

The L^AT_EX3 Sources

The L^AT_EX3 Project*

July 12, 2013

Abstract

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

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

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

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

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

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

Contents

I	Introduction to <code>expl3</code> and this document	1
1	Naming functions and variables	1
1.1	Terminological inexactitude	3
2	Documentation conventions	3
3	Formal language conventions which apply generally	5
4	<code>TeX</code> concepts not supported by <code>LaTeX3</code>	5
II	The <code>l3bootstrap</code> package: Bootstrap code	6
1	Using the <code>LaTeX3</code> modules	6
1.1	Internal functions and variables	7
III	The <code>l3names</code> package: Namespace for primitives	8
1	Setting up the <code>LaTeX3</code> programming language	8
IV	The <code>l3basics</code> package: Basic definitions	9
1	No operation functions	9
2	Grouping material	9
3	Control sequences and functions	10
3.1	Defining functions	10
3.2	Defining new functions using parameter text	11
3.3	Defining new functions using the signature	13
3.4	Copying control sequences	15
3.5	Deleting control sequences	16
3.6	Showing control sequences	16
3.7	Converting to and from control sequences	17
4	Using or removing tokens and arguments	18
4.1	Selecting tokens from delimited arguments	20

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

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

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

15	Constant skips	85
16	Scratch skips	85
17	Inserting skips into the output	86
18	Creating and initialising muskip variables	86
19	Setting muskip variables	87
20	Using muskip expressions and variables	87
21	Viewing muskip variables	88
22	Constant muskips	88
23	Scratch muskips	88
24	Primitive conditional	88
25	Internal functions	89
XI	The l3tl package: Token lists	90
1	Creating and initialising token list variables	91
2	Adding data to token list variables	92
3	Modifying token list variables	92
4	Reassigning token list category codes	93
5	Reassigning token list character codes	93
6	Token list conditionals	94
7	Mapping to token lists	95
8	Using token lists	97
9	Working with the content of token lists	98
10	The first token from a token list	99
11	Viewing token lists	102
12	Constant token lists	102

13	Scratch token lists	102
14	Internal functions	103
XII	The l3seq package: Sequences and stacks	104
1	Creating and initialising sequences	104
2	Appending data to sequences	105
3	Recovering items from sequences	105
4	Recovering values from sequences with branching	106
5	Modifying sequences	107
6	Sequence conditionals	108
7	Mapping to sequences	108
8	Using the content of sequences directly	110
9	Sequences as stacks	110
10	Constant and scratch sequences	112
11	Viewing sequences	112
12	Internal sequence functions	112
XIII	The l3clist package: Comma separated lists	113
1	Creating and initialising comma lists	113
2	Adding data to comma lists	114
3	Modifying comma lists	115
4	Comma list conditionals	115
5	Mapping to comma lists	116
6	Using the content of comma lists directly	119
7	Comma lists as stacks	119
8	Viewing comma lists	121

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

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

4	Setting keys	157
5	Handling of unknown keys	158
6	Utility functions for keys	159
7	Low-level interface for parsing key-val lists	159
XX	The l3file package: File and I/O operations	161
1	File operation functions	161
1.1	Input-output stream management	162
1.2	Reading from files	163
2	Writing to files	164
2.1	Wrapping lines in output	165
2.2	Constant input-output streams	166
2.3	Primitive conditionals	166
2.4	Internal file functions and variables	166
2.5	Internal input-output functions	167
XXI	The l3fp package: floating points	168
1	Creating and initialising floating point variables	169
2	Setting floating point variables	169
3	Using floating point numbers	170
4	Floating point conditionals	171
5	Floating point expression loops	172
6	Some useful constants, and scratch variables	173
7	Floating point exceptions	174
8	Viewing floating points	175
9	Floating point expressions	176
9.1	Input of floating point numbers	176
9.2	Precedence of operators	177
9.3	Operations	177
10	Disclaimer and roadmap	182

XXII	The <code>l3luatex</code> package: LuaTeX-specific functions	185
1	Breaking out to Lua	185
2	Category code tables	186
XXIII	The <code>l3candidates</code> package: Experimental additions to <code>l3kernel</code>	188
1	Additions to <code>l3basics</code>	188
2	Additions to <code>l3box</code>	188
	2.1 Affine transformations	188
	2.2 Viewing part of a box	189
	2.3 Internal variables	190
3	Additions to <code>l3clist</code>	191
4	Additions to <code>l3coffins</code>	191
5	Additions to <code>l3file</code>	192
6	Additions to <code>l3fp</code>	193
7	Additions to <code>l3prop</code>	193
8	Additions to <code>l3seq</code>	194
9	Additions to <code>l3skip</code>	195
10	Additions to <code>l3tl</code>	196
11	Additions to <code>l3tokens</code>	197
XXIV	Implementation	198
1	<code>l3bootstrap</code> implementation	198
	1.1 Format-specific code	198
	1.2 Package-specific code part one	199
	1.3 The <code>\pdfstrcmp</code> primitive in XeTeX	200
	1.4 Engine requirements	200
	1.5 Package-specific code part two	201
	1.6 Dealing with package-mode meta-data	202
	1.7 The L ^A T _E X3 code environment	204
	1.8 Deprecated functions	205

2	l3names implementation	206
3	l3basics implementation	216
3.1	Renaming some TeX primitives (again)	217
3.2	Defining some constants	219
3.3	Defining functions	219
3.4	Selecting tokens	220
3.5	Gobbling tokens from input	222
3.6	Conditional processing and definitions	222
3.7	Dissecting a control sequence	228
3.8	Exist or free	229
3.9	Defining and checking (new) functions	232
3.10	More new definitions	233
3.11	Copying definitions	235
3.12	Undefining functions	235
3.13	Generating parameter text from argument count	236
3.14	Defining functions from a given number of arguments	237
3.15	Using the signature to define functions	238
3.16	Checking control sequence equality	240
3.17	Diagnostic functions	240
3.18	Engine specific definitions	241
3.19	Doing nothing functions	242
3.20	String comparisons	242
3.21	Breaking out of mapping functions	243
3.22	Deprecated functions	244
4	l3expan implementation	246
4.1	General expansion	246
4.2	Hand-tuned definitions	249
4.3	Definitions with the automated technique	252
4.4	Last-unbraced versions	253
4.5	Preventing expansion	254
4.6	Defining function variants	255
4.7	Variants which cannot be created earlier	261
5	l3prg implementation	262
5.1	Primitive conditionals	262
5.2	Defining a set of conditional functions	262
5.3	The boolean data type	262
5.4	Boolean expressions	264
5.5	Logical loops	270
5.6	Producing n copies	271
5.7	Detecting TeX's mode	272
5.8	Internal programming functions	273
5.9	Deprecated functions	275

6	l3quark implementation	278
6.1	Quarks	279
6.2	Scan marks	282
6.3	Deprecated quark functions	282
7	l3token implementation	283
7.1	Character tokens	283
7.2	Generic tokens	285
7.3	Token conditionals	286
7.4	Peeking ahead at the next token	296
7.5	Decomposing a macro definition	301
7.6	Deprecated functions	302
8	l3int implementation	305
8.1	Integer expressions	305
8.2	Creating and initialising integers	307
8.3	Setting and incrementing integers	309
8.4	Using integers	310
8.5	Integer expression conditionals	310
8.6	Integer expression loops	314
8.7	Integer step functions	315
8.8	Formatting integers	316
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	328
9.1	Length primitives renamed	328
9.2	Creating and initialising <code>dim</code> variables	328
9.3	Setting <code>dim</code> variables	329
9.4	Utilities for dimension calculations	330
9.5	Dimension expression conditionals	331
9.6	Dimension expression loops	332
9.7	Using <code>dim</code> expressions and variables	334
9.8	Viewing <code>dim</code> variables	335
9.9	Constant dimensions	335
9.10	Scratch dimensions	335
9.11	Creating and initialising <code>skip</code> variables	335
9.12	Setting <code>skip</code> variables	336
9.13	Skip expression conditionals	337
9.14	Using <code>skip</code> expressions and variables	338
9.15	Inserting skips into the output	338
9.16	Viewing <code>skip</code> variables	338
9.17	Constant skips	338

9.18	Scratch skips	339
9.19	Creating and initialising muskip variables	339
9.20	Setting muskip variables	340
9.21	Using muskip expressions and variables	340
9.22	Viewing muskip variables	341
9.23	Constant muskips	341
9.24	Scratch muskips	341
9.25	Deprecated functions	341
10	l3tl implementation	342
10.1	Functions	342
10.2	Constant token lists	344
10.3	Adding to token list variables	345
10.4	Reassigning token list category codes	346
10.5	Reassigning token list character codes	348
10.6	Modifying token list variables	348
10.7	Token list conditionals	350
10.8	Mapping to token lists	353
10.9	Using token lists	355
10.10	Working with the contents of token lists	355
10.11	Token by token changes	357
10.12	The first token from a token list	360
10.13	Viewing token lists	365
10.14	Scratch token lists	365
10.15	Deprecated functions	366
11	l3seq implementation	368
11.1	Allocation and initialisation	369
11.2	Appending data to either end	371
11.3	Modifying sequences	371
11.4	Sequence conditionals	373
11.5	Recovering data from sequences	374
11.6	Mapping to sequences	377
11.7	Using sequences	379
11.8	Sequence stacks	380
11.9	Viewing sequences	381
11.10	Scratch sequences	381
11.11	Deprecated interfaces	381

12	l3clist implementation	382
12.1	Allocation and initialisation	383
12.2	Removing spaces around items	384
12.3	Adding data to comma lists	385
12.4	Comma lists as stacks	386
12.5	Modifying comma lists	388
12.6	Comma list conditionals	390
12.7	Mapping to comma lists	390
12.8	Using comma lists	393
12.9	Viewing comma lists	394
12.10	Scratch comma lists	395
12.11	Deprecated interfaces	395
13	l3prop implementation	397
13.1	Allocation and initialisation	397
13.2	Accessing data in property lists	398
13.3	Property list conditionals	402
13.4	Recovering values from property lists with branching	403
13.5	Mapping to property lists	404
13.6	Viewing property lists	405
13.7	Deprecated interfaces	405
14	l3box implementation	406
14.1	Creating and initialising boxes	406
14.2	Measuring and setting box dimensions	408
14.3	Using boxes	408
14.4	Box conditionals	408
14.5	The last box inserted	409
14.6	Constant boxes	409
14.7	Scratch boxes	409
14.8	Viewing box contents	410
14.9	Horizontal mode boxes	411
14.10	Vertical mode boxes	412
14.11	Deprecated functions	414
15	l3coffins Implementation	414
15.1	Coffins: data structures and general variables	414
15.2	Basic coffin functions	416
15.3	Measuring coffins	420
15.4	Coffins: handle and pole management	420
15.5	Coffins: calculation of pole intersections	423
15.6	Aligning and typesetting of coffins	426
15.7	Coffin diagnostics	431
15.8	Messages	436
16	l3color Implementation	437

17	l3msg implementation	438
17.1	Creating messages	438
17.2	Messages: support functions and text	440
17.3	Showing messages: low level mechanism	441
17.4	Displaying messages	443
17.5	Kernel-specific functions	450
17.6	Expandable errors	455
17.7	Showing variables	457
17.8	Deprecated functions	458
18	l3keys Implementation	461
18.1	Low-level interface	461
18.2	Constants and variables	464
18.3	The key defining mechanism	466
18.4	Turning properties into actions	467
18.5	Creating key properties	471
18.6	Setting keys	475
18.7	Utilities	478
18.8	Messages	478
18.9	Deprecated functions	479
19	l3file implementation	481
19.1	File operations	482
19.2	Input operations	486
19.2.1	Variables and constants	486
19.2.2	Stream management	487
19.2.3	Reading input	489
19.3	Output operations	490
19.3.1	Variables and constants	490
19.4	Stream management	491
19.4.1	Deferred writing	492
19.4.2	Immediate writing	492
19.4.3	Special characters for writing	493
19.4.4	Hard-wrapping lines to a character count	493
19.5	Messages	498
19.6	Deprecated functions	499
20	l3fp implementation	500
21	l3fp-aux implementation	500

22	Internal storage of floating points numbers	501
22.1	Using arguments and semicolons	501
22.2	Constants, and structure of floating points	502
22.3	Overflow, underflow, and exact zero	504
22.4	Expanding after a floating point number	504
22.5	Packing digits	506
22.6	Decimate (dividing by a power of 10)	508
22.7	Functions for use within primitive conditional branches	510
22.8	Small integer floating points	511
22.9	Length of a floating point array	512
22.10	x-like expansion expandably	512
22.11	Messages	513
23	l3fp-traps Implementation	513
23.1	Flags	514
23.2	Traps	514
23.3	Errors	518
23.4	Messages	518
24	l3fp-round implementation	519
24.1	Rounding tools	519
24.2	The round function	522
25	l3fp-parse implementation	524
26	Precedences	524
27	Evaluating an expression	525
28	Work plan	525
28.1	Storing results	525
28.2	Precedence	527
28.3	Infix operators	528
28.4	Prefix operators, parentheses, and functions	530
28.5	Type detection	533
29	Internal representation	533

30	Internal parsing functions	534
30.1	Expansion control	535
30.2	Fp object type	536
30.3	Reading digits	536
30.4	Parsing one operand	537
30.4.1	Trimming leading zeros	542
30.4.2	Exact zero	544
30.4.3	Small significand	544
30.4.4	Large significand	546
30.4.5	Finding the exponent	548
30.4.6	Beyond 16 digits: rounding	551
30.5	Main functions	554
30.6	Main functions	555
30.7	Prefix operators	557
30.7.1	Identifiers	557
30.7.2	Unary minus, plus, not	559
30.7.3	Other prefixes	559
30.8	Infix operators	560
31	Messages	566
32	l3fp-logic Implementation	566
32.1	Syntax of internal functions	566
32.2	Existence test	567
32.3	Comparison	567
32.4	Floating point expression loops	569
32.5	Extrema	570
32.6	Boolean operations	572
32.7	Ternary operator	573
33	l3fp-basics Implementation	574
33.1	Common to several operations	574
33.2	Addition and subtraction	575
33.2.1	Sign, exponent, and special numbers	576
33.2.2	Absolute addition	578
33.2.3	Absolute subtraction	580
33.3	Multiplication	585
33.3.1	Signs, and special numbers	585
33.3.2	Absolute multiplication	586
33.4	Division	589
33.4.1	Signs, and special numbers	589
33.4.2	Work plan	590
33.4.3	Implementing the significand division	593
33.5	Unary operations	598

34	l3fp-extended implementation	598
	34.1 Description of extended fixed points	598
	34.2 Helpers for extended fixed points	599
	34.3 Dividing a fixed point number by a small integer	600
	34.4 Adding and subtracting fixed points	601
	34.5 Multiplying fixed points	602
	34.6 Combining product and sum of fixed points	603
	34.7 Converting from fixed point to floating point	605
35	l3fp-expo implementation	610
	35.1 Logarithm	610
	35.1.1 Work plan	610
	35.1.2 Some constants	611
	35.1.3 Sign, exponent, and special numbers	611
	35.1.4 Absolute ln	611
	35.2 Exponential	619
	35.2.1 Sign, exponent, and special numbers	619
	35.3 Power	623
36	Implementation	630
	36.1 Direct trigonometric functions	630
	36.1.1 Sign and special numbers	631
	36.1.2 Small and tiny arguments	634
	36.1.3 Reduction of large arguments	635
	36.2 Computing the power series	637
37	l3fp-convert implementation	640
	37.1 Trimming trailing zeros	640
	37.2 Scientific notation	640
	37.3 Decimal representation	642
	37.4 Token list representation	643
	37.5 Formatting	644
	37.6 Convert to dimension or integer	644
	37.7 Convert from a dimension	645
	37.8 Use and eval	646
	37.9 Convert an array of floating points to a comma list	647
38	l3fp-assign implementation	647
	38.1 Assigning values	647
	38.2 Updating values	648
	38.3 Showing values	649
	38.4 Some useful constants and scratch variables	649
39	l3fp-old implementation	650
	39.1 Compatibility	650

40	l3luatex implementation	653
40.1	Category code tables	654
40.2	Messages	657
40.3	Deprecated functions	657
41	l3candidates Implementation	658
41.1	Additions to l3box	658
41.2	Affine transformations	658
41.3	Viewing part of a box	666
41.4	Additions to l3clist	668
41.5	Additions to l3coffins	671
41.6	Rotating coffins	671
41.7	Resizing coffins	675
41.8	Additions to l3file	678
41.9	Additions to l3fp	679
41.10	Additions to l3prop	679
41.11	Additions to l3seq	680
41.12	Additions to l3skip	684
41.13	Additions to l3tl	685
41.14	Additions to l3tokens	688
	Index	691

Part I

Introduction to expl3 and this document

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

1 Naming functions and variables

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

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

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

- D** The **D** specifier means *do not use*. All of the `TEX` primitives are initially `\let` to a **D** name, and some are then given a second name. Only the kernel team should use anything with a **D** specifier!
- N and n** These mean *no manipulation*, of a single token for **N** and of a set of tokens given in braces for **n**. Both pass the argument through exactly as given. Usually, if you use a single token for an **n** argument, all will be well.
- c** This means *csname*, and indicates that the argument will be turned into a *csname* before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`.
- V and v** These mean *value of variable*. The **V** and **v** specifiers are used to get the content of a variable without needing to worry about the underlying `TEX` structure containing the data. A **V** argument will be a single token (similar to **N**), for example `\foo:V \MyVariable`; on the other hand, using **v** a *csname* is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.
- o** This means *expansion once*. In general, the **V** and **v** specifiers are favoured over **o** for recovering stored information. However, **o** is useful for correctly processing information with delimited arguments.

- x** The **x** specifier stands for *exhaustive expansion*: every token in the argument is fully expanded until only unexpandable ones remain. The `\edef` primitive carries out this type of expansion. Functions which feature an **x**-type argument are in general *not* expandable, unless specifically noted.
- f** The **f** specifier stands for *full expansion*, and in contrast to **x** stops at the first non-expandable item (reading the argument from left to right) without trying to expand it. For example, when setting a token list variable (a macro used for storage), the sequence

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

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

- T and F** For logic tests, there are the branch specifiers **T** (*true*) and **F** (*false*). Both specifiers treat the input in the same way as **n** (no change), but make the logic much easier to see.
- p** The letter **p** indicates `\TeX` *parameters*. Normally this will be used for delimited functions as `expl3` provides better methods for creating simple sequential arguments.
- w** Finally, there is the **w** specifier for *weird* arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some arbitrary string).

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

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

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

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

bool Either true or false.

box Box register.

¹The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the `int` module contains some scratch variables called `\l_tmpa_int`, `\l_tmpb_int`, and so on. In such a case adding the module name up front to denote the module and in the back to indicate the type, as in `\l_int_tmpa_int` would be very unreadable.

clist Comma separated list.

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

dim “Rigid” lengths.

fp floating-point values;

int Integer-valued count register.

prop Property list.

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

skip “Rubber” lengths.

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

tl Token list variables: placeholder for a token list.

1.1 Terminological inexactitude

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

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

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

2 Documentation conventions

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

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

`\ExplSyntaxOn`
`\ExplSyntaxOff`

`\ExplSyntaxOn ... \ExplSyntaxOff`

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

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

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

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

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

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

`\cs_to_str:N` ★

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

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

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

`\seq_map_function:NN` ☆

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

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

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

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

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

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

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

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

The normal description text.

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

3 Formal language conventions which apply generally

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

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

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

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

Part II

The l3bootstrap package

Bootstrap code

1 Using the L^AT_EX3 modules

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

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

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

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

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

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

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

These functions act broadly in the same way as the L^AT_EX 2_ε kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as L^AT_EX 2_ε provides in turning on `\makeatletter` within package and class code.)

`\GetIdInfo`
 Updated: 2012-06-04

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

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

To summarize: Every single package using this syntax should identify itself using one of the above methods. Special care is taken so that every package or class file loaded with `\RequirePackage` or alike are loaded with usual L^AT_EX 2_ε category codes and the L^AT_EX3 category code scheme is reloaded when needed afterwards. See implementation for details. If you use the `\GetIdInfo` command you can use the information when loading a package with

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

1.1 Internal functions and variables

`_expl_package_check:`

`_expl_package_check:`

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

`\l_kernel_expl_bool`

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

Part III

The l3names package Namespace for primitives

1 Setting up the L^AT_EX3 programming language

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

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

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

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

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

`\pdftex_...` Introduced by pdfT_EX;

`\xetex_...` Introduced by X_YT_EX;

`\luatex_...` Introduced by LuaT_EX.

Part IV

The l3basics package

Basic definitions

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

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

1 No operation functions

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

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

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

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

2 Grouping material

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

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

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

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

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

3 Control sequences and functions

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

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

3.1 Defining functions

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

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

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

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

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

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

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

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

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

N and n No manipulation.

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

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

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

3.2 Defining new functions using parameter text

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current \TeX group level: the assignment is global.

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

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

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

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

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

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

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

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

3.3 Defining new functions using the signature

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Updated: 2012-01-14

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

3.4 Copying control sequences

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

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

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

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

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

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

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

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

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

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

3.5 Deleting control sequences

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

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

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

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

Updated: 2011-09-15

3.6 Showing control sequences

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

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

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

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

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

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

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

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

Updated: 2012-09-09

3.7 Converting to and from control sequences

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

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

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

`\use:c { a b c }`

and

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

would be equivalent to

`\abc`

after two expansions of `\use:c`.

`\cs_if_exist_use:N` `\cs_if_exist_use:N ⟨control sequence⟩`

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

New: 2012-11-10

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

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

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

New: 2012-11-10

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

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

`\cs_end:` ★

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

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

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

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

and

```

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

```

would be equivalent to

```
\abc
```

after one expansion of `\cs:w`.

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

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

4 Using or removing tokens and arguments

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

```

\use:n ★ \use:n {<group1>}
\use:(nn|nnn|nnnn) ★ \use:nn {<group1>} {<group2>}
\use:nnn {<group1>} {<group2>} {<group3>}
\use:nnnn {<group1>} {<group2>} {<group3>} {<group4>}

```

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

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

will result in the input stream containing

```
abc { def }
```

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

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

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

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

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

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

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

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

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

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

will result in the input stream containing

```
abc { def }
```

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

<code>\use_none:n</code>	★	<code>\use_none:n {⟨group₁⟩}</code>
<code>\use_none:(nn nnn nnnn nnnnn nnnnnn nnnnnnn nnnnnnnn nnnnnnnnn)</code>	★	

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

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

4.1 Selecting tokens from delimited arguments

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

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

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

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

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

5 Predicates and conditionals

L^AT_EX3 has three concepts for conditional flow processing:

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

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

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

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

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

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

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

`\cs_if_free_p:N`

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

```
\bool_if:nTF {
  \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl
} {\langle true code \rangle} {\langle false code \rangle}
```

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

Primitive conditionals There is a third variety of conditional, which is the original concept used in plain $\text{T}_{\text{E}}\text{X}$ 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>	$\langle control\ sequence \rangle$
<code>\cs_if_free_p:c</code>	★	<code>\cs_if_free:NTF</code>	$\langle control\ sequence \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\cs_if_free:NTF</code>	★	Tests whether the $\langle control\ sequence \rangle$ is currently free to be defined. This test will be	
<code>\cs_if_free:cTF</code>	★	false if the $\langle control\ sequence \rangle$ currently exists (as defined by <code>\cs_if_exist:N</code>).	

5.2 Testing string equality

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

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

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

is logically true.

<code>\str_if_eq_x_p:nn</code>	★	<code>\str_if_eq_x_p:nn</code>	$\{\langle t_1 \rangle\}$ $\{\langle t_2 \rangle\}$
<code>\str_if_eq_x:nnTF</code>	★	<code>\str_if_eq_x:nnTF</code>	$\{\langle t_1 \rangle\}$ $\{\langle t_2 \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

New: 2012-06-05

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

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

is logically true.

<code>\str_case:nnn</code>	★	<code>\str_case:nnn</code>	$\{\langle test\ string \rangle\}$
<code>\str_case:onn</code>	★	{	
		$\{\langle string\ case_1 \rangle\}$ $\{\langle code\ case_1 \rangle\}$	
		$\{\langle string\ case_2 \rangle\}$ $\{\langle code\ case_2 \rangle\}$	
		...	
		$\{\langle string\ case_n \rangle\}$ $\{\langle code\ case_n \rangle\}$	
		}	
		$\{\langle else\ code \rangle\}$	

New: 2012-06-03

This function compares the $\langle test\ string \rangle$ in turn with each of the $\langle string\ cases \rangle$. If the two are equal (as described for `\str_if_eq:nnTF` then the associated $\langle code \rangle$ is left in the input stream. If none of the tests are **true** then the **else code** will be left in the input stream.

<code>\str_case_x:nnn</code> ★	<code>\str_case_x:nnn {<test string>}</code>
	<code>{</code>
	<code>{<string case₁>} {<code case₁>}</code>
	<code>{<string case₂>} {<code case₂>}</code>
	<code>...</code>
	<code>{<string case_n>} {<code case_n>}</code>
	<code>}</code>
	<code>{<else code>}</code>

New: 2012-06-05

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 none of the tests are `true` then the `else code` will be left in the input stream. The `<test string>` is expanded in each comparison, and must always yield the same result: for example, random numbers should not be used within this string.

5.3 Engine-specific conditionals

<code>\luatex_if_engine_p:</code> ★	<code>\luatex_if_engine:TF {<true code>} {<false code>}</code>
<code>\luatex_if_engine:TF</code> ★	Detects is the document is being compiled using LuaTeX.

Updated: 2011-09-06

<code>\pdfTeX_if_engine_p:</code> ★	<code>\pdfTeX_if_engine:TF {<true code>} {<false code>}</code>
<code>\pdfTeX_if_engine:TF</code> ★	Detects is the document is being compiled using pdfTeX.

Updated: 2011-09-06

<code>\xetex_if_engine_p:</code> ★	<code>\xetex_if_engine:TF {<true code>} {<false code>}</code>
<code>\xetex_if_engine:TF</code> ★	Detects is the document is being compiled using XeTeX.

Updated: 2011-09-06

5.4 Primitive conditionals

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

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

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

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

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

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

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

<code>\if:w</code>	★	<code>\if:w <token₁₂</code>
<code>\if_charcode:w</code>	★	<code>\if_catcode:w <token₁₂</code>
<code>\if_catcode:w</code>	★	These conditionals will expand any following tokens until two unexpandable tokens are left. If you wish to prevent this expansion, prefix the token in question with <code>\exp_not:N</code> . <code>\if_catcode:w</code> tests if the category codes of the two tokens are the same whereas <code>\if:w</code> tests if the character codes are identical. <code>\if_charcode:w</code> is an alternative name for <code>\if:w</code> .

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

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

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

6 Internal kernel functions

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

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

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

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

Part V

The l3expan package

Argument expansion

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

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

1 Defining new variants

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

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

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

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

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

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

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

2 Methods for defining variants

`\cs_generate_variant:Nn`

Updated: 2013-07-09

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

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

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

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

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

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

3 Introducing the variants

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

- Arguments that might need expansion should come first in the list of arguments to make processing faster.
- Arguments that should consist of single tokens should come first.
- Arguments that need full expansion (*i.e.*, are denoted with `x`) should be avoided if possible as they can not be processed expandably, *i.e.*, functions of this type will not work correctly in arguments that are themselves subject to `x` expansion.
- In general, unless in the last position, multi-token arguments `n`, `f`, and `o` will need special processing which is not fast. Therefore it is best to use the optimized functions, namely those that contain only `N`, `c`, `V`, and `v`, and, in the last position, `o`, `f`, with possible trailing `N` or `n`, which are not expanded.

The `V` type returns the value of a register, which can be one of `tl`, `num`, `int`, `skip`, `dim`, `toks`, or built-in T_EX registers. The `v` type is the same except it first creates a

control sequence out of its argument before returning the value. This recent addition to the argument specifiers may shake things up a bit as most places where `o` is used will be replaced by `V`. The documentation you are currently reading will therefore require a fair bit of re-writing.

