369   \int_use:N \__int_eval:w
14370   \if_meaning:w 0 #2
14371     \exp_after:wN \use_i:nn
14372   \else:
14373     \exp_after:wN \use_ii:nn
14374   \fi:
14375   { #5 \__fp_fixed_mul:wwn #3; #6; }
14376   {
14377     \__fp_fixed_mul:wwn #3; #4;

```

```

14378     {
14379         \exp_after:wN \__fp_atan_combine_aux:ww
14380         \int_use:N \__int_eval:w #2 / \c_two ; #2;
14381     }
14382 }
14383 { #7 \__fp_fixed_to_float:wN \__fp_fixed_to_float_rad:wN }
14384 #1
14385 }
14386 \cs_new:Npn \__fp_atan_combine_aux:ww #1; #2;
14387 {
14388     \__fp_fixed_mul_short:wwn
14389     {7853}{9816}{3397}{4483}{0961}{5661};
14390     {#1}{0000}{0000};
14391     {
14392         \if_int_odd:w #2 \exp_stop_f:
14393         \exp_after:wN \__fp_fixed_sub:wwn
14394         \else:
14395         \exp_after:wN \__fp_fixed_add:wwn
14396         \fi:
14397     }
14398 }
(End definition for \__fp_atan_combine_o:NwwwN and \__fp_atan_combine_aux:ww.)

```

32.2.2 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).

```

14399 \cs_new:Npn \__fp_asin_o:w #1 \s__fp \__fp_chk:w #2#3; @
14400 {
14401     \if_case:w #2 \exp_stop_f:
14402     \__fp_case_return_same_o:w
14403     \or:
14404     \__fp_case_use:nw
14405     { \__fp_asin_normal_o:NfwNnnnw #1 { #1 { asin } { asind } } }
14406     \or:
14407     \__fp_case_use:nw
14408     { \__fp_invalid_operation_o:fw { #1 { asin } { asind } } }
14409     \else:
14410     \__fp_case_return_same_o:w
14411     \fi:
14412     \s__fp \__fp_chk:w #2 #3;
14413 }
(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.

```

14414 \cs_new:Npn \__fp_acos_o:w #1 \s__fp \__fp_chk:w #2#3; @
14415 {
14416   \if_case:w #2 \exp_stop_f:
14417     \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 \c_four }
14418   \or:
14419     \__fp_case_use:nw
14420     {
14421       \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
14422       \__fp_reverse_args:Nww
14423     }
14424   \or:
14425     \__fp_case_use:nw
14426     { \__fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
14427   \else:
14428     \__fp_case_return_same_o:w
14429   \fi:
14430   \s__fp \__fp_chk:w #2 #3;
14431 }

```

(End definition for `__fp_acos_o:w`.)

`__fp_asin_normal_o:NfwNnnnnw` If the exponent #5 is strictly less than 1, the operand lies within $(-1, 1)$ and the operation is permitted: call `__fp_asin_auxi_o:nNww` with the appropriate arguments. If the number is exactly ± 1 (the test works because we know that $\#5 \geq 1$, $\#6\#7 \geq 10000000$, $\#8\#9 \geq 0$, with equality only for ± 1), we also call `__fp_asin_auxi_o:nNww`. Otherwise, `__fp_use_i:ww` gets rid of the `asin` auxiliary, and raises instead an invalid operation, because the operand is outside the domain of arcsine or arccosine.

```

14432 \cs_new:Npn \__fp_asin_normal_o:NfwNnnnnw
14433   #1#2#3 \s__fp \__fp_chk:w 1#4#5#6#7#8#9;
14434 {
14435   \if_int_compare:w #5 < \c_one
14436     \exp_after:wN \__fp_use_none_until_s:w
14437   \fi:
14438   \if_int_compare:w \__int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
14439     \exp_after:wN \__fp_use_none_until_s:w
14440   \fi:
14441   \__fp_use_i:ww
14442   \__fp_invalid_operation_o:fw {#2}
14443   \s__fp \__fp_chk:w 1#4{#5}{#6}{#7}{#8}{#9};
14444   \__fp_asin_auxi_o:NnNww
14445   #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
14446 }

```

(End definition for `__fp_asin_normal_o:NfwNnnnnw`.)

`__fp_asin_auxi_o:NnNww` We compute $x/\sqrt{1-x^2}$. This function is used by `asin` and `acos`, but also by `acsc` and `asec` after inverting the operand, thus it must manipulate extended-precision numbers. `__fp_asin_isqrt:wn` 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`.

```

14447 \cs_new:Npn \__fp_asin_auxi_o:NnNww #1#2#3#4,#5;
14448 {
14449   \__fp_ep_to_fixed:wn #4,#5;
14450   \__fp_asin_isqrt:wn
14451   \__fp_ep_mul:wwwwn #4,#5;
14452   \__fp_ep_to_ep:wnN
14453   \__fp_fixed_continue:wn
14454   { #2 \__fp_atan_test_o:NwwNwwN #3 }
14455   0 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1
14456 }
14457 \cs_new:Npn \__fp_asin_isqrt:wn #1;
14458 {
14459   \exp_after:wN \__fp_fixed_sub:wnn \c__fp_one_fixed_tl ; #1;
14460   {
14461     \__fp_fixed_add_one:wn #1;
14462     \__fp_fixed_continue:wn { \__fp_ep_mul:wwwwn 0, } 0,
14463   }
14464   \__fp_ep_isqrt:wnn
14465 }

```

(End definition for `__fp_asin_auxi_o:NnNww` and `__fp_asin_isqrt:wn`.)

32.2.3 Arccosecant and arcsecant

`__fp_acsc_o:w` Cases are mostly labelled by #2, except when #2 is 2: then we use #3#2, which is 02 = 2 when the number is $+\infty$ and 22 when the number is $-\infty$. The arccosecant of ± 0 raises an invalid operation exception. The arccosecant of $\pm\infty$ is ± 0 with the same sign. The arcosecant of NaN is itself. Otherwise, `__fp_acsc_normal_o:NfwNnw` does some more tests, keeping the function name (acsc or acscd) as an argument for invalid operation exceptions.

```

14466 \cs_new:Npn \__fp_acsc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14467 {
14468   \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
14469   \__fp_case_use:nw
14470   { \__fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
14471   \or: \__fp_case_use:nw
14472   { \__fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
14473   \or: \__fp_case_return_o:Nw \c_zero_fp
14474   \or: \__fp_case_return_same_o:w
14475   \else: \__fp_case_return_o:Nw \c_minus_zero_fp
14476   \fi:

```

```

14477 \s__fp \__fp_chk:w #2 #3 #4;
14478 }

```

(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.

```

14479 \cs_new:Npn \__fp_asec_o:w #1 \s__fp \__fp_chk:w #2#3; @
14480 {
14481   \if_case:w #2 \exp_stop_f:
14482     \__fp_case_use:nw
14483     { \__fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
14484   \or:
14485     \__fp_case_use:nw
14486     {
14487       \__fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
14488       \__fp_reverse_args:Nww
14489     }
14490   \or: \__fp_case_use:nw { \__fp_atan_inf_o:NNnw #1 0 \c_four }
14491   \else: \__fp_case_return_same_o:w
14492   \fi:
14493   \s__fp \__fp_chk:w #2 #3;
14494 }

```

(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 $\text{asec}(x) = \text{acos}(1/x)$.

```

14495 \cs_new:Npn \__fp_acsc_normal_o:NfwNnw #1#2#3 \s__fp \__fp_chk:w 1#4#5#6;
14496 {
14497   \int_compare:nNnTF {#5} < \c_one
14498   {
14499     \__fp_invalid_operation_o:fw {#2}
14500     \s__fp \__fp_chk:w 1#4{#5}#6;
14501   }
14502   {
14503     \__fp_ep_div:wwwn
14504     1,{1000}{0000}{0000}{0000}{0000}{0000};
14505     #5,#6{0000}{0000};
14506     { \__fp_asin_auxi_o:nNnw #1 {#3} #4 }
14507   }
14508 }

```

(End definition for __fp_acsc_normal_o:NfwNnw.)

```

14509 </initex | package)

```

33 13fp-convert implementation

```
14510 <*initex | package>
14511 <@@=fp>
```

33.1 Trimming trailing zeros

```

    \__fp_trim_zeros:w
\__fp_trim_zeros_loop:w
\__fp_trim_zeros_dot:w
\__fp_trim_zeros_end:w

```

If #1 ends with a 0, the loop auxiliary takes that zero as an end-delimiter for its first argument, and the second argument is the same loop auxiliary. Once the last trailing zero is reached, the second argument will be the dot auxiliary, which removes a trailing dot if any. We then clean-up with the end auxiliary, keeping only the number.

```

14512 \cs_new:Npn \__fp_trim_zeros:w #1 ;
14513 {
14514     \__fp_trim_zeros_loop:w #1
14515     ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s__stop
14516 }
14517 \cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
14518 \cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
14519 \cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__stop { #1 }

```

(End definition for __fp_trim_zeros:w.)

33.2 Scientific notation

```

\fp_to_scientific:N
\fp_to_scientific:c
\fp_to_scientific:n

```

The three public functions evaluate their argument, then pass it to __fp_to_scientific_dispatch:w.

```

14520 \cs_new:Npn \fp_to_scientific:N #1
14521 { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
14522 \cs_generate_variant:Nn \fp_to_scientific:N { c }
14523 \cs_new_nopar:Npn \fp_to_scientific:n
14524 {
14525     \exp_after:wN \__fp_to_scientific_dispatch:w
14526     \tex_romannumeral:D -'0 \__fp_parse:n
14527 }

```

(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”.

```

14528 \group_begin:
14529 \char_set_catcode_other:N E

```

```

14530 \tl_to_lowercase:n
14531 {
14532   \group_end:
14533   \cs_new:Npn \__fp_to_scientific_dispatch:w \s__fp \__fp_chk:w #1#2
14534   {
14535     \if_meaning:w 2 #2 \exp_after:wN - \tex_romannumeral:D -'0 \fi:
14536     \if_case:w #1 \exp_stop_f:
14537       \__fp_case_return:nw { 0 }
14538     \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
14539     \or:
14540       \__fp_case_use:nw
14541       {
14542         \__fp_invalid_operation:nnw
14543         {
14544           \exp_after:wN 1
14545           \exp_after:wN E
14546           \int_use:N \c__fp_max_exponent_int
14547         }
14548         { fp_to_scientific }
14549       }
14550     \or:
14551       \__fp_case_use:nw
14552       {
14553         \__fp_invalid_operation:nnw
14554         { 0 }
14555         { fp_to_scientific }
14556       }
14557     \fi:
14558     \s__fp \__fp_chk:w #1 #2
14559   }
14560   \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
14561   \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
14562   {
14563     \if_int_compare:w #2 = \c_one
14564       \exp_after:wN \__fp_to_scientific_normal:wNw
14565     \else:
14566       \exp_after:wN \__fp_to_scientific_normal:wNw
14567       \exp_after:wN E
14568       \int_use:N \__int_eval:w #2 - \c_one
14569     \fi:
14570     ; #3 #4 #5 #6 ;
14571   }
14572 }
14573 \cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
14574 { \__fp_trim_zeros:w #2.#3 ; #1 }
(End definition for \__fp_to_scientific_dispatch:w, \__fp_to_scientific_normal:wnnnnn, and \__fp_to_scientific_nor

```

33.3 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`

```

14575 \cs_new:Npn \fp_to_decimal:N #1
14576 { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
14577 \cs_generate_variant:Nn \fp_to_decimal:N { c }
14578 \cs_new_nopar:Npn \fp_to_decimal:n
14579 {
14580   \exp_after:wN \__fp_to_decimal_dispatch:w
14581   \tex_romannumeral:D -'0 \__fp_parse:n
14582 }

```

(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.

```

14583 \cs_new:Npn \__fp_to_decimal_dispatch:w \s__fp \__fp_chk:w #1#2
14584 {
14585   \if_meaning:w 2 #2 \exp_after:wN - \tex_romannumeral:D -'0 \fi:
14586   \if_case:w #1 \exp_stop_f:
14587     \__fp_case_return:nw { 0 }
14588   \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
14589   \or:
14590     \__fp_case_use:nw
14591     {
14592       \__fp_invalid_operation:nnw
14593       {
14594         \exp_after:wN \exp_after:wN \exp_after:wN 1
14595         \prg_replicate:nn \c__fp_max_exponent_int 0
14596       }
14597       { fp_to_decimal }
14598     }
14599   \or:
14600     \__fp_case_use:nw
14601     {
14602       \__fp_invalid_operation:nnw
14603       { 0 }
14604       { fp_to_decimal }
14605     }
14606   \fi:
14607   \s__fp \__fp_chk:w #1 #2
14608 }

```

```

14609 \cs_new:Npn \__fp_to_decimal_normal:wnnnnn
14610 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
14611 {
14612   \int_compare:nNnTF {#2} > \c_zero
14613   {
14614     \int_compare:nNnTF {#2} < \c_sixteen
14615     {
14616       \__fp_decimate:nNnnnn { \c_sixteen - #2 }
14617       \__fp_to_decimal_large:Nnnw
14618     }
14619     {
14620       \exp_after:wN \exp_after:wN
14621       \exp_after:wN \__fp_to_decimal_huge:wnnnn
14622       \prg_replicate:nn { #2 - \c_sixteen } { 0 } ;
14623     }
14624     {#3} {#4} {#5} {#6}
14625   }
14626   {
14627     \exp_after:wN \__fp_trim_zeros:w
14628     \exp_after:wN 0
14629     \exp_after:wN .
14630     \tex_romannumeral:D -'0 \prg_replicate:nn { - #2 } { 0 }
14631     #3#4#5#6 ;
14632   }
14633 }
14634 \cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
14635 {
14636   \exp_after:wN \__fp_trim_zeros:w \__int_value:w
14637   \if_int_compare:w #2 > \c_zero
14638   #2
14639   \fi:
14640   \exp_stop_f:
14641   #3.#4 ;
14642 }
14643 \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.)

```

33.4 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

```

14644 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
14645 \cs_generate_variant:Nn \fp_to_tl:N { c }
14646 \cs_new_nopar:Npn \fp_to_tl:n
14647 {
14648   \exp_after:wN \__fp_to_tl_dispatch:w
14649   \tex_romannumeral:D -'0 \__fp_parse:n
14650 }

```

(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.

```

14651 \cs_new:Npn \__fp_to_tl_dispatch:w \s__fp \__fp_chk:w #1#2
14652 {
14653   \if_meaning:w 2 #2 \exp_after:wN - \tex_romannumeral:D -'0 \fi:
14654   \if_case:w #1 \exp_stop_f:
14655     \__fp_case_return:nw { 0 }
14656   \or: \exp_after:wN \__fp_to_tl_normal:nnnnn
14657   \or: \__fp_case_return:nw { \tl_to_str:n {inf} }
14658   \else: \__fp_case_return:nw { \tl_to_str:n {nan} }
14659   \fi:
14660 }
14661 \cs_new:Npn \__fp_to_tl_normal:nnnnn #1
14662 {
14663   \if_int_compare:w #1 > \c_sixteen
14664     \exp_after:wN \__fp_to_scientific_normal:wnnnnn
14665   \else:
14666     \if_int_compare:w #1 < - \c_two
14667       \exp_after:wN \exp_after:wN
14668       \exp_after:wN \__fp_to_scientific_normal:wnnnnn
14669     \else:
14670       \exp_after:wN \exp_after:wN
14671       \exp_after:wN \__fp_to_decimal_normal:wnnnnn
14672     \fi:
14673   \fi:
14674   \s__fp \__fp_chk:w 1 0 {#1}
14675 }

```

(End definition for `__fp_to_tl_dispatch:w` and `__fp_to_tl_normal:nnnnn`.)

33.5 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.

33.6 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
14676 \cs_new:Npx \fp_to_dim:N #1
14677 { \exp_not:N \fp_to_decimal:N #1 \tl_to_str:n {pt} }
14678 \cs_generate_variant:Nn \fp_to_dim:N { c }
14679 \cs_new:Npx \fp_to_dim:n #1
14680 { \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`

```
14681 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
14682 \cs_generate_variant:Nn \fp_to_int:N { c }
14683 \cs_new_nopar:Npn \fp_to_int:n
14684 {
14685   \exp_after:wN \__fp_to_int_dispatch:w
14686   \tex_romannumeral:D -'0 \__fp_parse:n
14687 }
```

(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.

```
14688 \cs_new:Npn \__fp_to_int_dispatch:w #1;
14689 {
14690   \exp_after:wN \__fp_to_decimal_dispatch:w \tex_romannumeral:D -'0
14691   \__fp_round:Nwn \__fp_round_to_nearest:NNN #1; { 0 }
14692 }
```

(End definition for `__fp_to_int_dispatch:w`.)

33.7 Convert from a dimension

`\dim_to_fp:n` The dimension expression (which can in fact be a glue expression) is evaluated, converted to a number (*i.e.*, expressed in scaled points), then multiplied by $2^{-16} = 0.0000152587890625$ to give a value expressed in points. The auxiliary `__fp_mul_npos_o:Nww` expects the desired *final sign* and two floating point operands (of the form `\s__fp ...`;) as arguments. This set of functions is also used to convert dimension registers to floating points while parsing expressions: in this context there is an additional exponent, which is the first argument of `__fp_from_dim_test:ww`, and is combined with the exponent -4 of 2^{-16} . There is also a need to expand afterwards: this is performed by `__fp_mul_npos_o:Nww`, and cancelled by `\prg_do_nothing:` in `\dim_to_fp:n`.

```
14693 \cs_new:Npn \dim_to_fp:n #1
14694 {
14695   \exp_after:wN \__fp_from_dim_test:ww
14696   \exp_after:wN 0
14697   \exp_after:wN ,
14698   \__int_value:w \etex_glueexpr:D #1 ;
14699 }
14700 \cs_new:Npn \__fp_from_dim_test:ww #1, #2
14701 {
14702   \if_meaning:w 0 #2
14703     \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
14704   \else:
```

```

14705     \exp_after:wN \__fp_from_dim:wNw
14706     \int_use:N \__int_eval:w #1 - \c_four
14707         \if_meaning:w - #2
14708             \exp_after:wN , \exp_after:wN 2 \__int_value:w
14709         \else:
14710             \exp_after:wN , \exp_after:wN 0 \__int_value:w #2
14711         \fi:
14712     \fi:
14713 }
14714 \cs_new:Npn \__fp_from_dim:wNw #1,#2#3;
14715 {
14716     \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
14717     #3 000 0000 00 {10}987654321; #2 {#1}
14718 }
14719 \cs_new:Npn \__fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
14720 { \__fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
14721 \cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
14722 {
14723     \__fp_mul_npos_o:Nww #7
14724     \s__fp \__fp_chk:w 1 #7 {#5} #1 ;
14725     \s__fp \__fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
14726     \prg_do_nothing:
14727 }

```

(End definition for `\dim_to_fp:n`. This function is documented on page 82.)

33.8 Use and eval

`\fp_use:N` Those public functions are simple copies of the decimal conversions.

`\fp_use:c` 14728 `\cs_new_eq:NN \fp_use:N \fp_to_decimal:N`
`\fp_eval:n` 14729 `\cs_generate_variant:Nn \fp_use:N { c }`
14730 `\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 178.)

`\fp_abs:n` Trivial but useful. See the implementation of `\fp_add:Nn` for an explanation of why to use `__fp_parse:n`, namely, for better error reporting.

```

14731 \cs_new:Npn \fp_abs:n #1
14732 { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }

```

(End definition for `\fp_abs:n`. This function is documented on page 191.)

`\fp_max:nn` Similar to `\fp_abs:n`, for consistency with `\int_max:nn`, etc.

`\fp_min:nn` 14733 `\cs_new:Npn \fp_max:nn #1#2`
14734 `{ \fp_to_decimal:n { max (__fp_parse:n {#1} , __fp_parse:n {#2}) } }`
14735 `\cs_new:Npn \fp_min:nn #1#2`
14736 `{ \fp_to_decimal:n { min (__fp_parse:n {#1} , __fp_parse:n {#2}) } }`

(End definition for `\fp_max:nn` and `\fp_min:nn`. These functions are documented on page 191.)

33.9 Convert an array of floating points to a comma list

`__fp_array_to_clist:n` Converts an array of floating point numbers to a comma-list. If speed here ends up irrelevant, we can simplify the code for the auxiliary to become

```
\cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
{
  \use_none:n #1
  { , ~ } \fp_to_tl:n { #1 #2 ; }
  \__fp_array_to_clist_loop:Nw
}
```

The `\use_ii:nn` function is expanded after `__fp_expand:n` is done, and it removes `,~` from the start of the representation.

```
14737 \cs_new:Npn \__fp_array_to_clist:n #1
14738 {
14739   \tl_if_empty:nF {#1}
14740   {
14741     \__fp_expand:n
14742     {
14743       { \use_ii:nn }
14744       \__fp_array_to_clist_loop:Nw #1 { ? \__prg_break: } ;
14745       \__prg_break_point:
14746     }
14747   }
14748 }
14749 \cs_new:Npx \__fp_array_to_clist_loop:Nw #1#2;
14750 {
14751   \exp_not:N \use_none:n #1
14752   \exp_not:N \exp_after:wN
14753   {
14754     \exp_not:N \exp_after:wN ,
14755     \exp_not:N \exp_after:wN \c_space_tl
14756     \exp_not:N \tex_romannumeral:D -'0
14757     \exp_not:N \__fp_to_tl_dispatch:w #1 #2 ;
14758   }
14759   \exp_not:N \__fp_array_to_clist_loop:Nw
14760 }
(End definition for \__fp_array_to_clist:n.)
14761 </initex | package>
```

34 l3fp-assign implementation

```
14762 <*initex | package>
14763 <@@=fp>
```

34.1 Assigning values

`\fp_new:N` Floating point variables are initialized to be +0.

```

14764 \cs_new_protected:Npn \fp_new:N #1
14765 { \cs_new_eq:NN #1 \c_zero_fp }
14766 \cs_generate_variant:Nn \fp_new:N {c}

```

(End definition for `\fp_new:N`. This function is documented on page 177.)

```

\fp_set:Nn Simply use \__fp_parse:n within various f-expanding assignments.
\fp_set:cn 14767 \cs_new_protected:Npn \fp_set:Nn #1#2
\fp_gset:Nn 14768 { \tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_gset:cn 14769 \cs_new_protected:Npn \fp_gset:Nn #1#2
\fp_const:Nn 14770 { \tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_const:cn 14771 \cs_new_protected:Npn \fp_const:Nn #1#2
14772 { \tl_const:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
14773 \cs_generate_variant:Nn \fp_set:Nn {c}
14774 \cs_generate_variant:Nn \fp_gset:Nn {c}
14775 \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 14776 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
\fp_set_eq:Nc 14777 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\fp_set_eq:cc 14778 \cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }
\fp_gset_eq:NN 14779 \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_gzero:N
\fp_zero:c 14780 \cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\fp_gzero:N 14781 \cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\fp_gzero:c 14782 \cs_generate_variant:Nn \fp_zero:N { c }
14783 \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 14784 \cs_new_protected:Npn \fp_zero_new:N #1
\fp_gzero_new:N 14785 { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\fp_gzero_new:c 14786 \cs_new_protected:Npn \fp_gzero_new:N #1
14787 { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
14788 \cs_generate_variant:Nn \fp_zero_new:N { c }
14789 \cs_generate_variant:Nn \fp_gzero_new:N { c }

```

(End definition for `\fp_zero_new:N` and others. These functions are documented on page ??.)

34.2 Updating values

These match the equivalent functions in `l3int` and `l3skip`.

`\fp_add:Nn` For the sake of error recovery we should not simply set #1 to #1 ± (#2): for instance, if #2
`\fp_add:cn` is 0)+2, the parsing error would be raised at the last closing parenthesis rather than at
`\fp_gadd:Nn` the closing parenthesis in the user argument. Thus we evaluate #2 instead of just putting
`\fp_gadd:cn` parentheses. As an optimization we use `__fp_parse:n` rather than `\fp_eval:n`, which
`\fp_sub:Nn` would convert the result away from the internal representation and back.
`\fp_sub:cn` 14790 `\cs_new_protected_nopar:Npn \fp_add:Nn { __fp_add:NNNn \fp_set:Nn + }`
`\fp_gsub:Nn` 14791 `\cs_new_protected_nopar:Npn \fp_gadd:Nn { __fp_add:NNNn \fp_gset:Nn + }`
`\fp_gsub:cn` 14792 `\cs_new_protected_nopar:Npn \fp_sub:Nn { __fp_add:NNNn \fp_set:Nn - }`
`__fp_add:NNNn` 14793 `\cs_new_protected_nopar:Npn \fp_gsub:Nn { __fp_add:NNNn \fp_gset:Nn - }`
14794 `\cs_new_protected:Npn __fp_add:NNNn #1#2#3#4`
14795 `{ #1 #3 { #3 #2 __fp_parse:n {#4} } }`
14796 `\cs_generate_variant:Nn \fp_add:Nn { c }`
14797 `\cs_generate_variant:Nn \fp_gadd:Nn { c }`
14798 `\cs_generate_variant:Nn \fp_sub:Nn { c }`
14799 `\cs_generate_variant:Nn \fp_gsub:Nn { c }`
(End definition for `\fp_add:Nn` and others. These functions are documented on page ??.)

34.3 Showing values

`\fp_show:N` This shows the result of computing its argument. The `__msg_show_variable:n` auxil-
`\fp_show:c` iary expects its input in a slightly odd form, starting with >~, and displays the rest.
`\fp_show:n` 14800 `\cs_new_protected:Npn \fp_show:N #1`
14801 `{`
14802 `\fp_if_exist:NTF #1`
14803 `{ __msg_show_variable:n { > ~ \fp_to_tl:N #1 } }`
14804 `{`
14805 `__msg_kernel_error:nx { kernel } { variable-not-defined }`
14806 `{ \token_to_str:N #1 }`
14807 `}`
14808 `}`
14809 `\cs_new_protected:Npn \fp_show:n #1`
14810 `{ __msg_show_variable:n { > ~ \fp_to_tl:n {#1} } }`
14811 `\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 ??.)

34.4 Some useful constants and scratch variables

`\c_one_fp` Some constants.
`\c_e_fp` 14812 `\fp_const:Nn \c_e_fp { 2.718 2818 2845 9045 }`
14813 `\fp_const:Nn \c_one_fp { 1 }`
(End definition for `\c_one_fp` and `\c_e_fp`. These variables are documented on page 182.)

`\c_pi_fp` We simply round π to the closest multiple of 10^{-15} .
`\c_one_degree_fp` 14814 `\fp_const:Nn \c_pi_fp { 3.141 5926 5358 9793 }`
14815 `\fp_const:Nn \c_one_degree_fp { 0.0 1745 3292 5199 4330 }`
(End definition for `\c_pi_fp` and `\c_one_degree_fp`. These variables are documented on page 182.)

`\l_tmpa_fp` Scratch variables are simply initialized there.

`\l_tmpb_fp` 14816 `\fp_new:N \l_tmpa_fp`

`\g_tmpa_fp` 14817 `\fp_new:N \l_tmpb_fp`

`\g_tmpb_fp` 14818 `\fp_new:N \g_tmpa_fp`

14819 `\fp_new:N \g_tmpb_fp`

(End definition for `\l_tmpa_fp` and others. These variables are documented on page 182.)

14820 `\</initex | package>`

35 l3candidates Implementation

14821 `*initex | package>`

35.1 Additions to l3box

14822 `\@@=box>`

35.2 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.

14823 `\fp_new:N \l__box_angle_fp`

(End definition for `\l__box_angle_fp`. This variable is documented on page 197.)

`\l__box_cos_fp` These are used to hold the calculated sine and cosine values while carrying out a rotation.

`\l__box_sin_fp` 14824 `\fp_new:N \l__box_cos_fp`

14825 `\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 197.)

`\l__box_top_dim` These are the positions of the four edges of a box before manipulation.

`\l__box_bottom_dim` 14826 `\dim_new:N \l__box_top_dim`

`\l__box_left_dim` 14827 `\dim_new:N \l__box_bottom_dim`

`\l__box_right_dim` 14828 `\dim_new:N \l__box_left_dim`

14829 `\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` 14830 `\dim_new:N \l__box_top_new_dim`

`\l__box_left_new_dim` 14831 `\dim_new:N \l__box_bottom_new_dim`

`\l__box_right_new_dim` 14832 `\dim_new:N \l__box_left_new_dim`

14833 `\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.

14834 `\box_new:N \l__box_internal_box`

(End definition for `\l__box_internal_box`. This variable is documented on page 197.)

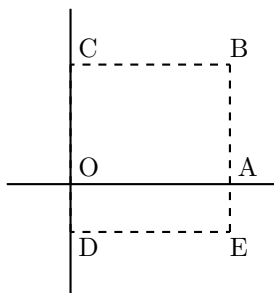


Figure 1: Co-ordinates of a box prior to rotation.

\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:

```

14835 \cs_new_protected:Npn \box_rotate:Nn #1#2
14836 {
14837   \hbox_set:Nn #1
14838   {
14839     \group_begin:
14840       \fp_set:Nn \l__box_angle_fp {#2}
14841       \fp_set:Nn \l__box_sin_fp { sind ( \l__box_angle_fp ) }
14842       \fp_set:Nn \l__box_cos_fp { cosd ( \l__box_angle_fp ) }
14843       \__box_rotate:N #1
14844     \group_end:
14845   }
14846 }

```

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.

```

14847 \cs_new_protected:Npn \__box_rotate:N #1
14848 {
14849   \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
14850   \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
14851   \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
14852   \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.)

```

14853 \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
14854 {
14855   \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
14856   { \__box_rotate_quadrant_one: }
14857   { \__box_rotate_quadrant_two: }
14858 }
14859 {
14860   \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
14861   { \__box_rotate_quadrant_three: }
14862   { \__box_rotate_quadrant_four: }
14863 }

```

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.

```

14864 \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
14865 \hbox_set:Nn \l__box_internal_box
14866 {
14867   \tex_kern:D -\l__box_left_new_dim
14868   \hbox:n
14869   {
14870     \__driver_box_rotate_begin:
14871     \box_use:N \l__box_internal_box
14872     \__driver_box_rotate_end:
14873   }
14874 }

```

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.

```

14875 \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
14876 \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
14877 \box_set_wd:Nn \l__box_internal_box
14878 { \l__box_right_new_dim - \l__box_left_new_dim }
14879 \box_use:N \l__box_internal_box
14880 }

```

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.

```

14881 \cs_new_protected:Npn \__box_rotate_x:nnN #1#2#3
14882 {
14883   \dim_set:Nn #3
14884   {

```



```

14885         \fp_to_dim:n
14886         {
14887             \l__box_cos_fp * \dim_to_fp:n {#1}
14888             - \l__box_sin_fp * \dim_to_fp:n {#2}
14889         }
14890     }
14891 }
14892 \cs_new_protected:Npn \__box_rotate_y:nnN #1#2#3
14893 {
14894     \dim_set:Nn #3
14895     {
14896         \fp_to_dim:n
14897         {
14898             \l__box_sin_fp * \dim_to_fp:n {#1}
14899             + \l__box_cos_fp * \dim_to_fp:n {#2}
14900         }
14901     }
14902 }

```

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.

```

14903 \cs_new_protected:Npn \__box_rotate_quadrant_one:
14904 {
14905     \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
14906     \l__box_top_new_dim
14907     \__box_rotate_y:nnN \l__box_left_dim \l__box_bottom_dim
14908     \l__box_bottom_new_dim
14909     \__box_rotate_x:nnN \l__box_left_dim \l__box_top_dim
14910     \l__box_left_new_dim
14911     \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
14912     \l__box_right_new_dim
14913 }
14914 \cs_new_protected:Npn \__box_rotate_quadrant_two:
14915 {
14916     \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
14917     \l__box_top_new_dim
14918     \__box_rotate_y:nnN \l__box_left_dim \l__box_top_dim
14919     \l__box_bottom_new_dim
14920     \__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
14921     \l__box_left_new_dim
14922     \__box_rotate_x:nnN \l__box_left_dim \l__box_bottom_dim
14923     \l__box_right_new_dim
14924 }
14925 \cs_new_protected:Npn \__box_rotate_quadrant_three:
14926 {
14927     \__box_rotate_y:nnN \l__box_left_dim \l__box_bottom_dim
14928     \l__box_top_new_dim

```

```

14929 \l__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
14930 \l__box_bottom_new_dim
14931 \l__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
14932 \l__box_left_new_dim
14933 \l__box_rotate_x:nnN \l__box_left_dim \l__box_top_dim
14934 \l__box_right_new_dim
14935 }
14936 \cs_new_protected:Npn \l__box_rotate_quadrant_four:
14937 {
14938 \l__box_rotate_y:nnN \l__box_left_dim \l__box_top_dim
14939 \l__box_top_new_dim
14940 \l__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
14941 \l__box_bottom_new_dim
14942 \l__box_rotate_x:nnN \l__box_left_dim \l__box_bottom_dim
14943 \l__box_left_new_dim
14944 \l__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
14945 \l__box_right_new_dim
14946 }

```

(End definition for `\box_rotate:Nn`. This function is documented on page 196.)

`\l__box_scale_x_fp` Scaling is potentially-different in the two axes.
`\l__box_scale_y_fp`

```

14947 \fp_new:N \l__box_scale_x_fp
14948 \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 197.)

`\box_resize:Nnn` Resizing a box starts by working out the various dimensions of the existing box.

```

\box_resize:cnn
\__box_resize_set_corners:N
\__box_resize:Nn
14949 \cs_new_protected:Npn \box_resize:Nnn #1#2#3
14950 {
14951 \hbox_set:Nn #1
14952 {
14953 \group_begin:
14954 \__box_resize_set_corners:N #1

```

The x -scaling and resulting box size is easy enough to work out: the dimension is that given as #2, and the scale is simply the new width divided by the old one.

```

14955 \fp_set:Nn \l__box_scale_x_fp
14956 { \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.

```

14957 \fp_set:Nn \l__box_scale_y_fp
14958 {
14959 \dim_to_fp:n {#3}
14960 / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
14961 }

```

Hand off to the auxiliary which does the work.

```

14962 \__box_resize:Nn #1 {#2}
14963 \group_end:
14964 }

```

```

14965 }
14966 \cs_generate_variant:Nn \box_resize:Nnn { c }
14967 \cs_new_protected:Npn \__box_resize_set_corners:N #1
14968 {
14969   \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
14970   \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
14971   \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
14972   \dim_zero:N \l__box_left_dim
14973 }

```

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.

```

14974 \cs_new_protected:Npn \__box_resize:Nn #1#2
14975 {
14976   \dim_set:Nn \l__box_right_new_dim { \dim_abs:n {#2} }
14977   \dim_set:Nn \l__box_bottom_new_dim
14978     { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
14979   \dim_set:Nn \l__box_top_new_dim
14980     { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
14981   \__box_resize_common:N #1
14982 }

```

(End definition for `\box_resize:Nnn` and `\box_resize:cnn`. These functions are documented on page ??.)

```

\box_resize_to_ht:Nn
\box_resize_to_ht:cn
\box_resize_to_ht_plus_dp:Nn
\box_resize_to_ht_plus_dp:cn
\box_resize_to_wd:Nn
\box_resize_to_wd:cn
\box_resize_to_wd_and_ht:Nnn
\box_resize_to_wd_and_ht:cnn

```

Scaling to a (total) height or to a width is a simplified version of the main resizing operation, with the scale simply copied between the two parts. The internal auxiliary is called using the scaling value twice, as the sign for both parts is needed (as this allows the same internal code to be used as for the general case).

```

14983 \cs_new_protected:Npn \box_resize_to_ht:Nn #1#2
14984 {
14985   \hbox_set:Nn #1
14986   {
14987     \group_begin:
14988       \__box_resize_set_corners:N #1
14989       \fp_set:Nn \l__box_scale_y_fp
14990         {
14991           \dim_to_fp:n {#2}
14992           / \dim_to_fp:n { \l__box_top_dim }
14993         }
14994       \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
14995       \__box_resize:Nn #1 {#2}
14996     \group_end:
14997   }
14998 }
14999 \cs_generate_variant:Nn \box_resize_to_ht:Nn { c }
15000 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2

```

```

15001 {
15002   \hbox_set:Nn #1
15003   {
15004     \group_begin:
15005     \__box_resize_set_corners:N #1
15006     \fp_set:Nn \l__box_scale_y_fp
15007     {
15008       \dim_to_fp:n {#2}
15009       / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
15010     }
15011     \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
15012     \__box_resize:Nn #1 {#2}
15013   \group_end:
15014   }
15015 }
15016 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
15017 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
15018 {
15019   \hbox_set:Nn #1
15020   {
15021     \group_begin:
15022     \__box_resize_set_corners:N #1
15023     \fp_set:Nn \l__box_scale_x_fp
15024     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
15025     \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
15026     \__box_resize:Nn #1 {#2}
15027   \group_end:
15028   }
15029 }
15030 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
15031 \cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
15032 {
15033   \hbox_set:Nn #1
15034   {
15035     \group_begin:
15036     \__box_resize_set_corners:N #1
15037     \fp_set:Nn \l__box_scale_x_fp
15038     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
15039     \fp_set:Nn \l__box_scale_y_fp
15040     {
15041       \dim_to_fp:n {#3}
15042       / \dim_to_fp:n { \l__box_top_dim }
15043     }
15044     \__box_resize:Nn #1 {#2}
15045   \group_end:
15046   }
15047 }
15048 \cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }

```

(End definition for \box_resize_to_ht:Nn and \box_resize_to_ht:cn. These functions are documented on page ??.)

`\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.

```

15049 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
15050 {
15051   \hbox_set:Nn #1
15052   {
15053     \group_begin:
15054       \fp_set:Nn \l__box_scale_x_fp {#2}
15055       \fp_set:Nn \l__box_scale_y_fp {#3}
15056       \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
15057       \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
15058       \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
15059       \dim_zero:N \l__box_left_dim
15060       \dim_set:Nn \l__box_top_new_dim
15061         { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
15062       \dim_set:Nn \l__box_bottom_new_dim
15063         { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
15064       \dim_set:Nn \l__box_right_new_dim
15065         { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
15066       \__box_resize_common:N #1
15067     \group_end:
15068   }
15069 }
15070 \cs_generate_variant:Nn \box_scale:Nnn { c }

```

(End definition for `\box_scale:Nnn` and `\box_scale:cnn`. 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.

```

15071 \cs_new_protected:Npn \__box_resize_common:N #1
15072 {
15073   \hbox_set:Nn \l__box_internal_box
15074   {
15075     \__driver_box_scale_begin:
15076     \hbox_overlap_right:n { \box_use:N #1 }
15077     \__driver_box_scale_end:
15078   }

```

The new height and depth can be applied directly.

```

15079 \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
15080 {
15081   \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
15082   \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
15083 }
15084 {
15085   \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
15086   \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }

```

```
15087     }
```

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.

```
15088     \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
15089     {
15090         \hbox_to_wd:nn { \l__box_right_new_dim }
15091         {
15092             \tex_kern:D \l__box_right_new_dim
15093             \box_use:N \l__box_internal_box
15094             \tex_hss:D
15095         }
15096     }
15097     {
15098         \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
15099         \hbox:n
15100         {
15101             \tex_kern:D \c_zero_dim
15102             \box_use:N \l__box_internal_box
15103             \tex_hss:D
15104         }
15105     }
15106 }
```

(End definition for `__box_resize_common:N`.)

35.3 Viewing part of a box

`\box_clip:N` A wrapper around the driver-dependent code.

```
\box_clip:c 15107 \cs_new_protected:Npn \box_clip:N #1
15108     { \hbox_set:Nn #1 { \__driver_box_use_clip:N #1 } }
15109 \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.

`\box_trim:cnnnn`

```
15110 \cs_new_protected:Npn \box_trim:Nnnnn #1#2#3#4#5
15111 {
15112     \hbox_set:Nn \l__box_internal_box
15113     {
15114         \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
15115         \box_use:N #1
15116         \tex_kern:D -\__dim_eval:w #4 \__dim_eval_end:
15117     }
```

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`.

```

15118 \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
15119 {
15120   \hbox_set:Nn \l__box_internal_box
15121   {
15122     \box_move_down:nn \c_zero_dim
15123     { \box_use:N \l__box_internal_box }
15124   }
15125   \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
15126 }
15127 {
15128   \hbox_set:Nn \l__box_internal_box
15129   {
15130     \box_move_down:nn { #3 - \box_dp:N #1 }
15131     { \box_use:N \l__box_internal_box }
15132   }
15133   \box_set_dp:Nn \l__box_internal_box \c_zero_dim
15134 }

```

Same thing, this time from the top of the box.

```

15135 \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
15136 {
15137   \hbox_set:Nn \l__box_internal_box
15138   { \box_move_up:nn \c_zero_dim { \box_use:N \l__box_internal_box } }
15139   \box_set_ht:Nn \l__box_internal_box
15140   { \box_ht:N \l__box_internal_box - (#5) }
15141 }
15142 {
15143   \hbox_set:Nn \l__box_internal_box
15144   {
15145     \box_move_up:nn { #5 - \box_ht:N \l__box_internal_box }
15146     { \box_use:N \l__box_internal_box }
15147   }
15148   \box_set_ht:Nn \l__box_internal_box \c_zero_dim
15149 }
15150 \box_set_eq:NN #1 \l__box_internal_box
15151 }
15152 \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.

```

15153 \cs_new_protected:Npn \box_viewport:Nnnnn #1#2#3#4#5
15154 {
15155   \hbox_set:Nn \l__box_internal_box
15156   {

```

```

15157 \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
15158 \box_use:N #1
15159 \tex_kern:D \__dim_eval:w #4 - \box_wd:N #1 \__dim_eval_end:
15160 }
15161 \dim_compare:nNnTF {#3} < \c_zero_dim
15162 {
15163   \hbox_set:Nn \l__box_internal_box
15164   {
15165     \box_move_down:nn \c_zero_dim
15166     { \box_use:N \l__box_internal_box }
15167   }
15168   \box_set_dp:Nn \l__box_internal_box { -\dim_eval:n {#3} }
15169 }
15170 {
15171   \hbox_set:Nn \l__box_internal_box
15172   { \box_move_down:nn {#3} { \box_use:N \l__box_internal_box } }
15173   \box_set_dp:Nn \l__box_internal_box \c_zero_dim
15174 }
15175 \dim_compare:nNnTF {#5} > \c_zero_dim
15176 {
15177   \hbox_set:Nn \l__box_internal_box
15178   { \box_move_up:nn \c_zero_dim { \box_use:N \l__box_internal_box } }
15179   \box_set_ht:Nn \l__box_internal_box
15180   {
15181     #5
15182     \dim_compare:nNnT {#3} > \c_zero_dim
15183     { - (#3) }
15184   }
15185 }
15186 {
15187   \hbox_set:Nn \l__box_internal_box
15188   {
15189     \box_move_up:nn { -\dim_eval:n {#5} }
15190     { \box_use:N \l__box_internal_box }
15191   }
15192   \box_set_ht:Nn \l__box_internal_box \c_zero_dim
15193 }
15194 \box_set_eq:NN #1 \l__box_internal_box
15195 }
15196 \cs_generate_variant:Nn \box_viewport:Nnnnn { c }

```

(End definition for \box_viewport:Nnnnn and \box_viewport:cnnnn. These functions are documented on page ??.)

35.4 Additions to l3coffins

15197 <@@=coffin>

35.5 Rotating coffins

\l__coffin_sin_fp Used for rotations to get the sine and cosine values.
 \l__coffin_cos_fp


```

15198 \fp_new:N \l__coffin_sin_fp
15199 \fp_new:N \l__coffin_cos_fp
(End definition for \l__coffin_sin_fp. This variable is documented on page ??.)

```

\l__coffin_bounding_prop A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```

15200 \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.

```

15201 \dim_new:N \l__coffin_bounding_shift_dim
(End definition for \l__coffin_bounding_shift_dim. This variable is documented on page ??.)

```

\l__coffin_left_corner_dim These are used to hold maxima for the various corner values: these thus define the minimum size of the bounding box after rotation.

\l__coffin_right_corner_dim

\l__coffin_bottom_corner_dim

\l__coffin_top_corner_dim

```

15202 \dim_new:N \l__coffin_left_corner_dim
15203 \dim_new:N \l__coffin_right_corner_dim
15204 \dim_new:N \l__coffin_bottom_corner_dim
15205 \dim_new:N \l__coffin_top_corner_dim
(End definition for \l__coffin_left_corner_dim. This variable is documented on page ??.)

```

\coffin_rotate:Nn Rotating a coffin requires several steps which can be conveniently run together. The sine and cosine of the angle in degrees are computed. This is then used to set **\l__coffin_sin_fp** and **\l__coffin_cos_fp**, which are carried through unchanged for the rest of the procedure.

\coffin_rotate:cn

```

15206 \cs_new_protected:Npn \coffin_rotate:Nn #1#2
15207 {
15208   \fp_set:Nn \l__coffin_sin_fp { sind ( #2 ) }
15209   \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.

```

15210 \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
15211 { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
15212 \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
15213 { \__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.

```

15214 \__coffin_set_bounding:N #1
15215 \prop_map_inline:Nn \l__coffin_bounding_prop
15216 { \__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.

```

15217 \__coffin_find_corner_maxima:N #1
15218 \__coffin_find_bounding_shift:
15219 \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.

```

15220 \hbox_set:Nn \l__coffin_internal_box
15221 {
15222   \tex_kern:D
15223   \__dim_eval:w
15224   \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim
15225   \__dim_eval_end:
15226   \box_move_down:nn { \l__coffin_bottom_corner_dim }
15227   { \box_use:N #1 }
15228 }

```

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.

```

15229 \box_set_ht:Nn \l__coffin_internal_box
15230 { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
15231 \box_set_dp:Nn \l__coffin_internal_box { 0 pt }
15232 \box_set_wd:Nn \l__coffin_internal_box
15233 { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
15234 \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.

```

15235 \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
15236 { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
15237 \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
15238 { \__coffin_shift_pole:Nnnnnn #1 {##1} ##2 }
15239 }
15240 \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.

```

15241 \cs_new_protected:Npn \__coffin_set_bounding:N #1
15242 {
15243   \prop_put:Nnx \l__coffin_bounding_prop { tl }
15244   { { 0 pt } { \dim_use:N \box_ht:N #1 } }
15245   \prop_put:Nnx \l__coffin_bounding_prop { tr }
15246   { { \dim_use:N \box_wd:N #1 } { \dim_use:N \box_ht:N #1 } }
15247   \dim_set:Nn \l__coffin_internal_dim { - \box_dp:N #1 }
15248   \prop_put:Nnx \l__coffin_bounding_prop { bl }

```

```

15249     { { 0 pt } { \dim_use:N \l__coffin_internal_dim } }
15250     \prop_put:Nnx \l__coffin_bounding_prop { br }
15251     { { \dim_use:N \box_wd:N #1 } { \dim_use:N \l__coffin_internal_dim } }
15252   }

```

(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.

```

15253 \cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
15254 {
15255   \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
15256   \prop_put:Nnx \l__coffin_bounding_prop {#1}
15257   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
15258 }
15259 \cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
15260 {
15261   \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
15262   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
15263   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
15264 }

```

(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.

```

15265 \cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
15266 {
15267   \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
15268   \__coffin_rotate_vector:nnNN {#5} {#6}
15269   \l__coffin_x_prime_dim \l__coffin_y_prime_dim
15270   \__coffin_set_pole:Nnx #1 {#2}
15271   {
15272     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
15273     { \dim_use:N \l__coffin_x_prime_dim }
15274     { \dim_use:N \l__coffin_y_prime_dim }
15275   }
15276 }

```

(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.

```

15277 \cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
15278 {
15279   \dim_set:Nn #3
15280   {
15281     \fp_to_dim:n

```

```

15282         {
15283             \dim_to_fp:n {#1} * \l__coffin_cos_fp
15284             - \dim_to_fp:n {#2} * \l__coffin_sin_fp
15285         }
15286     }
15287     \dim_set:Nn #4
15288     {
15289         \fp_to_dim:n
15290         {
15291             \dim_to_fp:n {#1} * \l__coffin_sin_fp
15292             + \dim_to_fp:n {#2} * \l__coffin_cos_fp
15293         }
15294     }
15295 }

```

(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.

```

15296 \cs_new_protected:Npn \__coffin_find_corner_maxima:N #1
15297 {
15298     \dim_set:Nn \l__coffin_top_corner_dim { -\c_max_dim }
15299     \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
15300     \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
15301     \dim_set:Nn \l__coffin_left_corner_dim { \c_max_dim }
15302     \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
15303     { \__coffin_find_corner_maxima_aux:nn ##2 }
15304 }
15305 \cs_new_protected:Npn \__coffin_find_corner_maxima_aux:nn #1#2
15306 {
15307     \dim_set:Nn \l__coffin_left_corner_dim
15308     { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
15309     \dim_set:Nn \l__coffin_right_corner_dim
15310     { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
15311     \dim_set:Nn \l__coffin_bottom_corner_dim
15312     { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
15313     \dim_set:Nn \l__coffin_top_corner_dim
15314     { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
15315 }

```

(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.

```

15316 \cs_new_protected_nopar:Npn \__coffin_find_bounding_shift:
15317 {
15318     \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
15319     \prop_map_inline:Nn \l__coffin_bounding_prop

```

```

15320         { \__coffin_find_bounding_shift_aux:nn ##2 }
15321     }
15322 \cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
15323 {
15324     \dim_set:Nn \l__coffin_bounding_shift_dim
15325     { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
15326 }

```

(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.

```

15327 \cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
15328 {
15329     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _ prop } {#2}
15330     {
15331         { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
15332         { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
15333     }
15334 }
15335 \cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
15336 {
15337     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _ prop } {#2}
15338     {
15339         { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
15340         { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
15341         {#5} {#6}
15342     }
15343 }

```

(End definition for __coffin_shift_corner:Nnnn. This function is documented on page ??.)

35.6 Resizing coffins

\l__coffin_scale_x_fp
\l__coffin_scale_y_fp

Storage for the scaling factors in x and y , respectively.

```

15344 \fp_new:N \l__coffin_scale_x_fp
15345 \fp_new:N \l__coffin_scale_y_fp

```

(End definition for \l__coffin_scale_x_fp. This variable is documented on page ??.)

\l__coffin_scaled_total_height_dim
\l__coffin_scaled_width_dim

When scaling, the values given have to be turned into absolute values.

```

15346 \dim_new:N \l__coffin_scaled_total_height_dim
15347 \dim_new:N \l__coffin_scaled_width_dim

```

(End definition for \l__coffin_scaled_total_height_dim. This variable is documented on page ??.)

\coffin_resize:Nnn
\coffin_resize:cnn

Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.

```

15348 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3

```

```

15349 {
15350   \fp_set:Nn \l__coffin_scale_x_fp
15351   { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
15352   \fp_set:Nn \l__coffin_scale_y_fp
15353   {
15354     \dim_to_fp:n {#3}
15355     / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
15356   }
15357   \box_resize:Nnn #1 {#2} {#3}
15358   \__coffin_resize_common:Nnn #1 {#2} {#3}
15359 }
15360 \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.

```

15361 \cs_new_protected:Npn \__coffin_resize_common:Nnn #1#2#3
15362 {
15363   \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
15364   { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
15365   \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
15366   { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }

```

Negative x -scaling values will place the poles in the wrong location: this is corrected here.

```

15367   \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
15368   {
15369     \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
15370     { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
15371     \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
15372     { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
15373   }
15374 }

```

(End definition for __coffin_resize_common:Nnn. This function is documented on page ??.)

`\coffin_scale:Nnn` For scaling, the opposite calculation is done to find the new dimensions for the coffin.
`\coffin_scale:cnn` Only the total height is needed, as this is the shift required for corners and poles. The scaling is done the TeX way as this works properly with floating point values without needing to use the fp module.

```

15375 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
15376 {
15377   \fp_set:Nn \l__coffin_scale_x_fp {#2}
15378   \fp_set:Nn \l__coffin_scale_y_fp {#3}
15379   \box_scale:Nnn #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
15380   \dim_set:Nn \l__coffin_internal_dim
15381   { \coffin_ht:N #1 + \coffin_dp:N #1 }
15382   \dim_set:Nn \l__coffin_scaled_total_height_dim
15383   { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }

```

```

15384 \dim_set:Nn \l__coffin_scaled_width_dim
15385 { -\fp_abs:n { \l__coffin_scale_x_fp } \coffin_wd:N #1 }
15386 \__coffin_resize_common:Nnn #1
15387 { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
15388 }
15389 \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 function scales a vector from the origin using the pre-set scale factors in x and y . This is a much less complex operation than rotation, and as a result the code is a lot clearer.

```

15390 \cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
15391 {
15392   \dim_set:Nn #3
15393   { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
15394   \dim_set:Nn #4
15395   { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
15396 }

```

(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
15397 \cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
15398 {
15399   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
15400   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
15401   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
15402 }
15403 \cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
15404 {
15405   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
15406   \__coffin_set_pole:Nnx #1 {#2}
15407   {
15408     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
15409     {#5} {#6}
15410   }
15411 }

```

(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

```

15412 \cs_new_protected:Npn \__coffin_x_shift_corner:Nnnn #1#2#3#4
15413 {
15414   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
15415   {
15416     { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
15417   }
15418 }
15419 \cs_new_protected:Npn \__coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6

```

```

15420 {
15421     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } {#2}
15422     {
15423         { \dim_eval:n #3 + \box_wd:N #1 } {#4}
15424         {#5} {#6}
15425     }
15426 }

```

(End definition for __coffin_x_shift_corner:Nnnn. This function is documented on page ??.)

35.7 Additions to l3file

```

15427 <@@=file>

```

\file_if_exist_input:nTF Input of a file with a test for existence cannot be done the usual way as the tokens to insert are in an odd place.

```

15428 \cs_new_protected:Npn \file_if_exist_input:n #1
15429 {
15430     \file_if_exist:nT {#1}
15431     { \__file_input:V \l__file_internal_name_tl }
15432 }
15433 \cs_new_protected:Npn \file_if_exist_input:nT #1#2
15434 {
15435     \file_if_exist:nT {#1}
15436     {
15437         #2
15438         \__file_input:V \l__file_internal_name_tl
15439     }
15440 }
15441 \cs_new_protected:Npn \file_if_exist_input:nF #1
15442 {
15443     \file_if_exist:nTF {#1}
15444     { \__file_input:V \l__file_internal_name_tl }
15445 }
15446 \cs_new_protected:Npn \file_if_exist_input:nTF #1#2
15447 {
15448     \file_if_exist:nTF {#1}
15449     {
15450         #2
15451         \__file_input:V \l__file_internal_name_tl
15452     }
15453 }

```

(End definition for \file_if_exist_input:nTF. This function is documented on page 198.)

```

15454 <@@=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

```

15455 \cs_new_nopar:Npn \ior_map_break:
15456 { \__prg_map_break:Nn \ior_map_break: { } }
15457 \cs_new_nopar:Npn \ior_map_break:n

```



```
15458 { \_prg_map_break:Nn \ior_map_break: }
```

(End definition for \ior_map_break: and \ior_map_break:n. These functions are documented on page 199.)

```
\ior_map_inline:Nn
\ior_str_map_inline:Nn
\_ior_map_inline:NNn
\_ior_map_inline:NNNn
\_ior_map_inline_loop:NNN
\_l\_ior_internal_tl
```

Mapping to an input stream can be done on either a token or a string basis, hence the set up. Within that, there is a check to avoid reading past the end of a file, hence the two applications of \ior_if_eof:N. This mapping cannot be nested as the stream has only one “current line”.

```
15459 \cs_new_protected_nopar:Npn \ior_map_inline:Nn
15460 { \_ior_map_inline:NNn \ior_get:NN }
15461 \cs_new_protected_nopar:Npn \ior_str_map_inline:Nn
15462 { \_ior_map_inline:NNn \ior_get_str:NN }
15463 \cs_new_protected_nopar:Npn \_ior_map_inline:NNn
15464 {
15465   \int_gincr:N \g\_prg_map_int
15466   \exp_args:Nc \_ior_map_inline:NNNn
15467   { \_prg_map_ \int_use:N \g\_prg_map_int :n }
15468 }
15469 \cs_new_protected:Npn \_ior_map_inline:NNNn #1#2#3#4
15470 {
15471   \cs_set:Npn #1 ##1 {#4}
15472   \ior_if_eof:NF #3 { \_ior_map_inline_loop:NNN #1#2#3 }
15473   \_prg_break_point:Nn \ior_map_break:
15474   { \int_gdecr:N \g\_prg_map_int }
15475 }
15476 \cs_new_protected:Npn \_ior_map_inline_loop:NNN #1#2#3
15477 {
15478   #2 #3 \_l\_ior_internal_tl
15479   \ior_if_eof:NF #3
15480   {
15481     \exp_args:No #1 \_l\_ior_internal_tl
15482     \_ior_map_inline_loop:NNN #1#2#3
15483   }
15484 }
15485 \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 ??.)

35.8 Additions to l3prop

```
15486 <@@=prop>
```

```
\prop_map_tokens:Nn
\prop_map_tokens:cn
\_prop_map_tokens:nwwn
```

The mapping is very similar to \prop_map_function:NN. It grabs one key–value pair at a time, and stops when reaching the marker key \q_recursion_tail, which cannot appear in normal keys since those are strings. The odd construction \use:n {#1} allows #1 to contain any token without interfering with \prop_map_break:. Argument #2 of _prop_map_tokens:nwwn is \s_prop the first time, and is otherwise empty.

```
15487 \cs_new:Npn \prop_map_tokens:Nn #1#2
15488 {
```

```

15489 \exp_last_unbraced:Nno \__prop_map_tokens:nwn {#2} #1
15490 \__prop_pair:wn \q_recursion_tail \s__prop { }
15491 \__prg_break_point:Nn \prop_map_break: { }
15492 }
15493 \cs_new:Npn \__prop_map_tokens:nwn #1#2 \__prop_pair:wn #3 \s__prop #4
15494 {
15495 \if_meaning:w \q_recursion_tail #3
15496 \exp_after:wN \prop_map_break:
15497 \fi:
15498 \use:n {#1} {#3} {#4}
15499 \__prop_map_tokens:nwn {#1}
15500 }
15501 \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 ??.)

35.9 Additions to l3seq

```

15502 <@@=seq>

```

```

\seq_mapthread_function:NNN
\seq_mapthread_function:NcN
\seq_mapthread_function:cNN
\seq_mapthread_function:ccN
  \_seq_mapthread_function:wNN
  \_seq_mapthread_function:wNw
  \_seq_mapthread_function:Nnnwnn

```

The idea here is to first expand both sequences, adding the usual { ? __prg_break: } { } to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first tokens of #2 and #5, which for items in the sequences will both be \s__seq __seq_item:n. The function to be mapped will then be applied to the two entries. When the code hits the end of one of the sequences, the break material will stop the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

```

15503 \cs_new:Npn \seq_mapthread_function:NNN #1#2#3
15504 { \exp_after:wN \__seq_mapthread_function:wNN #2 \q_stop #1 #3 }
15505 \cs_new:Npn \__seq_mapthread_function:wNN \s__seq #1 \q_stop #2#3
15506 {
15507 \exp_after:wN \__seq_mapthread_function:wNw #2 \q_stop #3
15508 #1 { ? \__prg_break: } { }
15509 \__prg_break_point:
15510 }
15511 \cs_new:Npn \__seq_mapthread_function:wNw \s__seq #1 \q_stop #2
15512 {
15513 \_seq_mapthread_function:Nnnwnn #2
15514 #1 { ? \__prg_break: } { }
15515 \q_stop
15516 }
15517 \cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \q_stop #5#6
15518 {
15519 \use_none:n #2
15520 \use_none:n #5
15521 #1 {#3} {#6}
15522 \__seq_mapthread_function:Nnnwnn #1 #4 \q_stop
15523 }
15524 \cs_generate_variant:Nn \seq_mapthread_function:NNN { Nc }

```

```
15525 \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_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.

```
15526 \cs_new_protected_nopar:Npn \seq_set_filter:NNn
15527 { \__seq_set_filter:NNNn \tl_set:Nx }
15528 \cs_new_protected_nopar:Npn \seq_gset_filter:NNn
15529 { \__seq_set_filter:NNNn \tl_gset:Nx }
15530 \cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
15531 {
15532   \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
15533   #1 #2 { #3 }
15534   \__seq_pop_item_def:
15535 }
```

(End definition for `\seq_set_filter:NNn` and `\seq_gset_filter:NNn`. These functions are documented on page 200.)

```
\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.

```
15536 \cs_new_protected_nopar:Npn \seq_set_map:NNn
15537 { \__seq_set_map:NNNn \tl_set:Nx }
15538 \cs_new_protected_nopar:Npn \seq_gset_map:NNn
15539 { \__seq_set_map:NNNn \tl_gset:Nx }
15540 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
15541 {
15542   \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
15543   #1 #2 { #3 }
15544   \__seq_pop_item_def:
15545 }
```

(End definition for `\seq_set_map:NNn` and `\seq_gset_map:NNn`. These functions are documented on page 200.)

35.10 Additions to l3skip

```
15546 <@@=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.

```
15547 \cs_new:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
15548 {
15549   \skip_if_finite:nTF {#1}
15550   {
15551     #3 = \etex_gluestretch:D #1 \scan_stop:
```

```

15552         #4 = \etex_glueshrink:D #1 \scan_stop:
15553     }
15554     {
15555         #3 = \c_zero_skip
15556         #4 = \c_zero_skip
15557         #2
15558     }
15559 }

```

(End definition for `\skip_split_finite_else_action:nnNN`. This function is documented on page 201.)

35.11 Additions to l3tl

```

15560 <@@=tl>

```

`\tl_if_single_token:p:n` There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. Otherwise, compare with a single space, only case where we have a single token.

```

15561 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
15562 {
15563     \tl_if_head_is_N_type:nTF {#1}
15564     { \__str_if_eq_x_return:nn { \exp_not:o { \use_none:n #1 } } { } }
15565     { \__str_if_eq_x_return:nn { \exp_not:n {#1} } { ~ } }
15566 }

```

(End definition for `\tl_if_single_token:n`. These functions are documented on page 201.)

`\tl_reverse_tokens:n`
`__tl_reverse_group:nn`

The same as `\tl_reverse:n` but with recursion within brace groups.

```

15567 \cs_new:Npn \tl_reverse_tokens:n #1
15568 {
15569     \etex_unexpanded:D \exp_after:wN
15570     {
15571         \tex_romannumeral:D
15572         \__tl_act:NNNnn
15573         \__tl_reverse_normal:nN
15574         \__tl_reverse_group:nn
15575         \__tl_reverse_space:n
15576         { }
15577         {#1}
15578     }
15579 }
15580 \cs_new:Npn \__tl_reverse_group:nn #1
15581 {
15582     \__tl_act_group_recurse:Nnn
15583     \__tl_act_reverse_output:n
15584     { \tl_reverse_tokens:n }
15585 }

```

`__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.

```

15586 \cs_new:Npn \__tl_act_group_recurse:Nnn #1#2#3
15587 {
15588   \exp_args:Nf #1
15589   { \exp_after:wN \exp_after:wN \exp_after:wN { #2 {#3} } }
15590 }

```

(End definition for `\tl_reverse_tokens:n`. This function is documented on page 201.)

`\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.

```

\__tl_act_count_normal:nN
\__tl_act_count_group:nn
\__tl_act_count_space:n
15591 \cs_new:Npn \tl_count_tokens:n #1
15592 {
15593   \int_eval:n
15594   {
15595     \__tl_act:NNNnn
15596     \__tl_act_count_normal:nN
15597     \__tl_act_count_group:nn
15598     \__tl_act_count_space:n
15599     { }
15600     {#1}
15601   }
15602 }
15603 \cs_new:Npn \__tl_act_count_normal:nN #1 #2 { 1 + }
15604 \cs_new:Npn \__tl_act_count_space:n #1 { 1 + }
15605 \cs_new:Npn \__tl_act_count_group:nn #1 #2
15606 { 2 + \tl_count_tokens:n {#2} + }

```

(End definition for `\tl_count_tokens:n`. This function is documented on page 201.)

`\c__tl_act_uppercase_tl` These constants contain the correspondence between lowercase and uppercase letters, in the form `aAbBcC...` and `AaBbCc...` respectively.

```

\c__tl_act_lowercase_tl
15607 \tl_const:Nn \c__tl_act_uppercase_tl
15608 {
15609   aA bB cC dD eE fF gG hH iI jJ kK lL mM
15610   nN oO pP qQ rR sS tT uU vV wW xX yY zZ
15611 }
15612 \tl_const:Nn \c__tl_act_lowercase_tl
15613 {
15614   Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm
15615   Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz
15616 }

```

(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,

```

\__tl_act_case_normal:nN
\__tl_act_case_group:nn
\__tl_act_case_space:n

```

and this time, we use the $\langle parameters \rangle$ 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 $\backslash\text{str_if_eq:nn}$ tests, and converted if necessary to upper/lowercase, before being output. For a group, we must perform the conversion within the group (the $\backslash\text{exp_after:wN}$ trigger $\backslash\text{romannumeral}$, which expands fully to give the converted group), then output.

```

15617 \cs_new:Npn \tl_expandable_uppercase:n #1
15618 {
15619   \etex_unexpanded:D \exp_after:wN
15620   {
15621     \tex_romannumeral:D
15622     \__tl_act_case_aux:nn { \c__tl_act_uppercase_tl } {#1}
15623   }
15624 }
15625 \cs_new:Npn \tl_expandable_lowercase:n #1
15626 {
15627   \etex_unexpanded:D \exp_after:wN
15628   {
15629     \tex_romannumeral:D
15630     \__tl_act_case_aux:nn { \c__tl_act_lowercase_tl } {#1}
15631   }
15632 }
15633 \cs_new:Npn \__tl_act_case_aux:nn
15634 {
15635   \__tl_act:NNNnn
15636   \__tl_act_case_normal:nN
15637   \__tl_act_case_group:nn
15638   \__tl_act_case_space:n
15639 }
15640 \cs_new:Npn \__tl_act_case_space:n #1 { \__tl_act_output:n {~} }
15641 \cs_new:Npn \__tl_act_case_normal:nN #1 #2
15642 {
15643   \exp_args:Nf \__tl_act_output:n
15644   {
15645     \exp_args:NNo \str_case:nnF #2 {#1}
15646     { \exp_stop_f: #2 }
15647   }
15648 }
15649 \cs_new:Npn \__tl_act_case_group:nn #1 #2
15650 {
15651   \exp_after:wN \__tl_act_output:n \exp_after:wN
15652   { \exp_after:wN { \tex_romannumeral:D \__tl_act_case_aux:nn {#1} {#2} } }
15653 }

```

(End definition for $\backslash\text{tl_expandable_uppercase:n}$ and $\backslash\text{tl_expandable_lowercase:n}$. These functions are documented on page 202.)

```

\tl_set_from_file:Nnn
\tl_set_from_file:cnn
\tl_gset_from_file:Nnn
\tl_gset_from_file:cnn
\__tl_set_from_file:NNnn
\__tl_from_file_do:w

```

The approach here is similar to that for doing a rescan, and so the same internals can be reused. Thus the plan is to insert a pair of tokens of the same charcode but different

catcodes after the file has been read. This plus `\exp_not:N` allows the primitive to be used to carry out a set operation.

```

15654 \cs_new_protected_nopar:Npn \tl_set_from_file:Nnn
15655 { \tl_set_from_file:NNnn \tl_set:Nn }
15656 \cs_new_protected_nopar:Npn \tl_gset_from_file:Nnn
15657 { \tl_set_from_file:NNnn \tl_gset:Nn }
15658 \cs_generate_variant:Nn \tl_set_from_file:Nnn { c }
15659 \cs_generate_variant:Nn \tl_gset_from_file:Nnn { c }
15660 \cs_new_protected:Npn \tl_set_from_file:NNnn #1#2#3#4
15661 {
15662   \__file_if_exist:nT {#4}
15663   {
15664     \group_begin:
15665     \exp_args:No \etex_everyeof:D
15666     { \c_tl_rescan_marker_tl \exp_not:N }
15667     #3 \scan_stop:
15668     \exp_after:wN \tl_from_file_do:w
15669     \exp_after:wN \prg_do_nothing:
15670     \tex_input:D \l__file_internal_name_tl \scan_stop:
15671     \exp_args:NNNo \group_end:
15672     #1 #2 \l__tl_internal_a_tl
15673   }
15674 }
15675 \exp_args:Nno \use:nn
15676 { \cs_set_protected:Npn \tl_from_file_do:w #1 }
15677 { \c_tl_rescan_marker_tl }
15678 { \tl_set:No \l__tl_internal_a_tl {#1} }

```

(End definition for `\tl_set_from_file:Nnn` and others. These functions are documented on page ??.)

When reading a file and allowing expansion of the content, the set up only needs to prevent TeX complaining about the end of the file. That is done simply, with a group then used to trap the definition needed. Once the business is done using some scratch space, the tokens can be transferred to the real target.

`\tl_set_from_file_x:Nnn`
`\tl_set_from_file_x:cnn`
`\tl_gset_from_file_x:Nnn`
`\tl_gset_from_file_x:cnn`
`__tl_set_from_file_x:NNnn`

```

15679 \cs_new_protected_nopar:Npn \tl_set_from_file_x:Nnn
15680 { \tl_set_from_file_x:NNnn \tl_set:Nn }
15681 \cs_new_protected_nopar:Npn \tl_gset_from_file_x:Nnn
15682 { \tl_set_from_file_x:NNnn \tl_gset:Nn }
15683 \cs_generate_variant:Nn \tl_set_from_file_x:Nnn { c }
15684 \cs_generate_variant:Nn \tl_gset_from_file_x:Nnn { c }
15685 \cs_new_protected:Npn \tl_set_from_file_x:NNnn #1#2#3#4
15686 {
15687   \__file_if_exist:nT {#4}
15688   {
15689     \group_begin:
15690     \etex_everyeof:D { \exp_not:N }
15691     #3 \scan_stop:
15692     \tl_set:Nx \l__tl_internal_a_tl
15693     { \tex_input:D \l__file_internal_name_tl \c_space_token }
15694     \exp_args:NNNo \group_end:

```

```

15695         #1 #2 \l__tl_internal_a_tl
15696     }
15697 }

```

(End definition for `\tl_set_from_file_x:Nnn` and others. These functions are documented on page ??.)

35.11.1 Unicode case changing

The mechanisms needed for case changing are somewhat involved, particularly to allow for all of the special cases. These functions also require the appropriate data extracted from the Unicode documentation (either manually or automatically), which is covered by `l3unicode-data`.

`\tl_lower_case:n` `\tl_upper_case:n` `\tl_mixed_case:n` The user level functions here are all wrappers around the internal functions for case changing. Note that `\tl_mixed_case:nn` could be done without an internal, but this way the logic is slightly clearer as everything essentially follows the same path.

```

\tl_lower_case:nn 15698 \cs_new_nopar:Npn \tl_lower_case:n { \__tl_change_case:nnn { lower } { } }
\tl_upper_case:nn 15699 \cs_new_nopar:Npn \tl_upper_case:n { \__tl_change_case:nnn { upper } { } }
\tl_mixed_case:nn 15700 \cs_new_nopar:Npn \tl_mixed_case:n { \__tl_mixed_case:nn { } }
15701 \cs_new_nopar:Npn \tl_lower_case:nn { \__tl_change_case:nnn { lower } }
15702 \cs_new_nopar:Npn \tl_upper_case:nn { \__tl_change_case:nnn { upper } }
15703 \cs_new_nopar:Npn \tl_mixed_case:nn { \__tl_mixed_case:nn }

```

(End definition for `\tl_lower_case:n`, `\tl_upper_case:n`, and `\tl_mixed_case:n`. These functions are documented on page ??.)

```

\__tl_change_case:nnn
\__tl_change_case_loop:wnn
  \_tl_change_case_N_type:Nwnn
\__tl_change_case_group:nwnn
\__tl_change_case_space:wnn
  \_tl_change_case_char:NNNNNNNn
  \_tl_change_case_lower_sigma:Nnn
  \_tl_change_case_upper_sigma:Nnn
  \_tl_change_case_mixed_sigma:Nnn
  \_tl_change_case_lower_sigma:Nw
  \_tl_change_case_lower_sigma_loop:Nw

```

The mechanism for the core conversion of case is based on the approach used in `__tl_act:NNNnn`. Thus the idea is to use a loop which will grab the entire token list plus a quark: the latter is used as an end marker and to avoid any brace stripping. Depending on the nature of the first item in the grabbed argument, it can either be processed as a single token, treated as a group or treated as a space (the latter requires special treatment). In contrast to `__tl_act:NNNnn`, there is no need for this process to be `f`-type expandable: things are done using only `x`-type requirements. Also, for “normal” tokens there is a bit more work to do here: to allow selection of case matches using character code, it’s important that control sequences are filtered out before doing the lookup.

```

15704 \cs_new:Npn \__tl_change_case:nnn #1#2#3
15705 {
15706   \__tl_change_case_loop:wnn #3
15707   \q_recursion_tail \q_recursion_stop {#1} {#2}
15708 }
15709 \cs_new:Npn \__tl_change_case_loop:wnn #1 \q_recursion_stop
15710 {
15711   \tl_if_head_is_N_type:nTF {#1}
15712   { \__tl_change_case_N_type:Nwnn }
15713   {
15714     \tl_if_head_is_group:nTF {#1}
15715     { \__tl_change_case_group:nwnn }
15716     { \__tl_change_case_space:wnn }
15717   }
15718   #1 \q_recursion_stop

```



```

15719 }
15720 \cs_new:Npn \__tl_change_case_N_type:Nwnn #1#2 \q_recursion_stop #3#4
15721 {
15722   \quark_if_recursion_tail_stop_do:Nn #1 { \use_none:nn }
15723   \token_if_cs:NTF #1
15724     { \exp_not:N #1 }
15725     {
15726       \cs_if_exist_use:cF { __tl_change_case_ #3 _ #4 :Nnn }
15727       { \use_iii:nnn }
15728       #1 {#2}
15729       {
15730         \use:c { __tl_change_case_ #3 _ sigma:Nnn } #1 {#2}
15731         {
15732           \exp_after:wN \__tl_change_case_char:NNNNNNNNn
15733           \int_use:N \__int_eval:w 1000000 + '#1 \__int_eval_end:
15734             #1 {#3}
15735         }
15736       }
15737     }
15738   \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
15739 }
15740 \cs_new:Npn \__tl_change_case_group:nwnn #1#2 \q_recursion_stop
15741 {
15742   { \exp_not:n {#1} }
15743   \__tl_change_case_loop:wnn #2 \q_recursion_stop
15744 }
15745 \exp_last_unbraced:NNo \cs_new:Npn \__tl_change_case_space:wnn \c_space_tl
15746 {
15747   \c_space_tl
15748   \__tl_change_case_loop:wnn
15749 }

```

Actually look for the char in the appropriate table.

```

15750 \cs_new:Npn \__tl_change_case_char:NNNNNNNNn #1#2#3#4#5#6#7#8#9
15751 {
15752   \exp_args:NNv \str_case:nnF #8
15753     { c__tl_ #9 _ #6 _ #7 _tl }
15754     { \exp_not:N #8 }
15755 }

```

If the current char is an upper case sigma, the a check is made on the next item in the input. If it is another N-type token then further tests are needed to decide what to do.

```

15756 \cs_new:Npn \__tl_change_case_lower_sigma:Nnn #1#2
15757 {
15758   \int_compare:nNnTF { '#1 } = { "03A3 }
15759   {
15760     \tl_if_head_is_N_type:nTF {#2}
15761     { \__tl_change_case_lower_sigma:Nw #2 \q_recursion_stop }
15762     {
15763       \tl_if_head_is_group:nTF {#2}

```

```

15764         { \c__tl_std_sigma_tl }
15765         { \c__tl_final_sigma_tl }
15766     }
15767 }
15768 }

```

Assuming the next token is not a control sequence, a loop is used to test if the next char is something that can be interpreted as the end of a word. Rather than use all of the Unicode data for this, the simplifying assumption is made that in real text the end of a word will be indicated by a small number of chars. As this may have to be extended over time to other cases, the easiest handling is offered by using the numerical values for these chars.

```

15769 \cs_new:Npn \__tl_change_case_lower_sigma:Nw #1#2 \q_recursion_stop
15770 {
15771     \token_if_cs:NTF #1
15772     { \c__tl_std_sigma_tl }
15773     {
15774         \exp_after:wN \__tl_change_case_lower_sigma_loop:Nw
15775         \exp_after:wN #1 \c__tl_after_final_sigma_clist
15776         , \q_recursion_tail , \q_recursion_stop
15777     }
15778 }
15779 \cs_new:Npn \__tl_change_case_lower_sigma_loop:Nw #1#2 ,
15780 {
15781     \quark_if_recursion_tail_stop_do:nn {#2}
15782     { \c__tl_std_sigma_tl }
15783     \int_compare:nNnT { '#1 } = { "#2 }
15784     { \use_i_delimit_by_q_recursion_stop:nw { \c__tl_final_sigma_tl } }
15785     \__tl_change_case_lower_sigma_loop:Nw #1
15786 }
15787 \cs_new_eq:NN \__tl_change_case_upper_sigma:Nnn \use_iii:nnn
15788 \cs_new_eq:NN \__tl_change_case_mixed_sigma:Nnn \use_iii:nnn
(End definition for \__tl_change_case:nnn.)

```

```

\__tl_mixed_case:nn
\__tl_mixed_case_loop:wn
\__tl_mixed_case_N_type:Nwn
\__tl_mixed_case_skip:Nwn
\__tl_mixed_case_skip_tidy:nNwn
\__tl_mixed_case_group:nwn
\__tl_mixed_case_space:wn

```

Mixed (title) casing needs an entire set of functions to itself. These are more or less the same as those for general case changes but the requirements here are subtly different. What is needed is a loop which looks for the first “real” char in the input (skipping any pre-letter chars). Once one is found, it is case changed to upper case but first checking that there is not an entry in the exceptions list. Once that process is done, all remaining chars are lower cased so there is a switch to the normal system. Note that simply grabbing the first token in the input is no good here: it can’t handle pre-letter tokens or any special treatment of the first letter found (*e.g.* words starting with *i* in Turkish). Spaces at the start of the input are passed through without counting as being the “start” of the first word, while a brace group forces a switch to the standard lower casing routine.

```

15789 \cs_new:Npn \__tl_mixed_case:nn #1#2
15790 {
15791     \__tl_mixed_case_loop:wn #2
15792     \q_recursion_tail \q_recursion_stop {#1}
15793 }

```

```

15794 \cs_new:Npn \__tl_mixed_case_loop:wn #1 \q_recursion_stop
15795 {
15796   \tl_if_head_is_N_type:nTF {#1}
15797   { \__tl_mixed_case_N_type:Nwn }
15798   {
15799     \tl_if_head_is_group:nTF {#1}
15800     { \__tl_mixed_case_group:nwn }
15801     { \__tl_mixed_case_space:wn }
15802   }
15803   #1 \q_recursion_stop
15804 }
15805 \cs_new:Npn \__tl_mixed_case_N_type:Nwn #1#2 \q_recursion_stop #3
15806 {
15807   \quark_if_recursion_tail_stop_do:nN #1 { \use_none:n }
15808   \token_if_cs:NTF #1
15809   { \exp_not:N #1 }
15810   {
15811     \cs_if_exist_use:cF { __tl_change_case_mixed_ #3 :Nnn }
15812     {
15813       \cs_if_exist_use:cF { __tl_change_case_upper_ #3 :Nnn }
15814       { \use_iii:nnn }
15815     }
15816     #1 {#2}
15817     {
15818       \exp_after:wN \__tl_mixed_case_skip:Nwn \exp_after:wN #1
15819       \c__tl_mixed_skip_clist , \q_recursion_tail , \q_recursion_stop
15820       {
15821         \exp_args:NNV \str_case:nnF #1 \c__tl_mixed_exceptions_tl
15822         {
15823           \exp_after:wN \__tl_change_case_char:NNNNNNNNn
15824           \int_use:N \__int_eval:w 1000000 + ‘#1 \__int_eval_end:
15825           #1 { upper }
15826         }
15827       }
15828     }
15829   }
15830   \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } {#3}
15831 }

```

Looking for chars to skip when title casing uses the standard “loop around a list” approach. If there is a hit, there is a bit of tidying up to do: retain the char and switch the looping system to stick with the title loop rather than the lower case one. That means swapping an auxiliary and removing a trailing { lower }, which is easiest to do with a dedicated function.

```

15832 \cs_new:Npn \__tl_mixed_case_skip:Nwn #1#2 ,
15833 {
15834   \quark_if_recursion_tail_stop_do:nn {#2} { \use:n }
15835   \int_compare:nNnT { ‘#1 } = { “#2 }
15836   {
15837     \use_i_delimit_by_q_recursion_stop:nw

```

```

15838         {
15839             #1
15840             \_tl_mixed_case_skip_tidy:nNwn
15841         }
15842     }
15843     \_tl_mixed_case_skip:Nwn #1
15844 }
15845 \cs_new:Npn \_tl_mixed_case_skip_tidy:nNwn #1#2#3 \q_recursion_stop #4
15846 {
15847     \_tl_mixed_case_loop:wn #3 \q_recursion_stop
15848 }
15849 \cs_new:Npn \_tl_mixed_case_group:nwn #1#2 \q_recursion_stop
15850 {
15851     { \exp_not:n {#1} }
15852     \_tl_change_case_loop:wnn #2 \q_recursion_stop { lower }
15853 }
15854 \exp_last_unbraced:NNo \cs_new:Npn \_tl_mixed_case_space:wn \c_space_tl
15855 {
15856     \c_space_tl
15857     \_tl_mixed_case_loop:wn
15858 }

```

(End definition for _tl_mixed_case:nn.)

```

\_tl_change_case_lower_tr:Nnn
\_tl_change_case_lower_az:Nnn
\_tl_change_case_lower_tr:Nw
\_tl_change_case_upper_tr:Nnn
\_tl_change_case_upper_az:Nnn

```

The Turkic languages need special treatment for dotted-i and dotless-i. The lower casing rule can be expressed in terms of searching first for either a dotless-I or a dotted-I. In the latter case the mapping is easy, but in the former there is a second stage search.

```

15859 \cs_new:Npn \_tl_change_case_lower_tr:Nnn #1#2#3
15860 {
15861     \int_compare:nNnTF { '#1 } = { "0049 }
15862     {
15863         \tl_if_head_is_N_type:nTF {#2}
15864         { \_tl_change_case_lower_tr:Nw #2 \q_recursion_stop }
15865         { \c_tl_dotless_i_tl }
15866     }
15867     {
15868         \int_compare:nNnTF { '#1 } = { "0130 }
15869         { i }
15870         {#3}
15871     }
15872 }
15873 \cs_new_nopar:Npn \_tl_change_case_lower_az:Nnn
15874 { \_tl_change_case_lower_tr:Nnn }

```

After a dotless-I there may be a dot-above character. If there is then a dotted-i should be produced, otherwise output a dotless-i. When the combination is found both the dotless-I and the dot-above char have to be removed from the input, which is done by the `\use_i:nn` (it grabs `_tl_change_case_loop:wn` and the dot-above char and discards the latter).

```

15875 \cs_new:Npn \_tl_change_case_lower_tr:Nw #1#2 \q_recursion_stop

```

```

15876 {
15877   \bool_if:nTF
15878   {
15879     \token_if_cs_p:N #1
15880     || ! ( \int_compare_p:nNn { '#1 } = { "0307 } )
15881   }
15882   { \c__tl_dotless_i_tl }
15883   {
15884     i
15885     \use_i:nn
15886   }
15887 }

```

Upper casing is easier: just one exception with no context.

```

15888 \cs_new:Npn \__tl_change_case_upper_tr:Nnn #1#2#3
15889 {
15890   \int_compare:nNnTF { '#1 } = { "0069 }
15891   { \c__tl_dotted_I_tl }
15892   {#3}
15893 }
15894 \cs_new_nopar:Npn \__tl_change_case_upper_az:Nnn
15895 { \__tl_change_case_upper_tr:Nnn }

```

(End definition for __tl_change_case_lower_tr:Nnn and __tl_change_case_lower_az:Nnn.)

```

\__tl_change_case_lower_lt:Nnn
\__tl_change_case_lower_lt:Nw
\__tl_change_case_upper_lt:Nnn
\__tl_change_case_upper_lt:Nw

```

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. That means that there is some work to do when lower casing I and J. The first step is a simple match attempt: \c__tl_accents_lt_tl contains accented upper case letters which should gain a dot-above char in their lower case form. The second stage is to check for I, J and I-ogonek, and if the current char is a match to look for a following accent. As the current char is still needed in case-changed form, the standard code is inserted before hunting for an accent.

```

15896 \cs_new:Npn \__tl_change_case_lower_lt:Nnn #1#2#3
15897 {
15898   \exp_args:NNV \str_case:nnF #1 \c__tl_accents_lt_tl
15899   {
15900     #3
15901     \bool_if:nT
15902     {
15903       \int_compare_p:nNn { '#1 } = { "0049 }
15904       || \int_compare_p:nNn { '#1 } = { "004A }
15905       || \int_compare_p:nNn { '#1 } = { "012E }
15906     }
15907     {
15908       \tl_if_head_is_N_type:nT {#2}
15909       { \__tl_change_case_lower_lt:Nw #2 \q_recursion_stop }
15910     }
15911   }
15912 }

```

Grab the next char and see if it is one of the accents used in Lithuanian: if it is, add the dot-above char into the output.

```

15913 \cs_new:Npn \__tl_change_case_lower_lt:Nw #1#2 \q_recursion_stop
15914 {
15915   \bool_if:nT
15916   {
15917     ! ( \token_if_cs_p:N #1 )
15918     &&
15919     (
15920       \int_compare_p:nNn { '#1 } = { "0300 }
15921       || \int_compare_p:nNn { '#1 } = { "0301 }
15922       || \int_compare_p:nNn { '#1 } = { "0303 }
15923     )
15924   }
15925   { \c__tl_dot_above_tl }
15926 }

```

For upper casing, the chars themselves are always converted as normal. The test required here is for a dot-above char after an I, J or I-ogonek. If there is one, it's removed using `\use_i:nn` (which preserves the loop and discards the char).

```

15927 \cs_new:Npn \__tl_change_case_upper_lt:Nnn #1#2#3
15928 {
15929   #3
15930   \bool_if:nT
15931   {
15932     \tl_if_head_is_N_type_p:n {#2}
15933     &&
15934     (
15935       \int_compare_p:nNn { '#1 } = { "0069 }
15936       || \int_compare_p:nNn { '#1 } = { "006A }
15937       || \int_compare_p:nNn { '#1 } = { "012F }
15938     )
15939   }
15940   { \__tl_change_case_upper_lt:Nw #2 \q_recursion_stop }
15941 }
15942 \cs_new:Npn \__tl_change_case_upper_lt:Nw #1#2 \q_recursion_stop
15943 {
15944   \bool_if:nT
15945   {
15946     ! ( \token_if_cs_p:N #1 )
15947     &&
15948     \int_compare_p:nNn { '#1 } = { "0307 }
15949   }
15950   { \use_i:nn }
15951 }

```

(End definition for `__tl_change_case_lower_lt:Nnn`.)

```

\__tl_change_case_mixed_n1:Nnn
\__tl_change_case_mixed_n1:Nw

```

For Dutch, there is a single look-ahead test for ij when title casing. If the appropriate letters are found, produce IJ and gobble the j.

```

15952 \cs_new:Npn \__tl_change_case_mixed_n1:Nnn #1#2
15953 {
15954   \int_compare:nNnTF { '#1 } = { 'i }
15955   {
15956     I
15957     \tl_if_head_is_N_type:nT {#2}
15958     { \__tl_change_case_mixed_n1:Nw #2 \q_recursion_stop }
15959   }
15960 }
15961 \cs_new:Npn \__tl_change_case_mixed_n1:Nw #1#2 \q_recursion_stop
15962 {
15963   \bool_if:nT
15964   {
15965     ! ( \token_if_cs_p:N #1 )
15966     &&
15967     \int_compare_p:nNn { '#1 } = { 'j }
15968   }
15969   {
15970     J
15971     \use_i:nn
15972   }
15973 }
(End definition for \__tl_change_case_mixed_n1:Nnn.)

```

35.12 Additions to l3tokens

```

15974 <@@=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
15975 \group_begin:
15976 \char_set_catcode_active:N ^^@
15977 \cs_set:Npn \char_tmp:NN #1#2
15978 {
15979   \cs_new:Npn #1 ##1
15980   {
15981     \char_set_catcode_active:n { '##1 }
15982     \group_begin:
15983     \char_set_lccode:nn { '^@ } { '##1 }
15984     \tl_to_lowercase:n { \group_end: #2 ^^@ }
15985   }
15986 }
15987 \char_tmp:NN \char_set_active:Npn \cs_set:Npn
15988 \char_tmp:NN \char_set_active:Npx \cs_set:Npx
15989 \char_tmp:NN \char_gset_active:Npn \cs_gset:Npn
15990 \char_tmp:NN \char_gset_active:Npx \cs_gset:Npx
15991 \char_tmp:NN \char_set_active_eq:NN \cs_set_eq:NN
15992 \char_tmp:NN \char_gset_active_eq:NN \cs_gset_eq:NN
15993 \group_end:

```

(End definition for `\char_set_active:Npn` and `\char_set_active:Npx`. These functions are documented on page 204.)

15994 `<@@=peek>`

`\peek_N_type:TF`
`__peek_execute_branches_N_type:`
`__peek_N_type:w`
`__peek_N_type_aux:nnw`

All tokens are N-type tokens, except in four cases: begin-group tokens, end-group tokens, space tokens with character code 32, and outer tokens. Since `\l_peek_token` might be outer, we cannot use the convenient `\bool_if:nTF` function, and must resort to the old trick of using `\ifodd` to expand a set of tests. The `false` branch of this test is taken if the token is one of the first three kinds of non-N-type tokens (explicit or implicit), thus we call `__peek_false:w`. In the `true` branch, we must detect outer tokens, without impacting performance too much for non-outer tokens. The first filter is to search for `outer` in the `\meaning` of `\l_peek_token`. If that is absent, `\use_none_delimit_by_q_stop:w` cleans up, and we call `__peek_true:w`. Otherwise, the token can be a non-outer macro or a primitive mark whose parameter or replacement text contains `outer`, it can be the primitive `\outer`, or it can be an outer token. Macros and marks would have `ma` in the part before the first occurrence of `outer`; the meaning of `\outer` has nothing after `outer`, contrarily to outer macros; and that covers all cases, calling `__peek_true:w` or `__peek_false:w` as appropriate. Here, there is no `<search token>`, so we feed a dummy `\scan_stop:` to the `__peek_token_generic:NNTF` function.

```

15995 \group_begin:
15996   \char_set_catcode_other:N \0
15997   \char_set_catcode_other:N \U
15998   \char_set_catcode_other:N \T
15999   \char_set_catcode_other:N \E
16000   \char_set_catcode_other:N \R
16001   \tl_to_lowercase:n
16002   {
16003     \cs_new_protected_nopar:Npn \__peek_execute_branches_N_type:
16004     {
16005       \if_int_odd:w
16006         \if_catcode:w \exp_not:N \l_peek_token { \c_two \fi:
16007         \if_catcode:w \exp_not:N \l_peek_token } \c_two \fi:
16008         \if_meaning:w \l_peek_token \c_space_token \c_two \fi:
16009         \c_one
16010         \exp_after:wN \__peek_N_type:w
16011         \token_to_meaning:N \l_peek_token
16012         \q_mark \__peek_N_type_aux:nnw
16013         OUTER \q_mark \use_none_delimit_by_q_stop:w
16014         \q_stop
16015         \exp_after:wN \__peek_true:w
16016       \else:
16017         \exp_after:wN \__peek_false:w
16018       \fi:
16019     }
16020     \cs_new_protected:Npn \__peek_N_type:w #1 OUTER #2 \q_mark #3
16021     { #3 {#1} {#2} }
16022   }
16023 \group_end:

```



```

16024 \cs_new_protected:Npn \__peek_N_type_aux:nnw #1 #2 #3 \fi:
16025 {
16026   \fi:
16027   \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
16028     { \__peek_true:w }
16029     { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
16030 }
16031 \cs_new_protected_nopar:Npn \peek_N_type:TF
16032 { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }
16033 \cs_new_protected_nopar:Npn \peek_N_type:T
16034 { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
16035 \cs_new_protected_nopar:Npn \peek_N_type:F
16036 { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }

```

(End definition for \peek_N_type:TF. This function is documented on page 204.)

35.13 Deprecated candidates

\fp_set_from_dim:Nn Deprecated 2014-07-17.

```

\fp_set_from_dim:cn
\fp_gset_from_dim:Nn
\fp_gset_from_dim:cn
16037 \cs_new_protected:Npn \fp_set_from_dim:Nn #1#2
16038 { \fp_set:Nn #1 { \dim_to_fp:n {#2} } }
16039 \cs_new_protected:Npn \fp_gset_from_dim:Nn #1#2
16040 { \fp_gset:Nn #1 { \dim_to_fp:n {#2} } }
16041 \cs_generate_variant:Nn \fp_set_from_dim:Nn { c }
16042 \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 ??.)

```

16043 </initex | package>

```

36 l3drivers Implementation

```

16044 <*initex | package>
16045 <@@=driver>
16046 <*package>
16047 \ProvidesExplFile
16048 <*dvi pdfmx>
16049 {l3dvi pdfmx.def}{\ExplFileDate}{\ExplFileVersion}
16050 {L3 Experimental driver: dvi pdfmx}
16051 </dvi pdfmx>
16052 <*dvips>
16053 {l3dvips.def}{\ExplFileDate}{\ExplFileVersion}
16054 {L3 Experimental driver: dvips}
16055 </dvips>
16056 <*pdfmode>
16057 {l3pdfmode.def}{\ExplFileDate}{\ExplFileVersion}
16058 {L3 Experimental driver: PDF mode}
16059 </pdfmode>
16060 <*xdvi pdfmx>
16061 {l3xdvi pdfmx.def}{\ExplFileDate}{\ExplFileVersion}

```

```

16062 {L3 Experimental driver: xdvipdfmx}
16063 </xdvipdfmx>
16064 </package>

```

36.1 Settings for direct PDF output

If the driver loaded is `pdfmode` then direct PDF output is required. (This may of course alter: it might be that the driver is picked based on the value of `\pdfTEX_pdfoutput:D`.)

```

16065 <*initex>
16066 <*pdfmode>
16067 \pdfTEX_pdfoutput:D = 1 \scan_stop:
16068 </pdfmode>
16069 </initex>

```

Set up the driver for direct PDF output to set the PDF origin equal to T_EX's standard origin. The other settings make use of PDF 1.5, which is standard in T_EX Live 2011 and should be a reasonable baseline for the future.

```

16070 <*initex>
16071 <*pdfmode>
16072 \pdfTEX_pdfhorigin:D          = 1 true in \scan_stop:
16073 \pdfTEX_pdfvorigin:D          = 1 true in \scan_stop:
16074 \pdfTEX_pdfdecimaldigits:D    = 3          \scan_stop:
16075 \pdfTEX_pdfpkresolution:D     = 600        \scan_stop:
16076 \pdfTEX_pdfminorversion:D     = 5          \scan_stop:
16077 \pdfTEX_pdfcompresslevel:D    = 9          \scan_stop:
16078 \pdfTEX_pdfobjcompresslevel:D = 2          \scan_stop:
16079 </pdfmode>
16080 </initex>

```

36.2 Driver utility functions

`_driver_state_save:` All of the drivers have a stack for saving the graphic state. These have slightly different interfaces. For both `dvips` and `(x)dvipdfmx` this is done using an appropriate special. Note that here and later, the `dvipdfmx` documentation does not cover the `literal` key word but that this appears to behave in the same way as pdfT_EX's `\pdfliteral` (making life easier all-round).

```

16081 <*!pdfmode>
16082 \cs_new_protected_nopar:Npn \_driver_state_save:
16083 <*dvips>
16084 { \tex_special:D { ps:gsave } }
16085 </dvips>
16086 <*dvipdfmx | xdvipdfmx>
16087 { \tex_special:D { pdf:literal-q } }
16088 </dvipdfmx | xdvipdfmx>
16089 \cs_new_protected_nopar:Npn \_driver_state_restore:
16090 <*dvips>
16091 { \tex_special:D { ps:grestore } }
16092 </dvips>
16093 <*dvipdfmx | xdvipdfmx>

```

```

16094 { \tex_special:D { pdf:literal-Q } }
16095 </dvipdfmx|xdvipdfmx>
16096 <!/pdfmode>

```

For direct PDF output there is also a need to worry about the version of pdfT_EX in use: the `\pdfsave` primitive was only introduced in version 1.40.0.

```

16097 <*pdfmode>
16098 \cs_if_exist:NTF \pdfTeX_pdfsave:D
16099 {
16100   \cs_new_eq:NN \__driver_state_save: \pdfTeX_pdfsave:D
16101   \cs_new_eq:NN \__driver_state_restore: \pdfTeX_pdfrestore:D
16102 }
16103 {
16104   \cs_new_protected_nopar:Npn \__driver_state_save:
16105     { \pdfTeX_pdfliteral:D { q } }
16106   \cs_new_protected_nopar:Npn \__driver_state_restore:
16107     { \pdfTeX_pdfliteral:D { Q } }
16108 }
16109 </pdfmode>

```

(End definition for __driver_state_save: and __driver_state_restore:. These functions are documented on page ??.)

`__driver_literal:n` The driver code needs to pass on a lot of “raw” information to the underlying binary. The exact command is driver-dependent but the concept is general enough to use a single function. However, it is important to remember this is a convenient shortcut: the arguments will be driver-specific. Note that these functions set the transformation matrix to the current position: contrast with `\@literal_direct:n`.

```

16110 \cs_new_protected:Npn \__driver_literal:n #1
16111 <*dvipdfmx|xdvipdfmx>
16112 { \tex_special:D { pdf:literal~ #1 } }
16113 </dvipdfmx|xdvipdfmx>

```

In the case of dvips there is no build-in saving of the current position, and so some additional PostScript is required to set up the transformation matrix and also to restore it afterwards. Notice the use of the stack to save the current position “up front” and to move back to it at the end of the process.

```

16114 <*dvips>
16115 {
16116   \tex_special:D
16117   {
16118     ps:
16119     currentpoint~
16120     currentpoint~translate~
16121     #1 ~
16122     neg~exch~neg~exch~translate
16123   }
16124 }
16125 </dvips>
16126 <*pdfmode>
16127 { \pdfTeX_pdfliteral:D {#1} }

```

```

16128 </pdfmode>
(End definition for \_driver_literal:n.)

```

`_driver_literal_direct:n` Even “lower level” than `\@_literal:n`, these commands do not set the transformation matrix but simply dump the driver code directly into the output. In the (x)dvipdfmx case this two-part keyword is documented (*cf.* `literal` alone).

```

16129 \cs_new_protected:Npn \_driver_literal_direct:n #1
16130 { *dvipdfmx | xdvipdfmx }
16131 { \tex_special:D { pdf:literal-direct~ #1 } }
16132 </dvipdfmx | xdvipdfmx>
16133 { *dvips }
16134 { \tex_special:D { ps:: #1 } }
16135 </dvips>
16136 { *pdfmode }
16137 { \pdfTeX_pdfliteral:D direct {#1} }
16138 </pdfmode>
(End definition for \_driver_literal_direct:n.)

```

`_driver_absolute_lengths:n` The `dvips` driver scales all absolute dimensions based on the output resolution selected and any \TeX magnification. Thus for any operation involving absolute lengths there is a correction to make. This is based on `normalscale` from `special.pro`.

```

16139 { *dvips }
16140 \cs_new:Npn \_driver_absolute_lengths:n #1
16141 {
16142     /savedmatrix~matrix~currentmatrix~def~
16143     Resolution~72~div~VResolution~72~div~scale~
16144     DVImag~dup~scale~
16145     #1 ~
16146     savedmatrix~setmatrix
16147 }
16148 </dvips>
(End definition for \_driver_absolute_lengths:n.)

```

`_driver_matrix:n` Here the appropriate function is set up to insert an affine matrix into the PDF. With a new enough pdf \TeX (version 1.40.0 or later) there is a primitive for this, which only needs the rotation/scaling/skew part. With an older pdf \TeX or with (x)dvipdfmx the matrix also has to include a translation part: that is always zero and so is built in here.

```

16149 { *pdfmode }
16150 \cs_if_exist:NTF \pdfTeX_pdfsetmatrix:D
16151 {
16152     \cs_new_protected:Npn \_driver_matrix:n #1
16153     { \pdfTeX_pdfsetmatrix:D {#1} }
16154 }
16155 {
16156     \cs_new_protected:Npn \_driver_matrix:n #1
16157     { \_driver_literal:n { #1 \c_space_tl 0~0~cm } }
16158 }
16159 </pdfmode>

```

```

16160 <*dvipdfmx | xdvipdfmx>
16161 \cs_new_protected:Npn \__driver_matrix:n #1
16162 { \__driver_literal:n { #1 \c_space_tl 0~0~cm } }
16163 </dvipdfmx | xdvipdfmx>
(End definition for \__driver_matrix:n.)

```

36.3 Box clipping

`__driver_box_use_clip:N`

The overall logic to clipping a box is the same in all cases. The general method is to save the current location, define a clipping path equivalent to the bounding box, then insert the content at the current position and in a zero width box. The “real” width is then made up using a horizontal skip before tidying up. There are other approaches that can be taken (for example using XForm objects), but the logic here shares as much code as possible and uses the same conversions (and so same rounding errors) in all three cases.

```

16164 \cs_new_protected:Npn \__driver_box_use_clip:N #1
16165 {
16166   \__driver_state_save:
16167   <*dvips>
16168   \__driver_literal:n
16169   {
16170     \__driver_absolute_lengths:n
16171     {
16172       0~
16173       \dim_to_decimal_in_bp:n { \box_dp:N #1 } ~
16174       \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
16175       \dim_to_decimal_in_bp:n { - \box_ht:N #1 - \box_dp:N #1 } ~
16176       rectclip
16177     }
16178   }
16179   </dvips>
16180   <*dvipdfmx | pdfmode | xdvipdfmx>
16181   \__driver_literal:n
16182   {
16183     0~
16184     \dim_to_decimal_in_bp:n { - \box_dp:N #1 } ~
16185     \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
16186     \dim_to_decimal_in_bp:n { \box_ht:N #1 + \box_dp:N #1 } ~
16187     re~W~n
16188   }
16189   </dvipdfmx | pdfmode | xdvipdfmx>

```

Insert the material in a box of no width, restore the graphic state and then insert the necessary width.

```

16190   \hbox_overlap_right:n { \box_use:N #1 }
16191   \__driver_state_restore:
16192   \skip_horizontal:n { \box_wd:N #1 }
16193 }

```

(End definition for __driver_box_use_clip:N. This function is documented on page 205.)

36.4 Box rotation and scaling

`__driver_box_rotate_begin:` The driver for `dvips` works with a simple rotation angle. In PDF mode, an affine matrix is used instead. The transformation for `(x)dvipdfmx` can be done either way: the affine approach is chosen here as where possible we pick the PDF-style route.

In both cases, some rounding code is included to limit the floating point values to five decimal places. There is no point using any more as \TeX 's dimensions are of that precision, and the extra figures will simply bloat the PDF and make values harder to trace. In the case where the sine and cosine are used, we store the rounded values to avoid rounding twice. There are also a couple of comparisons to ensure that `-0` is not written to the output, as this avoids any issues with problematic display programs. Note that numbers are compared to 0 after rounding.

```

16194 \cs_new_protected_nopar:Npn __driver_box_rotate_begin:
16195 {
16196   __driver_state_save:
16197   <*dvipdfmx|pdfmode|xdvipdfmx>
16198   \box_set_wd:Nn \l__box_internal_box \c_zero_dim
16199   \fp_set:Nn \l__box_cos_fp { round ( \l__box_cos_fp , 5 ) }
16200   \fp_compare:nNnT \l__box_cos_fp = \c_zero_fp
16201     { \fp_zero:N \l__box_cos_fp }
16202   \fp_set:Nn \l__box_sin_fp { round ( \l__box_sin_fp , 5 ) }
16203   __driver_matrix:n
16204   {
16205     \fp_use:N \l__box_cos_fp \c_space_tl
16206     \fp_compare:nNnTF \l__box_sin_fp = \c_zero_fp
16207       { 0~0 }
16208       {
16209         \fp_use:N \l__box_sin_fp
16210         \c_space_tl
16211         \fp_eval:n { -\l__box_sin_fp }
16212       }
16213     \c_space_tl
16214     \fp_use:N \l__box_cos_fp
16215   }
16216   </dvipdfmx|pdfmode|xdvipdfmx>
16217   <*dvips>
16218   \fp_set:Nn \l__box_angle_fp { round ( \l__box_angle_fp , 5 ) }
16219   __driver_literal:n
16220   {
16221     \fp_compare:nNnTF \l__box_angle_fp = \c_zero_fp
16222       { 0 }
16223       { \fp_eval:n { - \l__box_angle_fp } }
16224     \c_space_tl
16225     rotate
16226   }
16227   </dvips>
16228 }

```

The end of a rotation means tidying up the output grouping.

```

16229 \cs_new_eq:NN \__driver_box_rotate_end: \__driver_state_restore:
(End definition for \__driver_box_rotate_begin: and \__driver_box_rotate_end:. These functions
are documented on page 206.)

```

__driver_box_scale_begin: Scaling is not dissimilar to rotation, but the calculations are somewhat less complex.
__driver_box_scale_end:

```

16230 \cs_new_protected_nopar:Npn \__driver_box_scale_begin:
16231 {
16232   \__driver_state_save:
16233   \fp_set:Nn \l__box_scale_x_fp { round ( \l__box_scale_x_fp , 5 ) }
16234   \fp_set:Nn \l__box_scale_y_fp { round ( \l__box_scale_y_fp , 5 ) }
16235   \*dvips)
16236   \__driver_literal:n
16237   {
16238     \fp_use:N \l__box_scale_x_fp \c_space_tl
16239     \fp_use:N \l__box_scale_y_fp \c_space_tl
16240     scale
16241   }
16242   \*dvips)
16243   \*dvipdfmx | pdfmode | xdvipdfmx)
16244   \__driver_matrix:n
16245   {
16246     \fp_use:N \l__box_scale_x_fp \c_space_tl
16247     0~0~
16248     \fp_use:N \l__box_scale_y_fp
16249   }
16250   \*dvipdfmx | pdfmode | xdvipdfmx)
16251   }
16252 \cs_new_eq:NN \__driver_box_scale_end: \__driver_state_restore:
(End definition for \__driver_box_scale_begin: and \__driver_box_scale_end:. These functions are
documented on page 206.)

```

36.5 Color support

\l__driver_current_color_tl The current color is needed by all of the engines, but the way this is stored varies.

```

16253 \tl_new:N \l__driver_current_color_tl
16254 \*dvipdfmx | xdvipdfmx)
16255 \tl_set:Nn \l__driver_current_color_tl { gray~0 }
16256 \*dvips)
16257 \tl_set:Nn \l__driver_current_color_tl { Black }
16258 \*pdfmode)
16259 \tl_set:Nn \l__driver_current_color_tl { 0~g~0~G }
16260 \*pdfmode)
(End definition for \l__driver_current_color_tl. This variable is documented on page ??.)

```

\l__driver_color_stack_int pdfTeX (version 1.40.0 or later) and LuaTeX have multiple stacks available, and the color stack therefore needs a number when in PDF mode.

```

16263 \*pdfmode)

```

```

16264 \int_new:N \l__driver_color_stack_int
16265 </pdfmode>
(End definition for \l__driver_color_stack_int. This variable is documented on page ??.)

```

`__driver_color_ensure_current:` Setting the current color depends on the nature of the color stack available. In all cases
`__driver_color_reset:` there is a need to reset the color after the current group.

```

16266 <*dvipdfmx | dvips | xdvipdfmx>
16267 \cs_new_protected_nopar:Npn \__driver_color_ensure_current:
16268 {
16269   \tex_special:D { color~push~\l__driver_current_color_tl }
16270   \group_insert_after:N \__driver_color_reset:
16271 }
16272 \cs_new_protected_nopar:Npn \__driver_color_reset:
16273 { \tex_special:D { color~pop } }
16274 </dvipdfmx | dvips | xdvipdfmx>

```

Once again there is a version switch for pdfTeX, as the `\pdfcolorstack` primitive was introduced in version 1.40.0.

```

16275 <*pdfmode>
16276 \cs_if_exist:NTF \pdfTeX_pdfcolorstack:D
16277 {
16278   \cs_new_protected_nopar:Npn \__driver_color_ensure_current:
16279   {
16280     \pdfTeX_pdfcolorstack:D \l__driver_color_stack_int push
16281     { \l__driver_current_color_tl }
16282     \group_insert_after:N \__driver_color_reset:
16283   }
16284   \cs_new_protected_nopar:Npn \__driver_color_reset:
16285   { \pdfTeX_pdfcolorstack:D \l__driver_color_stack_int pop }
16286 }
16287 {
16288   \cs_new_protected_nopar:Npn \__driver_color_ensure_current:
16289   {
16290     \__driver_literal:n { \l__driver_current_color_tl }
16291     \group_insert_after:N \__driver_color_reset:
16292   }
16293   \cs_new_protected_nopar:Npn \__driver_color_reset:
16294   { \__driver_literal:n { \l__driver_current_color_tl } }
16295 }
16296 </pdfmode>
(End definition for \__driver_color_ensure_current:. This function is documented on page 206.)
16297 </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	
\!	7752
\"	2475, 2490
\#	6, 2490, 9570
\\$	2476, 2490
\%	2490, 9572
.	186
\	2477, 2490, 7752, 7753, 11258, 11332
\)	11211
**	187
*	2455, 2457, 2461, 2468, 9537, 9538, 11160
\+	11256, 11262
\,	8387, 8389, 11227
\-	212, 10391, 11261
\.bool_gset:N	8698
\.bool_gset:c	8698
\.bool_gset_inverse:N	8706
\.bool_gset_inverse:c	8706
\.bool_set:N	8698
\.bool_set:c	8698
\.bool_set_inverse:N	8706
\.bool_set_inverse:c	8706
\.choice:	8714
\.choice_code:n	9097
\.choice_code:x	9097
\.choices:Vn	8716
\.choices:nn	8716
\.choices:on	8716
\.choices:xn	8716
\.clist_gset:N	8726
\.clist_gset:c	8726
\.clist_set:N	8726
\.clist_set:c	8726
\.code:n	8724
\.default:V	8734
\.default:n	8734
\.default:o	8734
\.default:x	8734
\.dim_gset:N	8742
\.dim_gset:c	8742
\.dim_set:N	8742
\.dim_set:c	8742
\.fp_gset:N	8750
\.fp_gset:c	8750
\.fp_set:N	8750
\.fp_set:c	8750
\.generate_choices:n	9097
\.groups:n	8758
\.initial:V	8760
\.initial:n	8760
\.initial:o	8760
\.initial:x	8760
\.int_gset:N	8768
\.int_gset:c	8768
\.int_set:N	8768
\.int_set:c	8768
\.meta:n	8776
\.meta:nn	8778
\.multichoice:	8780
\.multichoices:Vn	8780
\.multichoices:nn	8780
\.multichoices:on	8780
\.multichoices:xn	8780
\.skip_gset:N	8790
\.skip_gset:c	8790
\.skip_set:N	8790
\.skip_set:c	8790
\.tl_gset:N	8798
\.tl_gset:c	8798
\.tl_gset_x:N	8798
\.tl_gset_x:c	8798
\.tl_set:N	8798
\.tl_set:c	8798
\.tl_set_x:N	8798
\.tl_set_x:c	8798
\.value_forbidden:	8814
\.value_required:	8814
\/	211, 11260
?:	186
\:	29, 999, 2567, 2766
\:::	33, 1462, 1463, 1464, 1464, 1465, 1466, 1467, 1468, 1470, 1472, 1479, 1484, 1490, 1609, 1610, 1611, 1612, 1613, 1614, 1615, 1616, 1617, 1618, 1619, 1620, 1621, 1622, 1623, 1624, 1625, 1626, 1627, 1628, 1629, 1630, 1631, 1632, 1633, 1634,

1635, 1636, 1638, 1643, 1645, 1650, 1655, 1692, 1693, 1694, 1695, 1696	_fp_decimate_auxi:Nnnnn 9895
\\:N 33, 1466, 1466,	_fp_decimate_auxii:Nnnnn 9895
1619, 1625, 1626, 1630, 1631, 1695	_fp_decimate_auxiii:Nnnnn 9895
\\:V 33, 1484, 1484, 1615	_fp_decimate_auxiv:Nnnnn 9895
\\:V_unbraced 1637, 1645	_fp_decimate_auxix:Nnnnn 9895
\\:c 33, 1468, 1468,	_fp_decimate_auxv:Nnnnn 9895
1610, 1618, 1620, 1627, 1634, 1635	_fp_decimate_auxvi:Nnnnn 9895
\\:f 33, 1472,	_fp_decimate_auxvii:Nnnnn 9895
1472, 1611, 1612, 1613, 1617, 1694	_fp_decimate_auxviii:Nnnnn ... 9895
\\:f_unbraced 1637, 1638	_fp_decimate_auxx:Nnnnn 9895
\\:n 33, 1465, 1465, 1610, 1613,	_fp_decimate_auxxi:Nnnnn 9895
1614, 1615, 1621, 1625, 1627, 1628,	_fp_decimate_auxxii:Nnnnn 9895
1630, 1632, 1633, 1635, 1692, 1695	_fp_decimate_auxxiii:Nnnnn ... 9895
\\:o .. 33, 1470, 1470, 1611, 1614, 1616,	_fp_decimate_auxxiv:Nnnnn 9895
1617, 1618, 1622, 1623, 1625, 1626,	_fp_decimate_auxxv:Nnnnn 9895
1628, 1629, 1631, 1633, 1636, 1693	_fp_decimate_auxxvi:Nnnnn 9895
\\:o_unbraced .. 1637, 1643, 1692, 1693, 1694, 1695	\\{ 4, 2490, 7750, 8199, 8365, 8369, 8370, 9569
\\:p 33, 1467, 1467	\\} 5, 2490, 7751, 8199, 8365, 8369, 8370, 9571
\\:v 33, 1484, 1490	\\^ 7, 10, 82, 2478, 2490,
\\:v_unbraced 1637, 1650	8276, 9544, 11259, 13489, 15976, 15983
\\:x 33, 1479, 1479, 1609, 1619,	\\ 28, 2479, 2490, 10391
1620, 1621, 1622, 1623, 1624, 1630,	_bool__0:w 2078
1631, 1632, 1633, 1634, 1635, 1636	_bool__1:w 2076
\\:x_unbraced 1637, 1655, 1696	_bool_(Nw 2056
\\:N 11226	_bool_)_0:w 2072
\\; 2567, 2765, 2766	_bool_)_1:w 2072
\\= 8386, 8388	_bool_:Nw 2054
\\? 11359	_bool_S_0:w 2072
\\@ 999, 1000	_bool_S_1:w 2072
\\@@_if_eq_x_return:nn 106	_bool_choose:NNN 2058, 2062, 2063, 2063
\\@@end 640, 641	_bool_eval_skip_to_end_auxi:Nw ...
\\@@hyph 644 2078, 2078, 2079, 2081, 2095
\\@@input 645	_bool_eval_skip_to_end_auxii:Nw ..
\\@@italiccorr 646 2078, 2083, 2087
\\@@underline 647	_bool_eval_skip_to_end_auxiii:Nw .
\\@addtofilelist 9292 2078, 2091, 2093
\\@currname 9160	_bool_get_next:NN 2042,
\\@filelist 9291, 9318, 9320, 9335	2044, 2044, 2055, 2059, 2076, 2077
\\@tempa 122, 124	_bool_if_left_parentheses:wwwn ...
\\ 1408, 2490, 7594, 2024, 2027, 2034, 2035
7595, 7596, 7597, 7687, 7705, 7707,	_bool_if_or:wwwn 2024, 2030, 2038, 2039
7712, 7713, 7727, 7728, 7731, 7732,	_bool_if_parse:NNNww . 2031, 2040, 2040
7803, 7892, 7900, 8109, 8116, 8128,	_bool_if_right_parentheses:wwwn ..
8129, 8144, 8145, 8198, 8199, 8200, 2024, 2029, 2036, 2037
8201, 8202, 8365, 8369, 8374, 9042,	_bool_p:Nw 2061
9055, 9056, 9062, 9075, 9088, 9094,	_box_resize:Nn 14949, 14962,
9575, 9715, 9722, 10152, 10155,	14974, 14995, 15012, 15026, 15044
10156, 10157, 10164, 10167, 10168	_box_resize_common:N 14981, 15066, 15071, 15071

__box_resize_set_corners:N
 [14949](#), [14954](#),
 [14967](#), [14988](#), [15005](#), [15022](#), [15036](#)
 __box_rotate:N [14835](#), [14843](#), [14847](#)
 __box_rotate_quadrant_four:
 [14835](#), [14862](#), [14936](#)
 __box_rotate_quadrant_one:
 [14835](#), [14856](#), [14903](#)
 __box_rotate_quadrant_three:
 [14835](#), [14861](#), [14925](#)
 __box_rotate_quadrant_two:
 [14835](#), [14857](#), [14914](#)
 __box_rotate_x:nnN
 [14835](#), [14881](#), [14909](#), [14911](#), [14920](#),
 [14922](#), [14931](#), [14933](#), [14942](#), [14944](#)
 __box_rotate_y:nnN
 [14835](#), [14892](#), [14905](#), [14907](#), [14916](#),
 [14918](#), [14927](#), [14929](#), [14938](#), [14940](#)
 __box_show:NNnn . [6664](#), [6674](#), [6681](#), [6681](#)
 __chk_if_exist_cs:N
 [24](#), [1150](#), [1150](#), [1159](#), [1718](#)
 __chk_if_exist_cs:c
 [969](#), [974](#), [979](#), [984](#), [1150](#), [1158](#)
 __chk_if_exist_var:N
 [24](#), [1138](#), [1140](#), [1942](#),
 [1947](#), [1952](#), [1957](#), [1962](#), [1967](#), [1972](#),
 [1977](#), [4416](#), [4438](#), [4439](#), [4444](#), [4445](#),
 [4452](#), [4453](#), [4454](#), [4459](#), [4460](#), [4461](#)
 __chk_if_free_cs:N
 . [24](#), [1115](#), [1115](#), [1125](#), [1137](#), [1164](#),
 [1212](#), [3109](#), [3124](#), [3870](#), [4117](#), [4207](#),
 [4274](#), [4280](#), [4285](#), [5254](#), [6303](#), [6570](#)
 __chk_if_free_cs:c [1115](#), [1136](#)
 __chk_if_free_msg:nn
 [7638](#), [7638](#), [7648](#), [7661](#)
 __clist_concat:NNNN [5819](#), [5820](#), [5822](#), [5823](#)
 __clist_count:n [6163](#), [6168](#), [6181](#)
 __clist_count:w . [6163](#), [6177](#), [6182](#), [6186](#)
 __clist_get:wN .. [5895](#), [5900](#), [5903](#), [5937](#)
 __clist_if_empty_n:w
 [6047](#), [6049](#), [6054](#), [6057](#)
 __clist_if_empty_n:wNw [6047](#), [6058](#), [6060](#)
 __clist_if_in_return:nn
 [6061](#), [6063](#), [6068](#), [6070](#)
 __clist_item:nnNn [6218](#), [6220](#), [6226](#), [6250](#)
 __clist_item_N_loop:nw
 [6218](#), [6223](#), [6241](#), [6245](#)
 __clist_item_n:nw [6248](#), [6253](#), [6256](#)
 __clist_item_n_end:n .. [6248](#), [6264](#), [6272](#)
 __clist_item_n_loop:nw
 [6248](#), [6257](#), [6258](#), [6261](#), [6266](#)
 __clist_item_n_strip:w [6248](#), [6275](#), [6277](#)
 __clist_map_function:Nw
 [6086](#), [6090](#), [6095](#), [6099](#), [6122](#)
 __clist_map_function_n:Nn
 [6102](#), [6104](#), [6108](#), [6112](#)
 __clist_map_unbrace:Nw [6102](#), [6111](#), [6115](#)
 __clist_map_variable:Nnw
 [6135](#), [6140](#), [6151](#), [6156](#)
 __clist_pop:NNN . [5906](#), [5907](#), [5909](#), [5910](#)
 __clist_pop:wN [5906](#), [5923](#), [5929](#)
 __clist_pop:wwNNN [5906](#), [5915](#), [5918](#), [5953](#)
 __clist_pop_TF:NNN [5932](#), [5945](#), [5947](#), [5948](#)
 __clist_put_left:NNNn
 [5869](#), [5870](#), [5872](#), [5873](#)
 __clist_put_right:NNNn
 [5882](#), [5883](#), [5885](#), [5886](#)
 __clist_remove_all: [5996](#), [6006](#), [6010](#), [6022](#)
 __clist_remove_all:NNn
 [5996](#), [5997](#), [5999](#), [6000](#)
 __clist_remove_all:w .. [5996](#), [6023](#), [6024](#)
 __clist_remove_duplicates:NN
 [5980](#), [5981](#), [5983](#), [5984](#)
 __clist_reverse:wwNww
 [6033](#), [6035](#), [6036](#), [6040](#)
 __clist_reverse_end:ww [6033](#), [6037](#), [6043](#)
 __clist_set_from_seq:NNNN
 [5791](#), [5792](#), [5794](#), [5795](#)
 __clist_set_from_seq:w [5791](#), [5810](#), [5814](#)
 __clist_tmp:w
 .. [5769](#), [5769](#), [6002](#), [6023](#), [6072](#), [6074](#)
 __clist_trim_spaces:n
 [5773](#), [5842](#), [5842](#), [5864](#), [5866](#)
 __clist_trim_spaces:nn
 [5842](#), [5845](#), [5849](#), [5855](#), [5860](#)
 __clist_trim_spaces_generic:nn
 [5836](#), [5839](#), [5841](#)
 __clist_trim_spaces_generic:nw [5836](#),
 [5836](#), [5844](#), [5854](#), [5859](#), [6104](#), [6112](#)
 __clist_use:nwnn [6188](#), [6202](#), [6213](#)
 __clist_use:nwwwnwnn
 [6188](#), [6199](#), [6201](#), [6210](#)
 __clist_use:wnn . [6188](#), [6195](#), [6196](#), [6209](#)
 __clist_wrap_item:n ... [5791](#), [5803](#), [5807](#)
 __coffin_align:NnnNnnnnN
 .. [7185](#), [7222](#), [7240](#), [7248](#), [7248](#), [7339](#)
 __coffin_calculate_intersection:Nnn
 [7077](#), [7077](#), [7250](#), [7253](#), [7539](#)

_coffin_calculate_intersection:nnnnnnnn _coffin_scale_corner:Nnnn
 [7077](#), [7083](#), [7092](#), [7485](#) [15364](#), [15397](#), [15397](#)
 _coffin_calculate_intersection_aux:nnnnnN_coffin_scale_pole:Nnnnnn
 [7077](#), [15366](#), [15397](#), [15403](#)
[7104](#), [7119](#), [7128](#), [7135](#), [7161](#), [7170](#) _coffin_scale_vector:nnNN
 [15390](#), [15390](#), [15399](#), [15405](#)
 _coffin_display_attach:Nnnnn
 [7451](#), [7490](#), [7512](#), [7531](#), [7537](#) _coffin_set_bounding:N
 [15214](#), [15241](#), [15241](#)
 _coffin_display_handles_aux:nnnn
 [7451](#), [7518](#), [7523](#), [7529](#) _coffin_set_eq_structure:NN
 [6958](#), [6991](#), [6991](#)
 _coffin_display_handles_aux:nnnnnn
 [7451](#), [7476](#), [7480](#) _coffin_set_pole:Nnn [7005](#), [7027](#), [7031](#)
 _coffin_find_bounding_shift:
 [15218](#), [15316](#), [15316](#) _coffin_set_pole:Nnx
 [6891](#), [6938](#), [7005](#), [7009](#), [7020](#), [7279](#),
 [7316](#), [7320](#), [7328](#), [7332](#), [15270](#), [15406](#)
 _coffin_find_bounding_shift_aux:nn
 [15316](#), [15320](#), [15322](#) _coffin_shift_corner:Nnnn
 [15236](#), [15327](#), [15327](#)
 _coffin_find_corner_maxima:N
 [15217](#), [15296](#), [15296](#) _coffin_shift_pole:Nnnnnn
 [15238](#), [15327](#), [15335](#)
 _coffin_find_corner_maxima_aux:nn
 [15296](#), [15303](#), [15305](#) _coffin_update_B:nnnnnnnnN
 [7301](#), [7309](#), [7324](#)
 _coffin_get_pole:NnN
 [6974](#), [6974](#), [7079](#), [7080](#), [7303](#),
 [7304](#), [7307](#), [7308](#), [7465](#), [7466](#), [7469](#) _coffin_update_T:nnnnnnnnN
 [7301](#), [7305](#), [7312](#)
 _coffin_gset_eq_structure:NN
 [6991](#), [6998](#) _coffin_update_corners:N
 [6868](#), [6889](#), [6913](#), [6936](#), [7032](#), [7032](#)
 _coffin_if_exist:NT
 [6826](#), [6826](#), [6837](#), [6857](#), [6874](#),
 [6904](#), [6921](#), [6955](#), [7007](#), [7018](#), [7563](#) _coffin_update_poles:N
 [6867](#), [6888](#),
 [6912](#), [6935](#), [7043](#), [7043](#), [7200](#), [7231](#)
 _coffin_mark_handle_aux:nnnnNnn
 [7396](#), [7434](#), [7439](#), [7443](#) _coffin_update_vertical_poles:NNN
 [7216](#), [7235](#), [7301](#), [7301](#)
 _coffin_offset_corner:Nnnnn
 [7286](#), [7289](#), [7291](#) _coffin_x_shift_corner:Nnnn
 [15370](#), [15412](#), [15412](#)
 _coffin_offset_corners:Nnn
 [7205](#), [7206](#), [7212](#), [7213](#), [7286](#), [7286](#) _coffin_x_shift_pole:Nnnnnn
 [15372](#), [15412](#), [15419](#)
 _coffin_offset_pole:Nnnnnnn
 [7267](#), [7270](#), [7272](#) _cs_count_signature:N
 [25](#), [1259](#), [1259](#), [1270](#)
 _coffin_offset_poles:Nnn [7203](#), [7204](#), [1259](#), [1269](#)
 [7209](#), [7210](#), [7232](#), [7233](#), [7267](#), [7267](#) _cs_count_signature:nnN [1259](#), [1260](#), [1261](#)
 _coffin_reset_structure:N
 [6840](#), [6866](#), [6887](#), [1288](#), [1292](#)
 [6911](#), [6934](#), [6984](#), [6984](#), [7197](#), [7227](#) _cs_generate_from_signature:nnNNn
 [1294](#), [1297](#)
 _coffin_resize_common:Nnn
 [15358](#), [15361](#), [15361](#), [15386](#) _cs_generate_internal_variant:n
 [1861](#), [1874](#), [1880](#)
 _coffin_rotate_bounding:nnn
 [15216](#), [15253](#), [15253](#) _cs_generate_internal_variant:wwnNwnn
 [1882](#), [1893](#)
 _coffin_rotate_corner:Nnnn
 [15211](#), [15253](#), [15259](#) _cs_generate_internal_variant:www
 [1874](#)
 _coffin_rotate_pole:Nnnnnn
 [15213](#), [15265](#), [15265](#) _cs_generate_internal_variant_loop:n
 [1874](#), [1891](#), [1900](#), [1903](#)
 _coffin_rotate_vector:nnNN [15255](#),
 [15261](#), [15267](#), [15268](#), [15277](#), [15277](#) _cs_generate_variant:N [1719](#), [1726](#), [1734](#)

__cs_generate_variant:Nnnw 1362, 1363, 1364, 1737, 1752, 1862,
 1762, 1764, 1764, 1782 1884, 4171, 4181, 4407, 4420, 4422
 __cs_generate_variant:nnNN __cs_to_str:N 989, 993, 995, 996
 1722, 1755, 1755 __cs_to_str:w 989, 992, 996
 __cs_generate_variant:ww 1726, 1739, 1747 __dim_abs:N 3914, 3916, 3919
 __cs_generate_variant:wwNN __dim_case:nnTF
 1771, 1854, 1854 .. 3990, 3993, 3998, 4003, 4008, 4010
 __cs_generate_variant:wwNw __dim_case:nw ... 3990, 4011, 4012, 4016
 1726, 1748, 1749 __dim_case_end:nw 3990, 4015, 4018
 __cs_generate_variant_loop:nNwN ... __dim_compare:w 3954, 3956, 3959
 1772, 1784, 1784, 1798 __dim_compare:wNN 3954, 3962, 3965, 3977
 __cs_generate_variant_loop_end:nwwwNNnn __dim_compare_...
 1774, 1784, 1805 __dim_compare_\efill
 __cs_generate_variant_loop_invalid:NNwNNnn :w 3954
 1784, 1791, 1827 __dim_compare_:w 3954
 __cs_generate_variant_loop_long:wNNnn __dim_compare_<:w 3954
 1777, 1784, 1814 __dim_compare_end:w 3962, 3988
 __cs_generate_variant_loop_same:w . __dim_eval:w
 1784, 1787, 1800 . 89, 3864, 3865, 3894, 3905, 3910,
 __cs_generate_variant_same:N 3917, 3923, 3924, 3925, 3931, 3932,
 1803, 1842, 1842 3933, 3948, 3951, 3957, 3978, 3983,
 __cs_get_function_name:N 25, 1017, 1017 4076, 4082, 4107, 6608, 6610, 6612,
 __cs_get_function_signature:N 6621, 6623, 6625, 6627, 6701, 6721,
 25, 1017, 1019 6734, 6749, 6777, 10502, 11045,
 __cs_parm_from_arg_count:nnF 15114, 15116, 15157, 15159, 15223
 847, 1229, 1229, 1273 __dim_eval_end: 89,
 __cs_parm_from_arg_count_test:nnF . 3864, 3866, 3894, 3905, 3910, 3917,
 1229, 1231, 1250 3927, 3935, 3948, 3951, 4076, 4082,
 __cs_show:www 1393, 1404, 1408 6608, 6610, 6612, 6621, 6623, 6625,
 __cs_split_function:NN 6627, 6701, 6721, 6734, 6749, 6777,
 25, 829, 842, 929, 930, 15114, 15116, 15157, 15159, 15225
 998, 1004, 1018, 1020, 1260, 1294, 1720 __dim_maxmin:wwN 3914, 3923, 3931, 3937
 __cs_split_function_auxi:w __dim_ratio:n 3945, 3946, 3947
 998, 1007, 1012 __dim_strip_bp:n 4267, 4267
 __cs_split_function_auxii:w __dim_strip_pt:n 4267, 4268
 998, 1013, 1014 __dim_to_decimal:w 4079, 4082, 4086
 __cs_tmp:w 25, 1160, 1160, 1168, 1169, __driver_absolute_lengths:n
 1170, 1171, 1172, 1173, 1174, 1175, 16139, 16140, 16170
 1176, 1178, 1179, 1180, 1181, 1182, __driver_box_rotate_begin:
 1183, 1184, 1185, 1186, 1187, 1188, 206, 14870, 16194, 16194
 1189, 1190, 1191, 1192, 1193, 1194, __driver_box_rotate_end:
 1195, 1196, 1197, 1198, 1199, 1200, 206, 14872, 16194, 16229
 1201, 1284, 1309, 1310, 1311, 1312, __driver_box_scale_begin:
 1313, 1314, 1315, 1316, 1317, 1318, 206, 15075, 16230, 16230
 1319, 1320, 1321, 1322, 1323, 1324, __driver_box_scale_end:
 1325, 1326, 1327, 1328, 1329, 1330, 206, 15077, 16230, 16252
 1331, 1332, 1333, 1341, 1342, 1343, __driver_box_use_clip:N
 1344, 1345, 1346, 1347, 1348, 1349, 205, 15108, 16164, 16164
 1350, 1351, 1352, 1353, 1354, 1355,
 1356, 1357, 1358, 1359, 1360, 1361,

_driver_color_ensure_current: 206, 7610,
 7616, 7619, 16266, 16267, 16278, 16288
 _driver_color_reset: 16266, 16270,
 16272, 16282, 16284, 16291, 16293
 _driver_literal:n 16110, 16110, 16157, 16162, 16168,
 16181, 16219, 16236, 16290, 16294
 _driver_literal_direct:n 16129, 16129
 _driver_matrix:n 16149,
 16152, 16156, 16161, 16203, 16244
 _driver_state_restore: 16081, 16089,
 16101, 16106, 16191, 16229, 16252
 _driver_state_save: . 16081, 16082,
 16100, 16104, 16166, 16196, 16232
 _exp_arg_last_unbraced:nn
 . . 1637, 1637, 1640, 1644, 1647, 1652
 _exp_arg_next:Nnn . . . 1462, 1463, 1469
 _exp_arg_next:nnn 1462,
 1462, 1471, 1474, 1482, 1486, 1492
 _exp_eval_error_msg:w 1496, 1500, 1509
 _exp_eval_register:N
 1487, 1496, 1496,
 1508, 1542, 1560, 1578, 1579, 1586,
 1648, 1661, 1673, 1679, 1688, 1708
 _exp_eval_register:c . . 1493, 1496,
 1507, 1537, 1554, 1653, 1663, 1713
 _exp_last_two_unbraced:noN
 1697, 1698, 1699
 _file_add_path:nN . . . 9221, 9222, 9223
 _file_add_path_search:nN
 9221, 9227, 9231
 _file_if_exist:nT
 9264, 9266, 9269, 15662, 15687
 _file_input:V
 . . . 9267, 15431, 15438, 15444, 15451
 _file_input:n 9278, 9284
 _file_input:n_file_input:V . . 9264
 _file_input_aux:n 9264, 9281, 9285, 9301
 _file_input_aux:o 9264, 9282
 _file_name_sanitizenn
 174, 9180, 9180, 9222,
 9274, 9303, 9311, 9354, 9364, 9469
 _file_name_sanitizenn_aux:n
 9180, 9193, 9216
 _file_path_include:n . 9302, 9303, 9304
 fp 11775, 11782
 _fp_o:ww 11772
 _fp*_o:ww 12150
 _fp+_o:ww 11868
 _fp_o:ww 11863
 _fp/_o:ww 12260
 _fp^o:ww 13489
 _fp_acos_o:w 14414, 14414
 _fp_acot_o:Nw 11068, 11070, 14237, 14242
 _fp_acotii_o:Nww
 14240, 14245, 14261, 14280
 _fp_acsc_normal_o:NfwNnw
 14472, 14487, 14495, 14495
 _fp_acsc_o:w 14466, 14466
 _fp_add:NNNn 14790,
 14790, 14791, 14792, 14793, 14794
 _fp_add_big_i_o:wNww
 11939, 11946, 11946, 13317
 _fp_add_big_ii_o:wNww
 11942, 11946, 11954
 _fp_add_inf_o:Nww . 11883, 11903, 11903
 _fp_add_normal_o:Nww 11882, 11923, 11923
 _fp_add_npos_o:NnwNnw
 11926, 11932, 11932
 _fp_add_return_ii_o:Nww
 11885, 11891, 11891, 11896
 _fp_add_significand_carry_o:wwwNN
 11979, 11994, 11994
 _fp_add_significand_no_carry_o:wwwNN
 11981, 11984, 11984
 _fp_add_significand_o:NnnwnnnN . .
 11949, 11957, 11962, 11962
 _fp_add_significand_pack:NNNNNN . .
 11962, 11966, 11969
 _fp_add_significand_test_o:N
 11962, 11964, 11976
 _fp_add_zeros_o:Nww 11881, 11893, 11893
 _fp_and_return:wNw 11772, 11778, 11784
 _fp_array_count:n
 9978, 9978, 10279, 14250
 _fp_array_count_loop:Nw
 9978, 9981, 9985, 9986
 _fp_array_to_clist:n
 . . 10306, 11310, 11311, 14737, 14737
 _fp_array_to_clist_loop:Nw
 14737, 14744, 14749, 14759
 _fp_asec_o:w 14479, 14479
 _fp_asin_auxi_o:NnNww
 14444, 14447, 14447, 14506
 _fp_asin_isqrt:wn . 14447, 14450, 14457
 _fp_asin_normal_o:NfwNnnnnw
 14405, 14421, 14432, 14432
 _fp_asin_o:w 14399, 14399

```

\__fp_atan_Taylor_break:w ..... 14349, 14352, 14362
\__fp_atan_Taylor_loop:www ..... 14344, 14349, 14349, 14357
\__fp_atan_auxi:ww .. 14324, 14338, 14338
\__fp_atan_auxii:w .. 14338, 14339, 14340
\__fp_atan_combine_aux:ww ..... 14365, 14379, 14386
\__fp_atan_combine_o:NwwwN ..... 14284, 14301, 14365, 14365
\__fp_atan_dispatch_o:NNnNw ..... 14237, 14239, 14244, 14247
\__fp_atan_div:wnwnw 14312, 14314, 14314
\__fp_atan_inf_o:NNNw . 14272, 14273, 14274, 14282, 14282, 14417, 14490
\__fp_atan_near:wwn 14314, 14320, 14326
\__fp_atan_near_aux:wn ..... 14314, 14331, 14333
\__fp_atan_normal_o:NNwNw ..... 14276, 14292, 14292
\__fp_atan_o:Nw 11072, 11074, 14237, 14237
\__fp_atan_test_o:NwNwN ..... 14295, 14299, 14299, 14454
\__fp_atanii_o:Nw ..... 14240, 14245, 14261, 14261, 14281
\__fp_basics_pack_high:NNNNw .... 11842, 11844, 11987, 12138, 12239, 12251, 12391, 12584, 13043
\__fp_basics_pack_high_carry:w .... 11842, 11847, 11851
\__fp_basics_pack_low:NNNNw 11842, 11842, 11989, 12140, 12241, 12253, 12393, 12533, 12535, 12586, 13045
\__fp_basics_pack_weird_high:NNNNNNNw ..... 11853, 11861, 11998, 12402
\__fp_basics_pack_weird_low:NNNNw .. 11853, 11853, 12000, 12404
\__fp_case_return:nw ..... 9926, 9926, 9940, 9943, 9948, 10366, 13250, 14272, 14273, 14274, 14537, 14587, 14655, 14657, 14658, 14703
\__fp_case_return_i_o:ww . 9933, 9933, 11884, 11898, 11907, 12183, 14264
\__fp_case_return_ii_o:ww ..... 9933, 9935, 12184, 13538, 13556, 14265
\__fp_case_return_o:Nw ..... 9927, 9927, 13285, 13290, 13293, 13493, 13498, 13520, 13529, 13750, 13780, 14473, 14475
\__fp_case_return_o:Nww ..... 9931, 9931, 12185, 12186, 12189, 12190, 13540, 13549, 13552
\__fp_case_return_same_o:w ..... 9929, 9929, 12414, 12418, 13070, 13297, 13517, 13735, 13743, 13758, 13773, 13788, 13795, 13803, 13818, 14402, 14410, 14428, 14474, 14491
\__fp_case_use:nw ..... 9925, 9925, 11909, 12181, 12182, 12187, 12188, 12268, 12271, 12416, 13063, 13066, 13523, 13736, 13741, 13751, 13756, 13766, 13771, 13781, 13786, 13796, 13801, 13811, 13816, 14404, 14407, 14417, 14419, 14425, 14469, 14471, 14482, 14485, 14490, 14540, 14551, 14590, 14600
\__fp_chk:w ..... 9744, 9745, 9748, 9757, 9758, 9759, 9760, 9761, 9763, 9764, 9767, 9773, 9777, 9797, 9800, 9801, 9811, 9821, 9834, 9853, 9937, 10120, 10125, 10309, 10318, 10320, 11025, 11595, 11620, 11621, 11731, 11732, 11735, 11746, 11747, 11755, 11756, 11764, 11775, 11778, 11790, 11815, 11869, 11888, 11889, 11891, 11892, 11893, 11901, 11904, 11920, 11921, 11923, 11932, 12008, 12159, 12193, 12194, 12197, 12276, 12412, 12420, 12422, 12599, 12603, 13060, 13072, 13074, 13282, 13299, 13301, 13490, 13509, 13511, 13512, 13514, 13526, 13531, 13533, 13558, 13559, 13561, 13577, 13662, 13675, 13677, 13681, 13685, 13732, 13745, 13747, 13760, 13762, 13775, 13777, 13790, 13792, 13805, 13807, 13820, 13830, 14262, 14277, 14278, 14282, 14293, 14399, 14412, 14414, 14430, 14433, 14443, 14466, 14477, 14479, 14493, 14495, 14500, 14533, 14558, 14561, 14583, 14607, 14610, 14651, 14674, 14724, 14725
\__fp_compare:wNNNNw ..... 11391
\__fp_compare_aux:wn 11603, 11606, 11614
\__fp_compare_back:ww ..... 11478, 11616, 11619, 11619, 11745
\__fp_compare_nan:w ..... 11619, 11624, 11625, 11646
\__fp_compare_npos:wnw .....