In general, the programmer should not need to be concerned with expansion control. When simply using the content of a variable, functions with a `V` specifier should be used. For those referred to by `(cs)name`, the `v` specifier is available for the same purpose. Only when specific expansion steps are needed, such as when using delimited arguments, should the lower-level functions with `o` specifiers be employed.

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

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

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

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

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

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

If you use this type of expansion in conditional processing then you should stick to using TF type functions only as it does not try to finish any `\if... \fi`: itself!

4 Manipulating the first argument

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

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

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

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

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

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

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

5 Manipulating two arguments

<hr/> <hr/>	<hr/> <hr/>
<code>\exp_args:NNo</code> ★	<code>\exp_args:NNo</code> $\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle tokens \rangle\}$
<code>\exp_args:(NNc NNv NNf Nco Ncf Ncc NVV)</code> ★	
	These optimized functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments.
<hr/> <hr/>	<hr/> <hr/>
<code>\exp_args:Nno</code> ★	<code>\exp_args:Nno</code> $\langle token \rangle$ $\{\langle tokens_1 \rangle\}$ $\{\langle tokens_2 \rangle\}$
<code>\exp_args:(NnV Nnf Noo Nof Noc Nff Nfo Nnc)</code> ★	
	Updated: 2012-01-14

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

<code>\exp_args:NNx</code>	<code>\exp_args:NNx <token₁> <token₂> {\tokens}</code>
<code>\exp_args:(Nnx Ncx Nox Nxo Nxx)</code>	

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

6 Manipulating three arguments

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

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

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

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

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

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

7 Unbraced expansion

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

Updated: 2012-02-12

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

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

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

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

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

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

<code>\exp_after:wN</code>	★	<code>\exp_after:wN</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$
----------------------------	---	----------------------------	---------------------------	---------------------------

Carries out a single expansion of $\langle token_2 \rangle$ (which may consume arguments) prior to the expansion of $\langle token_1 \rangle$. If $\langle token_2 \rangle$ is a T_EX primitive, it will be executed rather than expanded, while if $\langle token_2 \rangle$ has not expansion (for example, if it is a character) then it will be left unchanged. It is important to notice that $\langle token_1 \rangle$ may be *any* single token, including group-opening and -closing tokens (`{` or `}` assuming normal T_EX category codes). Unless specifically required, expansion should be carried out using an appropriate argument specifier variant or the appropriate `\exp_arg:N` function.

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

8 Preventing expansion

Despite the fact that the following functions are all about preventing expansion, they're designed to be used in an expandable context and hence are all marked as being 'expandable' since they themselves will not appear after the expansion has completed.

<hr/> <hr/>	<code>\exp_not:N</code> ★	<code>\exp_not:N</code> $\langle token \rangle$
		Prevents expansion of the $\langle token \rangle$ in a context where it would otherwise be expanded, for example an x -type argument.
		T_EXhackers note: This is the T _E X <code>\noexpand</code> primitive.
<hr/> <hr/>	<code>\exp_not:c</code> ★	<code>\exp_not:c</code> $\{\langle tokens \rangle\}$
		Expands the $\langle tokens \rangle$ until only unexpandable content remains, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited.
<hr/> <hr/>	<code>\exp_not:n</code> ★	<code>\exp_not:n</code> $\{\langle tokens \rangle\}$
		Prevents expansion of the $\langle tokens \rangle$ in a context where they would otherwise be expanded, for example an x -type argument.
		T_EXhackers note: This is the ε -T _E X <code>\unexpanded</code> primitive. Hence its argument <i>must</i> be surrounded by braces.
<hr/> <hr/>	<code>\exp_not:V</code> ★	<code>\exp_not:V</code> $\langle variable \rangle$
		Recovers the content of the $\langle variable \rangle$, then prevents expansion of this material in a context where it would otherwise be expanded, for example an x -type argument.
<hr/> <hr/>	<code>\exp_not:v</code> ★	<code>\exp_not:v</code> $\{\langle tokens \rangle\}$
		Expands the $\langle tokens \rangle$ until only unexpandable content remains, and then converts this into a control sequence (which should be a $\langle variable \rangle$ name). The content of the $\langle variable \rangle$ is recovered, and further expansion is prevented in a context where it would otherwise be expanded, for example an x -type argument.
<hr/> <hr/>	<code>\exp_not:o</code> ★	<code>\exp_not:o</code> $\{\langle tokens \rangle\}$
		Expands the $\langle tokens \rangle$ once, then prevents any further expansion in a context where they would otherwise be expanded, for example an x -type argument.
<hr/> <hr/>	<code>\exp_not:f</code> ★	<code>\exp_not:f</code> $\{\langle tokens \rangle\}$
		Expands $\langle tokens \rangle$ fully until the first unexpandable token is found. Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion.
<hr/> <hr/>	<code>\exp_stop_f:</code> ★	<code>\function:f</code> $\langle tokens \rangle$ <code>\exp_stop_f:</code> $\langle more tokens \rangle$
Updated: 2011-06-03		This function terminates an f -type expansion. Thus if a function <code>\function:f</code> starts an f -type expansion and all of $\langle tokens \rangle$ are expandable <code>\exp_stop:f</code> will terminate the expansion of tokens even if $\langle more tokens \rangle$ are also expandable. The function itself is an implicit space token. Inside an x -type expansion, it will retain its form, but when typeset it produces the underlying space (\sqcup).

9 Internal functions and variables

\l__exp_internal_tl

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

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

`\::N` Internal forms for the base expansion types. These names do *not* conform to the general 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 \langle name_1 \rangle: \langle arg spec_1 \rangle \langle name_2 \rangle: \langle arg spec_2 \rangle</code>
<code>\prg_set_eq_conditional:NNn</code>	<code>{\langle conditions \rangle}</code>

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

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

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

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

2 The boolean data type

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

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

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

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

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

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

3 Boolean expressions

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

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean $\langle true \rangle$ or $\langle false \rangle$. It supports the logical operations And, Or and Not as the well-known infix operators $\&\&$, $||$ and $!$ with their usual precedences. In addition to this, parentheses can be used to isolate sub-expressions. For example,

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

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

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

Updated: 2012-07-08

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

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

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

<hr/>	
<code>\bool_not_p:n</code> ☆	<code>\bool_not_p:n {<boolean expression>}</code>
Updated: 2012-07-08	Function version of <code>!(<boolean expression>)</code> within a boolean expression.
<hr/>	
<code>\bool_xor_p:nn</code> ☆	<code>\bool_xor_p:nn {<boolexpr₁>} {<boolexpr₂>}</code>
Updated: 2012-07-08	Implements an “exclusive or” operation between two boolean expressions. There is no infix operation for this logical operator.
<hr/>	

4 Logical loops

Loops using either boolean expressions or stored boolean values.

<hr/>	
<code>\bool_do_until:Nn</code> ☆	<code>\bool_do_until:Nn <boolean> {<code>}</code>
<code>\bool_do_until:cn</code> ☆	Places the <code><code></code> in the input stream for T _E X to process, and then checks the logical value of the <code><boolean></code> . If it is false then the <code><code></code> will be inserted into the input stream again and the process will loop until the <code><boolean></code> is true .
<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 true then the <code><code></code> will be inserted into the input stream again and the process will loop until the <code><boolean></code> is false .
<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 false 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 true .
<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 true 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 false .
<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 false 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 true .
<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 true 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 false .
<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

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

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

`\if_bool:N` ★ `\if_bool:N <boolean> <true code> \else: <false code> \fi:`

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

8 Internal programming functions

`\group_align_safe_begin:` ★
`\group_align_safe_end:` ★
`\group_align_safe_end:`

Updated: 2011-08-11

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

`\scan_align_safe_stop:` `\scan_align_safe_stop:`

Updated: 2011-09-06

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

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

`__prg_variable_get_scope:N` ★ `__prg_variable_get_scope:N <variable>`

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

`__prg_variable_get_type:N` ★ `__prg_variable_get_type:N <variable>`

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

<u>_prg_break_point:Nn</u> ★	_prg_break_point:Nn \\<type>_map_break: <tokens> Used to mark the end of a recursion or mapping: the functions \\<type>_map_break: and \\<type>_map_break:n use this to break out of the loop. After the loop ends, the <tokens> are inserted into the input stream. This occurs even if the break functions are <i>not</i> applied: _prg_break_point:Nn is functionally-equivalent in these cases to \\use_ii:nn.
<u>_prg_map_break:Nn</u> ★	_prg_map_break:Nn \\<type>_map_break: {<user code>} ... _prg_break_point:Nn \\<type>_map_break: {<ending code>} Breaks a recursion in mapping contexts, inserting in the input stream the <user code> after the <ending code> for the loop. The function breaks loops, inserting their <ending code>, until reaching a loop with the same <type> as its first argument. This \\<type>_map_break: argument is simply used as a recognizable marker for the <type>.
<u>\\g__prg_map_int</u>	This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions _prg_map_1:w, _prg_map_2:w, <i>etc.</i> , labelled by \\g__prg_map_int hold functions to be mapped over various list datatypes in inline and variable mappings.
<u>_prg_break_point:</u> ★	This copy of \\prg_do_nothing: is used to mark the end of a fast short-term recursions: the function _prg_break:n uses this to break out of the loop.
<u>_prg_break:</u> ★ <u>_prg_break:n</u> ★	_prg_break:n {<tokens>} ... _prg_break_point: Breaks a recursion which has no <ending code> and which is not a user-breakable mapping (see for instance \\prop_get:Nn), and inserts <tokens> in the input stream.

Part VII

The l3quark package

Quarks

1 Introduction to quarks and scan marks

Two special types of constants in L^AT_EX3 are “quarks” and “scan marks”. By convention all constants of type quark start out with `\q_`, and scan marks start with `\s_`. Scan marks are for internal use by the kernel: they are not intended for more general use.

1.1 Quarks

Quarks are control sequences that expand to themselves and should therefore *never* be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter in weird functions, with the most command use case as the ‘stop token’ (*i.e.* `\q_stop`). For example, when writing a macro to parse a user-defined date

```
\date_parse:n {19/June/1981}
```

one might write a command such as

```
\cs_new:Npn \date_parse:n #1 { \date_parse_aux:w #1 \q_stop }
\cs_new:Npn \date_parse_aux:w #1 / #2 / #3 \q_stop
{ <do something with the date> }
```

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function `\prop_get:NnN` to retrieve a value stored in some key of a property list, if the key does not exist then the return value is the quark `\q_no_value`. As mentioned above, such quarks are extremely fragile and it is imperative when using such functions that code is carefully written to check for pathological cases to avoid leakage of a quark into an uncontrolled environment.

Quarks also permit the following ingenious trick when parsing tokens: when you pick up a token in a temporary variable and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary variable to the quark using `\tl_if_eq:NNTF`. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster. An example of the quark testing functions and their use in recursion can be seen in the implementation of `\clist_map_function:NN`.

2 Defining quarks

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

3 Quark tests

The method used to define quarks means that the single token (N) tests are faster than the multi-token (n) tests. The later should therefore only be used when the argument can definitely take more than a single token.

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

4 Recursion

This module provides a uniform interface to intercepting and terminating loops as when one is doing tail recursion. The building blocks follow below and an example is shown in Section 6.

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

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

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

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

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

Updated: 2011-09-06

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

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

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

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

Updated: 2011-09-06

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

5 Clearing quarks away

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

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

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

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

6 An example of recursion with quarks

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

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

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

The definition of the internal recursion function follows. First check if either of the input tokens are the termination quarks. Then, if not, apply the inline function to the two arguments.

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

Finally, recurse:

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

Note that contrarily to L^AT_EX3 built-in mapping functions, this mapping function cannot be nested, since the second map will overwrite the definition of `_my_map_dbl_fn:nn`.

7 Internal quark functions

<code>_quark_if_recursion_tail_break:NN</code>	<code>_quark_if_recursion_tail_break:nN {<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:`.

8 Scan marks

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

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

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

<code>_scan_new:N</code>	<code>_scan_new:N <scan mark></code>
----------------------------	--

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

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

<code>_use_none_delimit_by_s_stop:w</code>	<code>_use_none_delimit_by_s_stop:w <tokens> \s_stop</code>
--	--

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

Part VIII

The l3token package

Token manipulation

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

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

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

1 All possible tokens

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

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

```
\cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }  
\show_until_if:w \tex_if:D \if_charcode:w \if:w
```

2 Character tokens

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

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

`\char_set_catcode_other:N \%`

The assignment is local.

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

Sets the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer\ expression \rangle$. This version can be used to set up characters which cannot otherwise be given (*cf.* the N-type variants). The assignment is local.

<hr/> <hr/> <code>\char_set_catcode:nn</code>	<code>\char_set_catcode:nn {⟨integer_{pr1}⟩} {⟨integer_{pr2}⟩}</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 {⟨integer_{pr1}⟩} {⟨integer_{pr2}⟩}</code>
	This function set up the behaviour of <i>⟨character⟩</i> when found inside <code>\tl_to_lowercase:n</code> , such that <i>⟨character₁⟩</i> will be converted into <i>⟨character₂⟩</i> . The two <i>⟨characters⟩</i> may be specified using an <i>⟨integer expression⟩</i> for the character code concerned. This may include the T _E X ‘ <i>⟨character⟩</i> ’ method for converting a single character into its character code:
	<pre> \char_set_lccode:nn { ‘\A } { ‘\a } % Standard behaviour \char_set_lccode:nn { ‘\A } { ‘\A + 32 } \char_set_lccode:nn { 50 } { 60 } </pre>
	The setting applies within the current T _E X group.
<hr/> <hr/> <code>\char_value_lccode:n</code> ★	<code>\char_value_lccode:n {⟨integer expression⟩}</code>
	Expands to the current lower case code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> .
<hr/> <hr/> <code>\char_show_value_lccode:n</code>	<code>\char_show_value_lccode:n {⟨integer expression⟩}</code>
	Displays the current lower case code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> on the terminal.

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

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

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

The setting applies within the current T_EX group.

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

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

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

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

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

This function sets up the math code of $\langle character \rangle$. The $\langle character \rangle$ is specified as an $\langle integer expression \rangle$ which will be used as the character code of the relevant character. The setting applies within the current T_EX group.

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

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

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

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

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

This function sets up the space factor for the $\langle character \rangle$. The $\langle character \rangle$ is specified as an $\langle integer expression \rangle$ which will be used as the character code of the relevant character. The setting applies within the current T_EX group.

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

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

<hr/> <hr/>	<code>\char_show_value_sfcode:n</code>	<code>\char_show_value_sfcode:n {\langle integer expression \rangle}</code>
		Displays the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.

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

<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.
	New: 2012-01-23	

3 Generic tokens

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

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

4 Converting tokens

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

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

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

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

Converts the given $\langle token \rangle$ into a series of characters with category code 12 (other). The current escape character will be the first character in the sequence, although this will also have category code 12 (the escape character is part of the $\langle token \rangle$). This function requires only a single expansion.

\TeX hackers note: `\token_to_str:N` is the \TeX primitive `\string` renamed.

5 Token conditionals

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Tests if $\langle token \rangle$ has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid N-type argument.

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

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

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

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

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

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

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

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

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

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

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

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

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

Updated: 2011-05-23

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

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

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

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

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

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

Updated: 2012-01-20

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

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

Updated: 2012-01-20

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

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

Updated: 2012-01-20

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

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

Updated: 2012-01-20

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

\TeX hackers note: Booleans, boxes and small integer constants are implemented as chardefs.

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

Updated: 2012-01-20

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

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

Updated: 2012-01-20

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

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

Updated: 2012-01-20

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

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

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

New: 2012-02-15

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

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

Updated: 2012-01-20

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

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

Updated: 2012-01-20

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

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

Updated: 2011-05-23

Tests if the $\langle token \rangle$ is an engine primitive.

6 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the “peek” functions. The generic `\peek_after:Nw` is provided along with a family of predefined tests for common cases. As peeking ahead does *not* skip spaces the predefined tests include both a space-respecting and space-skipping version.

`\peek_after:Nw`

`\peek_after:Nw <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.

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

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

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

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

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

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

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

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

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

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

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

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

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

Part IX

The l3int package

Integers

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

1 Integer expressions

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

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

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

and

`\tl_new:N \l_my_tl`
`\tl_set:Nn \l_my_tl { 5 }`
`\int_new:N \l_my_int`
`\int_set:Nn \l_my_int { 4 }`
`\int_eval:n { \l_my_tl + \l_my_int * 3 - (3 + 4 * 5) }`

both evaluate to -6 . The *⟨integer expression⟩* may contain the operators `+`, `-`, `*` and `/`, along with parenthesis `(` and `)`. After two expansions, `\int_eval:n` yields an *⟨integer denotation⟩* which is left in the input stream. This is *not* an *⟨internal integer⟩*, and therefore requires suitable termination if used in a 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 calculates the result of dividing the first value by the second, rounding any remainder. Ties are rounded away from zero. Note that this is identical to using `/` directly in an *⟨integer expression⟩*. The result is left in the input stream as an *⟨integer denotation⟩* after two expansions.

<hr/> <code>\int_div_truncate:nn</code> ★ <hr/>	<code>\int_div_truncate:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-02-09	Evaluates the two $\langle integer \text{ expressions} \rangle$ as described earlier, then calculates the result of dividing the first value by the second, truncating any remainder. Note that division using / rounds the result. The result is left in the input stream as an $\langle integer \text{ denotation} \rangle$ after two expansions.

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

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

2 Creating and initialising integers

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

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

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

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

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

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

New: 2012-03-03

3 Setting and incrementing integers

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

Updated: 2011-10-22

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

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

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

Updated: 2011-10-22

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

Updated: 2011-10-22

4 Using integers

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

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

Updated: 2011-10-22		
---------------------	--	--

Recovers the content of an *<integer>* and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where an *<integer>* is required (such as in the first and third arguments of `\int_compare:nNnTF`).

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

5 Integer expression conditionals

<code>\int_compare_p:nNn</code>	★	<code>\int_compare_p:nNn {<intexpr₁>} <relation> {<intexpr₂>}</code>
---------------------------------	---	--

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

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

Equal	=
Greater than	>
Less than	<

```

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

```

Updated: 2013-01-13

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

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

<code>\int_case:nnn</code> ★	<code>\int_case:nnn {<test integer expression>}</code>
New: 2012-06-03	<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>{<else 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 none of the tests are **true** then the **else code** will be left in the input stream. For example

```
\int_case:nnn
{ 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**.

<code>\int_do_while:nNnn</code> ☆	<code>\int_do_while: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 **true** then the *<code>* will be inserted into the input stream again and a loop will occur until the *<relation>* is **false**.

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

7 Integer step functions

`\int_step_function:nnnN` ☆

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

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

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

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

would print

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

`\int_step_inline:nnnn`

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

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

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

`\int_step_variable:nnnNn`

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

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

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

8 Formatting integers

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

`\int_to_arabic:n` ☆

Updated: 2011-10-22

`\int_to_arabic:n` { $\langle integer\ expression \rangle$ }

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

`\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_convert_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_convert_to_symbols:nnn {#1} { 26 }
  {
    { 1 } { a }
    { 2 } { b }
    ...
    { 26 } { z }
  }
}
```

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

Updated: 2011-09-17

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

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

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

<code>\int_to_base:nn</code> ★	<code>\int_to_base:nn {(integer expression)} {(base)}</code>
Updated: 2011-09-17	Calculates the value of the $\langle integer\ expression \rangle$ and converts it into the appropriate representation in the $\langle base \rangle$; the later may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by the upper case letters from the English alphabet. The maximum $\langle base \rangle$ value is 36.

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

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

9 Converting from other formats to integers

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

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

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

<hr/> <code>\int_from_octal:n</code> ★ <hr/>	<code>\int_from_octal:n {\langle octal number \rangle}</code> Converts the $\langle octal number \rangle$ into the integer (base 10) representation and leaves this in the input stream.
<hr/> <code>\int_from_roman:n</code> ★ <hr/>	<code>\int_from_roman:n {\langle roman numeral \rangle}</code> Converts the $\langle roman numeral \rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle roman numeral \rangle$ may be in upper or lower case; if the numeral is not valid then the resulting value will be -1 .
<hr/> <code>\int_from_base:nn</code> ★ <hr/>	<code>\int_from_base:nn {\langle number \rangle} {\langle base \rangle}</code> Converts the $\langle number \rangle$ in $\langle base \rangle$ into the appropriate value in base 10. The $\langle number \rangle$ should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum $\langle base \rangle$ value is 36.

10 Viewing integers

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

11 Constant integers

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

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

`\c_max_int`

The maximum value that can be stored as an integer.

`\c_max_register_int`

Maximum number of registers.

12 Scratch integers

`\l_tmpa_int`
`\l_tmpb_int`

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

`\g_tmpa_int`
`\g_tmpb_int`

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

13 Primitive conditionals

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

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

<code>\if_case:w</code> ★	<code>\if_case:w</code> $\langle integer \rangle$ $\langle case_0 \rangle$ <code>\or:</code> ★ $\langle case_1 \rangle$ <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:nnn` ☆
 New: 2012-06-03

```
\dim_case:nnn {<test dimension expression>}
{
  {<dimexpr case1>} {<code case1>}
  {<dimexpr case2>} {<code case2>}
  ...
  {<dimexpr casen>} {<code casen>}
}
{<else 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 none of the tests are **true** then the **else code** will be left in the input stream. For example

```
\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnn
{ 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**.

`\dim_until_do:nNnn` ☆

```
\dim_until_do:nNnn {<dimexpr1>} <relation> {<dimexpr2>} {<code>}
```

Evaluates the relationship between the two *<dimension expressions>* as described for `\dim_compare:nNnTF`, and then places the *<code>* in the input stream if the *<relation>* is **false**. After the *<code>* has been processed by T_EX 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/> <div>Updated: 2013-01-13</div>	<code>\dim_do_until:nn {<dimension relation>} {<code>}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is false then the <i><code></i> 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/> <div>Updated: 2013-01-13</div>	<code>\dim_do_while:nn {<dimension relation>} {<code>}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is true then the <i><code></i> 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/> <div>Updated: 2013-01-13</div>	<code>\dim_until_do:nn {<dimension relation>} {<code>}</code>
	Evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<hr/> <code>\dim_while_do:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_while_do:nn {<dimension relation>} {<code>}</code>
	Evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test 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/> <div>Updated: 2011-10-22</div>	<code>\dim_eval:n {<dimension expression>}</code>
	Evaluates the <i><dimension expression></i> , expanding any dimensions and token list variables within the <i><expression></i> to their content (without requiring <code>\dim_use:N/\tl_use:N</code>) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a <i><dimension denotation></i> after two expansions. This will be expressed in points (pt), and will require suitable termination if used in a T _E X-style assignment as it is <i>not</i> an <i><internal dimension></i> .

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

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

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

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

7 Viewing dim variables

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

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

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

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

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

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

8 Constant dimensions

<code>\c_max_dim</code>

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

<code>\c_zero_dim</code>

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

9 Scratch dimensions

<code>\l_tmpa_dim</code>

<code>\l_tmpb_dim</code>

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

<code>\g_tmpa_dim</code>

<code>\g_tmpb_dim</code>

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

10 Creating and initialising skip variables

`\skip_new:N`
`\skip_new:c`

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

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

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

New: 2012-03-05

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

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

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

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

New: 2012-01-07

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

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

New: 2012-03-03

11 Setting skip variables

`\skip_add:Nn`
`\skip_add:cn`
`\skip_gadd:Nn`
`\skip_gadd:cn`

`\skip_add:Nn` $\langle skip \rangle$ $\{ \langle skip \text{ expression} \rangle \}$
 Adds the result of the $\langle skip \text{ expression} \rangle$ to the current content of the $\langle skip \rangle$.

Updated: 2011-10-22

`\skip_set:Nn`
`\skip_set:cn`
`\skip_gset:Nn`
`\skip_gset:cn`

`\skip_set:Nn` $\langle skip \rangle$ $\{ \langle skip \text{ expression} \rangle \}$
 Sets $\langle skip \rangle$ to the value of $\langle skip \text{ expression} \rangle$, which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm).

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

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

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

20 Using muskip expressions and variables

<code>\muskip_eval:n</code> ★	<code>\muskip_eval:n {<muskip expression>}</code>
Updated: 2011-10-22	
	Evaluates the $\langle muskip expression \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ to their content (without requiring <code>\muskip_use:N/\tl_use:N</code>) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle muglue denotation \rangle$ after two expansions. This will be expressed in mu, and will require suitable termination if used in a TeX-style assignment as it is <i>not</i> an $\langle internal muglue \rangle$.
<code>\muskip_use:N</code> ★	<code>\muskip_use:N <muskip></code>
<code>\muskip_use:c</code> ★	
	Recovers the content of a $\langle skip \rangle$ and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ is required (such as in the argument of <code>\muskip_eval:n</code>).

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

21 Viewing muskip variables

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

24 Primitive conditional

<hr/> <code>\if_dim:w</code> <hr/>	<code>\if_dim:w</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`.

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

New: 2011-11-11

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

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

`_dim_strip_pt:n { 1 bp pt }`

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

Part XI

The l3tl package

Token lists

T_EX works with tokens, and L^AT_EX3 therefore provides a number of functions to deal with lists of tokens. Token lists may be present directly in the argument to a function:

```
\foo:n { a collection of \tokens }
```

or may be stored in a so-called “token list variable”, which have the suffix `tl`: a token list variable can also be used as the argument to a function, for example

```
\foo:N \l_some_tl
```

In both cases, functions are available to test and manipulate the lists of tokens, and these have the module prefix `tl`. In many cases, function which can be applied to token list variables are paired with similar functions for application to explicit lists of tokens: the two “views” of a token list are therefore collected together here.

A token list (explicit, or stored in a variable) can be seen either as a list of “items”, or a list of “tokens”. An item is whatever `\use:n` would grab as its argument: a single non-space token or a brace group, with optional leading explicit space characters (each item is thus itself a token list). A token is either a normal `N` argument, or `␣`, `{`, or `}` (assuming normal T_EX category codes). Thus for example

```
{ Hello } ~ world
```

contains six items (`Hello`, `w`, `o`, `r`, `l` and `d`), but thirteen tokens (`{`, `H`, `e`, `l`, `l`, `o`, `}`, `␣`, `w`, `o`, `r`, `l` and `d`). Functions which act on items are often faster than their analogue acting directly on tokens.

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

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

1 Creating and initialising token list variables

<code>\tl_new:N</code>	<code>\tl_new:N <tl var></code>
<code>\tl_new:c</code>	Creates a new <i><tl var></i> or raises an error if the name is already taken. The declaration is global. The <i><tl var></i> will initially be empty.

<code>\tl_const:Nn</code>	<code>\tl_const:Nn <tl var> {<token list>}</code>
<code>\tl_const:(Nx cn cx)</code>	Creates a new constant <i><tl var></i> or raises an error if the name is already taken. The value of the <i><tl var></i> will be set globally to the <i><token list></i> .

<code>\tl_clear:N</code>	<code>\tl_clear:N <tl var></code>
<code>\tl_clear:c</code>	Clears all entries from the <i><tl var></i> within the scope of the current T _E X group.
<code>\tl_gclear:N</code>	
<code>\tl_gclear:c</code>	

<code>\tl_clear_new:N</code>	<code>\tl_clear_new:N <tl var></code>
<code>\tl_clear_new:c</code>	Ensures that the <i><tl var></i> exists globally by applying <code>\tl_new:N</code> if necessary, then applies
<code>\tl_gclear_new:N</code>	<code>\tl_(g)clear:N</code> to leave the <i><tl var></i> empty.
<code>\tl_gclear_new:c</code>	

<code>\tl_set_eq:NN</code>	<code>\tl_set_eq:NN <tl var₁> <tl var₂></code>
<code>\tl_set_eq:(cN Nc cc)</code>	Sets the content of <i><tl var₁></i> equal to that of <i><tl var₂></i> .
<code>\tl_gset_eq:NN</code>	
<code>\tl_gset_eq:(cN Nc cc)</code>	

<code>\tl_concat:NNN</code>	<code>\tl_concat:NNN <tl var₁> <tl var₂> <tl var₃></code>
<code>\tl_concat:ccc</code>	Concatenates the content of <i><tl var₂></i> and <i><tl var₃></i> together and saves the result in
<code>\tl_gconcat:NNN</code>	<i><tl var₁></i> . The <i><tl var₂></i> will be placed at the left side of the new token list.
<code>\tl_gconcat:ccc</code>	

New: 2012-05-18

<code>\tl_if_exist_p:N</code> ★	<code>\tl_if_exist_p:N <tl var></code>
<code>\tl_if_exist_p:c</code> ★	<code>\tl_if_exist:NTF <tl var> {<true code>} {<false code>}</code>
<code>\tl_if_exist:NTF</code> ★	Tests whether the <i><tl var></i> is currently defined. This does not check that the <i><tl var></i>
<code>\tl_if_exist:cTF</code> ★	really is a token list variable.

New: 2012-03-03

2 Adding data to token list variables

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

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

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

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

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

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

3 Modifying token list variables

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

Updated: 2011-08-11

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

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

Updated: 2011-08-11

Replaces all occurrences of $\langle old\ tokens \rangle$ in the $\langle tl\ var \rangle$ with $\langle new\ tokens \rangle$. $\langle Old\ tokens \rangle$ cannot contain $\{$, $\}$ or $\#$ (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern $\langle old\ tokens \rangle$ may remain after the replacement (see `\tl_remove_all:Nn` for an example).

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

Updated: 2011-08-11

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

```
\tl_remove_all:Nn
\tl_remove_all:cn
\tl_gremove_all:Nn
\tl_gremove_all:cn
```

Updated: 2011-08-11

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

Removes all occurrences of $\langle tokens \rangle$ from the $\langle tl var \rangle$. $\langle Tokens \rangle$ cannot contain $\{, \}$ or $\#$ (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern $\langle tokens \rangle$ may remain after the removal, for instance,

```
\tl_set:Nn \l_tmpa_tl {abbccd} \tl_remove_all:Nn \l_tmpa_tl {bc}
```

will result in \l_tmpa_tl containing `abcd`.

4 Reassigning token list category codes

```
\tl_set_rescan:Nnn
\tl_set_rescan:(Nno|Nnx|cnn|cno|cnx)
\tl_gset_rescan:Nnn
\tl_gset_rescan:(Nno|Nnx|cnn|cno|cnx)
```

Updated: 2011-12-18

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

Sets $\langle tl var \rangle$ to contain $\langle tokens \rangle$, applying the category code régime specified in the $\langle setup \rangle$ before carrying out the assignment. This allows the $\langle tl var \rangle$ to contain material with category codes other than those that apply when $\langle tokens \rangle$ are absorbed. See also $\backslash tl_rescan:nn$.

```
\tl_rescan:nn
```

Updated: 2011-12-18

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

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

5 Reassigning token list character codes

```
\tl_to_lowercase:n
```

Updated: 2012-09-08

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

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

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

`\tl_to_uppercase:n`

Updated: 2012-09-08

`\tl_to_uppercase:n {<tokens>}`

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

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

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

`\tl_if_empty_p:N` ★
`\tl_if_empty_p:c` ★
`\tl_if_empty:NTF` ★
`\tl_if_empty:cTF` ★

`\tl_if_empty_p:N <tl var>`
`\tl_if_empty:NTF <tl var> {<true code>} {<false code>}`
Tests if the *<token list variable>* 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 {<token list>}`
`\tl_if_empty:nTF {<token list>} {<true code>} {<false code>}`
Tests if the *<token list>* 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 {<tl var1>} {<tl var2>}`
`\tl_if_eq:NNTF {<tl var1>} {<tl var2>} {<true code>} {<false code>}`

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

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

yields **false**.

`\tl_if_eq:nnTF`

`\tl_if_eq:nnTF <token list1> {<token list2>} {<true code>} {<false code>}`

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

`\tl_if_in:NnTF`
`\tl_if_in:cnTF`

`\tl_if_in:NnTF <tl var> {<token list>} {<true code>} {<false 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).

`\tl_if_in:nnTF`
`\tl_if_in:(Vn|on|no)TF`

`\tl_if_in:nnTF {<token list1>} {<token list2>} {<true code>} {<false 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).}*}*}*

`\tl_if_single_p:N` ★
`\tl_if_single_p:c` ★
`\tl_if_single:NTF` ★
`\tl_if_single:cTF` ★

Updated: 2011-08-13

`\tl_if_single_p:N <tl var>`

`\tl_if_single:NTF <tl var> {<true code>} {<false code>}`

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`.

`\tl_if_single_p:n` ★
`\tl_if_single:nTF` ★

Updated: 2011-08-13

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

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

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

`\tl_case:Nnn` ★
`\tl_case:cnn` ★

New: 2012-06-03

`\tl_case:Nnn <test token list variable>`

```
{
  <token list variable case1> {<code case1>}
  <token list variable case2> {<code case2>}
  ...
  <token list variable casen> {<code casen>}
}
```

`{<else code>}`

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 none of the tests are **true** then the **else** code will be left in the input stream.

7 Mapping to token lists

`\tl_map_function:NN` ★
`\tl_map_function:cN` ★

Updated: 2012-06-23

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

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

<code>\tl_map_function:nN</code> ☆	<code>\tl_map_function:nN <token list> <function></code>
Updated: 2012-06-29	Applies <i><function></i> to every <i><item></i> in the <i><token list></i> , The <i><function></i> will receive one argument for each iteration. This may be a number of tokens if the <i><item></i> was stored within braces. Hence the <i><function></i> should anticipate receiving <i>n</i> -type arguments. See also <code>\tl_map_function:NN</code> .
<code>\tl_map_inline:Nn</code> <code>\tl_map_inline:cn</code>	<code>\tl_map_inline:Nn <tl var> {<inline function>}</code>
Updated: 2012-06-29	Applies the <i><inline function></i> to every <i><item></i> stored within the <i><tl var></i> . The <i><inline function></i> should consist of code which will receive the <i><item></i> as #1. One in line mapping can be nested inside another. See also <code>\tl_map_function:NN</code> .
<code>\tl_map_inline:nn</code>	<code>\tl_map_inline:nn <token list> {<inline function>}</code>
Updated: 2012-06-29	Applies the <i><inline function></i> to every <i><item></i> stored within the <i><token list></i> . The <i><inline function></i> should consist of code which will receive the <i><item></i> as #1. One in line mapping can be nested inside another. See also <code>\tl_map_function:nN</code> .
<code>\tl_map_variable:NNn</code> <code>\tl_map_variable:cNn</code>	<code>\tl_map_variable:NNn <tl var> <variable> {<function>}</code>
Updated: 2012-06-29	Applies the <i><function></i> to every <i><item></i> stored within the <i><tl var></i> . The <i><function></i> should consist of code which will receive the <i><item></i> stored in the <i><variable></i> . One variable mapping can be nested inside another. See also <code>\tl_map_inline:Nn</code> .
<code>\tl_map_variable:nNn</code>	<code>\tl_map_variable:nNn <token list> <variable> {<function>}</code>
Updated: 2012-06-29	Applies the <i><function></i> to every <i><item></i> stored within the <i><token list></i> . The <i><function></i> should consist of code which will receive the <i><item></i> stored in the <i><variable></i> . One variable mapping can be nested inside another. See also <code>\tl_map_inline:nn</code> .
<code>\tl_map_break:</code> ☆	<code>\tl_map_break:</code>
Updated: 2012-06-29	Used to terminate a <code>\tl_map...</code> function before all entries in the <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 TeX errors.

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

`\tl_map_break:n` ☆

Updated: 2012-06-29

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

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

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

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

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

8 Using token lists

`\tl_to_str:N` ☆
`\tl_to_str:c` ☆

`\tl_to_str:N` $\langle tl var \rangle$

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.

`\tl_to_str:n` ☆

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

Converts the given $\langle tokens \rangle$ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This $\langle string \rangle$ is then left in the input stream. Note that this function requires only a single expansion.

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

`\tl_use:N` ☆
`\tl_use:c` ☆

`\tl_use:N` $\langle tl var \rangle$

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:(V|o)` ★
 New: 2012-05-13

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

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 $\langle integer denotation \rangle$.

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

`\tl_count:N` $\langle tl var \rangle$

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:(V|o)` ★
 Updated: 2012-01-08

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

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

`\tl_reverse:N` $\langle tl var \rangle$

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` ★
 New: 2012-01-08

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

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.

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

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

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

10 The first token from a token list

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

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

`\tl_head:n { abc }`

and

`\tl_head:n { ~ abc }`

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

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

yields `␣ab`. A blank *⟨token list⟩* (see `\tl_if_blank:nTF`) will result in `\tl_head:n` leaving nothing in the input stream.

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

<code>\tl_head:w</code>	★	<code>\tl_head:w <token list> { } \q_stop</code>
-------------------------	---	--

Leaves in the input stream the first *<item>* in the *<token list>*, discarding the rest of the *<token list>*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank *<token list>* (which consists only of space characters) 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.

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

<code>\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.

<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**.

```

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

```

Updated: 2012-07-09

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

```

\tl_if_head_eq_meaning_p:nN ★ \tl_if_head_eq_meaning_p:nN {\token list} \test token
\tl_if_head_eq_meaning:nNTF ★ \tl_if_head_eq_meaning:nNTF {\token list} \test token
                               {\true code} {\false code}

```

Updated: 2012-07-09

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

```

\tl_if_head_is_group_p:n ★ \tl_if_head_is_group_p:n {\token list}
\tl_if_head_is_group:nTF ★ \tl_if_head_is_group:nTF {\token list} {\true code} {\false 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.

```

\tl_if_head_is_N_type_p:n ★ \tl_if_head_is_N_type_p:n {\token list}
\tl_if_head_is_N_type:nTF ★ \tl_if_head_is_N_type:nTF {\token list} {\true code} {\false 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.

```

\tl_if_head_is_space_p:n ★ \tl_if_head_is_space_p:n {\token list}
\tl_if_head_is_space:nTF ★ \tl_if_head_is_space:nTF {\token list} {\true code} {\false 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 Viewing token lists

`\tl_show:N`
`\tl_show:c`

Updated: 2012-09-09

`\tl_show:N` $\langle tl\ var \rangle$

Displays the content of the $\langle tl\ var \rangle$ on the terminal.

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

`\tl_show:n`

Updated: 2012-09-09

`\tl_show:n` $\langle token\ list \rangle$

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.

12 Constant token lists

`\c_empty_tl`

Constant that is always empty.

`\c_job_name_tl`

Updated: 2011-08-18

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

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

`\c_space_tl`

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

13 Scratch token lists

`\l_tmpa_tl`
`\l_tmpb_tl`

Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any 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_tl`
`\g_tmpb_tl`

Scratch token lists for global assignment. These are never used by the kernel code, and so are safe for use with any 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.

14 Internal functions

<code>_tl_trim_spaces:nn</code>	<code>_tl_trim_spaces:nn { \q_mark <token list> } {<continuation>}</code>
----------------------------------	--

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

Part XII

The l3seq package

Sequences and stacks

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

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

1 Creating and initialising sequences

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

`\seq_new:N` *⟨sequence⟩*

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

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

`\seq_clear:N` *⟨sequence⟩*

Clears all items from the *⟨sequence⟩*.

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

`\seq_clear_new:N` *⟨sequence⟩*

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

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

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

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

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

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

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

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

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

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

Concatenates the content of $\langle sequence_2 \rangle$ and $\langle sequence_3 \rangle$ together and saves the result in $\langle sequence_1 \rangle$. The items in $\langle sequence_2 \rangle$ will be placed at the left side of the new sequence.

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

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

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

New: 2012-03-03

2 Appending data to sequences

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

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

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

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

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

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

3 Recovering items from sequences

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

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

Updated: 2012-05-14

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

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

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

Updated: 2012-05-19

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4 Recovering values from sequences with branching

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

`\seq_get_left:NNTF`
`\seq_get_left:cNTF`
 New: 2012-05-14
 Updated: 2012-05-19

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

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

`\seq_get_right:NNTF`
`\seq_get_right:cNTF`
 New: 2012-05-19

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

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

<code>\seq_pop_left:NNTF</code> <code>\seq_pop_left:cNTF</code>	<code>\seq_pop_left:NNTF <sequence> <token list variable> {\true code} {\false code}</code>
<p style="margin: 0;">New: 2012-05-14 Updated: 2012-05-19</p>	<p>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>i.e.</i> removes the item from a $\langle sequence \rangle$. Both the $\langle sequence \rangle$ and the $\langle token list variable \rangle$ are assigned locally.</p>

<code>\seq_gpop_left:NNTF</code> <code>\seq_gpop_left:cNTF</code>	<code>\seq_gpop_left:NNTF <sequence> <token list variable> {\true code} {\false code}</code>
<p style="margin: 0;">New: 2012-05-14 Updated: 2012-05-19</p>	<p>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>i.e.</i> 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.</p>

<code>\seq_pop_right:NNTF</code> <code>\seq_pop_right:cNTF</code>	<code>\seq_pop_right:NNTF <sequence> <token list variable> {\true code} {\false code}</code>
<p style="margin: 0;">New: 2012-05-19</p>	<p>If the $\langle sequence \rangle$ 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>i.e.</i> removes the item from a $\langle sequence \rangle$. Both the $\langle sequence \rangle$ and the $\langle token list variable \rangle$ are assigned locally.</p>

<code>\seq_gpop_right:NNTF</code> <code>\seq_gpop_right:cNTF</code>	<code>\seq_gpop_right:NNTF <sequence> <token list variable> {\true code} {\false code}</code>
<p style="margin: 0;">New: 2012-05-19</p>	<p>If the $\langle sequence \rangle$ 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>i.e.</i> 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.</p>

5 Modifying sequences

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

<code>\seq_remove_duplicates:N</code> <code>\seq_remove_duplicates:c</code> <code>\seq_gremove_duplicates:N</code> <code>\seq_gremove_duplicates:c</code>	<code>\seq_remove_duplicates:N <sequence></code> <p>Removes duplicate items from the $\langle sequence \rangle$, 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 <code>\tl_if_eq:nn(TF)</code>.</p>
--	---

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.

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

6 Sequence conditionals

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

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

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

7 Mapping to sequences

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

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

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

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

`\seq_map_break:` ☆

Updated: 2012-06-29

`\seq_map_break:`

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

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

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

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

`\seq_map_break:n` ☆

Updated: 2012-06-29

`\seq_map_break:n {<tokens>}`

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

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

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

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

`\seq_count:N` ★

`\seq_count:c` ★

New: 2012-07-13

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

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

8 Using the content of sequences directly

`\seq_use:Nnnn` ★
`\seq_use:cnnn` ★

New: 2013-05-26

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

Places the contents of the $\langle seq\ var \rangle$ in the input stream, with the appropriate $\langle separator \rangle$ between the items. Namely, if the sequence has more than two items, the $\langle separator\ between\ more\ than\ two \rangle$ is placed between each pair of items except the last, for which the $\langle separator\ between\ final\ two \rangle$ is used. If the sequence has exactly two items, then they are placed in the input stream separated by the $\langle separator\ between\ two \rangle$. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error will be raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
```

will insert “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the sequence has more than 2 items.

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

`\seq_use:Nn` ★
`\seq_use:cn` ★

New: 2013-05-26

`\seq_use:Nn` $\langle seq\ var \rangle$ $\{\langle separator \rangle\}$
`\seq_use:cn` $\langle seq\ var \rangle$ $\{\langle separator \rangle\}$

Places the contents of the $\langle seq\ var \rangle$ in the input stream, with the $\langle separator \rangle$ between the items. If the sequence has a single item, it is placed in the input stream with no $\langle separator \rangle$, and an empty sequence produces no output. An error will be raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
```

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

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

9 Sequences as stacks

Sequences can be used as stacks, where data is pushed to and popped from the top of the sequence. (The left of a sequence is the top, for performance reasons.) The stack functions for sequences are not intended to be mixed with the general ordered data

functions detailed in the previous section: a sequence should either be used as an ordered data type or as a stack, but not in both ways.

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

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

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

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

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

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

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

10 Constant and scratch sequences

`\c_empty_seq` Constant that is always empty.

New: 2012-07-02

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

New: 2012-04-26

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

New: 2012-04-26

11 Viewing sequences

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

`\seq_show:c`

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

Updated: 2012-09-09

12 Internal sequence functions

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

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error will be raised. The definition should always be set globally.

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

`__seq_push_item_def:x`

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

`__seq_pop_item_def:` `__seq_pop_item_def:`

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

Part XIII

The l3clist package

Comma separated lists

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

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

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

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

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

1 Creating and initialising comma lists

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

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

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

<code>\clist_clear:N</code>	<code>\clist_clear:N <comma list></code>
-----------------------------	--

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

<code>\clist_gclear:N</code>	
------------------------------	--

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

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

<code>\clist_clear_new:N</code>	<code>\clist_clear_new:N <comma list></code>
---------------------------------	--

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

<code>\clist_gclear_new:N</code>	
----------------------------------	--

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

Ensures that the *<comma list>* exists globally by applying `\clist_new:N` if necessary, then applies `\clist_(g)clear:N` to leave the list empty.

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

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

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

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

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

Concatenates the content of $\langle comma list_2 \rangle$ and $\langle comma list_3 \rangle$ together and saves the result in $\langle comma list_1 \rangle$. The items in $\langle comma list_2 \rangle$ will be placed at the left side of the new comma list.

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

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

```
\clist_if_exist:NTF <comma list> {\true code} {\false code}
```

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

New: 2012-03-03

2 Adding data to comma lists

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

New: 2011-09-06

```
\clist_set:Nn <comma list> {\item1},...,\itemn}
```

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

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

Updated: 2011-09-05

```
\clist_put_left:Nn <comma list> {\item1},...,\itemn}
```

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

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

Updated: 2011-09-05

```
\clist_put_right:Nn <comma list> {\item1},...,\itemn}
```

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

3 Modifying comma lists

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

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

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

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

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

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

Updated: 2011-09-06

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

4 Comma list conditionals

<code>\clist_if_empty_p:N</code> ★	<code>\clist_if_empty_p:N <comma list></code>
<code>\clist_if_empty_p:c</code> ★	<code>\clist_if_empty:NNTF <comma list> {<true code>} {<false code>}</code>
<code>\clist_if_empty:NTF</code> ★	
<code>\clist_if_empty:cTF</code> ★	

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

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

Updated: 2011-09-06

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

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

yields **false**.

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

5 Mapping to comma lists

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

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

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

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

Updated: 2012-06-29

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

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

Updated: 2012-06-29

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

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

Updated: 2012-06-29

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

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

Updated: 2012-06-29

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

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

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

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

`\clist_map_break:n` ☆

Updated: 2012-06-29

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

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

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

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

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

`\clist_count:N` ☆

`\clist_count:(c|n)` ☆

New: 2012-07-13

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

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

6 Using the content of comma lists directly

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

New: 2013-05-26

Places the contents of the $\langle\textit{clist var}\rangle$ in the input stream, with the appropriate $\langle\textit{separator}\rangle$ between the items. Namely, if the comma list has more than two items, the $\langle\textit{separator between more than two}\rangle$ is placed between each pair of items except the last, for which the $\langle\textit{separator between final two}\rangle$ is used. If the comma list has exactly two items, then they are placed in the input stream separated by the $\langle\textit{separator between two}\rangle$. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error will be raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nnnn \l_tmpa_clist { ~and~ } { ,~ } { ,~and~ }
```

will insert “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the comma list has more than 2 items.