```


.. 11630, 11647, 11647, 12010, 12846
 _fp_compare_return:w 11590, 11592, 11595
 _fp_compare_significand:nnnnnnnn .
 11647, 11650, 11655
 _fp_cos_o:w 13747, 13747
 _fp_cot_o:w 13807, 13807
 _fp_cot_zero_o:Nfw
 13765, 13807, 13810, 13822
 _fp_csc_o:w 13762, 13762
 _fp_decimate:nNnnnn
 .. 9879, 9879, 9956, 10322, 11948,
 11956, 12035, 13333, 13337, 14616
 _fp_decimate_:Nnnnn 9891, 9891
 _fp_decimate_pack:nnnnnnnnnw
 9902, 9921, 9921
 _fp_decimate_pack:nnnnnnnw . 9922, 9923
 _fp_decimate_tiny:Nnnnn ... 9891, 9893
 _fp_div_npos_o:Nww 12265, 12275, 12275
 _fp_div_significand_calc:wwnnnnnnnn
 12292,
 12301, 12301, 12347, 13143, 13150
 _fp_div_significand_calc_i:wwnnnnnnnn
 12301, 12304, 12309
 _fp_div_significand_calc_ii:wwnnnnnnnn
 12301, 12306, 12326
 _fp_div_significand_i_o:wnnw
 12282, 12288, 12288
 _fp_div_significand_ii:wnn
 .. 12296, 12297, 12298, 12343, 12343
 _fp_div_significand_iii:wwnnnnnn ..
 12299, 12350, 12350
 _fp_div_significand_iv:wwnnnnnnnn .
 12353, 12358, 12358
 _fp_div_significand_large_o:wwNNNNwN
 12384, 12398, 12398
 _fp_div_significand_pack:NNN 12345,
 12378, 12378, 13130, 13148, 13156
 _fp_div_significand_small_o:wwNNNNwN
 12382, 12388, 12388
 _fp_div_significand_test_o:w
 12290, 12379, 12379
 _fp_div_significand_v:NN
 12363, 12365, 12368
 _fp_div_significand_v:NNw 12358
 _fp_div_significand_vi:Nw
 12358, 12361, 12369
 _fp_division_by_zero_o:NNww 10092,
 10132, 10136, 12269, 12272, 13525
 _fp_division_by_zero_o:Nnw 10084,
 10132, 10135, 13067, 13826, 13827
 _fp_ep_compare:www 12841, 12841, 14308
 _fp_ep_compare_aux:www
 12841, 12842, 12843
 _fp_ep_div:wwwn 12871, 12871, 12982,
 14227, 14323, 14327, 14336, 14503
 _fp_ep_div_eps_pack:NNNNw
 12901, 12905, 12907, 12910
 _fp_ep_div_epsi:wnNNNNnn
 12898, 12901, 12901
 _fp_ep_div_epsi:wnNNNNnn
 12901, 12903, 12912
 _fp_ep_div_esti:wwwn
 12877, 12880, 12880
 _fp_ep_div_estii:wwnnwn
 12880, 12882, 12888
 _fp_ep_div_estiii:NNNNwwww
 12880, 12890, 12895
 _fp_ep_inv_to_float:wN
 .. 12978, 12980, 12986, 13769, 13784
 _fp_ep_isqrt:wn .. 12924, 12924, 14464
 _fp_ep_isqrt_aux:wn 12924
 _fp_ep_isqrt_auxi:wn .. 12927, 12929
 _fp_ep_isqrt_auxii:wwnnwn
 12924, 12931, 12937
 _fp_ep_isqrt_epsi:wN 12961, 12964, 12964
 _fp_ep_isqrt_epsi:wwN
 .. 12964, 12967, 12968, 12969, 12971
 _fp_ep_isqrt_esti:wwnnwn
 12939, 12942, 12942, 12947
 _fp_ep_isqrt_estii:wwnnwn
 12942, 12945, 12952
 _fp_ep_isqrt_estiii:NNNNwwww
 12942, 12954, 12958
 _fp_ep_mul:wwwn 12856,
 12856, 14184, 14214, 14451, 14462
 _fp_ep_mul_raw:wwwN
 .. 12856, 12862, 12866, 13848, 14134
 _fp_ep_to_ep:wwN 12807, 12807, 12858,
 12861, 12873, 12876, 12926, 14452
 _fp_ep_to_ep_end:www 12807, 12821, 12825
 _fp_ep_to_ep_loop:N 12807,
 12812, 12816, 12823, 12826, 14135
 _fp_ep_to_ep_zero:ww 12807, 12831, 12839
 _fp_ep_to_fixed:wn 12789,
 12789, 13845, 14330, 14339, 14449
 _fp_ep_to_fixed_auxi:www
 12789, 12791, 12796
 _fp_ep_to_fixed_auxii:nnnnnnwn ..
 12789, 12802, 12805

_fp_ep_to_float:wwN . [12978](#), [12978](#),
 [12983](#), [12990](#), [13739](#), [13754](#), [14233](#)
 _fp_error:nfn [10061](#), [10070](#),
 [10095](#), [10123](#), [10146](#), [10283](#), [11309](#)
 _fp_error:nnfn [10053](#), [10087](#), [10146](#)
 _fp_error:nnnn [10146](#), [10146](#), [10148](#)
 _fp_exp_Taylor:Nnnwn
 [13334](#), [13350](#), [13350](#), [13486](#)
 _fp_exp_Taylor_break:Nww
 [13350](#), [13364](#), [13375](#)
 _fp_exp_Taylor_ii:ww [13356](#), [13359](#)
 _fp_exp_Taylor_loop:www
 [13350](#), [13360](#), [13361](#), [13370](#)
 _fp_exp_after_?_f:nw [10443](#)
 _fp_exp_after_array_f:w
 [9854](#), [9854](#), [9857](#),
 [10997](#), [11794](#), [11805](#), [11828](#), [11836](#)
 _fp_exp_after_f:nw
 [9801](#), [9821](#), [11024](#), [11156](#)
 _fp_exp_after_mark_f:nw [10443](#), [10452](#)
 _fp_exp_after_normal:Nwwwww [9843](#), [9851](#)
 _fp_exp_after_normal:nNNw
 [9804](#), [9814](#), [9824](#), [9841](#), [9841](#)
 _fp_exp_after_o:nw [9801](#), [9811](#)
 _fp_exp_after_o:w
 [9801](#), [9801](#), [9930](#), [9934](#), [9936](#),
 [10316](#), [10360](#), [10378](#), [11497](#), [11763](#),
 [11780](#), [11892](#), [12601](#), [13674](#), [13679](#)
 _fp_exp_after_special:nNNw
 [9806](#), [9816](#), [9826](#), [9831](#), [9831](#)
 _fp_exp_after_stop_f:nw [9854](#), [9860](#)
 _fp_exp_large:w [13377](#),
 [13386](#), [13391](#), [13392](#), [13393](#), [13394](#),
 [13395](#), [13396](#), [13397](#), [13398](#), [13399](#),
 [13407](#), [13408](#), [13409](#), [13410](#), [13411](#),
 [13412](#), [13413](#), [13414](#), [13415](#), [13423](#),
 [13424](#), [13425](#), [13426](#), [13427](#), [13428](#),
 [13429](#), [13430](#), [13431](#), [13439](#), [13440](#),
 [13441](#), [13442](#), [13443](#), [13444](#), [13445](#),
 [13446](#), [13447](#), [13455](#), [13456](#), [13457](#),
 [13458](#), [13459](#), [13460](#), [13461](#), [13462](#),
 [13463](#), [13471](#), [13472](#), [13473](#), [13474](#),
 [13475](#), [13476](#), [13477](#), [13478](#), [13479](#)
 _fp_exp_large:wN [13377](#), [13466](#), [13468](#)
 _fp_exp_large_after:wwn
 [13377](#), [13482](#), [13484](#)
 _fp_exp_large_i:wN [13377](#), [13450](#), [13452](#)
 _fp_exp_large_ii:wN [13377](#), [13434](#), [13436](#)
 _fp_exp_large_iii:wN [13377](#), [13418](#), [13420](#)
 _fp_exp_large_iv:wN [13377](#), [13402](#), [13404](#)
 _fp_exp_large_v:wN [13377](#), [13388](#), [13661](#)
 _fp_exp_normal:w [13287](#), [13301](#), [13301](#)
 _fp_exp_o:w [13282](#), [13282](#)
 _fp_exp_overflow: [13323](#), [13348](#)
 _fp_exp_pos:NNwnw [13304](#), [13306](#), [13309](#)
 _fp_exp_pos:Nwnw [13301](#)
 _fp_exp_pos_large:NnnNwn
 [13338](#), [13377](#), [13377](#)
 _fp_expand:n [9987](#), [9987](#), [14741](#)
 _fp_expand_loop:nwnN
 [9987](#), [9989](#), [9991](#), [9994](#)
 _fp_exponent:w [9777](#), [9777](#)
 _fp_fixed_add:Nnnnnwn
 [12683](#), [12683](#), [12684](#), [12685](#)
 _fp_fixed_add:nnNnnwn
 [12683](#), [12691](#), [12693](#)
 _fp_fixed_add:wn
 [12683](#), [12683](#), [12922](#), [13224](#),
 [13232](#), [13243](#), [13261](#), [14335](#), [14395](#)
 _fp_fixed_add_after:NNNNwn
 [12683](#), [12687](#), [12701](#)
 _fp_fixed_add_one:wN [12615](#),
 [12615](#), [12915](#), [13367](#), [13376](#), [14461](#)
 _fp_fixed_add_pack:NNNNwn
 [12683](#), [12689](#), [12696](#), [12699](#)
 _fp_fixed_continue:wn [12614](#),
 [12614](#), [12859](#), [12864](#), [12874](#), [13390](#),
 [13406](#), [13422](#), [13438](#), [13454](#), [13470](#),
 [13637](#), [13883](#), [14172](#), [14453](#), [14462](#)
 _fp_fixed_div_int:wnN
 [12652](#), [12657](#), [12665](#), [12677](#)
 _fp_fixed_div_int:wwN
 [12652](#), [12652](#), [13223](#), [13366](#), [14354](#)
 _fp_fixed_div_int_after:Nw
 [12652](#), [12654](#), [12682](#)
 _fp_fixed_div_int_auxi:wnn
 [12652](#), [12658](#),
 [12659](#), [12660](#), [12661](#), [12662](#), [12672](#)
 _fp_fixed_div_int_auxii:wnn
 [12652](#), [12663](#), [12680](#)
 _fp_fixed_div_int_pack:Nw
 [12652](#), [12675](#), [12681](#)
 _fp_fixed_div_myriad:wn
 [12620](#), [12620](#), [12919](#)
 _fp_fixed_inv_to_float:wN
 [12985](#), [12985](#), [13306](#), [13573](#)
 _fp_fixed_mul:nnnnnnw
 [12703](#), [12723](#), [12725](#)
 _fp_fixed_mul:wn [12703](#), [12703](#),
 [12868](#), [12899](#), [12914](#), [12916](#), [12920](#),

12973, 12976, 12989, 13225, 13235,
 13275, 13368, 13387, 13487, 13583,
 14141, 14195, 14342, 14375, 14377
 _fp_fixed_mul_add:Nwnnnwnnn
 .. 12736, 12746, 12756, 12760, 12760
 _fp_fixed_mul_add:nnnnwnnn
 12771, 12773, 12773
 _fp_fixed_mul_add:nnnnwnnwN
 12778, 12784, 12784
 _fp_fixed_mul_add:wwn .. 12730, 12730
 _fp_fixed_mul_after:wwn
 12622, 12628, 12628, 12631,
 12705, 12732, 12742, 12752, 13600
 _fp_fixed_mul_one_minus_mul:wwn 12730
 _fp_fixed_mul_short:wwn
 12629, 12629,
 12897, 12918, 12960, 12962, 14388
 _fp_fixed_mul_sub_back:wwn
 12730, 12740,
 12974, 14162, 14164, 14165, 14166,
 14167, 14168, 14169, 14170, 14171,
 14175, 14177, 14178, 14179, 14180,
 14181, 14182, 14183, 14208, 14210,
 14211, 14212, 14213, 14216, 14218,
 14219, 14220, 14221, 14355, 14363
 _fp_fixed_one_minus_mul:wwn ... 12750
 _fp_fixed_sub:wwn
 12683, 12684, 12966, 13241, 13257,
 13269, 13887, 14336, 14393, 14459
 _fp_fixed_to_float:Nw
 12992, 12992, 13250
 _fp_fixed_to_float:wN
 12979, 12992, 12992, 12993,
 13270, 13280, 13304, 13569, 14383
 _fp_fixed_to_float_pack:ww 13024, 13034
 _fp_fixed_to_float_rad:wN
 12987, 12987, 14383
 _fp_fixed_to_float_round_up:wnnnw
 13037, 13041
 _fp_fixed_to_float_zero:w 13020, 13029
 _fp_fixed_to_loop:N 12997, 13007, 13011
 _fp_fixed_to_loop_end:w . 13013, 13017
 _fp_from_dim:wNnnnnnn
 14693, 14716, 14719
 _fp_from_dim:wNw .. 14693, 14705, 14714
 _fp_from_dim:wnnnwNn .. 14720, 14721
 _fp_from_dim:wnnnwNw 14693
 _fp_from_dim_test:ww
 .. 10500, 11043, 14693, 14695, 14700
 _fp_function_apply:nw
 11512, 11528, 11550, 11550
 _fp_function_args:Nwn
 11517, 11530, 11540
 _fp_function_store:wwNwn
 11550, 11554, 11555, 11561
 _fp_function_store_end:wnn
 11550, 11556, 11564
 _fp_inf_fp:N 9763, 9764, 10108
 _fp_invalid_operation:nw
 10050, 10132, 10132,
 10144, 14542, 14553, 14592, 14602
 _fp_invalid_operation_o:Nww
 10058, 10132, 10133,
 11912, 11915, 12187, 12188, 13668
 _fp_invalid_operation_o:fw
 10143, 13742, 13757,
 13772, 13787, 13802, 13817, 14408,
 14426, 14442, 14470, 14483, 14499
 _fp_invalid_operation_o:nw
 .. 10143, 10143, 10145, 12416, 13063
 _fp_invalid_operation_tl_o:ff
 10067, 10132, 10134, 10304
 _fp_ln_Taylor:wwNw 13215, 13216, 13216
 _fp_ln_Taylor_break:w .. 13221, 13232
 _fp_ln_Taylor_loop:www
 13217, 13218, 13227
 _fp_ln_c:NwNw 13207, 13238, 13238
 _fp_ln_div_after:Nw 13110, 13159
 _fp_ln_div_i:w 13132, 13141
 _fp_ln_div_ii:wwn
 .. 13135, 13136, 13137, 13138, 13146
 _fp_ln_div_vi:wwn 13139, 13154
 _fp_ln_exponent:wn 13086, 13247, 13247
 _fp_ln_exponent_one:ww .. 13252, 13266
 _fp_ln_exponent_small:NNw
 13255, 13259, 13272
 _fp_ln_npos_o:w .. 13072, 13074, 13074
 _fp_ln_o:w 13060, 13060
 _fp_ln_significand:NNNnnnnN
 13085, 13088, 13088, 13581
 _fp_ln_square_t_after:w . 13183, 13214
 _fp_ln_square_t_pack:NNNNw
 .. 13185, 13187, 13189, 13191, 13212
 _fp_ln_t_large:NNw 13164, 13171, 13181
 _fp_ln_t_small:Nw 13162, 13169
 _fp_ln_twice_t_after:w .. 13195, 13211
 _fp_ln_twice_t_pack:Nw 13197,
 13199, 13201, 13203, 13205, 13210
 _fp_ln_x_ii:wnnnw . 13090, 13108, 13108

- _fp_ln_x_iii:NNNNNNw ... 13117, 13121
- _fp_ln_x_iii_var:NNNNNw . 13115, 13123
- _fp_ln_x_iv:wnnnnnnnn .. 13113, 13128
- _fp_max_fp:N 9765, 9771
- _fp_min_fp:N 9765, 9765
- _fp_minmax_auxi:ww
..... 11739, 11751, 11758, 11758
- _fp_minmax_auxii:ww
..... 11741, 11749, 11758, 11760
- _fp_minmax_break_o:w 11732, 11762, 11762
- _fp_minmax_loop:Nww
.. 11726, 11728, 11734, 11734, 11754
- _fp_minmax_o:Nw
..... 11076, 11078, 11723, 11723
- _fp_mul_cases_o:NnNww
..... 12152, 12158, 12262
- _fp_mul_cases_o:nNnnw 12158
- _fp_mul_npos_o:Nww
..... 12155, 12196, 12196, 14723
- _fp_mul_significand_drop:NNNNNw ..
..... 12205,
12214, 12216, 12218, 12220, 12224
- _fp_mul_significand_keep:NNNNNw ..
..... 12205, 12210, 12212, 12226
- _fp_mul_significand_large_f:NwwNNNN
..... 12233, 12237, 12237
- _fp_mul_significand_o:nnnnNnnnn ..
..... 12203, 12205, 12205
- _fp_mul_significand_small_f:NNwwN
..... 12231, 12248, 12248
- _fp_mul_significand_test_f:NNN ...
..... 12207, 12228, 12228
- _fp_neg_sign:N 9786, 9786, 11866
- _fp_new_function:NNnnn
..... 11517, 11524, 11539
- _fp_new_function:Ncfnn .. 11517, 11519
- _fp_not_o:w 10969, 11764
- _fp_overflow:w 9793, 10132, 10137
- _fp_pack:NNNNNw . 9861, 9864, 12624,
12634, 12637, 12640, 12643, 12646,
12707, 12710, 12713, 12716, 12719,
13602, 13605, 13608, 13611, 13614
- _fp_pack_Bigg:NNNNNNw
..... 9870, 9873, 12315,
12318, 12321, 12332, 12335, 12338
- _fp_pack_big:NNNNNNw .. 9865, 9868,
12465, 12468, 12471, 12474, 12477,
12480, 12483, 12487, 12734, 12744,
12754, 12763, 12766, 12769, 12776
- _fp_pack_eight:wNNNNNNNN 9877, 9877,
12131, 12436, 12798, 13854, 13855
- _fp_pack_twice_four:wNNNNNNNN 9875,
9875, 10353, 10354, 12074, 12075,
12799, 12800, 12801, 12833, 12834,
12835, 13022, 13023, 13353, 13354,
13355, 13856, 13857, 14070, 14716
- _fp_parse:n
.... 10384, 10497, 11123, 11123,
11552, 11593, 11607, 11617, 14526,
14581, 14649, 14686, 14732, 14734,
14736, 14768, 14770, 14772, 14795
- _fp_parse_after:ww 11123, 11126, 11133
- _fp_parse_apply_binary:NwNwN
..... 11147, 11147, 11269, 11314
- _fp_parse_apply_compare:NwNNNNNNwN
..... 11462, 11471
- _fp_parse_apply_compare_aux:NNwN .
..... 11482, 11485, 11490
- _fp_parse_apply_juxtapose:NwNwN ...
..... 11291, 11295, 11305
- _fp_parse_apply_unary:NNNwN
.. 10946, 10946, 10955, 11051, 11060
- _fp_parse_compare:NNNNNNNN
..... 11391, 11393, 11398,
11403, 11408, 11421, 11429, 11492
- _fp_parse_compare_auxi:NNNNNNNN ...
..... 11391, 11424, 11432, 11449
- _fp_parse_compare_auxii:NNNNN
..... 11391,
11440, 11441, 11442, 11443, 11447
- _fp_parse_compare_end:NNNNw
..... 11391, 11444, 11458
- _fp_parse_continue:NwN
..... 11136, 11139, 11146,
11149, 11500, 11802, 11825, 11833
- _fp_parse_continue_compare:NNwNN .
..... 11493, 11508
- _fp_parse_digits:N 10420, 10421
- _fp_parse_digits_i:N ... 10402, 10419
- _fp_parse_digits_ii:N .. 10402, 10418
- _fp_parse_digits_iii:N .. 10402, 10417
- _fp_parse_digits_iv:N .. 10402, 10416
- _fp_parse_digits_v:N ... 10402, 10415
- _fp_parse_digits_vi:N
..... 10402, 10414, 10656, 10704
- _fp_parse_digits_vii:N
..... 10402, 10643, 10693
- _fp_parse_excl_error:
..... 11391, 11416, 11425

_fp_parse_expand:w
 10384, 10384, 10386,
 10411, 10448, 10471, 10530, 10561,
 10599, 10601, 10620, 10622, 10644,
 10661, 10674, 10694, 10724, 10752,
 10768, 10779, 10802, 10831, 10841,
 10848, 10861, 10877, 10897, 10908,
 10965, 10991, 11000, 11056, 11065,
 11129, 11203, 11218, 11243, 11273,
 11298, 11338, 11350, 11367, 11383,
 11456, 11469, 11515, 11535, 11798
 _fp_parse_exponent:N 10470,
 10635, 10784, 10851, 10853, 10853
 _fp_parse_exponent:Nw
 10659, 10672, 10721,
 10749, 10800, 10829, 10848, 10848
 _fp_parse_exponent_aux:N
 10853, 10856, 10863
 _fp_parse_exponent_body:N
 10879, 10883, 10883
 _fp_parse_exponent_digits:N
 10887, 10899, 10899, 10903
 _fp_parse_exponent_keep:N 10910
 _fp_parse_exponent_keep:NTF
 10890, 10910
 _fp_parse_exponent_sign:N
 10869, 10873, 10873, 10876
 _fp_parse_function:NNN 11049, 11058,
 11068, 11070, 11072, 11074, 11076,
 11078, 11099, 11101, 11103, 11117
 _fp_parse_infix:NN
 10447, 10539, 10578, 10998, 11012,
 11024, 11046, 11156, 11161, 11216
 _fp_parse_infix_
 _fp_parse_infix_efill
 :N 11391
 _fp_parse_infix_ 10995, 11197, 11212,
 11222, 11226, 11228, 11233, 11281,
 11282, 11284, 11285, 11321, 11333,
 11336, 11345, 11348, 11360, 11371
 _fp_parse_infix_:Nw 11330
 _fp_parse_infix_(N 11289
 _fp_parse_infix_)N 11210
 _fp_parse_infix_*N 11316
 _fp_parse_infix_+N 10384, 11254
 _fp_parse_infix_-N 11254
 _fp_parse_infix_/N 11254
 _fp_parse_infix::N
 11358, 11374, 11387, 11787
 _fp_parse_infix_:N 11391
 _fp_parse_infix_<:N 11391
 _fp_parse_infix_?:N 11358
 _fp_parse_infix_^:N 11254
 _fp_parse_infix_after_operand:NwN
 .. 10464, 10506, 10972, 11154, 11154
 _fp_parse_infix_and:N
 11254, 11286, 11352
 _fp_parse_infix_check:NNN 11182, 11192
 _fp_parse_infix_comma:w . 11236, 11246
 _fp_parse_infix_comma_gobble:w ...
 11239, 11248
 _fp_parse_infix_end:N
 11130, 11134, 11208, 11208, 11209
 _fp_parse_infix_juxtapose:N 11171,
 11180, 11290, 11291, 11291, 11302
 _fp_parse_infix_mark:NNN
 11168, 11207, 11207
 _fp_parse_infix_mul:N
 11254, 11283, 11324
 _fp_parse_infix_or:N 11254, 11287, 11340
 _fp_parse_large:N . 10606, 10689, 10689
 _fp_parse_large_leading:wwNN
 10691, 10696, 10696
 _fp_parse_large_round:NN
 10732, 10804, 10804
 _fp_parse_large_round_aux:wNN
 10804, 10813, 10833
 _fp_parse_large_round_test:NN
 10804, 10817, 10822
 _fp_parse_large_trailing:wwNN
 10702, 10726, 10726
 _fp_parse_letters:N
 10521, 10532, 10543, 10556
 _fp_parse_lparen_after:NwN
 10978, 10983, 10993
 _fp_parse_one:Nw 10384,
 10422, 10422, 10583, 10945, 11142
 _fp_parse_one_digit:NN
 10435, 10504, 10504
 _fp_parse_one_fp:NN 10427, 10443, 10443
 _fp_parse_one_other:NN
 10438, 10512, 10512
 _fp_parse_one_register:NN
 10430, 10462, 10462
 _fp_parse_one_register_aux:Nw
 10462, 10467, 10481
 _fp_parse_one_register_auxii:wwwNw
 10462, 10484, 10491
 _fp_parse_one_register_dim:ww
 10462, 10486, 10494, 10498

_fp_parse_one_register_int:www ...	_fp_parse_unary_function:nNN ...
..... 10462 , 10488 , 10496 11049 , 11049 , 11080 ,
_fp_parse_one_register_mu:www ...	11082 , 11084 , 11086 , 11094 , 11096
..... 10462 , 10487 , 10493	_fp_parse_word:Nw . 10518 , 10532 , 10532
_fp_parse_operand:Nw	_fp_parse_word_abs:N ... 11079 , 11079
.... 10384 , 10961 , 10963 , 10987 ,	_fp_parse_word_acos:N 11087
10989 , 11056 , 11065 , 11128 , 11136 ,	_fp_parse_word_acosd:N 11087
11136 , 11242 , 11272 , 11297 , 11366 ,	_fp_parse_word_acot:N .. 11067 , 11067
11382 , 11469 , 11515 , 11535 , 11797	_fp_parse_word_acotd:N .. 11067 , 11069
_fp_parse_pack_carry:w	_fp_parse_word_acsc:N 11087
..... 10676 , 10684 , 10687	_fp_parse_word_acscd:N 11087
_fp_parse_pack_leading:NNNNnw ...	_fp_parse_word_asec:N 11087
..... 10639 , 10676 , 10681 , 10699	_fp_parse_word_asecd:N 11087
_fp_parse_pack_trailing:NNNNNNnw .	_fp_parse_word_asin:N 11087
..... 10649 ,	_fp_parse_word_asind:N 11087
10676 , 10676 , 10718 , 10729 , 10736	_fp_parse_word_atan:N .. 11067 , 11071
_fp_parse_prefix:NNN 10524 , 10563 , 10563	_fp_parse_word_atand:N .. 11067 , 11073
_fp_parse_prefix_ 10981	_fp_parse_word_bp:N 11020
_fp_parse_prefix_(Nw 10978	_fp_parse_word_cc:N 11020
_fp_parse_prefix_+Nw 10945	_fp_parse_word_ceil:N .. 11098 , 11102
_fp_parse_prefix_-Nw 10951	_fp_parse_word_cm:N 11020
_fp_parse_prefix_.Nw 10970	_fp_parse_word_cos:N 11087
_fp_parse_prefix:Nw 10951	_fp_parse_word_cosd:N 11087
_fp_parse_prefix_unknown:NNN ...	_fp_parse_word_cot:N 11087
..... 10563 , 10566 , 10571	_fp_parse_word_cotd:N 11087
_fp_parse_return_semicolon:w ...	_fp_parse_word_csc:N 11087
.... 10385 , 10385 , 10409 , 10559 ,	_fp_parse_word_cscd:N 11087
10766 , 10777 , 10859 , 10891 , 10906	_fp_parse_word_dd:N 11020
_fp_parse_round:Nw	_fp_parse_word_deg:N 11009
.. 11104 , 11107 , 11110 , 11113 , 11121	_fp_parse_word_em:N 11039
_fp_parse_round_after:wN	_fp_parse_word_ex:N 11039
.. 10781 , 10781 , 10786 , 10795 , 10836	_fp_parse_word_exp:N ... 11079 , 11081
_fp_parse_round_loop:N 10754 ,	_fp_parse_word_false:N 11009
10754 , 10759 , 10797 , 10815 , 10840	_fp_parse_word_floor:N .. 11098 , 11100
_fp_parse_round_up:N	_fp_parse_word_in:N 11020
..... 10754 , 10762 , 10770 , 10774	_fp_parse_word_inf:N 11009
_fp_parse_small:N . 10626 , 10637 , 10637	_fp_parse_word_ln:N 11079 , 11083
_fp_parse_small_leading:wwNN ...	_fp_parse_word_max:N ... 11067 , 11075
..... 10641 , 10646 , 10646 , 10708	_fp_parse_word_min:N ... 11067 , 11077
_fp_parse_small_round:NN	_fp_parse_word_mm:N 11020
..... 10668 , 10786 , 10786 , 10825	_fp_parse_word_nan:N 11009
_fp_parse_small_trailing:wwNN ...	_fp_parse_word_nc:N 11020
..... 10654 , 10663 , 10663 , 10740	_fp_parse_word_nd:N 11020
_fp_parse_strim_end:w	_fp_parse_word_pc:N 11020
..... 10612 , 10618 , 10622	_fp_parse_word_pi:N 11009
_fp_parse_strim_zeros:N	_fp_parse_word_pt:N 11020
.. 10593 , 10612 , 10612 , 10616 , 10976	_fp_parse_word_round:N .. 11104 , 11104
_fp_parse_trim_end:w 10586 , 10596 , 10601	_fp_parse_word_sec:N 11087
_fp_parse_trim_zeros:N	_fp_parse_word_secd:N 11087
..... 10510 , 10586 , 10586 , 10589	_fp_parse_word_sin:N 11087

_fp_parse_word_sind:N [11087](#)
 _fp_parse_word_sp:N [11020](#)
 _fp_parse_word_sqrt:N .. [11079](#), [11085](#)
 _fp_parse_word_tan:N [11087](#)
 _fp_parse_word_tand:N [11087](#)
 _fp_parse_word_true:N [11009](#)
 _fp_parse_word_trunc:N .. [11098](#), [11098](#)
 _fp_parse_zero:
 [10608](#), [10628](#), [10632](#), [10632](#)
 _fp_pow_B:wwN [13584](#), [13619](#)
 _fp_pow_C_neg:w [13622](#), [13639](#)
 _fp_pow_C_overflow:w [13627](#), [13634](#), [13655](#)
 _fp_pow_C_pack:w .. [13641](#), [13649](#), [13660](#)
 _fp_pow_C_pos:w [13625](#), [13644](#)
 _fp_pow_C_pos_loop:wN
 [13645](#), [13646](#), [13653](#)
 _fp_pow_exponent:Nwnnnnw
 [13590](#), [13593](#), [13598](#)
 _fp_pow_exponent:wnN ... [13582](#), [13587](#)
 _fp_pow_neg:www .. [13500](#), [13662](#), [13662](#)
 _fp_pow_neg_aux:wNN [13662](#), [13665](#), [13677](#)
 _fp_pow_neg_case:w [13664](#), [13685](#), [13685](#)
 _fp_pow_neg_case_aux:NNNNNNnw ...
 [13685](#), [13700](#), [13707](#), [13717](#)
 _fp_pow_neg_case_aux:nnnnn
 [13685](#), [13689](#), [13693](#)
 _fp_pow_normal:ww . [13505](#), [13533](#), [13533](#)
 _fp_pow_npos:Nww .. [13544](#), [13561](#), [13561](#)
 _fp_pow_npos_aux:NNnw
 [13567](#), [13571](#), [13577](#), [13577](#)
 _fp_pow_zero_or_inf:ww
 [13507](#), [13514](#), [13514](#)
 _fp_reverse_args:Nww
 [9740](#), [9740](#), [14225](#), [14310](#), [14422](#), [14488](#)
 _fp_round:NNN
 [10187](#), [10221](#), [10224](#), [11991](#), [12002](#),
 [12243](#), [12255](#), [12395](#), [12406](#), [12590](#)
 _fp_round:Nwn
 [10280](#), [10300](#), [10302](#), [10309](#), [14691](#)
 _fp_round:Nww [10281](#), [10300](#), [10300](#)
 _fp_round_digit:Nw
 [9901](#), [9921](#), [10238](#), [10238](#), [12005](#),
 [12147](#), [12246](#), [12258](#), [12409](#), [12595](#)
 _fp_round_name_from_cs:N
 [10284](#), [10289](#), [10289](#), [10305](#)
 _fp_round_neg:NNN
 [10249](#), [10275](#), [12109](#), [12124](#), [12142](#)
 _fp_round_normal:NNwNnn
 [10300](#), [10327](#), [10347](#)
 _fp_round_normal:NnnwNNnn
 [10300](#), [10323](#), [10325](#)
 _fp_round_normal:NwNNnw
 [10300](#), [10312](#), [10320](#)
 _fp_round_normal_end:wwNnn
 [10300](#), [10355](#), [10358](#)
 _fp_round_o:Nw [10276](#),
 [10276](#), [11099](#), [11101](#), [11103](#), [11118](#)
 _fp_round_pack:Nw . [10300](#), [10331](#), [10345](#)
 _fp_round_return_one:
 [10187](#), [10193](#), [10203](#),
 [10211](#), [10215](#), [10253](#), [10261](#), [10269](#)
 _fp_round_s:NNw
 [10222](#), [10222](#), [10790](#), [10808](#)
 _fp_round_special:NwwNnn
 [10300](#), [10350](#), [10363](#)
 _fp_round_special_aux:Nw
 [10300](#), [10369](#), [10376](#)
 _fp_round_to_nearest:NNN
 [10187](#), [10208](#),
 [10221](#), [10274](#), [11118](#), [11122](#), [14691](#)
 _fp_round_to_nearest_neg:NNN
 [10249](#), [10274](#), [10275](#)
 _fp_round_to_ninf:NNN
 [10187](#), [10189](#), [10293](#), [11101](#), [11113](#)
 _fp_round_to_ninf_neg:NNN [10249](#), [10249](#)
 _fp_round_to_pinf:NNN
 [10187](#), [10199](#), [10295](#), [11103](#), [11107](#)
 _fp_round_to_pinf_neg:NNN [10249](#), [10265](#)
 _fp_round_to_zero:NNN
 [10187](#), [10198](#), [10291](#), [11099](#), [11110](#)
 _fp_round_to_zero_neg:NNN [10249](#), [10258](#)
 _fp_rrot:www [9741](#), [9741](#), [14355](#)
 _fp_sanitiz:Nw [9788](#),
 [9788](#), [9799](#), [10361](#), [10379](#), [11934](#),
 [12028](#), [12199](#), [12278](#), [12424](#), [13076](#),
 [13312](#), [13563](#), [14187](#), [14231](#), [14367](#)
 _fp_sanitiz:wN [9788](#), [9799](#), [10509](#), [10975](#)
 _fp_sanitiz_zero:w .. [9788](#), [9795](#), [9800](#)
 _fp_sec_o:w [13777](#), [13777](#)
 _fp_set_sign_o:w .. [10968](#), [12599](#), [12599](#)
 _fp_sin_o:w [13732](#), [13732](#)
 _fp_sin_series_aux_o:NNnwww
 [14139](#), [14143](#), [14154](#)
 _fp_sin_series_o:NNwww ... [13738](#),
 [13753](#), [13768](#), [13783](#), [14139](#), [14139](#)
 _fp_small_int:wTF ... [9937](#), [9937](#), [10302](#)
 _fp_small_int_normal:NnwTF
 [9937](#), [9941](#), [9953](#)
 _fp_small_int_test:NnnwNTF [9937](#)

_fp_small_int_test:NnnwNnw 9957, 9964
 _fp_small_int_true:wTF
 9937, 9940, 9945, 9952, 9967
 _fp_sqrt_Newton_o:wnn
 .. 12431, 12442, 12443, 12443, 12450
 _fp_sqrt_auxi_o:NNNNwnnN
 12446, 12454, 12454
 _fp_sqrt_auxii_o:NnnnnnnnN
 .. 12456, 12460, 12460, 12540, 12552
 _fp_sqrt_auxiii_o:wnnnnnnnn
 12457, 12495, 12495, 12541
 _fp_sqrt_auxiv_o:NNNNNw
 12495, 12499, 12516
 _fp_sqrt_auxix_o:wnwnw
 12529, 12531, 12538
 _fp_sqrt_auxv_o:NNNNNw
 12495, 12503, 12518
 _fp_sqrt_auxvi_o:NNNNNw
 12495, 12507, 12520
 _fp_sqrt_auxvii_o:NNNNNw
 12495, 12510, 12522
 _fp_sqrt_auxviii_o:nnnnnnnn 12517,
 12519, 12521, 12527, 12529, 12529
 _fp_sqrt_auxx_o:Nnnnnnnn
 12525, 12543, 12543
 _fp_sqrt_auxxi_o:wnnnN
 12543, 12545, 12550
 _fp_sqrt_auxxii_o:nnnnnnnnnw
 12553, 12557, 12557
 _fp_sqrt_auxxiii_o:w 12557, 12564, 12577
 _fp_sqrt_auxxiv_o:wnnnnnnnN
 .. 12569, 12572, 12580, 12582, 12582
 _fp_sqrt_npos_auxi_o:wnnnN
 12422, 12428, 12433
 _fp_sqrt_npos_auxii_o:wNNNNNNN ..
 12422, 12437, 12441
 _fp_sqrt_npos_o:w . 12419, 12422, 12422
 _fp_sqrt_o:w 12412, 12412
 _fp_sub_back_far_o:NnnwnnnN
 12037, 12083, 12083
 _fp_sub_back_near_after:wNNNNw ..
 12043, 12045, 12052, 12120
 _fp_sub_back_near_o:nnnnnnnnN
 12033, 12043, 12043
 _fp_sub_back_near_pack:NNNNNw ..
 12043, 12047, 12050, 12122
 _fp_sub_back_not_far_o:wwwNN
 12097, 12117, 12117
 _fp_sub_back_quite_far_ii:NN
 12101, 12103, 12107
 _fp_sub_back_quite_far_o:wwNN
 12095, 12101, 12101
 _fp_sub_back_shift:wnnnn
 12055, 12059, 12059
 _fp_sub_back_shift_ii:ww
 12059, 12061, 12064
 _fp_sub_back_shift_iii:NNNNNNNNw .
 12059, 12069, 12072, 12081
 _fp_sub_back_shift_iv:nnnnw
 12059, 12076, 12082
 _fp_sub_back_very_far_ii_o:nnNwNN
 12129, 12132, 12136
 _fp_sub_back_very_far_o:wwwNN ...
 12096, 12129, 12129
 _fp_sub_eq_o:Nwnw 12008, 12011, 12019
 _fp_sub_npos_i_o:Nwnw
 12013, 12022, 12026, 12026
 _fp_sub_npos_ii_o:Nwnw
 12008, 12015, 12020
 _fp_sub_npos_o:NwnNw
 11928, 12008, 12008
 _fp_tan_o:w 13792, 13792
 _fp_tan_series_aux_o:Nnwvw
 14193, 14197, 14206
 _fp_tan_series_o:NNwww
 13799, 13814, 14193, 14193
 _fp_ternary:NwN .. 11364, 11785, 11785
 _fp_ternary_auxi:NwN
 11785, 11791, 11823
 _fp_ternary_auxii:NwN
 11380, 11785, 11813, 11831
 _fp_ternary_break_point:n
 11785, 11791, 11810, 11822
 _fp_ternary_loop:Nw
 11785, 11788, 11815, 11820
 _fp_ternary_loop_break:w
 11785, 11790, 11810
 _fp_ternary_map_break:
 11785, 11818, 11822
 _fp_tmp:w 9895, 9905, 9906,
 9907, 9908, 9909, 9910, 9911, 9912,
 9913, 9914, 9915, 9916, 9917, 9918,
 9919, 9920, 10402, 10414, 10415,
 10416, 10417, 10418, 10419, 10420,
 10951, 10968, 10969, 11009, 11014,
 11015, 11016, 11017, 11018, 11019,
 11020, 11028, 11029, 11030, 11031,
 11032, 11033, 11034, 11035, 11036,
 11037, 11038, 11263, 11281, 11282,
 11283, 11284, 11285, 11286, 11287

_fp_to_decimal_dispatch:w	_fp_trap_underflow_set_flag:
.. 14576, 14580, 14583, 14583, 14690 10101, 10111
_fp_to_decimal_huge:wnnnn	_fp_trap_underflow_set_none:
..... 14583, 14621, 14643 10101, 10113
_fp_to_decimal_large:Nnnw	_fp_trig:NNNNwn 13738, 13753, 13768,
..... 14583, 14617, 14634	13783, 13798, 13813, 13830, 13830
_fp_to_decimal_normal:wnnnn	_fp_trig_inverse_two_pi:
..... 14583, 14588, 14609, 14671 13891, 13891, 14060
_fp_to_int_dispatch:w	_fp_trig_large:ww . 13838, 14055, 14055
..... 14681, 14685, 14688, 14688	_fp_trig_large_auxi:wwwww
_fp_to_scientific_dispatch:w 14055, 14057, 14063
..... 14521, 14525, 14528, 14533	_fp_trig_large_auxii:ww
_fp_to_scientific_normal:wNw 14055, 14065, 14074
..... 14528, 14564, 14566, 14573	_fp_trig_large_auxiii:wNNNNNNNN ..
_fp_to_scientific_normal:wnnnn 14055, 14067, 14075
.. 14528, 14538, 14560, 14664, 14668	_fp_trig_large_auxiv:wN
_fp_to_tl_dispatch:w 14055, 14069, 14077
.. 14644, 14648, 14651, 14651, 14757	_fp_trig_large_auxix:Nw
_fp_to_tl_normal:nnnnn 14098, 14108, 14111, 14115
..... 14651, 14656, 14661	_fp_trig_large_auxv:www
_fp_trap_division_by_zero_set:N 14071, 14078, 14078
.. 10075, 10076, 10078, 10080, 10081	_fp_trig_large_auxvi:wnnnnnnnn ..
_fp_trap_division_by_zero_set_error: 14078, 14084, 14089
..... 10075, 10075	_fp_trig_large_auxvii:w
_fp_trap_division_by_zero_set_flag: 14081, 14098, 14098
..... 10075, 10077	_fp_trig_large_auxviii:w
_fp_trap_division_by_zero_set_none: 14100, 14104
..... 10075, 10079	_fp_trig_large_auxx:wNNNNN
_fp_trap_invalid_operation_set:N 14098, 14121, 14125
.. 10041, 10042, 10044, 10046, 10047	_fp_trig_large_auxxi:w
_fp_trap_invalid_operation_set_error: 14098, 14118, 14132
..... 10041, 10041	_fp_trig_large_pack:NNNNw
_fp_trap_invalid_operation_set_flag: 14078, 14091, 14096, 14127
..... 10041, 10043	_fp_trig_small:ww
_fp_trap_invalid_operation_set_none:	.. 13840, 13844, 13844, 13850, 14137
..... 10041, 10045	_fp_trigd_large:ww 13838, 13852, 13852
_fp_trap_overflow_set:N	_fp_trigd_large_auxi:nnnnwNNNN ..
.. 10101, 10102, 10104, 10106, 10107 13852, 13858, 13864
_fp_trap_overflow_set:NnNw	_fp_trigd_large_auxii:wNw
..... 10101, 10108, 10116, 10117 13852, 13866, 13872
_fp_trap_overflow_set_error:	_fp_trigd_large_auxiii:www
..... 10101, 10101 13852, 13875, 13879
_fp_trap_overflow_set_flag:	_fp_trigd_small:ww
..... 10101, 10103 13840, 13846, 13846, 13889
_fp_trap_overflow_set_none:	_fp_trim_zeros:w
..... 10101, 10105	.. 14512, 14512, 14574, 14627, 14636
_fp_trap_underflow_set:N	_fp_trim_zeros_dot:w 14512, 14515, 14518
.. 10101, 10110, 10112, 10114, 10115	_fp_trim_zeros_end:w 14512, 14518, 14519
_fp_trap_underflow_set_error:	_fp_trim_zeros_loop:w
..... 10101, 10109 14512, 14514, 14515, 14517

_fp_type_from_scan:N	11965, 11967, 11988, 11990, 11999,
..... 9856, 10387, 10395, 10445	12001, 12023, 12030, 12036, 12046,
_fp_type_from_scan:w 10387, 10397, 10400	12048, 12121, 12123, 12139, 12141,
_fp_underflow:w ... 9794, 10132, 10138	12145, 12161, 12201, 12209, 12211,
_fp_use_i:ww .. 9742, 9742, 12836, 14441	12213, 12215, 12217, 12219, 12221,
_fp_use_i:www 9742, 9743	12240, 12242, 12252, 12254, 12280,
_fp_use_i_until_s:nw .. 9737, 9738,	12283, 12291, 12293, 12313, 12316,
9782, 9992, 13882, 14080, 14086, 14117	12319, 12322, 12330, 12333, 12336,
_fp_use_ii_until_s:nnw 9737, 9739, 9780	12339, 12346, 12348, 12354, 12362,
_fp_use_none_stop_f:n	12364, 12366, 12372, 12392, 12394,
.... 9734, 9734, 13000, 13001, 13002	12403, 12405, 12426, 12447, 12451,
_fp_use_none_until_s:w	12463, 12466, 12469, 12472, 12475,
9737, 9737, 12448, 13671, 14436, 14439	12478, 12481, 12484, 12488, 12500,
_fp_use_s:n 9735, 9735	12504, 12508, 12511, 12532, 12534,
_fp_use_s:nn 9735, 9736	12536, 12546, 12585, 12587, 12596,
_fp_zero_fp:N . 9763, 9763, 10116, 10367	12618, 12623, 12625, 12632, 12635,
_int_abs:N 3042, 3044, 3048	12638, 12641, 12644, 12647, 12656,
_int_case:nnTF	12668, 12676, 12678, 12688, 12690,
.. 3269, 3272, 3277, 3282, 3287, 3289	12697, 12706, 12708, 12711, 12714,
_int_case:nw ... 3269, 3290, 3291, 3295	12717, 12720, 12733, 12735, 12743,
_int_case_end:nw 3269, 3294, 3297	12745, 12753, 12755, 12764, 12767,
_int_compare:NNw 3208, 3220, 3224	12770, 12777, 12792, 12810, 12813,
_int_compare:Nw 3208, 3216, 3218, 3245	12869, 12883, 12885, 12891, 12904,
_int_compare:nnN .. 3208, 3240, 3248,	12906, 12908, 12932, 12948, 12955,
3250, 3252, 3254, 3256, 3258, 3260	12956, 12979, 12995, 12999, 13044,
_int_compare:w 3208, 3210, 3213	13046, 13084, 13095, 13114, 13116,
__int_compare__	13118, 13131, 13144, 13149, 13151,
_int_compare_efill	13157, 13174, 13175, 13176, 13177,
:NNw 3208	13178, 13179, 13184, 13186, 13188,
_int_compare_<:NNw 3208	13190, 13192, 13196, 13198, 13200,
_int_compare_=:NNw 3208	13202, 13204, 13206, 13228, 13236,
_int_constdef:Nw 3114, 3125, 3137, 3141	13314, 13318, 13371, 13580, 13601,
_int_div_truncate:NwNw	13603, 13606, 13609, 13612, 13615,
..... 3074, 3077, 3082, 3105	13631, 13657, 13667, 13683, 13835,
_int_eval:w ... 74, 1234, 2147, 2358,	13867, 13876, 14058, 14059, 14082,
2360, 2362, 2428, 2430, 2432, 2434,	14092, 14101, 14119, 14128, 14135,
2436, 2438, 2440, 2442, 2444, 2446,	14146, 14156, 14189, 14199, 14224,
2448, 2450, 3031, 3032, 3037, 3040,	14233, 14250, 14286, 14303, 14305,
3045, 3053, 3054, 3061, 3062, 3076,	14317, 14318, 14358, 14369, 14380,
3078, 3079, 3096, 3099, 3100, 3101,	14438, 14568, 14706, 15733, 15824
3126, 3163, 3165, 3187, 3211, 3245,	_int_eval_end: 74, 1234, 2147, 2358,
3263, 3300, 3308, 3373, 3374, 3375,	2360, 2362, 2428, 2430, 2432, 2434,
3569, 3596, 3602, 3629, 3786, 3830,	2436, 2438, 2440, 2442, 2444, 2446,
5224, 9692, 9787, 9883, 9886, 9980,	2448, 2450, 3031, 3033, 3037, 3040,
10227, 10231, 10243, 10244, 10279,	3045, 3080, 3096, 3102, 3126, 3163,
10328, 10332, 10371, 10510, 10515,	3165, 3187, 3263, 3300, 3308, 3569,
10550, 10640, 10651, 10700, 10731,	3596, 3602, 3629, 5224, 9695, 9787,
10737, 10738, 10784, 10794, 10796,	9983, 10279, 10342, 10346, 11438,
10812, 10814, 10837, 10839, 10976,	11608, 11858, 12023, 12145, 12180,
11175, 11438, 11608, 11936, 11944,	12368, 12678, 12813, 13631, 13683,

13868, 13877, 14146, 14156, 14199,
 14224, 14250, 14318, 15733, 15824
 _int_from_alph:N 3710, 3726, 3729
 _int_from_alph:n 3710, 3715, 3718
 _int_from_alph:nN 3710, 3719, 3720, 3725
 _int_from_base:N 3731, 3748, 3752
 _int_from_base:nn 3731, 3736, 3740
 _int_from_base:nnN 3731, 3741, 3742, 3747
 _int_from_roman:NN
 3781, 3787, 3790, 3815, 3819
 _int_from_roman_clean_up:w
 3781, 3798, 3805, 3807, 3826
 _int_from_roman_end:w 3781, 3785, 3824
 _int_get_digits:n 3676, 3681, 3715, 3737
 _int_get_sign:n 3676, 3676, 3714, 3735
 _int_get_sign_and_digits:nnnn
 3676, 3678, 3683, 3686, 3709
 _int_get_sign_and_digits:nnnn
 3676, 3692, 3696, 3702
 _int_maxmin:wwN 3042, 3052, 3060, 3066
 _int_mod:ww 3074, 3099, 3104
 _int_step:NNnnnn 3400, 3403, 3410, 3419
 _int_step:NnnnN
 3370, 3380, 3387, 3391, 3396
 _int_step:wwwN 3370, 3372, 3377
 _int_to_Base:nn 3510, 3513, 3520
 _int_to_Base:nnN
 3510, 3523, 3524, 3546, 3560
 _int_to_Base:nnnN 3510, 3551, 3558
 _int_to_Letter:n 3510, 3549, 3552, 3599
 _int_to_Roman_Q:w 3640, 3675
 _int_to_Roman_aux:N .. 3652, 3655, 3658
 _int_to_Roman_c:w 3640, 3672
 _int_to_Roman_d:w 3640, 3673
 _int_to_Roman_i:w 3640, 3668
 _int_to_Roman_l:w 3640, 3671
 _int_to_Roman_m:w 3640, 3674
 _int_to_Roman_v:w 3640, 3669
 _int_to_Roman_x:w 3640, 3670
 _int_to_base:nn 3510, 3511, 3514
 _int_to_base:nnN
 3510, 3517, 3518, 3526, 3540
 _int_to_base:nnnN 3510, 3531, 3538
 _int_to_letter:n 3510, 3529, 3532, 3566
 _int_to_roman:N
 3640, 3640, 3642, 3645, 3648
 _int_to_roman:w . 73, 715, 716, 816,
 818, 991, 2145, 3031, 3221, 3643, 3653
 _int_to_roman_Q:w 3640, 3667
 _int_to_roman_c:w 3640, 3664
 _int_to_roman_d:w 3640, 3665
 _int_to_roman_i:w 3640, 3660
 _int_to_roman_l:w 3640, 3663
 _int_to_roman_m:w 3640, 3666
 _int_to_roman_v:w 3640, 3661
 _int_to_roman_x:w 3640, 3662
 _int_to_symbols:nnnn . 3426, 3430, 3440
 _int_value:w 74, 997,
 2059, 2062, 2147, 3031, 3031, 3037,
 3040, 3044, 3052, 3060, 3096, 3099,
 3100, 3101, 3215, 3245, 3596, 3629,
 3786, 3948, 6816, 6847, 6848, 6849,
 6851, 6977, 6986, 6988, 6993, 6994,
 6995, 6996, 7000, 7001, 7002, 7003,
 7028, 7034, 7036, 7038, 7040, 7045,
 7050, 7055, 7062, 7069, 7199, 7229,
 7230, 7269, 7288, 7294, 7464, 7568,
 9692, 9845, 9846, 9847, 9848, 9849,
 9900, 9945, 9968, 10314, 10330,
 10335, 10469, 10501, 10502, 10635,
 10642, 10655, 10692, 10703, 10709,
 10720, 10741, 10851, 10928, 10930,
 11045, 11532, 11623, 11881, 11882,
 11883, 11885, 11939, 11942, 12005,
 12062, 12070, 12078, 12144, 12147,
 12246, 12258, 12296, 12297, 12298,
 12299, 12409, 12570, 12573, 12593,
 12605, 13001, 13002, 13003, 13077,
 13091, 13112, 13135, 13136, 13137,
 13138, 13139, 13252, 13257, 13261,
 13314, 13383, 13565, 13622, 13625,
 13627, 13653, 13655, 14060, 14145,
 14198, 14636, 14698, 14708, 14710,
 15210, 15212, 15235, 15237, 15262,
 15302, 15329, 15337, 15363, 15365,
 15369, 15371, 15400, 15414, 15421
 _ior_list_streams:Nn
 9413, 9414, 9415, 9508
 _ior_map_inline:NNn 15459, 15466, 15469
 _ior_map_inline:NNn
 15459, 15460, 15462, 15463
 _ior_map_inline_loop:NNN
 15459, 15472, 15476, 15482
 _ior_open:Nn
 ... 175, 9225, 9245, 9378, 9378, 9394
 _ior_open:No 9361, 9374, 9378
 _ior_open_aux:Nn 9353, 9354, 9356
 _ior_open_aux:NnTF ... 9363, 9364, 9368
 _ior_open_stream:Nn
 9378, 9382, 9390, 9395

```

__iow_indent:n ..... 9558, 9559, 9577
__iow_list_streams:Nn . 9506, 9507, 9508
__iow_open:Nn ... 9468, 9469, 9471, 9487
__iow_open_stream:Nn .....
..... 9468, 9475, 9483, 9488
__iow_wrap_end: ..... 9682
__iow_wrap_end:w ..... 9655
__iow_wrap_indent: ..... 9670
__iow_wrap_indent:w ..... 9655
__iow_wrap_loop:p .....
..... 9598, 9610, 9610, 9625, 9660
__iow_wrap_newline: ..... 9662
__iow_wrap_newline:w ..... 9655
__iow_wrap_set:Nx .... 9565, 9583, 9608
__iow_wrap_special:w .....
..... 9614, 9655, 9655, 9659
__iow_wrap_unindent: ..... 9676
__iow_wrap_unindent:w ..... 9655
__iow_wrap_word: .... 9615, 9617, 9617
__iow_wrap_word_fits: . 9617, 9623, 9627
__iow_wrap_word_newline: 9617, 9624, 9643
__kernel_primitive:NN .....
. 201, 201, 210, 211, 212, 213, 214,
215, 216, 217, 218, 219, 220, 221,
222, 223, 224, 225, 226, 227, 228,
229, 230, 231, 232, 233, 234, 235,
236, 237, 238, 239, 240, 241, 242,
243, 244, 245, 246, 247, 248, 249,
250, 251, 252, 253, 254, 255, 256,
257, 258, 259, 260, 261, 262, 263,
264, 265, 266, 267, 268, 269, 270,
271, 272, 273, 274, 275, 276, 277,
278, 279, 280, 281, 282, 283, 284,
285, 286, 287, 288, 289, 290, 291,
292, 293, 294, 295, 296, 297, 298,
299, 300, 301, 302, 303, 304, 305,
306, 307, 308, 309, 310, 311, 312,
313, 314, 315, 316, 317, 318, 319,
320, 321, 322, 323, 324, 325, 326,
327, 328, 329, 330, 331, 332, 333,
334, 335, 336, 337, 338, 339, 340,
341, 342, 343, 344, 345, 346, 347,
348, 349, 350, 351, 352, 353, 354,
355, 356, 357, 358, 359, 360, 361,
362, 363, 364, 365, 366, 367, 368,
369, 370, 371, 372, 373, 374, 375,
376, 377, 378, 379, 380, 381, 382,
383, 384, 385, 386, 387, 388, 389,
390, 391, 392, 393, 394, 395, 396,
397, 398, 399, 400, 401, 402, 403,
404, 405, 406, 407, 408, 409, 410,
411, 412, 413, 414, 415, 416, 417,
418, 419, 420, 421, 422, 423, 424,
425, 426, 427, 428, 429, 430, 431,
432, 433, 434, 435, 436, 437, 438,
439, 440, 441, 442, 443, 444, 445,
446, 447, 448, 449, 450, 451, 452,
453, 454, 455, 456, 457, 458, 459,
460, 461, 462, 463, 464, 465, 466,
467, 468, 469, 470, 471, 472, 473,
474, 475, 476, 477, 478, 479, 480,
481, 482, 483, 484, 485, 486, 487,
488, 489, 490, 491, 492, 493, 494,
495, 496, 497, 498, 499, 500, 501,
502, 503, 504, 505, 506, 507, 508,
509, 510, 511, 512, 513, 514, 515,
516, 517, 518, 519, 520, 521, 522,
523, 524, 525, 526, 527, 528, 529,
530, 531, 532, 533, 534, 535, 536,
537, 538, 539, 540, 541, 542, 543,
544, 545, 546, 547, 548, 549, 550,
551, 552, 553, 554, 555, 556, 557,
558, 559, 560, 561, 562, 563, 564,
565, 566, 567, 568, 569, 570, 571,
572, 573, 574, 575, 576, 577, 578,
579, 580, 581, 582, 583, 584, 585,
586, 587, 588, 589, 590, 591, 592,
593, 594, 595, 596, 597, 598, 599,
600, 601, 602, 603, 604, 605, 606,
607, 608, 609, 610, 611, 612, 613,
614, 615, 616, 617, 618, 619, 620,
621, 622, 623, 624, 625, 626, 627,
628, 629, 630, 631, 632, 633, 634, 635
__kernel_register_show:N . 25, 1382,
1382, 1392, 3827, 4104, 4194, 4256
__kernel_register_show:c 1382, 1391, 3828
__keys_bool_set:Nn .....
..... 8559, 8559, 8574, 8699, 8703
__keys_bool_set:cn .... 8559, 8701, 8705
__keys_bool_set_inverse:Nn .....
..... 8575, 8575, 8590, 8707, 8711
__keys_bool_set_inverse:cn .....
..... 8575, 8709, 8713
__keys_check_groups: ..... 8924, 8937
__keys_choice_code_store:n .....
..... 9097, 9097, 9108, 9110
__keys_choice_code_store:x . 9097, 9112
__keys_choice_find:n .....
..... 8592, 9015, 9015, 9021

```

_keys_choice_make:	8562,	_keys_parent:wn	8591, 8621, 8623, 8628
8578, 8591, 8591, 8632, 8715, 9118		_keys_property_find:n	8511, 8522, 8522
_keys_choice_make:N		_keys_property_find:w	
8591, 8592, 8594, 8595		8522, 8526, 8532, 8538	
_keys_choice_make_aux:N		_keys_set:nnn	8818, 8820, 8827
8591, 8605, 8607, 8609		_keys_set:onn	8818, 8819
_keys_choices_generate:n		_keys_set_elt:n	8823, 8875, 8875
9097, 9113, 9147		_keys_set_elt:nn	8823, 8875, 8880
_keys_choices_generate_aux:n		_keys_set_elt_aux:	8875,
9097, 9120, 9127		8892, 8894, 8927, 8933, 8955, 8959	
_keys_choices_make:Nnn		_keys_set_elt_aux:nn	
8631, 8632, 8634, 8635		8875, 8878, 8883, 8885	
_keys_choices_make:nn		_keys_set_elt_selective:	
8631, 8631, 8717, 8719, 8721, 8723		8875, 8891, 8918	
_keys_cmd_set:Vn	8651, 8681	_keys_set_filter:nnnnN	8846, 8849, 8856
_keys_cmd_set:Vo	8651, 8677	_keys_set_filter:onnN	8846, 8847
_keys_cmd_set:nn	8567, 8583,	_keys_set_known:nnnN	8828, 8831, 8838
8611, 8614, 8651, 8651, 8656, 8725		_keys_set_known:onnN	8828, 8829
_keys_cmd_set:nx	8563, 8565,	_keys_store_unused:	8928,
8579, 8581, 8642, 8651, 8694, 9130		8934, 8954, 8960, 8983, 8988, 9006	
_keys_default_set:n	8572, 8588,	_keys_value_or_default:n	
8657, 8657, 8735, 8737, 8739, 8741		8889, 8963, 8963	
_keys_define:nnn	8490, 8492, 8498	_keys_value_requirement:n	
_keys_define:onn	8490, 8491	8682, 8682, 8815, 8817	
_keys_define_elt:n	8495, 8499, 8499	_keys_variable_set:NnnN	
_keys_define_elt:nn	8495, 8499, 8504	8691, 8691, 8697, 8727, 8731,	
_keys_define_elt_aux:nn		8743, 8747, 8751, 8755, 8769, 8773,	
8499, 8502, 8507, 8509		8791, 8795, 8799, 8803, 8807, 8811	
_keys_define_key:n	8513, 8542, 8542	_keys_variable_set:cnnN	
_keys_define_key:w	8542, 8546, 8557	8691, 8729, 8733,	
_keys_execute:	8915, 8983, 8983	8745, 8749, 8753, 8757, 8771, 8775,	
_keys_execute:nn		8793, 8797, 8801, 8805, 8809, 8813	
8983, 8984, 8990, 8997, 9017, 9018		_keyval_parse:n	8385, 8393, 8462
_keys_execute_unknown: 8983, 8984, 8985		_keyval_parse_elt:w	
_keys_groups_set:n	8662, 8662, 8759	8400, 8406, 8406, 8413	
_keys_if_value:n	8973	_keyval_split:Nn	
_keys_if_value_p:n	8898, 8908, 8973	8428, 8435, 8436, 8436, 8446	
_keys_initialise:n		_keyval_split:Nw	8436, 8440, 8442
8671, 8671, 8761, 8763, 8765, 8767		_keyval_split_key:w	8415, 8419, 8434
_keys_initialise:wn	8671, 8672, 8673	_keyval_split_key_value:w	
_keys_meta_make:n	8675, 8675, 8777	8411, 8415, 8415	
_keys_meta_make:nn	8675, 8680, 8779	_keyval_split_value:w	8430, 8444, 8444
_keys_multichoice_find:n		_msg_class_chk_exist:nT	
8594, 9015, 9020		7912, 7912, 7926, 7992, 8002, 8007	
_keys_multichoice_make:		_msg_class_new:nn	7807, 7808, 7845,
8591, 8593, 8634, 8781		7856, 7867, 7888, 7896, 7904, 7910	
_keys_multichoices_make:nn		_msg_error:cnnnnn	7867, 7869, 7881
8631, 8633, 8783, 8785, 8787, 8789		_msg_error_code:nnnnnn	8093
_keys_parent:n	8591, 8620, 8622	_msg_expandable_error:n	
_keys_parent:o	8591, 8597, 8599, 8603	154, 8275, 8283, 8300	