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

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

New: 2013-05-26

Places the contents of the $\langle\textit{clist var}\rangle$ in the input stream, with the $\langle\textit{separator}\rangle$ between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error will be raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { ~and~ }
```

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

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

7 Comma lists as stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The

stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

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

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

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

8 Viewing comma lists

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

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

9 Constant and scratch comma lists

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

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

<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 $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_clist</code>	
New: 2011-09-06	

Part XIV

The l3prop package

Property lists

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

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

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

1 Creating and initialising property lists

 $\backslash\text{prop_new:N}$
 $\backslash\text{prop_new:c}$

 $\backslash\text{prop_new:N}$ $\langle property\ list \rangle$

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.

 $\backslash\text{prop_clear:N}$
 $\backslash\text{prop_clear:c}$
 $\backslash\text{prop_gclear:N}$
 $\backslash\text{prop_gclear:c}$

 $\backslash\text{prop_clear:N}$ $\langle property\ list \rangle$

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

 $\backslash\text{prop_clear_new:N}$
 $\backslash\text{prop_clear_new:c}$
 $\backslash\text{prop_gclear_new:N}$
 $\backslash\text{prop_gclear_new:c}$

 $\backslash\text{prop_clear_new:N}$ $\langle property\ list \rangle$

Ensures that the $\langle property\ list \rangle$ exists globally by applying $\backslash\text{prop_new:N}$ if necessary, then applies $\backslash\text{prop_gclear:N}$ to leave the list empty.

 $\backslash\text{prop_set_eq:NN}$
 $\backslash\text{prop_set_eq:(cN|Nc|cc)}$
 $\backslash\text{prop_gset_eq:NN}$
 $\backslash\text{prop_gset_eq:(cN|Nc|cc)}$

 $\backslash\text{prop_set_eq:NN}$ $\langle property\ list_1 \rangle$ $\langle property\ list_2 \rangle$

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

2 Adding entries to property lists

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

Updated: 2012-07-09

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

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

3 Recovering values from property lists

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

Updated: 2011-08-28

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

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

Updated: 2011-08-18

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

Updated: 2011-08-18

4 Modifying property lists

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

New: 2012-05-12

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

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

5 Property list conditionals

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

New: 2012-03-03

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

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

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

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

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

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

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

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

Updated: 2011-09-15

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

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

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

6 Recovering values from property lists with branching

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

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

Updated: 2012-05-19

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

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

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

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

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

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

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

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

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

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

7 Mapping to property lists

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

Updated: 2013-01-08

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

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

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

Updated: 2013-01-08

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

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

\prop_map_break: ☆

Updated: 2012-06-29

\prop_map_break:

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

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

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

\prop_map_break:n ☆

Updated: 2012-06-29

\prop_map_break:n {*⟨tokens⟩*}

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

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

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

8 Viewing property lists

\prop_show:N

\prop_show:c

Updated: 2012-09-09

\prop_show:N *⟨property list⟩*

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

9 Scratch property lists

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

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

10 Constants

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

11 Internal property list functions

<code>\s__prop</code>	The internal token used to separate out property list entries, surrounding each $\langle key \rangle$.
-----------------------	---

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

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

Part XV

The l3box package

Boxes

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

1 Creating and initialising boxes

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

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

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

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

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

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

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

New: 2012-03-03

2 Using boxes

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

T_EXhackers note: This is the T_EX primitive `\copy`.

<code>\box_use_clear:N</code>	<code>\box_use_clear:N</code> $\langle box \rangle$
<code>\box_use_clear:c</code>	Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting, then globally clears the content of the $\langle box \rangle$.

T_EXhackers note: This is the T_EX primitive `\box`.

<code>\box_move_right:nn</code>	<code>\box_move_right:nn</code> $\{\langle dimexpr \rangle\}$ $\{\langle box\ function \rangle\}$
<code>\box_move_left:nn</code>	This function operates in vertical mode, and inserts the material specified by the $\langle box\ function \rangle$ such that its reference point is displaced horizontally by the given $\langle dimexpr \rangle$ from the reference point for typesetting, to the right or left as appropriate. The $\langle box\ function \rangle$ should be a box operation such as <code>\box_use:N</code> $\langle box \rangle$ or a “raw” box specification such as <code>\vbox:n</code> { xyz }.

<code>\box_move_up:nn</code>	<code>\box_move_up:nn</code> $\{\langle dimexpr \rangle\}$ $\{\langle box\ function \rangle\}$
<code>\box_move_down:nn</code>	This function operates in horizontal mode, and inserts the material specified by the $\langle box\ function \rangle$ such that its reference point is displaced vertical by the given $\langle dimexpr \rangle$ from the reference point for typesetting, up or down as appropriate. The $\langle box\ function \rangle$ should be a box operation such as <code>\box_use:N</code> $\langle box \rangle$ or a “raw” box specification such as <code>\vbox:n</code> { xyz }.

3 Measuring and setting box dimensions

<code>\box_dp:N</code>	<code>\box_dp:N</code> $\langle box \rangle$
<code>\box_dp:c</code>	Calculates the depth (below the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension\ expression \rangle$.

T_EXhackers note: This is the T_EX primitive `\dp`.

<code>\box_ht:N</code>	<code>\box_ht:N <box></code>
<code>\box_ht:c</code>	Calculates the height (above the baseline) of the <code><box></code> in a form suitable for use in a <code><dimension expression></code> .

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 <code><box></code> in a form suitable for use in a <code><dimension expression></code> .

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 <code><box></code> to the value of the <code>{<dimension expression>}</code> . 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 <code><box></code> to the value of the <code>{<dimension expression>}</code> . 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 <code><box></code> to the value of the <code>{<dimension expression>}</code> . 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:NNTF <box> {<true code>} {<false code>}</code>
<code>\box_if_empty:NTF</code>	★ Tests if <code><box></code> 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:NNTF <box> {<true code>} {<false code>}</code>
<code>\box_if_horizontal:NTF</code>	★ Tests if <code><box></code> 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:NNTF <box> {<true code>} {<false code>}</code>
<code>\box_if_vertical:NTF</code>	★ Tests if <code><box></code> is a vertical box.
<code>\box_if_vertical:cTF</code>	★

5 The last box inserted

<hr/> <code>\box_set_to_last:N</code>	<code>\box_set_to_last:N</code> $\langle box \rangle$
<code>\box_set_to_last:c</code>	
<code>\box_gset_to_last:N</code>	Sets the $\langle box \rangle$ equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the $\langle box \rangle$ will always be void as it is not possible to recover the last added item.
<code>\box_gset_to_last:c</code>	

6 Constant boxes

<hr/> <code>\c_empty_box</code>	This is a permanently empty box, which is neither set as horizontal nor vertical.
<code>Updated: 2012-11-04</code>	

7 Scratch boxes

<hr/> <code>\l_tmpa_box</code>	Scratch boxes for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_box</code>	
<code>Updated: 2012-11-04</code>	

<hr/> <code>\g_tmpa_box</code>	Scratch boxes for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_box</code>	

8 Viewing box contents

<hr/> <code>\box_show:N</code>	<code>\box_show:N</code> $\langle box \rangle$
<code>\box_show:c</code>	Shows full details of the content of the $\langle box \rangle$ in the terminal.
<code>Updated: 2012-05-11</code>	
<hr/> <code>\box_show:Nnn</code>	<code>\box_show:Nnn</code> $\langle box \rangle$ $\langle intexpr_1 \rangle$ $\langle intexpr_2 \rangle$
<code>\box_show:cnn</code>	Display the contents of $\langle box \rangle$ in the terminal, showing the first $\langle intexpr_1 \rangle$ items of the box, and descending into $\langle intexpr_2 \rangle$ group levels.
<code>New: 2012-05-11</code>	
<hr/> <code>\box_log:N</code>	<code>\box_log:N</code> $\langle box \rangle$
<code>\box_log:c</code>	Writes full details of the content of the $\langle box \rangle$ to the log.
<code>New: 2012-05-11</code>	

<hr/> <code>\box_log:Nnn</code> <hr/>	<code>\box_show: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`.

`\vbox_unpack:N`
`\vbox_unpack:c`

`\vbox_unpack:N` $\langle box \rangle$

Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.

T_EXhackers note: This is the T_EX primitive `\unvcopy`.

`\vbox_unpack_clear:N`
`\vbox_unpack_clear:c`

`\vbox_unpack:N` $\langle box \rangle$

Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The $\langle box \rangle$ is then cleared globally.

T_EXhackers note: This is the T_EX primitive `\unvbox`.

11 Primitive box conditionals

`\if_hbox:N` ★

`\if_hbox:N` $\langle box \rangle$
 $\langle true\ code \rangle$

`\else:`
 $\langle false\ code \rangle$

`\fi:`

Tests if $\langle box \rangle$ is a horizontal box.

T_EXhackers note: This is the T_EX primitive `\ifhbox`.

`\if_vbox:N` ★

`\if_vbox:N` $\langle box \rangle$
 $\langle true\ code \rangle$

`\else:`
 $\langle false\ code \rangle$

`\fi:`

Tests if $\langle box \rangle$ is a vertical box.

T_EXhackers note: This is the T_EX primitive `\ifvbox`.

`\if_box_empty:N` ★

`\if_box_empty:N` $\langle box \rangle$
 $\langle true\ code \rangle$

`\else:`
 $\langle false\ code \rangle$

`\fi:`

Tests if $\langle box \rangle$ is an empty (void) box.

T_EXhackers note: This is the T_EX primitive `\ifvoid`.

Part XVI

The l3coffins package

Coffin code layer

The material in this module provides the low-level support system for coffins. For details about the design concept of a coffin, see the xcoffins module (in the l3experimental bundle).

1 Creating and initialising coffins

`\coffin_new:N`

`\coffin_new:c`

New: 2011-08-17

`\coffin_new:N` $\langle coffin \rangle$

Creates a new $\langle coffin \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle coffin \rangle$ will initially be empty.

`\coffin_clear:N`

`\coffin_clear:c`

New: 2011-08-17

`\coffin_clear:N` $\langle coffin \rangle$

Clears the content of the $\langle coffin \rangle$ within the current TeX group level.

`\coffin_set_eq:NN`

`\coffin_set_eq:(Nc|cN|cc)`

New: 2011-08-17

`\coffin_set_eq:NN` $\langle coffin_1 \rangle$ $\langle coffin_2 \rangle$

Sets both the content and poles of $\langle coffin_1 \rangle$ equal to those of $\langle coffin_2 \rangle$ within the current TeX group level.

`\coffin_if_exist_p:N` ★

`\coffin_if_exist_p:c` ★

`\coffin_if_exist:NTF` ★

`\coffin_if_exist:cTF` ★

New: 2012-06-20

`\coffin_if_exist_p:N` $\langle box \rangle$

`\coffin_if_exist:NTF` $\langle box \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests whether the $\langle coffin \rangle$ is currently defined.

2 Setting coffin content and poles

All coffin functions create and manipulate coffins locally within the current TeX group level.

`\hcoffin_set:Nn`

`\hcoffin_set:cn`

New: 2011-08-17

Updated: 2011-09-03

`\hcoffin_set:Nn` $\langle coffin \rangle$ $\{\langle material \rangle\}$

Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.

<hr/> <code>\hcoffin_set:Nw</code> <code>\hcoffin_set:cw</code> <code>\hcoffin_set_end:</code> <hr/> New: 2011-09-10	<code>\hcoffin_set:Nw <coffin> <material> \hcoffin_set_end:</code> Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.
<hr/> <code>\vcoffin_set:Nnn</code> <code>\vcoffin_set:cnn</code> <hr/> New: 2011-08-17 Updated: 2012-05-22	<code>\vcoffin_set:Nnn <coffin> {<width>} {<material>}</code> Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.
<hr/> <code>\vcoffin_set:Nnw</code> <code>\vcoffin_set:cnw</code> <code>\vcoffin_set_end:</code> <hr/> New: 2011-09-10 Updated: 2012-05-22	<code>\vcoffin_set:Nnw <coffin> {<width>} <material> \vcoffin_set_end:</code> Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.
<hr/> <code>\coffin_set_horizontal_pole:Nnn</code> <code>\coffin_set_horizontal_pole:cnn</code> <hr/> New: 2012-07-20	<code>\coffin_set_horizontal_pole:Nnn <coffin> {<pole>} {<offset>}</code> Sets the $\langle pole \rangle$ to run horizontally through the $\langle coffin \rangle$. The $\langle pole \rangle$ will be located at the $\langle offset \rangle$ from the bottom edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.
<hr/> <code>\coffin_set_vertical_pole:Nnn</code> <code>\coffin_set_vertical_pole:cnn</code> <hr/> New: 2012-07-20	<code>\coffin_set_vertical_pole:Nnn <coffin> {<pole>} {<offset>}</code> Sets the $\langle pole \rangle$ to run vertically through the $\langle coffin \rangle$. The $\langle pole \rangle$ will be located at the $\langle offset \rangle$ from the left-hand edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.

3 Joining and using coffins

<hr/>	<hr/>
<code>\coffin_attach:NnnNnnnn</code>	<code>\coffin_attach:NnnNnnnn</code>
<code>\coffin_attach:(cnnNnnnn Nnnnnnn cnnnnnn)</code>	<code>\coffin_attach:(cnnNnnnn Nnnnnnn cnnnnnn)</code>
	<code>\coffin_1 \{ \coffin_1-pole_1 \} \{ \coffin_1-pole_2 \}</code>
	<code>\coffin_2 \{ \coffin_2-pole_1 \} \{ \coffin_2-pole_2 \}</code>
	<code>\{ \langle x-offset \rangle \} \{ \langle y-offset \rangle \}</code>

This function attaches $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ is not altered, *i.e.* $\langle coffin_2 \rangle$ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

<hr/>	<hr/>
<code>\coffin_join:NnnNnnnn</code>	<code>\coffin_join:NnnNnnnn</code>
<code>\coffin_join:(cnnNnnnn Nnnnnnn cnnnnnn)</code>	<code>\coffin_join:(cnnNnnnn Nnnnnnn cnnnnnn)</code>
	<code>\coffin_1 \{ \coffin_1-pole_1 \} \{ \coffin_1-pole_2 \}</code>
	<code>\coffin_2 \{ \coffin_2-pole_1 \} \{ \coffin_2-pole_2 \}</code>
	<code>\{ \langle x-offset \rangle \} \{ \langle y-offset \rangle \}</code>

This function joins $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ may expand. The new bounding box will cover the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

<hr/>	<hr/>
<code>\coffin_typeset:Nnnnn</code>	<code>\coffin_typeset:Nnnnn \langle coffin \rangle \{ \langle pole_1 \rangle \} \{ \langle pole_2 \rangle \}</code>
<code>\coffin_typeset:cnnnn</code>	<code>\{ \langle x-offset \rangle \} \{ \langle y-offset \rangle \}</code>
<hr/>	<hr/>

Updated: 2012-07-20

Typesetting is carried out by first calculating $\langle handle \rangle$, the point of intersection of $\langle pole_1 \rangle$ and $\langle pole_2 \rangle$. The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the $\langle handle \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the “parent” coffin is the current insertion point.

4 Measuring coffins

<hr/>	<hr/>
<code>\coffin_dp:N</code>	<code>\coffin_dp:N \langle coffin \rangle</code>
<code>\coffin_dp:c</code>	
<hr/>	<hr/>

Calculates the depth (below the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

<hr/> <code>\coffin_ht:N</code> <hr/>	<code>\coffin_ht:N</code> $\langle\textit{coffin}\rangle$
<code>\coffin_ht:c</code> <hr/>	Calculates the height (above the baseline) of the $\langle\textit{coffin}\rangle$ in a form suitable for use in a $\langle\textit{dimension expression}\rangle$.
<hr/> <code>\coffin_wd:N</code> <hr/>	<code>\coffin_wd:N</code> $\langle\textit{coffin}\rangle$
<code>\coffin_wd:c</code> <hr/>	Calculates the width of the $\langle\textit{coffin}\rangle$ in a form suitable for use in a $\langle\textit{dimension expression}\rangle$.

5 Coffin diagnostics

<hr/> <code>\coffin_display_handles:Nn</code> <hr/>	<code>\coffin_display_handles:Nn</code> $\langle\textit{coffin}\rangle$ $\{\langle\textit{colour}\rangle\}$
<code>\coffin_display_handles:cn</code> <hr/>	This function first calculates the intersections between all of the $\langle\textit{poles}\rangle$ of the $\langle\textit{coffin}\rangle$ to give a set of $\langle\textit{handles}\rangle$. It then prints the $\langle\textit{coffin}\rangle$ at the current location in the source, with the position of the $\langle\textit{handles}\rangle$ marked on the coffin. The $\langle\textit{handles}\rangle$ will be labelled as part of this process: the locations of the $\langle\textit{handles}\rangle$ and the labels are both printed in the $\langle\textit{colour}\rangle$ specified.
Updated: 2011-09-02 <hr/>	

<hr/> <code>\coffin_mark_handle:Nnnn</code> <hr/>	<code>\coffin_mark_handle:Nnnn</code> $\langle\textit{coffin}\rangle$ $\{\langle\textit{pole}_1\rangle\}$ $\{\langle\textit{pole}_2\rangle\}$ $\{\langle\textit{colour}\rangle\}$
<code>\coffin_mark_handle:cnnn</code> <hr/>	This function first calculates the $\langle\textit{handle}\rangle$ for the $\langle\textit{coffin}\rangle$ as defined by the intersection of $\langle\textit{pole}_1\rangle$ and $\langle\textit{pole}_2\rangle$. It then marks the position of the $\langle\textit{handle}\rangle$ on the $\langle\textit{coffin}\rangle$. The $\langle\textit{handle}\rangle$ will be labelled as part of this process: the location of the $\langle\textit{handle}\rangle$ and the label are both printed in the $\langle\textit{colour}\rangle$ specified.
Updated: 2011-09-02 <hr/>	

<hr/> <code>\coffin_show_structure:N</code> <hr/>	<code>\coffin_show_structure:N</code> $\langle\textit{coffin}\rangle$
<code>\coffin_show_structure:c</code> <hr/>	This function shows the structural information about the $\langle\textit{coffin}\rangle$ in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.
Updated: 2012-09-09 <hr/>	

Notice that the poles of a coffin are defined by four values: the x and y co-ordinates of a point that the pole passes through and the x - and y -components of a vector denoting the direction of the pole. It is the ratio between the later, rather than the absolute values, which determines the direction of the pole.

5.1 Constants and variables

<hr/> <code>\c_empty_coffin</code> <hr/>	A permanently empty coffin.
<hr/> <code>\l_tmpa_coffin</code> <hr/>	Scratch coffins for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_coffin</code> <hr/>	
New: 2012-06-19 <hr/>	

Part XVII

The l3color package

Colour support

This module provides support for color in L^AT_EX3. At present, the material here is mainly intended to support a small number of low-level requirements in other l3kernel modules.

1 Colour in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

```
\color_group_begin:
\color_group_end:
```

New: 2011-09-03

```
\color_group_begin:
```

```
...
```

```
\color_group_end:
```

Creates a color group: one used to “trap” color settings.

```
\color_ensure_current:
```

New: 2011-09-03

```
\color_ensure_current:
```

Ensures that material inside a box will use the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin: ... \color_group_end: group`.

Part XVIII

The l3msg package

Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The `l3msg` module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by `l3msg` to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message *class* has to be made, for example `error`, `warning` or `info`.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behaviour can be altered and messages can be entirely suppressed.

1 Creating new messages

All messages have to be created before they can be used. The text of messages will automatically be wrapped to the length available in the console. As a result, formatting is only needed where it will help to show meaning. In particular, `\\` may be used to force a new line and `_` forces an explicit space. Additionally, `\{`, `\#`, `\}`, `\%` and `\~` can be used to produce the corresponding character.

Messages may be subdivided *by one level* using the `/` character. This is used within the message filtering system to allow for example the L^AT_EX kernel messages to belong to the module `LaTeX` while still being filterable at a more granular level. Thus for example

```
\msg_new:nnnn { mymodule } { submodule / message } ...
```

will allow only those messages from the `submodule` to be filtered out.

```
\msg_new:nnnn
```

```
\msg_new:nnn
```

Updated: 2011-08-16

```
\msg_new:nnnn {<module>} {<message>} {<text>} {<more text>}
```

Creates a *<message>* for a given *<module>*. The message will be defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (**#1** to **#4**) can be used: these will be supplied at the time the message is used. An error will be raised if the *<message>* already exists.

```
\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 proceeded by the text **on line**.

```
\msg_line_number: ★
```

```
\msg_line_number:
```

Prints the current line number when a message is given.

```
\msg_fatal_text:n ★
```

```
\msg_fatal_text:n {<module>}
```

Produces the standard text

Fatal *<module>* error

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *<module>* to be included.

```
\msg_critical_text:n ★
```

```
\msg_critical_text:n {<module>}
```

Produces the standard text

Critical *<module>* error

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *<module>* to be included.

```
\msg_error_text:n ★
```

```
\msg_error_text:n {<module>}
```

Produces the standard text

***<module>* error**

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *<module>* to be included.

<code>\msg_warning_text:n</code>	★	<code>\msg_warning_text:n {<module>}</code>
----------------------------------	---	---

Produces the standard text

`<module> warning`

This function can be redefined to alter the language in which the message is given, using #1 as the name of the `<module>` to be included.

<code>\msg_info_text:n</code>	★	<code>\msg_info_text:n {<module>}</code>
-------------------------------	---	--

Produces the standard text:

`<module> info`

This function can be redefined to alter the language in which the message is given, using #1 as the name of the `<module>` to be included.

<code>\msg_see_documentation_text:n</code>	★	<code>\msg_see_documentation_text:n {<module>}</code>
--	---	---

Produces the standard text

`See the <module> documentation for further information.`

This function can be redefined to alter the language in which the message is given, using #1 as the name of the `<module>` to be included.

3 Issuing messages

Messages behave differently depending on the message class. In all cases, the message may be issued supplying 0 to 4 arguments. If the number of arguments supplied here does not match the number in the definition of the message, extra arguments will be ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments will be converted to strings before being added to the message text: the `x`-type variants should be used to expand material.

<code>\msg_fatal:nnnnnn</code>	<code>\msg_fatal:nnnnnn {<module>} {<message>} {<arg one>}</code>
<code>\msg_fatal:(nnnnn nnnn nnn nn nnxxxx nnxxx nnxx nnx)</code>	<code>{<arg two>} {<arg three>} {<arg four>}</code>

Updated: 2012-08-11

Issues `<module> error <message>`, passing `<arg one>` to `<arg four>` to the text-creating functions. After issuing a fatal error the `TEX` run will halt.

```
\msg_critical:nnnnnn  
\msg_critical:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing a critical error, T_EX will stop reading the current input file. This may halt the T_EX run (if the current file is the main file) or may abort reading a sub-file.

T_EXhackers note: The T_EX `\endinput` primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

```
\msg_error:nnnnnn  
\msg_error:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The error will interrupt processing and issue the text at the terminal. After user input, the run will continue.

```
\msg_warning:nnnnnn  
\msg_warning:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues $\langle module \rangle$ warning $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The warning text will be added to the log file and the terminal, but the T_EX run will not be interrupted.

```
\msg_info:nnnnnn  
\msg_info:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues $\langle module \rangle$ information $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The information text will be added to the log file.

```
\msg_log:nnnnnn  
\msg_log:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues $\langle module \rangle$ information $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The information text will be added to the log file: the output is briefer than `\msg_info:nnnnnn`.

<code>\msg_none:nnnnnn</code> <code>\msg_none:(nnnnn nnnn nnn nn nnxxxx nnxxx nnxx nnx)</code>	<code>\msg_none:nnnnnn {<module>} {<message>} {<arg one>}</code> <code>{<arg two>} {<arg three>} {<arg four>}</code>
---	---

Updated: 2012-08-11

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

4 Redirecting messages

Each message has a “name”, which can be used to alter the behaviour of the message when it is given. Thus we might have

```
\msg_new:nnnn { module } { my-message } { Some-text } { Some-more-text }
```

to define a message, with

```
\msg_error:nn { module } { my-message }
```

when it is used. With no filtering, this will raise an error. However, we could alter the behaviour with

```
\msg_redirect_class:nn { error } { warning }
```

to turn all errors into warnings, or with

```
\msg_redirect_module:nnn { module } { error } { warning }
```

to alter only messages from that module, or even

```
\msg_redirect_name:nnn { module } { my-message } { warning }
```

to target just one message. Redirection applies first to individual messages, then to messages from one module and finally to messages of one class. Thus it is possible to select out an individual message for special treatment even if the entire class is already redirected.

Multiple redirections are possible. Redirections can be cancelled by providing an empty argument for the target class. Redirection to a missing class will raise errors immediately. Infinite loops are prevented by eliminating the redirection starting from the target of the redirection that caused the loop to appear. Namely, if redirections are requested as $A \rightarrow B$, $B \rightarrow C$ and $C \rightarrow A$ in this order, then the $A \rightarrow B$ redirection is cancelled.

<code>\msg_redirect_class:nn</code>	<code>\msg_redirect_class:nn {<class one>} {<class two>}</code>
-------------------------------------	---

Updated: 2012-04-27

Changes the behaviour of messages of *<class one>* so that they are processed using the code for those of *<class two>*.

<hr/> <code>\msg_redirect_module:nnn</code> <hr/>	<code>\msg_redirect_module:nnn {<module>} {<class one>} {<class two>}</code>
Updated: 2012-04-27	Redirects message of <i><class one></i> for <i><module></i> to act as though they were from <i><class two></i> . Messages of <i><class one></i> from sources other than <i><module></i> are not affected by this redirection. This function can be used to make some messages “silent” by default. For example, all of the warning messages of <i><module></i> could be turned off with:

```
\msg_redirect_module:nnn { module } { warning } { none }
```

<hr/> <code>\msg_redirect_name:nnn</code> <hr/>	<code>\msg_redirect_name:nnn {<module>} {<message>} {<class>}</code>
Updated: 2012-04-27	Redirects a specific <i><message></i> from a specific <i><module></i> to act as a member of <i><class></i> of messages. No further redirection is performed. This function can be used to make a selected message “silent” without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

5 Low-level message functions

The lower-level message functions should usually be accessed from the higher-level system. However, there are occasions where direct access to these functions is desirable.

<hr/> <code>\msg_interrupt:nnn</code> <hr/>	<code>\msg_interrupt:nnn {<first line>} {<text>} {<extra text>}</code>
New: 2012-06-28	Interrupts the TeX run, issuing a formatted message comprising <i><first line></i> and <i><text></i> laid out in the format

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! <first line>
!
! <text>
!.....
```

where the *<text>* will be wrapped to fit within the current line length. The user may then request more information, at which stage the *<extra text>* will be shown in the terminal in the format

```
|,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
| <extra text>
|.....
```

where the *<extra text>* will be wrapped within the current line length. Wrapping of both *<text>* and *<more text>* takes place using `\iow_wrap:nnn`; the documentation for the latter should be consulted for full details.

`\msg_log:n`
New: 2012-06-28

`\msg_log:n {⟨text⟩}`
Writes to the log file with the $\langle text \rangle$ laid out in the format

```
.....
. <text>
.....
```

where the $\langle text \rangle$ 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`
New: 2012-06-28

`\msg_term:n {⟨text⟩}`
Writes to the terminal and log file with the $\langle text \rangle$ laid out in the format

```
*****
* <text>
*****
```

where the $\langle text \rangle$ 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 $\langle message \rangle$ for a given $\langle module \rangle$. The message will be defined to first give $\langle text \rangle$ and then $\langle more text \rangle$ if the user requests it. If no $\langle more text \rangle$ is available then a standard text is given instead. Within $\langle text \rangle$ and $\langle more text \rangle$ four parameters (**#1** to **#4**) can be used: these will be supplied and expanded at the time the message is used. An error will be raised if the $\langle message \rangle$ 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 $\langle message \rangle$ for a given $\langle module \rangle$. The message will be defined to first give $\langle text \rangle$ and then $\langle more text \rangle$ if the user requests it. If no $\langle more text \rangle$ is available then a standard text is given instead. Within $\langle text \rangle$ and $\langle more text \rangle$ four parameters (**#1** to **#4**) can be used: these will be supplied 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.

Part XIX

The l3keys package

Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

```
\MyModuleSetup{
  key-one = value one,
  key-two = value two
}
```

or

```
\MyModuleMacro[
  key-one = value one,
  key-two = value two
]{argument}
```

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { mymodule }
{
  key-one .code:n    = code including parameter #1,
  key-two .tl_set:N = \l_mymodule_store_tl
}
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }
{
  key-one = value one,
  key-two = value two
}
```

At a document level, `\keys_set:nn` will be used within a document function, for example

```
\DeclareDocumentCommand \MyModuleSetup { m }
{ \keys_set:nn { mymodule } { #1 } }
\DeclareDocumentCommand \MyModuleMacro { o m }
{
```

```

\group_begin:
  \keys_set:nn { mymodule } { #1 }
  % Main code for \MyModuleMacro
\group_end:
}

```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As will be discussed in section 2, it is suggested that the character `/` is reserved for sub-division of keys into logical groups. Functions and variables are *not* expanded when creating key names, and so

```

\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
{
  \l_mymodule_tmp_tl .code:n = code
}

```

will create a key called `\l_mymodule_tmp_tl`, and not one called `key`.

1 Creating keys

```

\keys_define:nn \keys_define:nn {<module>} {<keyval list>}

```

Parses the *<keyval list>* and defines the keys listed there for *<module>*. The *<module>* name should be a text value, but there are no restrictions on the nature of the text. In practice the *<module>* should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The *<keyval list>* should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```

\keys_define:nn { mymodule }
{
  keyname .code:n = Some~code~using~#1,
  keyname .value_required:
}

```

where the properties of the key begin from the `.` after the key name.

The various properties available take either no arguments at all, or require one or more arguments. This is indicated in the name of the property using an argument specification. In the following discussion, each property is illustrated attached to an arbitrary *<key>*, which when used may be supplied with a *<value>*. All key *definitions* are local.

```
.bool_set:N
.bool_set:c
.bool_gset:N
.bool_gset:c
```

Updated: 2013-07-08

$\langle key \rangle$.bool_set:N = $\langle boolean \rangle$

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to $\langle value \rangle$ (which must be either **true** or **false**). If the variable does not exist, it will be created globally at the point that the key is set up.

```
.bool_set_inverse:N
.bool_set_inverse:c
.bool_gset_inverse:N
.bool_gset_inverse:c
```

New: 2011-08-28
Updated: 2013-07-08

$\langle key \rangle$.bool_set_inverse:N = $\langle boolean \rangle$

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to the logical inverse of $\langle value \rangle$ (which must be either **true** or **false**). If the $\langle boolean \rangle$ does not exist, it will be created globally at the point that the key is set up.

```
.choice:
```

$\langle key \rangle$.choice:

Sets $\langle key \rangle$ to act as a choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section [3](#).

```
.choices:nn
.choices:Vn
.choices:on
.choices:xn
```

New: 2011-08-21
Updated: 2013-07-10

$\langle key \rangle$.choices:nn $\langle choices \rangle$ $\langle code \rangle$

Sets $\langle key \rangle$ to act as a choice key, and defines a series $\langle choices \rangle$ which are implemented using the $\langle code \rangle$. Inside $\langle code \rangle$, $\backslash l_keys_choice_tl$ 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>	<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
<code>.default:o</code>	if only the key name is given, but not if a blank <i><value></i> is given:
<code>.default:x</code>	
<hr/>	
Updated: 2013-07-09	<pre> \keys_define:nn { mymodule } { key .code:n = Hello~#1, key .default:n = World } \keys_set:nn { mymodule } { key = Fred, % Prints 'Hello Fred' key, % Prints 'Hello World' key = , % Prints 'Hello ' } </pre>
<hr/>	
<code>.dim_set:N</code>	<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
<code>.dim_gset:N</code>	variable does not exist, it will be created globally at the point that the key is set up.
<code>.dim_gset:c</code>	
<hr/>	
<code>.fp_set:N</code>	<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).
<code>.fp_gset:N</code>	If the variable does not exist, it will be created globally at the point that the key is set
<code>.fp_gset:c</code>	up.
<hr/>	
<code>.initial:n</code>	<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>	<code>\keys_set:nn {<module>} { <key> = <value> }</code>
<hr/>	
Updated: 2013-07-09	
<hr/>	
<code>.int_set:N</code>	<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
<code>.int_gset:N</code>	variable does not exist, it will be created globally at the point that the key is set up.
<code>.int_gset:c</code>	
<hr/>	
<code>.meta:n</code>	<code><key> .meta:n = {<keyval list>}</code>
<hr/>	
Updated: 2013-07-10	Makes <i><key></i> a meta-key, which will set <i><keyval list></i> in one go. If <i><key></i> is given with a
	value at the time the key is used, then the value will be passed through to the subsidiary
	<i><keys></i> for processing (as #1).

<hr/> <code>.meta:nn</code> <hr/>	<code><key> .meta:nn = {<path>} {<keyval list>}</code>
<hr/> New: 2013-07-10 <hr/>	Makes <code><key></code> a meta-key, which will set <code><keyval list></code> in one go using the <code><path></code> in place of the current one. If <code><key></code> is given with a value at the time the key is used, then the value will be passed through to the subsidiary <code><keys></code> for processing (as #1).
<hr/> <code>.multichoice:</code> <hr/>	<code><key> .multichoice:</code>
<hr/> New: 2011-08-21 <hr/>	Sets <code><key></code> to act as a multiple choice key. Each valid choice for <code><key></code> must then be created, as discussed in section 3.
<hr/> <code>.multichoices:nn</code> <code>.multichoices:Vn</code> <code>.multichoices:on</code> <code>.multichoices:xn</code> <hr/>	<code><key> .multichoices:nn <choices> <code></code> Sets <code><key></code> to act as a multiple choice key, and defines a series <code><choices></code> which are implemented using the <code><code></code> . Inside <code><code></code> , <code>\l_keys_choice_tl</code> will be the name of the choice made, and <code>\l_keys_choice_int</code> will be the position of the choice in the list of <code><choices></code> (indexed from 1). Choices are discussed in detail in section 3.
<hr/> New: 2011-08-21 Updated: 2013-07-10 <hr/>	
<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)
```

New: 2011-08-23

```
\keys_set_known:nn {<module>} {<keyval list>} {<tl>}
```

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>*).

6 Utility functions for keys

```
\keys_if_exist_p:nn ★ \keys_if_exist_p:nn <module> <key>
\keys_if_exist:nnTF ★ \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_p:nnn <module> <key> <choice>
\keys_if_choice_exist:nnnTF ★ \keys_if_choice_exist:nnnTF <module> <key> <choice> {\true code} {\false
code}
```

New: 2011-08-21

Tests if the $\langle choice \rangle$ is defined for the $\langle key \rangle$ within the $\langle module \rangle$, *i.e.* if any code has been defined for $\langle key \rangle / \langle choice \rangle$. The test is **false** if the $\langle key \rangle$ itself is not defined.

```
\keys_show:nn \keys_show:nn {\module} {\key}
```

Shows the function which is used to actually implement a $\langle key \rangle$ for a $\langle module \rangle$.

7 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 XX

The l3file package

File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix `\file_...`, while others are used to work with files on a line by line basis and have prefix `\ior_...` (reading) or `\iow_...` (writing).

It is important to remember that when reading external files \TeX will attempt to locate them both the operating system path and entries in the \TeX file database (most \TeX systems use such a database). Thus the “current path” for \TeX is somewhat broader than that for other programs.

For functions which expect a $\langle file\ name \rangle$ argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Any active characters (as declared in `\l_char_active_seq`) will *not* be expanded, allowing the direct use of these in file names. Spaces are not allowed in file names.

1 File operation functions

<hr/> <code>\g_file_current_name_tl</code> <hr/>	Contains the name of the current \LaTeX 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 \LaTeX run and will be modified each time a file is read using <code>\file_input:n</code> .
<hr/> <code>\file_if_exist:nTF</code> <hr/> <div>Updated: 2012-02-10</div> <hr/>	<code>\file_if_exist:nTF {$\langle file\ name \rangle$} {$\langle true\ code \rangle$} {$\langle false\ code \rangle$}</code> Searches for $\langle file\ name \rangle$ using the current \TeX search path and the additional paths controlled by <code>\file_path_include:n</code> .
<hr/> <code>\file_add_path:nN</code> <hr/> <div>Updated: 2012-02-10</div> <hr/>	<code>\file_add_path:nN {$\langle file\ name \rangle$} $\langle tl\ var \rangle$</code> Searches for $\langle file\ name \rangle$ in the path as detailed for <code>\file_if_exist:nTF</code> , and if found sets the $\langle tl\ var \rangle$ the fully-qualified name of the file, <i>i.e.</i> the path and file name. If the file is not found then the $\langle tl\ var \rangle$ will contain the marker <code>\q_no_value</code> .
<hr/> <code>\file_input:n</code> <hr/> <div>Updated: 2012-02-17</div> <hr/>	<code>\file_input:n {$\langle file\ name \rangle$}</code> Searches for $\langle file\ name \rangle$ in the path as detailed for <code>\file_if_exist:nTF</code> , and if found reads in the file as additional \LaTeX source. All files read are recorded for information and the file name stack is updated by this function. An error will be raised if the file is not found.

<hr/> <code>\file_path_include:n</code> <hr/>	<code>\file_path_include:n {<path>}</code>
Updated: 2012-07-04	Adds $\langle path \rangle$ to the list of those used to search when reading files. The assignment is local. The $\langle path \rangle$ is processed in the same way as a $\langle file\ name \rangle$, <i>i.e.</i> , with <code>x</code> -type expansion except active characters. Spaces are not allowed in the $\langle path \rangle$.

<hr/> <code>\file_path_remove:n</code> <hr/>	<code>\file_path_remove:n {<path>}</code>
Updated: 2012-07-04	Removes $\langle path \rangle$ from the list of those used to search when reading files. The assignment is local. The $\langle path \rangle$ is processed in the same way as a $\langle file\ name \rangle$, <i>i.e.</i> , with <code>x</code> -type expansion except active characters. Spaces are not allowed in the $\langle path \rangle$.

<hr/> <code>\file_list:</code> <hr/>	<code>\file_list:</code>
	This function will list all files loaded using <code>\file_input:n</code> in the log file.

1.1 Input–output stream management

As \TeX is limited to 16 input streams and 16 output streams, direct use of the streams by the programmer is not supported in $\text{\LaTeX}3$. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

<hr/> <code>\ior_new:N</code> <hr/>	<code>\ior_new:N <stream></code>
<code>\ior_new:c</code>	<code>\iow_new:N <stream></code>
<code>\iow_new:N</code>	Globally reserves the name of the $\langle stream \rangle$, either for reading or for writing as appropriate. The $\langle stream \rangle$ is not opened until the appropriate <code>\..._open:Nn</code> function is used. Attempting to use a $\langle stream \rangle$ which has not been opened is an error, and the $\langle stream \rangle$ will behave as the corresponding <code>\c_term_...</code>
<code>\iow_new:c</code>	
New: 2011-09-26	
Updated: 2011-12-27	

<hr/> <code>\ior_open:Nn</code> <hr/>	<code>\ior_open:Nn <stream> {<file name>}</code>
<code>\ior_open:cn</code>	Opens $\langle file\ name \rangle$ for reading using $\langle stream \rangle$ as the control sequence for file access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle file\ name \rangle$ until a <code>\ior_close:N</code> instruction is given or the \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.

T_EXhackers note: This protected macro expands to the T_EX primitive `\read` along with the `to` keyword.

<code>\ior_get_str:NN</code>	<code>\ior_get_str:NN <stream> {token list variable}</code>
------------------------------	---

New: 2012-06-24

Updated: 2012-07-31

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).

T_EXhackers note: This protected macro is a wrapper around the ε -T_EX primitive `\readline`. However, the end-line character normally added by this primitive is not included in the result of `\ior_get_str:NN`.

<hr/>	<hr/>
<code>\ior_if_eof_p:N</code> ★	<code>\ior_if_eof_p:N</code> $\langle stream \rangle$
<code>\ior_if_eof:NTF</code> ★	<code>\ior_if_eof:NTF</code> $\langle stream \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<hr/> Updated: 2012-02-10 <hr/>	Tests if the end of a $\langle stream \rangle$ has been reached during a reading operation. The test will also return a <code>true</code> value if the $\langle stream \rangle$ is not open.

2 Writing to files

<hr/>	<hr/>
<code>\iow_now:Nn</code>	<code>\iow_now:Nn</code> $\langle stream \rangle$ $\{\langle tokens \rangle\}$
<code>\iow_now:Nx</code>	
<hr/> Updated: 2012-06-05 <hr/>	This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ immediately (<i>i.e.</i> the write operation is called on expansion of <code>\iow_now:Nn</code>).

<hr/>	<hr/>
<code>\iow_log:n</code>	<code>\iow_log:n</code> $\{\langle tokens \rangle\}$
<code>\iow_log:x</code>	
<hr/>	This function writes the given $\langle tokens \rangle$ to the log (transcript) file immediately: it is a dedicated version of <code>\iow_now:Nn</code> .

<hr/>	<hr/>
<code>\iow_term:n</code>	<code>\iow_term:n</code> $\{\langle tokens \rangle\}$
<code>\iow_term:x</code>	
<hr/>	This function writes the given $\langle tokens \rangle$ to the terminal file immediately: it is a dedicated version of <code>\iow_now:Nn</code> .

<hr/>	<hr/>
<code>\iow_shipout:Nn</code>	<code>\iow_shipout:Nn</code> $\langle stream \rangle$ $\{\langle tokens \rangle\}$
<code>\iow_shipout:Nx</code>	
<hr/>	This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (<i>i.e.</i> at shipout). The <code>x</code> -type variants expand the $\langle tokens \rangle$ at the point where the function is used but <i>not</i> when the resulting tokens are written to the $\langle stream \rangle$ (<i>cf.</i> <code>\iow_shipout_x:Nn</code>).

<hr/>	<hr/>
<code>\iow_shipout_x:Nn</code>	<code>\iow_shipout_x:Nn</code> $\langle stream \rangle$ $\{\langle tokens \rangle\}$
<code>\iow_shipout_x:Nx</code>	
<hr/> Updated: 2012-09-08 <hr/>	This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (<i>i.e.</i> at shipout). The $\langle tokens \rangle$ are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

T_EXhackers note: This is a wrapper around the T_EX primitive `\write`.

<hr/>	<hr/>
<code>\iow_char:N</code> ★	<code>\iow_char:N</code> $\backslash\langle char \rangle$
<hr/>	Inserts $\langle char \rangle$ into the output stream. Useful when trying to write difficult characters such as <code>%</code> , <code>{</code> , <code>}</code> , <i>etc.</i> 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 $\{\langle text \rangle\} \{\langle run-on text \rangle\} \{\langle set up \rangle\} \langle function \rangle$

New: 2012-06-28

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 **\iow_indent:n** $\{\langle text \rangle\}$

New: 2011-09-21

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.

<hr/> <code>\l_iow_line_count_int</code> <hr/>	
<hr/> <small>New: 2012-06-24</small> <hr/>	
<hr/>	
<hr/> <code>\c_catcode_other_space_tl</code> <hr/>	
<hr/> <small>New: 2011-09-05</small> <hr/>	
<hr/>	

The maximum number of characters in a line to be written by the `\iow_wrap:nnnN` function. This value depends on the T_EX system in use: the standard value is 78, which is typically correct for unmodified T_EXlive and MiK_TE_X systems.

Token list containing one character with category code 12, (“other”), and character code 32 (space).

2.2 Constant input–output streams

<hr/> <code>\c_term_iow</code> <hr/>	
<hr/>	
<hr/> <code>\c_log_iow</code> <code>\c_term_iow</code> <hr/>	

Constant input stream for reading from the terminal. Reading from this stream using `\ior_get:NN` or similar will result in a prompt from T_EX of the form

`<tl>=`

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/>	
<hr/>	
<hr/>	<pre> \if_eof:w <stream> <true code> \else: <false code> \fi: </pre>

Tests if the `<stream>` returns “end of file”, which is true for non-existent files. The `\else:` branch is optional.

T_EXhackers note: This is the T_EX primitive `\ifeof`.

2.4 Internal file functions and variables

<hr/> <code>\l__file_internal_name_iow</code> <hr/>	
<hr/>	
<hr/> <code>\l__file_internal_name_tl</code> <hr/>	

Used to test for the existence of files when opening.

Used to return the full name of a file for internal use.

_file_name_sanitize:n

New: 2012-02-09

_file_name_sanitize:n {<name>} {<tokens>}

Exhaustively-expands the <name> with the exception of any category <active> (catcode 13) tokens, which are not expanded. The list of <active> tokens is taken from `\l_char_active_seq`. The <sanitized name> is then inserted (in braces) after the <tokens>, which should further process the file name. If any spaces are found in the name after expansion, an error is raised.

2.5 Internal input–output functions

_ior_open:N

_ior_open:N

New: 2012-01-23

_ior_open:N <stream> {<file name>}

This function has identical syntax to the public version. However, it does not take precautions against active characters in the <file name>, and it does not attempt to add a <path> to the <file name>: it is therefore intended to be used by higher-level functions which have already fully expanded the <file name> and which need to perform multiple open or close operations. See for example the implementation of `\file_add_path:n`,

Part XXI

The l3fp package: floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. Floating point expressions support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x * y$, division x / y , and parentheses.
- Comparison operators: $x < y$, $x \leq y$, $x > y$, $x \neq y$ etc.
- Boolean logic: negation $!x$, conjunction $x \& \& y$, disjunction $x || y$, ternary operator $x ? y : z$.
- Exponentials: $\exp x$, $\ln x$, x^y .
- Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$.

(*not yet*) Inverse trigonometric functions: $\asin x$, $\acos x$, $\atan x$, $\acot x$, $\asec x$, $\acsc x$.

(*not yet*) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\sech x$, \csch , and $\asinh x$, $\acosh x$, $\atanh x$, $\acoth x$, $asech x$, $acsch x$.

- Extrema: $\max(x, y, \dots)$, $\min(x, y, \dots)$, $\abs(x)$.
- Rounding functions: $\text{round}(x, n)$ round to closest, $\text{round0}(x, n)$ round towards zero, $\text{round}\pm(x, n)$ round towards $\pm\infty$. And (*not yet*) modulo, and “quantize”.
- Constants: `pi`, `deg` (one degree in radians).
- Dimensions, automatically expressed in points, *e.g.*, `pc` is 12.
- Automatic conversion (no need for `\langle type \rangle_use:N`) of integer, dimension, and skip variables to floating points, expressing dimensions in points and ignoring the stretch and shrink components of skips.

Floating point numbers can be given either explicitly (in a form such as `1.234e-34`, or `-.0001`), or as a stored floating point variable, which is automatically replaced by its current value. See section 9.1 for a description of what a floating point is, section 9.2 for details about how an expression is parsed, and section 9.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 7.

An example of use could be the following.

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}$  
= \ExplSyntaxOn \fp_to_decimal:n {\sin 3.5 /2 + 2e-3} $.
```

But in all fairness, this module is mostly meant as an underlying tool for higher-level commands. For example, one could provide a function to typeset nicely the result of floating point computations.

```
\usepackage{xparse, siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnun } { m }
  { \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\calcnun { 2 pi * sin ( 2.3 ^ 5 ) }
```

1 Creating and initialising floating point variables

<hr/> <code>\fp_new:N</code> <code>\fp_new:c</code> <hr/>	<code>\fp_new:N <fp var></code>
Updated: 2012-05-08 <hr/>	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.
<hr/> <code>\fp_const:Nn</code> <code>\fp_const:cn</code> <hr/>	<code>\fp_const:Nn <fp var> {<floating point expression>}</code>
Updated: 2012-05-08 <hr/>	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> .
<hr/> <code>\fp_zero:N</code> <code>\fp_zero:c</code> <code>\fp_gzero:N</code> <code>\fp_gzero:c</code> <hr/>	<code>\fp_zero:N <fp var></code>
Updated: 2012-05-08 <hr/>	Sets the <i><fp var></i> to +0.
<hr/> <code>\fp_zero_new:N</code> <code>\fp_zero_new:c</code> <code>\fp_gzero_new:N</code> <code>\fp_gzero_new:c</code> <hr/>	<code>\fp_zero_new:N <fp var></code>
Updated: 2012-05-08 <hr/>	Ensures that the <i><fp var></i> exists globally by applying <code>\fp_new:N</code> if necessary, then applies <code>\fp_(g)zero:N</code> to leave the <i><fp var></i> set to zero.

2 Setting floating point variables

<hr/> <code>\fp_set:Nn</code> <code>\fp_set:cn</code> <code>\fp_gset:Nn</code> <code>\fp_gset:cn</code> <hr/>	<code>\fp_set:Nn <fp var> {<floating point expression>}</code>
Updated: 2012-05-08 <hr/>	Sets <i><fp var></i> equal to the result of computing the <i><floating point expression></i> .

<code>\fp_set_eq:NN</code>	<code>\fp_set_eq:NN <fp var₁> <fp var₂></code>
<code>\fp_set_eq:(cN Nc cc)</code>	
<code>\fp_gset_eq:NN</code>	Sets the floating point variable <i><fp var₁></i> equal to the current value of <i><fp var₂></i> .
<code>\fp_gset_eq:(cN Nc cc)</code>	

Updated: 2012-05-08

<code>\fp_add:Nn</code>	<code>\fp_add:Nn <fp var> {<floating point expression>}</code>
<code>\fp_add:cn</code>	
<code>\fp_gadd:Nn</code>	Adds the result of computing the <i><floating point expression></i> to the <i><fp var></i> .
<code>\fp_gadd:cn</code>	

Updated: 2012-05-08

<code>\fp_sub:Nn</code>	<code>\fp_sub:Nn <fp var> {<floating point expression>}</code>
<code>\fp_sub:cn</code>	
<code>\fp_gsub:Nn</code>	Subtracts the result of computing the <i><floating point expression></i> from the <i><fp var></i> .
<code>\fp_gsub:cn</code>	

Updated: 2012-05-08

3 Using floating point numbers

<code>\fp_eval:n</code> ★	<code>\fp_eval:n {<floating point expression>}</code>
New: 2012-05-08	
Updated: 2012-07-08	Evaluates the <i><floating point expression></i> and expresses the result as a decimal number with 16 significant figures and no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an “invalid operation” exception. This function is identical to <code>\fp_to_decimal:n</code> .

<code>\fp_to_decimal:N</code> ★	<code>\fp_to_decimal:N <fp var></code>
<code>\fp_to_decimal:(c n)</code> ★	<code>\fp_to_decimal:n {<floating point expression>}</code>
New: 2012-05-08	
Updated: 2012-07-08	Evaluates the <i><floating point expression></i> and expresses the result as a decimal number with 16 significant figures and no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an “invalid operation” exception.

<code>\fp_to_dim:N</code> ★	<code>\fp_to_dim:N <fp var></code>
<code>\fp_to_dim:(c n)</code> ★	<code>\fp_to_dim:n {<floating point expression>}</code>
Updated: 2012-07-08	Evaluates the <i><floating point expression></i> and expresses the result as a dimension (in pt) suitable for use in dimension expressions. The output is identical to <code>\fp_to_decimal:n</code> , with an additional trailing pt . In particular, the result may be outside the range $[-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]$ of valid T _E X dimensions, leading to overflow errors if used as a dimension. The values $\pm\infty$ and NaN trigger an “invalid operation” exception.

<code>\fp_to_int:N</code> ★	<code>\fp_to_int:N</code> $\langle fp\ var \rangle$
<code>\fp_to_int:(c n)</code> ★	<code>\fp_to_int:n</code> $\{\langle floating\ point\ expression \rangle\}$
Updated: 2012-07-08	Evaluates the $\langle floating\ point\ expression \rangle$, and rounds the result to the closest integer, with ties rounded to an even integer. The result may be outside the range $[-2^{31} + 1, 2^{31} - 1]$ of valid \TeX integers, triggering \TeX errors if used in an integer expression. The values $\pm\infty$ and NaN trigger an “invalid operation” exception.

<code>\fp_to_scientific:N</code> ★	<code>\fp_to_scientific:N</code> $\langle fp\ var \rangle$
<code>\fp_to_scientific:(c n)</code> ★	<code>\fp_to_scientific:n</code> $\{\langle floating\ point\ expression \rangle\}$
New: 2012-05-08 Updated: 2012-07-08	Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result in scientific notation with 16 significant figures:

$\langle optional\ - \rangle \langle digit \rangle . \langle 15\ digits \rangle e \langle optional\ sign \rangle \langle exponent \rangle$

The leading $\langle digit \rangle$ is non-zero except in the case of ± 0 . The values $\pm\infty$ and NaN trigger an “invalid operation” exception.

<code>\fp_to_tl:N</code> ★	<code>\fp_to_tl:N</code> $\langle fp\ var \rangle$
<code>\fp_to_tl:(c n)</code> ★	<code>\fp_to_tl:n</code> $\{\langle floating\ point\ expression \rangle\}$
Updated: 2012-07-08	Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result in (almost) the shortest possible form. Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed (see <code>\fp_to_scientific:n</code>). Numbers in the range $[10^{-3}, 10^{16})$ are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see <code>\fp_to_decimal:n</code>). Negative numbers start with $-$. The special values ± 0 , $\pm\text{inf}$ 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</code> $\langle fp\ var \rangle$
<code>\fp_use:c</code> ★	Inserts the value of the $\langle fp\ var \rangle$ into the input stream as a decimal number with 16 significant figures and no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an “invalid operation” exception. This function is identical to <code>\fp_to_decimal:N</code> .
Updated: 2012-07-08	

4 Floating point conditionals

<code>\fp_if_exist_p:N</code> ★	<code>\fp_if_exist_p:N</code> $\langle fp\ var \rangle$
<code>\fp_if_exist_p:c</code> ★	<code>\fp_if_exist:NTF</code> $\langle fp\ var \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\fp_if_exist:NTF</code> ★	Tests whether the $\langle fp\ var \rangle$ is currently defined. This does not check that the $\langle fp\ var \rangle$ really is a floating point variable.
<code>\fp_if_exist:cTF</code> ★	
Updated: 2012-05-08	

<code>\fp_compare_p:nNn</code> ★	<code>\fp_compare_p:nNn {<fpexpr₁>} <relation> {<fpexpr₂>}</code>
<code>\fp_compare_p:n</code> ★	<code>\fp_compare:nNnTF {<fpexpr₁>} <relation> {<fpexpr₂>} {<true code>} {<false code>}</code>
<code>\fp_compare:nNnTF</code> ★	<code>\fp_compare_p:n {<fpexpr₁>} <relation> {<fpexpr₂>}</code>
<code>\fp_compare:nTF</code> ★	<code>\fp_compare:nTF {<fpexpr₁>} <relation> {<fpexpr₂>} {<true code>} {<false code>}</code>

Updated: 2012-05-08

Compares the $\langle fpexpr_1 \rangle$ and the $\langle fpexpr_2 \rangle$, and returns **true** if the $\langle relation \rangle$ is obeyed. Two floating point numbers x and y may obey four mutually exclusive relations: $x \langle y, x=y, x \rangle y$, or x and y are not ordered. The latter case occurs exactly when one of the operands is NaN, and this relations is denoted by the symbol $?$. The **nNn** functions support the $\langle relations \rangle$ $<$, $=$, $>$, and $?$. The **n** functions support as a $\langle relation \rangle$ any non-empty string of those four symbols, plus optional leading $!$ (which negate the $\langle relation \rangle$), with the restriction that the $\langle relation \rangle$ may not start with $?$. Common choices of $\langle relation \rangle$ include $>=$ (greater or equal), $!=$ (not equal), $!?$ (comparable). Note that a NaN is distinct from any value, even another NaN, hence $x = x$ is not true for a NaN. Since a NaN is not comparable to any floating point, to test if a value is NaN, one can use the following, where 0 is an arbitrary floating point.

```
\fp_compare:nNnTF { <value> } ? { 0 }
{ } % <value> is nan
{ } % <value> is not nan
```

5 Floating point expression loops

<code>\fp_do_until:nNnn</code> ☆	<code>\fp_do_until:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
----------------------------------	--

New: 2012-08-16

Places the $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle floating point expressions \rangle$ as described for `\fp_compare:nNnTF`. If the test is **false** then the $\langle code \rangle$ will be inserted into the input stream again and a loop will occur until the $\langle relation \rangle$ is **true**.

<code>\fp_do_while:nNnn</code> ☆	<code>\fp_do_while:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
----------------------------------	--

New: 2012-08-16

Places the $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle floating point expressions \rangle$ as described for `\fp_compare:nNnTF`. If the test is **true** then the $\langle code \rangle$ will be inserted into the input stream again and a loop will occur until the $\langle relation \rangle$ is **false**.

<code>\fp_until_do:nNnn</code> ☆	<code>\fp_until_do:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
----------------------------------	--

New: 2012-08-16

Evaluates the relationship between the two $\langle floating point expressions \rangle$ as described for `\fp_compare:nNnTF`, and then places the $\langle code \rangle$ in the input stream if the $\langle relation \rangle$ is **false**. After the $\langle code \rangle$ has been processed by T_EX the test will be repeated, and a loop will occur until the test is **true**.

<hr/> <code>\fp_while_do:nNnn</code> ☆ <hr/>	<code>\fp_while_do:nNnn {<fpexpr1>} <relation> {<fpexpr2>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .
<hr/> <code>\fp_do_until:nn</code> ☆ <hr/>	<code>\fp_do_until:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is false then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is true .
<hr/> <code>\fp_do_while:nn</code> ☆ <hr/>	<code>\fp_do_while:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is true then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<hr/> <code>\fp_until_do:nn</code> ☆ <hr/>	<code>\fp_until_do:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<hr/> <code>\fp_while_do:nn</code> ☆ <hr/>	<code>\fp_while_do:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .

6 Some useful constants, and scratch variables

<hr/> <code>\c_zero_fp</code> <code>\c_minus_zero_fp</code> <hr/>	Zero, with either sign.
New: 2012-05-08	
<hr/> <code>\c_one_fp</code> <hr/>	One as an fp: useful for comparisons in some places.
New: 2012-05-08	

<hr/> <code>\c_inf_fp</code> <code>\c_minus_inf_fp</code> <hr/>	Infinity, with either sign. These can be input directly in a floating point expression as <code>inf</code> and <code>-inf</code> .
<hr/> New: 2012-05-08 <hr/>	
<hr/> <code>\c_e_fp</code> <hr/>	The value of the base of the natural logarithm, $e = \exp(1)$.
<hr/> Updated: 2012-05-08 <hr/>	
<hr/> <code>\c_pi_fp</code> <hr/>	The value of π . This can be input directly in a floating point expression as <code>pi</code> . The value is rounded in a slightly odd way, to ensure for instance that <code>sin(pi)</code> yields an exact 0.
<hr/> Updated: 2012-05-08 <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, suitable to be used for trigonometric functions. Within floating point expressions, this can be accessed as <code>deg</code> . Note that <code>180 deg = pi</code> exactly.
<hr/> New: 2012-05-08 <hr/>	
<hr/> <code>\l_tmpa_fp</code> <code>\l_tmpb_fp</code> <hr/>	Scratch floating points for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_fp</code> <code>\g_tmpb_fp</code> <hr/>	Scratch floating points for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

7 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

“Exceptions” may occur when performing some floating point operations, such as `0 / 0`, or `10 ** 1e9999`. The IEEE standard defines 5 types of exceptions.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm\infty$.
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ± 0 .
- *Invalid operation* occurs for operations with no defined outcome, for instance `0/0`, or `sin(∞)`, and almost any operation involving a NaN. This normally results in a NaN, except for conversion functions whose target type does not have a notion of NaN (e.g., `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating e.g., `ln(0)` or `cot(0)`. This results in $\pm\infty$.

- *Inexact* occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in L^AT_EX3.

To each exception is associated a “flag”, which can be either *on* or *off*. By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions only raise the corresponding flag. The state of the flag can be tested and modified. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and turn the flag on, or only turn the flag on, or do nothing at all.

<code>\fp_if_flag_on_p:n</code> ★	<code>\fp_if_flag_on_p:n {<exception>}</code>
<code>\fp_if_flag_on:nTF</code> ★	<code>\fp_if_flag_on:nTF {<exception>} {<true code>} {<false code>}</code>
New: 2012-08-08	Tests if the flag for the <i><exception></i> is on, which normally means the given <i><exception></i> has occurred. <i>This function is experimental, and may be altered or removed.</i>
<code>\fp_flag_off:n</code>	<code>\fp_flag_off:n {<exception>}</code>
New: 2012-08-08	Locally turns off the flag which indicates whether the <i><exception></i> has occurred. <i>This function is experimental, and may be altered or removed.</i>
<code>\fp_flag_on:n</code> ★	<code>\fp_flag_on:n {<exception>}</code>
New: 2012-08-08	Locally turns on the flag to indicate (or pretend) that the <i><exception></i> has occurred. Note that this function is expandable: it is used internally by l3fp to signal when exceptions do occur. <i>This function is experimental, and may be altered or removed.</i>
<code>\fp_trap:nn</code>	<code>\fp_trap:nn {<exception>} {<trap type>}</code>
New: 2012-07-19 Updated: 2012-08-08	All occurrences of the <i><exception></i> (<i>invalid_operation</i> , <i>division_by_zero</i> , <i>overflow</i> , or <i>underflow</i>) within the current group are treated as <i><trap type></i> , which can be <ul style="list-style-type: none"> • none: the <i><exception></i> will be entirely ignored, and leave no trace; • flag: the <i><exception></i> will turn the corresponding flag on when it occurs; • error: additionally, the <i><exception></i> will halt the T_EX run and display some information about the current operation in the terminal.

This function is experimental, and may be altered or removed.

8 Viewing floating points

<code>\fp_show:N</code>	<code>\fp_show:N {<fp var>}</code>
<code>\fp_show:(c n)</code>	<code>\fp_show:n {<floating point expression>}</code>
New: 2012-05-08 Updated: 2012-08-14	Evaluates the <i><floating point expression></i> and displays the result in the terminal.

9 Floating point expressions

9.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm 0.d_1d_2 \dots d_{16} \cdot 10^n$, a normal floating point number, with $d_i \in [0, 9]$, $d_1 \neq 0$, and $|n| \leq 10000$;
- ± 0 , zero, with a given sign;
- $\pm \infty$, infinity, with a given sign;
- **NaN**, is “not a number”, and can be either quiet or signalling (*not yet*: this distinction is currently unsupported);

(*not yet*) subnormal numbers $\pm 0.d_1d_2 \dots d_{16} \cdot 10^{-10000}$ with $d_1 = 0$.

Normal floating point numbers are stored in base 10, with 16 significant figures.

On input, a normal floating point number consists of:

- $\langle sign \rangle$: a possibly empty string of + and - characters;
- $\langle significand \rangle$: a non-empty string of digits together with zero or one dot;
- $\langle exponent \rangle$ optionally: the character **e**, followed by a possibly empty string of + and - tokens, and a non-empty string of digits.

The sign of the resulting number is + if $\langle sign \rangle$ contains an even number of -, and - otherwise, hence, an empty $\langle sign \rangle$ denotes a non-negative input. The stored significand is obtained from $\langle significand \rangle$ by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input $\langle significand \rangle$ has at most 16 digits. The stored $\langle exponent \rangle$ is obtained by combining the input $\langle exponent \rangle$ (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting $\langle exponent \rangle$ is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by $\pm \infty$), or an underflow (resulting in ± 0).

The result is thus ± 0 if and only if $\langle significand \rangle$ contains no non-zero digit (*i.e.*, consists only in 0 characters, and an optional . character), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to +0, but that is not guaranteed to remain true.

Special numbers are input as follows:

- **inf** represents $+\infty$, and can be preceded by any $\langle sign \rangle$, yielding $\pm \infty$ as appropriate.
- **nan** represents a (quiet) non-number. It can be preceded by any sign, but that will be ignored.
- Any unrecognizable string triggers an error, and produces a **NaN**.