_msg_expandable_error:w	8275 , 8289 , 8295	_msg_kernel_fatal:nnxx	8089
_msg_fatal_code:nnnnnn	8089	_msg_kernel_fatal:nnxxx	8089
_msg_interrupt_more_text:n	7735 , 7737 , 7740	_msg_kernel_fatal:nnxxxx	8089
_msg_interrupt_text:n	7738 , 7749 , 7757	_msg_kernel_info:nn	8094
_msg_interrupt_wrap:nn	7727 , 7731 , 7735 , 7735	_msg_kernel_info:nnn	8094
_msg_kernel_class_new:nN	8051 , 8052 , 8089 , 8093 , 8094 , 8095	_msg_kernel_info:nnnn	8094
_msg_kernel_class_new_aux:nN	8051 , 8053 , 8054	_msg_kernel_info:nnnnn	8094
_msg_kernel_error:nn	1097 , 1111 , 7087 , 8089 , 8092 , 8431	_msg_kernel_info:nnnnnn	153 , 8094
_msg_kernel_error:nnn	8089	_msg_kernel_info:nnx	8094
_msg_kernel_error:nnnn	8089	_msg_kernel_info:nnxx	8094
_msg_kernel_error:nnnnn	8089	_msg_kernel_info:nnxxx	8094
_msg_kernel_error:nnnnnn	153 , 8089	_msg_kernel_info:nnxxxx	8094
_msg_kernel_error:nnx	860 , 901 , 939 , 944 , 1097 , 1109 , 1144 , 1154 , 1305 , 1387 , 1758 , 1999 , 2339 , 4521 , 5070 , 6690 , 6831 , 7915 , 8089 , 8091 , 8348 , 8528 , 8569 , 8585 , 8902 , 9123 , 9207 , 9275 , 9360 , 9747 , 10038 , 14805	_msg_kernel_new:nnn	7592 , 8043 , 8045 , 8237 , 8239 , 8241 , 8243 , 8245 , 8247 , 8249 , 8257 , 8264 , 8271 , 8273 , 10172 , 10174 , 10176 , 10178 , 10180 , 10182 , 11567 , 11569 , 11571 , 11573 , 11575 , 11577 , 11579 , 11581 , 11583
_msg_kernel_error:nnxx	851 , 880 , 958 , 1097 , 1097 , 1110 , 1112 , 1119 , 1129 , 1275 , 1819 , 6979 , 7642 , 7652 , 7940 , 8089 , 8090 , 8517 , 8550 , 8602 , 8616 , 8912 , 8992 , 10035	_msg_kernel_new:nnnn	152 , 7574 , 7582 , 7585 , 8043 , 8043 , 8097 , 8105 , 8113 , 8120 , 8131 , 8139 , 8148 , 8155 , 8162 , 8169 , 8178 , 8185 , 8195 , 8207 , 8214 , 8221 , 8229 , 8465 , 9036 , 9039 , 9045 , 9050 , 9059 , 9065 , 9071 , 9078 , 9085 , 9091 , 9139 , 9706 , 9712 , 9719 , 9726 , 10000 , 10149 , 10161
_msg_kernel_error:nnxxx	8089	_msg_kernel_set:nnn	8043 , 8049
_msg_kernel_error:nnxxxx	1834 , 8089	_msg_kernel_set:nnnn	152 , 8043 , 8047
_msg_kernel_expandable_error:nn	2175 , 5245 , 6298 , 8298 , 8322 , 10454 , 11250	_msg_kernel_warning:nn	8094
_msg_kernel_expandable_error:nnn	1513 , 3204 , 3384 , 4755 , 5705 , 6206 , 8298 , 8317 , 10459 , 10536 , 10575 , 10581 , 10917 , 10922 , 10933 , 10940 , 11003 , 11195 , 11215	_msg_kernel_warning:nnn	8094
_msg_kernel_expandable_error:nnnn	8298 , 8312 , 11377 , 11418 , 11800	_msg_kernel_warning:nnnn	8094
_msg_kernel_expandable_error:nnnnn	8298 , 8307 , 10147 , 11545 , 14255	_msg_kernel_warning:nnnnn	153 , 8094
_msg_kernel_expandable_error:nnnnnn	153 , 8298 , 8298 , 8309 , 8314 , 8319 , 8324	_msg_kernel_warning:nnxxx	8094
_msg_kernel_fatal:nn	8089 , 9384 , 9477	_msg_kernel_warning:nnxxx	8094
_msg_kernel_fatal:nnn	8089	_msg_kernel_warning:nnxxxx	8027 , 8094
_msg_kernel_fatal:nnnn	8089	_msg_no_more_text:nnnn	7867 , 7883 , 7887
_msg_kernel_fatal:nnnnn	153 , 8089	_msg_redirect:nnn	7996 , 7997 , 7999 , 8000
_msg_kernel_fatal:nnnnnn	153 , 8089	_msg_redirect_loop_chk:nnn	7996 , 8012 , 8017 , 8041
_msg_kernel_fatal:nnx	8089	_msg_redirect_loop_chk:onn	8037
		_msg_redirect_loop_list:n	7996 , 8033 , 8042
		_msg_show_item:n	154 , 5757 , 6281 , 8363 , 8363
		_msg_show_item:nn	6559 , 8363 , 8367
		_msg_show_item_unbraced:nn	154 , 7569 , 8363 , 8372 , 9421

- _msg_show_variable:Nnn
154, 5756, 6280, 6558, 7565, 8340, 8340
- _msg_show_variable:n
154, 1401, 2006, 2007, 5076,
8340, 8345, 8352, 9420, 14803, 14810
- _msg_show_variable_aux:n
8340, 8353, 8354
- _msg_show_variable_aux:w
8340, 8358, 8362
- _msg_term:nn 8327, 8338
- _msg_term:nnn .. 8327, 8336, 8344, 9417
- _msg_term:nnnn 8327, 8334
- _msg_term:nnnnV 8327
- _msg_term:nnnnn
154, 8327, 8327, 8333, 8335, 8337, 8339
- _msg_use:nnnnnn 7816, 7922, 7922
- _msg_use_code:
7922, 7929, 7942, 7946, 7971, 7982
- _msg_use_hierarchy:nwN
7922, 7949, 7950, 7956
- _msg_use_redirect_module:n
7922, 7953, 7961, 7974
- _msg_use_redirect_name:n
7922, 7937, 7943
- _my_map_dbl:nn 4, 6, 11
- _my_map_dbl_fn:nn 3, 10
- _peek_N_type:w ... 15995, 16010, 16020
- _peek_N_type_aux:nnw 15995, 16012, 16024
- _peek_def:nnnn 2931, 2932,
2948, 2952, 2956, 2960, 2964, 2968,
2972, 2976, 2980, 2984, 2988, 2992
- _peek_def:nnnnn
2931, 2934, 2935, 2936, 2938
- _peek_execute_branches: ... 2928, 2943
- _peek_execute_branches_N_type: ...
15995, 16003, 16032, 16034, 16036
- _peek_execute_branches_catcode: ...
2888, 2888, 2951, 2953, 2959, 2961
- _peek_execute_branches_catcode_aux:
2888, 2889, 2891, 2892
- _peek_execute_branches_catcode_auxii:N
2888, 2896, 2902
- _peek_execute_branches_catcode_auxiii:
2888, 2899, 2912
- _peek_execute_branches_charcode: ...
2888, 2890, 2967, 2969, 2975, 2977
- _peek_execute_branches_meaning: ...
2880, 2880, 2983, 2985, 2991, 2993
- _peek_false:w 2827, 2829, 2850,
2868, 2885, 2908, 2918, 16017, 16029
- _peek_get_prefix_arg_replacement:wN
2997, 2998, 3005, 3014, 3023
- _peek_ignore_spaces_execute_branches:
2921, 2921, 2925,
2955, 2963, 2971, 2979, 2987, 2995
- _peek_tmp:w 2827, 2830, 2839
- _peek_token_generic:NNF .. 2860, 16036
- _peek_token_generic:NNT .. 2858, 16034
- _peek_token_generic:NNTF
2841, 2841, 2859, 2861, 16032
- _peek_token_remove_generic:NNF . 2878
- _peek_token_remove_generic:NNT . 2876
- _peek_token_remove_generic:NNTF ..
2862, 2862, 2877, 2879
- _peek_true:w 2827, 2827, 2845, 2866,
2883, 2906, 2916, 16015, 16028, 16029
- _peek_true_aux:w 2827, 2828, 2838, 2867
- _peek_true_remove:w .. 2835, 2835, 2866
- _prg_break: 42, 1456, 1457, 2235, 5058,
5606, 6496, 9981, 14744, 15508, 15514
- _prg_break:n 1456,
1458, 2235, 5060, 5470, 5613, 6397
- _prg_break_point: 42, 1456,
1456, 1457, 1458, 2235, 5054, 5467,
5607, 6392, 6481, 9982, 14745, 15509
- _prg_break_point:Nn 42, 1447,
1447, 1448, 2235, 3423, 4707, 4724,
4733, 5628, 5663, 5674, 6092, 6106,
6125, 6143, 6525, 6544, 15473, 15491
- _prg_case_end:nw
25, 3297, 4018, 4668, 4700, 4702, 5210
- _prg_compare_error: 74, 3193,
3193, 3197, 3211, 3213, 3957, 3959
- _prg_compare_error:Nw 3193, 3199, 3233
- _prg_generate_F_form:wnnnnnn 889, 910
- _prg_generate_TF_form:wnnnnnn 889, 915
- _prg_generate_T_form:wnnnnnn 889, 905
- _prg_generate_conditional:nnNnnnnn
829, 848, 857, 857
- _prg_generate_conditional:nnnnnnw
857, 866, 872, 887
- _prg_generate_conditional_count:nnNnn
832, 833, 835, 837, 839, 840
- _prg_generate_conditional_count:nnNnnnn
832, 842, 845
- _prg_generate_conditional_parm:nnNpnn
819, 820, 822, 824, 826, 827
- _prg_generate_p_form:wnnnnnn 889, 889

- _prg_map_break:Nn . . . [42](#), [1447](#), [1448](#),
[1454](#), [2235](#), [4746](#), [4748](#), [5618](#), [5620](#),
[6160](#), [6162](#), [6553](#), [6555](#), [15456](#), [15458](#)
- _prg_replicate:N [2143](#), [2150](#), [2151](#), [2153](#)
- _prg_replicate_ [2143](#), [2154](#)
- _prg_replicate_0:n [2143](#)
- _prg_replicate_1:n [2143](#)
- _prg_replicate_2:n [2143](#)
- _prg_replicate_3:n [2143](#)
- _prg_replicate_4:n [2143](#)
- _prg_replicate_5:n [2143](#)
- _prg_replicate_6:n [2143](#)
- _prg_replicate_7:n [2143](#)
- _prg_replicate_8:n [2143](#)
- _prg_replicate_9:n [2143](#)
- _prg_replicate_first:N [2143](#), [2146](#), [2152](#)
- _prg_replicate_first_-:n [2143](#)
- _prg_replicate_first_0:n [2143](#)
- _prg_replicate_first_1:n [2143](#)
- _prg_replicate_first_2:n [2143](#)
- _prg_replicate_first_3:n [2143](#)
- _prg_replicate_first_4:n [2143](#)
- _prg_replicate_first_5:n [2143](#)
- _prg_replicate_first_6:n [2143](#)
- _prg_replicate_first_7:n [2143](#)
- _prg_replicate_first_8:n [2143](#)
- _prg_replicate_first_9:n [2143](#)
- _prg_set_eq_conditional:NNNn
. [920](#), [921](#), [923](#), [924](#)
- _prg_set_eq_conditional:nnNnnNNw .
. [928](#), [936](#), [936](#)
- _prg_set_eq_conditional_F_form:nnn
. [936](#)
- _prg_set_eq_conditional_F_form:wNnnnn
. [982](#)
- _prg_set_eq_conditional_TF_form:nnn
. [936](#)
- _prg_set_eq_conditional_TF_form:wNnnnn
. [972](#)
- _prg_set_eq_conditional_T_form:nnn
. [936](#)
- _prg_set_eq_conditional_T_form:wNnnnn
. [977](#)
- _prg_set_eq_conditional_loop:nnnnNw
. [936](#), [948](#), [950](#), [965](#)
- _prg_set_eq_conditional_p_form:nnn
. [936](#)
- _prg_set_eq_conditional_p_form:wNnnnn
. [967](#)
- _prg_variable_get_scope:N
. [41](#), [2201](#), [2207](#)
- _prg_variable_get_scope:w
. [2201](#), [2210](#), [2213](#)
- _prg_variable_get_type:N [41](#), [2201](#), [2222](#)
- _prg_variable_get_type:w
. [2201](#), [2224](#), [2227](#), [2231](#)
- _prop_if_in:N [6476](#), [6486](#), [6489](#)
- _prop_if_in:nwn [6476](#), [6478](#), [6483](#), [6487](#)
- _prop_item_Nn:nwn [6388](#), [6390](#), [6394](#), [6398](#)
- _prop_map_function:Nwn
. [6521](#), [6523](#), [6527](#), [6533](#)
- _prop_map_tokens:nwn
. [15487](#), [15489](#), [15493](#), [15499](#)
- _prop_pair:wn
[132](#), [6297](#), [6297](#), [6336](#), [6339](#), [6391](#),
[6394](#), [6433](#), [6456](#), [6479](#), [6483](#), [6524](#),
[6527](#), [6540](#), [6542](#), [6547](#), [15490](#), [15493](#)
- _prop_put:NNnn . [6427](#), [6427](#), [6428](#), [6429](#)
- _prop_put_if_new:NNnn
. [6448](#), [6449](#), [6451](#), [6452](#)
- _prop_split:NnTF [132](#),
[6331](#), [6331](#), [6344](#), [6350](#), [6360](#), [6368](#),
[6377](#), [6403](#), [6413](#), [6436](#), [6459](#), [6508](#)
- _prop_split_aux:NnTF . [6331](#), [6332](#), [6333](#)
- _prop_split_aux:w [6331](#), [6335](#), [6338](#), [6341](#)
- _quark_if_recursion_tail_break:NN
. [46](#), [2276](#), [2276](#), [2351](#), [4740](#)
- _quark_if_recursion_tail_break:nN
. [2276](#),
[2282](#), [2353](#), [4713](#), [5058](#), [6097](#), [6110](#)
- _scan_new:N [47](#), [2335](#),
[2335](#), [2347](#), [2349](#), [6296](#), [9744](#), [9750](#),
[9751](#), [9752](#), [9753](#), [9754](#), [9755](#), [9756](#)
- _seq_count:n [5678](#), [5683](#), [5686](#)
- _seq_get_left:wnw [5495](#), [5499](#), [5503](#)
- _seq_get_right_loop:nn
. [5520](#), [5523](#), [5529](#), [5532](#)
- _seq_if_in: [5452](#), [5461](#), [5469](#)
- _seq_item:n [116](#), [5243](#), [5243](#), [5353](#),
[5361](#), [5371](#), [5373](#), [5378](#), [5428](#), [5429](#),
[5431](#), [5436](#), [5457](#), [5500](#), [5503](#), [5513](#),
[5541](#), [5542](#), [5553](#), [5640](#), [5645](#), [5651](#),
[5655](#), [5699](#), [5711](#), [5714](#), [5717](#), [15542](#)
- _seq_item:nnn [5592](#), [5596](#), [5609](#), [5614](#)
- _seq_item:wNn [5592](#), [5593](#), [5594](#)
- _seq_map_function:NNn
. [5621](#), [5624](#), [5630](#), [5634](#)
- _seq_mapthread_function:Nnnwnn . . .
. [15503](#), [15513](#), [15517](#), [15522](#)

- _seq_mapthread_function:wNN 15503, 15504, 15505
- _seq_mapthread_function:wNw 15503, 15507, 15511
- _seq_pop:NNNN 5477, 5477, 5507, 5509, 5536, 5538
- _seq_pop_TF:NNNN 5477, 5485, 5563, 5565, 5573, 5575, 5577, 5579
- _seq_pop_item_def: 116, 5418, 5637, 5653, 5663, 5674, 15534, 15544
- _seq_pop_left:NNN 5506, 5507, 5509, 5510, 5573, 5575
- _seq_pop_left:wnwNNN . 5506, 5511, 5512
- _seq_pop_right:NNN 5535, 5536, 5538, 5539, 5577, 5579
- _seq_pop_right_loop:nn 5535, 5546, 5555, 5558
- _seq_push_item_def: 5637, 5639, 5644, 5647
- _seq_push_item_def:n 116, 5402, 5637, 5637, 5661, 15532, 15542
- _seq_push_item_def:x . 5637, 5642, 5668
- _seq_put_left_aux:w 5349, 5354, 5362, 5365
- _seq_remove_all_aux:NNn 5396, 5397, 5399, 5400
- _seq_remove_duplicates:NN 5380, 5381, 5383, 5384
- _seq_reverse:NN 5422, 5423, 5425, 5426
- _seq_reverse_item:nwn 5422, 5429, 5433
- _seq_set_filter:NNNn 15526, 15527, 15529, 15530
- _seq_set_map:NNNn 15536, 15537, 15539, 15540
- _seq_set_split:NNnn 5304, 5305, 5307, 5308
- _seq_set_split_auxi:w 5304, 5318, 5325, 5331
- _seq_set_split_auxii:w 5304, 5333, 5337
- _seq_set_split_end: 5304, 5320, 5324, 5331, 5335, 5337
- _seq_tmp:w 5250, 5250, 5428, 5431, 5541, 5553
- _seq_use:NNnNnn 5688, 5695, 5696, 5708
- _seq_use:nwn 5688, 5701, 5717
- _seq_use:nwwwnwn 5688, 5700, 5709, 5710
- _seq_use_setup:w 5688, 5699, 5709
- _seq_wrap_item:n 5281, 5286, 5291, 5296, 5313, 5338, 5378, 5378, 5414, 15532
- _skip_if_finite:wwNw . 4171, 4175, 4179
- _str_case:nnTF 5150, 5153, 5158, 5163, 5168, 5170
- _str_case:nw ... 5150, 5171, 5176, 5180
- _str_case_end:nw 5150, 5179, 5207, 5210
- _str_case_x:nnTF 5150, 5185, 5190, 5195, 5200, 5202
- _str_case_x:nw . 5150, 5203, 5204, 5208
- _str_count_ignore_spaces:N 9620, 9688, 9688
- _str_count_ignore_spaces:n 9688, 9689, 9690
- _str_count_loop:NNNNNNNN 9688, 9693, 9697, 9703
- _str_escape_x:n 5101, 5110, 5111, 5115
- _str_fold_auxi:w 5211, 5213, 5216, 5219, 5236
- _str_fold_auxii:N 5211, 5218, 5221, 5234
- _str_fold_auxiii:NNNNNNNN 5211, 5223, 5226
- _str_fold_end:w 5211, 5214, 5236
- _str_head:w 5086, 5088, 5092
- _str_if_eq_x:nn 106, 2260, 2267, 2284, 2311, 2320, 2580, 4164, 5101, 5101, 5104, 5125, 5133, 5147, 10915, 10928, 11165, 13535
- _str_if_eq_x_return:nn 2617, 2626, 2642, 2664, 2684, 2702, 2720, 2733, 2744, 2754, 5010, 5123, 5123, 5150, 15564, 15565
- _str_tail:w 5086, 5096, 5100
- _tl_act:NNNnn 4833, 4833, 4884, 15572, 15595, 15635
- _tl_act_case_aux:nn 15622, 15630, 15633, 15652
- _tl_act_case_group:nn 15617, 15637, 15649
- _tl_act_case_normal:nN 15617, 15636, 15641
- _tl_act_case_space:n 15617, 15638, 15640
- _tl_act_count_group:nn 15591, 15597, 15605
- _tl_act_count_normal:nN 15591, 15596, 15603
- _tl_act_count_space:n 15591, 15598, 15604
- _tl_act_end:w 4833
- _tl_act_end:wn 4854, 4860
- _tl_act_group:nwnNNN . 4833, 4846, 4862

_tl_act_group_recurse:Nnn	_tl_change_case_space:wnn
..... 15582, 15586, 15586 15704, 15716, 15745
_tl_act_loop:w	_tl_change_case_upper_az:Nnn
.. 4833, 4836, 4840, 4857, 4865, 4872 15859, 15894
_tl_act_normal:NwnNNN 4833, 4843, 4851	_tl_change_case_upper_lt:Nnn
_tl_act_output:n 15896, 15927
.... 4833, 4875, 15640, 15643, 15651	_tl_change_case_upper_lt:Nw
_tl_act_result:n 15896, 15940, 15942
.. 4838, 4860, 4875, 4876, 4877, 4878	_tl_change_case_upper_sigma:Nnn ..
_tl_act_reverse_output:n 15704, 15787
. 4833, 4877, 4894, 4896, 4898, 15583	_tl_change_case_upper_tr:Nnn
_tl_act_space:wnnNNN . 4833, 4847, 4869 15859, 15888, 15895
_tl_case:NnTF 4671, 4676, 4681, 4686, 4688	_tl_count:n 4758, 4761, 4766, 4768
_tl_case:Nw 4668, 4689, 4690, 4694	_tl_from_file_do:w 15654, 15668, 15676
_tl_case:nnTF	_tl_head_auxi:nw 4905, 4908, 4910, 4921
4668	_tl_head_auxii:n 4905, 4911, 4912
_tl_case_end:nw 4668, 4693, 4702	_tl_if_blank_p:NNw
_tl_change_case:nnn 15698,	_tl_if_empty_return:o
15699, 15701, 15702, 15704, 15704 4568, 4601, 4601, 4611
_tl_change_case_N_type:Nwnn	_tl_if_head_eq_meaning_normal:nN .
..... 15704, 15712, 15720 4976, 4980
_tl_change_case_char:NNNNNNNNn ...	_tl_if_head_eq_meaning_special:nN
..... 15704, 15732, 15750, 15823 4977, 4989
_tl_change_case_group:nwnn	_tl_if_head_is_space:w 5029, 5032, 5034
..... 15704, 15715, 15740	_tl_if_single:nTF
_tl_change_case_loop:wnn	4658
..... 15704, 15706, 15709,	_tl_if_single:nnw
15738, 15743, 15748, 15830, 15852	4660, 4667
_tl_change_case_lower_az:Nnn	_tl_if_single_p:n
..... 15859, 15873	4658
_tl_change_case_lower_lt:Nnn	_tl_item:nn 5041, 5043, 5056, 5061
..... 15896, 15896	_tl_map_function:Nn
_tl_change_case_lower_lt:Nw 4703, 4705, 4711, 4714, 4721
..... 15896, 15909, 15913	_tl_map_variable:Nnn
_tl_change_case_lower_sigma:Nnn 4729, 4731, 4737, 4742
..... 15704, 15756	_tl_mixed_case:nn
_tl_change_case_lower_sigma:Nw 15700, 15703, 15789, 15789
..... 15704, 15761, 15769	_tl_mixed_case_N_type:Nwn
_tl_change_case_lower_sigma_loop:Nw 15789, 15797, 15805
..... 15704, 15774, 15779, 15785	_tl_mixed_case_group:nwn
_tl_change_case_lower_tr:Nnn 15789, 15800, 15849
..... 15859, 15859, 15874	_tl_mixed_case_loop:wn
_tl_change_case_lower_tr:Nw 15789, 15791, 15794, 15847, 15857
..... 15859, 15864, 15875	_tl_mixed_case_skip:Nwn
_tl_change_case_mixed_nl:Nnn 15789, 15818, 15832, 15843
..... 15952, 15952	_tl_mixed_case_skip_tidy:nNwn
_tl_change_case_mixed_nl:Nw 15789, 15840, 15845
..... 15952, 15958, 15961	_tl_mixed_case_space:wn
_tl_change_case_mixed_sigma:Nnn 15789, 15801, 15854
..... 15704, 15788	_tl_replace:NNNnn
	.. 4505, 4506, 4508, 4510, 4512, 4517
	_tl_replace:w .. 4505, 4542, 4545, 4550

_tl_replace_all:	_token_if_primitive_nullfont:N ...
..... 4505, 4510, 4512, 4543, 4546 2763, 2784, 2788
_tl_replace_once: 4505, 4506, 4508, 4548	_token_if_primitive_space:w
_tl_replace_once_end:w 4505, 4551, 4553 2763, 2782, 2787
_tl_rescan:w	_token_if_primitive_undefined:N ..
4467, 4485, 4493 2763, 2808, 2814
_tl_reverse_group:nn 15567, 15574, 15580	_token_if_protected_macro:w
_tl_reverse_group_preserve:nn 2604, 2735, 2740
4879, 4886, 4895	_token_if_skip_register:w
_tl_reverse_items:nwNwn 2604, 2704, 2711
..... 4771, 4773, 4774, 4778, 4781	_token_if_toks_register:w
_tl_reverse_items:wn 2604, 2722, 2729
..... 4771, 4775, 4782, 4785	_use_none_delimit_by_s_stop:w ...
_tl_reverse_normal:nN 47, 2348, 2348
..... 4879, 4885, 4893, 15573	\l
_tl_reverse_space:n	11331
..... 4879, 4887, 4897, 15575	\~
_tl_set_from_file:NNnn	2480, 2490, 9573
..... 15654, 15655, 15657, 15660	
_tl_set_from_file_x:NNnn	Numbers
..... 15679, 15680, 15682, 15685	\8
_tl_set_rescan:NNnn	8388
..... 4467, 4468, 4470, 4472, 4473	\9
_tl_tmp:w	8389
4526, 4546, 4551, 4647, 4648, 4795, 4832	_
_tl_trim_spaces:nn	210, 992,
... 103, 4788, 4795, 4797, 5838, 6274	2490, 2647, 2669, 2689, 2707, 2725,
_tl_trim_spaces_auxi:w	2738, 2749, 2759, 7750, 7751, 8101,
..... 4795, 4799, 4809, 4812, 4818	8146, 8165, 8199, 8201, 8277, 8365,
_tl_trim_spaces_auxii:w ... 4803, 4817	8369, 8370, 8374, 8375, 9538, 9576
_tl_trim_spaces_auxii:w_tl_trim_spaces_auxiii:w	
..... 4795	A
_tl_trim_spaces_auxiii:w	\A
4804, 4820, 4823, 4827	1728, 2566
_tl_trim_spaces_auxiv:w 4795, 4806, 4829	\above
_token_if_chardef:w	331
..... 2604, 2619, 2628, 2633	\abovedisplayshortskip
_token_if_dim_register:w	344
..... 2604, 2644, 2651	\abovedisplayskip
_token_if_int_register:w	345
..... 2604, 2666, 2675	\abovewithdelims
_token_if_long_macro:w	332
..... 2604, 2746, 2756, 2761	\accent
_token_if_macro_p:w .. 2564, 2575, 2578	382
_token_if_muskip_register:w	\adjdemerits
..... 2604, 2686, 2693	419
_token_if_primitive:NNw 2763, 2776, 2780	\advance
_token_if_primitive:Nw 2763, 2799, 2805	226
_token_if_primitive_loop:N	\afterassignment
..... 2763, 2783, 2796, 2802	236
	\aftergroup
	237
	false
	191
	nan
	190
	tan
	188
	tand
	189
	\AtBeginDocument
	7614, 9333
	\atop
	333
	\atopwithdelims
	334
	max
	188
	B
	\badness
	481

<code>\baselineskip</code>	409	<code>\bool_if_exist_p:c</code>	2014
<code>\batchmode</code>	302	<code>\bool_if_exist_p:N</code>	37, 2014
<code>\begingroup</code> 13, 19, 23, 60, 100, 121, 200, 240		<code>\bool_if_p:c</code>	1982
<code>\beginL</code>	594	<code>\bool_if_p:N</code>	37, 1982, 1990
<code>\beginR</code>	596	<code>\bool_if_p:n</code> 38, 1933, 1935, 1973, 1978,	
<code>\belowdisplayshortskip</code>	346	2016, 2018, 2024, 2024, 2097, 2100	
<code>\belowdisplayskip</code>	347	<code>\bool_new:c</code>	1910
<code>\binoppenalty</code>	370	<code>\bool_new:N</code> 36, 1910, 1910, 1911, 2010,	
<code>\bodydir</code>	631	2011, 2012, 2013, 6803, 8480, 8481,	
<code>\bool_do_until:cn</code>	2110	8484, 8485, 8489, 8561, 8577, 9535	
<code>\bool_do_until:Nn</code> 39, 2110, 2112, 2113, 2115		<code>\bool_not_p:n</code>	39, 2097, 2097
<code>\bool_do_until:nn</code> .. 39, 2116, 2137, 2140		<code>.bool_set:c</code>	158
<code>\bool_do_while:cn</code>	2110	<code>\bool_set:cn</code>	1932
<code>\bool_do_while:Nn</code> 39, 2110, 2110, 2111, 2114		<code>.bool_set:N</code>	158
<code>\bool_do_while:nn</code> .. 39, 2116, 2124, 2127		<code>\bool_set:Nn</code> ... 37, 1932, 1932, 1936, 1970	
<code>.bool_gset:c</code>	158	<code>\bool_set_eq:cc</code>	1924, 1927
<code>\bool_gset:cn</code>	1932	<code>\bool_set_eq:cN</code>	1924, 1926
<code>.bool_gset:N</code>	158	<code>\bool_set_eq:Nc</code>	1924, 1925
<code>\bool_gset:Nn</code> .. 37, 1932, 1934, 1937, 1975		<code>\bool_set_eq:NN</code> ... 37, 1924, 1924, 1960	
<code>\bool_gset_eq:cc</code>	1924, 1931	<code>\bool_set_false:c</code>	1912
<code>\bool_gset_eq:cN</code>	1924, 1930	<code>\bool_set_false:N</code> 36, 179, 1912, 1914,	
<code>\bool_gset_eq:Nc</code>	1924, 1929	1921, 1945, 7081, 7484, 8506, 8843,	
<code>\bool_gset_eq:NN</code> ... 37, 1924, 1928, 1965		8863, 8869, 8872, 8882, 8939, 9631	
<code>\bool_gset_false:c</code>	1912	<code>.bool_set_inverse:c</code>	158
<code>\bool_gset_false:N</code>		<code>.bool_set_inverse:N</code>	158
..... 36, 1912, 1918, 1923, 1955		<code>\bool_set_true:c</code>	1912
<code>.bool_gset_inverse:c</code>	158	<code>\bool_set_true:N</code>	
<code>.bool_gset_inverse:N</code>	158	.. 37, 193, 1912, 1912, 1920, 1940,	
<code>\bool_gset_true:c</code>	1912	7099, 7114, 7145, 8501, 8841, 8859,	
<code>\bool_gset_true:N</code> 37, 1912, 1916, 1922, 1950		8860, 8868, 8877, 8946, 9594, 9668	
<code>\bool_if:cTF</code>	1982	<code>\bool_show:c</code>	1994
<code>\bool_if:N</code>	1982	<code>\bool_show:N</code>	37, 1994, 1994, 2009
<code>\bool_if:n</code>	2016	<code>\bool_show:n</code>	37, 1994, 1997, 2003
<code>\bool_if:NF</code>		<code>\bool_until_do:cn</code>	2104
164, 1992, 2107, 2113, 3704, 7486, 9011		<code>\bool_until_do:Nn</code> 39, 2104, 2106, 2107, 2109	
<code>\bool_if:nF</code>	2131, 2140	<code>\bool_until_do:nn</code> .. 40, 2116, 2129, 2134	
<code>\bool_if:NT</code>	1991,	<code>\bool_while_do:cn</code>	2104
2105, 2111, 3704, 3705, 7085, 8193		<code>\bool_while_do:Nn</code> 39, 2104, 2104, 2105, 2108	
<code>\bool_if:nT</code>	2118, 2127, 15532,	<code>\bool_while_do:nn</code> .. 40, 2116, 2116, 2121	
15901, 15915, 15930, 15944, 15963		<code>\bool_xor_p:nn</code>	39, 2098, 2098
<code>\bool_if:NTF</code>	37, 1299, 1982,	<code>\botmark</code>	317
1993, 3690, 8544, 8890, 8926, 8932,		<code>\botmarks</code>	543
8951, 8953, 8958, 8965, 8987, 9629		<code>\box</code>	525
<code>\bool_if:nTF</code>		<code>\box_clear:c</code>	6575
... 38, 2005, 2016, 8896, 8906, 15877		<code>\box_clear:N</code>	133, 6575,
<code>\bool_if_exist:c</code>	2015	6575, 6579, 6582, 6839, 6898, 6947	
<code>\bool_if_exist:cTF</code>	2014	<code>\box_clear_new:c</code>	6581
<code>\bool_if_exist:N</code>	2014	<code>\box_clear_new:N</code> .. 133, 6581, 6581, 6585	
<code>\bool_if_exist:NF</code>	8561, 8577	<code>\box_clip:c</code>	15107
<code>\bool_if_exist:NTF</code>	37, 1996, 2014	<code>\box_clip:N</code>	196, 15107, 15107, 15109

\box_dp:c	6601, 6969	\box_if_vertical:NT	6640
\box_dp:N	134, 6601, 6602, 6605, 6608, 6968, 7039, 7041, 7058, 7072, 7225, 7243, 7556, 7596, 14850, 14970, 15057, 15118, 15125, 15130, 15247, 16173, 16175, 16184, 16186	\box_if_vertical:NTF	135, 6631, 6642
\box_gclear:c	6575	\box_if_vertical_p:c	6631
\box_gclear:N	133, 6575, 6577, 6580, 6584	\box_if_vertical_p:N	135, 6631, 6639
\box_gclear_new:c	6581	\box_log:c	6666
\box_gclear_new:N	133, 6581, 6583, 6586	\box_log:cnn	6666
\box_gset_eq:cc	6587	\box_log:N	136, 6666, 6666, 6668
\box_gset_eq:cN	6587	\box_log:Nnn	137, 6666, 6667, 6669, 6680
\box_gset_eq:Nc	6587	\box_move_down:nn	134, 6620, 6626, 15122, 15130, 15165, 15172, 15226
\box_gset_eq:NN	133, 6578, 6587, 6589, 6592	\box_move_left:nn	134, 6620, 6620
\box_gset_eq_clear:cc	6593	\box_move_right:nn	134, 6620, 6622
\box_gset_eq_clear:cN	6593	\box_move_up:nn	134, 6620, 6624, 7263, 7551, 15138, 15145, 15178, 15189
\box_gset_eq_clear:Nc	6593	\box_new:c	6567
\box_gset_eq_clear:NN	133, 6593, 6595, 6598	\box_new:N	133, 6567, 6568, 6574, 6582, 6584, 6655, 6656, 6657, 6658, 6659, 6781, 6846, 14834
\box_gset_to_last:c	6649	\box_resize:cnn	14949
\box_gset_to_last:N	136, 6649, 6651, 6654	\box_resize:Nnn	195, 14949, 14949, 14966, 15357
\box_ht:c	6601, 6971	\box_resize_to_ht:cn	14983
\box_ht:N	135, 6601, 6601, 6604, 6610, 6894, 6942, 6970, 7035, 7037, 7058, 7065, 7224, 7242, 7554, 7595, 14849, 14969, 15056, 15135, 15140, 15145, 15244, 15246, 16175, 16186	\box_resize_to_ht:Nn	195, 14983, 14983, 14999
\box_if_empty:cTF	6643	\box_resize_to_ht_plus_dp:cn	14983
\box_if_empty:N	6643	\box_resize_to_ht_plus_dp:Nn	195, 14983, 15000, 15016
\box_if_empty:NF	6647	\box_resize_to_wd:cn	14983
\box_if_empty:NT	6646	\box_resize_to_wd:Nn	195, 14983, 15017, 15030
\box_if_empty:NTF	135, 6643, 6648	\box_resize_to_wd_and_ht:cnn	14983
\box_if_empty_p:c	6643	\box_resize_to_wd_and_ht:Nnn	196, 14983, 15031, 15048
\box_if_empty_p:N	135, 6643, 6645	\box_rotate:Nn	196, 14835, 14835, 15219
\box_if_exist:c	6600	\box_scale:cnn	15049
\box_if_exist:cTF	6599	\box_scale:Nnn	196, 15049, 15049, 15070, 15379
\box_if_exist:N	6599	\box_set_dp:cn	6607
\box_if_exist:NTF	134, 6582, 6584, 6599, 6687	\box_set_dp:Nn	135, 6607, 6607, 6614, 7225, 7243, 7555, 14876, 15082, 15085, 15125, 15133, 15168, 15173, 15231
\box_if_exist_p:c	6599	\box_set_eq:cc	6587
\box_if_exist_p:N	134, 6599	\box_set_eq:cN	6587
\box_if_horizontal:cTF	6631	\box_set_eq:Nc	6587
\box_if_horizontal:N	6631	\box_set_eq:NN	133, 6576, 6587, 6587, 6590, 6591, 6957, 7245, 7559, 15150, 15194
\box_if_horizontal:NF	6637	\box_set_eq_clear:cc	6593
\box_if_horizontal:NT	6636	\box_set_eq_clear:cN	6593
\box_if_horizontal:NTF	135, 6631, 6638		
\box_if_horizontal_p:c	6631		
\box_if_horizontal_p:N	135, 6631, 6635		
\box_if_vertical:cTF	6631		
\box_if_vertical:N	6633		
\box_if_vertical:NF	6641		

<code>\box_set_eq_clear:Nc</code>	6593	<code>\c__coffin_poles_prop</code>	6789
<code>\box_set_eq_clear:NN</code>			6789 , 6791 , 6792 , 6793 , 6795 , 6796 , 6797 , 6798 , 6799 , 6800 , 6852 , 6989
<code>\box_set_ht:cn</code>	6607	<code>\c__fp_big_leading_shift_int</code>	9865 , 9865 , 12463 , 12733 , 12743 , 12753
<code>\box_set_ht:Nn</code>		<code>\c__fp_big_middle_shift_int</code>	
	135 , 6607 , 6609 , 6613 , 7224 , 7242 , 7553 , 14875 , 15081 , 15086 , 15139 , 15148 , 15179 , 15192 , 15229		9865 , 9866 , 12466 , 12469 , 12472 , 12475 , 12478 , 12481 , 12485 , 12735 , 12745 , 12755 , 12764 , 12767 , 12770
<code>\box_set_to_last:c</code>	6649	<code>\c__fp_big_trailing_shift_int</code>	
<code>\box_set_to_last:N</code>			9865 , 9867 , 12489 , 12777
	136 , 6649 , 6649 , 6652 , 6653	<code>\c__fp_Bigg_leading_shift_int</code>	
<code>\box_set_wd:cn</code>	6607		9870 , 9870 , 12313 , 12330
<code>\box_set_wd:Nn</code>		<code>\c__fp_Bigg_middle_shift_int</code>	
	135 , 6607 , 6611 , 6615 , 7226 , 7244 , 7557 , 14877 , 15098 , 15232 , 16198		9870 , 9871 , 12316 , 12319 , 12333 , 12336
<code>\box_show:c</code>	6660	<code>\c__fp_Bigg_trailing_shift_int</code>	
<code>\box_show:cnn</code>	6660		9870 , 9872 , 12322 , 12339
<code>\box_show:N</code>	136 , 6660 , 6660 , 6662	<code>\c__fp_leading_shift_int</code>	
<code>\box_show:Nnn</code>	136 , 6660 , 6661 , 6663 , 6665		9861 , 9861 , 12623 , 12632 , 12706 , 13601 , 14082 , 14119
<code>\box_trim:cnnnn</code>	15110	<code>\c__fp_ln_i_fixed_tl</code>	13051 , 13051
<code>\box_trim:Nnnnn</code>	197 , 15110 , 15110 , 15152	<code>\c__fp_ln_ii_fixed_tl</code>	13051 , 13052
<code>\box_use:c</code>	6616	<code>\c__fp_ln_iii_fixed_tl</code>	13051 , 13053
<code>\box_use:N</code>	134 , 6616 , 6617 , 6619 , 7260 , 7263 , 7341 , 7478 , 7548 , 7551 , 14864 , 14871 , 14879 , 15076 , 15093 , 15102 , 15115 , 15123 , 15131 , 15138 , 15146 , 15158 , 15166 , 15172 , 15178 , 15190 , 15227 , 15234 , 16190	<code>\c__fp_ln_iv_fixed_tl</code>	13051 , 13054
<code>\box_use_clear:c</code>	6616	<code>\c__fp_ln_ix_fixed_tl</code>	13051 , 13058
<code>\box_use_clear:N</code>	134 , 6616 , 6616 , 6618	<code>\c__fp_ln_vi_fixed_tl</code>	13051 , 13055
<code>\box_viewport:cnnnn</code>	15153	<code>\c__fp_ln_vii_fixed_tl</code>	13051 , 13056
<code>\box_viewport:Nnnnn</code>		<code>\c__fp_ln_viii_fixed_tl</code>	13051 , 13057
	197 , 15153 , 15153 , 15196	<code>\c__fp_ln_x_fixed_tl</code>	13051 , 13059 , 13269 , 13276
<code>\box_wd:c</code>	6601 , 6973	<code>\c__fp_max_exponent_int</code>	
<code>\box_wd:N</code>			9762 , 9762 , 9768 , 9774 , 9790 , 9791 , 12830 , 13031 , 13322 , 13349 , 13636 , 14546 , 14595
	135 , 6601 , 6603 , 6606 , 6612 , 6972 , 7037 , 7041 , 7047 , 7052 , 7193 , 7226 , 7244 , 7261 , 7549 , 7558 , 7597 , 14851 , 14971 , 15058 , 15159 , 15246 , 15251 , 15416 , 15423 , 16174 , 16185 , 16192	<code>\c__fp_middle_shift_int</code>	9861 , 9862 , 12635 , 12638 , 12641 , 12644 , 12708 , 12711 , 12714 , 12717 , 13603 , 13606 , 13609 , 13612 , 14085 , 14092 , 14122 , 14128
<code>\boxmaxdepth</code>	487	<code>\c__fp_one_fixed_tl</code>	12612 , 12612 , 13223 , 13381 , 13637 , 13661 , 142

<code>\c__int_from_roman_i_int</code>	3767	<code>\c__msg_return_text_tl</code>	
<code>\c__int_from_roman_L_int</code>	3767	.. 7684 , 7700 , 7703 , 8103 , 8111 , 8118	
<code>\c__int_from_roman_l_int</code>	3767	<code>\c__msg_text_prefix_tl</code>	7631 ,
<code>\c__int_from_roman_M_int</code>	3767	7631 , 7635 , 7668 , 7677 , 7850 , 7861 ,	
<code>\c__int_from_roman_m_int</code>	3767	7876 , 7893 , 7901 , 7907 , 8303 , 8330	
<code>\c__int_from_roman_V_int</code>	3767	<code>\c__msg_trouble_text_tl</code>	7684 , 7710
<code>\c__int_from_roman_v_int</code>	3767	<code>\c__tl_accents_lt_tl</code>	15898
<code>\c__int_from_roman_X_int</code>	3767	<code>\c__tl_act_lowercase_tl</code>	
<code>\c__int_from_roman_x_int</code>	3767 15607 , 15612 , 15630	
<code>\c__iow_wrap_end_marker_tl</code> ..	9541 , 9600	<code>\c__tl_act_uppercase_tl</code>	
<code>\c__iow_wrap_indent_marker_tl</code>	9541 , 9561 15607 , 15607 , 15622	
<code>\c__iow_wrap_marker_tl</code>		<code>\c__tl_after_final_sigma_clist</code> ..	15775
..... 9541 , 9543 , 9552 , 9613 , 9658		<code>\c__tl_dot_above_tl</code>	15925
<code>\c__iow_wrap_newline_marker_tl</code>		<code>\c__tl_dotless_i_tl</code>	15865 , 15882
..... 9541 , 9575		<code>\c__tl_dotted_I_tl</code>	15891
<code>\c__iow_wrap_unindent_marker_tl</code>		<code>\c__tl_final_sigma_tl</code>	15765 , 15784
..... 9541 , 9563		<code>\c__tl_mixed_exceptions_tl</code>	15821
<code>\c__keys_code_root_tl</code> ...	8472 , 8472 ,	<code>\c__tl_mixed_skip_clist</code>	15819
8654 , 8999 , 9001 , 9024 , 9030 , 9035		<code>\c__tl_rescan_marker_tl</code>	
<code>\c__keys_info_root_tl</code> 4466 , 4466 , 4476 , 4494 , 15666 , 15677	
8472 , 8473 , 8597 , 8599 , 8612 , 8653 ,		<code>\c__tl_std_sigma_tl</code> .	15764 , 15772 , 15782
8659 , 8660 , 8664 , 8667 , 8684 , 8686 ,		<code>\c_alignment_token</code> ..	52 , 2452 , 2458 , 2508
8687 , 8688 , 8920 , 8922 , 8967 , 8975 ,		<code>\c_catcode_active_tl</code> 52 , 2467 , 2469 , 2546	
8977 , 9100 , 9103 , 9105 , 9116 , 9136		<code>\c_catcode_letter_token</code>	
<code>\c__keys_props_root_tl</code> ..	8474 , 8474 , 52 , 2452 , 2464 , 2536	
8512 , 8548 , 8555 , 8698 , 8700 , 8702 ,		<code>\c_catcode_other_space_tl</code>	
8704 , 8706 , 8708 , 8710 , 8712 , 8714 ,		174 , 9536 , 9539 , 9551 , 9553 , 9555 , 9576	
8716 , 8718 , 8720 , 8722 , 8724 , 8726 ,		<code>\c_catcode_other_token</code> 52 , 2452 , 2465 , 2541	
8728 , 8730 , 8732 , 8734 , 8736 , 8738 ,		<code>\c_e_fp</code>	182 , 14812 , 14812
8740 , 8742 , 8744 , 8746 , 8748 , 8750 ,		<code>\c_eight</code> 72 , 2380 , 2412 , 3766 , 3831 , 3836 ,	
8752 , 8754 , 8756 , 8758 , 8760 , 8762 ,		9969 , 10698 , 10730 , 13315 , 13695 ,	
8764 , 8766 , 8768 , 8770 , 8772 , 8774 ,		13708 , 14066 , 14068 , 14070 , 14129	
8776 , 8778 , 8780 , 8782 , 8784 , 8786 ,		<code>\c_eleven</code>	72 , 2386 ,
8788 , 8790 , 8792 , 8794 , 8796 , 8798 ,		2418 , 3831 , 3839 , 12046 , 12121 , 12162	
8800 , 8802 , 8804 , 8806 , 8808 , 8810 ,		<code>\c_empty_box</code>	
8812 , 8814 , 8816 , 9109 , 9111 , 9146		... 136 , 6576 , 6578 , 6655 , 6655 , 7338	
<code>\c__max_constdef_int</code> 3114 , 3118 , 3138 , 3142		<code>\c_empty_clist</code>	
<code>\c__msg_coding_error_text_tl</code>		125 , 5767 , 5767 , 5897 , 5912 , 5934 , 5950	
..... 155 , 7577 , 7588 , 7684 ,		<code>\c_empty_coffin</code> 144 , 6962 , 6962 , 6963 , 7339	
7684 , 8100 , 8108 , 8134 , 8142 , 8151 ,		<code>\c_empty_prop</code>	
8158 , 8172 , 8181 , 8188 , 8210 , 8217 ,		132 , 6300 , 6300 , 6304 , 6308 , 6311 , 6469	
8224 , 8232 , 9053 , 9074 , 9081 , 9142		<code>\c_empty_seq</code>	116 , 5251 , 5251 ,
<code>\c__msg_continue_text_tl</code> 7684 , 7689 , 7728		5255 , 5259 , 5262 , 5442 , 5479 , 5487	
<code>\c__msg_critical_text_tl</code> 7684 , 7691 , 7864		<code>\c_empty_tl</code> 103 , 3518 , 3524 , 4275 , 4291 ,	
<code>\c__msg_fatal_text_tl</code> ..	7684 , 7693 , 7853	4293 , 4318 , 4318 , 4579 , 5214 , 5767	
<code>\c__msg_help_text_tl</code> ...	7684 , 7695 , 7732	<code>\c_false_bool</code>	22 , 859 ,
<code>\c__msg_more_text_prefix_tl</code>		898 , 938 , 943 , 987 , 988 , 1009 , 1245 ,	
..... 7631 , 7632 , 7670 , 7679 , 7870		1252 , 1421 , 1423 , 1432 , 1444 , 1757 ,	
<code>\c__msg_no_info_text_tl</code> 7684 , 7697 , 7727		1910 , 1915 , 1919 , 1948 , 1958 , 2072 ,	
<code>\c__msg_on_line_text_tl</code> 7684 , 7702 , 7719		2074 , 2078 , 2101 , 3679 , 3684 , 3693	

- \c_fifteen 72,
2394, 2426, 3831, 3842, 11056, 11281
- \c_five ... 72, 2374, 2406, 3831, 3835,
10210, 11286, 12360, 12955, 13322
- \c_four 72,
2372, 2404, 3831, 3834, 9672, 9678,
11287, 12794, 12810, 12995, 13036,
13094, 13274, 13868, 13874, 13877,
14146, 14273, 14417, 14490, 14706
- \c_fourteen 72, 2392, 2424, 3831, 3841, 11281
- \c_group_begin_token 52, 2452,
2452, 2493, 4965, 5000, 6707, 6755
- \c_group_end_token
52, 2452, 2453, 2498, 6712, 6713, 6763
- \c_inf_fp 182,
9757, 9759, 11014, 12189, 12269,
13291, 13525, 13529, 13550, 13826
- \c_job_name_tl 103, 4319, 4332, 4336
- \c_log_iow ... 174, 9453, 9453, 9518, 9519
- \c_math_subscript_token
..... 52, 2452, 2462, 2526
- \c_math_superscript_token
..... 52, 2452, 2460, 2521
- \c_math_toggle_token 52, 2452, 2456, 2503
- \c_max_dim 82, 4108, 4109, 4199,
15298, 15299, 15300, 15301, 15318
- \c_max_int 72, 3849, 3849, 6661, 6667
- \c_max_muskip 88, 4260, 4261
- \c_max_register_int
..... 72, 746, 747, 749, 3031, 8165
- \c_max_skip 85, 4198, 4199
- \c_minus_inf_fp 182,
9757, 9760, 12190, 12272, 13067, 13827
- \c_minus_one 72, 734, 735, 738, 739, 1094,
1266, 3116, 3177, 3831, 4477, 4478,
7794, 9403, 9446, 9453, 9496, 9542,
9568, 11128, 11436, 11728, 11878,
12656, 13688, 13697, 13723, 14351
- \c_minus_zero_fp
..... 182, 9757, 9758, 12186, 14475
- \c_nan_fp ... 9757, 9761, 10064, 10072,
10144, 10285, 10455, 10460, 10538,
10577, 11015, 11547, 13498, 14257
- \c_nine ... 72, 2382, 2414, 3831, 3837,
10406, 10433, 10605, 10625, 10652,
10666, 10701, 10728, 10756, 10772,
10788, 10806, 10885, 10901, 11284,
11285, 12172, 13868, 13874, 13877
- \c_one 72, 2366, 2398,
3090, 3175, 3831, 3831, 4768, 5048,
5059, 5601, 5612, 5686, 6181, 6185,
6232, 6664, 8028, 8034, 9219, 9589,
9790, 9986, 10188, 10240, 10241,
10244, 10279, 10346, 10371, 10615,
10678, 10688, 10757, 10773, 10777,
10814, 10987, 11230, 11235, 11242,
11393, 11398, 11403, 11411, 11412,
11413, 11414, 11726, 11843, 11852,
11856, 11972, 11997, 12036, 12090,
12093, 12115, 12119, 12139, 12163,
12179, 12250, 12283, 12372, 12374,
12401, 12435, 12498, 12502, 12506,
12555, 12560, 12589, 12668, 12819,
12883, 12885, 12934, 12935, 13010,
13046, 13078, 13125, 13220, 13316,
13318, 13363, 13657, 13667, 13690,
13710, 13713, 13726, 13799, 13837,
13885, 14224, 14250, 14270, 14329,
14435, 14497, 14563, 14568, 16009
- \c_one_degree_fp 182, 11017, 14814, 14815
- \c_one_fp 182, 11018, 11463,
11483, 11767, 13285, 13493, 13540,
13750, 13780, 14251, 14812, 14813
- \c_one_hundred 72, 3846, 3846
- \c_one_thousand 72, 3846, 3847
- \c_parameter_token
..... 52, 2452, 2459, 2512, 2515
- \c_pi_fp 182, 11016, 14814, 14814
- \c_seven 72, 734,
744, 2378, 2410, 3831, 10737, 11423,
11469, 12067, 12174, 14287, 14305
- \c_six 72, 734, 743, 2376, 2408, 3831
- \c_sixteen 72, 734, 741, 1096,
3764, 3831, 9340, 9403, 9428, 9454,
9496, 9883, 9956, 10322, 10986,
11065, 11515, 11535, 13337, 13696,
13701, 14614, 14616, 14622, 14663
- \c_space_tl 103, 4338, 4338, 4869, 5218,
6177, 6186, 7720, 9297, 9587, 9601,
9674, 14755, 15745, 15747, 15854,
15856, 16157, 16162, 16205, 16210,
16213, 16224, 16238, 16239, 16246
- \c_space_token 52, 2452, 2463,
2531, 2923, 4966, 5001, 15693, 16008
- \c_ten 72, 2384, 2416, 3569,
3602, 3831, 3838, 11282, 11283,
11293, 11297, 12171, 13095, 14268
- \c_ten_thousand 72, 3846, 3848,
12618, 12649, 12650, 12727, 12787
- \c_term_ior 174, 9340, 9340, 9351, 9409, 9502

- \c_term_iow [174](#), [9453](#), [9454](#), [9466](#), [9520](#), [9521](#)
- \c_thirteen [72](#),
[2390](#), [2422](#), [3831](#), [3840](#), [14083](#), [14120](#)
- \c_thirty_two
. [72](#), [3843](#), [3843](#), [10516](#), [10551](#), [11176](#)
- \c_three [72](#), [2370](#), [2402](#), [3831](#), [3833](#), [9792](#),
[10517](#), [10554](#), [11178](#), [11362](#), [11366](#),
[11376](#), [11876](#), [12166](#), [13814](#), [14309](#)
- \c_token_A_int [2763](#), [2798](#)
- \c_true_bool [22](#), [898](#), [987](#),
[987](#), [1008](#), [1263](#), [1422](#), [1433](#), [1443](#),
[1913](#), [1917](#), [1943](#), [1953](#), [1984](#), [2073](#),
[2075](#), [2079](#), [2102](#), [3679](#), [3684](#), [3697](#)
- \c_twelve ... [72](#), [734](#), [745](#), [2203](#), [2218](#),
[2388](#), [2420](#), [2611](#), [3831](#), [10968](#), [10969](#)
- \c_two [72](#), [2368](#), [2400](#), [3090](#), [3762](#),
[3831](#), [3832](#), [8029](#), [9195](#), [9787](#), [9791](#),
[11382](#), [11521](#), [11646](#), [11797](#), [12023](#),
[12154](#), [12169](#), [12172](#), [12174](#), [12354](#),
[12430](#), [12433](#), [12435](#), [12451](#), [12680](#),
[12830](#), [12934](#), [12935](#), [12949](#), [13031](#),
[13228](#), [13349](#), [13636](#), [13667](#), [13683](#),
[13754](#), [13784](#), [13874](#), [14146](#), [14156](#),
[14199](#), [14224](#), [14270](#), [14272](#), [14358](#),
[14380](#), [14666](#), [16006](#), [16007](#), [16008](#)
- \c_two_hundred_fifty_five [72](#), [3844](#), [3844](#)
- \c_two_hundred_fifty_six . [72](#), [3844](#), [3845](#)
- \c_zero [72](#), [734](#),
[742](#), [898](#), [908](#), [913](#), [918](#), [995](#), [997](#),
[1505](#), [1514](#), [2174](#), [2177](#), [2178](#), [2179](#),
[2180](#), [2181](#), [2182](#), [2183](#), [2184](#), [2185](#),
[2186](#), [2196](#), [2198](#), [2261](#), [2268](#), [2285](#),
[2312](#), [2321](#), [2364](#), [2396](#), [2580](#), [3085](#),
[3144](#), [3145](#), [3195](#), [3203](#), [3379](#), [3382](#),
[3516](#), [3522](#), [3831](#), [4090](#), [4165](#), [4701](#),
[4861](#), [5037](#), [5038](#), [5047](#), [5125](#), [5134](#),
[5147](#), [5600](#), [6228](#), [6243](#), [6263](#), [6673](#),
[6674](#), [8291](#), [9197](#), [9792](#), [9955](#), [9980](#),
[10192](#), [10196](#), [10198](#), [10202](#), [10206](#),
[10219](#), [10231](#), [10242](#), [10243](#), [10252](#),
[10256](#), [10260](#), [10263](#), [10268](#), [10272](#),
[10329](#), [10334](#), [10510](#), [10548](#), [10650](#),
[10719](#), [10766](#), [10796](#), [10839](#), [10844](#),
[10866](#), [10915](#), [10929](#), [10976](#), [10989](#),
[11135](#), [11166](#), [11214](#), [11394](#), [11399](#),
[11404](#), [11410](#), [11476](#), [11638](#), [11812](#),
[11817](#), [12088](#), [12372](#), [12524](#), [12559](#),
[12561](#), [12562](#), [12563](#), [13250](#), [13254](#),
[13268](#), [13326](#), [13332](#), [13346](#), [13536](#),
[13548](#), [13566](#), [13589](#), [13621](#), [13704](#),
[13705](#), [13719](#), [13721](#), [13739](#), [13769](#),
[13837](#), [14135](#), [14259](#), [14268](#), [14274](#),
[14308](#), [14322](#), [14327](#), [14612](#), [14637](#)
- \c_zero_dim [82](#), [3881](#), [4108](#), [4108](#),
[4198](#), [6722](#), [6736](#), [7095](#), [7098](#), [7101](#),
[7110](#), [7113](#), [7116](#), [7125](#), [7132](#), [7189](#),
[7194](#), [7201](#), [15101](#), [15122](#), [15133](#),
[15138](#), [15148](#), [15161](#), [15165](#), [15173](#),
[15175](#), [15178](#), [15182](#), [15192](#), [16198](#)
- \c_zero_fp [182](#), [9757](#),
[9757](#), [9800](#), [11019](#), [11475](#), [11486](#),
[11769](#), [12019](#), [12185](#), [13294](#), [13520](#),
[13553](#), [14473](#), [14703](#), [14765](#), [14780](#),
[14781](#), [14853](#), [14855](#), [14860](#), [15079](#),
[15088](#), [15367](#), [16200](#), [16206](#), [16221](#)
- \c_zero_muskip [88](#), [4219](#), [4260](#), [4260](#)
- \c_zero_skip
... [85](#), [4128](#), [4198](#), [4198](#), [15555](#), [15556](#)
- \catcode [4](#),
[5](#), [6](#), [7](#), [10](#), [28](#), [29](#), [139](#), [140](#), [141](#),
[142](#), [143](#), [144](#), [145](#), [146](#), [147](#), [152](#),
[153](#), [154](#), [155](#), [156](#), [157](#), [158](#), [159](#), [529](#)
- \catcodetable [622](#)
- \char [383](#), [2622](#)
- \char_gset_active:Npn . [204](#), [15975](#), [15989](#)
- \char_gset_active:Npx [15975](#), [15990](#)
- \char_gset_active:eq:NN [204](#), [15975](#), [15992](#)
- \char_set_active:Npn .. [204](#), [15975](#), [15987](#)
- \char_set_active:Npx [15975](#), [15988](#)
- \char_set_active:eq:NN [204](#), [15975](#), [15991](#)
- \char_set_catcode:nn [50](#), [168](#),
[169](#), [170](#), [171](#), [172](#), [173](#), [174](#), [175](#),
[176](#), [2357](#), [2357](#), [2364](#), [2366](#), [2368](#),
[2370](#), [2372](#), [2374](#), [2376](#), [2378](#), [2380](#),
[2382](#), [2384](#), [2386](#), [2388](#), [2390](#), [2392](#),
[2394](#), [2396](#), [2398](#), [2400](#), [2402](#), [2404](#),
[2406](#), [2408](#), [2410](#), [2412](#), [2414](#), [2416](#),
[2418](#), [2420](#), [2422](#), [2424](#), [2426](#), [2611](#)
- \char_set_catcode_active:N
. [49](#), [2363](#), [2389](#), [2468](#), [2475](#), [2476](#),
[2477](#), [2478](#), [2479](#), [2480](#), [7753](#), [15976](#)
- \char_set_catcode_active:n
... [49](#), [2395](#), [2421](#), [8386](#), [8387](#), [15981](#)
- \char_set_catcode_alignment:N
..... [49](#), [2363](#), [2371](#), [2457](#)
- \char_set_catcode_alignment:n
..... [49](#), [186](#), [2395](#), [2403](#)
- \char_set_catcode_comment:N
..... [49](#), [2363](#), [2391](#)

- \char_set_catcode_comment:n 49, 2395, 2423
- \char_set_catcode_end_line:N 49, 2363, 2373
- \char_set_catcode_end_line:n 49, 2395, 2405
- \char_set_catcode_escape:N 49, 2363, 2363
- \char_set_catcode_escape:n 49, 2395, 2395
- \char_set_catcode_group_begin:N 49, 2363, 2365
- \char_set_catcode_group_begin:n 49, 2395, 2397
- \char_set_catcode_group_end:N 49, 2363, 2367
- \char_set_catcode_group_end:n 49, 2395, 2399
- \char_set_catcode_ignore:N 49, 2363, 2381
- \char_set_catcode_ignore:n 49, 183, 184, 2395, 2413
- \char_set_catcode_invalid:N 49, 2363, 2393
- \char_set_catcode_invalid:n 49, 2395, 2425
- \char_set_catcode_letter:N 49, 2363, 2385, 10979, 10980, 11160, 11211, 11227, 11259, 11260, 11261, 11262, 11317, 11331, 11332, 11359, 11773, 11774
- \char_set_catcode_letter:n 49, 187, 189, 2395, 2417
- \char_set_catcode_math_subscript:N . 49, 2363, 2379, 2461
- \char_set_catcode_math_subscript:n . 49, 2395, 2411
- \char_set_catcode_math_superscript:N 49, 2363, 2377, 8276
- \char_set_catcode_math_superscript:n 49, 188, 2395, 2409
- \char_set_catcode_math_toggle:N 49, 2363, 2369, 2455
- \char_set_catcode_math_toggle:n 49, 2395, 2401
- \char_set_catcode_other:N 49, 2363, 2387, 2565, 2566, 2765, 9537, 10388, 10389, 10390, 10474, 10475, 10476, 10477, 11258, 14529, 15996, 15997, 15998, 15999, 16000
- \char_set_catcode_other:n 49, 185, 190, 2395, 2419
- \char_set_catcode_parameter:N 49, 2363, 2375
- \char_set_catcode_parameter:n 49, 2395, 2407
- \char_set_catcode_space:N 49, 2363, 2383
- \char_set_catcode_space:n 49, 191, 2395, 2415
- \char_set_lccode:nn 50, 2427, 2433, 2567, 2568, 2569, 2605, 2606, 2607, 2608, 2609, 2766, 2767, 2768, 7750, 7751, 7752, 8277, 8278, 8279, 8280, 8388, 8389, 9538, 10391, 15983
- \char_set_mathcode:nn ... 51, 2427, 2427
- \char_set_sfcode:nn 51, 2427, 2445
- \char_set_uccode:nn 51, 2427, 2439
- \char_show_value_catcode:n 50, 2357, 2361
- \char_show_value_lccode:n 50, 2427, 2437
- \char_show_value_mathcode:n 51, 2427, 2431
- \char_show_value_sfcode:n 52, 2427, 2449
- \char_show_value_uccode:n 51, 2427, 2443
- \char_tmp:NN 15977, 15987, 15988, 15989, 15990, 15991, 15992
- \char_value_catcode:n 50, 168, 169, 170, 171, 172, 173, 174, 175, 176, 2357, 2359
- \char_value_lccode:n 50, 2427, 2435
- \char_value_mathcode:n .. 51, 2427, 2429
- \char_value_sfcode:n 51, 2427, 2447
- \char_value_uccode:n 51, 2427, 2441
- \chardef 149, 161, 218
- .choice: 158
- .choices:nn 158
- .choices:on 158
- .choices:Vn 158
- .choices:xn 158
- \cleaders 401
- \clist_clear:c 5775, 5776
- \clist_clear:N 117, 5775, 5775, 5792, 5986, 8833, 8851
- \clist_clear_new:c 5779, 5780
- \clist_clear_new:N 118, 5779, 5779
- \clist_concat:ccc 5819
- \clist_concat:NNN 118, 5819, 5819, 5832, 5870, 5883
- \clist_const:cn 5772
- \clist_const:cx 5772
- \clist_const:Nn ... 117, 5772, 5772, 5774
- \clist_const:Nx 5772
- \clist_count:c 6163

\clist_count:N	5980
... 122, 6163, 6163, 6171, 6192, 6221	
\clist_count:n	5995
... 6163, 6172, 6251	
\clist_gclear:c	6027
\clist_gclear:N	6032
... 117, 5775, 5777, 5794	
\clist_gclear_new:c	158
... 5779, 5782	
\clist_gclear_new:N	5863
... 118, 5779, 5781	
\clist_gconcat:ccc	5863
... 5819	
\clist_gconcat:NNN	5863
... 118, 5819, 5821, 5833, 5872, 5885	
\clist_get:cn	158
... 5895	
\clist_get:cNTF	5868
... 5932	
\clist_get:NN	5863
... 124, 5895, 5895, 5905, 5932	
\clist_get:NNF	5863
... 5942	
\clist_get:NNT	5863
... 5941	
\clist_get:NNTF	5863
... 124, 5932, 5943	
\clist_gpop:cn	5790
... 5906	
\clist_gpop:cNTF	5789
... 5932	
\clist_gpop:NN	5788
... 124, 5906, 5908, 5931, 5946	
\clist_gpop:NNF	5983
... 5961	
\clist_gpop:NNT	5791
... 5960	
\clist_gpop:NNTF	5791
... 124, 5932, 5962	
\clist_gpush:cn	5791
... 5963, 5975	
\clist_gpush:co	5791
... 5963, 5977	
\clist_gpush:cV	6046
... 5963, 5976	
\clist_gpush:cx	6045
... 5963, 5978	
\clist_gpush:Nn	6045
... 125, 5963, 5971	
\clist_gpush:No	6047
... 5963, 5973	
\clist_gpush:NV	5828, 6013, 6088, 6118, 6137
... 5963, 5972	
\clist_gpush:Nx	5835
... 5963, 5974	
\clist_gput_left:cn	5834
... 5869, 5975	
\clist_gput_left:co	5834
... 5869, 5977	
\clist_gput_left:cV	9318
... 5869, 5976	
\clist_gput_left:cx	118, 5834, 6190, 9291
... 5869, 5978	
\clist_gput_left:Nn	5834
... 119, 5869, 5871, 5880, 5881, 5971	
\clist_gput_left:No	5834
... 5869, 5973	
\clist_gput_left:NV	5834
... 5869, 5972	
\clist_gput_left:Nx	5834
... 5869, 5974	
\clist_gput_right:cn	6061
... 5882	
\clist_gput_right:co	6061
... 5882	
\clist_gput_right:cV	6061
... 5882	
\clist_gput_right:cx	6061
... 5882	
\clist_gput_right:Nn	6061
... 119, 5882, 5884, 5893, 5894	
\clist_gput_right:No	6061
... 5882	
\clist_gput_right:NV	6065
... 5882	
\clist_gput_right:Nx	6065
... 5882	
\clist_gremove_all:cn	6084
... 5996	
\clist_gremove_all:Nn	6077, 6078
... 119, 5996, 5998, 6026	
\clist_gremove_duplicates:c	5980
\clist_gremove_duplicates:N	5980
... 119, 5980, 5982, 5995	
\clist_greverse:c	6027
\clist_greverse:N	6032
... 120, 6027, 6029, 6032	
\clist_gset:c	158
\clist_gset:cn	5863
\clist_gset:co	5863
\clist_gset:cV	5863
\clist_gset:cx	5863
\clist_gset:N	158
\clist_gset:Nn	118, 5863, 5865, 5868
\clist_gset:No	5863
\clist_gset:NV	5863
\clist_gset:Nx	5863
\clist_gset_eq:cc	5783, 5790
\clist_gset_eq:cN	5783, 5789
\clist_gset_eq:Nc	5783, 5788
\clist_gset_eq:NN	118, 5783, 5787, 5983
\clist_gset_from_seq:cc	5791
\clist_gset_from_seq:cN	5791
\clist_gset_from_seq:Nc	5791
\clist_gset_from_seq:NN	118, 5791, 5793, 5817, 5818
\clist_if_empty:c	6046
\clist_if_empty:CTF	6045
\clist_if_empty:N	6045
\clist_if_empty:n	6047
\clist_if_empty:NF	5828, 6013, 6088, 6118, 6137
... 5828, 6013, 6088, 6118, 6137	
\clist_if_empty:NTF	120, 6045, 8253
\clist_if_empty:nTF	120, 6047
\clist_if_empty_p:c	6045
\clist_if_empty_p:N	120, 6045
\clist_if_empty_p:n	120, 6047
\clist_if_exist:c	5835
\clist_if_exist:CTF	5834
\clist_if_exist:N	5834
\clist_if_exist:NT	9318
\clist_if_exist:NTF	118, 5834, 6190, 9291
\clist_if_exist_p:c	5834
\clist_if_exist_p:N	118, 5834
\clist_if_in:cnTF	6061
\clist_if_in:coTF	6061
\clist_if_in:cVTF	6061
\clist_if_in:Nn	6061
\clist_if_in:nn	6065
\clist_if_in:NnF	5989, 6079, 6080
\clist_if_in:nnF	6084
\clist_if_in:NnT	6077, 6078