Note that `e-1` is not a representation of 10^{-1} , because it could be mistaken with the difference of “e” and 1. This is consistent with several other programming languages. However, in order to avoid confusions, `e-1` is not considered to be this difference either. To input the base of natural logarithms, use `exp(1)` or `\c_e_fp`.

9.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Implicit multiplication by juxtaposition (`2pi`, *etc*).
- Function calls (`sin`, `ln`, *etc*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Binary `*`, `/` and `%`.
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, *etc*.
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).

The precedence of operations can be overridden using parentheses. In particular, those precedences imply that

$$\begin{aligned}\text{sin2pi} &= \sin(2\pi) = 0, \\ 2^{2\text{max}(3,4)} &= 2^{2 \max(3,4)} = 256.\end{aligned}$$

Functions are called on the value of their argument, contrarily to \TeX macros.

9.3 Operations

We now present the various operations allowed in floating point expressions, from the lowest precedence to the highest. When used as a truth value, a floating point expression is `false` if it is ± 0 , and `true` otherwise, including when it is `NaN`.

```
?: \fp_eval:n { <operand1> ? <operand2> : <operand3> }
```

The ternary operator `?:` results in $\langle operand_2 \rangle$ if $\langle operand_1 \rangle$ is true, and $\langle operand_3 \rangle$ if it is false (equal to ± 0). All three $\langle operands \rangle$ are evaluated in all cases. The operator is right associative, hence

```
\fp_eval:n
{
  1 + 3 > 4 ? 1 :
  2 + 4 > 5 ? 2 :
  3 + 5 > 6 ? 3 : 4
}
```

first tests whether $1 + 3 > 4$; since this isn't true, the branch following `:` is taken, and $2 + 4 > 5$ is compared; since this is true, the branch before `:` is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

```
TWOBARS \fp_eval:n { <operand1> || <operand2> }
```

If $\langle operand_1 \rangle$ is true (non-zero), use that value, otherwise the value of $\langle operand_2 \rangle$. Both $\langle operands \rangle$ are evaluated in all cases.

```
&& \fp_eval:n { <operand1> && <operand2> }
```

If $\langle operand_1 \rangle$ is false (equal to ± 0), use that value, otherwise the value of $\langle operand_2 \rangle$. Both $\langle operands \rangle$ are evaluated in all cases.

```
< \fp_eval:n { <operand1> <comparison> <operand2> }
```

The $\langle comparison \rangle$ consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by `!`. It may not start with `?`. This evaluates to `+1` if the $\langle comparison \rangle$ between the $\langle operand_1 \rangle$ and $\langle operand_2 \rangle$ is true, and `+0` otherwise.

```
+ \fp_eval:n { <operand1> + <operand2> }
```

```
- \fp_eval:n { <operand1> - <operand2> }
```

Computes the sum or the difference of its two $\langle operands \rangle$. The “invalid operation” exception occurs for $\infty - \infty$. “Underflow” and “overflow” occur when appropriate.

```
* \fp_eval:n { <operand1> * <operand2> }
```

```
/ \fp_eval:n { <operand1> / <operand2> }
```

Computes the product or the ratio of its two $\langle operands \rangle$. The “invalid operation” exception occurs for ∞/∞ , $0/0$, or $0 * \infty$. “Division by zero” occurs when dividing a finite non-zero number by ± 0 . “Underflow” and “overflow” occur when appropriate.

```

+ \fp_eval:n { + <operand> }
- \fp_eval:n { - <operand> }
! \fp_eval:n { ! <operand> }

```

The unary + does nothing, the unary - changes the sign of the $\langle operand \rangle$, and ! $\langle operand \rangle$ evaluates to 1 if $\langle operand \rangle$ is false and 0 otherwise (this is the `not` boolean function). Those operations never raise exceptions.

```

** \fp_eval:n { <operand1> ** <operand2> }
^ \fp_eval:n { <operand1> ^ <operand2> }

```

Raises $\langle operand_1 \rangle$ to the power $\langle operand_2 \rangle$. This operation is right associative, hence `2 ** 2 ** 3` equals $2^{2^3} = 256$. The “invalid operation” exception occurs if $\langle operand_1 \rangle$ is negative or -0 , and $\langle operand_2 \rangle$ is not an integer, unless the result is zero (in that case, the sign is chosen arbitrarily to be $+0$). “Division by zero” occurs when raising ± 0 to a strictly negative power. “Underflow” and “overflow” occur when appropriate.

```

abs \fp_eval:n { abs( <fpexpr> ) }

```

Computes the absolute value of the $\langle fpexpr \rangle$. This function does not raise any exception beyond those raised when computing its operand $\langle fpexpr \rangle$. See also `\fp_abs:n`.

```

exp \fp_eval:n { exp( <fpexpr> ) }

```

Computes the exponential of the $\langle fpexpr \rangle$. “Underflow” and “overflow” occur when appropriate.

```

ln \fp_eval:n { ln( <fpexpr> ) }

```

Computes the natural logarithm of the $\langle fpexpr \rangle$. Negative numbers have no (real) logarithm, hence the “invalid operation” is raised in that case, including for $\ln(-0)$. “Division by zero” occurs when evaluating $\ln(+0) = -\infty$. “Underflow” and “overflow” occur when appropriate.

```

max \fp_eval:n { max( <fpexpr1> , <fpexpr2> , ... ) }
min \fp_eval:n { min( <fpexpr1> , <fpexpr2> , ... ) }

```

Evaluates each $\langle fpexpr \rangle$ and computes the largest (smallest) of those. If any of the $\langle fpexpr \rangle$ is a NaN, the result is NaN. Those operations do not raise exceptions.

<code>round</code>	<code>\fp_eval:n { round <option> (<fpexpr>) }</code>
<code>round0</code>	<code>\fp_eval:n { round <option> (<fpexpr₁> , <fpexpr₂>) }</code>
<code>round+</code>	Rounds $\langle fpexpr_1 \rangle$ to $\langle fpexpr_2 \rangle$ places. When $\langle fpexpr_2 \rangle$ is omitted, it is assumed to be 0, <i>i.e.</i> , $\langle fpexpr_1 \rangle$ is rounded to an integer. The $\langle option \rangle$ controls the rounding direction:
<code>round-</code>	

- by default, the operation rounds to the closest allowed number (rounding ties to even);
- with 0, the operation rounds towards 0, *i.e.*, truncates;
- with +, the operation rounds towards $+\infty$;
- with -, the operation rounds towards $-\infty$.

If $\langle fpexpr_2 \rangle$ does not yield an integer less than 10^8 in absolute value, then an “invalid operation” exception is raised. “Overflow” may occur if the result is infinite (this cannot happen unless $\langle fpexpr_2 \rangle < -9984$).

<code>sin</code>	<code>\fp_eval:n { sin(<fpexpr>) }</code>
<code>cos</code>	<code>\fp_eval:n { cos(<fpexpr>) }</code>
<code>tan</code>	<code>\fp_eval:n { tan(<fpexpr>) }</code>
<code>cot</code>	<code>\fp_eval:n { cot(<fpexpr>) }</code>
<code>csc</code>	<code>\fp_eval:n { csc(<fpexpr>) }</code>
<code>sec</code>	<code>\fp_eval:n { sec(<fpexpr>) }</code>

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$. 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>inf</code>	The special values $+\infty$, $-\infty$, and NaN are represented as <code>inf</code> , <code>-inf</code> and <code>nan</code> (see <code>\c_-nan</code> , <code>\c_minus_inf_fp</code> and <code>\c_nan_fp</code>).
<code>nan</code>	

<code>pi</code>	The value of π (see <code>\c_pi_fp</code>).
-----------------	--

<code>deg</code>	The value of 1° in radians (see <code>\c_one_degree_fp</code>).
------------------	---

<hr/>	Those units of measurement are equal to their values in <code>pt</code> , namely
<code>em</code>	
<code>ex</code>	
<code>in</code>	$1\text{in} = 72.27\text{pt}$
<code>pt</code>	$1\text{pt} = 1\text{pt}$
<code>pc</code>	
<code>cm</code>	$1\text{pc} = 12\text{pt}$
<code>mm</code>	
<code>dd</code>	$1\text{cm} = \frac{1}{2.54}\text{in} = 28.45275590551181\text{pt}$
<code>cc</code>	
<code>nd</code>	$1\text{mm} = \frac{1}{25.4}\text{in} = 2.845275590551181\text{pt}$
<code>nc</code>	
<code>bp</code>	$1\text{dd} = 0.376065\text{mm} = 1.07000856496063\text{pt}$
<code>sp</code>	$1\text{cc} = 12\text{dd} = 12.84010277952756\text{pt}$
<hr/>	$1\text{nd} = 0.375\text{mm} = 1.066978346456693\text{pt}$
	$1\text{nc} = 12\text{nd} = 12.80374015748031\text{pt}$
	$1\text{bp} = \frac{1}{72}\text{in} = 1.00375\text{pt}$
	$1\text{sp} = 2^{-16}\text{pt} = 1.52587890625e - 5\text{pt}.$

The values of the (font-dependent) units `em` and `ex` are gathered from \TeX when the surrounding floating point expression is evaluated.

<hr/>	
<code>true</code>	Other names for 1 and +0.
<code>false</code>	
<hr/>	
<code>\dim_to_fp:n</code> ★	<code>\dim_to_fp:n {⟨dimexpr⟩}</code>
New: 2012-05-08	Expands to an internal floating point number equal to the value of the <i>⟨dimexpr⟩</i> in <code>pt</code> . 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.
<hr/>	
<code>\fp_abs:n</code> ★	<code>\fp_abs:n {⟨floating point expression⟩}</code>
New: 2012-05-14 Updated: 2012-07-08	Evaluates the <i>⟨floating point expression⟩</i> as described for <code>\fp_eval:n</code> and leaves the absolute value of the result in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, <code>abs()</code> can be used.
<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 resulting larger (<code>max</code>) or smaller (<code>min</code>) value in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, <code>max()</code> and <code>min()</code> can be used.
New: 2012-09-26	

10 Disclaimer and roadmap

The package may break down if:

- the escape character is either a digit, or an underscore,
- the `\uccodes` are changed: the test for whether a character is a letter actually tests if the upper-case code of the character is between A and Z.

The following need to be done. I'll try to time-order the items.

- Decide what exponent range to consider.
- Change the internal representation of fp, by replacing braced groups of 4 digits by delimited arguments. Also consider changing the fp structure a bit to allow using `\pdfTEX_strcmp:D` to compare (not in LuaTeX: it is too slow)?
- Modulo and remainder, and rounding functions `quantize`, `quantize0`, `quantize+`, `quantize-`, `quantize=`, `round=`. Should the modulo also be provided as (catcode 12) `%`?
- `\fp_format:nn {<fpexpr>} {<format>}`, but what should `<format>` be? More general pretty printing?
- Add `and`, `or`, `xor`? Perhaps under the names `all`, `any`, and `xor`?
- Add `csc` and `sec`.
- Add `log(x, b)` for logarithm of x in base b .
- `hypot` (Euclidean length) and `atan(x, y) = atan(x/y)`, also called `atan2` in other math packages. Cartesian-to-polar transform. Other inverse trigonometric functions `acos`, `asin`, `atan` (one and two arguments). Also `asec`, `acsc`?
- 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.
- Parse $-3 < -2 < -1$ as it should, not $(-3 < -2) < -1$.
- 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?

`Pgfmath` also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs. (Exclamation points mark important bugs.)

- ! Some functions are not monotonic when they should. For instance, $\sin(1 - 10^{-16})$ is wrongly greater than $\sin(1)$.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- `round` should accept any integer as its second argument.
- Logarithms of numbers very close to 1 are inaccurate.
- `tan` and `cot` give very slightly wrong results for arguments near 10^{-8} .
- When rounding towards $-\infty$, `\dim_to_fp:n {0pt}` should return -0 , not $+0$.
- The result of $(\pm 0) + (\pm 0)$ should depend on the rounding mode.
- `0e9999999999` gives a T_EX “number too large” error.
- Conversion to integers with `\fp_to_int:n` does not check for overflow.
- Subnormals are not implemented.
- `max(-inf)` will lose any information attached to this `-inf`.
- The overflow trap receives the wrong argument in `l3fp-expo` (see `exp(1e5678)` in `m3fp-traps001`).

Possible optimizations/improvements.

- Optimize argument reduction for trigonometric functions: we don’t need 6×4 digits here, only 4×4 .
- 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 , using methods similar to `__fp_fixed_div_to_float:ww`. 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?

Part XXII

The l3luatex package

LuaTeX-specific functions

1 Breaking out to Lua

The LuaTeX engine provides access to the Lua programming language, and with it access to the “internals” of T_EX. In order to use this within the framework provided here, a family of functions is available. When used with pdfT_EX or XeT_EX these will raise an error: use `\luatex_if_engine:T` to avoid this. Details of coding the LuaTeX engine are detailed in the LuaTeX manual.

`\lua_now:n` ★ `\lua_now:n {⟨token list⟩}`

`\lua_now:x` ★

Updated: 2012-08-02

The `⟨token list⟩` is first tokenized by T_EX, which will include converting line ends to spaces in the usual T_EX manner and which respects currently-applicable T_EX category codes. The resulting `⟨Lua input⟩` is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the `⟨Lua input⟩` immediately, and in an expandable manner.

`\lua_now_x:n` ★ `\lua_now_x:n {⟨token list⟩}`

`\lua_now_x:x` ★

New: 2012-08-02

The `⟨token list⟩` is first tokenized and expanded by T_EX, which will include converting line ends to spaces in the usual T_EX manner and which respects currently-applicable T_EX category codes. The resulting `⟨Lua input⟩` is passed to the Lua interpreter for processing. Each `\lua_now_x:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the `⟨Lua input⟩` immediately, and in an expandable manner.

T_EXhackers note: `\lua_now_x:n` is the LuaTeX primitive `\directlua` renamed.

`\lua_shipout:n` `\lua_shipout:n {⟨token list⟩}`

`\lua_shipout:x`

The `⟨token list⟩` is first tokenized by T_EX, which will include converting line ends to spaces in the usual T_EX manner and which respects currently-applicable T_EX category codes. The resulting `⟨Lua input⟩` is passed to the Lua interpreter when the current page is finalised (*i.e.* at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the `⟨Lua input⟩` during the page-building routine: no T_EX expansion of the `⟨Lua input⟩` will occur at this stage.

T_EXhackers note: At a T_EX level, the `⟨Lua input⟩` is stored as a “whatsit”.

<code>\lua_shipout_x:n</code> <code>\lua_shipout_x:x</code>	<code>\lua_shipout:n {⟨token list⟩}</code> <p>The <i>⟨token list⟩</i> is first tokenized by \TeX, which will include converting line ends to spaces in the usual \TeX manner and which respects currently-applicable \TeX category codes. The resulting <i>⟨Lua input⟩</i> is passed to the Lua interpreter when the current page is finalised (<i>i.e.</i> at shipout). Each <code>\lua_shipout:n</code> block is treated by Lua as a separate chunk. The Lua interpreter will execute the <i>⟨Lua input⟩</i> during the page-building routine: the <i>⟨Lua input⟩</i> is expanded during this process in addition to any expansion when the argument was read. This makes these functions suitable for including material finalised during the page building process (such as the page number).</p>
--	--

\TeX hackers note: `\lua_shipout_x:n` is the Lua \TeX primitive `\latelua` named using the \LaTeX 3 scheme.

At a \TeX level, the *⟨Lua input⟩* is stored as a “whatsit”.

2 Category code tables

As well as providing methods to break out into Lua, there are places where additional \LaTeX 3 functions are provided by the Lua \TeX engine. In particular, Lua \TeX provides category code tables. These can be used to ensure that a set of category codes are in force in a more robust way than is possible with other engines. These are therefore used by `\ExplSyntaxOn` and `\ExplSyntaxOff` when using the Lua \TeX engine.

<code>\cctab_new:N</code>	<code>\cctab_new:N ⟨category code table⟩</code> <p>Creates a new category code table, initially with the codes as used by <code>\iniTeX</code>.</p>
---------------------------	--

<code>\cctab_gset:Nn</code>	<code>\cctab_gset:Nn ⟨category code table⟩ {⟨category code set up⟩}</code> <p>Sets the <i>⟨category code table⟩</i> to apply the category codes which apply when the prevailing régime is modified by the <i>⟨category code set up⟩</i>. Thus within a standard code block the starting point will be the code applied by <code>\c_code_cctab</code>. The assignment of the table is global: the underlying primitive does not respect grouping.</p>
-----------------------------	---

<code>\cctab_begin:N</code>	<code>\cctab_begin:N ⟨category code table⟩</code> <p>Switches the category codes in force to those stored in the <i>⟨category code table⟩</i>. The prevailing codes before the function is called are added to a stack, for use with <code>\cctab_end:</code>.</p>
-----------------------------	---

<code>\cctab_end:</code>	<code>\cctab_end:</code> <p>Ends the scope of a <i>⟨category code table⟩</i> started using <code>\cctab_begin:N</code>, retuning the codes to those in force before the matching <code>\cctab_begin:N</code> was used.</p>
--------------------------	---

<code>\c_code_cctab</code>	<p>Category code table for the code environment. This does not include setting the behaviour of the line-end character, which is only altered by <code>\ExplSyntaxOn</code>.</p>
----------------------------	--

<u><code>\c_document_cctab</code></u>	Category code table for a standard L ^A T _E X document. This does not include setting the behaviour of the line-end character, which is only altered by <code>\ExplSyntaxOff</code> .
<u><code>\c_initex_cctab</code></u>	Category code table as set up by <code>iniT_EX</code> .
<u><code>\c_other_cctab</code></u>	Category code table where all characters have category code 12 (other).
<u><code>\c_str_cctab</code></u>	Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space).

Part XXIII

The l3candidates package

Experimental additions to l3kernel

This module provides a space in which functions can be added to l3kernel (expl3) while still being experimental. As such, the functions here may not remain in their current form, or indeed at all, in l3kernel in the future. In contrast to the material in l3experimental, the functions here are all *small* additions to the kernel. We encourage programmers to test them out and report back on the LaTeX-L mailing list.

1 Additions to l3basics

`\cs_if_exist_use:NTF` ★
`\cs_if_exist_use:cTF` ★

`\cs_if_exist_use:NTF` $\langle control\ sequence \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

If the $\langle control\ sequence \rangle$ exists, leave it in the input stream, followed by the $\langle true\ code \rangle$ (unbraced). Otherwise, leave the $\langle false \rangle$ code in the input stream. For example,

```
\cs_set:Npn \mypkg_use_character:N #1
{ \cs_if_exist_use:cF { mypkg_#1:n } { \mypkg_default:N #1 } }
```

calls the function `\mypkg_#1:n` if it exists, and falls back to a default action otherwise. This could also be done (more slowly) using `\str_case_x:nnn`.

T_EXhackers note: The `c` variants do not introduce the $\langle control\ sequence \rangle$ in the hash table if it is not there.

2 Additions to l3box

2.1 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in T_EX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.

`\box_resize:Nnn`
`\box_resize:cnn`

`\box_resize:Nnn` $\langle box \rangle$ $\{\langle x-size \rangle\}$ $\{\langle y-size \rangle\}$

Resize the $\langle box \rangle$ to $\langle x-size \rangle$ horizontally and $\langle y-size \rangle$ vertically (both of the sizes are dimension expressions). The $\langle y-size \rangle$ is the vertical size (height plus depth) of the box. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. The resizing applies within the current T_EX group level.

<code>\box_resize_to_ht_plus_dp:Nn</code>	<code>\box_resize_to_ht_plus_dp:Nn <box> {<y-size>}</code>
<code>\box_resize_to_ht_plus_dp:cn</code>	

Resize the $\langle box \rangle$ to $\langle y-size \rangle$ vertically, scaling the horizontal size by the same amount ($\langle y-size \rangle$ is a dimension expression). The $\langle y-size \rangle$ is the vertical size (height plus depth) of the box. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative size will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. The resizing applies within the current \TeX group level.

<code>\box_resize_to_wd:Nn</code>	<code>\box_resize_to_wd:Nn <box> {<x-size>}</code>
<code>\box_resize_to_wd:cn</code>	

Resize the $\langle box \rangle$ to $\langle x-size \rangle$ horizontally, scaling the vertical size by the same amount ($\langle x-size \rangle$ is a dimension expression). The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative size will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. The resizing applies within the current \TeX group level.

<code>\box_rotate:Nn</code>	<code>\box_rotate:Nn <box> {<angle>}</code>
<code>\box_rotate:cn</code>	

Rotates the $\langle box \rangle$ by $\langle angle \rangle$ (in degrees) anti-clockwise about its reference point. The reference point of the updated box will be moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the rotation is applied. The rotation applies within the current \TeX group level.

<code>\box_scale:Nnn</code>	<code>\box_scale:Nnn <box> {<x-scale>} {<y-scale>}</code>
<code>\box_scale:cnn</code>	

Scales the $\langle box \rangle$ by factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the scaling is applied. Negative scalings will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. The scaling applies within the current \TeX group level.

2.2 Viewing part of a box

<code>\box_clip:N</code>	<code>\box_clip:N <box></code>
<code>\box_clip:c</code>	

Clips the $\langle box \rangle$ in the output so that only material inside the bounding box is displayed in the output. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the clipping is applied. The clipping applies within the current \TeX group level.

These functions require the $\text{\LaTeX}3$ 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 places in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

<code>\box_trim:Nnnnn</code>	<code>\box_trim:Nnnnn <box> {\left} {\bottom} {\right} {\top}</code>
<code>\box_trim:cnnnn</code>	

Adjusts the bounding box of the `<box>` `<left>` is removed from the left-hand edge of the bounding box, `<right>` from the right-hand edge and so fourth. All adjustments are *<dimension expressions>*. Material output of the bounding box will still be displayed in the output unless `\box_clip:N` is subsequently applied. The updated `<box>` will be an hbox, irrespective of the nature of the `<box>` before the trim operation is applied. The adjustment applies within the current T_EX group level. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

<code>\box_viewport:Nnnnn</code>	<code>\box_viewport:Nnnnn <box> {\llx} {\lly} {\urx} {\ury}</code>
<code>\box_viewport:cnnnn</code>	

Adjusts the bounding box of the `<box>` such that it has lower-left co-ordinates (`<llx>`, `<lly>`) and upper-right co-ordinates (`<urx>`, `<ury>`). All four co-ordinate positions are *<dimension expressions>*. Material output of the bounding box will still be displayed in the output unless `\box_clip:N` is subsequently applied. The updated `<box>` will be an hbox, irrespective of the nature of the `<box>` before the viewport operation is applied. The adjustment applies within the current T_EX group level.

2.3 Internal variables

<code>\l__box_angle_fp</code>	The angle through which a box is rotated by <code>\box_rotate:Nn</code> , given in degrees counter-clockwise. This value is required by the underlying driver code in <code>l3driver</code> to carry out the driver-dependent part of box rotation.
-------------------------------	---

<code>\l__box_cos_fp</code>	The sine and cosine of the angle through which a box is rotated by <code>\box_rotate:Nn</code> : the values refer to the angle counter-clockwise. These values are required by the underlying driver code in <code>l3driver</code> to carry out the driver-dependent part of box rotation.
<code>\l__box_sin_fp</code>	

<code>\l__box_scale_x_fp</code>	The scaling factors by which a box is scaled by <code>\box_scale:Nnn</code> or <code>\box_resize:Nnn</code> . These values are required by the underlying driver code in <code>l3driver</code> to carry out the driver-dependent part of box rotation.
<code>\l__box_scale_y_fp</code>	

<code>\l__box_internal_box</code>	Box used for affine transformations, which is used to contain rotated material when applying <code>\box_rotate:Nn</code> . This box must be correctly constructed for the driver-dependent code in <code>l3driver</code> to function correctly.
-----------------------------------	---

3 Additions to l3clist

<hr/> <code>\clist_item:Nn</code> <hr/>	<code>\clist_item:Nn <comma list> {<integer expression>}</code>
<code>\clist_item:(cn nn)</code> *	Indexing items in the <i><comma list></i> from 1 at the top (left), this function will evaluate the <i><integer expression></i> and leave the appropriate item from the comma list in the input stream. If the <i><integer expression></i> is negative, indexing occurs from the bottom (right) of the comma list. When the <i><integer expression></i> is larger than the number of items in the <i><comma list></i> (as calculated by <code>\clist_count:N</code>) then the function will expand to nothing.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* will not expand further when appearing in an x-type argument expansion.

<hr/> <code>\clist_set_from_seq:NN</code> <hr/>	<code>\clist_set_from_seq:NN <comma list> <sequence></code>
<code>\clist_set_from_seq:(cn Nc cc)</code>	
<hr/> <code>\clist_gset_from_seq:NN</code> <hr/>	
<code>\clist_gset_from_seq:(cn Nc cc)</code>	

Sets the *<comma list>* to be equal to the content of the *<sequence>*. Items which contain either spaces or commas are surrounded by braces.

<hr/> <code>\clist_const:Nn</code> <hr/>	<code>\clist_const:Nn <clist var> {<comma list>}</code>
<code>\clist_const:(Nx cn cx)</code>	Creates a new constant <i><clist var></i> or raises an error if the name is already taken. The value of the <i><clist var></i> will be set globally to the <i><comma list></i> .

<hr/> <code>\clist_if_empty_p:n</code> *	<code>\clist_if_empty_p:n {<comma list>}</code>
<code>\clist_if_empty:nTF</code> *	<code>\clist_if_empty:nTF {<comma list>} {<true code>} {<false code>}</code>
	Tests if the <i><comma list></i> is empty (containing no items). The rules for space trimming are as for other n-type comma-list functions, hence the comma list <i>{~,~,~,~}</i> (without outer braces) is empty, while <i>{~,{}},}</i> (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

4 Additions to l3coffins

<hr/> <code>\coffin_resize:Nnn</code> <hr/>	<code>\coffin_resize:Nnn <coffin> {<width>} {<total-height>}</code>
<code>\coffin_resize:cnn</code>	Resized the <i><coffin></i> to <i><width></i> and <i><total-height></i> , both of which should be given as dimension expressions.
<hr/> <code>\coffin_rotate:Nn</code> <hr/>	<code>\coffin_rotate:Nn <coffin> {<angle>}</code>
<code>\coffin_rotate:cn</code>	Rotates the <i><coffin></i> by the given <i><angle></i> (given in degrees counter-clockwise). This process will rotate both the coffin content and poles. Multiple rotations will not result in the bounding box of the coffin growing unnecessarily.

`\coffin_scale:Nnn`
`\coffin_scale:cnn`

`\coffin_scale:Nnn` $\langle coffin \rangle$ $\{\langle x-scale \rangle\}$ $\{\langle y-scale \rangle\}$

Scales the $\langle coffin \rangle$ by a factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

5 Additions to l3file

`\ior_map_inline:Nn`

New: 2012-02-11

`\ior_map_inline:Nn` $\langle stream \rangle$ $\{\langle inline function \rangle\}$

Applies the $\langle inline function \rangle$ to $\langle lines \rangle$ obtained by reading one or more lines (until an equal number of left and right braces are found) from the $\langle stream \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle line \rangle$ as #1. Note that T_EX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. T_EX also ignores any trailing new-line marker from the file it reads.

`\ior_str_map_inline:Nn`

New: 2012-02-11

`\ior_str_map_inline:Nn` $\{\langle stream \rangle\}$ $\{\langle inline function \rangle\}$

Applies the $\langle inline function \rangle$ to every $\langle line \rangle$ in the $\langle stream \rangle$. The material is read from the $\langle stream \rangle$ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The $\langle inline function \rangle$ should consist of code which will receive the $\langle line \rangle$ as #1. Note that T_EX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. T_EX also ignores any trailing new-line marker from the file it reads.

`\ior_map_break:`

New: 2012-06-29

`\ior_map_break:`

Used to terminate a `\ior_map...` function before all lines from the $\langle stream \rangle$ have been processed. This will normally take place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \ior_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\ior_map...` scenario will lead to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before further items are taken from the input stream. This will depend on the design of the mapping function.

\ior_map_break:n

New: 2012-06-29

\ior_map_break:n {<tokens>}

Used to terminate a `\ior_map...` function before all lines in the *<stream>* have been processed, inserting the *<tokens>* after the mapping has ended. This will normally take place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \ior_map_break:n { <tokens> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\ior_map...` scenario will lead to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before the *<tokens>* are inserted into the input stream. This will depend on the design of the mapping function.

6 Additions to l3fp

\fp_set_from_dim:Nn**\fp_set_from_dim:cn****\fp_gset_from_dim:Nn****\fp_gset_from_dim:cn**

\fp_set_from_dim:Nn <floating point variable> {<dimexpr>}

Sets the *<floating point variable>* to the distance represented by the *<dimension expression>* in the units points. This means that distances given in other units are first converted to points before being assigned to the *<floating point variable>*.

7 Additions to l3prop

\prop_map_tokens:Nn ☆**\prop_map_tokens:cn** ☆

\prop_map_tokens:Nn <property list> {<code>}

Analogue of `\prop_map_function:NN` which maps several tokens instead of a single function. The *<code>* receives each key-value pair in the *<property list>* as two trailing brace groups. For instance,

```
\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }
```

will expand to the value corresponding to `mykey`: for each pair in `\l_my_prop` the function `\str_if_eq:nnT` receives `mykey`, the *<key>* and the *<value>* as its three arguments. For that specific task, `\prop_get:Nn` is faster.

<code>\prop_get:Nn</code> ★	<code>\prop_get:Nn</code> $\langle property\ list \rangle$ $\{\langle key \rangle\}$
<code>\prop_get:cn</code> ★	Expands to the $\langle value \rangle$ corresponding to the $\langle key \rangle$ in the $\langle property\ list \rangle$. If the $\langle key \rangle$ is missing, this has an empty expansion.

T_EXhackers note: This function is slower than the non-expandable analogue `\prop_get:NnN`. The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle value \rangle$ will not expand further when appearing in an x-type argument expansion.

8 Additions to l3seq

<code>\seq_item:Nn</code> ★	<code>\seq_item:Nn</code> $\langle sequence \rangle$ $\{\langle integer\ expression \rangle\}$
<code>\seq_item:cn</code> ★	Indexing items in the $\langle sequence \rangle$ from 1 at the top (left), this function will evaluate the $\langle integer\ expression \rangle$ and leave the appropriate item from the sequence in the input stream. If the $\langle integer\ expression \rangle$ is negative, indexing occurs from the bottom (right) of the sequence. When the $\langle integer\ expression \rangle$ is larger than the number of items in the $\langle sequence \rangle$ (as calculated by <code>\seq_count:N</code>) then the function will expand to nothing.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ will not expand further when appearing in an x-type argument expansion.

<code>\seq_mapthread_function:NNN</code> ★	<code>\seq_mapthread_function:NNN</code> $\langle seq_1 \rangle$ $\langle seq_2 \rangle$ $\langle function \rangle$
<code>\seq_mapthread_function:(NcN cNN ccN)</code> ★	

Applies $\langle function \rangle$ to every pair of items $\langle seq_1\text{-}item \rangle$ – $\langle seq_2\text{-}item \rangle$ from the two sequences, returning items from both sequences from left to right. The $\langle function \rangle$ will receive two n-type arguments for each iteration. The mapping will terminate when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

<code>\seq_set_from_clist:NN</code>	<code>\seq_set_from_clist:NN</code> $\langle sequence \rangle$ $\langle comma\text{-}list \rangle$
<code>\seq_set_from_clist:(cN Nc cc Nn cn)</code>	
<code>\seq_gset_from_clist:NN</code>	
<code>\seq_gset_from_clist:(cN Nc cc Nn cn)</code>	

Sets the $\langle sequence \rangle$ within the current T_EX group to be equal to the content of the $\langle comma\text{-}list \rangle$.

<code>\seq_reverse:N</code>	<code>\seq_reverse:N</code> $\langle sequence \rangle$
<code>\seq_greverse:N</code>	Reverses the order of items in the $\langle sequence \rangle$, and assigns the result to $\langle sequence \rangle$, locally or globally according to the variant chosen.

`\seq_set_filter:NNn`
`\seq_gset_filter:NNn`

`\seq_set_filter:NNn` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\{\langle inline\ boolexpr \rangle\}$

Evaluates the $\langle inline\ boolexpr \rangle$ for every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline\ boolexpr \rangle$ will receive the $\langle item \rangle$ as #1. The sequence of all $\langle items \rangle$ for which the $\langle inline\ boolexpr \rangle$ evaluated to `true` is assigned to $\langle sequence_1 \rangle$.

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level T_EX errors.

`\seq_set_map:NNn`
`\seq_gset_map:NNn`

New: 2011-12-22

`\seq_set_map:NNn` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\{\langle inline\ function \rangle\}$

Applies $\langle inline\ function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline\ function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. The sequence resulting from x-expanding $\langle inline\ function \rangle$ applied to each $\langle item \rangle$ is assigned to $\langle sequence_1 \rangle$. As such, the code in $\langle inline\ function \rangle$ should be expandable.

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level T_EX errors.

9 Additions to l3skip

`\dim_to_pt:n` ★

New: 2013-05-06

`\dim_to_pt:n` $\{\langle dimexpr \rangle\}$

Evaluates the $\langle dimension\ expression \rangle$, and leaves the result, expressed in points (`pt`) in the input stream, with *no units*. The result is rounded by T_EX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

If the $\langle dimension\ expression \rangle$ contains additional tokens such as redundant units, these will be ignored, so for example

`\dim_to_pt:n { 1 bp pt }`

leaves 1.00374 in the input stream, *i.e.* the magnitude of one “big point” when converted to points.

`\dim_to_unit:nn` ★

New: 2013-05-06

`\dim_to_unit:nn` $\{\langle dimexpr_1 \rangle\}$ $\{\langle dimexpr_2 \rangle\}$

Evaluates the $\langle dimension\ expressions \rangle$, and leaves the value of $\langle dimexpr_1 \rangle$, expressed in a unit given by $\langle dimexpr_2 \rangle$, in the input stream. The result is a decimal number, rounded by T_EX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

If the $\langle dimension\ expressions \rangle$ contain additional tokens such as redundant units, these will be ignored, so for example

`\dim_to_unit:nn { 1 bp pt } { 1 mm }`

leaves 0.35277 in the input stream, *i.e.* the magnitude of one “big point” when converted to millimeters.

<code>\skip_split_finite_else_action:nnNN</code>	<code>\skip_split_finite_else_action:nnNN {<skipexpr>} {<action>}</code>
	<code><dimen₁> <dimen₂></code>

Checks if the `<skipexpr>` contains finite glue. If it does then it assigns `<dimen1>` the stretch component and `<dimen2>` the shrink component. If it contains infinite glue set `<dimen1>` and `<dimen2>` to 0pt and place #2 into the input stream: this is usually an error or warning message of some sort.

10 Additions to l3tl

<code>\tl_if_single_token_p:n</code> ★	<code>\tl_if_single_token_p:n {<token list>}</code>
<code>\tl_if_single_token:nTF</code> ★	<code>\tl_if_single_token:nTF {<token list>} {<true code>} {<false code>}</code>

Tests if the token list consists of exactly one token, *i.e.* is either a single space character or a single “normal” token. Token groups (`{...}`) are not single tokens.

<code>\tl_reverse_tokens:n</code> ★	<code>\tl_reverse_tokens:n {<tokens>}</code>
-------------------------------------	--

This function, which works directly on T_EX tokens, reverses the order of the `<tokens>`: the first will be the last and the last will become first. Spaces are preserved. The reversal also operates within brace groups, but the braces themselves are not exchanged, as this would lead to an unbalanced token list. For instance, `\tl_reverse_tokens:n {a~{b()}}` leaves `{() (b)~a` in the input stream. This function requires two steps of expansion.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the token list will not expand further when appearing in an x-type argument expansion.

<code>\tl_count_tokens:n</code> ★	<code>\tl_count_tokens:n {<tokens>}</code>
-----------------------------------	--

Counts the number of T_EX tokens in the `<tokens>` and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of `a~{bc}` is 6. This function requires three expansions, giving an *<integer denotation>*.

<code>\tl_expandable_uppercase:n</code> ★	<code>\tl_expandable_uppercase:n {<tokens>}</code>
<code>\tl_expandable_lowercase:n</code> ★	<code>\tl_expandable_lowercase:n {<tokens>}</code>

The `\tl_expandable_uppercase:n` function works through all of the `<tokens>`, replacing characters in the range a–z (with arbitrary category code) by the corresponding letter in the range A–Z, with category code 11 (letter). Similarly, `\tl_expandable_lowercase:n` replaces characters in the range A–Z by letters in the range a–z, and leaves other tokens unchanged. This function requires two steps of expansion.

T_EXhackers note: Begin-group and end-group characters are normalized and become `{` and `}`, respectively. The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the token list will not expand further when appearing in an x-type argument expansion.

<code>\tl_item:nn</code>	★	<code>\tl_item:nn {⟨token list⟩} {⟨integer expression⟩}</code>
<code>\tl_item:(Nn cn)</code>	★	

Indexing items in the $\langle token list \rangle$ from 1 on the left, this function will evaluate the $\langle integer expression \rangle$ and leave the appropriate item from the $\langle token list \rangle$ in the input stream. If the $\langle integer expression \rangle$ is negative, indexing occurs from the right of the token list, starting at -1 for the right-most item. If the index is out of bounds, then the function expands to nothing.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ will not expand further when appearing in an x-type argument expansion.

11 Additions to l3tokens

<code>\char_set_active:Npn</code>	<code>\char_set_active:Npn ⟨char⟩ ⟨parameters⟩ {⟨code⟩}</code>
<code>\char_set_active:Npx</code>	

Makes $\langle char \rangle$ 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 TeX group level, and the definition is also local.

<code>\char_gset_active:Npn</code>	<code>\char_gset_active:Npn ⟨char⟩ ⟨parameters⟩ {⟨code⟩}</code>
<code>\char_gset_active:Npx</code>	

Makes $\langle char \rangle$ 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 TeX 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).

<code>\char_set_active_eq:NN</code>	<code>\char_set_active_eq:NN ⟨char⟩ ⟨function⟩</code>
-------------------------------------	---

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 TeX group level, and the definition is also local.

<code>\char_gset_active_eq:NN</code>	<code>\char_gset_active_eq:NN ⟨char⟩ ⟨function⟩</code>
--------------------------------------	--

Makes $\langle char \rangle$ 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 TeX 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 *<token>* in the input stream can be safely grabbed as an N-type argument. The test will be *<false>* if the next *<token>* is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), or an outer token (never used in L^AT_EX3) and *<true>* in all other cases. Note that a *<true>* result ensures that the next *<token>* is a valid N-type argument. However, if the next *<token>* is for instance `\c_space_token`, the test will take the *<false>* branch, even though the next *<token>* is in fact a valid N-type argument. The *<token>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

Part XXIV

Implementation

1 l3bootstrap implementation

```
1 <*initex | package>
2 <@@=expl>
```

1.1 Format-specific code

The very first thing to do is to bootstrap the iniT_EX system so that everything else will actually work. T_EX does not start with some pretty basic character codes set up.

```
3 <*initex>
4 \catcode '\{ = 1 \relax
5 \catcode '\} = 2 \relax
6 \catcode '\# = 6 \relax
7 \catcode '\^ = 7 \relax
8 </initex>
```

Tab characters should not show up in the code, but to be on the safe side.

```
9 <*initex>
10 \catcode '\^^I = 10 \relax
11 </initex>
```

For LuaT_EX the extra primitives need to be enabled before they can be used. No `\ifdefined` yet, so do it the old-fashioned way. The primitive `\strcmp` is simulated using some Lua code, which currently has to be applied to every job as the Lua code is not part of the format. Thanks to Taco Hoekwater for this code. The odd `\csname` business is needed so that the later deletion code will work.

```
12 <*initex>
13 \begingroup\expandafter\expandafter\expandafter\endgroup
14 \expandafter\ifx\csname directlua\endcsname\relax
15 \else
16 \directlua
17 {
```

```

18 tex.enableprimitives('',tex.extraprimitives ())
19 lua.bytecode[1] = function ()
20   function strcmp (A, B)
21     if A == B then
22       tex.write("0")
23     elseif A < B then
24       tex.write("-1")
25     else
26       tex.write("1")
27     end
28   end
29 end
30 lua.bytecode[1]()
31 }
32 \everyjob\expandafter
33 {\csname\detokenize{luatex_directlua:D}\endcsname{lua.bytecode[1]()}}
34 \long\edef\pdfstrcmp#1#2%
35 {%
36   \expandafter\noexpand\csname\detokenize{luatex_directlua:D}\endcsname
37   {%
38     strcmp%
39     (%
40       "\noexpand\luaescapestring{#1}",%
41       "\noexpand\luaescapestring{#2}"%
42     )%
43   }%
44 }
45 \fi
46 \</initex>

```

1.2 Package-specific code part one

The package starts by identifying itself: the information itself is taken from the SVN Id string at the start of the source file.

```

47 <*package>
48 \ProvidesPackage{l3bootstrap}
49 [%
50   \ExplFileDate\space v\ExplFileVersion\space
51   L3 Experimental bootstrap code%
52 ]
53 </package>

```

For LuaTeX the functionality of the `\pdfstrcmp` primitive needs to be provided: the `pdftexmcds` package is used to do this if necessary. At present, there is also a need to deal with some low-level allocation stuff that could usefully be added to `lualatex.ini`. As it is currently not, load Heiko Oberdiek's `luatex` package instead.

```

54 <*package>
55 \def\@tempa%
56 {%

```

```

57 \def\@tempa{}%
58 \RequirePackage{luatex}%
59 \RequirePackage{pdftexcmds}%
60 \let\pdfstrcmp\pdf@strcmp
61 }
62 \begingroup\expandafter\expandafter\expandafter\endgroup
63 \expandafter\ifx\csname directlua\endcsname\relax
64 \else
65 \expandafter\@tempa
66 \fi
67 \endpackage

```

1.3 The `\pdfstrcmp` primitive in XeTeX

Only pdfTeX has a primitive called `\pdfstrcmp`. The XeTeX version is just `\strcmp`, so there is some shuffling to do.

```

68 \begingroup\expandafter\expandafter\expandafter\endgroup
69 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
70 \let\pdfstrcmp\strcmp
71 \fi

```

1.4 Engine requirements

The code currently requires functionality equivalent to `\pdfstrcmp` in addition to ε -TeX. The former is therefore used as a test for a suitable engine.

```

72 \begingroup\expandafter\expandafter\expandafter\endgroup
73 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
74 \*package
75 \PackageError{expl3}{Required primitives not found}
76 {%
77 LaTeX3 requires the e-TeX primitives and \string\pdfstrcmp.\MessageBreak
78 \MessageBreak
79 These are available in engine versions:\MessageBreak
80 - pdfTeX 1.30\MessageBreak
81 - XeTeX 0.9994\MessageBreak
82 - LuaTeX 0.40\MessageBreak
83 or later.\MessageBreak
84 \MessageBreak
85 Loading of expl3 will abort!%
86 }
87 \expandafter\endinput
88 \endpackage
89 \*initex
90 \newlinechar'\^^J\relax
91 \errhelp{%
92 LaTeX3 requires the e-TeX primitives and \pdfstrcmp.\^^J%
93 \^^J%
94 These are available in engine versions:\^^J%
95 - pdfTeX 1.30\^^J%

```

```

96      - XeTeX 0.9994^^J%
97      - LuaTeX 0.40^^J%
98      or later.^^J%
99      ^^J%
100     For pdfTeX and XeTeX the '-etex' command-line switch is also
101     needed.^^J%
102     ^^J%
103     Format building will abort!%
104 }
105 \errmessage{Required primitives not found}%
106 \expandafter\end
107 </initex>
108 \fi

```

1.5 Package-specific code part two

`\ExplSyntaxOff` Experimental syntax switching is set up here for the package-loading process. These are redefined in `expl3` for the package and in `l3final` for the format.

```

109 <*package>
110 \protected\edef\ExplSyntaxOff
111 {%
112   \catcode 9 = \the\catcode 9\relax
113   \catcode 32 = \the\catcode 32\relax
114   \catcode 34 = \the\catcode 34\relax
115   \catcode 38 = \the\catcode 38\relax
116   \catcode 58 = \the\catcode 58\relax
117   \catcode 94 = \the\catcode 94\relax
118   \catcode 95 = \the\catcode 95\relax
119   \catcode 124 = \the\catcode 124\relax
120   \catcode 126 = \the\catcode 126\relax
121   \endlinechar = \the\endlinechar\relax
122   \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0 \relax
123 }
124 \protected\edef\ExplSyntaxOn
125 {
126   \catcode 9 = 9 \relax
127   \catcode 32 = 9 \relax
128   \catcode 34 = 12 \relax
129   \catcode 58 = 11 \relax
130   \catcode 94 = 7 \relax
131   \catcode 95 = 11 \relax
132   \catcode 124 = 12 \relax
133   \catcode 126 = 10 \relax
134   \endlinechar = 32 \relax
135   \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 1 \relax
136 }
137 </package>

```

(End definition for `\ExplSyntaxOff` and `\ExplSyntaxOn`. These functions are documented on page 6.)

`\l__kernel_expl_bool` The status for experimental code syntax: this is off at present. This code is used by both the package and the format.

```

138 \expandafter\chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0 \relax

```

(End definition for `\l__kernel_expl_bool`. This variable is documented on page 7.)

1.6 Dealing with package-mode meta-data

`\GetIdInfo` This is implemented right at the start of `l3bootstrap.dtx`.
(End definition for `\GetIdInfo`. This function is documented on page 6.)

`\ProvidesExplPackage`
`\ProvidesExplClass`
`\ProvidesExplFile` For other packages and classes building on this one it is convenient not to need `\ExplSyntaxOn` each time.

```

139 <*package>
140 \protected\def\ProvidesExplPackage
141   {%
142     \@ifpackageloaded{expl3}
143     {}
144     {%
145       \PackageError{expl3}
146         {Cannot load the expl3 modules separately}
147         {%
148           The expl3 modules cannot be loaded separately;\MessageBreak
149           please \string\usepackage\string{expl3\string} instead.%
150         }%
151     }%
152   \protected\def\ProvidesExplPackage##1##2##3##4%
153     {%
154       \ProvidesPackage{##1}[##2 v##3 ##4]%
155       \ExplSyntaxOn
156     }%
157   \ProvidesExplPackage
158   }
159 \protected\def\ProvidesExplClass#1#2#3#4%
160   {%
161     \ProvidesClass{#1}[#2 v#3 #4]%
162     \ExplSyntaxOn
163   }
164 \protected\def\ProvidesExplFile#1#2#3#4%
165   {%
166     \ProvidesFile{#1}[#2 v#3 #4]%
167     \ExplSyntaxOn
168   }
169 </package>

```

(End definition for `\ProvidesExplPackage`, `\ProvidesExplClass`, and `\ProvidesExplFile`. These functions are documented on page 6.)

`\@pushfilename`
`\@popfilename` The idea here is to use L^AT_EX 2_ε's `\@pushfilename` and `\@popfilename` to track the current syntax status. This can be achieved by saving the current status flag at each

push to a stack, then recovering it at the pop stage and checking if the code environment should still be active.

```

170 <*package>
171 \edef\@pushfilename
172 {%
173   \edef\expandafter\noexpand
174   \csname\detokenize{l__expl_status_stack_tl}\endcsname
175   {%
176     \noexpand\ifodd\expandafter\noexpand
177     \csname\detokenize{l__kernel_expl_bool}\endcsname
178     1%
179     \noexpand\else
180     0%
181     \noexpand\fi
182     \expandafter\noexpand
183     \csname\detokenize{l__expl_status_stack_tl}\endcsname
184   }%
185   \ExplSyntaxOff
186   \unexpanded\expandafter{\@pushfilename}%
187 }
188 \edef\@popfilename
189 {%
190   \unexpanded\expandafter{\@popfilename}%
191   \noexpand\if a\expandafter\noexpand\csname
192     \detokenize{l__expl_status_stack_tl}\endcsname a%
193   \ExplSyntaxOff
194   \noexpand\else
195     \noexpand\expandafter
196     \expandafter\noexpand\csname
197       \detokenize{__expl_status_pop:w}\endcsname
198     \expandafter\noexpand\csname
199       \detokenize{l__expl_status_stack_tl}\endcsname
200     \noexpand\@nil
201   \noexpand\fi
202 }
203 </package>

```

(End definition for \@pushfilename and \@popfilename. These functions are documented on page ??.)

`\l__expl_status_stack_tl` As expl3 itself cannot be loaded with the code environment already active, at the end of the package `\ExplSyntaxOff` can safely be called.

```

204 <*package>
205 \@namedef{\detokenize{l__expl_status_stack_tl}}{0}
206 </package>

```

(End definition for `\l__expl_status_stack_tl`. This function is documented on page ??.)

`__expl_status_pop:w` The pop auxiliary function removes the first item from the stack, saves the rest of the stack and then does the test. As `\ExplSyntaxOff` is already defined as a protected macro, there is no need for `\noexpand` here.

```

207 <*package>

```

```

208 \expandafter\edef\csname\detokenize{__expl_status_pop:w}\endcsname#1#2\@nil
209 {%
210   \def\expandafter\noexpand
211     \csname\detokenize{l__expl_status_stack_tl}\endcsname{#2}%
212     \noexpand\ifodd#1\space
213     \noexpand\expandafter\noexpand\ExplSyntaxOn
214     \noexpand\else
215     \noexpand\expandafter\ExplSyntaxOff
216     \noexpand\fi
217   }
218 \</package>
(End definition for \__expl_status_pop:w.)

```

`__expl_package_check:` We want the expl3 bundle to be loaded “as one”; this command is used to ensure that one of the 13 packages isn’t loaded on its own.

```

219 \<*package>
220 \expandafter\protected\expandafter\def
221   \csname\detokenize{__expl_package_check:}\endcsname
222   {%
223     \@ifpackageloaded{expl3}
224     {}
225     {%
226       \PackageError{expl3}
227         {Cannot load the expl3 modules separately}
228         {%
229           The expl3 modules cannot be loaded separately;\MessageBreak
230           please \string\usepackage\string{expl3\string} instead.%
231         }%
232     }%
233   }
234 \</package>
(End definition for \__expl_package_check:.)

```

1.7 The L^AT_EX3 code environment

The code environment is now set up for the format: the package deals with this using `\ProvidesExplPackage`.

```

235 \<*initex>
236 \catcode 9 = 9 \relax
237 \catcode 32 = 9 \relax
238 \catcode 34 = 12 \relax
239 \catcode 58 = 11 \relax
240 \catcode 94 = 7 \relax
241 \catcode 95 = 11 \relax
242 \catcode 124 = 12 \relax
243 \catcode 126 = 10 \relax
244 \endlinechar = 32 \relax
245 \</initex>

```


`\ExplSyntaxOn` The idea here is that multiple `\ExplSyntaxOn` calls are not going to mess up category codes, and that multiple calls to `\ExplSyntaxOff` are also not wasting time.

```

246 <*initex>
247 \protected \def \ExplSyntaxOn
248 {
249   \bool_if:NF \l__kernel_expl_bool
250   {
251     \cs_set_protected_nopar:Npx \ExplSyntaxOff
252     {
253       \char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
254       \char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
255       \char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
256       \char_set_catcode:nn { 38 } { \char_value_catcode:n { 38 } }
257       \char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
258       \char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
259       \char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }
260       \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
261       \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
262       \tex_endlinechar:D =
263       \tex_the:D \tex_endlinechar:D \scan_stop:
264       \bool_set_false:N \l__kernel_expl_bool
265       \cs_set_protected_nopar:Npn \ExplSyntaxOff { }
266     }
267   }
268   \char_set_catcode_ignore:n { 9 } % tab
269   \char_set_catcode_ignore:n { 32 } % space
270   \char_set_catcode_other:n { 34 } % double quote
271   \char_set_catcode_alignment:n { 38 } % ampersand
272   \char_set_catcode_letter:n { 58 } % colon
273   \char_set_catcode_math_superscript:n { 94 } % circumflex
274   \char_set_catcode_letter:n { 95 } % underscore
275   \char_set_catcode_other:n { 124 } % pipe
276   \char_set_catcode_space:n { 126 } % tilde
277   \tex_endlinechar:D = 32 \scan_stop:
278   \bool_set_true:N \l__kernel_expl_bool
279 }
280 \protected \def \ExplSyntaxOff { }
281 </initex>

```

(End definition for `\ExplSyntaxOn` and `\ExplSyntaxOff`. These functions are documented on page 6.)

`\l__kernel_expl_bool` A flag to show the current syntax status.

```

282 <*initex>
283 \chardef \l__kernel_expl_bool = 0 ~
284 </initex>

```

(End definition for `\l__kernel_expl_bool`. This variable is documented on page 7.)

1.8 Deprecated functions

Deprecated 2012-06-19 for removal after 2012-12-31.

`\ExplSyntaxNamesOn` These can be set up early, as they are not used anywhere in the package or format itself.
`\ExplSyntaxNamesOff` Using an `\edef` here makes the definitions that bit clearer later.

```

285 <*deprecated>
286 \protected\edef\ExplSyntaxNamesOn
287 {%
288   \expandafter\noexpand
289     \csname\detokenize{char_set_catcode_letter:n}\endcsname{58}%
290   \expandafter\noexpand
291     \csname\detokenize{char_set_catcode_letter:n}\endcsname{95}%
292 }
293 \protected\edef\ExplSyntaxNamesOff
294 {%
295   \expandafter\noexpand
296     \csname\detokenize{char_set_catcode_other:n}\endcsname{58}%
297   \expandafter\noexpand
298     \csname\detokenize{char_set_catcode_math_subscript:n}\endcsname{95}%
299 }
300 </deprecated>

```

(End definition for `\ExplSyntaxNamesOn` and `\ExplSyntaxNamesOff`. These functions are documented on page ??.)

```

301 </initex | package>

```

2 l3names implementation

```

302 <*initex | package>
303 <*package>
304 \ProvidesExplPackage
305   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
306 </package>

```

The code here simply renames all of the primitives to new, internal, names. In format mode, it also deletes all of the existing names (although some do come back later).

`\tex_undefined:D` This function does not exist at all, but is the name used by the plain T_EX format for an undefined function. So it should be marked here as “taken”.

(End definition for `\tex_undefined:D`. This function is documented on page ??.)

The `\let` primitive is renamed by hand first as it is essential for the entire process to follow. This also uses `\global`, as that way we avoid leaving an unneeded csname in the hash table.

```

307 \let \tex_global:D \global
308 \let \tex_let:D \let

```

Everything is inside a (rather long) group, which keeps `__expl_primitive:NN` trapped.

```

309 \begingroup

```

`__expl_primitive:NN` A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

```

310 \long \def \_expl_primitive:NN #1#2
311 {
312   \tex_global:D \tex_let:D #2 #1
313 \*initex>
314   \tex_global:D \tex_let:D #1 \tex_undefined:D
315 \</initex>
316 }

```

(End definition for _expl_primitive:NN.)