\clist_if_in:nnT	6083	\clist_put_left:No	5869, 5965
\clist_if_in:NnTF	120, 6061, 6081, 6082	\clist_put_left:Nv	5869, 5964
\clist_if_in:nnTF	6061, 6085, 10031	\clist_put_left:Nx	5869, 5966
\clist_if_in:NoTF	6061	\clist_put_right:cn	5882
\clist_if_in:noTF	6061	\clist_put_right:co	5882
\clist_if_in:NvTF	6061	\clist_put_right:cV	5882
\clist_if_in:nvTF	6061	\clist_put_right:cx	5882
\clist_item:cn	6218	\clist_put_right:Nn	119, 5882, 5882, 5891, 5892, 5990
\clist_item:Nn	125, 6218, 6218, 6247	\clist_put_right:No	5882
\clist_item:nn	6248, 6248	\clist_put_right:Nv	5882
\clist_map_break:	122, 6092, 6097, 6106, 6110, 6125, 6143, 6159, 6159, 6160, 6162	\clist_put_right:Nx	5882, 9008
\clist_map_break:n	122, 6159, 6161, 8947	\clist_remove_all:cn	5996
\clist_map_function:cn	6086	\clist_remove_all:Nn	119, 5996, 5996, 6025
\clist_map_function:NN	121, 5281, 5291, 6086, 6086, 6101, 6168, 6281	\clist_remove_duplicates:c	5980
\clist_map_function:nN	5286, 5296, 6102, 6102, 9021, 9120	\clist_remove_duplicates:N	119, 5980, 5980, 5994
\clist_map_inline:cn	6116	\clist_reverse:c	6027
\clist_map_inline:Nn	121, 5987, 6116, 6116, 6132, 6134, 8942, 9320, 9335	\clist_reverse:N	120, 6027, 6027, 6031
\clist_map_inline:nn	6116, 6129, 8639	\clist_reverse:n	120, 6028, 6030, 6033, 6033
\clist_map_variable:cNn	6135	\clist_set:c	158
\clist_map_variable:NNn	121, 6135, 6135, 6149, 6158	\clist_set:cn	5863
\clist_map_variable:nNn	6135, 6146	\clist_set:co	5863
\clist_new:c	5770, 5771	\clist_set:cV	5863
\clist_new:N	117, 5770, 5770, 5979, 6289, 6290, 6291, 6292, 8477	\clist_set:cx	5863
\clist_pop:cn	5906	\clist_set:N	118, 5863, 5863, 5867, 5870, 5872, 5883, 5885, 6067, 6131, 6148, 6285, 8666
\clist_pop:cNTF	5932	\clist_set:No	5863
\clist_pop:NN	124, 5906, 5906, 5930, 5944	\clist_set:Nv	5863
\clist_pop:NNF	5958	\clist_set:Nx	5863
\clist_pop:NNT	5957	\clist_set_eq:cc	5783, 5786
\clist_pop:NNTF	124, 5932, 5959	\clist_set_eq:cN	5783, 5785
\clist_push:cn	5963, 5967	\clist_set_eq:Nc	5783, 5784
\clist_push:co	5963, 5969	\clist_set_eq:NN	118, 5783, 5783, 5981
\clist_push:cV	5963, 5968	\clist_set_from_seq:cc	5791
\clist_push:cx	5963, 5970	\clist_set_from_seq:cN	5791
\clist_push:Nn	125, 5963, 5963	\clist_set_from_seq:Nc	5791
\clist_push:No	5963, 5965	\clist_set_from_seq:NN	118, 5791, 5791, 5815, 5816
\clist_push:Nv	5963, 5964	\clist_show:c	6278
\clist_push:Nx	5963, 5966	\clist_show:N	125, 6278, 6278, 6286, 6288
\clist_put_left:cn	5869, 5967	\clist_show:n	125, 6278, 6283
\clist_put_left:co	5869, 5969	\clist_use:cn	6188
\clist_put_left:cV	5869, 5968	\clist_use:cnnn	6188
\clist_put_left:cx	5869, 5970	\clist_use:Nn	123, 6188, 6215, 6217
\clist_put_left:Nn	119, 5869, 5869, 5878, 5879, 5963	\clist_use:Nnnn	123, 6188, 6188, 6208, 6216
		\closein	277
		\closeout	272

\clubpenalties	585	\coffin_set_horizontal_pole:Nnn	142, 7005, 7005, 7029
\clubpenalty	412	\coffin_set_vertical_pole:cnn ...	7005
.code:n	158	\coffin_set_vertical_pole:Nnn	142, 7005, 7016, 7030
\coffin_attach:cnncnnnn	7220	\coffin_show_structure:c	7561
\coffin_attach:cnnNnnnn	7220	\coffin_show_structure:N	144, 7561, 7561, 7573
\coffin_attach:Nnncnnnn	7220	\coffin_typeset:cnnnn	7336
\coffin_attach:NnnNnnnn	143, 7220, 7220, 7247	\coffin_typeset:Nnnnn 143, 7336, 7336, 7343	
\coffin_attach_mark:NnnNnnnn	7220, 7238, 7408, 7429, 7445	\coffin_wd:c	6968, 6973
\coffin_clear:c	6835	\coffin_wd:N 144, 6968, 6972, 15351, 15385	
\coffin_clear:N ...	141, 6835, 6835, 6843	\color	7404, 7416, 7459, 7499
\coffin_display_handles:cn	7451	\color_ensure_current: 145, 6862, 6906,	
\coffin_display_handles:Nn	144, 7451, 7451, 7536	6929, 7608, 7609, 7613, 7618, 7623	
\coffin_dp:c	6968, 6969	\color_group_begin:	145, 6861, 6883, 6906, 6929, 7602, 7602
\coffin_dp:N 143, 6968, 6968, 15355, 15381		\color_group_end:	145, 6864, 6885, 6909, 6932, 7602, 7603
\coffin_ht:c	6968, 6971	\columnwidth	6881, 6927
\coffin_ht:N 144, 6968, 6970, 15355, 15381		\copy	469
\coffin_if_exist:cTF	6812	acos	189
\coffin_if_exist:N	6812	acosd	189
\coffin_if_exist:Nf	6824	acot	190
\coffin_if_exist:NT	6823	acotd	190
\coffin_if_exist:NTF 141, 6812, 6825, 6828		\count	520
\coffin_if_exist_p:c	6812	\countdef	219
\coffin_if_exist_p:N ...	141, 6812, 6822	\cr	244
\coffin_join:cnncnnnn	7183	\crr	245
\coffin_join:cnnNnnnn	7183	\cs:w	18, 702, 704, 719,
\coffin_join:Nnncnnnn	7183	721, 780, 1041, 1069, 1289, 1338,	
\coffin_join:NnnNnnnn 143, 7183, 7183, 7219		1469, 1508, 1522, 1524, 1528, 1529,	
\coffin_mark_handle:cnnn	7396	1530, 1565, 1571, 1591, 1593, 1598,	
\coffin_mark_handle:Nnnn	144, 7396, 7396, 7450	1605, 1606, 1668, 1672, 1702, 1902,	
\coffin_new:c	6844	2151, 2153, 3110, 3192, 3871, 4118,	
\coffin_new:N	141, 6844, 6844, 6854, 6962, 6964,	4208, 6571, 9388, 9481, 9856, 9881,	
6965, 6966, 6967, 7344, 7345, 7346		10445, 10527, 10710, 10742, 11054,	
\coffin_resize:cnn	15348	11151, 11183, 13111, 13380, 13894	
\coffin_resize:Nnn 197, 15348, 15348, 15360		\cs_end: 18, 702, 705, 719, 721, 725, 780,	
\coffin_rotate:cn	15206	1035, 1041, 1063, 1069, 1222, 1289,	
\coffin_rotate:Nn 198, 15206, 15206, 15240		1338, 1469, 1508, 1522, 1524, 1528,	
\coffin_scale:cnn	15375	1529, 1530, 1565, 1571, 1591, 1593,	
\coffin_scale:Nnn 198, 15375, 15375, 15389		1598, 1605, 1606, 1668, 1672, 1702,	
\coffin_set_eq:cc	6953	1902, 2148, 2154, 2155, 2156, 2157,	
\coffin_set_eq:cN	6953	2159, 2161, 2163, 2165, 2167, 2169,	
\coffin_set_eq:Nc	6953	2171, 3110, 3192, 3871, 4118, 4208,	
\coffin_set_eq:NN	141, 6953,	6571, 9388, 9481, 9856, 9889, 10017,	
6953, 6961, 7217, 7236, 7265, 7472		10445, 10527, 10714, 10746, 11054,	
\coffin_set_horizontal_pole:cnn ..	7005	11151, 11186, 13111, 13380, 14053	

<code>\cs_generate_from_arg_count:cNnn . . .</code>	5568, 5569, 5570, 5571, 5580, 5581,
. 1271 , 1280	5582, 5583, 5584, 5585, 5586, 5587,
<code>\cs_generate_from_arg_count:Ncnn . . .</code>	5588, 5589, 5590, 5591, 5616, 5636,
. 1271 , 1282	5665, 5676, 5677, 5687, 5707, 5721,
<code>\cs_generate_from_arg_count:NNnn . . .</code>	5759, 5774, 5815, 5816, 5817, 5818,
. . . . 16 , 1271 , 1271 , 1281 , 1283 , 1301	5832, 5833, 5867, 5868, 5878, 5879,
<code>\cs_generate_variant:Nn</code>	5880, 5881, 5891, 5892, 5893, 5894,
. 27 , 1716 , 1716, 1911, 1920, 1921,	5905, 5930, 5931, 5941, 5942, 5943,
1922, 1923, 1936, 1937, 1990, 1991,	5957, 5958, 5959, 5960, 5961, 5962,
1992, 1993, 2009, 2108, 2109, 2114,	5994, 5995, 6025, 6026, 6031, 6032,
2115, 2274, 2275, 2305, 2306, 2307,	6077, 6078, 6079, 6080, 6081, 6082,
2308, 2327, 2328, 2329, 2330, 3113,	6083, 6084, 6085, 6101, 6134, 6158,
3134, 3146, 3147, 3152, 3153, 3155,	6171, 6208, 6217, 6247, 6288, 6306,
3156, 3158, 3159, 3170, 3171, 3172,	6309, 6312, 6315, 6318, 6354, 6355,
3173, 3182, 3183, 3184, 3185, 3189,	6356, 6357, 6364, 6365, 6384, 6385,
3190, 3709, 3874, 3880, 3883, 3884,	6386, 6387, 6400, 6421, 6422, 6423,
3889, 3890, 3896, 3897, 3899, 3900,	6424, 6425, 6426, 6440, 6442, 6444,
3902, 3903, 3907, 3908, 3912, 3913,	6446, 6463, 6464, 6472, 6473, 6474,
4078, 4105, 4121, 4127, 4130, 4131,	6475, 6498, 6499, 6500, 6501, 6502,
4136, 4137, 4143, 4144, 4146, 4147,	6503, 6504, 6505, 6515, 6516, 6517,
4149, 4150, 4154, 4155, 4159, 4160,	6518, 6519, 6520, 6535, 6536, 6551,
4185, 4192, 4193, 4195, 4211, 4217,	6561, 6574, 6579, 6580, 6585, 6586,
4221, 4222, 4227, 4228, 4234, 4235,	6591, 6592, 6597, 6598, 6604, 6605,
4237, 4238, 4240, 4241, 4245, 4246,	6606, 6613, 6614, 6615, 6618, 6619,
4250, 4251, 4255, 4257, 4277, 4288,	6635, 6636, 6637, 6638, 6639, 6640,
4289, 4294, 4295, 4300, 4301, 4314,	6641, 6642, 6645, 6646, 6647, 6648,
4315, 4351, 4352, 4353, 4354, 4355,	6653, 6654, 6662, 6665, 6668, 6680,
4356, 4373, 4374, 4375, 4376, 4377,	6698, 6699, 6704, 6705, 6710, 6711,
4378, 4379, 4380, 4397, 4398, 4399,	6729, 6730, 6740, 6741, 6746, 6747,
4400, 4401, 4402, 4403, 4404, 4497,	6752, 6753, 6758, 6759, 6774, 6775,
4498, 4499, 4500, 4513, 4514, 4515,	6822, 6823, 6824, 6825, 6843, 6854,
4516, 4559, 4560, 4565, 4566, 4569,	6871, 6901, 6918, 6952, 6961, 7029,
4570, 4571, 4572, 4573, 4574, 4575,	7030, 7031, 7219, 7247, 7343, 7450,
4576, 4585, 4586, 4587, 4588, 4597,	7536, 7573, 8041, 8333, 8498, 8574,
4598, 4599, 4600, 4620, 4621, 4622,	8590, 8622, 8656, 8697, 8826, 8827,
4623, 4642, 4643, 4644, 4651, 4652,	8830, 8838, 8845, 8848, 8856, 8865,
4653, 4696, 4697, 4698, 4699, 4716,	8874, 9108, 9284, 9301, 9352, 9355,
4728, 4744, 4751, 4757, 4769, 4770,	9365, 9366, 9367, 9394, 9412, 9467,
4793, 4794, 4892, 4903, 4904, 4923,	9470, 9487, 9505, 9511, 9514, 9517,
4933, 4950, 4951, 4952, 4953, 5064,	10145, 10148, 11539, 14522, 14577,
5074, 5137, 5138, 5139, 5140, 5141,	14645, 14678, 14682, 14729, 14766,
5142, 5143, 5144, 5172, 5173, 5174,	14773, 14774, 14775, 14778, 14779,
5175, 5257, 5260, 5263, 5266, 5269,	14782, 14783, 14788, 14789, 14796,
5298, 5299, 5300, 5301, 5302, 5303,	14797, 14798, 14799, 14811, 14966,
5339, 5340, 5345, 5346, 5366, 5367,	14999, 15016, 15030, 15048, 15070,
5368, 5369, 5374, 5375, 5376, 5377,	15109, 15152, 15196, 15240, 15360,
5394, 5395, 5420, 5421, 5438, 5439,	15389, 15501, 15524, 15525, 15658,
5448, 5449, 5450, 5451, 5471, 5472,	15659, 15683, 15684, 16041, 16042
5473, 5474, 5475, 5476, 5505, 5518,	<code>\cs_gset:cn</code> 1333
5519, 5534, 5560, 5561, 5566, 5567,	<code>\cs_gset:cpn</code>

- ... [1184](#), [1186](#), [4720](#), [6121](#), [7677](#), [7679](#)
- \cs_gset:cpx [1184](#), [1187](#)
- \cs_gset:cx [1333](#)
- \cs_gset:Nn [15](#), [1284](#)
- \cs_gset:Npn [13](#),
[765](#), [767](#), [1170](#), [1186](#), [5640](#), [6542](#), [15989](#)
- \cs_gset:Npx
... [765](#), [769](#), [1171](#), [1187](#), [5645](#), [15990](#)
- \cs_gset:Nx [1284](#)
- \cs_gset_eq:cc ... [1202](#), [1209](#), [1931](#), [4309](#)
- \cs_gset_eq:cN [1202](#), [1208](#), [1227](#),
[1930](#), [4307](#), [5650](#), [6539](#), [8458](#), [8460](#)
- \cs_gset_eq:Nc
... [1202](#), [1207](#), [1929](#), [4308](#), [5655](#), [6547](#)
- \cs_gset_eq:NN [17](#),
[1202](#), [1206](#), [1207](#), [1208](#), [1209](#), [1219](#),
[1426](#), [1427](#), [1428](#), [1429](#), [1430](#), [1431](#),
[1432](#), [1433](#), [1437](#), [1438](#), [1439](#), [1440](#),
[1441](#), [1442](#), [1443](#), [1444](#), [1917](#), [1919](#),
[1928](#), [1953](#), [1958](#), [1968](#), [4275](#), [4306](#),
[4446](#), [5255](#), [6304](#), [9409](#), [9502](#), [15992](#)
- \cs_gset_nopar:cn [1333](#)
- \cs_gset_nopar:cpn [1176](#), [1180](#)
- \cs_gset_nopar:cpx [1176](#), [1181](#)
- \cs_gset_nopar:cx [1333](#)
- \cs_gset_nopar:Nn [15](#), [1284](#)
- \cs_gset_nopar:Npn [13](#), [765](#), [765](#),
[768](#), [772](#), [776](#), [1168](#), [1180](#), [3404](#), [7717](#)
- \cs_gset_nopar:Npx [765](#), [766](#), [770](#),
[774](#), [778](#), [1169](#), [1181](#), [3411](#), [4281](#),
[4286](#), [4346](#), [4348](#), [4350](#), [4366](#), [4368](#),
[4370](#), [4372](#), [4390](#), [4392](#), [4394](#), [4396](#)
- \cs_gset_nopar:Nx [1284](#)
- \cs_gset_protected:cn [1333](#)
- \cs_gset_protected:cpn [1196](#), [1198](#)
- \cs_gset_protected:cpx [1196](#), [1199](#)
- \cs_gset_protected:cx [1333](#)
- \cs_gset_protected:Nn [16](#), [1284](#)
- \cs_gset_protected:Npn
..... [13](#), [765](#), [775](#), [1174](#), [1198](#), [7648](#)
- \cs_gset_protected:Npx [765](#), [777](#), [1175](#), [1199](#)
- \cs_gset_protected:Nx [1284](#)
- \cs_gset_protected_nopar:cn [1333](#)
- \cs_gset_protected_nopar:cpn [1190](#), [1192](#)
- \cs_gset_protected_nopar:cpx [1190](#), [1193](#)
- \cs_gset_protected_nopar:cx [1333](#)
- \cs_gset_protected_nopar:Nn ... [16](#), [1284](#)
- \cs_gset_protected_nopar:Npn
..... [14](#), [765](#), [771](#), [1172](#), [1192](#)
- \cs_gset_protected_nopar:Npx
..... [765](#), [773](#), [1173](#), [1193](#)
- \cs_gset_protected_nopar:Nx [1284](#)
- \cs_if_eq:ccF [1381](#)
- \cs_if_eq:ccT [1380](#)
- \cs_if_eq:ccTF [1365](#), [1379](#)
- \cs_if_eq:cNF [1373](#)
- \cs_if_eq:cNT [1372](#)
- \cs_if_eq:cNTF [1365](#), [1371](#), [7883](#)
- \cs_if_eq:NcF [1377](#)
- \cs_if_eq:NcT [1376](#)
- \cs_if_eq:NcTF [1365](#), [1375](#)
- \cs_if_eq:NN [1365](#)
- \cs_if_eq:NNF [1373](#), [1377](#), [1381](#)
- \cs_if_eq:NNT [1372](#), [1376](#), [1380](#)
- \cs_if_eq:NNTF [22](#), [1365](#),
[1371](#), [1375](#), [1379](#), [10291](#), [10293](#), [10295](#)
- \cs_if_eq_p:cc [1365](#), [1378](#)
- \cs_if_eq_p:cN [1365](#), [1370](#)
- \cs_if_eq_p:Nc [1365](#), [1374](#)
- \cs_if_eq_p:NN . [22](#), [1365](#), [1370](#), [1374](#), [1378](#)
- \cs_if_exist:c
[1033](#), [2015](#), [3161](#), [3892](#), [4139](#), [4230](#),
[4317](#), [5348](#), [5835](#), [6466](#), [6600](#), [11589](#)
- \cs_if_exist:cF [9099](#)
- \cs_if_exist:cTF [1021](#),
[1086](#), [1088](#), [1090](#), [1092](#), [6816](#), [7635](#),
[8512](#), [8999](#), [9024](#), [9030](#), [9115](#), [10573](#)
- \cs_if_exist:N
[1021](#), [2014](#), [3160](#), [3891](#), [4138](#), [4229](#),
[4316](#), [5347](#), [5834](#), [6465](#), [6599](#), [11588](#)
- \cs_if_exist:NF [1142](#), [1152](#)
- \cs_if_exist:NT [1424](#), [1435](#), [7622](#), [9235](#), [9253](#)
- \cs_if_exist:NTF [22](#), [1021](#),
[1078](#), [1080](#), [1082](#), [1084](#), [1384](#), [6814](#),
[7616](#), [8342](#), [9426](#), [16098](#), [16150](#), [16276](#)
- \cs_if_exist_p:c [1021](#)
- \cs_if_exist_p:N [22](#), [1021](#)
- \cs_if_exist_use:c [1077](#), [1091](#)
- \cs_if_exist_use:cF [1087](#),
[10029](#), [10534](#), [15726](#), [15811](#), [15813](#)
- \cs_if_exist_use:cT [1089](#)
- \cs_if_exist_use:cTF [1077](#), [1085](#)
- \cs_if_exist_use:N [18](#), [1077](#), [1083](#)
- \cs_if_exist_use:NF [1079](#)
- \cs_if_exist_use:NT [1081](#)
- \cs_if_exist_use:NTF [18](#), [1077](#), [1077](#)
- \cs_if_free:c [1061](#)
- \cs_if_free:cT [1897](#)
- \cs_if_free:cTF [1049](#), [7914](#)

<code>\cs_if_free:N</code>	1049	3510, 3512, 3514, 3520, 3526, 3538,
<code>\cs_if_free:NF</code>	1117, 1127	3546, 3558, 3566, 3599, 3632, 3634,
<code>\cs_if_free:NTF</code>	23, 1049, 1858	3636, 3638, 3640, 3645, 3650, 3655,
<code>\cs_if_free_p:c</code>	1049	3675, 3676, 3681, 3686, 3710, 3718,
<code>\cs_if_free_p:N</code>	23, 1049	3720, 3729, 3731, 3740, 3742, 3752,
<code>\cs_meaning:c</code>	722, 723, 733	3761, 3763, 3765, 3781, 3790, 3824,
<code>\cs_meaning:N</code>	17, 709, 711, 730, 1404	3826, 3914, 3919, 3937, 3945, 3947,
<code>\cs_new:cn</code>	1333	3959, 3965, 3988, 3990, 3995, 4000,
<code>\cs_new:cpn</code>	1184,	4005, 4010, 4012, 4075, 4079, 4086,
	1188, 2054, 2056, 2061, 2155, 2156,	4094, 4096, 4179, 4182, 4187, 4190,
	2157, 2158, 2160, 2162, 2164, 2166,	4252, 4493, 4542, 4553, 4601, 4654,
	2168, 2170, 2172, 2177, 2178, 2179,	4655, 4656, 4657, 4667, 4668, 4673,
	2180, 2181, 2182, 2183, 2184, 2185,	4678, 4683, 4688, 4690, 4700, 4703,
	2186, 3235, 3247, 3249, 3251, 3253,	4711, 4750, 4752, 4758, 4763, 4768,
	3255, 3257, 3259, 3980, 3982, 3984,	4771, 4778, 4785, 4787, 4797, 4809,
	3986, 9897, 10404, 10457, 10953,	4817, 4823, 4829, 4833, 4840, 4851,
	10970, 11289, 11318, 11391, 11396,	4860, 4862, 4869, 4875, 4877, 4879,
	11401, 11406, 11764, 11868, 13489	4893, 4895, 4897, 4905, 4910, 4912,
<code>\cs_new:cpx</code>	1184, 1189	4924, 4926, 4980, 4989, 5034, 5041,
<code>\cs_new:cx</code>	1333	5056, 5086, 5092, 5094, 5100, 5101,
<code>\cs_new:Nn</code>	6, 14, 1284	5115, 5123, 5150, 5155, 5160, 5165,
<code>\cs_new:Npn</code> . . .	1, 12, 1160, 1170, 1188,	5170, 5176, 5182, 5187, 5192, 5197,
	1259, 1261, 1408, 1448, 1457, 1458,	5202, 5204, 5211, 5216, 5221, 5226,
	1462, 1463, 1464, 1465, 1466, 1467,	5236, 5243, 5331, 5337, 5365, 5378,
	1468, 1470, 1472, 1484, 1490, 1496,	5433, 5503, 5555, 5592, 5594, 5609,
	1507, 1509, 1516, 1517, 1519, 1521,	5621, 5630, 5678, 5686, 5688, 5708,
	1523, 1525, 1532, 1534, 1539, 1544,	5709, 5710, 5717, 5719, 5807, 5814,
	1550, 1556, 1562, 1568, 1574, 1581,	5836, 5841, 5842, 5849, 5929, 6022,
	1588, 1595, 1602, 1637, 1638, 1643,	6024, 6033, 6040, 6043, 6054, 6060,
	1645, 1650, 1660, 1662, 1664, 1665,	6086, 6095, 6102, 6108, 6115, 6163,
	1667, 1669, 1675, 1681, 1683, 1690,	6181, 6188, 6209, 6210, 6213, 6215,
	1697, 1699, 1701, 1702, 1703, 1705,	6218, 6226, 6241, 6248, 6256, 6258,
	1710, 1784, 1800, 1805, 1814, 1827,	6272, 6277, 6297, 6341, 6388, 6394,
	1842, 1900, 2024, 2034, 2036, 2038,	6483, 6489, 6521, 6527, 7796, 7797,
	2040, 2044, 2063, 2081, 2087, 2093,	7798, 7799, 7800, 7801, 7887, 7912,
	2097, 2098, 2104, 2106, 2110, 2112,	8042, 8295, 8298, 8307, 8312, 8317,
	2116, 2124, 2129, 2137, 2143, 2150,	8322, 8362, 8363, 8367, 8372, 8620,
	2152, 2154, 2207, 2213, 2222, 2227,	8623, 8997, 9015, 9020, 9216, 9690,
	2244, 2250, 2258, 2265, 2276, 2282,	9697, 9734, 9735, 9736, 9737, 9738,
	2348, 2350, 2352, 2359, 2429, 2435,	9739, 9740, 9741, 9742, 9743, 9763,
	2441, 2447, 2578, 2633, 2651, 2675,	9764, 9765, 9771, 9777, 9786, 9788,
	2693, 2711, 2729, 2740, 2761, 2780,	9799, 9800, 9801, 9811, 9821, 9831,
	2787, 2788, 2796, 2805, 2814, 2830,	9841, 9851, 9854, 9864, 9868, 9873,
	2902, 2998, 3001, 3010, 3019, 3040,	9875, 9877, 9879, 9891, 9893, 9895,
	3042, 3048, 3066, 3074, 3082, 3095,	9921, 9923, 9925, 9926, 9927, 9929,
	3097, 3104, 3192, 3199, 3213, 3218,	9931, 9933, 9935, 9937, 9952, 9953,
	3224, 3240, 3269, 3274, 3279, 3284,	9964, 9978, 9985, 9987, 9994, 10013,
	3289, 3291, 3314, 3322, 3330, 3336,	10132, 10133, 10134, 10135, 10136,
	3342, 3350, 3358, 3364, 3370, 3377,	10137, 10138, 10146, 10187, 10189,
	3391, 3425, 3426, 3440, 3446, 3478,	10198, 10199, 10208, 10222, 10238,

10249, 10258, 10265, 10276, 10289,
 10300, 10309, 10320, 10325, 10345,
 10347, 10358, 10363, 10376, 10384,
 10385, 10395, 10400, 10422, 10443,
 10452, 10462, 10481, 10491, 10493,
 10496, 10498, 10504, 10512, 10532,
 10543, 10563, 10571, 10586, 10601,
 10612, 10622, 10632, 10637, 10646,
 10663, 10676, 10681, 10687, 10689,
 10696, 10726, 10754, 10770, 10781,
 10786, 10804, 10822, 10833, 10848,
 10853, 10863, 10873, 10883, 10899,
 10946, 10981, 10993, 11049, 11058,
 11079, 11104, 11121, 11123, 11133,
 11136, 11146, 11147, 11154, 11161,
 11192, 11207, 11208, 11212, 11228,
 11246, 11248, 11265, 11291, 11305,
 11333, 11345, 11360, 11374, 11416,
 11421, 11432, 11447, 11458, 11471,
 11490, 11508, 11510, 11537, 11540,
 11550, 11561, 11564, 11595, 11614,
 11619, 11646, 11647, 11655, 11667,
 11673, 11679, 11687, 11695, 11701,
 11707, 11715, 11723, 11734, 11758,
 11760, 11762, 11775, 11784, 11785,
 11810, 11815, 11822, 11823, 11831,
 11842, 11844, 11851, 11853, 11861,
 11891, 11893, 11903, 11923, 11932,
 11946, 11954, 11962, 11969, 11976,
 11984, 11994, 12008, 12019, 12020,
 12026, 12043, 12050, 12052, 12059,
 12064, 12081, 12082, 12083, 12101,
 12107, 12117, 12129, 12136, 12158,
 12196, 12205, 12224, 12226, 12228,
 12237, 12248, 12275, 12288, 12301,
 12309, 12326, 12343, 12350, 12358,
 12368, 12369, 12378, 12379, 12388,
 12398, 12412, 12422, 12433, 12441,
 12443, 12454, 12460, 12495, 12516,
 12518, 12520, 12522, 12529, 12538,
 12543, 12550, 12557, 12577, 12582,
 12599, 12614, 12615, 12620, 12628,
 12629, 12652, 12665, 12672, 12680,
 12681, 12682, 12685, 12693, 12699,
 12701, 12703, 12725, 12730, 12740,
 12750, 12760, 12773, 12784, 12789,
 12796, 12805, 12807, 12816, 12825,
 12839, 12841, 12843, 12856, 12866,
 12871, 12880, 12888, 12895, 12901,
 12910, 12912, 12924, 12929, 12937,
 12942, 12952, 12958, 12964, 12971,
 12978, 12980, 12985, 12987, 12992,
 12993, 13007, 13017, 13029, 13034,
 13041, 13060, 13074, 13088, 13108,
 13121, 13123, 13128, 13141, 13146,
 13154, 13159, 13169, 13181, 13210,
 13211, 13212, 13214, 13216, 13218,
 13232, 13238, 13247, 13266, 13272,
 13282, 13301, 13309, 13348, 13350,
 13359, 13361, 13375, 13377, 13386,
 13388, 13404, 13420, 13436, 13452,
 13468, 13484, 13514, 13533, 13561,
 13577, 13587, 13598, 13619, 13634,
 13639, 13644, 13646, 13660, 13662,
 13677, 13685, 13693, 13717, 13732,
 13747, 13762, 13777, 13792, 13807,
 13822, 13830, 13844, 13846, 13852,
 13864, 13872, 13879, 14055, 14063,
 14074, 14075, 14077, 14078, 14089,
 14096, 14098, 14104, 14125, 14132,
 14139, 14154, 14193, 14206, 14247,
 14261, 14280, 14282, 14299, 14314,
 14326, 14333, 14338, 14340, 14349,
 14362, 14365, 14386, 14399, 14414,
 14432, 14447, 14457, 14466, 14479,
 14495, 14512, 14517, 14518, 14519,
 14520, 14533, 14560, 14573, 14575,
 14583, 14609, 14634, 14643, 14644,
 14651, 14661, 14681, 14688, 14693,
 14700, 14714, 14719, 14721, 14731,
 14733, 14735, 14737, 15487, 15493,
 15503, 15505, 15511, 15517, 15547,
 15567, 15580, 15586, 15591, 15603,
 15604, 15605, 15617, 15625, 15633,
 15640, 15641, 15649, 15704, 15709,
 15720, 15740, 15745, 15750, 15756,
 15769, 15779, 15789, 15794, 15805,
 15832, 15845, 15849, 15854, 15859,
 15875, 15888, 15896, 15913, 15927,
 15942, 15952, 15961, 15979, 16140
 \cs_new:Npx . . . 1160, 1171, 1189, 6172,
 6182, 8283, 9559, 14676, 14679, 14749
 \cs_new:Nx 1284
 \cs_new_eq:cc 923, 1202, 1217
 \cs_new_eq:cN 1202, 1215, 10945
 \cs_new_eq:Nc 1202, 1216
 \cs_new_eq:NN . . . 16, 1202, 1210, 1215,
 1216, 1217, 1412, 1413, 1414, 1415,
 1416, 1417, 1418, 1419, 1420, 1421,
 1422, 1423, 1447, 1456, 1478, 1910,

- 1924, 1925, 1926, 1927, 1928, 1929,
 1930, 1931, 2344, 2451, 2452, 2453,
 2823, 2824, 2825, 3031, 3032, 3033,
 3034, 3035, 3137, 3141, 3191, 3297,
 3827, 3828, 3854, 3855, 3856, 3857,
 3858, 3859, 3860, 3864, 3865, 3866,
 4018, 4077, 4104, 4184, 4186, 4189,
 4194, 4254, 4256, 4266, 4267, 4268,
 4302, 4303, 4304, 4305, 4306, 4307,
 4308, 4309, 4702, 4749, 5081, 5082,
 5210, 5237, 5238, 5239, 5250, 5270,
 5271, 5272, 5273, 5274, 5275, 5276,
 5277, 5722, 5723, 5724, 5725, 5726,
 5727, 5728, 5729, 5730, 5731, 5732,
 5733, 5734, 5735, 5736, 5737, 5738,
 5739, 5740, 5741, 5742, 5743, 5744,
 5745, 5746, 5747, 5767, 5770, 5771,
 5775, 5776, 5777, 5778, 5779, 5780,
 5781, 5782, 5783, 5784, 5785, 5786,
 5787, 5788, 5789, 5790, 5963, 5964,
 5965, 5966, 5967, 5968, 5969, 5970,
 5971, 5972, 5973, 5974, 5975, 5976,
 5977, 5978, 6319, 6320, 6321, 6322,
 6323, 6324, 6325, 6326, 6562, 6563,
 6601, 6602, 6603, 6616, 6617, 6628,
 6629, 6630, 6712, 6713, 6714, 6715,
 6716, 6717, 6718, 6719, 6727, 6728,
 6765, 6766, 6767, 6768, 6769, 6770,
 6771, 6772, 6773, 6968, 6969, 6970,
 6971, 6972, 6973, 7602, 9340, 9351,
 9423, 9453, 9454, 9466, 9508, 9523,
 9608, 9860, 10023, 10024, 10025,
 10026, 10221, 10274, 10275, 14728,
 14730, 14765, 14776, 14777, 15787,
 15788, 16100, 16101, 16229, 16252
 \cs_new_nopar:cn [1333](#)
 \cs_new_nopar:cpn [1176](#),
 1182, 2072, 2073, 2074, 2075, 2076,
 2077, 2078, 2079, 11011, 11022,
 11041, 11093, 11095, 12150, 12260
 \cs_new_nopar:cpx [1176](#), 1183, 1888, 11863
 \cs_new_nopar:cx [1333](#)
 \cs_new_nopar:Nn [14](#), [1284](#)
 \cs_new_nopar:Npn [12](#),
[1160](#), 1168, 1182, 1269, 1370, 1371,
 1372, 1373, 1374, 1375, 1376, 1377,
 1378, 1379, 1380, 1381, 1446, 1610,
 1611, 1612, 1613, 1614, 1615, 1616,
 1617, 1618, 1625, 1626, 1627, 1628,
 1629, 1692, 1693, 1694, 1695, 2195,
 2197, 2827, 2828, 2829, 2880, 2888,
 2890, 2892, 2912, 3660, 3661, 3662,
 3663, 3664, 3665, 3666, 3667, 3668,
 3669, 3670, 3671, 3672, 3673, 3674,
 4543, 4548, 4709, 4745, 4747, 4925,
 4934, 5063, 5469, 5617, 5619, 6159,
 6161, 6552, 6554, 7716, 9522, 9688,
 10143, 10421, 11067, 11069, 11071,
 11073, 11075, 11077, 11081, 11083,
 11085, 11098, 11100, 11102, 11526,
 11782, 12683, 12684, 14115, 14237,
 14242, 14523, 14578, 14646, 14683,
 15455, 15457, 15698, 15699, 15700,
 15701, 15702, 15703, 15873, 15894
 \cs_new_nopar:Npx
 [1160](#), 1169, 1183, 1743, 13891
 \cs_new_nopar:Nx [1284](#)
 \cs_new_protected:cn [1333](#)
 \cs_new_protected:cpn
 [1196](#), 1200, 7811, 7812,
 8056, 8698, 8700, 8702, 8704, 8706,
 8708, 8710, 8712, 8716, 8718, 8720,
 8722, 8724, 8726, 8728, 8730, 8732,
 8734, 8736, 8738, 8740, 8742, 8744,
 8746, 8748, 8750, 8752, 8754, 8756,
 8758, 8760, 8762, 8764, 8766, 8768,
 8770, 8772, 8774, 8776, 8778, 8782,
 8784, 8786, 8788, 8790, 8792, 8794,
 8796, 8798, 8800, 8802, 8804, 8806,
 8808, 8810, 8812, 9109, 9111, 9146
 \cs_new_protected:cpx
[1196](#), 1201, 7821, 7823, 7825, 7827,
 7829, 7838, 7840, 7842, 8065, 8067,
 8069, 8071, 8073, 8082, 8084, 8086
 \cs_new_protected:cx [1333](#)
 \cs_new_protected:Nn [14](#), [1284](#)
 \cs_new_protected:Npn [12](#),
[1160](#), 1174, 1200, 1202, 1210, 1218,
 1220, 1271, 1292, 1297, 1382, 1399,
 1479, 1655, 1716, 1734, 1747, 1749,
 1755, 1764, 1854, 1880, 1893, 1910,
 1912, 1914, 1916, 1918, 1932, 1934,
 1994, 2003, 2237, 2335, 2357, 2361,
 2363, 2365, 2367, 2369, 2371, 2373,
 2375, 2377, 2379, 2381, 2383, 2385,
 2387, 2389, 2391, 2393, 2395, 2397,
 2399, 2401, 2403, 2405, 2407, 2409,
 2411, 2413, 2415, 2417, 2419, 2421,
 2423, 2425, 2427, 2431, 2433, 2437,
 2439, 2443, 2445, 2449, 2451, 2835,

2841, 2858, 2860, 2862, 2876, 2878,
 3107, 3114, 3144, 3145, 3148, 3150,
 3154, 3157, 3162, 3164, 3174, 3176,
 3186, 3407, 3419, 3829, 3868, 3875,
 3881, 3882, 3885, 3887, 3893, 3895,
 3898, 3901, 3904, 3906, 3909, 3911,
 4106, 4115, 4122, 4128, 4129, 4132,
 4134, 4140, 4142, 4145, 4148, 4151,
 4153, 4156, 4158, 4196, 4205, 4212,
 4218, 4220, 4223, 4225, 4231, 4233,
 4236, 4239, 4242, 4244, 4247, 4249,
 4258, 4272, 4278, 4283, 4290, 4292,
 4296, 4298, 4310, 4312, 4339, 4341,
 4343, 4345, 4347, 4349, 4357, 4359,
 4361, 4363, 4365, 4367, 4369, 4371,
 4381, 4383, 4385, 4387, 4389, 4391,
 4393, 4395, 4473, 4501, 4503, 4517,
 4555, 4557, 4561, 4563, 4717, 4726,
 4729, 4737, 4789, 4791, 4899, 4901,
 5065, 5075, 5252, 5258, 5261, 5264,
 5267, 5278, 5283, 5288, 5293, 5308,
 5341, 5343, 5349, 5357, 5370, 5372,
 5380, 5382, 5384, 5396, 5398, 5400,
 5426, 5477, 5485, 5495, 5510, 5512,
 5520, 5529, 5539, 5637, 5642, 5647,
 5659, 5666, 5754, 5769, 5772, 5791,
 5793, 5795, 5823, 5863, 5865, 5873,
 5886, 5895, 5903, 5910, 5918, 5948,
 5980, 5982, 5984, 5996, 5998, 6000,
 6027, 6029, 6070, 6116, 6129, 6135,
 6146, 6151, 6278, 6283, 6301, 6307,
 6310, 6313, 6316, 6331, 6333, 6342,
 6348, 6358, 6366, 6375, 6429, 6452,
 6537, 6556, 6568, 6575, 6577, 6581,
 6583, 6587, 6589, 6593, 6595, 6607,
 6609, 6611, 6620, 6622, 6624, 6626,
 6649, 6651, 6660, 6666, 6669, 6681,
 6695, 6696, 6697, 6700, 6702, 6706,
 6708, 6720, 6722, 6723, 6725, 6731,
 6732, 6733, 6735, 6737, 6739, 6742,
 6744, 6748, 6750, 6754, 6756, 6760,
 6776, 6826, 6835, 6844, 6855, 6872,
 6902, 6919, 6953, 6974, 6984, 6991,
 6998, 7005, 7016, 7027, 7032, 7043,
 7077, 7092, 7170, 7183, 7220, 7238,
 7248, 7267, 7272, 7286, 7291, 7301,
 7312, 7324, 7336, 7396, 7443, 7451,
 7480, 7529, 7537, 7561, 7638, 7659,
 7664, 7666, 7673, 7675, 7682, 7723,
 7735, 7740, 7757, 7781, 7787, 7881,
 7922, 7943, 7956, 7961, 7987, 7998,
 8000, 8017, 8043, 8045, 8047, 8049,
 8327, 8334, 8336, 8338, 8340, 8352,
 8354, 8393, 8406, 8415, 8434, 8436,
 8442, 8444, 8455, 8490, 8492, 8499,
 8504, 8509, 8522, 8532, 8542, 8557,
 8559, 8575, 8635, 8651, 8657, 8662,
 8671, 8673, 8675, 8680, 8682, 8691,
 8820, 8831, 8839, 8849, 8857, 8866,
 8875, 8880, 8885, 8963, 9034, 9097,
 9113, 9127, 9180, 9221, 9223, 9231,
 9264, 9269, 9278, 9285, 9302, 9304,
 9309, 9351, 9353, 9356, 9368, 9378,
 9395, 9401, 9415, 9440, 9442, 9466,
 9468, 9471, 9488, 9494, 9509, 9512,
 9515, 9558, 9565, 9610, 9655, 9745,
 10011, 10027, 10047, 10081, 10107,
 10115, 10117, 11517, 11524, 14292,
 14764, 14767, 14769, 14771, 14780,
 14781, 14784, 14786, 14794, 14800,
 14809, 14835, 14847, 14881, 14892,
 14903, 14914, 14925, 14936, 14949,
 14967, 14974, 14983, 15000, 15017,
 15031, 15049, 15071, 15107, 15110,
 15153, 15206, 15241, 15253, 15259,
 15265, 15277, 15296, 15305, 15322,
 15327, 15335, 15348, 15361, 15375,
 15390, 15397, 15403, 15412, 15419,
 15428, 15433, 15441, 15446, 15469,
 15476, 15530, 15540, 15660, 15685,
 16020, 16024, 16037, 16039, 16110,
 16129, 16152, 16156, 16161, 16164
 \cs_new_protected:Npx .. 1160, 1175, 1201
 \cs_new_protected:Nx 1284
 \cs_new_protected_nopar:cn 1333
 \cs_new_protected_nopar:cpn
 .. 1190, 1194, 8714, 8780, 8814, 8816
 \cs_new_protected_nopar:cpx
 .. 1190, 1195, 1286, 1335, 1885, 2940
 \cs_new_protected_nopar:cx 1333
 \cs_new_protected_nopar:Nn 14, 1284
 \cs_new_protected_nopar:Npn 12, 1160,
 1172, 1177, 1194, 1203, 1204, 1205,
 1206, 1207, 1208, 1209, 1215, 1216,
 1217, 1280, 1282, 1391, 1410, 1609,
 1619, 1620, 1621, 1622, 1623, 1624,
 1630, 1631, 1632, 1633, 1634, 1635,
 1636, 1696, 2199, 2831, 2833, 2921,
 3166, 3168, 3178, 3180, 3188, 3193,
 3400, 4467, 4469, 4471, 4505, 4507,

- 4509, 4511, 4639, 4640, 4641, 4735,
5304, 5306, 5422, 5424, 5506, 5508,
5535, 5537, 5653, 5819, 5821, 5869,
5871, 5882, 5884, 5906, 5908, 6427,
6428, 6448, 6450, 6663, 6917, 6951,
7603, 7609, 7613, 7942, 7996, 8591,
8593, 8595, 8609, 8631, 8633, 8818,
8828, 8846, 8894, 8918, 8937, 8983,
8985, 9006, 9314, 9413, 9506, 9519,
9521, 9617, 9627, 9643, 9662, 9676,
9682, 10041, 10043, 10045, 10075,
10077, 10079, 10101, 10103, 10105,
10109, 10111, 10113, 14790, 14791,
14792, 14793, 15316, 15459, 15461,
15463, 15526, 15528, 15536, 15538,
15654, 15656, 15679, 15681, 16003,
16031, 16033, 16035, 16082, 16089,
16104, 16106, 16194, 16230, 16267,
16272, 16278, 16284, 16288, 16293
- \cs_new_protected_nopar:Npx . . . 1160,
1173, 1195, 1737, 1741, 1884, 9670
- \cs_new_protected_nopar:Nx 1284
- \cs_set:cn 1333
- \cs_set:cpn
. 1184, 1184, 7668, 7670, 8654, 10120
- \cs_set:cpx 1184, 1185
- \cs_set:cx 1333
- \cs_set:Nn 15, 1284
- \cs_set:Npn . . . 3, 12, 751, 753, 780, 786,
787, 788, 789, 790, 791, 792, 793,
794, 795, 796, 797, 798, 799, 800,
801, 802, 803, 804, 805, 806, 807,
808, 809, 810, 811, 812, 813, 814,
967, 972, 977, 982, 995, 996, 1004,
1012, 1014, 1017, 1019, 1077, 1079,
1081, 1083, 1085, 1087, 1089, 1091,
1160, 1176, 1184, 1284, 1333, 2932,
2938, 3037, 3050, 3058, 3921, 3929,
4019, 4027, 4035, 4041, 4047, 4055,
4063, 4069, 4647, 4795, 5104, 6002,
6072, 6335, 10050, 10058, 10067,
10084, 10092, 15471, 15977, 15987
- \cs_set:Npx . . . 751, 755, 1185, 4526, 15988
- \cs_set:Nx 1284
- \cs_set_eq:cc . 921, 1202, 1205, 1927, 4305
- \cs_set_eq:cN 1202, 1203, 1926, 4303, 10012
- \cs_set_eq:Nc 1202, 1204, 1925, 4304
- \cs_set_eq:NN 17, 1202, 1202, 1203, 1204,
1205, 1206, 1213, 1737, 1752, 1884,
1913, 1915, 1924, 1943, 1948, 1963,
2512, 2839, 2843, 2864, 2866, 2943,
4302, 4440, 5428, 5429, 5431, 5541,
5542, 5553, 9575, 9576, 9577, 15991
- \cs_set_nopar:cn 1333
- \cs_set_nopar:cpn 1176, 1178
- \cs_set_nopar:cpx 1176, 1179
- \cs_set_nopar:cx 1333
- \cs_set_nopar:Nn 15, 1284
- \cs_set_nopar:Npn 13, 751, 751,
753, 754, 755, 757, 758, 759, 762,
779, 815, 817, 989, 1113, 1178, 11256
- \cs_set_nopar:Npx
. 751, 752, 756, 760, 764, 783,
1179, 1481, 1657, 2845, 2850, 2867,
2868, 4340, 4342, 4344, 4358, 4360,
4362, 4364, 4382, 4384, 4386, 4388,
9184, 9569, 9570, 9571, 9572, 9573
- \cs_set_nopar:Nx 1284
- \cs_set_protected:cn 1333
- \cs_set_protected:cpn 1196, 1196
- \cs_set_protected:cpx 1196, 1197
- \cs_set_protected:cx 1333
- \cs_set_protected:Nn 15, 1284
- \cs_set_protected:Npn 13, 751, 761, 781,
827, 840, 845, 857, 872, 889, 905,
910, 915, 924, 936, 950, 1097, 1109,
1111, 1115, 1125, 1140, 1150, 1162,
1196, 1229, 1250, 1940, 1945, 1950,
1955, 1960, 1965, 1970, 1975, 4171,
4407, 4414, 4436, 4442, 4450, 4457,
5457, 6930, 7808, 8052, 8054, 10402,
10951, 11009, 11020, 11263, 15676
- \cs_set_protected:Npx . . . 751, 763, 1197
- \cs_set_protected:Nx 1284
- \cs_set_protected_nopar:cn 1333
- \cs_set_protected_nopar:cpn . 1190, 1190
- \cs_set_protected_nopar:cpx . 1190, 1191
- \cs_set_protected_nopar:cx 1333
- \cs_set_protected_nopar:Nn 15, 1284
- \cs_set_protected_nopar:Npn
. 13, 180, 751, 757, 761, 763, 767,
769, 771, 773, 775, 777, 819, 821,
823, 825, 832, 834, 836, 838, 920,
922, 1093, 1095, 1136, 1158, 1190,
6907, 7618, 7623, 7767, 9518, 9520
- \cs_set_protected_nopar:Npx
. 166, 751, 759, 1191, 7929
- \cs_set_protected_nopar:Nx 1284
- \cs_show:c 1393, 1410, 9035
- \cs_show:N 17, 1393, 1399, 1411, 5068

<code>\cs_to_str:N</code>	4,	<code>\dim_compare:nT</code>	4021, 4038
19, 989, 989, 1008, 2211, 9523, 11520		<code>\dim_compare:nTF</code>	78, 3954
<code>\cs_undefine:c</code>	1218, 1220	<code>\dim_compare_p:n</code>	78, 3954
<code>\cs_undefine:N</code>		<code>\dim_compare_p:nNn</code>	77, 3949
.... 17, 1218, 1218, 8090, 8091, 8092		<code>\dim_const:cn</code>	3875
<code>acsc</code>	189	<code>\dim_const:Nn</code>	
<code>acscd</code>	189 75, 3875, 3875, 3880, 4108, 4109	
<code>\csname</code> 14, 20, 24, 74, 75, 98, 128, 149, 307		<code>\dim_do_until:nn</code> ...	80, 4019, 4041, 4045
<code>\currentgrouplevel</code>	559	<code>\dim_do_until:nNnn</code> ..	79, 4047, 4069, 4073
<code>\currentgrouptype</code>	560	<code>\dim_do_while:nn</code> ...	80, 4019, 4035, 4039
<code>\currentifbranch</code>	556	<code>\dim_do_while:nNnn</code> ..	79, 4047, 4063, 4067
<code>\currentiflevel</code>	555	<code>\dim_eval:n</code>	80, 3993, 3998, 4003,
<code>\currentiftype</code>	557	4008, 4075, 4075, 6894, 6942, 7011,	
		7022, 7039, 7041, 7047, 7058, 7072,	
		7297, 7298, 15168, 15189, 15331,	
		15332, 15339, 15340, 15416, 15423	
		<code>\dim_gadd:cn</code>	3904
		<code>\dim_gadd:Nn</code>	76, 3904, 3906, 3908
		<code>.dim_gset:c</code>	159
		<code>\dim_gset:cn</code>	3893
		<code>.dim_gset:N</code>	159
		<code>\dim_gset:Nn</code> ...	76, 3878, 3893, 3895, 3897
		<code>\dim_gset_eq:cc</code>	3898
		<code>\dim_gset_eq:cN</code>	3898
		<code>\dim_gset_eq:Nc</code>	3898
		<code>\dim_gset_eq:NN</code> 76, 3898, 3901, 3902, 3903	
		<code>\dim_gsub:cn</code>	3904
		<code>\dim_gsub:Nn</code>	76, 3904, 3911, 3913
		<code>\dim_gzero:c</code>	3881
		<code>\dim_gzero:N</code> ...	75, 3881, 3882, 3884, 3888
		<code>\dim_gzero_new:c</code>	3885
		<code>\dim_gzero_new:N</code> ...	75, 3885, 3887, 3890
		<code>\dim_if_exist:c</code>	3892
		<code>\dim_if_exist:cTF</code>	3891
		<code>\dim_if_exist:N</code>	3891
		<code>\dim_if_exist:NTF</code> ..	75, 3886, 3888, 3891
		<code>\dim_if_exist_p:c</code>	3891
		<code>\dim_if_exist_p:N</code>	75, 3891
		<code>\dim_max:nn</code> ...	76, 3914, 3921, 15310, 15314
		<code>\dim_min:nn</code>	
	 76, 3914, 3929, 15308, 15312, 15325	
		<code>\dim_new:c</code>	3867
		<code>\dim_new:N</code>	75, 3867, 3868,
		3874, 3877, 3886, 3888, 4110, 4111,	
		4112, 4113, 6782, 6804, 6805, 6808,	
		6809, 6810, 6811, 7384, 7386, 7387,	
		14826, 14827, 14828, 14829, 14830,	
		14831, 14832, 14833, 15201, 15202,	
		15203, 15204, 15205, 15346, 15347	
		<code>\dim_ratio:nn</code>	77, 3945, 3945, 4101
<code>\cs_to_str:N</code>	4,		
19, 989, 989, 1008, 2211, 9523, 11520			
<code>\cs_undefine:c</code>	1218, 1220		
<code>\cs_undefine:N</code>			
.... 17, 1218, 1218, 8090, 8091, 8092			
<code>acsc</code>	189		
<code>acscd</code>	189		
<code>\csname</code> 14, 20, 24, 74, 75, 98, 128, 149, 307			
<code>\currentgrouplevel</code>	559		
<code>\currentgrouptype</code>	560		
<code>\currentifbranch</code>	556		
<code>\currentiflevel</code>	555		
<code>\currentiftype</code>	557		
D			
<code>dd</code>	191		
<code>nd</code>	191		
<code>\day</code>	515		
<code>\deadcycles</code>	449		
<code>\def</code> 30, 61, 62, 63, 79, 97, 99, 101,			
122, 123, 125, 134, 138, 162, 201, 214			
<code>.default:n</code>	159		
<code>.default:o</code>	159		
<code>.default:V</code>	159		
<code>.default:x</code>	159		
<code>\defaultthyphenchar</code>	499		
<code>\defaultskewchar</code>	500		
<code>\delcode</code>	530		
<code>\delimiter</code>	324		
<code>\delimiterfactor</code>	373		
<code>\delimitershortfall</code>	372		
<code>\detokenize</code>	149, 547		
<code>\dim_abs:n</code>	76, 3914, 3914, 14976		
<code>\dim_add:cn</code>	3904		
<code>\dim_add:Nn</code> 76, 3904, 3904, 3906, 3907			
<code>\dim_case:nn</code>	3990, 4005		
<code>\dim_case:nnF</code>	4000, 4266		
<code>\dim_case:nnn</code>	4266, 4266		
<code>\dim_case:nnT</code>	3995		
<code>\dim_case:nnTF</code>	79, 3990, 3990		
<code>\dim_compare:n</code>	3954		
<code>\dim_compare:nF</code>	4029, 4044		
<code>\dim_compare:nNn</code>	3949		
<code>\dim_compare:nNnF</code>	4057, 4072		
<code>\dim_compare:nNnT</code>			
.... 4049, 4066, 7189, 7194, 15182			
<code>\dim_compare:nNnTF</code>			
.... 77, 3949, 4014, 7095, 7098, 7101,			
7110, 7113, 7116, 7125, 7132, 7201,			
7314, 7326, 15118, 15135, 15161, 15175			

\dim_set:c	159	\dim_while_do:nNnn	80, 4047, 4047, 4052
\dim_set:cn	3893	\dim_zero:c	3881
.dim_set:N	159	\dim_zero:N	75, 3881, 3881, 3882, 3883, 3886, 7088, 7089, 14852, 14972, 15059
\dim_set:Nn	76, 3893, 3893, 3895, 3896, 6878, 6924, 7097, 7102, 7112, 7117, 7127, 7134, 7147, 7172, 7192, 7251, 7252, 7254, 7256, 7274, 7275, 7385, 7488, 7489, 7540, 7541, 7542, 7544, 14849, 14850, 14851, 14883, 14894, 14969, 14970, 14971, 14976, 14977, 14979, 15056, 15057, 15058, 15060, 15062, 15064, 15247, 15279, 15287, 15298, 15299, 15300, 15301, 15307, 15309, 15311, 15313, 15318, 15324, 15380, 15382, 15384, 15392, 15394	\dim_zero_new:c	3885
\dim_set_eq:cc	3898	\dim_zero_new:N	75, 3885, 3885, 3889
\dim_set_eq:cN	3898	\dimen	521
\dim_set_eq:Nc	3898	\dimendef	220
\dim_set_eq:NN	76, 3898, 3898, 3899, 3900, 6880, 6881, 6926, 6927	\dimexpr	574
\dim_show:c	4104	\directlua	16, 43, 45, 623
\dim_show:N	82, 4104, 4104, 4105	\discretionary	384
\dim_show:n	82, 4106, 4106	\displayindent	349
\dim_sub:cn	3904	\displaylimits	359
\dim_sub:Nn	76, 3904, 3909, 3911, 3912	\displaystyle	337
\dim_to_decimal:n	81, 4079, 4079, 4095, 4098, 4268	\displaywidowpenalties	587
\dim_to_decimal_in_bp:n	81, 4094, 4094, 4267, 16173, 16174, 16175, 16184, 16185, 16186	\displaywidowpenalty	348
\dim_to_decimal_in_unit:nn	81, 4096, 4096	\displaywidth	350
\dim_to_fp:n	82, 4104, 7140, 7142, 7152, 7153, 7154, 7155, 7176, 7177, 7178, 7179, 14693, 14693, 14887, 14888, 14898, 14899, 14956, 14959, 14960, 14991, 14992, 15008, 15009, 15024, 15038, 15041, 15042, 15283, 15284, 15291, 15292, 15351, 15354, 15355, 15393, 15395, 16038, 16040	\divide	227
\dim_until_do:nn	80, 4019, 4027, 4032	\doublehyphendemerits	417
\dim_until_do:nNnn	80, 4047, 4055, 4060	\dp	528
\dim_use:c	4077	\dump	511
\dim_use:N	81, 3917, 3923, 3924, 3925, 3931, 3932, 3933, 3957, 3978, 4076, 4077, 4077, 4078, 4082, 4107, 7035, 7037, 7041, 7052, 7065, 7282, 7595, 7596, 7597, 15244, 15246, 15249, 15251, 15257, 15263, 15272, 15273, 15274, 15401, 15408	E	
\dim_while_do:nn	80, 4019, 4019, 4024	\E	15999
		sec	188
		secd	189
		\edef	80, 108, 136, 215
		deg	190
		ceil	188
		\else	15, 25, 27, 76, 268
		\else:	23, 688, 691, 727, 893, 1025, 1028, 1037, 1043, 1053, 1056, 1065, 1071, 1224, 1245, 1254, 1265, 1368, 1502, 1738, 1788, 1789, 1790, 1846, 1849, 1986, 2020, 2049, 2068, 2188, 2190, 2192, 2194, 2254, 2270, 2293, 2301, 2314, 2323, 2494, 2499, 2504, 2509, 2516, 2522, 2527, 2532, 2537, 2542, 2547, 2552, 2557, 2562, 2582, 2590, 2596, 2599, 2638, 2641, 2660, 2663, 2680, 2683, 2698, 2701, 2716, 2719, 2792, 2801, 2809, 2818, 2884, 2898, 2907, 2917, 2927, 3049, 3070, 3086, 3089, 3238, 3265, 3302, 3310, 3596, 3629, 3920, 3941, 3952, 3962, 3989, 4167, 4581, 4593, 4606, 4616, 4632, 4663, 4917, 4946, 4969, 4985, 4993, 5003, 5025, 5127, 5135, 5148, 5444, 5481, 5490, 5899, 5914, 5936, 5952, 6493, 6632, 6634, 6644, 9430,

9433, 9701, 9781, 9790, 9791, 9792, 9805, 9815, 9825, 9885, 9948, 9959, 9971, 9974, 10019, 10212, 10240, 10241, 10282, 10315, 10333, 10368, 10372, 10408, 10428, 10432, 10436, 10523, 10549, 10558, 10591, 10595, 10607, 10617, 10627, 10658, 10671, 10706, 10716, 10735, 10748, 10761, 10765, 10776, 10799, 10816, 10828, 10842, 10858, 10866, 10868, 10878, 10889, 10905, 10920, 10926, 10931, 10938, 10962, 10988, 11108, 11111, 11169, 11173, 11181, 11200, 11219, 11234, 11238, 11274, 11299, 11308, 11323, 11339, 11351, 11368, 11384, 11426, 11437, 11444, 11477, 11479, 11484, 11499, 11599, 11610, 11631, 11634, 11637, 11640, 11651, 11660, 11663, 11727, 11740, 11743, 11750, 11768, 11782, 11799, 11874, 11877, 11885, 11897, 11908, 11914, 11927, 11940, 11980, 12014, 12034, 12071, 12089, 12092, 12097, 12111, 12146, 12164, 12167, 12170, 12173, 12232, 12305, 12373, 12374, 12383, 12418, 12501, 12505, 12509, 12571, 12820, 12849, 12853, 13012, 13021, 13069, 13080, 13096, 13104, 13163, 13242, 13253, 13258, 13292, 13305, 13321, 13325, 13328, 13499, 13506, 13528, 13551, 13566, 13570, 13592, 13623, 13626, 13651, 13654, 13690, 13698, 13703, 13709, 13712, 13722, 13725, 13743, 13758, 13773, 13788, 13803, 13818, 13839, 13884, 14110, 14148, 14149, 14158, 14202, 14254, 14268, 14269, 14270, 14372, 14394, 14409, 14427, 14475, 14491, 14565, 14658, 14665, 14669, 14704, 14709, 16016	\errorcontextlines 289 \errorstopmode 303 \escapechar 321 \etex_beginL:D 594 \etex_beginR:D 596 \etex_botmarks:D 543 \etex_clubpenalties:D 585 \etex_currentgrouplevel:D 559 \etex_currentgroupstype:D 566 \etex_currentifbranch:D 556 \etex_currentiflevel:D 555 \etex_currentiftypetype:D 557 \etex_detokenize:D 547, 868, 932, 1724, 4749, 4750, 5119 \etex_dimexpr:D 574, 3865 \etex_displaywidowpenalties:D 587 \etex_endL:D 595 \etex_endR:D 597 \etex_eTeXrevision:D 539 \etex_eTeXversion:D 538 \etex_everyeof:D . 599, 4476, 15665, 15690 \etex_firstmarks:D 542 \etex_fontchardp:D 567 \etex_fontcharht:D 566 \etex_fontcharic:D 569 \etex_fontcharwd:D 568 \etex_glueexpr:D 575, 4141, 4152, 4157, 4176, 4183, 4188, 4191, 4197, 14698 \etex_glueshrink:D 578, 15552 \etex_glueshrinkorder:D 580 \etex_gluestretch:D 577, 15551 \etex_gluestretchorder:D 579 \etex_gluetomu:D 581 \etex_ifcsname:D 536, 703 \etex_ifdefined:D 535, 640, 660, 672, 680, 702, 746 \etex_iffontchar:D 565 \etex_interactionmode:D 563, 6673, 6676, 6677 \etex_interlinepenalties:D 584 \etex_lastlinefit:D 583 \etex_lastnodetype:D 564 \etex_marks:D 540 \etex_middle:D 588 \etex_muexpr:D 576, 4232, 4243, 4248, 4253, 4259 \etex_mutogluue:D 582 \etex_numexpr:D 573, 3032 \etex_pagediscards:D 591 \etex_parshapedimen:D 572
\emergencystretch 432	
\end 93, 306	
\endcsname 14, 20, 24, 74, 75, 98, 128, 149, 308	
\endgroup 13, 19, 59, 61, 92, 103, 112, 131, 241	
\endinput 113, 280	
\endL 595	
\endlinechar 148, 160, 322	
\endR 597	
\eqno 342	
\errhelp 83, 104, 288	
\errmessage 91, 105, 282	

<code>\etex_parshapeindent:D</code>	570	1541, 1546, 1547, 1548, 1552, 1553,
<code>\etex_parshapelength:D</code>	571	1554, 1558, 1559, 1560, 1564, 1565,
<code>\etex_predisplaydirection:D</code>	598	1566, 1570, 1571, 1572, 1576, 1577,
<code>\etex_protected:D</code>		1578, 1579, 1583, 1584, 1585, 1586,
..... 600, 753, 755, 757, 758,		1590, 1591, 1592, 1597, 1598, 1599,
759, 760, 762, 764, 772, 774, 776, 778		1600, 1604, 1605, 1606, 1607, 1640,
<code>\etex_readline:D</code>	550, 9447	1641, 1644, 1647, 1648, 1652, 1653,
<code>\etex_savinghyphcodes:D</code>	589	1661, 1663, 1664, 1666, 1668, 1671,
<code>\etex_savingvdiscards:D</code>	590	1672, 1677, 1678, 1682, 1685, 1686,
<code>\etex_scantokens:D</code>	548, 4487	1687, 1691, 1698, 1700, 1701, 1702,
<code>\etex_showgroups:D</code>	561	1704, 1707, 1712, 1720, 1721, 1722,
<code>\etex_showifs:D</code>	562	1723, 1736, 1739, 1760, 1767, 1787,
<code>\etex_showtokens:D</code>		1902, 2055, 2058, 2062, 2146, 2209,
549, 676, 3830, 4107, 4197, 4259, 8359		2210, 2224, 2247, 2253, 2255, 2262,
<code>\etex_splitbotmarks:D</code>	545	2269, 2271, 2279, 2286, 2575, 2594,
<code>\etex_splitdiscards:D</code>	592	2619, 2628, 2644, 2666, 2686, 2704,
<code>\etex_splitfirstmarks:D</code>	544	2722, 2735, 2746, 2756, 2776, 2799,
<code>\etex_TeXeTstate:D</code>	593	2800, 2802, 2808, 2811, 2883, 2885,
<code>\etex_topmarks:D</code>	541	2895, 2896, 2897, 2899, 2905, 2906,
<code>\etex_tracingassigns:D</code>	551	2908, 2915, 2916, 2918, 2924, 2925,
<code>\etex_tracinggroups:D</code>	558	2928, 3005, 3014, 3023, 3044, 3049,
<code>\etex_tracingifs:D</code>	554	3052, 3053, 3060, 3061, 3077, 3078,
<code>\etex_tracingnesting:D</code>	553	3099, 3100, 3210, 3215, 3220, 3243,
<code>\etex_tracingscantokens:D</code>	552	3245, 3372, 3373, 3374, 3568, 3596,
<code>\etex_unexpanded:D</code>	546, 670,	3601, 3629, 3642, 3652, 3785, 3830,
708, 1701, 1704, 1707, 1712, 3226,		3916, 3920, 3923, 3924, 3931, 3932,
4881, 4907, 4928, 15569, 15619, 15627		3956, 3961, 3974, 3977, 4081, 4107,
<code>\etex_unless:D</code>	537, 693	4175, 4197, 4259, 4322, 4329, 4410,
<code>\etex_widowpenalties:D</code>	586	4485, 4486, 4537, 4545, 4550, 4591,
<code>\eTeXrevision</code>	539	4603, 4604, 4660, 4661, 4750, 4854,
<code>\eTeXversion</code>	538	4881, 4911, 4914, 4915, 4916, 4918,
<code>\everycr</code>	250	4931, 4941, 4960, 4982, 4992, 4995,
<code>\everydisplay</code>	351	5017, 5018, 5019, 5037, 5038, 5039,
<code>\everyeof</code>	599	5088, 5096, 5119, 5213, 5223, 5232,
<code>\everyhbox</code>	490	5342, 5344, 5354, 5362, 5461, 5499,
<code>\everyjob</code>	51, 53, 519	5511, 5522, 5523, 5524, 5545, 5546,
<code>\everymath</code>	375	5593, 5623, 5624, 5625, 5695, 5696,
<code>\everypar</code>	438	5699, 5900, 5915, 5937, 5953, 6010,
<code>\everyvbox</code>	491	6018, 6023, 6195, 6196, 6199, 6200,
<code>\exhyphenpenalty</code>	414	6338, 6530, 7776, 8288, 8289, 8290,
<code>\exp_after:wN</code>	31, 706, 706,	8291, 8302, 8359, 8360, 8400, 8402,
719, 721, 726, 728, 816, 818, 862,		8546, 8672, 8678, 9153, 9165, 9693,
875, 892, 894, 941, 946, 953, 993,		9700, 9703, 9780, 9782, 9783, 9793,
997, 1006, 1007, 1036, 1038, 1041,		9794, 9795, 9804, 9806, 9814, 9816,
1064, 1066, 1069, 1223, 1225, 1233,		9824, 9826, 9833, 9834, 9835, 9836,
1253, 1255, 1289, 1338, 1404, 1452,		9837, 9838, 9843, 9844, 9845, 9846,
1462, 1469, 1471, 1474, 1475, 1482,		9847, 9848, 9849, 9899, 9901, 9928,
1486, 1487, 1492, 1493, 1498, 1503,		9932, 9941, 9945, 9960, 9967, 9996,
1505, 1508, 1516, 1518, 1520, 1522,		10064, 10072, 10089, 10098, 10144,
1524, 1527, 1528, 1529, 1533, 1536,		10188, 10224, 10225, 10226, 10285,

10312, 10313, 10316, 10327, 10331,
10338, 10339, 10350, 10351, 10360,
10367, 10369, 10370, 10378, 10397,
10407, 10426, 10427, 10429, 10430,
10434, 10435, 10437, 10438, 10447,
10448, 10455, 10460, 10464, 10465,
10467, 10468, 10470, 10483, 10484,
10485, 10500, 10501, 10506, 10507,
10509, 10518, 10519, 10520, 10521,
10524, 10525, 10526, 10527, 10538,
10555, 10556, 10566, 10567, 10577,
10589, 10593, 10606, 10608, 10616,
10626, 10628, 10634, 10639, 10641,
10643, 10649, 10650, 10654, 10656,
10668, 10669, 10691, 10693, 10699,
10702, 10704, 10708, 10713, 10718,
10719, 10729, 10730, 10732, 10733,
10736, 10740, 10745, 10759, 10762,
10774, 10783, 10790, 10791, 10792,
10793, 10795, 10797, 10808, 10809,
10810, 10811, 10813, 10815, 10817,
10818, 10819, 10825, 10826, 10836,
10840, 10841, 10843, 10844, 10845,
10850, 10856, 10867, 10869, 10876,
10877, 10879, 10880, 10887, 10893,
10903, 10955, 10956, 10957, 10958,
10972, 10973, 10975, 10983, 10984,
10998, 10999, 11012, 11043, 11044,
11045, 11051, 11052, 11053, 11054,
11060, 11061, 11062, 11063, 11126,
11139, 11140, 11142, 11143, 11149,
11150, 11167, 11168, 11170, 11171,
11179, 11180, 11182, 11185, 11197,
11198, 11199, 11201, 11202, 11203,
11216, 11217, 11218, 11220, 11221,
11222, 11231, 11232, 11233, 11236,
11239, 11243, 11268, 11269, 11270,
11273, 11275, 11276, 11277, 11294,
11295, 11298, 11300, 11301, 11302,
11321, 11322, 11324, 11325, 11326,
11336, 11337, 11338, 11340, 11341,
11342, 11348, 11349, 11350, 11352,
11353, 11354, 11363, 11364, 11367,
11369, 11370, 11371, 11379, 11380,
11383, 11385, 11386, 11387, 11408,
11409, 11410, 11411, 11412, 11413,
11414, 11424, 11425, 11427, 11428,
11429, 11449, 11450, 11451, 11452,
11453, 11454, 11455, 11456, 11461,
11462, 11463, 11464, 11465, 11466,
11467, 11482, 11483, 11485, 11486,
11493, 11494, 11495, 11500, 11501,
11503, 11512, 11513, 11528, 11530,
11531, 11532, 11592, 11606, 11616,
11624, 11625, 11726, 11728, 11767,
11769, 11791, 11792, 11795, 11802,
11803, 11806, 11807, 11813, 11818,
11825, 11826, 11833, 11834, 11881,
11882, 11883, 11885, 11896, 11912,
11913, 11915, 11916, 11926, 11928,
11934, 11935, 11939, 11942, 11964,
11966, 11979, 11981, 11987, 11989,
11992, 11998, 12000, 12002, 12003,
12004, 12006, 12011, 12013, 12015,
12019, 12022, 12028, 12029, 12033,
12035, 12036, 12037, 12045, 12047,
12048, 12055, 12061, 12068, 12069,
12074, 12075, 12076, 12077, 12095,
12096, 12097, 12103, 12104, 12105,
12110, 12112, 12120, 12122, 12124,
12125, 12127, 12138, 12140, 12142,
12143, 12148, 12199, 12200, 12207,
12208, 12210, 12212, 12214, 12216,
12218, 12220, 12222, 12231, 12233,
12239, 12241, 12243, 12244, 12245,
12251, 12253, 12255, 12256, 12257,
12278, 12279, 12282, 12290, 12292,
12296, 12297, 12298, 12299, 12304,
12306, 12312, 12315, 12318, 12321,
12329, 12332, 12335, 12338, 12345,
12347, 12353, 12361, 12363, 12365,
12382, 12384, 12391, 12393, 12396,
12402, 12404, 12406, 12407, 12408,
12410, 12424, 12425, 12428, 12446,
12448, 12450, 12462, 12465, 12468,
12471, 12474, 12477, 12480, 12483,
12487, 12499, 12503, 12507, 12510,
12525, 12531, 12533, 12535, 12545,
12569, 12572, 12584, 12586, 12590,
12591, 12592, 12594, 12595, 12597,
12601, 12602, 12603, 12604, 12617,
12622, 12624, 12631, 12634, 12637,
12640, 12643, 12646, 12654, 12655,
12667, 12675, 12677, 12687, 12689,
12696, 12705, 12707, 12710, 12713,
12716, 12719, 12732, 12734, 12742,
12744, 12752, 12754, 12763, 12766,
12769, 12776, 12791, 12792, 12809,
12811, 12812, 12869, 12882, 12884,
12890, 12903, 12905, 12907, 12931,

12945, 12947, 12954, 12956, 12996,
 12997, 12998, 13000, 13001, 13002,
 13004, 13005, 13011, 13013, 13014,
 13020, 13022, 13023, 13024, 13025,
 13037, 13043, 13045, 13076, 13083,
 13090, 13110, 13111, 13113, 13115,
 13117, 13130, 13135, 13136, 13137,
 13138, 13139, 13143, 13148, 13150,
 13156, 13162, 13164, 13165, 13171,
 13172, 13173, 13174, 13175, 13176,
 13177, 13178, 13183, 13185, 13187,
 13189, 13191, 13195, 13197, 13199,
 13201, 13203, 13205, 13223, 13227,
 13235, 13236, 13241, 13243, 13252,
 13255, 13256, 13257, 13259, 13260,
 13261, 13269, 13275, 13287, 13290,
 13291, 13293, 13294, 13312, 13313,
 13317, 13323, 13327, 13329, 13346,
 13364, 13370, 13379, 13380, 13381,
 13382, 13390, 13406, 13422, 13438,
 13454, 13470, 13496, 13500, 13501,
 13505, 13507, 13538, 13544, 13545,
 13547, 13549, 13550, 13552, 13553,
 13563, 13564, 13567, 13568, 13569,
 13571, 13572, 13573, 13590, 13591,
 13593, 13594, 13600, 13602, 13605,
 13608, 13611, 13614, 13622, 13625,
 13627, 13630, 13637, 13641, 13649,
 13650, 13653, 13655, 13657, 13661,
 13665, 13670, 13671, 13679, 13680,
 13681, 13682, 13699, 13700, 13706,
 13707, 13832, 13833, 13834, 13836,
 13854, 13855, 13856, 13857, 13858,
 13859, 13866, 13875, 13882, 13883,
 13893, 14057, 14058, 14059, 14080,
 14081, 14091, 14100, 14108, 14109,
 14111, 14112, 14117, 14118, 14127,
 14134, 14143, 14144, 14157, 14159,
 14187, 14188, 14197, 14200, 14225,
 14231, 14232, 14251, 14257, 14259,
 14284, 14285, 14287, 14301, 14302,
 14310, 14320, 14354, 14357, 14367,
 14368, 14371, 14373, 14379, 14393,
 14395, 14436, 14439, 14459, 14521,
 14525, 14535, 14538, 14544, 14545,
 14564, 14566, 14567, 14576, 14580,
 14585, 14588, 14594, 14620, 14621,
 14627, 14628, 14629, 14636, 14644,
 14648, 14653, 14656, 14664, 14667,
 14668, 14670, 14671, 14681, 14685,
 14690, 14695, 14696, 14697, 14703,
 14705, 14708, 14710, 14752, 14754,
 14755, 15496, 15504, 15507, 15569,
 15589, 15619, 15627, 15651, 15652,
 15668, 15669, 15732, 15774, 15775,
 15818, 15823, 16010, 16015, 16017
 \exp_args:cc
 . . . [718](#), [720](#), [897](#), [907](#), [912](#), [917](#), [1521](#)
 \exp_args:Nc [28](#), [718](#), [718](#), [722](#), [730](#), [1137](#),
[1159](#), [1177](#), [1203](#), [1208](#), [1215](#), [1270](#),
[1281](#), [1337](#), [1370](#), [1371](#), [1372](#), [1373](#),
[1392](#), [1521](#), [4721](#), [9001](#), [10014](#), [15466](#)
 \exp_args:Ncc [1205](#), [1209](#), [1217](#),
[1378](#), [1379](#), [1380](#), [1381](#), [1521](#), [1523](#)
 \exp_args:Nccc [1521](#), [1525](#)
 \exp_args:Ncco [1581](#), [1602](#)
 \exp_args:Nccx [1625](#), [1634](#)
 \exp_args:Ncf [1544](#), [1568](#)
 \exp_args:NcNc [1581](#), [1588](#)
 \exp_args:NcNo [1581](#), [1595](#)
 \exp_args:Ncnx [1625](#), [1635](#)
 \exp_args:Nco [1544](#), [1562](#)
 \exp_args:Ncx [1610](#), [1620](#)
 \exp_args:Nf
 . [29](#), [1532](#), [1532](#), [3272](#), [3277](#), [3282](#),
[3287](#), [3442](#), [3511](#), [3513](#), [3531](#), [3540](#),
[3551](#), [3560](#), [3688](#), [3701](#), [3715](#), [3725](#),
[3736](#), [3747](#), [3993](#), [3998](#), [4003](#), [4008](#),
[5043](#), [5061](#), [5596](#), [5614](#), [6184](#), [6232](#),
[6245](#), [6250](#), [6266](#), [8300](#), [15588](#), [15643](#)
 \exp_args:Nff [1610](#), [1612](#)
 \exp_args:Nfo [1610](#), [1611](#), [6220](#)
 \exp_args:NNc [1204](#), [1207](#),
[1216](#), [1283](#), [1374](#), [1375](#), [1376](#), [1377](#),
[1411](#), [1521](#), [1521](#), [3403](#), [3410](#), [8302](#)
 \exp_args:Nnc [1610](#), [1610](#)
 \exp_args:NNf [1544](#), [1544](#), [3396](#), [13826](#), [13827](#)
 \exp_args:Nnf [1610](#), [1613](#)
 \exp_args:Nnnc [1625](#), [1627](#)
 \exp_args:NNNo [30](#), [1516](#), [1519](#), [15671](#), [15694](#)
 \exp_args:NNno [1625](#), [1625](#)
 \exp_args:Nnno [1625](#), [1628](#)
 \exp_args:NNNV [1581](#), [1581](#)
 \exp_args:NNnx [30](#), [1625](#), [1630](#)
 \exp_args:Nnnx [1625](#), [1632](#)
 \exp_args:NNo [29](#),
[1516](#), [1517](#), [3430](#), [6332](#), [9604](#), [15645](#)
 \exp_args:Nno [29](#), [1610](#), [1614](#),
[2997](#), [3964](#), [6139](#), [10049](#), [10057](#),
[10066](#), [10083](#), [10091](#), [10119](#), [15675](#)

`\exp_args:NNoo` 30, 1625, 1626
`\exp_args:NNox` 1625, 1631
`\exp_args:Nnox` 1625, 1633
`\exp_args:NNV` ... 1544, 1556, 15821, 15898
`\exp_args:NNv` ... 1544, 1550, 5228, 15752
`\exp_args:NnV` 1610, 1615
`\exp_args:NNx` 30, 1610, 1619
`\exp_args:Nnx` 1610, 1621
`\exp_args:No`
 . 28, 1516, 1516, 3430, 3517, 3523,
 4181, 4476, 4639, 4640, 4641, 4654,
 4655, 4656, 4657, 4710, 4727, 4736,
 4790, 4792, 4900, 4902, 4925, 4934,
 5063, 5334, 5839, 6017, 6028, 6030,
 6063, 6068, 6260, 6264, 8721, 8739,
 8765, 8787, 9001, 9689, 15481, 15665
`\exp_args:Noc` 1610, 1618
`\exp_args:Nof` 1610, 1617
`\exp_args:Noo` 1610, 1616
`\exp_args:Nooo` 1625, 1629
`\exp_args:Noox` 1625, 1636
`\exp_args:Nox` 1610, 1622
`\exp_args:NV`
 29, 1532, 1539, 8719, 8737, 8763, 8785
`\exp_args:Nv` 29, 1532, 1534
`\exp_args:NVV` 1544, 1574
`\exp_args:Nx` 29, 1231, 1609,
 1609, 7742, 8723, 8741, 8767, 8789
`\exp_args:Nxo` 1610, 1623
`\exp_args:Nxx` 1610, 1624
`\exp_last_two_unbraced:Noo`
 ... 31, 1697, 1697, 7082, 7305, 7309
`\exp_last_unbraced:Nco` . 1660, 1667, 6122
`\exp_last_unbraced:NcV` 1660, 1669
`\exp_last_unbraced:Nf`
 ... 31, 1660, 1665, 3529, 3549, 5802
`\exp_last_unbraced:Nfo` 1660, 1694
`\exp_last_unbraced:NNNo` ... 1660, 1690
`\exp_last_unbraced:NnNo` ... 1660, 1695
`\exp_last_unbraced:NNNV` ... 1660, 1683
`\exp_last_unbraced:NNo` .. 1660, 1681,
 4868, 6090, 6523, 7279, 15745, 15854
`\exp_last_unbraced:Nno` 1660, 1692, 15489
`\exp_last_unbraced:NNV` 1660, 1675
`\exp_last_unbraced:No` 1660,
 1664, 6275, 7434, 7439, 7517, 7523
`\exp_last_unbraced:Noo`
 ... 1660, 1693, 6390, 6478
`\exp_last_unbraced:NV` 1660, 1660
`\exp_last_unbraced:Nv` 1660, 1662
`\exp_last_unbraced:Nx` ... 31, 1660, 1696
`\exp_not:c` 32,
 1701, 1702, 1812, 1862, 2944, 7822,
 7824, 7826, 7828, 7834, 7839, 7841,
 7843, 8066, 8068, 8070, 8072, 8078,
 8083, 8085, 8087, 8293, 8422, 8449,
 8564, 8566, 8580, 8582, 8695, 11865
`\exp_not:f` 32, 1701,
 1703, 5354, 5362, 14768, 14770, 14772
`\exp_not:N` 32, 706, 707,
 866, 928, 931, 1288, 1289, 1337,
 1338, 1462, 1498, 1702, 1736, 1771,
 1780, 1809, 1810, 1811, 1862, 1902,
 2261, 2268, 2285, 2312, 2321, 2469,
 2493, 2498, 2503, 2508, 2515, 2521,
 2526, 2531, 2536, 2541, 2546, 2556,
 2561, 2589, 2594, 2847, 2852, 2870,
 2894, 2897, 2904, 2905, 2914, 2915,
 3415, 4086, 4476, 4483, 4493, 4495,
 4529, 4530, 4938, 4941, 4957, 4960,
 4991, 4998, 5333, 5335, 5670, 6174,
 6177, 6185, 6186, 6433, 6456, 6674,
 7833, 8077, 8293, 8564, 8566, 8580,
 8582, 8644, 8645, 8695, 9132, 9133,
 9602, 9673, 10424, 10485, 10547,
 10588, 10592, 10614, 10707, 10739,
 10824, 10838, 10855, 10865, 10875,
 10912, 10915, 11163, 11165, 11320,
 11335, 11347, 11435, 14677, 14680,
 14751, 14752, 14754, 14755, 14756,
 14757, 14759, 15542, 15666, 15690,
 15724, 15754, 15809, 16006, 16007
`\exp_not:n` 32, 706, 708,
 867, 869, 933, 1233, 1462, 1657,
 1816, 1831, 2261, 2268, 2285, 2312,
 2321, 2848, 2853, 2867, 2871, 2943,
 2945, 3416, 4281, 4340, 4346, 4358,
 4366, 4382, 4390, 4414, 4416, 4531,
 5012, 5060, 5133, 5353, 5361, 5378,
 5430, 5433, 5436, 5504, 5557, 5613,
 5671, 5708, 5718, 5811, 5812, 5858,
 6024, 6178, 6184, 6209, 6214, 6244,
 6277, 6397, 6434, 6437, 6457, 7816,
 7833, 7931, 8060, 8077, 8285, 8647,
 8695, 9213, 9447, 9513, 9516, 9597,
 11866, 13893, 15565, 15742, 15851
`\exp_not:o` 32,
 1701, 1701, 4311, 4313, 4342, 4348,
 4358, 4360, 4362, 4364, 4366, 4368,
 4370, 4372, 4382, 4384, 4386, 4388,

- 4390, 4392, 4394, 4396, 4417, 4455,
 4462, 4495, 4542, 4554, 4786, 4788,
 5011, 5410, 5827, 5829, 5924, 6017,
 6044, 6438, 6461, 8424, 8439, 8451,
 8452, 8835, 8853, 9010, 9012, 15564
 \exp_not:V 32, 1701,
 1705, 4360, 4368, 4384, 4392, 9203
 \exp_not:v 32, 1701, 1710, 9135
 \exp_stop_f: 32, 1472,
 1478, 2211, 3046, 3056, 3064, 3237,
 3242, 3972, 5365, 8302, 9734, 9939,
 10214, 10230, 10406, 10433, 10605,
 10625, 10652, 10666, 10701, 10728,
 10737, 10756, 10772, 10788, 10806,
 10866, 10885, 10901, 11478, 11644,
 11646, 11649, 11657, 11658, 11873,
 11875, 11895, 11937, 12010, 12032,
 12086, 12087, 12427, 12445, 12589,
 12606, 12846, 12847, 12944, 13065,
 13083, 13092, 13249, 13284, 13383,
 13495, 13537, 13542, 13624, 13687,
 13720, 13734, 13749, 13764, 13779,
 13794, 13809, 13881, 14107, 14319,
 14392, 14401, 14416, 14468, 14481,
 14536, 14586, 14640, 14654, 15646
 \expandafter 13, 14, 19,
 20, 24, 51, 53, 54, 74, 75, 98, 131, 238
 \expanded 624
 \ExplFileDate . 16049, 16053, 16057, 16061
 \ExplFileVersion 16049, 16053, 16057, 16061
 \ExplSyntaxOff
 ... 4, 7, 134, 134, 136, 138, 166, 180
 \ExplSyntaxOn 4, 7, 162, 162
- F**
- \F 2569, 2768, 10389
 \fam 230
 \fi 17, 22,
 56, 57, 58, 107, 116, 117, 118, 130, 269
 \fi: 23, 688, 692, 729, 863, 876,
 895, 942, 947, 954, 992, 997, 1030,
 1031, 1039, 1045, 1058, 1059, 1067,
 1073, 1134, 1148, 1226, 1246, 1256,
 1267, 1368, 1453, 1501, 1504, 1511,
 1512, 1745, 1761, 1768, 1777, 1792,
 1793, 1794, 1808, 1828, 1830, 1851,
 1852, 1988, 2022, 2049, 2068, 2188,
 2190, 2192, 2194, 2196, 2198, 2248,
 2256, 2263, 2272, 2280, 2287, 2295,
 2303, 2316, 2325, 2494, 2499, 2504,
 2509, 2516, 2522, 2527, 2532, 2537,
 2542, 2547, 2552, 2557, 2562, 2584,
 2590, 2601, 2602, 2648, 2649, 2670,
 2671, 2690, 2691, 2708, 2709, 2726,
 2727, 2794, 2803, 2812, 2820, 2886,
 2900, 2909, 2919, 2929, 3049, 3072,
 3089, 3090, 3092, 3195, 3203, 3230,
 3238, 3244, 3267, 3304, 3312, 3597,
 3630, 3920, 3943, 3952, 3971, 3975,
 3989, 4169, 4411, 4583, 4595, 4608,
 4618, 4635, 4665, 4855, 4908, 4911,
 4919, 4921, 4948, 4971, 4987, 4996,
 5005, 5027, 5031, 5039, 5100, 5129,
 5135, 5148, 5406, 5409, 5446, 5462,
 5483, 5493, 5544, 5549, 5901, 5916,
 5939, 5955, 6495, 6531, 6632, 6634,
 6644, 7657, 9435, 9436, 9704, 9784,
 9792, 9796, 9807, 9817, 9827, 9887,
 9925, 9926, 9927, 9928, 9929, 9930,
 9931, 9932, 9933, 9934, 9935, 9936,
 9946, 9949, 9961, 9968, 9973, 9976,
 10021, 10124, 10194, 10195, 10204,
 10205, 10216, 10217, 10218, 10228,
 10229, 10233, 10234, 10242, 10245,
 10246, 10254, 10255, 10262, 10270,
 10271, 10286, 10317, 10336, 10337,
 10346, 10352, 10372, 10373, 10386,
 10410, 10431, 10439, 10440, 10516,
 10529, 10551, 10553, 10560, 10568,
 10597, 10598, 10601, 10603, 10604,
 10609, 10619, 10622, 10624, 10629,
 10660, 10673, 10678, 10684, 10687,
 10688, 10722, 10723, 10750, 10751,
 10764, 10767, 10778, 10801, 10820,
 10830, 10846, 10860, 10866, 10870,
 10875, 10881, 10896, 10907, 10925,
 10935, 10937, 10943, 10964, 10990,
 11114, 11115, 11116, 11172, 11176,
 11187, 11188, 11204, 11223, 11241,
 11244, 11278, 11303, 11313, 11327,
 11343, 11355, 11372, 11388, 11430,
 11439, 11445, 11458, 11460, 11480,
 11481, 11487, 11505, 11601, 11612,
 11624, 11625, 11626, 11633, 11635,
 11636, 11642, 11643, 11652, 11653,
 11661, 11662, 11664, 11665, 11729,
 11742, 11752, 11753, 11758, 11759,
 11760, 11761, 11762, 11763, 11770,
 11779, 11784, 11808, 11810, 11812,
 11819, 11848, 11851, 11852, 11857,

- 11879, 11880, 11886, 11899, 11917,
 11919, 11929, 11943, 11973, 11982,
 12016, 12038, 12056, 12073, 12090,
 12091, 12093, 12094, 12098, 12113,
 12146, 12175, 12176, 12177, 12178,
 12179, 12192, 12234, 12307, 12372,
 12374, 12375, 12385, 12414, 12417,
 12418, 12429, 12449, 12512, 12513,
 12514, 12526, 12565, 12566, 12567,
 12568, 12574, 12577, 12579, 12589,
 12606, 12822, 12826, 12828, 12832,
 12839, 12840, 12850, 12851, 12854,
 12946, 13015, 13026, 13038, 13064,
 13071, 13082, 13098, 13105, 13166,
 13222, 13232, 13234, 13244, 13262,
 13263, 13295, 13298, 13307, 13309,
 13311, 13330, 13344, 13345, 13365,
 13386, 13387, 13400, 13416, 13432,
 13448, 13464, 13480, 13494, 13502,
 13508, 13518, 13521, 13530, 13539,
 13541, 13547, 13554, 13557, 13566,
 13574, 13595, 13628, 13629, 13656,
 13658, 13672, 13673, 13691, 13702,
 13711, 13714, 13715, 13724, 13727,
 13744, 13759, 13774, 13789, 13804,
 13819, 13822, 13824, 13841, 13886,
 14113, 14149, 14150, 14160, 14201,
 14202, 14226, 14258, 14264, 14265,
 14268, 14270, 14271, 14275, 14287,
 14306, 14311, 14318, 14321, 14353,
 14363, 14364, 14374, 14396, 14411,
 14429, 14437, 14440, 14468, 14476,
 14492, 14535, 14557, 14569, 14585,
 14606, 14639, 14653, 14659, 14672,
 14673, 14711, 14712, 15497, 16006,
 16007, 16008, 16018, 16024, 16026
 \file_add_path:nN
 ... 168, 9221, 9221, 9259, 9358, 9370
 \file_if_exist:n 9257
 \file_if_exist:nT 15430, 15435
 \file_if_exist:nTF
 168, 9257, 9271, 15443, 15448
 \file_if_exist_input:n 15428
 \file_if_exist_input:nF 15441
 \file_if_exist_input:nT 15433
 \file_if_exist_input:nTF
 198, 15428, 15446
 \file_input:n 168, 9264, 9264
 \file_list: 169, 9314, 9314
 \file_path_include:n ... 169, 9302, 9302
 \file_path_remove:n 169, 9302, 9309
 \finalhyphendemerits 418
 \firstmark 316
 \firstmarks 542
 \floatingpenalty 463
 \fmtname 124
 \font 229
 \fontchardp 567
 \fontcharht 566
 \fontcharic 569
 \fontcharwd 568
 \fontdimen 496
 \fontname 320
 \fp_abs:n
 . 191, 14731, 14731, 14978, 14980,
 15061, 15063, 15065, 15383, 15385
 \fp_add:cn 14790
 \fp_add:Nn 178, 14790, 14790, 14796
 \fp_compare:n 11590
 \fp_compare:nF 11670, 11681
 \fp_compare:nNn 11603
 \fp_compare:nNnF 11698, 11709
 \fp_compare:nNnT 11704, 11717, 15367, 16200
 \fp_compare:nNnTF
 . . 180, 7143, 11603, 14853, 14855,
 14860, 15079, 15088, 16206, 16221
 \fp_compare:nT 11676, 11689
 \fp_compare:nTF 180, 11590
 \fp_compare_p:n 180, 11590
 \fp_compare_p:nNn 180, 11603
 \fp_const:cn 14767
 \fp_const:Nn 177, 14767, 14771,
 14775, 14812, 14813, 14814, 14815
 \fp_do_until:nn . 181, 11667, 11667, 11671
 \fp_do_until:nNnn 181, 11695, 11695, 11699
 \fp_do_while:nn . 181, 11667, 11673, 11677
 \fp_do_while:nNnn 181, 11695, 11701, 11705
 \fp_eval:n 178, 14728, 14730, 16211, 16223
 \fp_flag_off:n 183, 10011, 10011
 \fp_flag_on:n 183, 10013, 10013, 10054,
 10063, 10071, 10088, 10097, 10128
 \fp_function:Nw 11510, 11510
 \fp_gadd:cn 14790
 \fp_gadd:Nn 178, 14790, 14791, 14797
 .fp_gset:c 159
 \fp_gset:cn 14767
 .fp_gset:N 159
 \fp_gset:Nn 177, 14767,
 14769, 14774, 14791, 14793, 16040
 \fp_gset_eq:cc 14776

<code>\fp_gset_eq:cN</code>	14776	<code>\fp_sub:cn</code>	14790
<code>\fp_gset_eq:Nc</code>	14776	<code>\fp_sub:Nn</code>	178 , 14790 , 14792 , 14798
<code>\fp_gset_eq:NN</code>	178 , 14776 , 14777 , 14779 , 14781	<code>\fp_to_decimal:c</code>	14575
<code>\fp_gset_from_dim:cn</code>	16037	<code>\fp_to_decimal:N</code>	178 , 10004 , 14575 , 14575 , 14577 , 14677 , 14728
<code>\fp_gset_from_dim:Nn</code>	16037 , 16039 , 16042	<code>\fp_to_decimal:n</code>	14575 , 14578 , 14680 , 14730 , 14732 , 14734 , 14736
<code>\fp_gsub:cn</code>	14790	<code>\fp_to_dim:c</code>	14676
<code>\fp_gsub:Nn</code>	178 , 14790 , 14793 , 14799	<code>\fp_to_dim:N</code>	178 , 14676 , 14676 , 14678
<code>\fp_gzero:c</code>	14780	<code>\fp_to_dim:n</code>	7149 , 7174 , 14676 , 14679 , 14885 , 14896 , 15281 , 15289 , 15393 , 15395
<code>\fp_gzero:N</code>	177 , 14780 , 14781 , 14783 , 14787	<code>\fp_to_int:c</code>	14681
<code>\fp_gzero_new:c</code>	14784	<code>\fp_to_int:N</code>	179 , 14681 , 14681 , 14682
<code>\fp_gzero_new:N</code>	177 , 14784 , 14786 , 14789	<code>\fp_to_int:n</code>	14681 , 14683
<code>\fp_if_exist:c</code>	11589	<code>\fp_to_scientific:c</code>	14520
<code>\fp_if_exist:cTF</code>	11588	<code>\fp_to_scientific:N</code>	179 , 10005 , 14520 , 14520 , 14522
<code>\fp_if_exist:N</code>	11588	<code>\fp_to_scientific:n</code>	14520 , 14523
<code>\fp_if_exist:NTF</code>	179 , 11588 , 14785 , 14787 , 14802	<code>\fp_to_tl:c</code>	14644
<code>\fp_if_exist_p:c</code>	11588	<code>\fp_to_tl:N</code>	179 , 14644 , 14644 , 14645 , 14803
<code>\fp_if_exist_p:N</code>	179 , 11588	<code>\fp_to_tl:n</code>	9748 , 10053 , 10062 , 10087 , 10096 , 10125 , 14644 , 14646 , 14810
<code>\fp_if_flag_on:n</code>	10015	<code>\fp_trap:nn</code>	184 , 10027 , 10027 , 10139 , 10140 , 10141 , 10142
<code>\fp_if_flag_on:nTF</code>	183 , 10015	<code>\fp_until_do:nn</code>	181 , 11667 , 11679 , 11684
<code>\fp_if_flag_on_p:n</code>	183 , 10015	<code>\fp_until_do:nNnn</code>	181 , 11695 , 11707 , 11712
<code>\fp_max:nn</code>	191 , 14733 , 14733	<code>\fp_use:c</code>	14728
<code>\fp_min:nn</code>	191 , 14733 , 14735	<code>\fp_use:N</code>	179 , 14728 , 14728 , 14729 , 16205 , 16209 , 16214 , 16238 , 16239 , 16246 , 16248
<code>\fp_new:N</code>	177 , 6801 , 6802 , 14764 , 14764 , 14766 , 14785 , 14787 , 14816 , 14817 , 14818 , 14819 , 14823 , 14824 , 14825 , 14947 , 14948 , 15198 , 15199 , 15344 , 15345	<code>\fp_while_do:nn</code>	182 , 11667 , 11687 , 11692
<code>\fp_new_function:Npn</code>	11517 , 11517	<code>\fp_while_do:nNnn</code>	181 , 11695 , 11715 , 11720
<code>.fp_set:c</code>	159	<code>\fp_zero:c</code>	14780
<code>\fp_set:cn</code>	14767	<code>\fp_zero:N</code>	177 , 14780 , 14780 , 14782 , 14785 , 16201
<code>.fp_set:N</code>	159	<code>\fp_zero_new:c</code>	14784
<code>\fp_set:Nn</code>	177 , 7139 , 7141 , 14767 , 14767 , 14773 , 14790 , 14792 , 14840 , 14841 , 14842 , 14955 , 14957 , 14989 , 15006 , 15023 , 15037 , 15039 , 15054 , 15055 , 15208 , 15209 , 15350 , 15352 , 15377 , 15378 , 16038 , 16199 , 16202 , 16218 , 16233 , 16234	<code>\fp_zero_new:N</code>	177 , 14784 , 14784 , 14788
<code>\fp_set_eq:cc</code>	14776	<code>\frozen@everydisplay</code>	642
<code>\fp_set_eq:cN</code>	14776	<code>\frozen@everymath</code>	643
<code>\fp_set_eq:Nc</code>	14776	<code>\futurelet</code>	225
<code>\fp_set_eq:NN</code>	178 , 14776 , 14776 , 14778 , 14780 , 14994 , 15011 , 15025		
<code>\fp_set_from_dim:cn</code>	16037		
<code>\fp_set_from_dim:Nn</code>	16037 , 16037 , 16041		
<code>\fp_show:c</code>	14800		
<code>\fp_show:N</code>	184 , 14800 , 14800 , 14811		
<code>\fp_show:n</code>	14800 , 14809		

G

<code>\g__file_internal_ior</code>	174 , 9225 , 9226 , 9229 , 9245 , 9246 , 9451 , 9451
<code>\g__file_record_seq</code>	9163 , 9163 , 9168 , 9288 , 9293 , 9316 , 9336
<code>\g__file_stack_seq</code>	9162 , 9162 , 9295 , 9298
<code>\g__ior_streams_prop</code>	9347 , 9347 , 9349 , 9398 , 9406 , 9414

- \g_ior_streams_seq [9341](#),
[9341](#), [9343](#), [9381](#), [9407](#), [9408](#), [9457](#)
 - \g_iow_streams_prop [9460](#), [9460](#),
[9462](#), [9463](#), [9464](#), [9491](#), [9499](#), [9507](#)
 - \g_iow_streams_seq
... [9455](#), [9455](#), [9457](#), [9474](#), [9500](#), [9501](#)
 - \g_keyval_level_int [8380](#), [8380](#),
[8423](#), [8450](#), [8457](#), [8459](#), [8461](#), [8463](#)
 - \g_prg_map_int [42](#), [2234](#), [2234](#),
[3402](#), [3405](#), [3409](#), [3412](#), [3423](#), [4719](#),
[4720](#), [4722](#), [4724](#), [5649](#), [5650](#), [5656](#),
[5657](#), [6120](#), [6121](#), [6123](#), [6126](#), [6540](#),
[6541](#), [6546](#), [6548](#), [15465](#), [15467](#), [15474](#)
 - \g_scan_marks_tl [2334](#), [2334](#), [2337](#), [2343](#)
 - \g_file_current_name_tl
..... [168](#), [7692](#), [9151](#), [9151](#),
[9156](#), [9160](#), [9168](#), [9295](#), [9296](#), [9299](#)
 - \g_peek_token [57](#), [2823](#), [2824](#), [2834](#)
 - \g_tmpa_bool [37](#), [2010](#), [2012](#)
 - \g_tmpa_box [136](#), [6656](#), [6658](#)
 - \g_tmpa_clist [126](#), [6289](#), [6291](#)
 - \g_tmpa_dim [82](#), [4110](#), [4112](#)
 - \g_tmpa_fp [182](#), [14816](#), [14818](#)
 - \g_tmpa_int [72](#), [3850](#), [3852](#)
 - \g_tmpa_muskip [88](#), [4262](#), [4264](#)
 - \g_tmpa_prop [132](#), [6327](#), [6329](#)
 - \g_tmpa_seq [116](#), [5760](#), [5762](#)
 - \g_tmpa_skip [85](#), [4200](#), [4202](#)
 - \g_tmpa_tl [103](#), [5077](#), [5077](#)
 - \g_tmpb_bool [37](#), [2010](#), [2013](#)
 - \g_tmpb_box [136](#), [6656](#), [6659](#)
 - \g_tmpb_clist [126](#), [6289](#), [6292](#)
 - \g_tmpb_dim [82](#), [4110](#), [4113](#)
 - \g_tmpb_fp [182](#), [14816](#), [14819](#)
 - \g_tmpb_int [72](#), [3850](#), [3853](#)
 - \g_tmpb_muskip [88](#), [4262](#), [4265](#)
 - \g_tmpb_prop [132](#), [6327](#), [6330](#)
 - \g_tmpb_seq [116](#), [5760](#), [5763](#)
 - \g_tmpb_skip [85](#), [4200](#), [4203](#)
 - \g_tmpb_tl [103](#), [5077](#), [5078](#)
 - \gdef [216](#)
 - \GetIdInfo [7](#)
 - \global [51](#), [198](#), [231](#)
 - \globaldefs [235](#)
 - \glueexpr [575](#)
 - \glueshrink [578](#)
 - \glueshrinkorder [580](#)
 - \gluestretch [577](#)
 - \gluestretchorder [579](#)
 - \gluetomu [581](#)
 - \group_align_safe_begin: ... [41](#), [2026](#),
[2195](#), [2195](#), [2855](#), [2873](#), [4525](#), [4835](#)
 - \group_align_safe_end:
..... [41](#), [2074](#), [2075](#), [2195](#), [2197](#),
[2837](#), [2847](#), [2852](#), [2870](#), [4534](#), [4861](#)
 - \group_begin: [10](#), [712](#), [713](#), [998](#),
[1393](#), [1411](#), [1726](#), [1860](#), [1874](#), [2201](#),
[2216](#), [2454](#), [2467](#), [2474](#), [2511](#), [2564](#),
[2604](#), [2764](#), [2931](#), [4475](#), [4626](#), [5455](#),
[6683](#), [7602](#), [7749](#), [7766](#), [7807](#), [8051](#),
[8275](#), [8385](#), [8395](#), [9182](#), [9536](#), [9541](#),
[9567](#), [10387](#), [10473](#), [10978](#), [11159](#),
[11210](#), [11226](#), [11254](#), [11316](#), [11330](#),
[11358](#), [11772](#), [14528](#), [14839](#), [14953](#),
[14987](#), [15004](#), [15021](#), [15035](#), [15053](#),
[15664](#), [15689](#), [15975](#), [15982](#), [15995](#)
 - \group_end:
... [10](#), [712](#), [714](#), [1003](#), [1398](#), [1411](#),
[1733](#), [1863](#), [1879](#), [2206](#), [2221](#), [2466](#),
[2470](#), [2483](#), [2518](#), [2572](#), [2614](#), [2770](#),
[2996](#), [4482](#), [4630](#), [4633](#), [5465](#), [5470](#),
[6693](#), [7606](#), [7756](#), [7777](#), [7911](#), [8096](#),
[8297](#), [8392](#), [8402](#), [9212](#), [9540](#), [9545](#),
[9604](#), [10394](#), [10480](#), [11008](#), [11206](#),
[11225](#), [11253](#), [11288](#), [11329](#), [11357](#),
[11390](#), [11783](#), [14532](#), [14844](#), [14963](#),
[14996](#), [15013](#), [15027](#), [15045](#), [15067](#),
[15671](#), [15694](#), [15984](#), [15993](#), [16023](#)
 - \group_insert_after:N
... [10](#), [717](#), [717](#), [16270](#), [16282](#), [16291](#)
 - .groups:n [159](#)
- ## H
- \halign [242](#)
 - \hangafter [420](#)
 - \hangindent [421](#)
 - \hbadness [482](#)
 - \hbox [477](#)
 - \hbox:n [137](#),
[6695](#), [6695](#), [7401](#), [7456](#), [14868](#), [15099](#)
 - \hbox_gset:cn [6696](#)
 - \hbox_gset:cw [6706](#), [6718](#)
 - \hbox_gset:Nn [137](#), [6696](#), [6697](#), [6699](#)
 - \hbox_gset:Nw . [138](#), [6706](#), [6708](#), [6711](#), [6717](#)
 - \hbox_gset_end: ... [138](#), [6706](#), [6713](#), [6719](#)
 - \hbox_gset_inline_begin:c ... [6714](#), [6718](#)
 - \hbox_gset_inline_begin:N ... [6714](#), [6717](#)
 - \hbox_gset_inline_end: [6714](#), [6719](#)
 - \hbox_gset_to_wd:cn [6700](#)
 - \hbox_gset_to_wd:Nnn [137](#), [6700](#), [6702](#), [6705](#)

\hbox_overlap_left:n ...	137, 6723, 6723
\hbox_overlap_right:n	137, 6723, 6725, 15076, 16190
\hbox_set:cn	6696
\hbox_set:cw	6706, 6715
\hbox_set:Nn	137, 6696, 6696, 6697, 6698, 6859, 6963, 7187, 7258, 7546, 14837, 14864, 14865, 14951, 14985, 15002, 15019, 15033, 15051, 15073, 15108, 15112, 15120, 15128, 15137, 15143, 15155, 15163, 15171, 15177, 15187, 15220, 15234
\hbox_set:Nw	138, 6706, 6706, 6709, 6710, 6714, 6906
\hbox_set_end: ...	138, 6706, 6712, 6716, 6910
\hbox_set_inline_begin:c	6714, 6715
\hbox_set_inline_begin:N	6714, 6714
\hbox_set_inline_end:	6714, 6716
\hbox_set_to_wd:cn	6700
\hbox_set_to_wd:Nnn	137, 6700, 6700, 6703, 6704
\hbox_to_wd:nn ...	137, 6720, 6720, 15090
\hbox_to_zero:n	137, 6720, 6722, 6724, 6726
\hbox_unpack:c	6727
\hbox_unpack:N	138, 6727, 6727, 6729, 7191, 7338
\hbox_unpack_clear:c	6727
\hbox_unpack_clear:N	138, 6727, 6728, 6730
\hcoffin_set:cn	6855
\hcoffin_set:cw	6902
\hcoffin_set:Nn	141, 6855, 6855, 6871, 7398, 7410, 7453, 7493
\hcoffin_set:Nw ...	142, 6902, 6902, 6918
\hcoffin_set_end: .	142, 6902, 6907, 6917
\hfil	385
\hfill	387
\hfilneg	386
\hfuzz	484
\hoffset	459
\holdinginserts	462
\hrule	398
\hsize	423
\hskip	388
\hss	389
\ht	527
\hyphenation	513
\hyphenchar	497
\hyphenpenalty	415
I	
pi	190
\if	251
\if:w	24, 688, 694, 992, 1786, 1789, 1790, 1844, 1847, 2807, 10588, 10592, 10614, 10707, 10739, 10758, 10824, 10838, 10855, 10875, 11320, 11335, 11347, 13566
\if_bool:N	41, 1907, 1907
\if_box_empty:N ...	140, 6628, 6630, 6644
\if_case:w	73, 1234, 3031, 3035, 3569, 3602, 9790, 9939, 10278, 11434, 11478, 11871, 12010, 12085, 12109, 12161, 12606, 12845, 13065, 13092, 13249, 13284, 13390, 13406, 13422, 13438, 13454, 13470, 13495, 13542, 13664, 13687, 13734, 13749, 13764, 13779, 13794, 13809, 14249, 14266, 14318, 14401, 14416, 14468, 14481, 14536, 14586, 14654
\if_catcode:w	24, 688, 696, 2493, 2498, 2503, 2508, 2515, 2521, 2526, 2531, 2536, 2541, 2546, 2556, 2589, 2889, 2894, 4660, 4956, 4998, 5016, 9699, 10424, 10547, 10865, 10912, 11163, 11307, 11435, 16006, 16007
\if_charcode:w	24, 688, 695, 2561, 2891, 4937, 4991, 5097
\if_cs_exist:N	24, 702, 702, 1026, 1054, 2597, 2816
\if_cs_exist:w	702, 703, 725, 1035, 1063, 1222, 10017
\if_dim:w	88, 3864, 3864, 3939, 3951, 3976
\if_eof:w	174, 9423, 9423, 9431
\if_false:	23, 688, 689, 2196, 3215, 3961, 4908, 4911, 4921, 5031, 5039, 5406, 5409, 5544, 5549
\if_hbox:N	140, 6628, 6628, 6632
\if_int_compare:w	73, 715, 715, 2196, 2198, 2260, 2267, 2284, 2311, 2320, 2580, 2798, 3031, 3068, 3195, 3248, 3250, 3252, 3254, 3256, 3258, 3260, 3263, 4163, 5125, 5133, 5147, 9428, 9790, 9791, 9883, 9955, 9969, 10192, 10202, 10210, 10231, 10243, 10252, 10260, 10268, 10329, 10334, 10406, 10433, 10514, 10516, 10546, 10551, 10605, 10625, 10652, 10666, 10701, 10728, 10756, 10772, 10788, 10806, 10865, 10885, 10901,

- 10914, 10927, 10960, 10986, 11164,
 11174, 11176, 11214, 11230, 11235,
 11267, 11293, 11362, 11376, 11423,
 11605, 11635, 11638, 11649, 11652,
 11657, 11658, 11661, 11664, 11744,
 11817, 11875, 11895, 11937, 12032,
 12086, 12087, 12090, 12093, 12162,
 12171, 12372, 12445, 12498, 12502,
 12506, 12524, 12559, 12560, 12561,
 12562, 12563, 12589, 12847, 12850,
 12944, 13036, 13078, 13094, 13220,
 13254, 13315, 13322, 13326, 13363,
 13535, 13537, 13548, 13566, 13589,
 13621, 13624, 13667, 13695, 13696,
 13704, 13705, 13719, 13837, 13881,
 14270, 14307, 14316, 14351, 14435,
 14438, 14563, 14637, 14663, 14666
 \if_int_odd:w ... 73, 3031, 3034, 3300,
 3308, 10214, 10228, 10240, 11474,
 12145, 12427, 13720, 14107, 14146,
 14156, 14199, 14223, 14392, 16005
 \if_meaning:w 24, 688, 697, 859, 874, 891,
 938, 943, 952, 1023, 1041, 1051,
 1069, 1252, 1263, 1367, 1451, 1498,
 1499, 1736, 1757, 1766, 1984, 2049,
 2068, 2246, 2252, 2278, 2291, 2299,
 2551, 2594, 2636, 2639, 2658, 2661,
 2678, 2681, 2696, 2699, 2714, 2717,
 2790, 2882, 2923, 3049, 3084, 3089,
 3090, 3230, 3920, 3969, 4409, 4579,
 4591, 4603, 4614, 4629, 4853, 4914,
 4982, 5442, 5460, 5479, 5487, 5897,
 5912, 5934, 5950, 6491, 6529, 9779,
 9792, 9803, 9813, 9823, 9946, 9966,
 9968, 10124, 10191, 10201, 10213,
 10228, 10229, 10240, 10241, 10251,
 10267, 10311, 10346, 10349, 10365,
 10372, 10425, 10565, 10678, 10684,
 10913, 11106, 11109, 11112, 11194,
 11475, 11492, 11597, 11624, 11625,
 11626, 11627, 11628, 11629, 11725,
 11737, 11738, 11766, 11777, 11787,
 11846, 11855, 11872, 11906, 11911,
 11925, 11971, 11978, 12054, 12066,
 12165, 12168, 12179, 12230, 12303,
 12371, 12374, 12381, 12414, 12415,
 12418, 12818, 12829, 13009, 13019,
 13062, 13161, 13240, 13289, 13303,
 13492, 13504, 13516, 13519, 13522,
 13547, 13648, 13652, 14149, 14202,
 14264, 14265, 14267, 14268, 14287,
 14304, 14370, 14468, 14535, 14585,
 14653, 14702, 14707, 15495, 16008
 \if_mode_horizontal: .. 24, 698, 699, 2190
 \if_mode_inner: 24, 698, 701, 2192
 \if_mode_math: 24, 698, 698, 2194
 \if_mode_vertical: 24, 698, 700, 2188
 \if_predicate:w 41, 1907, 1908, 2018
 \if_true: 23, 688, 688
 \if_vbox:N 140, 6628, 6629, 6634
 \ifcase 252
 \ifcat 253
 \ifcsname 536
 \ifdefined 535
 \ifdim 256
 \ifeof 257
 \iffalse 262
 \iffontchar 565
 \ifhbox 258
 \ifhmode 264
 \ifinner 267
 \ifmmode 265
 \ifnum 26, 44, 77, 254
 \ifodd 255
 \iftrue 263
 \ifvbox 259
 \ifvmode 266
 \ifvoid 260
 \ifx 14, 20, 24, 74, 75, 98, 124, 261
 \ignorespaces 309
 \immediate 271
 min 188
 sin 188
 sind 189
 \indent 405
 \initcatcodetable 625
 .initial:n 159
 .initial:o 159
 .initial:V 159
 .initial:x 159
 \input 279
 \input@path 9235, 9238, 9253
 \inputlineno 281
 \insert 461
 \insertpenalties 464
 \int_abs:n 61, 3042, 3042
 \int_add:cn 3162
 \int_add:Nn 63, 3162,
 3162, 3167, 3170, 9621, 9634, 9672
 \int_case:nn 3269, 3284, 3432, 3438

\int_case:nnF 3279, 3854, 5692, 6192
\int_case:nnn 3854, 3854
\int_case:nnT 3274
\int_case:nnTF 66, 3269, 3269
\int_compare:n 3208
\int_compare:nF 3324, 3339
\int_compare:nNn 3261
\int_compare:nNnF 3352, 3367, 3393
\int_compare:nNnT
. 3344, 3361, 5047, 5600, 15783, 15835
\int_compare:nNnTF 64, 2100,
3116, 3118, 3261, 3293, 3379, 3382,
3428, 3516, 3522, 3528, 3548, 3730,
3754, 3758, 3808, 4090, 5059, 5612,
6228, 6230, 6235, 6243, 6263, 9188,
9622, 11542, 14497, 14612, 14614,
15758, 15861, 15868, 15890, 15954
\int_compare:nT . . 3316, 3333, 9403, 9496
\int_compare:nTF 65, 3208
\int_compare_p:n 65, 3208
\int_compare_p:nNn
. 64, 3261, 15880, 15903,
15904, 15905, 15920, 15921, 15922,
15935, 15936, 15937, 15948, 15967
\int_const:cn 3114, 3767, 3768,
3769, 3770, 3771, 3772, 3773, 3774,
3775, 3776, 3777, 3778, 3779, 3780
\int_const:Nn 62, 3114, 3114,
3134, 3831, 3832, 3833, 3834, 3835,
3836, 3837, 3838, 3839, 3840, 3841,
3842, 3843, 3844, 3845, 3846, 3847,
3848, 3849, 9762, 9861, 9862, 9863,
9865, 9866, 9867, 9870, 9871, 9872
\int_decr:c 3174
\int_decr:N 63, 3174, 3176, 3181, 3183
\int_div_round:nn 61, 3074, 3095
\int_div_truncate:nn
. 62, 3074, 3074, 3443, 3541, 3561
\int_do_until:nn . . . 67, 3314, 3336, 3340
\int_do_until:nNnn . . 66, 3342, 3364, 3368
\int_do_while:nn . . . 67, 3314, 3330, 3334
\int_do_while:nNnn . . 67, 3342, 3358, 3362
\int_eval:n 61, 1260, 1276, 3036,
3037, 3040, 3272, 3277, 3282, 3287,
3397, 3425, 3511, 3513, 3643, 3653,
3712, 3726, 3730, 3733, 3748, 3757,
4760, 4765, 5045, 5061, 5598, 5614,
5680, 6165, 6174, 6232, 6245, 6267,
6675, 9389, 9482, 9768, 11521, 15593
\int_from_alph:n 70, 3710, 3710
\int_from_base:nn
. 71, 3731, 3731, 3762, 3764, 3766
\int_from_bin:n 70, 3761, 3761, 3858
\int_from_binary:n 3855, 3858
\int_from_hex:n 70, 3761, 3763, 3859
\int_from_hexadecimal:n 3855, 3859
\int_from_oct:n 71, 3761, 3765, 3860
\int_from_octal:n 3855, 3860
\int_from_roman:n 71, 3781, 3781
\int_gadd:cn 3162
\int_gadd:Nn 63, 3162, 3166, 3171
\int_gdecr:c 3174
\int_gdecr:N 63, 3174, 3180, 3185, 3423,
4724, 5657, 6126, 6546, 8463, 15474
\int_gincr:c 3174
\int_gincr:N
. 63, 3174, 3178, 3184, 3402, 3409,
4719, 5649, 6120, 6541, 8457, 15465
.int_gset:c 159
\int_gset:cn 3186
.int_gset:N 159
\int_gset:Nn 63, 3121, 3131, 3186, 3188, 3190
\int_gset_eq:cc 3154
\int_gset_eq:cN 3154
\int_gset_eq:Nc 3154
\int_gset_eq:NN
. 63, 3154, 3157, 3158, 3159, 7794
\int_gsub:cn 3162
\int_gsub:Nn 63, 3162, 3168, 3173
\int_gzero:c 3144
\int_gzero:N . . . 62, 3144, 3145, 3147, 3151
\int_gzero_new:c 3148
\int_gzero_new:N . . . 62, 3148, 3150, 3153
\int_if_even:n 3306
\int_if_even:nTF 66, 3298
\int_if_even_p:n 66, 3298
\int_if_exist:c 3161
\int_if_exist:cF 3797, 3804, 3806
\int_if_exist:cTF 3160
\int_if_exist:N 3160
\int_if_exist:NTF . . . 63, 3149, 3151, 3160
\int_if_exist_p:c 3160
\int_if_exist_p:N 63, 3160
\int_if_odd:n 3298
\int_if_odd:nTF 66, 3298, 12933
\int_if_odd_p:n 66, 3298
\int_incr:c 3174
\int_incr:N 63, 3174,
3174, 3179, 3182, 8641, 9129, 9640
\int_max:nn . . . 62, 3042, 3050, 12794, 13861

- \int_min:nn 62, 3042, 3058
- \int_mod:nn 62, 3074, 3097, 3433, 3532, 3552, 9190
- \int_new:c 3106
- \int_new:N 62, 2234, 3106, 3107, 3113, 3120, 3130, 3149, 3151, 3850, 3851, 3852, 3853, 8380, 8475, 9524, 9526, 9527, 9528, 9529, 16264
- .int_set:c 159
- \int_set:cn 3186
- .int_set:N 159
- \int_set:Nn 63, 3186, 3186, 3188, 3189, 6684, 6685, 8645, 9133, 9448, 9525, 9574, 9588, 9619, 9647
- \int_set_eq:cc 3154
- \int_set_eq:cN 3154
- \int_set_eq:Nc 3154
- \int_set_eq:NN 63, 3154, 3154, 3155, 3156, 6686, 9446, 9542, 9568
- \int_show:c 3827, 3828
- \int_show:N 71, 3827, 3827
- \int_show:n 71, 3829, 3829
- \int_step_function:nnnN 68, 3370, 3370, 3422
- \int_step_inline:nnnn ... 68, 3400, 3400
- \int_step_variable:nnnNn . 68, 3400, 3407
- \int_sub:cn 3162
- \int_sub:Nn 63, 3162, 3164, 3169, 3172, 9678
- \int_to_Alph:n 69, 3446, 3478
- \int_to_alph:n 69, 3446, 3446
- \int_to_arabic:n 68, 3425, 3425
- \int_to_Base:nn 70, 3510, 3512, 3637
- \int_to_base:nn 70, 3510, 3510, 3633, 3635, 3639
- \int_to_bin:n 69, 3632, 3632, 3855
- \int_to_binary:n 3855, 3855
- \int_to_Hex:n 70, 3632, 3636, 3856
- \int_to_hex:n 70, 3632, 3634
- \int_to_hexadecimal:n 3855, 3856
- \int_to_oct:n 70, 3632, 3638, 3857
- \int_to_octal:n 3855, 3857
- \int_to_Roman:n 70, 3640, 3650
- \int_to_roman:n 70, 3640, 3640
- \int_to_symbols:nnn 69, 3426, 3426, 3442, 3448, 3480
- \int_until_do:nn ... 67, 3314, 3322, 3327
- \int_until_do:nNnn .. 67, 3342, 3350, 3355
- \int_use:c 3191, 3192
- \int_use:N 64, 3045, 3053, 3054, 3061, 3062, 3076, 3078, 3079, 3191, 3191, 3192, 3211, 3373, 3374, 3375, 3405, 3412, 3830, 4720, 4722, 5224, 5650, 5656, 6121, 6123, 6540, 6548, 7716, 8165, 8423, 8450, 8459, 8461, 8646, 9134, 9448, 9774, 9980, 10227, 10328, 10332, 10371, 10510, 10640, 10651, 10700, 10731, 10737, 10738, 10784, 10794, 10796, 10812, 10814, 10837, 10976, 11936, 11965, 11967, 11988, 11990, 11999, 12001, 12023, 12030, 12036, 12046, 12048, 12121, 12123, 12139, 12141, 12201, 12209, 12211, 12213, 12215, 12217, 12219, 12221, 12240, 12242, 12252, 12254, 12280, 12283, 12291, 12293, 12313, 12316, 12319, 12322, 12330, 12333, 12336, 12339, 12346, 12348, 12354, 12362, 12364, 12366, 12392, 12394, 12403, 12405, 12426, 12447, 12451, 12463, 12466, 12469, 12472, 12475, 12478, 12481, 12484, 12488, 12500, 12504, 12508, 12511, 12532, 12534, 12536, 12546, 12585, 12587, 12596, 12618, 12623, 12625, 12632, 12635, 12638, 12641, 12644, 12647, 12656, 12668, 12676, 12678, 12688, 12690, 12697, 12706, 12708, 12711, 12714, 12717, 12720, 12733, 12735, 12743, 12745, 12753, 12755, 12764, 12767, 12770, 12777, 12792, 12810, 12813, 12869, 12883, 12885, 12891, 12904, 12906, 12908, 12932, 12948, 12955, 12956, 12999, 13044, 13046, 13084, 13114, 13116, 13118, 13131, 13144, 13149, 13151, 13157, 13174, 13175, 13176, 13177, 13178, 13179, 13184, 13186, 13188, 13190, 13192, 13196, 13198, 13200, 13202, 13204, 13206, 13228, 13236, 13318, 13371, 13601, 13603, 13606, 13609, 13612, 13615, 13631, 13683, 13835, 13867, 13876, 14058, 14059, 14082, 14092, 14101, 14119, 14128, 14135, 14189, 14233, 14286, 14303, 14358, 14369, 14380, 14546, 14568, 14706, 15467, 15733, 15824
- \int_while_do:nn ... 67, 3314, 3314, 3319
- \int_while_do:nNnn .. 67, 3342, 3342, 3347
- \int_zero:c 3144
- \int_zero:N 62, 3144, 3144, 3146,

- 3149, 8638, 9119, 9590, 9592, 9666
 \int_zero_new:c 3148
 \int_zero_new:N 62, 3148, 3148, 3152
 \interactionmode 563
 \interlinepenalties 584
 \interlinepenalty 443
 \ior_close:c 9401
 \ior_close:N
 ... 170, 9229, 9380, 9401, 9401, 9412
 \ior_get:NN 170, 9440, 9440, 15460
 \ior_get_str:NN .. 171, 9442, 9442, 15462
 \ior_if_eof:N 9424
 \ior_if_eof:Nf 9246, 15472, 15479
 \ior_if_eof:Ntf 171, 9226, 9424
 \ior_if_eof_p:N 171, 9424
 \ior_list_streams: 170, 9413, 9413
 \ior_map_break:
 199, 15455, 15455, 15456, 15458, 15473
 \ior_map_break:n 199, 15455, 15457
 \ior_map_inline:Nn 198, 15459, 15459
 \ior_new:c 9351
 \ior_new:N ... 169, 9351, 9351, 9352, 9451
 \ior_open:cn 9353
 \ior_open:cnTF 9363
 \ior_open:Nn .. 169, 9353, 9353, 9355, 9363
 \ior_open:Nnf 9366
 \ior_open:NnT 9365
 \ior_open:NnTF 169, 9363, 9367
 \ior_str_map_inline:Nn 198, 15459, 15461
 \iow_char:N 172, 9523, 9523, 13489
 \iow_close:c 9494
 \iow_close:N .. 170, 9473, 9494, 9494, 9505
 \iow_indent:n 173, 8129,
 9056, 9558, 9558, 9577, 10153, 10165
 \iow_list_streams: 170, 9506, 9506
 \iow_log:n 171, 7783, 7784, 7785,
 7908, 9328, 9329, 9330, 9518, 9519
 \iow_log:x 1093,
 1093, 1132, 1866, 7655, 9518, 9518
 \iow_new:c 9466
 \iow_new:N 169, 9466, 9466, 9467
 \iow_newline:
 172, 7745, 7761, 7763, 9522, 9522, 9585
 \iow_now:cn 9515
 \iow_now:cx 9515
 \iow_now:Nn 171, 9515, 9515, 9517, 9519, 9521
 \iow_now:Nx 9515, 9518, 9520
 \iow_open:cn 9468
 \iow_open:Nn 170, 9468, 9468, 9470
 \iow_shipout:cn 9512
 \iow_shipout:cx 9512
 \iow_shipout:Nn ... 171, 9512, 9512, 9514
 \iow_shipout:Nx 9512
 \iow_shipout_x:cn 9509
 \iow_shipout_x:cx 9509
 \iow_shipout_x:Nn . 172, 9509, 9509, 9511
 \iow_shipout_x:Nx 9509
 \iow_term:n
 171, 7789, 7790, 7791, 8331, 9518, 9521
 \iow_term:x .. 1093, 1095, 7759, 9518, 9520
 \iow_wrap:nnnN . 173, 7737, 7738, 7784,
 7790, 7906, 8329, 8353, 9565, 9565
- J**
- \jobname 518
- K**
- \kern 396
 \keys_define:nn ... 157, 8490, 8490, 9054
 \keys_if_choice_exist:nnn 9028
 \keys_if_choice_exist:nnnTF .. 166, 9028
 \keys_if_choice_exist_p:nnn .. 166, 9028
 \keys_if_exist:nn 9022
 \keys_if_exist:nnTF 166, 9022
 \keys_if_exist_p:nn 166, 9022
 \keys_set:nn ... 163, 8674, 8678, 8681,
 8818, 8818, 8826, 8842, 8862, 8871
 \keys_set:no 8818
 \keys_set:nV 8818
 \keys_set:nv 8818
 \keys_set_filter:nnn 8846, 8852, 8857, 8865
 \keys_set_filter:nnnN 165, 8846, 8846, 8848
 \keys_set_filter:nnV 8846
 \keys_set_filter:nnv\keys_set_filter:nnno
 8846
 \keys_set_filter:nnVN 8846
 \keys_set_filter:nnvN\keys_set_filter:nnno
 8846
 \keys_set_groups:nnn 166, 8846, 8866, 8874
 \keys_set_groups:nnV 8846
 \keys_set_groups:nnv\keys_set_groups:nnno
 8846
 \keys_set_known:nn 8828, 8834, 8839, 8845
 \keys_set_known:nnN 164, 8828, 8828, 8830
 \keys_set_known:no 8828
 \keys_set_known:noN 8828
 \keys_set_known:nV 8828
 \keys_set_known:nv 8828
 \keys_set_known:nVN 8828
 \keys_set_known:nvN 8828

\keys_show:nn 166, 9034, 9034
 \keyval_parse:NNn 167, 8455, 8455, 8495, 8823

 L
 \l@expl@check@declarations@bool 1139, 1939, 4406, 8193
 \l@expl@log@functions@bool .. 1124, 7647
 \l__box_angle_fp 197, 14823, 14823, 14840, 14841, 14842, 16218, 16221, 16223
 \l__box_bottom_dim 14826, 14827, 14850, 14907, 14911, 14916, 14922, 14927, 14931, 14940, 14942, 14960, 14970, 14978, 15009, 15057, 15063
 \l__box_bottom_new_dim 14830, 14831, 14876, 14908, 14919, 14930, 14941, 14977, 15062, 15082, 15086
 \l__box_cos_fp 197, 14824, 14824, 14842, 14855, 14860, 14887, 14899, 16199, 16200, 16201, 16205, 16214
 \l__box_internal_box 197, 14834, 14834, 14864, 14865, 14871, 14875, 14876, 14877, 14879, 15073, 15081, 15082, 15085, 15086, 15093, 15098, 15102, 15112, 15120, 15123, 15125, 15128, 15131, 15133, 15135, 15137, 15138, 15139, 15140, 15143, 15145, 15146, 15148, 15150, 15155, 15163, 15166, 15168, 15171, 15172, 15173, 15177, 15178, 15179, 15187, 15190, 15192, 15194, 16198
 \l__box_left_dim 14826, 14828, 14852, 14907, 14909, 14918, 14922, 14927, 14933, 14938, 14942, 14972, 15059
 \l__box_left_new_dim 14830, 14832, 14867, 14878, 14910, 14921, 14932, 14943
 \l__box_right_dim 14826, 14829, 14851, 14905, 14911, 14916, 14920, 14929, 14931, 14940, 14944, 14956, 14971, 15024, 15038, 15058, 15065
 \l__box_right_new_dim . 14830, 14833, 14878, 14912, 14923, 14934, 14945, 14976, 15064, 15090, 15092, 15098
 \l__box_scale_x_fp 197, 14947, 14947, 14955, 14994, 15011, 15023, 15025, 15037, 15054, 15065, 15088, 16233, 16238, 16246
 \l__box_scale_y_fp 197, 14947, 14948, 14957, 14978, 14980, 14989, 14994, 15006, 15011, 15025, 15039, 15055, 15061, 15063, 15079, 16234, 16239, 16248
 \l__box_sin_fp 197, 14824, 14825, 14841, 14853, 14888, 14898, 16202, 16206, 16209, 16211
 \l__box_top_dim 14826, 14826, 14849, 14905, 14909, 14918, 14920, 14929, 14933, 14938, 14944, 14960, 14969, 14980, 14992, 15009, 15042, 15056, 15061
 \l__box_top_new_dim 14830, 14830, 14875, 14906, 14917, 14928, 14939, 14979, 15060, 15081, 15085
 \l__clist_internal_clist 5768, 5768, 5875, 5876, 5888, 5889, 6067, 6068, 6131, 6132, 6148, 6149, 6285, 6286, 8252
 \l__clist_internal_remove_clist 5979, 5979, 5986, 5989, 5990, 5992
 \l__coffin_aligned_coffin 6962, 6964, 7186, 7187, 7191, 7197, 7199, 7200, 7216, 7217, 7223, 7224, 7225, 7226, 7227, 7229, 7231, 7235, 7236, 7241, 7242, 7243, 7244, 7245, 7279, 7294, 7340, 7341, 7546, 7553, 7555, 7557, 7559
 \l__coffin_aligned_internal_coffin . 6962, 6965, 7258, 7265
 \l__coffin_bottom_corner_dim 15202, 15204, 15226, 15230, 15300, 15311, 15312, 15332, 15340
 \l__coffin_bounding_prop 15200, 15200, 15215, 15243, 15245, 15248, 15250, 15256, 15319
 \l__coffin_bounding_shift_dim 15201, 15201, 15224, 15318, 15324, 15325
 \l__coffin_cos_fp 15198, 15199, 15209, 15283, 15292
 \l__coffin_display_coffin 7344, 7344, 7472, 7478, 7548, 7549, 7554, 7556, 7558, 7559
 \l__coffin_display_coord_coffin 7344, 7345, 7410, 7430, 7446, 7493, 7513, 7532
 \l__coffin_display_font_tl 7389, 7389, 7391, 7394, 7418, 7501

\l__coffin_display_handles_prop
 7347, 7347, 7348, 7350, 7352, 7354,
 7356, 7358, 7360, 7362, 7364, 7366,
 7368, 7370, 7372, 7374, 7376, 7378,
 7380, 7382, 7421, 7425, 7504, 7508
 \l__coffin_display_offset_dim 7384,
 7384, 7385, 7447, 7448, 7533, 7534
 \l__coffin_display_pole_coffin
 . . 7344, 7346, 7398, 7409, 7453, 7491
 \l__coffin_display_poles_prop
 7388, 7388,
 7463, 7468, 7471, 7473, 7475, 7482
 \l__coffin_display_x_dim
 7386, 7386, 7488, 7543
 \l__coffin_display_y_dim
 7386, 7387, 7489, 7545
 \l__coffin_error_bool 6803, 6803, 7081,
 7085, 7099, 7114, 7145, 7484, 7486
 \l__coffin_internal_box . 6781, 6781,
 6890, 6894, 6898, 6937, 6942, 6947,
 15220, 15229, 15231, 15232, 15234
 \l__coffin_internal_dim
 . . . 6781, 6782, 7192, 7194, 7195,
 15247, 15249, 15251, 15380, 15383
 \l__coffin_internal_tl
 6781, 6783, 6790, 6791,
 6792, 6793, 6794, 6795, 6796, 6797,
 6798, 6799, 6800, 7277, 7278, 7280,
 7422, 7423, 7426, 7427, 7435, 7440,
 7505, 7506, 7509, 7510, 7519, 7524
 \l__coffin_left_corner_dim
 . . . 15202, 15202, 15224, 15233,
 15301, 15307, 15308, 15331, 15339
 \l__coffin_offset_x_dim
 6804, 6804, 7189,
 7190, 7193, 7201, 7203, 7205, 7211,
 7214, 7234, 7254, 7262, 7542, 7550
 \l__coffin_offset_y_dim
 . . . 6804, 6805, 7204, 7206, 7211,
 7214, 7234, 7256, 7263, 7544, 7551
 \l__coffin_pole_a_tl
 . . . 6806, 6806, 7079, 7084, 7303,
 7306, 7307, 7310, 7465, 7467, 7470
 \l__coffin_pole_b_tl
 . . . 6806, 6807, 7080, 7084, 7304, 7306,
 7308, 7310, 7466, 7467, 7469, 7470
 \l__coffin_right_corner_dim . 15202,
 15203, 15233, 15299, 15309, 15310
 \l__coffin_scale_x_fp
 15344, 15344, 15350,
 15367, 15377, 15379, 15385, 15393
 \l__coffin_scale_y_fp . 15344, 15345,
 15352, 15378, 15379, 15383, 15395
 \l__coffin_scaled_total_height_dim .
 15346, 15346, 15382, 15387
 \l__coffin_scaled_width_dim
 15346, 15347, 15384, 15387
 \l__coffin_sin_fp
 . . 15198, 15198, 15208, 15284, 15291
 \l__coffin_slope_x_fp
 . . 6801, 6801, 7139, 7144, 7152, 7158
 \l__coffin_slope_y_fp
 . . 6801, 6802, 7141, 7144, 7153, 7158
 \l__coffin_top_corner_dim . . 15202,
 15205, 15230, 15298, 15313, 15314
 \l__coffin_x_dim 6808, 6808, 7088, 7097,
 7117, 7120, 7127, 7136, 7147, 7162,
 7251, 7255, 7274, 7282, 7488, 7540,
 15255, 15257, 15261, 15263, 15267,
 15272, 15399, 15401, 15405, 15408
 \l__coffin_x_prime_dim . . 6808, 6810,
 7251, 7255, 7540, 7543, 15269, 15273
 \l__coffin_y_dim . . 6808, 6809, 7089,
 7102, 7105, 7112, 7129, 7134, 7163,
 7252, 7257, 7275, 7282, 7489, 7541,
 15255, 15257, 15261, 15263, 15267,
 15272, 15399, 15401, 15405, 15408
 \l__coffin_y_prime_dim . . 6808, 6811,
 7252, 7257, 7541, 7545, 15269, 15274
 \l__driver_color_stack_int
 16263, 16264, 16280, 16285
 \l__driver_current_color_tl
 . . . 16253, 16253, 16255, 16258,
 16261, 16269, 16281, 16290, 16294
 \l__exp_internal_tl . . . 33, 779, 779,
 783, 784, 1462, 1481, 1482, 1657, 1658
 \l__file_internal_name_tl
 . . . 174, 9172, 9172, 9185, 9186,
 9187, 9192, 9199, 9200, 9202, 9203,
 9208, 9213, 9259, 9260, 9267, 9358,
 9359, 9361, 9370, 9371, 9374, 15431,
 15438, 15444, 15451, 15670, 15693
 \l__file_internal_seq . . 9177, 9178,
 9238, 9240, 9316, 9322, 9327, 9329
 \l__file_internal_tl 9171, 9171, 9298, 9299
 \l__file_saved_search_path_seq
 9174, 9175, 9237, 9254
 \l__file_search_path_seq
 9173, 9173, 9237, 9239,
 9240, 9243, 9254, 9306, 9307, 9312

\l__fp_division_by_zero_flag_token .	\l__keys_property_tl
. 10023 , 10024 8483 , 8483 , 8512 , 8515 , 8518 ,
\l__fp_invalid_operation_flag_token	8529 , 8540 , 8547 , 8548 , 8551 , 8555
. 10023 , 10023	\l__keys_selective_bool 8484 ,
\l__fp_overflow_flag_token 10023 , 10025	8484 , 8859 , 8863 , 8868 , 8872 , 8890
\l__fp_underflow_flag_token 10023 , 10026	\l__keys_selective_seq
\l__ior_internal_tl 8486 , 8486 , 8861 , 8870 , 8940
. 15459 , 15478 , 15481 , 15485	\l__keys_tmp_bool
\l__ior_stream_tl 8489 , 8489 , 8939 , 8946 , 8951
. 9346 , 9346 , 9381 , 9389 , 9397	\l__keys_unused_clist
\l__iow_current_indentation_int 8487 , 8487 , 8829 , 8833 , 8835 ,
. 9527 , 9529 ,	8836 , 8847 , 8851 , 8853 , 8854 , 9008
9590 , 9635 , 9650 , 9672 , 9678 , 9680	\l__keyval_key_tl
\l__iow_current_indentation_tl 9530 , 8381 , 8381 , 8424 , 8428 , 8435 , 8451
9532 , 9591 , 9633 , 9653 , 9673 , 9679	\l__keyval_parse_tl
\l__iow_current_line_int 8383 , 8384 , 8399 , 8403 , 8420 , 8447
. 9527 , 9527 , 9592 ,	\l__keyval_sanitise_tl
9621 , 9622 , 9634 , 9640 , 9647 , 9666 8383 , 8383 , 8396 , 8397 , 8398 , 8401
\l__iow_current_line_tl	\l__keyval_value_tl 8381 , 8382 , 8446 , 8452
. 9530 , 9530 , 9593 , 9632 ,	\l__msg_class_loop_seq 7921 , 7921 ,
9638 , 9646 , 9652 , 9665 , 9667 , 9685	8011 , 8019 , 8028 , 8029 , 8032 , 8034
\l__iow_current_word_int	\l__msg_class_tl
. 9527 , 9528 , 9619 , 9621 , 9649	7917 , 7917 , 7933 , 7945 , 7966 , 7970 ,
\l__iow_current_word_tl 9530 , 9531 ,	7973 , 7981 , 8020 , 8022 , 8024 , 8037
9612 , 9613 , 9620 , 9633 , 9639 , 9653	\l__msg_current_class_tl
\l__iow_line_start_bool 7917 , 7918 , 7928 ,
. 9535 , 9535 , 9594 , 9629 , 9631 , 9668	7965 , 7970 , 7973 , 7981 , 8010 , 8024
\l__iow_newline_tl 9534 , 9534 ,	\l__msg_hierarchy_seq
9585 , 9586 , 9587 , 9589 , 9646 , 9665 7920 , 7920 , 7948 , 7958 , 7963
\l__iow_stream_tl	\l__msg_internal_tl
. 9459 , 9459 , 9474 , 9482 , 9490 7630 , 7630 , 8357 , 8358 , 8360
\l__iow_target_count_int	\l__msg_redirect_prop
. 9526 , 9526 , 9588 , 9622 7919 , 7919 , 7945 , 7990 , 7993
\l__iow_wrap_tl 9533 , 9533 , 9580 , 9583 ,	\l__peek_search_tl
9597 , 9599 , 9605 , 9645 , 9664 , 9684 2826 , 2826 , 2844 , 2865 , 2905 , 2915
\l__kernel_expl_bool	\l__peek_search_token
. 8 , 161 , 161 , 164 , 179 , 193 2825 , 2825 , 2843 , 2864 , 2882
\l__keys_filtered_bool 8484 , 8485 ,	\l__prop_internal_tl 132 , 6299 ,
8860 , 8869 , 8926 , 8932 , 8953 , 8958	6299 , 6431 , 6437 , 6438 , 6454 , 6461
\l__keys_groups_clist	\l__seq_internal_a_tl
. 8477 , 8477 , 8666 , 8668 , 8923 , 8942 5248 , 5248 , 5312 , 5316 , 5322 ,
\l__keys_module_tl 8479 ,	5327 , 5329 , 5411 , 5416 , 5456 , 5460
8479 , 8491 , 8494 , 8496 , 8524 , 8678 ,	\l__seq_internal_b_tl
8819 , 8822 , 8824 , 8888 , 8990 , 8993 5248 , 5249 , 5407 , 5411 , 5459 , 5460
\l__keys_no_value_bool	\l__seq_remove_seq
. 8480 , 8480 , 8501 , 8506 , 8544 , 5379 , 5379 , 5386 , 5389 , 5390 , 5392
8877 , 8882 , 8899 , 8909 , 8965 , 9011	\l__tl_internal_a_tl 4624 , 4627 ,
\l__keys_only_known_bool	4629 , 4637 , 15672 , 15678 , 15692 , 15695
. 8481 , 8481 , 8841 , 8843 , 8987	\l__tl_internal_b_tl 4624 , 4628 , 4629 , 4638

- \l_char_active_seq 52, 2471, 2471, 2484, 9183
- \l_char_special_seq . 52, 2471, 2488, 2489
- \l_iow_line_count_int 173, 9524, 9524, 9525, 9589
- \l_keys_choice_int 162, 8475, 8475, 8638, 8641, 8645, 8646, 9119, 9129, 9133, 9134
- \l_keys_choice_tl 162, 8475, 8476, 8644, 9132
- \l_keys_key_tl 164, 8478, 8478, 8570, 8586, 8887, 8888, 9010
- \l_keys_path_tl 164, 8482, 8482, 8518, 8524, 8534, 8537, 8552, 8563, 8565, 8567, 8579, 8581, 8583, 8597, 8599, 8603, 8611, 8612, 8614, 8617, 8642, 8659, 8660, 8664, 8667, 8672, 8677, 8681, 8684, 8686, 8687, 8688, 8694, 8725, 8888, 8903, 8913, 8920, 8922, 8967, 8975, 8977, 8984, 8993, 9017, 9018, 9100, 9103, 9105, 9116, 9124, 9130, 9136
- \l_keys_value_tl 164, 8488, 8488, 8600, 8913, 8968, 8969, 8971, 9002, 9012
- \l_peek_token 57, 2823, 2823, 2832, 2882, 2894, 2914, 2923, 16006, 16007, 16008, 16011
- \l_tmpa_bool 37, 2010, 2010
- \l_tmpa_box 136, 6656, 6656
- \l_tmpa_clist 125, 6289, 6289
- \l_tmpa_coffin 144, 6966, 6966
- \l_tmpa_dim 82, 4110, 4110
- \l_tmpa_fp 182, 14816, 14816
- \l_tmpa_int 72, 3850, 3850
- \l_tmpa_muskip 88, 4262, 4262
- \l_tmpa_prop 132, 6327, 6327
- \l_tmpa_seq 116, 5760, 5760
- \l_tmpa_skip 85, 4200, 4200
- \l_tmpa_tl 5, 103, 5079, 5079
- \l_tmpb_bool 37, 2010, 2011
- \l_tmpb_box 136, 6656, 6657
- \l_tmpb_clist 125, 6289, 6290
- \l_tmpb_coffin 144, 6966, 6967
- \l_tmpb_dim 82, 4110, 4111
- \l_tmpb_fp 182, 14816, 14817
- \l_tmpb_int 72, 3850, 3851
- \l_tmpb_muskip 88, 4262, 4263
- \l_tmpb_prop 132, 6327, 6328
- \l_tmpb_seq 116, 5760, 5761
- \l_tmpb_skip 85, 4200, 4201
- \l_tmpb_tl 103, 5079, 5080
- \language 313
- \lastbox 470
- \lastkern 403
- \lastlinefit 583
- \lastnodetype 564
- \lastpenalty 509
- \lastskip 404
- \latelua 626
- \lccode 532
- \leaders 400
- \left 368
- \lefthyphenmin 424
- \leftskip 426
- \leqno 343
- \let 21, 198, 199, 213
- \limits 360
- \LineBreak 65, 66, 67, 68, 69, 70, 71, 72, 79, 87, 88, 89, 97, 99
- \linepenalty 416
- \lineskip 410
- \lineskiplimit 411
- \linewidth 6880, 6926
- \ln 13590, 13593
- \long 201, 232
- \LongText 63, 85, 111
- floor 188
- \looseness 428
- \lower 465
- \lowercase 504
- \lua_now_x:n 4325
- \luaescapestring 627
- \luatex_bodydir:D 631, 654, 677
- \luatex_catcodetable:D 622, 648
- \luatex_directlua:D .. 54, 623, 1435, 5106
- \luatex_expanded:D 624, 5119
- \luatex_if_engine:F 1413, 1438
- \luatex_if_engine:T 1412, 1437, 4320, 5102
- \luatex_if_engine:TF 23, 1412, 1414, 1439
- \luatex_if_engine_p: 23, 1412, 1421, 1443
- \luatex_initcatcodetable:D 625, 649
- \luatex_latelua:D 626, 650
- \luatex_luaescapestring:D 627, 651, 5117
- \luatex_luatexversion:D 628, 746
- \luatex_mathdir:D 632, 655
- \luatex_pagedir:D 633, 656, 678
- \luatex_pardir:D 634, 657
- \luatex_savecatcodetable:D 629, 652
- \luatex_textdir:D 635, 658
- \luatex_Uchar:D 630, 653

<code>\luatexbodydir</code>	654	<code>\mode_if_inner:</code>	2191
<code>\luatexcatcodetable</code>	648	<code>\mode_if_inner:TF</code>	40, 2191
<code>\luatexinitcatcodetable</code>	649	<code>\mode_if_inner_p:</code>	40, 2191
<code>\luatexlatelua</code>	650	<code>\mode_if_math:</code>	2193
<code>\luatexluaescapestring</code>	651	<code>\mode_if_math:TF</code>	40, 2193
<code>\luatexmathdir</code>	655	<code>\mode_if_math_p:</code>	40, 2193
<code>\luatexpagedir</code>	656	<code>\mode_if_vertical:</code>	2187
<code>\luatexpardir</code>	657	<code>\mode_if_vertical:TF</code>	40, 2187
<code>\luatexsavecatcodetable</code>	652	<code>\mode_if_vertical_p:</code>	40, 2187
<code>\luatextextdir</code>	658	<code>\month</code>	516
<code>\luatexUchar</code>	653	<code>\moveleft</code>	466
<code>\luatexversion</code>	26, 77, 628	<code>\moveright</code>	467
M			
<code>\M</code>	1727, 2565, 10476	<code>\msg_critical:nn</code>	7856
<code>cm</code>	191	<code>\msg_critical:nnn</code>	7856
<code>em</code>	191	<code>\msg_critical:nnnn</code>	7856
<code>mm</code>	191	<code>\msg_critical:nnnnn</code>	7856
<code>\m@ne</code>	735	<code>\msg_critical:nnnnnn</code>	149, 7856
<code>\mag</code>	312	<code>\msg_critical:nnx</code>	7856
<code>\mark</code>	314	<code>\msg_critical:nnxx</code>	7856
<code>\marks</code>	540	<code>\msg_critical:nnxxx</code>	7856
<code>\mathaccent</code>	325	<code>\msg_critical:nnxxxx</code>	7856
<code>\mathbin</code>	355	<code>\msg_critical_text:n</code>	147, 7796, 7797, 7859
<code>\mathchar</code>	326, 2631	<code>\msg_error:nn</code>	7867
<code>\mathchardef</code>	223	<code>\msg_error:nnn</code>	7867
<code>\mathchoice</code>	323	<code>\msg_error:nnnn</code>	7867
<code>\mathclose</code>	356	<code>\msg_error:nnnnn</code>	7867
<code>\mathcode</code>	534	<code>\msg_error:nnnnnn</code>	149, 7867
<code>\mathdir</code>	632	<code>\msg_error:nnx</code>	7867
<code>\mathinner</code>	357	<code>\msg_error:nnxx</code>	7867
<code>\mathop</code>	358	<code>\msg_error:nnxxx</code>	7867
<code>\mathopen</code>	362	<code>\msg_error:nnxxxx</code>	7867
<code>\mathord</code>	363	<code>\msg_error_text:n</code>	147, 7796, 7798, 7874
<code>\mathpunct</code>	364	<code>\msg_fatal:nn</code>	7845
<code>\mathrel</code>	365	<code>\msg_fatal:nnn</code>	7845
<code>\mathsurround</code>	376	<code>\msg_fatal:nnnn</code>	7845
<code>\maxdeadcycles</code>	446	<code>\msg_fatal:nnnnn</code>	7845
<code>\maxdepth</code>	447	<code>\msg_fatal:nnnnnn</code>	148, 7845
<code>\meaning</code>	506	<code>\msg_fatal:nnx</code>	7845
<code>\medmuskip</code>	377	<code>\msg_fatal:nnxx</code>	7845
<code>\message</code>	283	<code>\msg_fatal:nnxxx</code>	7845
<code>\MessageBreak</code>	97	<code>\msg_fatal:nnxxxx</code>	7845
<code>.meta:n</code>	160	<code>\msg_fatal_text:n</code>	147, 7796, 7796, 7848
<code>.meta:nn</code>	160	<code>\msg_gset:nnn</code>	7659, 7682
<code>\middle</code>	588	<code>\msg_gset:nnnn</code>	147, 7659, 7662, 7675, 7683
<code>\mkern</code>	330	<code>\msg_if_exist:nn</code>	7633
<code>\mode_if_horizontal:</code>	2189	<code>\msg_if_exist:nnT</code>	7640, 7650
<code>\mode_if_horizontal:TF</code>	40, 2189	<code>\msg_if_exist:nnTF</code>	147, 7633, 7924
<code>\mode_if_horizontal_p:</code>	40, 2189	<code>\msg_if_exist_p:nn</code>	147, 7633
		<code>\msg_info:nn</code>	7896
		<code>\msg_info:nnn</code>	7896

<code>\msg_info:nnnn</code>	7896	<code>\msg_warning:nnxxx</code>	7888
<code>\msg_info:nnnnn</code>	7896	<code>\msg_warning:nnxxxx</code>	7888 , 8094
<code>\msg_info:nnnnnn</code>	149 , 7896	<code>\msg_warning_text:n</code>	148 , 7796 , 7799 , 7892
<code>\msg_info:nnx</code>	7896	<code>\mskip</code>	327
<code>\msg_info:nnxx</code>	7896	<code>\muexpr</code>	576
<code>\msg_info:nnxxx</code>	7896	<code>.multichoice:</code>	160
<code>\msg_info:nnxxxx</code>	7896 , 8095	<code>.multichoices:nn</code>	160
<code>\msg_info_text:n</code> ..	148 , 7796 , 7800 , 7900	<code>.multichoices:on</code>	160
<code>\msg_interrupt:nnn</code>		<code>.multichoices:Vn</code>	160
...	151 , 7723 , 7723 , 7847 , 7858 , 7873	<code>.multichoices:xn</code>	160
<code>\msg_line_context:</code>	147 , 1113 , 1113 , 1132 , 7655 , 7716 , 7717 , 8196	<code>\multiply</code>	228
<code>\msg_line_number:</code>		<code>\muskip</code>	524
...	147 , 7716 , 7716 , 7721 , 8466	<code>\muskip_add:cn</code>	4242
<code>\msg_log:n</code>	152 , 7781 , 7781 , 7898	<code>\muskip_add:Nn</code> ..	87 , 4242 , 4242 , 4244 , 4245
<code>\msg_log:nn</code>	7904	<code>\muskip_const:cn</code>	4212
<code>\msg_log:nnn</code>	7904	<code>\muskip_const:Nn</code>	
<code>\msg_log:nnnn</code>	7904	...	86 , 4212 , 4212 , 4217 , 4260 , 4261
<code>\msg_log:nnnnn</code>	7904	<code>\muskip_eval:n</code>	87 , 4252 , 4252
<code>\msg_log:nnnnn</code>	7904	<code>\muskip_gadd:cn</code>	4242
<code>\msg_log:nnnnnn</code>	149 , 7904	<code>\muskip_gadd:Nn</code>	87 , 4242 , 4244 , 4246
<code>\msg_log:nnx</code>	7904	<code>\muskip_gset:cn</code>	4231
<code>\msg_log:nnxx</code>	7904	<code>\muskip_gset:Nn</code> ..	87 , 4215 , 4231 , 4233 , 4235
<code>\msg_log:nnxxx</code>	7904	<code>\muskip_gset_eq:cc</code>	4236
<code>\msg_log:nnxxxx</code>	7904	<code>\muskip_gset_eq:cN</code>	4236
<code>\msg_new:nnn</code>	7659 , 7664 , 8046	<code>\muskip_gset_eq:Nc</code>	4236
<code>\msg_new:nnnn</code> ..	146 , 7659 , 7659 , 7665 , 8044	<code>\muskip_gset_eq:NN</code>	
<code>\msg_none:nn</code>	7910	...	87 , 4236 , 4239 , 4240 , 4241
<code>\msg_none:nnn</code>	7910	<code>\muskip_gsub:cn</code>	4242
<code>\msg_none:nnnn</code>	7910	<code>\muskip_gsub:Nn</code>	87 , 4242 , 4249 , 4251
<code>\msg_none:nnnnn</code>	7910	<code>\muskip_gzero:c</code>	4218
<code>\msg_none:nnnnnn</code>	150 , 7910	<code>\muskip_gzero:N</code> ..	86 , 4218 , 4220 , 4222 , 4226
<code>\msg_none:nnx</code>	7910	<code>\muskip_gzero_new:c</code>	4223
<code>\msg_none:nnxx</code>	7910	<code>\muskip_gzero_new:N</code> ..	86 , 4223 , 4225 , 4228
<code>\msg_none:nnxxx</code>	7910	<code>\muskip_if_exist:c</code>	4230
<code>\msg_none:nnxxxx</code>	7910	<code>\muskip_if_exist:cTF</code>	4229
<code>\msg_redirect_class:nn</code> ..	150 , 7996 , 7996	<code>\muskip_if_exist:N</code>	4229
<code>\msg_redirect_module:nnn</code> ..	151 , 7996 , 7998	<code>\muskip_if_exist:NTF</code> ..	86 , 4224 , 4226 , 4229
<code>\msg_redirect_name:nnn</code> ..	151 , 7987 , 7987	<code>\muskip_if_exist_p:c</code>	4229
<code>\msg_see_documentation_text:n</code>		<code>\muskip_if_exist_p:N</code>	86 , 4229
...	148 , 7801 , 7801 , 7851 , 7862 , 7877	<code>\muskip_new:c</code>	4204
<code>\msg_set:nnn</code>	7659 , 7673 , 8050	<code>\muskip_new:N</code> ..	86 , 4204 , 4205 , 4211 , 4214 , 4224 , 4226 , 4262 , 4263 , 4264 , 4265
<code>\msg_set:nnnn</code> ..	147 , 7659 , 7666 , 7674 , 8048	<code>\muskip_set:cn</code>	4231
<code>\msg_term:n</code>	152 , 7781 , 7787 , 7890	<code>\muskip_set:Nn</code> ..	87 , 4231 , 4231 , 4233 , 4234
<code>\msg_warning:nn</code>	7888	<code>\muskip_set_eq:cc</code>	4236
<code>\msg_warning:nnn</code>	7888	<code>\muskip_set_eq:cN</code>	4236
<code>\msg_warning:nnnn</code>	7888	<code>\muskip_set_eq:Nc</code>	4236
<code>\msg_warning:nnnnn</code>	7888	<code>\muskip_set_eq:NN</code> ..	87 , 4236 , 4236 , 4237 , 4238
<code>\msg_warning:nnnnnn</code>	149 , 7888	<code>\muskip_show:c</code>	4256
<code>\msg_warning:nnx</code>	7888	<code>\muskip_show:N</code>	88 , 4256 , 4256 , 4257
<code>\msg_warning:nnxx</code>	7888		

\muskip_show:n	88, 4258, 4258	
\muskip_sub:cn	4242	
\muskip_sub:Nn	87, 4242, 4247, 4249, 4250	
\muskip_use:c	4254	
\muskip_use:N	87, 4253, 4254, 4254, 4255	
\muskip_zero:c	4218	
\muskip_zero:N	86, 4218, 4218, 4220, 4221, 4224	
\muskip_zero_new:c	4223	
\muskip_zero_new:N	86, 4223, 4223, 4227	
\muskipdef	222	
\mutoglu	582	
\my_map_dbl:nn	1	
N		
\N	1876	
in	191	
ln	188	
\newlinechar	82, 278	
\next	61, 80, 108, 119, 123, 125, 132	
inf	190	
\noalign	246	
\noboundary	381	
\noexpand	82, 93, 97, 110, 113, 239	
\noindent	407	
\nolimits	361	
\nonscript	341	
\nonstopmode	304	
\normalend	660, 663	
\normaleveryjob	669	
\normalhoffset	673	
\normalinput	662	
\normalitaliccorrection	672, 674	
\normallanguage	664	
\normalleft	680, 681	
\normalmathop	667	
\normalmiddle	682	
\normalmonth	668	
\normalouter	661	
\normalover	666	
\normalright	683	
\normalshowtokens	676	
\normalunexpanded	670	
\normalvcneter	665	
\normalvoffset	675	
\nulldelimiterspace	374	
\nullfont	492	
\number	501	
\numexpr	573	
O		
\O	15996	
\omit	247	
\openin	273	
\openout	274	
\or	270	
\or:	23, 73, 688, 690, 1236, 1237, 1238, 1239, 1240, 1241, 1242, 1243, 1244, 3031, 3571, 3572, 3573, 3574, 3575, 3576, 3577, 3578, 3579, 3580, 3581, 3582, 3583, 3584, 3585, 3586, 3587, 3588, 3589, 3590, 3591, 3592, 3593, 3594, 3595, 3604, 3605, 3606, 3607, 3608, 3609, 3610, 3611, 3612, 3613, 3614, 3615, 3616, 3617, 3618, 3619, 3620, 3621, 3622, 3623, 3624, 3625, 3626, 3627, 3628, 9793, 9794, 9795, 9941, 9942, 10281, 11441, 11442, 11443, 11479, 11882, 11883, 11884, 12012, 12096, 12182, 12183, 12184, 12185, 12186, 12187, 12188, 12189, 12190, 12267, 12270, 12606, 12852, 13068, 13093, 13099, 13100, 13101, 13102, 13103, 13251, 13286, 13288, 13296, 13386, 13390, 13391, 13392, 13393, 13394, 13395, 13396, 13397, 13398, 13399, 13406, 13407, 13408, 13409, 13410, 13411, 13412, 13413, 13414, 13415, 13422, 13423, 13424, 13425, 13426, 13427, 13428, 13429, 13430, 13431, 13438, 13439, 13440, 13441, 13442, 13443, 13444, 13445, 13446, 13447, 13454, 13455, 13456, 13457, 13458, 13459, 13460, 13461, 13462, 13463, 13470, 13471, 13472, 13473, 13474, 13475, 13476, 13477, 13478, 13479, 13497, 13543, 13546, 13555, 13666, 13689, 13736, 13741, 13751, 13756, 13766, 13771, 13781, 13786, 13796, 13801, 13811, 13816, 14253, 14273, 14274, 14318, 14403, 14406, 14418, 14424, 14471, 14473, 14474, 14484, 14490, 14538, 14539, 14550, 14588, 14589, 14599, 14656, 14657	
cos	188	
cosd	189	
cot	188	
cotd	189	
round	188	

<code>\outer</code>	233	<code>\pdfsetmatrix</code>	615
<code>\output</code>	448	<code>\pdfstrcmp</code>	21, 65, 620
<code>\outputpenalty</code>	458	<code>\pdftex_if_engine:F</code>	1416, 1427, 1441
<code>\over</code>	335	<code>\pdftex_if_engine:T</code>	1415, 1426, 1440
<code>\overfullrule</code>	486	<code>\pdftex_if_engine:TF</code>	
<code>\overline</code>	366	23, 1412, 1417, 1428, 1442, 3135
<code>\overwithdelims</code>	336	<code>\pdftex_if_engine_p:</code>	
		23, 1412, 1422, 1432, 1444
P			
<code>\P</code>	1729, 10390, 10474	<code>\pdftex_pdfcolorstack:D</code>	
<code>bp</code>	191	602, 16276, 16280, 16285
<code>sp</code>	191	<code>\pdftex_pdfcompresslevel:D</code> ..	603, 16077
<code>\PackageError</code>	101, 110	<code>\pdftex_pdfcreationdate:D</code>	601
<code>\pagedepth</code>	450	<code>\pdftex_pdfdecimaldigits:D</code> ..	604, 16074
<code>\pagedir</code>	633	<code>\pdftex_pdfhorigin:D</code>	605, 16072
<code>\pagediscards</code>	591	<code>\pdftex_pdfinfo:D</code>	606
<code>\pagefilllstretch</code>	454	<code>\pdftex_pdflastxform:D</code>	607
<code>\pagefillstretch</code>	453	<code>\pdftex_pdfliteral:D</code>	
<code>\pagefilstretch</code>	452	608, 16105, 16107, 16127, 16137
<code>\pagegoal</code>	456	<code>\pdftex_pdfminorversion:D</code> ...	609, 16076
<code>\pageshrink</code>	455	<code>\pdftex_pdfobjcompresslevel:D</code>	610, 16078
<code>\pagestretch</code>	451	<code>\pdftex_pdfoutput:D</code>	611, 16067
<code>\pagetotal</code>	457	<code>\pdftex_pdfpkresolution:D</code> ...	616, 16075
<code>\par</code>	406, 6731, 6732, 6734, 6736, 6738, 6743, 6749, 6762	<code>\pdftex_pdfrefxform:D</code>	612
<code>\pardir</code>	634	<code>\pdftex_pdfrestore:D</code>	613, 16101
<code>\parfillskip</code>	437	<code>\pdftex_pdfsave:D</code>	614, 16098, 16100
<code>\parindent</code>	430	<code>\pdftex_pdfsetmatrix:D</code>	615, 16150, 16153
<code>\parshape</code>	422	<code>\pdftex_pdftextrevision:D</code>	617
<code>\parshapedimen</code>	572	<code>\pdftex_pdfvorigin:D</code>	618, 16073
<code>\parshapeindent</code>	570	<code>\pdftex_pdfxform:D</code>	619
<code>\parshapelength</code>	571	<code>\pdftex_strcmp:D</code>	620, 5101
<code>\parskip</code>	429	<code>\pdftexrevision</code>	617
<code>\patterns</code>	512	<code>\pdfvorigin</code>	618
<code>\pausing</code>	299	<code>\pdfxform</code>	619
<code>\pdfcolorstack</code>	602	<code>\peek_after:Nw</code>	
<code>\pdfcompresslevel</code>	603	57, 2831, 2831, 2856, 2874, 2924
<code>\pdfcreationdate</code>	601	<code>\peek_catcode:Ntf</code>	57, 2948
<code>\pdfdecimaldigits</code>	604	<code>\peek_catcode_ignore_spaces:Ntf</code>	57, 2948
<code>\pdfhorigin</code>	605	<code>\peek_catcode_remove:Ntf</code>	58, 2948
<code>\pdfinfo</code>	606	58, 2948
<code>\pdflastxform</code>	607	<code>\peek_charcode:Ntf</code>	58, 2964
<code>\pdfliteral</code>	608	<code>\peek_charcode_ignore_spaces:Ntf</code>	58, 2964
<code>\pdfminorversion</code>	609	58, 2964
<code>\pdfobjcompresslevel</code>	610	<code>\peek_charcode_remove:Ntf</code>	58, 2964
<code>\pdfoutput</code>	611	<code>\peek_charcode_remove_ignore_spaces:Ntf</code>	59, 2964
<code>\pdfpkresolution</code>	616	59, 2964
<code>\pdfrefxform</code>	612	<code>\peek_gafter:Nw</code>	57, 2831, 2833
<code>\pdfrestore</code>	613	<code>\peek_meaning:Ntf</code>	59, 2980
<code>\pdfsave</code>	614	<code>\peek_meaning_ignore_spaces:Ntf</code>	59, 2980
		<code>\peek_meaning_remove:Ntf</code>	59, 2980

- \peek_meaning_remove_ignore_spaces:NTF 59, 2980
- \peek_N_type:F 16035
- \peek_N_type:T 16033
- \peek_N_type:TF 204, 15995, 16031
- \penalty 507
- \postdisplaypenalty 354
- \predisplaydirection 598
- \predisplaypenalty 353
- \predisplaysize 352
- \pretolerance 433
- \prevdepth 480
- \prevgraf 439
- \prg_break: 2351, 2353
- \prg_do_nothing: 10, 929, 930, 1405, 1446, 1446, 1456, 1796, 1801, 4472, 4486, 4546, 4551, 5318, 5325, 5563, 5565, 5923, 6257, 6261, 6268, 9990, 10042, 10076, 10102, 10110, 11450, 14726, 15669
- \prg_new_conditional:Nnn 832, 834, 1909, 2289, 2297, 2309, 2318, 9424
- \prg_new_conditional:Npnn 34, 819, 821, 1365, 1909, 1982, 2016, 2187, 2189, 2191, 2193, 2491, 2496, 2501, 2506, 2513, 2519, 2524, 2529, 2534, 2539, 2544, 2549, 2554, 2559, 2573, 2587, 2592, 2615, 2624, 2634, 2652, 2676, 2694, 2712, 2730, 2742, 2751, 2771, 3208, 3261, 3298, 3306, 3949, 3954, 4161, 4173, 4567, 4577, 4589, 4610, 4612, 4658, 4935, 4954, 4973, 5008, 5014, 5029, 5131, 5145, 5440, 6047, 6467, 6476, 6631, 6633, 6643, 6812, 7633, 8973, 9022, 9028, 10015, 10910, 11590, 11603, 15561
- \prg_new_eq_conditional:NNn 36, 920, 922, 1909, 2014, 2015, 3160, 3161, 3891, 3892, 4138, 4139, 4229, 4230, 4316, 4317, 5347, 5348, 5748, 5749, 5750, 5751, 5752, 5753, 5834, 5835, 6045, 6046, 6465, 6466, 6599, 6600, 11588, 11589
- \prg_new_protected_conditional:Nnn 832, 838, 1909
- \prg_new_protected_conditional:Npnn 34, 819, 825, 1909, 4624, 4645, 5452, 5562, 5564, 5572, 5574, 5576, 5578, 5932, 5944, 5946, 6061, 6065, 6401, 6411, 6506, 9257, 9363
- \prg_replicate:nn 40, 2143, 2143, 8244, 9680, 12794, 13642, 13701, 13708, 13861, 14065, 14066, 14068, 14070, 14083, 14120, 14595, 14622, 14630
- \prg_return_false: 36, 815, 817, 1024, 1029, 1042, 1047, 1055, 1072, 1368, 1909, 1987, 2021, 2188, 2190, 2192, 2194, 2294, 2302, 2315, 2324, 2494, 2499, 2504, 2509, 2516, 2522, 2527, 2532, 2537, 2542, 2547, 2552, 2557, 2562, 2583, 2590, 2595, 2600, 2637, 2640, 2659, 2662, 2679, 2682, 2697, 2700, 2715, 2718, 2774, 2793, 2810, 2819, 3206, 3238, 3243, 3266, 3303, 3309, 3952, 3973, 3988, 3989, 4168, 4176, 4582, 4594, 4607, 4617, 4634, 4649, 4664, 4947, 4970, 4986, 4994, 5004, 5024, 5038, 5128, 5135, 5148, 5445, 5466, 5489, 5935, 5951, 6050, 6075, 6409, 6419, 6470, 6492, 6513, 6632, 6634, 6644, 6818, 6820, 7636, 8979, 8981, 9026, 9032, 9261, 9372, 9434, 10020, 10924, 10936, 11598, 11611
- \prg_return_true: 36, 815, 815, 1027, 1044, 1052, 1057, 1070, 1075, 1368, 1909, 1985, 2019, 2188, 2190, 2192, 2194, 2292, 2300, 2313, 2322, 2494, 2499, 2504, 2509, 2516, 2522, 2527, 2532, 2537, 2542, 2547, 2552, 2557, 2562, 2581, 2590, 2598, 2654, 2656, 2791, 2817, 3238, 3264, 3301, 3311, 3952, 3989, 4166, 4177, 4580, 4592, 4605, 4615, 4631, 4649, 4662, 4945, 4968, 4984, 5002, 5026, 5037, 5126, 5135, 5148, 5443, 5470, 5492, 5938, 5954, 6051, 6075, 6407, 6417, 6470, 6494, 6511, 6632, 6634, 6644, 6817, 7636, 8978, 9025, 9031, 9262, 9375, 9429, 9432, 9438, 10018, 10919, 10942, 11600, 11609
- \prg_set_conditional:Nnn . 832, 832, 1909
- \prg_set_conditional:Npnn 34, 819, 819, 1021, 1033, 1049, 1061, 1909
- \prg_set_eq_conditional:NNn 36, 920, 920, 1909
- \prg_set_protected_conditional:Nnn . 832, 836, 1909
- \prg_set_protected_conditional:Npnn 34, 819, 823, 1909

<code>\prop_clear:c</code>	6307 , 7198	<code>\prop_gput:Noo</code>	6427
<code>\prop_clear:N</code> .	127 , 6307 , 6307 , 6309 , 6314	<code>\prop_gput:NvN</code>	6427 , 9398 , 9491
<code>\prop_clear_new:c</code>	6313 , 6847 , 6848 , 8653	<code>\prop_gput:NvV</code>	6427
<code>\prop_clear_new:N</code> .	127 , 6313 , 6313 , 6315	<code>\prop_gput_if_new:cnN</code>	6448
<code>\prop_gclear:c</code>	6307	<code>\prop_gput_if_new:Nnn</code> 128 , 6448 , 6450 , 6464	
<code>\prop_gclear:N</code> 127 , 6307 , 6310 , 6312 , 6317		<code>\prop_gremove:cn</code>	6342
<code>\prop_gclear_new:c</code>	6313	<code>\prop_gremove:cV</code>	6342
<code>\prop_gclear_new:N</code> .	127 , 6313 , 6316 , 6318	<code>\prop_gremove:Nn</code> 129 , 6342 , 6348 , 6356 , 6357	
<code>\prop_get:cn</code>	6562 , 6563	<code>\prop_gremove:Nv</code>	6342 , 9406 , 9499
<code>\prop_get:cnN</code>	6358	<code>\prop_gset_eq:cc</code> .	6319 , 6326 , 7000 , 7002
<code>\prop_get:cnNF</code>	6976 , 8967	<code>\prop_gset_eq:cN</code> .	6319 , 6325 , 6849 , 6851
<code>\prop_get:cnNT</code>	8020	<code>\prop_gset_eq:Nc</code>	6319 , 6324
<code>\prop_get:cnNTF</code> .	6506 , 7965 , 8599 , 8922	<code>\prop_gset_eq:NN</code> .	127 , 6311 , 6319 , 6323
<code>\prop_get:coN</code>	6358	<code>\prop_if_empty:cTF</code>	6467
<code>\prop_get:coNTF</code>	6506	<code>\prop_if_empty:N</code>	6467
<code>\prop_get:cVN</code>	6358	<code>\prop_if_empty:Nf</code>	6474
<code>\prop_get:cVNTF</code>	6506	<code>\prop_if_empty:NT</code>	6473
<code>\prop_get:Nn</code>	6562 , 6562	<code>\prop_if_empty:NTF</code>	
<code>\prop_get:NnN</code> .	128 , 6358 , 6358 , 6364 , 6365 , 6506 , 7421 , 7425 , 7504 , 7508	129 , 6467 , 6475 , 8260 , 9418
<code>\prop_get:NnNF</code>	6516 , 6519	<code>\prop_if_empty_p:c</code>	6467
<code>\prop_get:NnNT</code>	6515 , 6518	<code>\prop_if_empty_p:N</code>	129 , 6467 , 6472
<code>\prop_get:NnNTF</code> 130 , 6506 , 6517 , 6520 , 7945		<code>\prop_if_exist:c</code>	6466
<code>\prop_get:NoN</code>	6358	<code>\prop_if_exist:cT</code>	8659 , 8664 , 8684
<code>\prop_get:NoNTF</code>	6506	<code>\prop_if_exist:cTF</code> 6465 , 8597 , 8920 , 8975	
<code>\prop_get:NvN</code>	6358	<code>\prop_if_exist:N</code>	6465
<code>\prop_get:NvNTF</code>	6506	<code>\prop_if_exist:NTF</code> .	129 , 6314 , 6317 , 6465
<code>\prop_gpop:cnN</code>	6366	<code>\prop_if_exist_p:c</code>	6465
<code>\prop_gpop:cnNTF</code>	6401	<code>\prop_if_exist_p:N</code>	129 , 6465
<code>\prop_gpop:coN</code>	6366	<code>\prop_if_in:cnTF</code>	6476 , 8977
<code>\prop_gpop:NnN</code>		<code>\prop_if_in:coTF</code>	6476
...	128 , 6366 , 6375 , 6386 , 6387 , 6411	<code>\prop_if_in:cVTF</code>	6476
<code>\prop_gpop:NnNF</code>	6425	<code>\prop_if_in:Nn</code>	6476
<code>\prop_gpop:NnNT</code>	6424	<code>\prop_if_in:NnF</code>	6502 , 6503
<code>\prop_gpop:NnNTF</code>	130 , 6401 , 6426	<code>\prop_if_in:NnT</code>	6500 , 6501
<code>\prop_gpop:NoN</code>	6366	<code>\prop_if_in:NnTF</code> .	129 , 6476 , 6504 , 6505
<code>\prop_gput:cnN</code>	6427	<code>\prop_if_in:NoTF</code>	6476
<code>\prop_gput:cno</code>	6427	<code>\prop_if_in:NvTF</code>	6476
<code>\prop_gput:cnV</code>	6427	<code>\prop_if_in_p:cn</code>	6476
<code>\prop_gput:cnx</code>	6427	<code>\prop_if_in_p:co</code>	6476
<code>\prop_gput:con</code>	6427	<code>\prop_if_in_p:cV</code>	6476
<code>\prop_gput:coo</code>	6427	<code>\prop_if_in_p:Nn</code> .	129 , 6476 , 6498 , 6499
<code>\prop_gput:cVn</code>	6427	<code>\prop_if_in_p:No</code>	6476
<code>\prop_gput:cVv</code>	6427	<code>\prop_if_in_p:Nv</code>	6476
<code>\prop_gput:Nnn</code>		<code>\prop_item:cn</code>	6388 , 6563
...	128 , 6427 , 6428 , 6444 , 6446 , 9349	<code>\prop_item:Nn</code> .	129 , 6388 , 6388 , 6400 , 6562
<code>\prop_gput:Nno</code>	6427	<code>\prop_map_break:</code> 131 , 6525 , 6530 , 6544 , 6552 , 6552 , 6553 , 6555 , 15491 , 15496	
<code>\prop_gput:NnV</code>	6427	<code>\prop_map_break:n</code>	131 , 6552 , 6554
<code>\prop_gput:Nnx</code>	6427	<code>\prop_map_function:cc</code>	6521
<code>\prop_gput:Non</code>	6427	<code>\prop_map_function:cN</code>	6521 , 7567

\prop_map_function:Nc	6521	\prop_put:Nv	6427
\prop_map_function:NN	130, 6521, 6521, 6535, 6536, 6559, 9421	\prop_put:NVV	6427
\prop_map_inline:cn	6537, 7269, 7288, 15210, 15212, 15235, 15237, 15302, 15363, 15365, 15369, 15371	\prop_put_if_new:cnn	6448
\prop_map_inline:Nn	131, 6537, 6537, 6551, 7473, 7482, 15215, 15319	\prop_put_if_new:Nnn	128, 6448, 6448, 6463
\prop_map_tokens:cn	15487	\prop_remove:cn	6342, 8005, 8686, 8687
\prop_map_tokens:Nn	200, 15487, 15487, 15501	\prop_remove:cV	6342
\prop_new:c	6301, 7810	\prop_remove:Nn	129, 6342, 6342, 6354, 6355, 7468, 7471, 7475, 7990
\prop_new:N	127, 6301, 6301, 6306, 6314, 6317, 6327, 6328, 6329, 6330, 6784, 6789, 7347, 7388, 7919, 9347, 9460, 15200	\prop_remove:NV	6342
\prop_pop:cnN	6366	\prop_set_eq:cc	6319, 6322, 6993, 6995, 7228
\prop_pop:cnNTF	6401	\prop_set_eq:cN	6319, 6321, 6986, 6988
\prop_pop:coN	6366	\prop_set_eq:Nc	6319, 6320, 7463
\prop_pop:NnN	128, 6366, 6366, 6384, 6385, 6401	\prop_set_eq:NN	127, 6308, 6319, 6319
\prop_pop:NnNF	6422	\prop_show:c	6556
\prop_pop:NnNT	6421	\prop_show:N	131, 6556, 6556, 6561
\prop_pop:NnNTF	130, 6401, 6423	\protected	134, 136, 138, 162, 600, 2759
\prop_pop:NoN	6366	\protected@edef	9608
\prop_put:cnn	6427, 7028, 8009, 8026, 8612, 8660, 8688	\ProvidesExplClass	7
\prop_put:cno	6427	\ProvidesExplFile	7, 16047
\prop_put:cnV	6427, 8667	\ProvidesExplPackage	7
\prop_put:cnx	6427, 7034, 7036, 7038, 7040, 7045, 7050, 7055, 7062, 7069, 7293, 15262, 15329, 15337, 15400, 15414, 15421		
\prop_put:con	6427		
\prop_put:coo	6427		
\prop_put:cVn	6427		
\prop_put:cVV	6427		
\prop_put:Nnn	128, 6427, 6427, 6440, 6442, 6785, 6786, 6787, 6788, 7348, 7350, 7352, 7354, 7356, 7358, 7360, 7362, 7364, 7366, 7368, 7370, 7372, 7374, 7376, 7378, 7380, 7382, 7993, 9462, 9463, 9464		
\prop_put:Nno	6427, 6791, 6792, 6793, 6795, 6796, 6797, 6798, 6799, 6800		
\prop_put:NnV	6427		
\prop_put:Nnx	6427, 15243, 15245, 15248, 15250, 15256		
\prop_put:Non	6427		
\prop_put:Noo	6427		

<code>\q_no_value</code>	44, 2084, 2238, 2240, 2299, 2321, 5480, 5488, 5500, 5524, 5898, 5913, 6362, 6373, 6382, 9233
<code>\q_recursion_stop</code> 4, 45, 802, 805, 869, 933, 1724, 2242, 2243, 4433, 5847, 6142, 6178, 8401, 15707, 15709, 15718, 15720, 15738, 15740, 15743, 15761, 15769, 15776, 15792, 15794, 15803, 15805, 15819, 15830, 15845, 15847, 15849, 15852, 15864, 15875, 15909, 15913, 15940, 15942, 15958, 15961
<code>\q_recursion_tail</code> 4, 45, 869, 874, 933, 952, 2242, 2242, 2246, 2252, 2261, 2268, 2278, 2285, 4706, 4723, 4732, 5053, 5847, 6091, 6105, 6124, 6142, 6178, 6480, 6491, 6524, 6529, 8401, 15490, 15495, 15707, 15776, 15792, 15819
<code>\q_stop</code>	44, 801, 804, 885, 889, 905, 910, 915, 963, 967, 972, 977, 982, 1010, 1012, 1013, 1014, 1744, 1750, 1779, 1806, 1810, 1814, 1822, 1828, 1837, 1855, 1889, 1894, 2032, 2040, 2084, 2087, 2211, 2213, 2225, 2227, 2231, 2238, 2241, 2576, 2578, 2620, 2629, 2633, 2645, 2651, 2667, 2675, 2687, 2693, 2705, 2711, 2723, 2729, 2736, 2741, 2747, 2757, 2761, 2777, 2780, 2783, 2805, 2999, 3006, 3015, 3024, 3200, 3216, 3218, 3222, 3235, 3290, 3787, 3824, 3962, 3988, 4011, 4177, 4179, 4538, 4553, 4661, 4667, 4689, 4700, 4776, 4778, 4783, 4785, 4807, 4829, 4908, 4910, 4924, 4942, 4961, 4983, 5090, 5098, 5100, 5171, 5203, 5500, 5503, 5511, 5513, 5593, 5594, 5702, 5712, 5715, 5717, 5900, 5903, 5915, 5918, 5926, 5929, 5937, 5953, 6011, 6038, 6041, 6042, 6052, 6060, 6203, 6211, 6212, 6213, 6239, 6272, 6336, 6339, 7952, 8411, 8415, 8419, 8430, 8434, 8440, 8442, 8444, 8526, 8532, 8538, 8547, 8557, 8621, 8623, 8628, 8672, 8673, 9602, 9694, 10398, 10400, 10489, 10492, 15504, 15505, 15507, 15511, 15515, 15517, 15522, 16014
<code>sqrt</code>	190
<code>\quark_if_nil:N</code>	2289
<code>\quark_if_nil:n</code>	2309
<code>\quark_if_nil:nF</code>	2330
<code>\quark_if_nil:nT</code>	2329
<code>\quark_if_nil:NTF</code>	44, 2289, 3722, 3744
<code>\quark_if_nil:nTF</code>	44, 2309, 2328
<code>\quark_if_nil:oTF</code>	2309, 8438
<code>\quark_if_nil:VTF</code>	2309
<code>\quark_if_nil_p:N</code>	44, 2289
<code>\quark_if_nil_p:n</code>	44, 2309, 2327
<code>\quark_if_nil_p:o</code>	2309
<code>\quark_if_nil_p:V</code>	2309
<code>\quark_if_no_value:cTF</code>	2289
<code>\quark_if_no_value:N</code>	2297
<code>\quark_if_no_value:n</code>	2318
<code>\quark_if_no_value:NF</code>	2307
<code>\quark_if_no_value:NT</code>	2306
<code>\quark_if_no_value:NTF</code> 44, 2089, 2289, 2308, 7423, 7427, 7506, 7510, 9260, 9359, 9371
<code>\quark_if_no_value:nTF</code>	44, 2309
<code>\quark_if_no_value_p:c</code>	2289
<code>\quark_if_no_value_p:N</code>	44, 2289, 2305
<code>\quark_if_no_value_p:n</code>	44, 2309
<code>\quark_if_recursion_tail_break:N</code>	2350, 2350
<code>\quark_if_recursion_tail_break:n</code>	2350, 2352
<code>\quark_if_recursion_tail_stop:N</code>	45, 2244, 2244, 6154
<code>\quark_if_recursion_tail_stop:n</code>	8, 9, 45, 2258, 2258, 2274, 5851, 6184
<code>\quark_if_recursion_tail_stop:o</code>	2258, 8410
<code>\quark_if_recursion_tail_stop_do:Nn</code>	45, 2244, 2250, 15722, 15807
<code>\quark_if_recursion_tail_stop_do:nn</code>	45, 2258, 2265, 2275, 15781, 15834
<code>\quark_if_recursion_tail_stop_do:on</code>	2258
<code>\quark_new:N</code>	44, 2237, 2237, 2238, 2239, 2240, 2241, 2242, 2243, 2331, 2332
R	
<code>\R</code>	1730, 16000
<code>\radical</code>	328
<code>\raise</code>	468
<code>\read</code>	275
<code>\readline</code>	550
<code>\relax</code>	4, 5, 6, 7, 10, 14, 20, 24, 74, 75, 82, 98, 139, 140, 141, 142, 143, 144,

- 145, 146, 147, 148, 149, 152, 153,
 154, 155, 156, 157, 158, 159, 160, 310
 \relpenalty 371
 \RequirePackage 127
 \reverse_if:N 23, 688, 693,
 3248, 3250, 3252, 3254, 3976, 3981,
 3985, 3987, 5097, 13547, 14200, 14223
 \right 369
 \righthyphenmin 425
 \rightskip 427
 \romannumeral 502
 true 191
 \rule 7405, 7460
 trunc 188
- S**
- \S 10388
 \s__fp ... 9744, 9744, 9748, 9757, 9758,
 9759, 9760, 9761, 9763, 9764, 9767,
 9773, 9777, 9797, 9800, 9801, 9811,
 9821, 9833, 9853, 9925, 9927, 9929,
 9930, 9931, 9933, 9934, 9935, 9937,
 10120, 10125, 10309, 10318, 10320,
 10915, 11025, 11595, 11620, 11621,
 11731, 11732, 11735, 11746, 11747,
 11755, 11756, 11758, 11759, 11760,
 11762, 11763, 11764, 11775, 11778,
 11790, 11815, 11863, 11866, 11869,
 11888, 11889, 11891, 11892, 11893,
 11901, 11904, 11920, 11921, 11923,
 11932, 12008, 12159, 12193, 12194,
 12197, 12276, 12412, 12420, 12422,
 12599, 12602, 13060, 13072, 13074,
 13282, 13299, 13301, 13490, 13509,
 13511, 13512, 13514, 13526, 13531,
 13533, 13558, 13559, 13561, 13577,
 13662, 13675, 13677, 13680, 13685,
 13732, 13745, 13747, 13760, 13762,
 13775, 13777, 13790, 13792, 13805,
 13807, 13820, 13830, 14262, 14277,
 14278, 14282, 14293, 14399, 14412,
 14414, 14430, 14433, 14443, 14466,
 14477, 14479, 14493, 14495, 14500,
 14533, 14558, 14561, 14583, 14607,
 14610, 14651, 14674, 14724, 14725
 \s__fp_division 9752, 9755
 \s__fp_exact 9752, 9756,
 9757, 9758, 9759, 9760, 9761, 11731
 \s__fp_invalid 9752, 9752
 \s__fp_mark 9750, 9750, 9991,
 9992, 9994, 9998, 11130, 11165,
 11555, 11556, 11559, 11562, 11563
 \s__fp_overflow 9752, 9754, 9764
 \s__fp_stop 9750, 9751, 10997,
 11131, 11134, 11557, 11562, 11563,
 11565, 11794, 11805, 11828, 11836
 \s__fp_underflow 9752, 9753, 9763
 \s__prop
 132, 6296, 6296, 6297, 6300, 6336,
 6339, 6391, 6394, 6434, 6457, 6479,
 6483, 6524, 6527, 6542, 15490, 15493
 \s__seq 116, 2349, 2349, 5243, 5251, 5281,
 5286, 5291, 5296, 5329, 5353, 5361,
 5365, 5544, 5594, 5709, 15505, 15511
 \s__stop 47, 2347,
 2347, 2348, 13360, 13375, 14515, 14519
 \savecatcodetable 629
 \savinghyphcodes 589
 \savingvdiscards 590
 csc 188
 \scan_align_safe_stop: .. 41, 2199, 2199
 \scan_stop:
 . 10, 178, 192, 712, 712, 891, 999,
 1023, 1041, 1051, 1069, 1394, 1395,
 1499, 1724, 1727, 1728, 1729, 1730,
 1766, 1808, 1875, 1876, 2202, 2217,
 2344, 2512, 2589, 2894, 3008, 3017,
 3026, 3423, 4141, 4152, 4157, 4183,
 4188, 4191, 4232, 4243, 4248, 4253,
 4479, 5098, 5542, 6677, 6695, 7401,
 7456, 9397, 9399, 9490, 9492, 10424,
 10425, 10547, 10565, 10865, 10912,
 10913, 11163, 11194, 11435, 15551,
 15552, 15667, 15670, 15691, 16032,
 16034, 16036, 16067, 16072, 16073,
 16074, 16075, 16076, 16077, 16078
 \scantokens 548
 cscd 189
 \scriptfont 494
 \scriptscriptfont 495
 \scriptscriptstyle 340
 \scriptspace 380
 \scriptstyle 339
 \scrollmode 305
 asec 189
 ascd 189
 \seq_clear:c 5258
 \seq_clear:N 107, 5258,
 5258, 5260, 5265, 5386, 7948, 8011

<code>\seq_clear_new:c</code>	5264	<code>\seq_gpush:co</code>	5722 , 5740
<code>\seq_clear_new:N</code> ..	107 , 5264 , 5264 , 5266	<code>\seq_gpush:cV</code>	5722 , 5738
<code>\seq_concat:ccc</code>	5341	<code>\seq_gpush:cv</code>	5722 , 5739
<code>\seq_concat:NNN</code> 108 , 5341 , 5341 , 5345 , 9239		<code>\seq_gpush:cx</code>	5722 , 5741
<code>\seq_count:c</code>	5678	<code>\seq_gpush:Nn</code>	115 , 5722 , 5732
<code>\seq_count:N</code>		<code>\seq_gpush:No</code>	5722 , 5735 , 9295
... 113 , 5601 , 5678 , 5678 , 5687 , 5692		<code>\seq_gpush:Nv</code>	5722 , 5733 , 9408 , 9501
<code>\seq_gclear:c</code>	5258	<code>\seq_gpush:Nx</code>	5722 , 5736
<code>\seq_gclear:N</code> ..	107 , 5258 , 5261 , 5263 , 5268	<code>\seq_gput_left:cn</code>	5349 , 5737
<code>\seq_gclear_new:c</code>	5264	<code>\seq_gput_left:co</code>	5349 , 5740
<code>\seq_gclear_new:N</code> ..	107 , 5264 , 5267 , 5269	<code>\seq_gput_left:cV</code>	5349 , 5738
<code>\seq_gconcat:ccc</code>	5341	<code>\seq_gput_left:cv</code>	5349 , 5739
<code>\seq_gconcat:NNN</code> ..	108 , 5341 , 5343 , 5346	<code>\seq_gput_left:cx</code>	5349 , 5741
<code>\seq_get:cN</code>	5742 , 5743 , 5749	<code>\seq_gput_left:Nn</code>	
<code>\seq_get:cNTF</code>	5748	... 108 , 5349 , 5357 , 5368 , 5369 , 5732	
<code>\seq_get:NN</code>	115 , 5742 , 5742 , 5748	<code>\seq_gput_left:No</code>	5349 , 5735
<code>\seq_get:NNTF</code>	115 , 5748	<code>\seq_gput_left:Nv</code>	5349 , 5733
<code>\seq_get_left:cN</code>	5495 , 5743 , 5749	<code>\seq_gput_left:Nv</code>	5349 , 5734
<code>\seq_get_left:cNTF</code>	5562	<code>\seq_gput_left:Nx</code>	5349 , 5736
<code>\seq_get_left:NN</code>	109 , 5495 , 5495 , 5505 , 5562 , 5563 , 5742 , 5748	<code>\seq_gput_right:cn</code>	5370
<code>\seq_get_left:NNF</code>	5567	<code>\seq_gput_right:co</code>	5370
<code>\seq_get_left:NNT</code>	5566	<code>\seq_gput_right:cV</code>	5370
<code>\seq_get_left:NNTF</code>	110 , 5562 , 5568	<code>\seq_gput_right:cv</code>	5370
<code>\seq_get_right:cN</code>	5520	<code>\seq_gput_right:cx</code>	5370
<code>\seq_get_right:cNTF</code>	5562	<code>\seq_gput_right:Nn</code>	
<code>\seq_get_right:NN</code> 108 , 5370 , 5372 , 5374 , 5375 , 9288 , 9293	
... 109 , 5520 , 5520 , 5534 , 5564 , 5565		<code>\seq_gput_right:No</code>	5370 , 9336
<code>\seq_get_right:NNF</code>	5570	<code>\seq_gput_right:Nv</code>	5370 , 9168
<code>\seq_get_right:NNT</code>	5569	<code>\seq_gput_right:Nv</code>	5370
<code>\seq_get_right:NNTF</code>	110 , 5562 , 5571	<code>\seq_gput_right:Nx</code>	5370
<code>\seq_gpop:cN</code>	5742 , 5747 , 5753	<code>\seq_gremove_all:cn</code>	5396
<code>\seq_gpop:cNTF</code>	5748	<code>\seq_gremove_all:Nn</code> 111 , 5396 , 5398 , 5421	
<code>\seq_gpop:NN</code> ..	115 , 5742 , 5746 , 5752 , 9298	<code>\seq_gremove_duplicates:c</code>	5380
<code>\seq_gpop:NNTF</code>	115 , 5748 , 9381 , 9474	<code>\seq_gremove_duplicates:N</code>	
<code>\seq_gpop_left:cN</code>	5506 , 5747 , 5753	... 111 , 5380 , 5382 , 5395	
<code>\seq_gpop_left:cNTF</code>	5572	<code>\seq_greverse:c</code>	5422
<code>\seq_gpop_left:NN</code>		<code>\seq_greverse:N</code> ...	111 , 5422 , 5424 , 5439
... 109 , 5506 , 5508 , 5519 , 5574 , 5746 , 5752		<code>\seq_gset_eq:cc</code>	5270 , 5277
<code>\seq_gpop_left:NNF</code>	5584	<code>\seq_gset_eq:cN</code>	5270 , 5276
<code>\seq_gpop_left:NNT</code>	5583	<code>\seq_gset_eq:Nc</code>	5270 , 5275
<code>\seq_gpop_left:NNTF</code>	110 , 5572 , 5585	<code>\seq_gset_eq:NN</code>	
<code>\seq_gpop_right:cN</code>	5535	... 107 , 5262 , 5270 , 5274 , 5383 , 9457	
<code>\seq_gpop_right:cNTF</code>	5572	<code>\seq_gset_filter:NNn</code> ..	200 , 15526 , 15528
<code>\seq_gpop_right:NN</code>		<code>\seq_gset_from_clist:cc</code>	5278
... 109 , 5535 , 5537 , 5561 , 5578		<code>\seq_gset_from_clist:cN</code>	5278
<code>\seq_gpop_right:NNF</code>	5590	<code>\seq_gset_from_clist:cn</code>	5278
<code>\seq_gpop_right:NNT</code>	5589	<code>\seq_gset_from_clist:Nc</code>	5278
<code>\seq_gpop_right:NNTF</code> ...	111 , 5572 , 5591	<code>\seq_gset_from_clist:NN</code>	
<code>\seq_gpush:cn</code>	5722 , 5737	... 107 , 5278 , 5288 , 5301 , 5302	

<code>\seq_gset_from_clist:Nn</code>	5278 , 5293 , 5303	<code>\seq_map_variable:Ncn</code>	5666
<code>\seq_gset_map:NNn</code>	200 , 15536 , 15538	<code>\seq_map_variable:NNn</code>	112 , 5666 , 5666 , 5676 , 5677
<code>\seq_gset_split:Nnn</code>	108 , 5304 , 5306 , 5340 , 9343	<code>\seq_mapthread_function:ccN</code>	15503
<code>\seq_gset_split:NnV</code>	5304	<code>\seq_mapthread_function:cNN</code>	15503
<code>\seq_if_empty:cTF</code>	5440	<code>\seq_mapthread_function:NcN</code>	15503
<code>\seq_if_empty:N</code>	5440	<code>\seq_mapthread_function:NNN</code>	200 , 15503 , 15503 , 15524 , 15525
<code>\seq_if_empty:NF</code>	5450	<code>\seq_new:c</code>	5252
<code>\seq_if_empty:NT</code>	5449	<code>\seq_new:N</code>	4 , 107 , 2471 , 2488 , 5252 , 5252 , 5257 , 5265 , 5268 , 5379 , 5760 , 5761 , 5762 , 5763 , 7920 , 7921 , 8486 , 9162 , 9163 , 9173 , 9175 , 9178 , 9341 , 9455
<code>\seq_if_empty:NTF</code>	112 , 5440 , 5451 , 5797 , 8267	<code>\seq_pop:cN</code>	5742 , 5745 , 5751
<code>\seq_if_empty_p:c</code>	5440	<code>\seq_pop:cNTF</code>	5748
<code>\seq_if_empty_p:N</code>	112 , 5440 , 5448	<code>\seq_pop:NN</code>	115 , 5742 , 5744 , 5750
<code>\seq_if_exist:c</code>	5348	<code>\seq_pop:NNTF</code>	115 , 5748
<code>\seq_if_exist:cTF</code>	5347	<code>\seq_pop_left:cN</code>	5506 , 5745 , 5751
<code>\seq_if_exist:N</code>	5347	<code>\seq_pop_left:cNTF</code>	5572
<code>\seq_if_exist:NTF</code>	108 , 5265 , 5268 , 5347 , 5690	<code>\seq_pop_left:NN</code>	109 , 5506 , 5506 , 5518 , 5572 , 5744 , 5750
<code>\seq_if_exist_p:c</code>	5347	<code>\seq_pop_left:NNF</code>	5581
<code>\seq_if_exist_p:N</code>	108 , 5347	<code>\seq_pop_left:NNT</code>	5580
<code>\seq_if_in:cnTF</code>	5452	<code>\seq_pop_left:NNTF</code>	110 , 5572 , 5582
<code>\seq_if_in:coTF</code>	5452	<code>\seq_pop_right:cN</code>	5535
<code>\seq_if_in:cVTF</code>	5452	<code>\seq_pop_right:cNTF</code>	5572
<code>\seq_if_in:cxTF</code>	5452	<code>\seq_pop_right:NN</code>	109 , 5535 , 5535 , 5560 , 5576
<code>\seq_if_in:Nn</code>	5452	<code>\seq_pop_right:NNF</code>	5587
<code>\seq_if_in:NnF</code>	5389 , 5473 , 5474 , 9306	<code>\seq_pop_right:NNT</code>	5586
<code>\seq_if_in:NnT</code>	5471 , 5472	<code>\seq_pop_right:NNTF</code>	111 , 5572 , 5588
<code>\seq_if_in:NnTF</code>	112 , 5452 , 5475 , 5476	<code>\seq_push:cn</code>	5722 , 5727
<code>\seq_if_in:NoTF</code>	5452	<code>\seq_push:co</code>	5722 , 5730
<code>\seq_if_in:NVF</code>	9407 , 9500	<code>\seq_push:cV</code>	5722 , 5722 , 5728
<code>\seq_if_in:NVTF</code>	5452	<code>\seq_push:cv</code>	5729
<code>\seq_if_in:NvTF</code>	5452	<code>\seq_push:cx</code>	5722 , 5731
<code>\seq_if_in:NxTF</code>	5452	<code>\seq_push:Nn</code>	115 , 5722 , 5722
<code>\seq_item:cn</code>	5592	<code>\seq_push:No</code>	5722 , 5725
<code>\seq_item:Nn</code>	110 , 5592 , 5592 , 5616 , 8028 , 8029 , 8034	<code>\seq_push:Nv</code>	5722 , 5723
<code>\seq_map_break:</code>	113 , 5617 , 5617 , 5618 , 5620 , 5627 , 5628 , 5663 , 5674 , 8947 , 9249	<code>\seq_push:Nv</code>	5722 , 5724
<code>\seq_map_break:n</code>	113 , 5617 , 5619 , 7968 , 7982	<code>\seq_push:Nx</code>	5722 , 5726
<code>\seq_map_function:cN</code>	5621	<code>\seq_put_left:cn</code>	5349 , 5727
<code>\seq_map_function:NN</code>	4 , 112 , 5621 , 5621 , 5636 , 5683 , 5757 , 5803 , 8032	<code>\seq_put_left:co</code>	5349 , 5730
<code>\seq_map_inline:cn</code>	5659	<code>\seq_put_left:cV</code>	5349 , 5728
<code>\seq_map_inline:Nn</code>	112 , 5387 , 5659 , 5659 , 5665 , 7963 , 8940 , 9183 , 9243 , 9329	<code>\seq_put_left:cv</code>	5349 , 5729
<code>\seq_map_variable:ccn</code>	5666	<code>\seq_put_left:cx</code>	5349 , 5731
<code>\seq_map_variable:cNn</code>	5666	<code>\seq_put_left:Nn</code>	108 , 5349 , 5349 , 5366 , 5367 , 5722 , 7958
		<code>\seq_put_left:No</code>	5349 , 5725
		<code>\seq_put_left:Nv</code>	5349 , 5723

<code>\seq_put_left:Nv</code>	5349 , 5724	<code>\shipout</code>	441
<code>\seq_put_left:Nx</code>	5349 , 5726	<code>\ShortText</code>	62 , 91 , 110
<code>\seq_put_right:cn</code>	5370	<code>\show</code>	284
<code>\seq_put_right:co</code>	5370	<code>\showbox</code>	286
<code>\seq_put_right:cV</code>	5370	<code>\showboxbreadth</code>	300
<code>\seq_put_right:cv</code>	5370	<code>\showboxdepth</code>	301
<code>\seq_put_right:cx</code>	5370	<code>\showgroups</code>	561
<code>\seq_put_right:Nn</code>	108 , 5370 , 5370 , 5376 , 5377 , 5390 , 8019 , 9307	<code>\showifs</code>	562
<code>\seq_put_right:No</code>	5370 , 9322	<code>\showlists</code>	287
<code>\seq_put_right:Nv</code>	5370	<code>\showthe</code>	285
<code>\seq_put_right:Nx</code>	5370	<code>\showtokens</code>	549
<code>\seq_remove_all:cn</code>	5396	<code>asin</code>	189
<code>\seq_remove_all:Nn</code>		<code>asind</code>	189
.....	111 , 5396 , 5396 , 5420 , 9312	<code>\skewchar</code>	498
<code>\seq_remove_duplicates:c</code>	5380	<code>\skip</code>	522
<code>\seq_remove_duplicates:N</code>		<code>\skip_add:cn</code>	4151
.....	111 , 5380 , 5380 , 5394 , 9327	<code>\skip_add:Nn</code> ...	83 , 4151 , 4151 , 4153 , 4154
<code>\seq_reverse:c</code>	5422	<code>\skip_const:cn</code>	4122
<code>\seq_reverse:N</code>	111 , 5422 , 5422 , 5438	<code>\skip_const:Nn</code>	
<code>\seq_set_eq:cc</code>	5270 , 5273	83 , 4122 , 4122 , 4127 , 4198 , 4199
<code>\seq_set_eq:cN</code>	5270 , 5272	<code>\skip_eval:n</code>	84 , 4164 , 4182 , 4182
<code>\seq_set_eq:Nc</code>	5270 , 5271	<code>\skip_gadd:cn</code>	4151
<code>\seq_set_eq:NN</code>	107 , 5259 , 5270 , 5270 , 5381 , 9237 , 9254 , 9316	<code>\skip_gadd:Nn</code>	83 , 4151 , 4153 , 4155
<code>\seq_set_filter:NNn</code> ...	200 , 15526 , 15526	<code>.skip_gset:c</code>	160
<code>\seq_set_from_clist:cc</code>	5278	<code>\skip_gset:cn</code>	4140
<code>\seq_set_from_clist:cN</code>	5278	<code>.skip_gset:N</code>	160
<code>\seq_set_from_clist:cn</code>	5278	<code>\skip_gset:Nn</code> ..	83 , 4125 , 4140 , 4142 , 4144
<code>\seq_set_from_clist:Nc</code>	5278	<code>\skip_gset_eq:cc</code>	4145
<code>\seq_set_from_clist:NN</code>		<code>\skip_gset_eq:cN</code>	4145
.....	107 , 5278 , 5278 , 5298 , 5299	<code>\skip_gset_eq:Nc</code>	4145
<code>\seq_set_from_clist:Nn</code>		<code>\skip_gset_eq:NN</code>	84 , 4145 , 4148 , 4149 , 4150
.....	5278 , 5283 , 5300 , 8861 , 8870	<code>\skip_gsub:cn</code>	4151
<code>\seq_set_map:NNn</code>	200 , 15536 , 15536	<code>\skip_gsub:Nn</code>	84 , 4151 , 4158 , 4160
<code>\seq_set_split:Nnn</code>		<code>\skip_gzero:c</code>	4128
...	108 , 2484 , 2489 , 5304 , 5304 , 5339	<code>\skip_gzero:N</code> ..	83 , 4128 , 4129 , 4131 , 4135
<code>\seq_set_split:NnV</code>	5304 , 9238	<code>\skip_gzero_new:c</code>	4132
<code>\seq_show:c</code>	5754	<code>\skip_gzero_new:N</code> ..	83 , 4132 , 4134 , 4137
<code>\seq_show:N</code>	116 , 5754 , 5754 , 5759	<code>\skip_horizontal:c</code>	4186
<code>\seq_use:cn</code>	5688	<code>\skip_horizontal:N</code>	
<code>\seq_use:cnnn</code>	5688	86 , 4186 , 4186 , 4188 , 4192
<code>\seq_use:Nn</code>	114 , 5688 , 5719 , 5721	<code>\skip_horizontal:n</code>	4186 , 4187 , 16192
<code>\seq_use:Nnnn</code> .	114 , 5688 , 5688 , 5707 , 5720	<code>\skip_if_eq:nn</code>	4161
<code>\set@color</code>	7622 , 7623	<code>\skip_if_eq:nnTF</code>	84 , 4161
<code>\setbox</code>	476	<code>\skip_if_eq:p:nn</code>	84 , 4161
<code>\setlanguage</code>	234	<code>\skip_if_exist:c</code>	4139
<code>\sfcode</code>	531	<code>\skip_if_exist:cTF</code>	4138
<code>\sffamily</code>	7394	<code>\skip_if_exist:N</code>	4138
		<code>\skip_if_exist:NTF</code> ..	83 , 4133 , 4135 , 4138
		<code>\skip_if_exist_p:c</code>	4138
		<code>\skip_if_exist_p:N</code>	83 , 4138

<code>\skip_if_finite:n</code>	4173	<code>\str_case:on</code>	5150
<code>\skip_if_finite:nTF</code>	84, 4171, 15549	<code>\str_case:onF</code>	5238
<code>\skip_if_finite_p:n</code>	84, 4171	<code>\str_case:onn</code>	5237, 5238
<code>\skip_new:c</code>	4114	<code>\str_case:onTF</code>	5150
<code>\skip_new:N</code> 83, 4114, 4115, 4121, 4124,		<code>\str_case_x:nn</code>	5150, 5182
4133, 4135, 4200, 4201, 4202, 4203		<code>\str_case_x:nnF</code>	5192, 5228, 5239
<code>.skip_set:c</code>	160	<code>\str_case_x:nnn</code>	5237, 5239
<code>\skip_set:cn</code>	4140	<code>\str_case_x:nnT</code>	5187
<code>.skip_set:N</code>	160	<code>\str_case_x:nnTF</code>	105, 5150, 5197
<code>\skip_set:Nn</code> ... 83, 4140, 4140, 4142, 4143		<code>\str_fold_case:n</code>	106, 5211, 5211
<code>\skip_set_eq:cc</code>	4145	<code>\str_head:n</code> ... 104, 4944, 4991, 5086, 5086	
<code>\skip_set_eq:cN</code>	4145	<code>\str_if_eq:nn</code>	5131
<code>\skip_set_eq:Nc</code>	4145	<code>\str_if_eq:nnF</code>	5141, 5142, 8252
<code>\skip_set_eq:NN</code> 84, 4145, 4145, 4146, 4147		<code>\str_if_eq:nnT</code>	
<code>\skip_show:c</code>	4194	... 5139, 5140, 5404, 7979, 8944, 9218	
<code>\skip_show:N</code>	85, 4194, 4194, 4195	<code>\str_if_eq:nnTF</code>	104, 3792,
<code>\skip_show:n</code>	85, 4196, 4196	3795, 5131, 5143, 5144, 5178, 7804	
<code>\skip_split_finite_else_action:nnNN</code>		<code>\str_if_eq:noTF</code>	5131
..... 201, 15547, 15547		<code>\str_if_eq:nVTF</code>	5131
<code>\skip_sub:cn</code>	4151	<code>\str_if_eq:onTF</code>	5131
<code>\skip_sub:Nn</code> ... 84, 4151, 4156, 4158, 4159		<code>\str_if_eq:VnTF</code>	5131
<code>\skip_use:c</code>	4184	<code>\str_if_eq:VVTF</code>	5131
<code>\skip_use:N</code> 85, 4176, 4183, 4184, 4184, 4185		<code>\str_if_eq_p:nn</code> ... 104, 5131, 5137, 5138	
<code>\skip_vertical:c</code>	4186	<code>\str_if_eq_p:no</code>	5131
<code>\skip_vertical:N</code> 86, 4186, 4189, 4191, 4193		<code>\str_if_eq_p:nV</code>	5131
<code>\skip_vertical:n</code>	4186, 4190	<code>\str_if_eq_p:on</code>	5131
<code>\skip_zero:c</code>	4128	<code>\str_if_eq_p:Vn</code>	5131
<code>\skip_zero:N</code> 83, 4128, 4128, 4129, 4130, 4133		<code>\str_if_eq_p:VV</code>	5131
<code>\skip_zero_new:c</code>	4132	<code>\str_if_eq_x:nn</code>	5145
<code>\skip_zero_new:N</code> ... 83, 4132, 4132, 4136		<code>\str_if_eq_x:nnF</code>	8022, 8515
<code>\skipdef</code>	221	<code>\str_if_eq_x:nnTF</code>	
<code>\spac_directions_normal_body_dir</code> ..	677	... 105, 5131, 5206, 6396, 6485, 9658	
<code>\spac_directions_normal_page_dir</code> ..	678	<code>\str_if_eq_x_p:nn</code>	105, 5131
<code>\spacefactor</code>	440	<code>\str_tail:n</code>	104, 5086, 5094
<code>\spaceskip</code>	435	<code>\strcmp</code>	21
<code>\span</code>	248	<code>\string</code>	65, 503
<code>\special</code>	510		
<code>\splitbotmark</code>	319		
<code>\splitbotmarks</code>	545		
<code>\splitdiscards</code>	592		
<code>\splitfirstmark</code>	318		
<code>\splitfirstmarks</code>	544		
<code>\splitmaxdepth</code>	488		
<code>\splittopskip</code>	489		
<code>\str_case:nn</code>	5150, 5150, 5172		
<code>\str_case:nnF</code>	5160,		
5174, 5237, 15645, 15752, 15821, 15898			
<code>\str_case:nnn</code>	5237, 5237		
<code>\str_case:nnT</code>	5155, 5173		
<code>\str_case:nnTF</code>	105, 5150, 5165, 5175		

T

<code>\T</code>	2568, 2767, 10475, 15998
<code>pt</code>	191
<code>\tabskip</code>	249
<code>atan</code>	190
<code>atand</code>	190
<code>\tempa</code>	30, 43, 54
<code>T_EX</code> and <code>L^AT_EX 2_ε</code> commands:	
<code>\...mark</code>	290, 295
<code>\@firstoftwo</code>	227
<code>\@secondoftwo</code>	227
<code>\botmark</code>	295
<code>\box</code>	134

<code>\chardef</code>	291, 306	<code>\noexpand</code>	32
<code>\copy</code>	134	<code>\nullfont</code>	295
<code>\count</code>	292	<code>\number</code>	74, 591
<code>\countdef</code>	292	<code>\numexpr</code>	74
<code>\cr</code>	277	<code>\omit</code>	277
<code>\csname</code>	18	<code>\or</code>	73
<code>\detokenize</code>	353	<code>\outer</code>	723
<code>\dimen</code>	292	<code>\par</code>	444
<code>\dimendef</code>	292	<code>\pdfcolorstack</code>	731
<code>\dimexpr</code>	89	<code>\pdfliteral</code>	725
<code>\dp</code>	134	<code>\pdfsave</code>	726
<code>\edef</code>	2, 341	<code>\pdfstrcmp</code>	207, 208, 221
<code>\endcsname</code>	18	<code>\pretolerance</code>	277
<code>\endinput</code>	149	<code>\protect</code>	543
<code>\endlinechar</code>	295, 346	<code>\read</code>	170
<code>\endtemplate</code>	41, 277	<code>\readline</code>	171
<code>\errhelp</code>	448	<code>\relax</code> 231, 236, 247, 507, 510, 529, 558	
<code>\errmessage</code>	448, 449	<code>\RequirePackage</code>	7
<code>\escapechar</code>	98, 234, 502	<code>\romannumeral</code>	73,
<code>\etex_unexpanded:D</code>	361	251, 252, 358, 359, 463, 541, 564, 713	
<code>\exp_not:n</code>	99, 120	<code>\scantokens</code>	346
<code>\expandafter</code>	31	<code>\show</code>	17, 102, 247
<code>\firstmark</code>	259, 295	<code>\showthe</code>	247, 325, 334, 337, 340
<code>\futurelet</code>	277, 297, 299	<code>\showtokens</code>	103
<code>\global</code>	212	<code>\space</code>	295
<code>\halign</code>	41, 277	<code>\splitbotmark</code>	295
<code>\hbox</code>	137	<code>\splitfirstmark</code>	295
<code>\hskip</code>	86	<code>\strcmp</code>	207, 221
<code>\ht</code>	135	<code>\string</code>	53
<code>\hyphen</code>	295	<code>\tex_showbox:D</code>	417
<code>\ifcase</code>	73	<code>\tex_tracingonline:D</code>	417
<code>\ifdim</code>	88	<code>\the</code>	64, 81, 85, 87, 252
<code>\ifeof</code>	174	<code>\topmark</code>	295
<code>\ifhbox</code>	140	<code>\unexpanded</code>	32, 102, 110, 114,
<code>\ifmmode</code>	277	123, 125, 129, 201, 202, 309, 341, 360	
<code>\ifnum</code>	73	<code>\unhbox</code>	138
<code>\ifodd</code>	73, 723	<code>\unhcopy</code>	138
<code>\ifvbox</code>	140	<code>\unless</code>	23
<code>\ifvoid</code>	140	<code>\unvbox</code>	140
<code>\ifx</code>	24	<code>\unvcopy</code>	140
<code>\italiccorr</code>	295	<code>\uppercase</code>	94
<code>\jobname</code>	103	<code>\valign</code>	277
<code>\let</code>	212	<code>\vbox</code>	138
<code>\lower</code>	698	<code>\vskip</code>	86
<code>\lowercase</code>	93	<code>\vsplit</code>	139
<code>\makeatletter</code>	7	<code>\vtop</code>	138, 424
<code>\mathchardef</code>	291, 306	<code>\wd</code>	135
<code>\meaning</code>	17, 53, 292, 299, 723	<code>\write</code>	172, 500
<code>\newlinechar</code>	346	<code>\tex_above:D</code>	331
<code>\noalign</code>	277	<code>\tex_abovedisplaysortskip:D</code>	344

<code>\tex_abovedisplayskip:D</code>	345	<code>\tex_dimendef:D</code>	220, 2639
<code>\tex_abovewithdelims:D</code>	332	<code>\tex_discretionary:D</code>	384
<code>\tex_accent:D</code>	382	<code>\tex_displayindent:D</code>	349
<code>\tex_adjdemerits:D</code>	419	<code>\tex_displaylimits:D</code>	359
<code>\tex_advance:D</code>		<code>\tex_displaystyle:D</code>	337
..... 226, 3163, 3165, 3175, 3177,		<code>\tex_displaywidowpenalty:D</code>	348
3905, 3910, 4152, 4157, 4243, 4248		<code>\tex_displaywidth:D</code>	350
<code>\tex_afterassignment:D</code>	236, 2838	<code>\tex_divide:D</code>	227
<code>\tex_aftergroup:D</code>	237, 717	<code>\tex_doublehyphendemerits:D</code>	417
<code>\tex_atop:D</code>	333	<code>\tex_dp:D</code>	528, 6602
<code>\tex_atopwithdelims:D</code>	334	<code>\tex_dump:D</code>	511
<code>\tex_badness:D</code>	481	<code>\tex_edef:D</code>	215, 752
<code>\tex_baselineskip:D</code>	409	<code>\tex_else:D</code>	268, 691, 748
<code>\tex_batchmode:D</code>	302	<code>\tex_emergencystretch:D</code>	432
<code>\tex_begingroup:D</code>	240, 713	<code>\tex_end:D</code> 306, 641, 663, 1107, 7854	
<code>\tex_belowdisplayshortskip:D</code>	346	<code>\tex_endcsname:D</code>	308, 705
<code>\tex_belowdisplayskip:D</code>	347	<code>\tex_endgroup:D</code>	241, 638, 714
<code>\tex_binoppenalty:D</code>	370	<code>\tex_endinput:D</code>	280, 7865
<code>\tex_botmark:D</code>	317	<code>\tex_endlinechar:D</code>	
<code>\tex_box:D</code>	525, 6594, 6616	. 177, 178, 192, 322, 4477, 9446, 9448	
<code>\tex_boxmaxdepth:D</code>	487	<code>\tex_eqno:D</code>	342
<code>\tex_brokenpenalty:D</code>	444	<code>\tex_errhelp:D</code>	288, 7742
<code>\tex_catcode:D</code>	529,	<code>\tex_errmessage:D</code>	282, 1099, 7769
1000, 1395, 1727, 1728, 1729, 1730,		<code>\tex_errorcontextlines:D</code>	289, 7794
1875, 2203, 2218, 2358, 2360, 2362		<code>\tex_errorstopmode:D</code>	303
<code>\tex_char:D</code>	383	<code>\tex_escapechar:D</code> . 321, 9542, 9568, 9574	
<code>\tex_chardef:D</code>		<code>\tex_everycr:D</code>	250
..... 218, 741, 742, 743, 744, 745,		<code>\tex_everydisplay:D</code>	351, 642
747, 987, 988, 1933, 1935, 1973,		<code>\tex_everyhbox:D</code>	490
1978, 2763, 3141, 3142, 9397, 9490		<code>\tex_everyjob:D</code> . 519, 669, 4322, 4324,	
<code>\tex_cleaders:D</code>	401	4329, 4331, 9153, 9155, 9165, 9167	
<code>\tex_closein:D</code>	277, 9405	<code>\tex_everymath:D</code>	375, 643
<code>\tex_closeout:D</code>	272, 9498	<code>\tex_everypar:D</code>	438
<code>\tex_clubpenalty:D</code>	412	<code>\tex_everyvbox:D</code>	491
<code>\tex_copy:D</code>	469, 6588, 6617	<code>\tex_exhyphenpenalty:D</code>	414
<code>\tex_count:D</code>	520, 2658	<code>\tex_expandafter:D</code>	238, 706
<code>\tex_countdef:D</code>	219, 738, 2661	<code>\tex_fam:D</code>	230
<code>\tex_cr:D</code>	244	<code>\tex_fi:D</code>	269,
<code>\tex_crcr:D</code>	245	659, 671, 679, 684, 692, 750, 1980, 4464	
<code>\tex_csname:D</code>	307, 704	<code>\tex_finalhyphendemerits:D</code>	418
<code>\tex_day:D</code>	515	<code>\tex_firstmark:D</code>	316
<code>\tex_deadcycles:D</code>	449	<code>\tex_floatingpenalty:D</code>	463
<code>\tex_def:D</code> ... 214, 718, 720, 722, 723, 751		<code>\tex_font:D</code>	229
<code>\tex_defaulthyphenchar:D</code>	499	<code>\tex_fontdimen:D</code>	496
<code>\tex_defaultskewchar:D</code>	500	<code>\tex_fontname:D</code>	320
<code>\tex_delcode:D</code>	530	<code>\tex_futurelet:D</code>	225, 2832, 2834
<code>\tex_delimiter:D</code>	324	<code>\tex_gdef:D</code>	216, 765
<code>\tex_delimiterfactor:D</code>	373	<code>\tex_global:D</code>	198, 203,
<code>\tex_delimitershortfall:D</code>	372	205, 231, 1206, 1213, 1935, 1978,	
<code>\tex_dimen:D</code>	521, 2636	2834, 3125, 3145, 3157, 3167, 3169,	

- 3179, 3181, 3188, 3882, 3895, 3901,
3906, 3911, 4129, 4142, 4148, 4153,
4158, 4220, 4233, 4239, 4244, 4249,
6590, 6596, 6652, 6697, 6703, 6709,
6739, 6745, 6751, 6757, 9397, 9490
- `\tex_globaldefs:D` 235
- `\tex_halign:D` 242
- `\tex_hangafter:D` 420
- `\tex_hangindent:D` 421
- `\tex_hbadness:D` 482
- `\tex_hbox:D`
477, 6695, 6696, 6701, 6707, 6721, 6722
- `\tex_hfil:D` 385
- `\tex_hfill:D` 387
- `\tex_hfilneg:D` 386
- `\tex_hfuzz:D` 484
- `\tex_hoffset:D` 459, 673
- `\tex_holdinginserts:D` 462
- `\tex_hruler:D` 398
- `\tex_hsize:D`
423, 6878, 6880, 6881, 6924, 6926, 6927
- `\tex_hskip:D` 388, 4186
- `\tex_hss:D` . 389, 6724, 6726, 15094, 15103
- `\tex_ht:D` 527, 6601
- `\tex_hyphen:D` 212, 644
- `\tex_hyphenation:D` 513
- `\tex_hyphenchar:D` 497
- `\tex_hyphenpenalty:D` 415
- `\tex_if:D` 251, 694, 695
- `\tex_ifcase:D` 252, 3035
- `\tex_ifcat:D` 253, 696
- `\tex_ifdim:D` 256, 3864
- `\tex_ifeof:D` 257, 9423
- `\tex_iffalse:D` 262, 689
- `\tex_ifhbox:D` 258, 6628
- `\tex_ifhmode:D` 264, 699
- `\tex_ifinner:D` 267, 701
- `\tex_ifmmode:D` 265, 698
- `\tex_ifnum:D` 254, 715
- `\tex_ifodd:D` 255, 1124, 1139,
1907, 1908, 1939, 3034, 4406, 7647
- `\tex_iftrue:D` 263, 688
- `\tex_ifvbox:D` 259, 6629
- `\tex_ifvmode:D` 266, 700
- `\tex_ifvoid:D` 260, 6630
- `\tex_ifx:D` 261, 697
- `\tex_ignorespaces:D` 309
- `\tex_immediate:D`
... 271, 1094, 1096, 9492, 9498, 9516
- `\tex_indent:D` 405
- `\tex_input:D`
... 279, 645, 662, 9297, 15670, 15693
- `\tex_inputlineno:D` . 281, 1114, 1870, 7716
- `\tex_insert:D` 461
- `\tex_insertpenalties:D` 464
- `\tex_interlinepenalty:D` 443
- `\tex_italiccorrection:D` .. 211, 646, 674
- `\tex_jobname:D` 518, 4332, 4336, 9156
- `\tex_kern:D` 396, 7190, 7195, 7261, 7262,
7549, 7550, 14867, 15092, 15101,
15114, 15116, 15157, 15159, 15222
- `\tex_language:D` 313, 664
- `\tex_lastbox:D` 470, 6650
- `\tex_lastkern:D` 403
- `\tex_lastpenalty:D` 509
- `\tex_lastskip:D` 404
- `\tex_lccode:D` 532, 999, 1394,
1876, 2202, 2217, 2434, 2436, 2438
- `\tex_leaders:D` 400
- `\tex_left:D` 368, 681
- `\tex_lefthyphenmin:D` 424
- `\tex_leftskip:D` 426
- `\tex_leqno:D` 343
- `\tex_let:D` 199, 203, 205,
213, 641, 642, 643, 644, 645, 646,
647, 648, 649, 650, 651, 652, 653,
654, 655, 656, 657, 658, 661, 662,
663, 664, 665, 666, 667, 668, 669,
670, 673, 674, 675, 676, 677, 678,
681, 682, 683, 688, 689, 690, 691,
692, 693, 694, 695, 696, 697, 698,
699, 700, 701, 702, 703, 704, 705,
706, 707, 708, 709, 710, 711, 712,
713, 714, 715, 716, 717, 733, 735,
751, 752, 765, 766, 1202, 1907, 1908
- `\tex_limits:D` 360
- `\tex_linepenalty:D` 416
- `\tex_lineskip:D` 410
- `\tex_lineskiplimit:D` 411
- `\tex_long:D` 232, 718, 720, 723,
754, 756, 762, 764, 768, 770, 776, 778
- `\tex_looseness:D` 428
- `\tex_lower:D` 465, 6627
- `\tex_lowercase:D`
... 504, 1001, 1396, 1731, 1877, 4502
- `\tex_mag:D` 312
- `\tex_mark:D` 314
- `\tex_mathaccent:D` 325
- `\tex_mathbin:D` 355
- `\tex_mathchar:D` 326

<code>\tex_mathchardef:D</code> . . .	223, 749, 3137, 3138	<code>\tex_pagefilstretch:D</code>	452
<code>\tex_mathchoice:D</code>	323	<code>\tex_pagegoal:D</code>	456
<code>\tex_mathclose:D</code>	356	<code>\tex_pageshrink:D</code>	455
<code>\tex_mathcode:D</code>	534, 2428, 2430, 2432	<code>\tex_pagestretch:D</code>	451
<code>\tex_mathinner:D</code>	357	<code>\tex_pagetotal:D</code>	457
<code>\tex_mathop:D</code>	358, 667	<code>\tex_par:D</code>	406, 7605
<code>\tex_mathopen:D</code>	362	<code>\tex_parfillskip:D</code>	437
<code>\tex_mathord:D</code>	363	<code>\tex_parindent:D</code>	430
<code>\tex_mathpunct:D</code>	364	<code>\tex_parshape:D</code>	422
<code>\tex_mathrel:D</code>	365	<code>\tex_parskip:D</code>	429
<code>\tex_mathsurround:D</code>	376	<code>\tex_patterns:D</code>	512
<code>\tex_maxdeadcycles:D</code>	446	<code>\tex_pausing:D</code>	299
<code>\tex_maxdepth:D</code>	447	<code>\tex_penalty:D</code>	507
<code>\tex_meaning:D</code>	506, 709, 711	<code>\tex_postdisplaypenalty:D</code>	354
<code>\tex_medmuskip:D</code>	377	<code>\tex_predisdisplaypenalty:D</code>	353
<code>\tex_message:D</code>	283	<code>\tex_predisplaysize:D</code>	352
<code>\tex_middle:D</code>	682	<code>\tex_pretolerance:D</code>	433
<code>\tex_mkern:D</code>	330	<code>\tex_prevdepth:D</code>	480
<code>\tex_month:D</code>	516, 668	<code>\tex_prevgraf:D</code>	439
<code>\tex_moveleft:D</code>	466, 6621	<code>\tex_radical:D</code>	328
<code>\tex_moveright:D</code>	467, 6623	<code>\tex_raise:D</code>	468, 6625
<code>\tex_mskip:D</code>	327	<code>\tex_read:D</code>	275, 9441
<code>\tex_multiply:D</code>	228	<code>\tex_relax:D</code>	310, 712, 3033, 3866
<code>\tex_muskip:D</code>	524, 2678	<code>\tex_relpenny:D</code>	371
<code>\tex_muskipdef:D</code>	222, 2681	<code>\tex_right:D</code>	369, 683
<code>\tex_newlinechar:D</code>	278, 4478	<code>\tex_rightthyphenmin:D</code>	425
<code>\tex_noalign:D</code>	246	<code>\tex_rightskip:D</code>	427
<code>\tex_noboundary:D</code>	381	<code>\tex_romannumeral:D</code>	502, 716,
<code>\tex_noexpand:D</code>	239, 707		1475, 1487, 1493, 1533, 1537, 1542,
<code>\tex_noindent:D</code>	407		1548, 1554, 1560, 1572, 1577, 1579,
<code>\tex_nolimits:D</code>	361		1586, 1641, 1648, 1653, 1661, 1663,
<code>\tex_nonscript:D</code>	341		1666, 1673, 1679, 1688, 1704, 1708,
<code>\tex_nonstopmode:D</code>	304		1713, 2926, 3271, 3276, 3281, 3286,
<code>\tex_nulldelimiterspace:D</code>	374		3961, 3992, 3997, 4002, 4007, 4670,
<code>\tex_nullfont:D</code>	492, 2790		4675, 4680, 4685, 4883, 5031, 5152,
<code>\tex_number:D</code>	501, 3031		5157, 5162, 5167, 5184, 5189, 5194,
<code>\tex_omit:D</code>	247		5199, 8287, 9828, 9886, 9996, 10188,
<code>\tex_openin:D</code>	273, 9399		10280, 10281, 10285, 10360, 10378,
<code>\tex_openout:D</code>	274, 9492		10407, 10448, 10455, 10460, 10466,
<code>\tex_or:D</code>	270, 690		10471, 10508, 10521, 10522, 10528,
<code>\tex_outer:D</code>	233, 661		10538, 10556, 10557, 10577, 10590,
<code>\tex_output:D</code>	448		10594, 10616, 10644, 10657, 10670,
<code>\tex_outputpenalty:D</code>	458		10694, 10705, 10712, 10715, 10734,
<code>\tex_over:D</code>	335, 666		10744, 10747, 10760, 10763, 10775,
<code>\tex_overfullrule:D</code>	486		10798, 10827, 10841, 10857, 10877,
<code>\tex_overline:D</code>	366		10888, 10894, 10904, 10949, 10959,
<code>\tex_overwithdelims:D</code>	336		10974, 10985, 11000, 11012, 11046,
<code>\tex_pagedepth:D</code>	450		11055, 11064, 11125, 11127, 11141,
<code>\tex_pagefilllstretch:D</code>	454		11144, 11151, 11152, 11203, 11218,
<code>\tex_pagefillstretch:D</code>	453		11237, 11240, 11271, 11296, 11338,

11350, 11365, 11381, 11456, 11468, 11496, 11498, 11502, 11504, 11514, 11534, 11593, 11607, 11617, 11793, 11796, 11804, 11827, 11835, 12793, 13111, 13320, 13324, 13343, 13380, 13501, 13669, 13860, 14252, 14253, 14257, 14526, 14535, 14581, 14585, 14630, 14649, 14653, 14686, 14690, 14756, 15571, 15621, 15629, 15652	4077, 4184, 4197, 4254, 4259, 4324, 4331, 6677, 9155, 9167, 10485, 10928
<code>\tex_scriptfont:D</code>	494
<code>\tex_scriptscriptfont:D</code>	495
<code>\tex_scriptscriptstyle:D</code>	340
<code>\tex_scriptspace:D</code>	380
<code>\tex_scriptstyle:D</code>	339
<code>\tex_scrollmode:D</code>	305
<code>\tex_setbox:D</code>	476, 6588, 6594, 6650, 6696, 6701, 6707, 6738, 6743, 6749, 6755, 6777
<code>\tex_setlanguage:D</code>	234
<code>\tex_sfcode:D</code>	531, 2446, 2448, 2450
<code>\tex_shipout:D</code>	441
<code>\tex_show:D</code>	284
<code>\tex_showbox:D</code>	286, 6688
<code>\tex_showboxbreadth:D</code>	300, 6684
<code>\tex_showboxdepth:D</code>	301, 6685
<code>\tex_showlists:D</code>	287
<code>\tex_showthe:D</code>	285, 1385, 2362, 2432, 2438, 2444, 2450
<code>\tex_skewchar:D</code>	498
<code>\tex_skip:D</code>	522, 2696
<code>\tex_skipdef:D</code>	221, 2699
<code>\tex_space:D</code>	210
<code>\tex_spacefactor:D</code>	440
<code>\tex_spaceskip:D</code>	435
<code>\tex_span:D</code>	248
<code>\tex_special:D</code>	510, 16084, 16087, 16091, 16094, 16112, 16116, 16131, 16134, 16269, 16273
<code>\tex_splitbotmark:D</code>	319
<code>\tex_splitfirstmark:D</code>	318
<code>\tex_splitmaxdepth:D</code>	488
<code>\tex_splittopskip:D</code>	489
<code>\tex_string:D</code>	503, 710
<code>\tex_tabskip:D</code>	249
<code>\tex_textfont:D</code>	493
<code>\tex_textstyle:D</code>	338
<code>\tex_the:D</code>	178, 311, 1114, 1505, 1509, 1870, 2360, 2430, 2436, 2442, 2448, 3191,
<code>\tex_thickmuskip:D</code>	379
<code>\tex_thinmuskip:D</code>	378
<code>\tex_time:D</code>	514
<code>\tex_toks:D</code>	523, 2714
<code>\tex_toksdef:D</code>	224, 2717
<code>\tex_tolerance:D</code>	434
<code>\tex_topmark:D</code>	315
<code>\tex_topskip:D</code>	445
<code>\tex_tracingcommands:D</code>	290
<code>\tex_tracinglostchars:D</code>	291
<code>\tex_tracingmacros:D</code>	292
<code>\tex_tracingonline:D</code>	293, 6686
<code>\tex_tracingoutput:D</code>	294
<code>\tex_tracingpages:D</code>	295
<code>\tex_tracingparagraphs:D</code>	296
<code>\tex_tracingrestores:D</code>	297
<code>\tex_tracingstats:D</code>	298
<code>\tex_uccode:D</code>	533, 2440, 2442, 2444
<code>\tex_uchyph:D</code>	431
<code>\tex_undefined:D</code>	198, 205, 1219, 1227, 10012, 10023, 10024, 10025, 10026
<code>\tex_underline:D</code>	367, 647
<code>\tex_unhbox:D</code>	472, 6728
<code>\tex_unhcopy:D</code>	473, 6727
<code>\tex_unkern:D</code>	397
<code>\tex_unpenalty:D</code>	508
<code>\tex_unskip:D</code>	395
<code>\tex_unvbox:D</code>	474, 6773
<code>\tex_unvcopy:D</code>	475, 6772
<code>\tex_uppercase:D</code>	505, 4504
<code>\tex_vadjust:D</code>	408
<code>\tex_valign:D</code>	243
<code>\tex_vbadness:D</code>	483
<code>\tex_vbox:D</code>	478, 6731, 6734, 6736, 6738, 6749, 6755
<code>\tex_vcenter:D</code>	329, 665
<code>\tex_vfil:D</code>	390
<code>\tex_vfill:D</code>	392
<code>\tex_vfilneg:D</code>	391
<code>\tex_vfuzz:D</code>	485
<code>\tex_voffset:D</code>	460, 675
<code>\tex_vrule:D</code>	399, 7401, 7456
<code>\tex_vsize:D</code>	442
<code>\tex_vskip:D</code>	393, 4189
<code>\tex_vsplit:D</code>	471, 6777
<code>\tex_vss:D</code>	394
<code>\tex_vtop:D</code>	479, 6732, 6743
<code>\tex_wd:D</code>	526, 6603

<code>\tex_widowpenalty:D</code>	413	<code>\tl_count:o</code>	4758
<code>\tex_write:D</code>		<code>\tl_count:V</code>	4758
... 276, 1094, 1096, 9510, 9513, 9516		<code>\tl_count_tokens:n</code> 201, 15591, 15591, 15606	
<code>\tex_xdef:D</code>	217, 766	<code>\tl_expandable_lowercase:n</code>	
<code>\tex_xleaders:D</code>	402	... 202, 15617, 15625	
<code>\tex_xspaceskip:D</code>	436	<code>\tl_expandable_uppercase:n</code>	
<code>\tex_year:D</code>	517	... 202, 15617, 15617	
<code>\textdir</code>	635	<code>\tl_gclear:c</code>	4290, 5778
<code>\textfont</code>	493	<code>\tl_gclear:N</code> 91, 4290, 4292, 4295, 4299, 5777	
<code>\textstyle</code>	338	<code>\tl_gclear_new:c</code>	4296, 5782
<code>\TeXeTstate</code>	593	<code>\tl_gclear_new:N</code> 91, 4296, 4298, 4301, 5781	
<code>\the</code>	53, 139, 140, 141,	<code>\tl_gconcat:ccc</code>	4310
142, 143, 144, 145, 146, 147, 148, 311		<code>\tl_gconcat:NNN</code> 91, 4310, 4312, 4315, 4457	
<code>\thickmuskip</code>	379	<code>\tl_gput_left:cn</code>	4357
<code>\thinmuskip</code>	378	<code>\tl_gput_left:co</code>	4357
<code>\time</code>	514	<code>\tl_gput_left:cV</code>	4357
<code>\tiny</code>	7394	<code>\tl_gput_left:cx</code>	4357
<code>\tl_case:cn</code>	4668	<code>\tl_gput_left:Nn</code> 92, 4357, 4365, 4377, 4427	
<code>\tl_case:cnF</code>	5082	<code>\tl_gput_left:No</code> . 4357, 4369, 4379, 4428	
<code>\tl_case:cnn</code>	5081, 5082	<code>\tl_gput_left:NV</code> . 4357, 4367, 4378, 4427	
<code>\tl_case:cnTF</code>	4668	<code>\tl_gput_left:Nx</code> . 4357, 4371, 4380, 4428	
<code>\tl_case:Nn</code>	4668, 4668, 4696	<code>\tl_gput_right:cn</code>	4381
<code>\tl_case:NnF</code>	4678, 4698, 5081	<code>\tl_gput_right:co</code>	4381
<code>\tl_case:Nnn</code>	5081, 5081	<code>\tl_gput_right:cV</code>	4381
<code>\tl_case:NnT</code>	4673, 4697	<code>\tl_gput_right:cx</code>	4381
<code>\tl_case:NnTF</code>	95, 4668, 4683, 4699	<code>\tl_gput_right:Nn</code>	
<code>\tl_clear:c</code>	4290, 5776	92, 2343, 4381, 4389, 4401, 4431, 5373	
<code>\tl_clear:N</code>	91,	<code>\tl_gput_right:No</code> 4381, 4393, 4403, 4432	
4290, 4290, 4294, 4297, 5775, 8357,		<code>\tl_gput_right:NV</code> 4381, 4391, 4402, 4431	
8399, 8969, 9591, 9593, 9597, 9667		<code>\tl_gput_right:Nx</code> 4381, 4395, 4404, 4432	
<code>\tl_clear_new:c</code>	4296, 5780	<code>\tl_gremove_all:cn</code>	4561
<code>\tl_clear_new:N</code> 91, 4296, 4296, 4300, 5779		<code>\tl_gremove_all:Nn</code> . 93, 4561, 4563, 4566	
<code>\tl_concat:ccc</code>	4310	<code>\tl_gremove_once:cn</code>	4555
<code>\tl_concat:NNN</code> . 91, 4310, 4310, 4314, 4450		<code>\tl_gremove_once:Nn</code> . 92, 4555, 4557, 4560	
<code>\tl_const:cn</code>	4278	<code>\tl_greplace_all:cnn</code>	4505
<code>\tl_const:cx</code>	4278, 9549	<code>\tl_greplace_all:Nnn</code>	
<code>\tl_const:Nn</code>	91, 2237, 2469,	... 92, 4505, 4511, 4516, 4564	
4278, 4278, 4288, 4318, 4338, 5251,		<code>\tl_greplace_once:cnn</code>	4505
6300, 7631, 7632, 7684, 7689, 7691,		<code>\tl_greplace_once:Nnn</code>	
7693, 7695, 7697, 7702, 7703, 7710,		... 92, 4505, 4507, 4514, 4558	
8472, 8473, 8474, 9539, 9757, 9758,		<code>\tl_greverse:c</code>	4899
9759, 9760, 9761, 12612, 13051,		<code>\tl_greverse:N</code> . 99, 4899, 4901, 4904	
13052, 13053, 13054, 13055, 13056,		<code>.tl_gset:c</code>	160
13057, 13058, 13059, 15607, 15612		<code>\tl_gset:cf</code>	4339
<code>\tl_const:Nx</code>	4278, 4283, 4289,	<code>\tl_gset:cn</code>	4339
4332, 4336, 4466, 5773, 9543, 14772		<code>\tl_gset:co</code>	4339
<code>\tl_count:c</code>	4758	<code>\tl_gset:cV</code>	4339
<code>\tl_count:N</code> . 99, 4758, 4763, 4770, 9589		<code>\tl_gset:cv</code>	4339
<code>\tl_count:n</code> . 98, 849, 853, 1264, 1302,		<code>\tl_gset:cx</code>	4339
4758, 4758, 4769, 5048, 11521, 11542		<code>.tl_gset:N</code>	160

<code>\tl_gset:Nf</code>	4339 , 5344	<code>\tl_if_blank:VTF</code>	4567
<code>\tl_gset:Nn</code>	92 , 4339 , 4345 , 4354 , 4356 , 4424 , 4470 , 5509 , 5575 , 6351 , 6380 , 6416 , 9296 , 15657 , 15682	<code>\tl_if_blank_p:n</code> ...	94 , 4567 , 4569 , 4573
<code>\tl_gset:No</code>	4339 , 4347 , 4424	<code>\tl_if_blank_p:o</code>	4567
<code>\tl_gset:Nv</code>	4339	<code>\tl_if_blank_p:V</code>	4567
<code>\tl_gset:Nx</code>	4313 , 4339 , 4349 , 4355 , 4424 , 4462 , 4508 , 4512 , 4792 , 4902 , 5290 , 5295 , 5307 , 5359 , 5399 , 5425 , 5538 , 5579 , 5794 , 5822 , 5866 , 5909 , 5947 , 5999 , 6030 , 6428 , 6451 , 9156 , 14770 , 15529 , 15539	<code>\tl_if_empty:c</code>	6046
<code>\tl_gset_eq:cc</code> 4302 , 4309 , 5277 , 5790 , 6326		<code>\tl_if_empty:cTF</code>	4577
<code>\tl_gset_eq:cN</code> 4302 , 4307 , 5276 , 5789 , 6325		<code>\tl_if_empty:N</code>	4577 , 6045
<code>\tl_gset_eq:Nc</code> 4302 , 4308 , 5275 , 5788 , 6324		<code>\tl_if_empty:n</code>	4589
<code>\tl_gset_eq:NN</code>		<code>\tl_if_empty:NF</code>	4587
..... 91 , 4293 , 4302 , 4306 , 4442 , 5274 , 5787 , 6323 , 9160 , 9299 , 14777		<code>\tl_if_empty:nF</code>	878 , 956 , 2942 , 4600 , 6111 , 8123 , 8127 , 14739
<code>\tl_gset_from_file:cnn</code>	15654	<code>\tl_if_empty:NT</code>	4586
<code>\tl_gset_from_file:Nnn</code>	203 , 15654 , 15656 , 15659	<code>\tl_if_empty:nT</code>	4599
<code>\tl_gset_from_file_x:cnn</code>	15679	<code>\tl_if_empty:NTF</code>	94 , 4577 , 4588
<code>\tl_gset_from_file_x:Nnn</code>	204 , 15679 , 15681 , 15684	<code>\tl_if_empty:nTF</code>	94 , 3825 , 4519 , 4589 , 4598 , 5310 , 5852 , 7725 , 7989 , 8004 , 8356 , 8558 , 16029
<code>\tl_gset_rescan:cnn</code>	4467	<code>\tl_if_empty:o</code>	4610
<code>\tl_gset_rescan:cno</code>	4467	<code>\tl_if_empty:oTF</code>	2782 , 4601 , 4648 , 5036 , 5810 , 6056 , 6073
<code>\tl_gset_rescan:cnx</code>	4467	<code>\tl_if_empty:VTF</code>	4589
<code>\tl_gset_rescan:Nnn</code>	93 , 4467 , 4469 , 4499 , 4500	<code>\tl_if_empty_p:c</code>	4577
<code>\tl_gset_rescan:Nno</code>	4467	<code>\tl_if_empty_p:N</code>	94 , 4577 , 4585
<code>\tl_gset_rescan:Nnx</code>	4467	<code>\tl_if_empty_p:n</code>	94 , 4589 , 4597
<code>.tl_gset_x:c</code>	160	<code>\tl_if_empty_p:o</code>	4601
<code>.tl_gset_x:N</code>	160	<code>\tl_if_empty_p:V</code>	4589
<code>\tl_gtrim_spaces:c</code>	4787	<code>\tl_if_eq:ccTF</code>	4612
<code>\tl_gtrim_spaces:N</code> ..	100 , 4787 , 4791 , 4794	<code>\tl_if_eq:cNTF</code>	4612
<code>\tl_head:f</code>	4905	<code>\tl_if_eq:NcTF</code>	4612
<code>\tl_head:N</code>	100 , 4905 , 4925	<code>\tl_if_eq:NN</code>	4612
<code>\tl_head:n</code>	4905 , 4905 , 4923 , 4925	<code>\tl_if_eq:nn</code>	4624
<code>\tl_head:V</code>	4905	<code>\tl_if_eq:NNTF</code>	4623
<code>\tl_head:v</code>	4905	<code>\tl_if_eq:NNT</code>	4622 , 5411 , 7467 , 7470
<code>\tl_head:w</code>	100 , 4905 , 4924 , 4942 , 4961 , 4983 , 5093	<code>\tl_if_eq:NNTF</code>	94 , 4612 , 4621 , 4692 , 6469 , 7970 , 8024 , 9613
<code>\tl_if_blank:n</code>	4567	<code>\tl_if_eq:nnTF</code>	94 , 4624
<code>\tl_if_blank:nF</code> ..	3783 , 4571 , 4575 , 6185	<code>\tl_if_eq_p:cc</code>	4612
<code>\tl_if_blank:nT</code>	4570 , 4574	<code>\tl_if_eq_p:cN</code>	4612
<code>\tl_if_blank:nTF</code>	94 , 4567 , 4572 , 4576 , 4929 , 6260 , 8417 , 8625	<code>\tl_if_eq_p:Nc</code>	4612
<code>\tl_if_blank:oF</code>	8408	<code>\tl_if_eq_p:NN</code>	94 , 4612 , 4620
<code>\tl_if_blank:oTF</code>	4567 , 8429	<code>\tl_if_exist:c</code>	4317
		<code>\tl_if_exist:cTF</code>	4316
		<code>\tl_if_exist:N</code>	4316
		<code>\tl_if_exist:NTF</code>	91 , 4297 , 4299 , 4316 , 4754 , 5067
		<code>\tl_if_exist_p:c</code>	4316
		<code>\tl_if_exist_p:N</code>	91 , 4316
		<code>\tl_if_head_eq_catcode:nN</code>	4954
		<code>\tl_if_head_eq_catcode:nNTF</code> ..	101 , 4935
		<code>\tl_if_head_eq_catcode_p:nN</code> ..	101 , 4935

<code>\tl_if_head_eq_charcode:fNTF</code>	4935	<code>\tl_item:cn</code>	5041
<code>\tl_if_head_eq_charcode:nN</code>	4935	<code>\tl_item:Nn</code>	5041 , 5063 , 5064
<code>\tl_if_head_eq_charcode:nNF</code>	4953	<code>\tl_item:nn</code>	102 , 5041 , 5041 , 5063
<code>\tl_if_head_eq_charcode:nNT</code>	4952	<code>\tl_lower_case:n</code>	202 , 15698 , 15698
<code>\tl_if_head_eq_charcode:nNTF</code>		<code>\tl_lower_case:nn</code>	15698 , 15701
.	101 , 3688 , 3701 , 4935 , 4951	<code>\tl_map_break:</code>	97 , 4707 , 4713 , 4724 , 4733 , 4740 , 4745 , 4745 , 4746 , 4748
<code>\tl_if_head_eq_charcode_p:fN</code>	4935	<code>\tl_map_break:n</code>	97 , 4745 , 4747
<code>\tl_if_head_eq_charcode_p:nN</code>		<code>\tl_map_function:cN</code>	4703
.	101 , 4935 , 4950	<code>\tl_map_function:NN</code>	
<code>\tl_if_head_eq_meaning:nN</code>	4973	96 , 4703 , 4709 , 4716 , 4766 , 9192
<code>\tl_if_head_eq_meaning:nNTF</code>	101 , 4935	<code>\tl_map_function:nN</code>	
<code>\tl_if_head_eq_meaning_p:nN</code>	101 , 4935	96 , 4703 , 4703 , 4710 , 4761 , 5313
<code>\tl_if_head_is_group:n</code>	5014	<code>\tl_map_inline:cn</code>	4717
<code>\tl_if_head_is_group:nTF</code>	101 , 4845 , 4964 , 4999 , 5014 , 15714 , 15763 , 15799	<code>\tl_map_inline:Nn</code>	96 , 4717 , 4726 , 4728
<code>\tl_if_head_is_group_p:n</code>	101 , 5014	<code>\tl_map_inline:nn</code>	96 , 2610 , 4717 , 4717 , 4727 , 9546 , 11039 , 11087
<code>\tl_if_head_is_N_type:n</code>	5008	<code>\tl_map_variable:cNn</code>	4729
<code>\tl_if_head_is_N_type:nT</code>	15908 , 15957	<code>\tl_map_variable:NNn</code>	96 , 4729 , 4735 , 4744
<code>\tl_if_head_is_N_type:nTF</code>		<code>\tl_map_variable:nNn</code>	96 , 4729 , 4729 , 4736
.	102 , 4842 , 4939 , 4958 , 4975 , 5008 , 15563 , 15711 , 15760 , 15796 , 15863	<code>\tl_mixed_case:n</code>	202 , 15698 , 15700
<code>\tl_if_head_is_N_type_p:n</code>		<code>\tl_mixed_case:nn</code>	15698 , 15703
.	102 , 5008 , 15932	<code>\tl_new:c</code>	4272 , 5771 , 9102
<code>\tl_if_head_is_space:n</code>	5029	<code>\tl_new:N</code>	91 , 2334 , 2826 , 4272 , 4272 , 4277 , 4297 , 4299 , 4637 , 4638 , 5077 , 5078 , 5079 , 5080 , 5248 , 5249 , 5768 , 5770 , 6299 , 6783 , 6806 , 6807 , 7389 , 7630 , 7917 , 7918 , 8201 , 8381 , 8382 , 8383 , 8384 , 8476 , 8478 , 8479 , 8482 , 8483 , 8487 , 8488 , 9151 , 9171 , 9172 , 9346 , 9459 , 9530 , 9531 , 9532 , 9533 , 9534 , 15485 , 16253
<code>\tl_if_head_is_space:nTF</code>	102 , 5029	<code>\tl_put_left:cn</code>	4357
<code>\tl_if_head_is_space_p:n</code>	102 , 5029	<code>\tl_put_left:co</code>	4357
<code>\tl_if_in:cnTF</code>	4639	<code>\tl_put_left:cV</code>	4357
<code>\tl_if_in:nn</code>	4645	<code>\tl_put_left:cx</code>	4357
<code>\tl_if_in:NnF</code>	4640 , 4643	<code>\tl_put_left:Nn</code>	92 , 4357 , 4357 , 4373 , 4425
<code>\tl_if_in:nnF</code>	4640 , 4652	<code>\tl_put_left:No</code>	4357 , 4361 , 4375 , 4426
<code>\tl_if_in:NnT</code>	4639 , 4642 , 9200	<code>\tl_put_left:NV</code>	4357 , 4359 , 4374 , 4425
<code>\tl_if_in:nnT</code>	4639 , 4651	<code>\tl_put_left:Nx</code>	4357 , 4363 , 4376 , 4426
<code>\tl_if_in:NnTF</code>	95 , 2337 , 4639 , 4641 , 4644	<code>\tl_put_right:cn</code>	4381
<code>\tl_if_in:nnTF</code>	95 , 4641 , 4645 , 4653 , 7276 , 8525 , 8535 , 9280	<code>\tl_put_right:co</code>	4381
<code>\tl_if_in:noTF</code>	4645 , 16027	<code>\tl_put_right:cV</code>	4381
<code>\tl_if_in:onTF</code>	4645	<code>\tl_put_right:cx</code>	4381
<code>\tl_if_in:VnTF</code>	4645	<code>\tl_put_right:Nn</code>	
<code>\tl_if_single:n</code>	4658	92 , 4381 , 4381 , 4397 , 4429 , 5371
<code>\tl_if_single:Nf</code>	4656	<code>\tl_put_right:No</code>	4381 , 4385 , 4399 , 4430
<code>\tl_if_single:nf</code>	4656	<code>\tl_put_right:Nv</code>	4381 , 4383 , 4398 , 4429
<code>\tl_if_single:NT</code>	4655		
<code>\tl_if_single:nT</code>	4655		
<code>\tl_if_single:NfT</code>	95 , 4654 , 4657		
<code>\tl_if_single:nTF</code>	95 , 4657 , 4658		
<code>\tl_if_single_p:N</code>	95 , 4654 , 4654		
<code>\tl_if_single_p:n</code>	95 , 4654 , 4658		
<code>\tl_if_single_token:n</code>	15561		
<code>\tl_if_single_token:nTF</code>	201 , 15561		
<code>\tl_if_single_token_p:n</code>	201 , 15561		

\tl_put_right:Nx	4790, 4900, 5280, 5285, 5305, 5327, 4381, 4387, 4400, 4430, 8420, 8447, 9632, 9638, 9645, 9664, 9673, 9684
\tl_remove_all:cn	4561
\tl_remove_all:Nn 93, 4561, 4561, 4565, 9199	
\tl_remove_once:cn	4555
\tl_remove_once:Nn .. 92, 4555, 4555, 4559	
\tl_replace_all:cnn	4505
\tl_replace_all:Nnn .. 92, 4505, 4509, 4515, 4562, 5322, 8397, 8398, 9587	
\tl_replace_once:cnn	4505
\tl_replace_once:Nnn	92, 4505, 4505, 4513, 4556
\tl_rescan:nn	93, 4467, 4471
\tl_reverse:c	4899
\tl_reverse:N	99, 4899, 4899, 4903
\tl_reverse:n	99, 4879, 4879, 4892, 4900, 4902
\tl_reverse:o	4879
\tl_reverse:V	4879
\tl_reverse_items:n	99, 4771, 4771
\tl_reverse_tokens:n	201, 15567, 15567, 15584
.tl_set:c	160
\tl_set:cf	4339
\tl_set:cn	4339, 9105
\tl_set:co	4339
\tl_set:cV	4339
\tl_set:cv	4339
\tl_set:cx	4339
.tl_set:N	160
\tl_set:Nf	4339, 5342, 8358
\tl_set:Nn	92, 2844, 2865, 3415, 4339, 4339, 4351, 4353, 4423, 4468, 4627, 4628, 4739, 5312, 5316, 5407, 5416, 5456, 5459, 5480, 5488, 5507, 5516, 5526, 5573, 5670, 5898, 5904, 5913, 5920, 6153, 6345, 6361, 6362, 6370, 6371, 6373, 6379, 6382, 6405, 6406, 6415, 6431, 6454, 6510, 6790, 6794, 6981, 7277, 7278, 7391, 7394, 7928, 8010, 8199, 8396, 8496, 8529, 8540, 8644, 8824, 8836, 8854, 8971, 9132, 9228, 9233, 9612, 15655, 15680, 16255, 16258, 16261
\tl_set:No	4339, 4341, 4423, 15678
\tl_set:Nv	4339
\tl_set:Nv	4339
\tl_set:Nx	4311, 4339, 4343, 4352, 4423, 4455, 4506, 4510,
\tl_set_eq:cc 4302, 4305, 5273, 5786, 6322	
\tl_set_eq:cN 4302, 4303, 5272, 5785, 6321	
\tl_set_eq:Nc 4302, 4304, 5271, 5784, 6320	
\tl_set_eq:NN 91, 4291, 4302, 4302, 4436, 5270, 5783, 6319, 7973, 7981, 14776	
\tl_set_from_file:cnn	15654
\tl_set_from_file:Nnn	203, 15654, 15654, 15658
\tl_set_from_file_x:cnn	15679
\tl_set_from_file_x:Nnn	204, 15679, 15679, 15683
\tl_set_rescan:cnn	4467
\tl_set_rescan:cno	4467
\tl_set_rescan:cnx	4467
\tl_set_rescan:Nnn	93, 4467, 4467, 4497, 4498
\tl_set_rescan:Nno	4467
\tl_set_rescan:Nnx	4467
.tl_set_x:c	160
.tl_set_x:N	160
\tl_show:c	5065
\tl_show:N	102, 5065, 5065, 5074
\tl_show:n	103, 5075, 5075
\tl_tail:f	4905
\tl_tail:N	101, 4905, 4934
\tl_tail:n	4905, 4926, 4933, 4934
\tl_tail:V	4905
\tl_tail:v	4905
\tl_to_lowercase:n	93, 2204, 2219, 2570, 2612, 2769, 4501, 4501, 7754, 8281, 8390, 9539, 10392, 10478, 14530, 15984, 16001
\tl_to_str:c	4750
\tl_to_str:N	98, 4750, 4750, 4751, 9187, 9586, 9599, 9600
\tl_to_str:n 98, 731, 2999, 3966, 4087, 4181, 4522, 4591, 4604, 4661, 4749, 4749, 4915, 5076, 5089, 5098, 5213, 6332, 6390, 6391, 6433, 6456, 6478, 6479, 7419, 7502, 7817, 7818, 8061, 8062, 8365, 8369, 8370, 8374, 8375, 8494, 8534, 8822, 8887, 9017, 9035,	

9282, 9323, 9336, 9544, 9693, 11307, 14657, 14658, 14677, 14680, 16027	\token_if_group_end:N 2496
\tl_to_uppercase:n 94, 4501, 4503	\token_if_group_end:NTF 53, 2496
\tl_trim_spaces:c 4787	\token_if_group_end_p:N 53, 2496
\tl_trim_spaces:N 100, 4787, 4789, 4793	\token_if_int_register:N 2652
\tl_trim_spaces:n	\token_if_int_register:NTF 56, 2604
99, 4787, 4787, 4790, 4792, 5334, 8443	\token_if_int_register_p:N 56, 2604
\tl_upper_case:n 202, 15698, 15699	\token_if_letter:N 2534
\tl_upper_case:nn 15698, 15702	\token_if_letter:NTF 54, 2534
\tl_use:c 4752	\token_if_letter_p:N 54, 2534
\tl_use:N 98, 4752, 4752, 4757	\token_if_long_macro:N 2742
\token_get_arg_spec:N 60, 2997, 3010	\token_if_long_macro:NTF 55, 2604
\token_get_prefix_spec:N 60, 2997, 3001	\token_if_long_macro_p:N 55, 2604
\token_get_replacement_spec:N	\token_if_macro:N 2573
. 60, 2997, 3019	\token_if_macro:NTF
\token_if_active:N 2544 55, 2564, 2773, 3003, 3012, 3021
\token_if_active:NTF 54, 2544	\token_if_macro_p:N 55, 2564
\token_if_active_p:N 54, 2544	\token_if_math_subscript:N 2524
\token_if_alignment:N 2506	\token_if_math_subscript:NTF 54, 2524
\token_if_alignment:NTF 53, 2506	\token_if_math_subscript_p:N 54, 2524
\token_if_alignment_p:N 53, 2506	\token_if_math_superscript:N 2519
\token_if_chardef:N 2615	\token_if_math_superscript:NTF 54, 2519
\token_if_chardef:NTF 55, 2604, 2654	\token_if_math_superscript_p:N 54, 2519
\token_if_chardef_p:N 55, 2604	\token_if_math_toggle:N 2501
\token_if_cs:N 2587	\token_if_math_toggle:NTF 53, 2501
\token_if_cs:NTF	\token_if_math_toggle_p:N 53, 2501
. 55, 2587, 15723, 15771, 15808	\token_if_mathchardef:N 2624
\token_if_cs_p:N	\token_if_mathchardef:NTF 56, 2604, 2656
. 55, 2587, 15879, 15917, 15946, 15965	\token_if_mathchardef_p:N 56, 2604
\token_if_dim_register:N 2634	\token_if_muskip_register:N 2676
\token_if_dim_register:NTF 56, 2604	\token_if_muskip_register:NTF 56, 2604
\token_if_dim_register_p:N 56, 2604	\token_if_muskip_register_p:N 56, 2604
\token_if_eq_catcode:NN 2554	\token_if_other:N 2539
\token_if_eq_catcode:NNTF 54, 2554	\token_if_other:NTF 54, 2539
\token_if_eq_catcode_p:NN 54, 2554	\token_if_other_p:N 54, 2539
\token_if_eq_charcode:NN 2559	\token_if_parameter:N 2513
\token_if_eq_charcode:NNTF 54, 2559	\token_if_parameter:NTF 54, 2511
\token_if_eq_charcode_p:NN 54, 2559	\token_if_parameter_p:N 54, 2511
\token_if_eq_meaning:NN 2549	\token_if_primitive:N 2771
\token_if_eq_meaning:NNTF 10126	\token_if_primitive:NTF 56, 2763
\token_if_eq_meaning:NNT 2214	\token_if_primitive_p:N 56, 2763
\token_if_eq_meaning:NNTF	\token_if_protected_long_macro:N 2751
. 55, 2229, 2549, 10995, 13825	\token_if_protected_long_macro:NTF
\token_if_eq_meaning_p:NN 55, 2549 55, 2604
\token_if_expandable:N 2592	\token_if_protected_long_macro_p:N
\token_if_expandable:NTF 55, 2592 55, 2604
\token_if_expandable_p:N 55, 2592	\token_if_protected_macro:N 2730
\token_if_group_begin:N 2491	\token_if_protected_macro:NTF 55, 2604
\token_if_group_begin:NTF 53, 2491	\token_if_protected_macro_p:N 55, 2604
\token_if_group_begin_p:N 53, 2491	\token_if_skip_register:N 2694
	\token_if_skip_register:NTF 56, 2604

\token_if_skip_register_p:N	... 56, 2604	\tracingscantokens	552
\token_if_space:N	2529	\tracingstats	298
\token_if_space:NTF	54, 2529		
\token_if_space_p:N	54, 2529	U	
\token_if_toks_register:N	2712	\U	10477, 15997
\token_if_toks_register:NNTF	56, 2604	\uccode	533
\token_if_toks_register_p:N	56, 2604	\uchar	630
\token_new:Nn	52, 2451, 2451, 2456, 2458, 2459, 2460, 2462, 2463, 2464, 2465	\uchyph	431
\token_to_meaning:c	722, 733, 2451	\underline	367
\token_to_meaning:N	53, 709, 709, 1120, 1130, 1740, 2451, 2576, 2620, 2629, 2645, 2667, 2687, 2705, 2723, 2736, 2747, 2757, 2777, 3006, 3015, 3024, 16011	\unexpanded	546
\token_to_str:c	722, 722, 852, 861, 882, 902, 940, 945, 960, 1759, 2451	\unhbox	472
\token_to_str:N	5, 53, 709, 710, 722, 992, 993, 1120, 1130, 1132, 1145, 1155, 1276, 1306, 1388, 1403, 1820, 1835, 1868, 2000, 2225, 2340, 2451, 2622, 2631, 2647, 2669, 2689, 2707, 2725, 2738, 2749, 2759, 3229, 4466, 5020, 5071, 6691, 6832, 6980, 7594, 7598, 8252, 8259, 8266, 8349, 9184, 9569, 9570, 9571, 9572, 9573, 10004, 10005, 10398, 10406, 10407, 10433, 10605, 10625, 10640, 10652, 10653, 10666, 10667, 10692, 10701, 10703, 10728, 10731, 10756, 10758, 10772, 10788, 10806, 10875, 10885, 10886, 10901, 10902, 13893, 14806	\unhcopy	473
\toks	523	\unkern	397
\toksdef	224	\unless	537
\tolerance	434	\unpenalty	508
\topmark	315	\unskip	395
\topmarks	541	\unvbox	474
\topskip	445	\unvcopy	475
\tracingassigns	551	\uppercase	505
\tracingcommands	290	\use:c	18, 780, 780, 877, 955, 1086, 1088, 1090, 1092, 2046, 2065, 3227, 3647, 3657, 3800, 3809, 3811, 3813, 3814, 3818, 3970, 7850, 7861, 7876, 7885, 7893, 7901, 7907, 7933, 8330, 8548, 8555, 8693, 9657, 15730
\tracinggroups	558	\use:n	19, 786, 786, 908, 1223, 1385, 1413, 1415, 1419, 1427, 1429, 1437, 1441, 2472, 4472, 4741, 4992, 5011, 6155, 6688, 8293, 10049, 10057, 10066, 10083, 10091, 10119, 15498, 15834
\tracingifs	554	\use:nn	786, 787, 1478, 2481, 2997, 3964, 6139, 10483, 13501, 15675
\tracinglostchars	291	\use:nnn	786, 788, 1273
\tracingmacros	292	\use:nnnn	786, 789
\tracingnesting	553	\use:x	21, 781, 781, 864, 926, 1769, 4084, 4412, 4480, 4491, 6671, 7814, 7831, 8058, 8075, 9210, 9444, 9595
\tracingonline	293	\use_i:nn	20, 726, 790, 790, 816, 892, 1036, 1064, 1255, 1417, 1431, 1439, 1891, 2042, 2055, 2059, 2076, 2077, 4916, 5342, 5344, 6338, 9975, 11068, 11072, 11094, 12110, 12594, 12811, 13327, 13496, 14157, 14371, 15885, 15950, 15971
\tracingoutput	294	\use_i:nnn	
\tracingpages	295	\use_i:nnnn	20, 792, 792, 1018, 3006, 5545, 12068
\tracingparagraphs	296	\use_i:nnnn	20, 792, 796, 12086, 12093, 12283
\tracingrestores	297	\use_i_delimit_by_q_nil:nw	21, 803, 803
		\use_i_delimit_by_q_recursion_stop:nw	21, 803, 805, 2253, 2269, 15784, 15837

<code>\use_i_delimit_by_q_stop:nw</code>	243
..... 21, 803, 804, 6244	
<code>\use_i_ii:nnn</code> 20, 792, 795, 1503, 5522, 5623	
<code>\use_ii:nn</code>	483
..... 20, 728, 790, 791,	
818, 894, 1038, 1066, 1253, 1414,	
1420, 1428, 1442, 1447, 2055, 4918,	
6339, 9948, 11070, 11074, 11096,	
12112, 13329, 14159, 14373, 14743	
<code>\use_ii:nnn</code> . 20, 792, 793, 1020, 3015, 8439	
<code>\use_ii:nnnn</code>	6731
..... 20, 792, 797	
<code>\use_iii:nnn</code>	6737
..... 20, 792, 794, 1452,	
3024, 9960, 15727, 15787, 15788, 15814	
<code>\use_iii:nnnn</code>	6741
..... 20, 792, 798	
<code>\use_iv:nnnn</code>	6754
..... 20, 792, 799	
<code>\use_none:n</code> 20, 806, 806, 908, 1225, 1412,	
1416, 1418, 1426, 1430, 1438, 1440,	
2255, 2271, 2811, 3517, 3523, 3693,	
3697, 3702, 4568, 4830, 4915, 4931,	
4995, 5017, 5036, 5039, 5232, 5246,	
5611, 5632, 5802, 5925, 6018, 6044,	
7772, 8123, 8127, 8408, 8410, 8429,	
8626, 9907, 9911, 9915, 9919, 9986,	
10014, 11005, 11209, 11221, 11232,	
11247, 11251, 11276, 11301, 11370,	
11386, 11428, 12087, 12090, 12998,	
13893, 14751, 15519, 15520, 15564	
<code>\use_none:nn</code>	6771
..... 806, 807,	
854, 862, 1405, 4661, 4786, 4942,	
4961, 5412, 5531, 5552, 6056, 9860,	
9906, 9910, 9914, 9918, 15722, 15807	
<code>\use_none:nnn</code>	6776
..... 806,	
808, 4983, 8438, 9905, 9909, 9913, 9917	
<code>\use_none:nnnn</code> . 806, 809, 1821, 1836, 3385	
<code>\use_none:nnnnn</code>	6786
..... 806, 810, 10044, 10078, 10104, 10112	
<code>\use_none:nnnnnn</code>	6792
..... 806, 811, 962	
<code>\use_none:nnnnnnn</code> 806, 812, 884, 10046,	
10080, 10106, 10114, 10340, 12126	
<code>\use_none:nnnnnnnn</code>	6797
..... 806, 813	
<code>\use_none:nnnnnnnnn</code>	6803
..... 806, 814	
<code>\use_none_delimit_by_q_nil:w</code> 21, 800, 800	
<code>\use_none_delimit_by_q_recursion_stop:w</code>	
..... 21, 800, 802, 875, 941,	
946, 953, 1760, 1767, 2247, 2262, 4410	
<code>\use_none_delimit_by_q_stop:w</code> .. 21,	
800, 801, 3243, 3974, 4530, 6005,	
6231, 6236, 7951, 9686, 9700, 16013	
<code>\valign</code>	6808
<code>.value_forbidden:</code>	6814
<code>.value_required:</code>	6819
<code>\vbadness</code>	6825
<code>\vbox</code>	6831
<code>\vbox:n</code>	6837
<code>\vbox_gset:cn</code>	6843
<code>\vbox_gset:cw</code>	6849
<code>\vbox_gset:Nn</code>	6855
<code>\vbox_gset:Nw</code> . 139, 6754, 6756, 6759, 6769	
<code>\vbox_gset_end:</code> ... 139, 6754, 6765, 6771	
<code>\vbox_gset_inline_begin:c</code> ... 6766, 6770	
<code>\vbox_gset_inline_begin:N</code> ... 6766, 6769	
<code>\vbox_gset_inline_end:</code> ... 6766, 6771	
<code>\vbox_gset_to_ht:cnn</code>	6777
<code>\vbox_gset_to_ht:Nnn</code> 139, 6748, 6750, 6753	
<code>\vbox_gset_top:cn</code>	6759
<code>\vbox_gset_top:Nn</code> . 139, 6742, 6744, 6747	
<code>\vbox_set:cn</code>	6765
<code>\vbox_set:cw</code>	6771
<code>\vbox_set:Nn</code>	6777
... 139, 6737, 6737, 6739, 6740, 6876	
<code>\vbox_set:Nw</code>	6783
139, 6754, 6754, 6757, 6758, 6766, 6923	
<code>\vbox_set_end:</code>	6789
... 139, 6754, 6760, 6765, 6768, 6933	
<code>\vbox_set_inline_begin:c</code> 6766, 6767	
<code>\vbox_set_inline_begin:N</code> 6766, 6766	
<code>\vbox_set_inline_end:</code> 6766, 6768	
<code>\vbox_set_split_to_ht:Nnn</code> 139, 6776, 6776	
<code>\vbox_set_to_ht:cnn</code>	6785
<code>\vbox_set_to_ht:Nnn</code>	6791
..... 139, 6748, 6748, 6751, 6752	
<code>\vbox_set_top:cn</code>	6797
<code>\vbox_set_top:Nn</code>	6803
139, 6742, 6742, 6745, 6746, 6890, 6937	
<code>\vbox_to_ht:nn</code> 139, 6733, 6733, 6733	
<code>\vbox_to_zero:n</code> ... 139, 6733, 6733, 6735	
<code>\vbox_top:n</code>	6741
..... 138, 6731, 6732	
<code>\vbox_unpack:c</code>	6747
<code>\vbox_unpack:N</code>	6753
... 140, 6772, 6772, 6774, 6890, 6937	
<code>\vbox_unpack_clear:c</code>	6759
<code>\vbox_unpack_clear:N</code> 140, 6772, 6773, 6775	
<code>\vcenter</code>	6765
<code>\vcoffin_set:cnn</code>	6771
<code>\vcoffin_set:cnw</code>	6777
<code>\vcoffin_set:Nnn</code> .. 142, 6872, 6872, 6901	
<code>\vcoffin_set:Nnw</code> .. 142, 6919, 6919, 6952	
<code>\vcoffin_set_end:</code> . 142, 6919, 6930, 6951	
<code>\vadjust</code>	6957

\vfil	390		
\vfill	392		
\vfilneg	391		
\vfuzz	485		
\voffset	460		
\vrule	399		
\vsize	442		
\vskip	393		
\vsplit	471		
\vss	394		
\vtop	479		
		W	
\wd	526		
\widowpenalties	586		
\widowpenalty	413		
TWOBARS	186		
\write	276		
		X	
\X	1875		
ex	191		
\xdef	217		
\xetex_if_engine:F	1419, 1430		
\xetex_if_engine:T	1418, 1429		
\xetex_if_engine:TF	5, 23, 1412, 1420, 1431		
\xetex_if_engine_p:	23, 1412, 1423, 1433		
\xetex_XeTeXversion:D	621, 1424		
\XeTeXversion	621		
\xleaders	402		
exp	187		
\xspaceskip	436		
		Y	
\year	517		