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.

```

317 \_expl_primitive:NN \           \tex_space:D
318 \_expl_primitive:NN \</           \tex_italiccorrection:D
319 \_expl_primitive:NN \-           \tex_hyphen:D

```

Now all the other primitives.

```

320 \_expl_primitive:NN \let           \tex_let:D
321 \_expl_primitive:NN \def           \tex_def:D
322 \_expl_primitive:NN \edef          \tex_edef:D
323 \_expl_primitive:NN \gdef          \tex_gdef:D
324 \_expl_primitive:NN \xdef          \tex_xdef:D
325 \_expl_primitive:NN \chardef       \tex_chardef:D
326 \_expl_primitive:NN \countdef      \tex_countdef:D
327 \_expl_primitive:NN \dimendef      \tex_dimendef:D
328 \_expl_primitive:NN \skipdef       \tex_skipdef:D
329 \_expl_primitive:NN \muskipdef     \tex_muskipdef:D
330 \_expl_primitive:NN \mathchardef   \tex_mathchardef:D
331 \_expl_primitive:NN \toksdef       \tex_toksdef:D
332 \_expl_primitive:NN \futurelet     \tex_futurelet:D
333 \_expl_primitive:NN \advance       \tex_advance:D
334 \_expl_primitive:NN \divide        \tex_divide:D
335 \_expl_primitive:NN \multiply      \tex_multiply:D
336 \_expl_primitive:NN \font          \tex_font:D
337 \_expl_primitive:NN \fam           \tex_fam:D
338 \_expl_primitive:NN \global        \tex_global:D
339 \_expl_primitive:NN \long          \tex_long:D
340 \_expl_primitive:NN \outer         \tex_outer:D
341 \_expl_primitive:NN \setlanguage    \tex_setlanguage:D
342 \_expl_primitive:NN \globaldefs     \tex_globaldefs:D
343 \_expl_primitive:NN \afterassignment \tex_afterassignment:D
344 \_expl_primitive:NN \aftergroup     \tex_aftergroup:D
345 \_expl_primitive:NN \expandafter   \tex_expandafter:D
346 \_expl_primitive:NN \noexpand      \tex_noexpand:D
347 \_expl_primitive:NN \begingroup     \tex_begingroup:D
348 \_expl_primitive:NN \endgroup       \tex_endgroup:D
349 \_expl_primitive:NN \halign        \tex_halign:D
350 \_expl_primitive:NN \valign        \tex_valign:D
351 \_expl_primitive:NN \cr             \tex_cr:D

```

352	_expl_primitive:NN	\crrc	\tex_crrc:D
353	_expl_primitive:NN	\noalign	\tex_noalign:D
354	_expl_primitive:NN	\omit	\tex_omit:D
355	_expl_primitive:NN	\span	\tex_span:D
356	_expl_primitive:NN	\tabskip	\tex_tabskip:D
357	_expl_primitive:NN	\everycr	\tex_everycr:D
358	_expl_primitive:NN	\if	\tex_if:D
359	_expl_primitive:NN	\ifcase	\tex_ifcase:D
360	_expl_primitive:NN	\ifcat	\tex_ifcat:D
361	_expl_primitive:NN	\ifnum	\tex_ifnum:D
362	_expl_primitive:NN	\ifodd	\tex_ifodd:D
363	_expl_primitive:NN	\ifdim	\tex_ifdim:D
364	_expl_primitive:NN	\ifeof	\tex_ifeof:D
365	_expl_primitive:NN	\ifhbox	\tex_ifhbox:D
366	_expl_primitive:NN	\ifvbox	\tex_ifvbox:D
367	_expl_primitive:NN	\ifvoid	\tex_ifvoid:D
368	_expl_primitive:NN	\ifx	\tex_ifx:D
369	_expl_primitive:NN	\iffalse	\tex_iffalse:D
370	_expl_primitive:NN	\iftrue	\tex_iftrue:D
371	_expl_primitive:NN	\ifhmode	\tex_ifhmode:D
372	_expl_primitive:NN	\ifmmode	\tex_ifmmode:D
373	_expl_primitive:NN	\ifvmode	\tex_ifvmode:D
374	_expl_primitive:NN	\ifinner	\tex_ifinner:D
375	_expl_primitive:NN	\else	\tex_else:D
376	_expl_primitive:NN	\fi	\tex_fi:D
377	_expl_primitive:NN	\or	\tex_or:D
378	_expl_primitive:NN	\immediate	\tex_immediate:D
379	_expl_primitive:NN	\closeout	\tex_closeout:D
380	_expl_primitive:NN	\openin	\tex_openin:D
381	_expl_primitive:NN	\openout	\tex_openout:D
382	_expl_primitive:NN	\read	\tex_read:D
383	_expl_primitive:NN	\write	\tex_write:D
384	_expl_primitive:NN	\closein	\tex_closein:D
385	_expl_primitive:NN	\newlinechar	\tex_newlinechar:D
386	_expl_primitive:NN	\input	\tex_input:D
387	_expl_primitive:NN	\endinput	\tex_endinput:D
388	_expl_primitive:NN	\inputlineno	\tex_inputlineno:D
389	_expl_primitive:NN	\errmessage	\tex_errmessage:D
390	_expl_primitive:NN	\message	\tex_message:D
391	_expl_primitive:NN	\show	\tex_show:D
392	_expl_primitive:NN	\showthe	\tex_showthe:D
393	_expl_primitive:NN	\showbox	\tex_showbox:D
394	_expl_primitive:NN	\showlists	\tex_showlists:D
395	_expl_primitive:NN	\errhelp	\tex_errhelp:D
396	_expl_primitive:NN	\errorcontextlines	\tex_errorcontextlines:D
397	_expl_primitive:NN	\tracingcommands	\tex_tracingcommands:D
398	_expl_primitive:NN	\tracinglostchars	\tex_tracinglostchars:D
399	_expl_primitive:NN	\tracingmacros	\tex_tracingmacros:D
400	_expl_primitive:NN	\tracingonline	\tex_tracingonline:D
401	_expl_primitive:NN	\tracingoutput	\tex_tracingoutput:D

402	_expl_primitive:NN	\tracingpages	\tex_tracingpages:D
403	_expl_primitive:NN	\tracingparagraphs	\tex_tracingparagraphs:D
404	_expl_primitive:NN	\tracingrestores	\tex_tracingrestores:D
405	_expl_primitive:NN	\tracingstats	\tex_tracingstats:D
406	_expl_primitive:NN	\pausing	\tex_pausing:D
407	_expl_primitive:NN	\showboxbreadth	\tex_showboxbreadth:D
408	_expl_primitive:NN	\showboxdepth	\tex_showboxdepth:D
409	_expl_primitive:NN	\batchmode	\tex_batchmode:D
410	_expl_primitive:NN	\errorstopmode	\tex_errorstopmode:D
411	_expl_primitive:NN	\nonstopmode	\tex_nonstopmode:D
412	_expl_primitive:NN	\scrollmode	\tex_scrollmode:D
413	_expl_primitive:NN	\end	\tex_end:D
414	_expl_primitive:NN	\csname	\tex_csname:D
415	_expl_primitive:NN	\endcsname	\tex_endcsname:D
416	_expl_primitive:NN	\ignorespaces	\tex_ignorespaces:D
417	_expl_primitive:NN	\relax	\tex_relax:D
418	_expl_primitive:NN	\the	\tex_the:D
419	_expl_primitive:NN	\mag	\tex_mag:D
420	_expl_primitive:NN	\language	\tex_language:D
421	_expl_primitive:NN	\mark	\tex_mark:D
422	_expl_primitive:NN	\topmark	\tex_topmark:D
423	_expl_primitive:NN	\firstmark	\tex_firstmark:D
424	_expl_primitive:NN	\botmark	\tex_botmark:D
425	_expl_primitive:NN	\splitfirstmark	\tex_splitfirstmark:D
426	_expl_primitive:NN	\splitbotmark	\tex_splitbotmark:D
427	_expl_primitive:NN	\fontname	\tex_fontname:D
428	_expl_primitive:NN	\escapechar	\tex_escapechar:D
429	_expl_primitive:NN	\endlinechar	\tex_endlinechar:D
430	_expl_primitive:NN	\mathchoice	\tex_mathchoice:D
431	_expl_primitive:NN	\delimiter	\tex_delimiter:D
432	_expl_primitive:NN	\mathaccent	\tex_mathaccent:D
433	_expl_primitive:NN	\mathchar	\tex_mathchar:D
434	_expl_primitive:NN	\mskip	\tex_mskip:D
435	_expl_primitive:NN	\radical	\tex_radical:D
436	_expl_primitive:NN	\vcenter	\tex_vcenter:D
437	_expl_primitive:NN	\mkern	\tex_mkern:D
438	_expl_primitive:NN	\above	\tex_above:D
439	_expl_primitive:NN	\abovewithdelims	\tex_abovewithdelims:D
440	_expl_primitive:NN	\atop	\tex_atop:D
441	_expl_primitive:NN	\atopwithdelims	\tex_atopwithdelims:D
442	_expl_primitive:NN	\over	\tex_over:D
443	_expl_primitive:NN	\overwithdelims	\tex_overwithdelims:D
444	_expl_primitive:NN	\displaystyle	\tex_displaystyle:D
445	_expl_primitive:NN	\textstyle	\tex_textstyle:D
446	_expl_primitive:NN	\scriptstyle	\tex_scriptstyle:D
447	_expl_primitive:NN	\scriptscriptstyle	\tex_scriptscriptstyle:D
448	_expl_primitive:NN	\nonscript	\tex_nonscript:D
449	_expl_primitive:NN	\eqno	\tex_eqno:D
450	_expl_primitive:NN	\leqno	\tex_leqno:D
451	_expl_primitive:NN	\abovedisplayshortskip	\tex_abovedisplayshortskip:D

452	_expl_primitive:NN	\abovedisplayskip	\tex_abovedisplayskip:D
453	_expl_primitive:NN	\belowdisplaysshortskip	\tex_belowdisplaysshortskip:D
454	_expl_primitive:NN	\belowdisplayskip	\tex_belowdisplayskip:D
455	_expl_primitive:NN	\displaywidowpenalty	\tex_displaywidowpenalty:D
456	_expl_primitive:NN	\displayindent	\tex_displayindent:D
457	_expl_primitive:NN	\displaywidth	\tex_displaywidth:D
458	_expl_primitive:NN	\everydisplay	\tex_everydisplay:D
459	_expl_primitive:NN	\predisplaysize	\tex_predisplaysize:D
460	_expl_primitive:NN	\predisplaypenalty	\tex_predisplaypenalty:D
461	_expl_primitive:NN	\postdisplaypenalty	\tex_postdisplaypenalty:D
462	_expl_primitive:NN	\mathbin	\tex_mathbin:D
463	_expl_primitive:NN	\mathclose	\tex_mathclose:D
464	_expl_primitive:NN	\mathinner	\tex_mathinner:D
465	_expl_primitive:NN	\mathop	\tex_mathop:D
466	_expl_primitive:NN	\displaylimits	\tex_displaylimits:D
467	_expl_primitive:NN	\limits	\tex_limits:D
468	_expl_primitive:NN	\nolimits	\tex_nolimits:D
469	_expl_primitive:NN	\mathopen	\tex_mathopen:D
470	_expl_primitive:NN	\mathord	\tex_mathord:D
471	_expl_primitive:NN	\mathpunct	\tex_mathpunct:D
472	_expl_primitive:NN	\mathrel	\tex_mathrel:D
473	_expl_primitive:NN	\overline	\tex_overline:D
474	_expl_primitive:NN	\underline	\tex_underline:D
475	_expl_primitive:NN	\left	\tex_left:D
476	_expl_primitive:NN	\right	\tex_right:D
477	_expl_primitive:NN	\binoppenalty	\tex_binoppenalty:D
478	_expl_primitive:NN	\relpenalty	\tex_relpenalty:D
479	_expl_primitive:NN	\delimitershortfall	\tex_delimitershortfall:D
480	_expl_primitive:NN	\delimiterfactor	\tex_delimiterfactor:D
481	_expl_primitive:NN	\nulldelimiterspace	\tex_nulldelimiterspace:D
482	_expl_primitive:NN	\everymath	\tex_everymath:D
483	_expl_primitive:NN	\mathsurround	\tex_mathsurround:D
484	_expl_primitive:NN	\medmuskip	\tex_medmuskip:D
485	_expl_primitive:NN	\thinmuskip	\tex_thinmuskip:D
486	_expl_primitive:NN	\thickmuskip	\tex_thickmuskip:D
487	_expl_primitive:NN	\scriptspace	\tex_scriptspace:D
488	_expl_primitive:NN	\noboundary	\tex_noboundary:D
489	_expl_primitive:NN	\accent	\tex_accent:D
490	_expl_primitive:NN	\char	\tex_char:D
491	_expl_primitive:NN	\discretionary	\tex_discretionary:D
492	_expl_primitive:NN	\hfil	\tex_hfil:D
493	_expl_primitive:NN	\hfilneg	\tex_hfilneg:D
494	_expl_primitive:NN	\hfill	\tex_hfill:D
495	_expl_primitive:NN	\hskip	\tex_hskip:D
496	_expl_primitive:NN	\hss	\tex_hss:D
497	_expl_primitive:NN	\vfil	\tex_vfil:D
498	_expl_primitive:NN	\vfilneg	\tex_vfilneg:D
499	_expl_primitive:NN	\vfill	\tex_vfill:D
500	_expl_primitive:NN	\vskip	\tex_vskip:D
501	_expl_primitive:NN	\vss	\tex_vss:D

502	_expl_primitive:NN	\unskip	\tex_unskip:D
503	_expl_primitive:NN	\kern	\tex_kern:D
504	_expl_primitive:NN	\unkern	\tex_unkern:D
505	_expl_primitive:NN	\hrule	\tex_hrule:D
506	_expl_primitive:NN	\vrule	\tex_vrule:D
507	_expl_primitive:NN	\leaders	\tex_leaders:D
508	_expl_primitive:NN	\cleaders	\tex_cleaders:D
509	_expl_primitive:NN	\xleaders	\tex_xleaders:D
510	_expl_primitive:NN	\lastkern	\tex_lastkern:D
511	_expl_primitive:NN	\lastskip	\tex_lastskip:D
512	_expl_primitive:NN	\indent	\tex_indent:D
513	_expl_primitive:NN	\par	\tex_par:D
514	_expl_primitive:NN	\noindent	\tex_noindent:D
515	_expl_primitive:NN	\vadjust	\tex_vadjust:D
516	_expl_primitive:NN	\baselineskip	\tex_baselineskip:D
517	_expl_primitive:NN	\lineskip	\tex_lineskip:D
518	_expl_primitive:NN	\lineskiplimit	\tex_lineskiplimit:D
519	_expl_primitive:NN	\clubpenalty	\tex_clubpenalty:D
520	_expl_primitive:NN	\widowpenalty	\tex_widowpenalty:D
521	_expl_primitive:NN	\exhyphenpenalty	\tex_exhyphenpenalty:D
522	_expl_primitive:NN	\hyphenpenalty	\tex_hyphenpenalty:D
523	_expl_primitive:NN	\linepenalty	\tex_linepenalty:D
524	_expl_primitive:NN	\doublehyphendemerits	\tex_doublehyphendemerits:D
525	_expl_primitive:NN	\finalhyphendemerits	\tex_finalhyphendemerits:D
526	_expl_primitive:NN	\adjdemerits	\tex_adjdemerits:D
527	_expl_primitive:NN	\hangafter	\tex_hangafter:D
528	_expl_primitive:NN	\hangindent	\tex_hangindent:D
529	_expl_primitive:NN	\parshape	\tex_parshape:D
530	_expl_primitive:NN	\hsize	\tex_hsize:D
531	_expl_primitive:NN	\lefthyphenmin	\tex_lefthyphenmin:D
532	_expl_primitive:NN	\righthyphenmin	\tex_righthyphenmin:D
533	_expl_primitive:NN	\leftskip	\tex_leftskip:D
534	_expl_primitive:NN	\rightskip	\tex_rightskip:D
535	_expl_primitive:NN	\looseness	\tex_looseness:D
536	_expl_primitive:NN	\parskip	\tex_parskip:D
537	_expl_primitive:NN	\parindent	\tex_parindent:D
538	_expl_primitive:NN	\uchyph	\tex_uchyph:D
539	_expl_primitive:NN	\emergencystretch	\tex_emergencystretch:D
540	_expl_primitive:NN	\pretolerance	\tex_pretolerance:D
541	_expl_primitive:NN	\tolerance	\tex_tolerance:D
542	_expl_primitive:NN	\spaceskip	\tex_spaceskip:D
543	_expl_primitive:NN	\xspaceskip	\tex_xspaceskip:D
544	_expl_primitive:NN	\parfillskip	\tex_parfillskip:D
545	_expl_primitive:NN	\everypar	\tex_everypar:D
546	_expl_primitive:NN	\prevgraf	\tex_prevgraf:D
547	_expl_primitive:NN	\spacefactor	\tex_spacefactor:D
548	_expl_primitive:NN	\shipout	\tex_shipout:D
549	_expl_primitive:NN	\vsize	\tex_vsize:D
550	_expl_primitive:NN	\interlinepenalty	\tex_interlinepenalty:D
551	_expl_primitive:NN	\brokenpenalty	\tex_brokenpenalty:D

552	_expl_primitive:NN	\topskip	\tex_topskip:D
553	_expl_primitive:NN	\maxdeadcycles	\tex_maxdeadcycles:D
554	_expl_primitive:NN	\maxdepth	\tex_maxdepth:D
555	_expl_primitive:NN	\output	\tex_output:D
556	_expl_primitive:NN	\deadcycles	\tex_deadcycles:D
557	_expl_primitive:NN	\pagedepth	\tex_pagedepth:D
558	_expl_primitive:NN	\pagestretch	\tex_pagestretch:D
559	_expl_primitive:NN	\pagefilstretch	\tex_pagefilstretch:D
560	_expl_primitive:NN	\pagefillstretch	\tex_pagefillstretch:D
561	_expl_primitive:NN	\pagefilllstretch	\tex_pagefilllstretch:D
562	_expl_primitive:NN	\pageshrink	\tex_pageshrink:D
563	_expl_primitive:NN	\pagegoal	\tex_pagegoal:D
564	_expl_primitive:NN	\pagetotal	\tex_pagetotal:D
565	_expl_primitive:NN	\outputpenalty	\tex_outputpenalty:D
566	_expl_primitive:NN	\hoffset	\tex_hoffset:D
567	_expl_primitive:NN	\voffset	\tex_voffset:D
568	_expl_primitive:NN	\insert	\tex_insert:D
569	_expl_primitive:NN	\holdinginserts	\tex_holdinginserts:D
570	_expl_primitive:NN	\floatingpenalty	\tex_floatingpenalty:D
571	_expl_primitive:NN	\insertpenalties	\tex_insertpenalties:D
572	_expl_primitive:NN	\lower	\tex_lower:D
573	_expl_primitive:NN	\moveleft	\tex_moveleft:D
574	_expl_primitive:NN	\moveright	\tex_moveright:D
575	_expl_primitive:NN	\raise	\tex_raise:D
576	_expl_primitive:NN	\copy	\tex_copy:D
577	_expl_primitive:NN	\lastbox	\tex_lastbox:D
578	_expl_primitive:NN	\vsplit	\tex_vsplit:D
579	_expl_primitive:NN	\unhbox	\tex_unhbox:D
580	_expl_primitive:NN	\unhcopy	\tex_unhcopy:D
581	_expl_primitive:NN	\unvbox	\tex_unvbox:D
582	_expl_primitive:NN	\unvcopy	\tex_unvcopy:D
583	_expl_primitive:NN	\setbox	\tex_setbox:D
584	_expl_primitive:NN	\hbox	\tex_hbox:D
585	_expl_primitive:NN	\vbox	\tex_vbox:D
586	_expl_primitive:NN	\vtop	\tex_vtop:D
587	_expl_primitive:NN	\prevdepth	\tex_prevdepth:D
588	_expl_primitive:NN	\badness	\tex_badness:D
589	_expl_primitive:NN	\hbadness	\tex_hbadness:D
590	_expl_primitive:NN	\vbadness	\tex_vbadness:D
591	_expl_primitive:NN	\hfuzz	\tex_hfuzz:D
592	_expl_primitive:NN	\vfuzz	\tex_vfuzz:D
593	_expl_primitive:NN	\overfullrule	\tex_overfullrule:D
594	_expl_primitive:NN	\boxmaxdepth	\tex_boxmaxdepth:D
595	_expl_primitive:NN	\splitmaxdepth	\tex_splitmaxdepth:D
596	_expl_primitive:NN	\splittopskip	\tex_splittopskip:D
597	_expl_primitive:NN	\everyhbox	\tex_everyhbox:D
598	_expl_primitive:NN	\everyvbox	\tex_everyvbox:D
599	_expl_primitive:NN	\nullfont	\tex_nullfont:D
600	_expl_primitive:NN	\textfont	\tex_textfont:D
601	_expl_primitive:NN	\scriptfont	\tex_scriptfont:D

602	<code>__expl_primitive:NN \scriptscriptfont</code>	<code>\tex_scriptscriptfont:D</code>
603	<code>__expl_primitive:NN \fontdimen</code>	<code>\tex_fontdimen:D</code>
604	<code>__expl_primitive:NN \hyphenchar</code>	<code>\tex_hyphenchar:D</code>
605	<code>__expl_primitive:NN \skewchar</code>	<code>\tex_skewchar:D</code>
606	<code>__expl_primitive:NN \defaultthyphenchar</code>	<code>\tex_defaultthyphenchar:D</code>
607	<code>__expl_primitive:NN \defaultskewchar</code>	<code>\tex_defaultskewchar:D</code>
608	<code>__expl_primitive:NN \number</code>	<code>\tex_number:D</code>
609	<code>__expl_primitive:NN \romannumeral</code>	<code>\tex_romannumeral:D</code>
610	<code>__expl_primitive:NN \string</code>	<code>\tex_string:D</code>
611	<code>__expl_primitive:NN \lowercase</code>	<code>\tex_lowercase:D</code>
612	<code>__expl_primitive:NN \uppercase</code>	<code>\tex_uppercase:D</code>
613	<code>__expl_primitive:NN \meaning</code>	<code>\tex_meaning:D</code>
614	<code>__expl_primitive:NN \penalty</code>	<code>\tex_penalty:D</code>
615	<code>__expl_primitive:NN \unpenalty</code>	<code>\tex_unpenalty:D</code>
616	<code>__expl_primitive:NN \lastpenalty</code>	<code>\tex_lastpenalty:D</code>
617	<code>__expl_primitive:NN \special</code>	<code>\tex_special:D</code>
618	<code>__expl_primitive:NN \dump</code>	<code>\tex_dump:D</code>
619	<code>__expl_primitive:NN \patterns</code>	<code>\tex_patterns:D</code>
620	<code>__expl_primitive:NN \hyphenation</code>	<code>\tex_hyphenation:D</code>
621	<code>__expl_primitive:NN \time</code>	<code>\tex_time:D</code>
622	<code>__expl_primitive:NN \day</code>	<code>\tex_day:D</code>
623	<code>__expl_primitive:NN \month</code>	<code>\tex_month:D</code>
624	<code>__expl_primitive:NN \year</code>	<code>\tex_year:D</code>
625	<code>__expl_primitive:NN \jobname</code>	<code>\tex_jobname:D</code>
626	<code>__expl_primitive:NN \everyjob</code>	<code>\tex_everyjob:D</code>
627	<code>__expl_primitive:NN \count</code>	<code>\tex_count:D</code>
628	<code>__expl_primitive:NN \dimen</code>	<code>\tex_dimen:D</code>
629	<code>__expl_primitive:NN \skip</code>	<code>\tex_skip:D</code>
630	<code>__expl_primitive:NN \toks</code>	<code>\tex_toks:D</code>
631	<code>__expl_primitive:NN \muskip</code>	<code>\tex_muskip:D</code>
632	<code>__expl_primitive:NN \box</code>	<code>\tex_box:D</code>
633	<code>__expl_primitive:NN \wd</code>	<code>\tex_wd:D</code>
634	<code>__expl_primitive:NN \ht</code>	<code>\tex_ht:D</code>
635	<code>__expl_primitive:NN \dp</code>	<code>\tex_dp:D</code>
636	<code>__expl_primitive:NN \catcode</code>	<code>\tex_catcode:D</code>
637	<code>__expl_primitive:NN \delcode</code>	<code>\tex_delcode:D</code>
638	<code>__expl_primitive:NN \sfcode</code>	<code>\tex_sfcode:D</code>
639	<code>__expl_primitive:NN \lccode</code>	<code>\tex_lccode:D</code>
640	<code>__expl_primitive:NN \uccode</code>	<code>\tex_uccode:D</code>
641	<code>__expl_primitive:NN \mathcode</code>	<code>\tex_mathcode:D</code>

Since L^AT_EX3 requires at least the ε -T_EX extensions, we also rename the additional primitives. These are all given the prefix `\etex_`.

642	<code>__expl_primitive:NN \ifdefined</code>	<code>\etex_ifdefined:D</code>
643	<code>__expl_primitive:NN \ifcsname</code>	<code>\etex_ifcsname:D</code>
644	<code>__expl_primitive:NN \unless</code>	<code>\etex_unless:D</code>
645	<code>__expl_primitive:NN \eTeXversion</code>	<code>\etex_eTeXversion:D</code>
646	<code>__expl_primitive:NN \eTeXrevision</code>	<code>\etex_eTeXrevision:D</code>
647	<code>__expl_primitive:NN \marks</code>	<code>\etex_marks:D</code>
648	<code>__expl_primitive:NN \topmarks</code>	<code>\etex_topmarks:D</code>

649	_expl_primitive:NN	\firstmarks	\etex_firstmarks:D
650	_expl_primitive:NN	\botmarks	\etex_botmarks:D
651	_expl_primitive:NN	\splitfirstmarks	\etex_splitfirstmarks:D
652	_expl_primitive:NN	\splitbotmarks	\etex_splitbotmarks:D
653	_expl_primitive:NN	\unexpanded	\etex_unexpanded:D
654	_expl_primitive:NN	\detokenize	\etex_detokenize:D
655	_expl_primitive:NN	\scantokens	\etex_scantokens:D
656	_expl_primitive:NN	\showtokens	\etex_showtokens:D
657	_expl_primitive:NN	\readline	\etex_readline:D
658	_expl_primitive:NN	\tracingassigns	\etex_tracingassigns:D
659	_expl_primitive:NN	\tracingscantokens	\etex_tracingscantokens:D
660	_expl_primitive:NN	\tracingnesting	\etex_tracingnesting:D
661	_expl_primitive:NN	\tracingifs	\etex_tracingifs:D
662	_expl_primitive:NN	\currentiflevel	\etex_currentiflevel:D
663	_expl_primitive:NN	\currentifbranch	\etex_currentifbranch:D
664	_expl_primitive:NN	\currentifttype	\etex_currentifttype:D
665	_expl_primitive:NN	\tracinggroups	\etex_tracinggroups:D
666	_expl_primitive:NN	\currentgrouplevel	\etex_currentgrouplevel:D
667	_expl_primitive:NN	\currentgrouptype	\etex_currentgrouptype:D
668	_expl_primitive:NN	\showgroups	\etex_showgroups:D
669	_expl_primitive:NN	\showifs	\etex_showifs:D
670	_expl_primitive:NN	\interactionmode	\etex_interactionmode:D
671	_expl_primitive:NN	\lastnodetype	\etex_lastnodetype:D
672	_expl_primitive:NN	\iffontchar	\etex_iffontchar:D
673	_expl_primitive:NN	\fontcharht	\etex_fontcharht:D
674	_expl_primitive:NN	\fontcharhp	\etex_fontcharhp:D
675	_expl_primitive:NN	\fontcharwd	\etex_fontcharwd:D
676	_expl_primitive:NN	\fontcharic	\etex_fontcharic:D
677	_expl_primitive:NN	\parshapeindent	\etex_parshapeindent:D
678	_expl_primitive:NN	\parshapelength	\etex_parshapelength:D
679	_expl_primitive:NN	\parshapedimen	\etex_parshapedimen:D
680	_expl_primitive:NN	\numexpr	\etex_numexpr:D
681	_expl_primitive:NN	\dimexpr	\etex_dimexpr:D
682	_expl_primitive:NN	\glueexpr	\etex_glueexpr:D
683	_expl_primitive:NN	\muexpr	\etex_muexpr:D
684	_expl_primitive:NN	\gluestretch	\etex_gluestretch:D
685	_expl_primitive:NN	\glueshrink	\etex_glueshrink:D
686	_expl_primitive:NN	\gluestretchorder	\etex_gluestretchorder:D
687	_expl_primitive:NN	\glueshrinkorder	\etex_glueshrinkorder:D
688	_expl_primitive:NN	\gluetomu	\etex_gluetomu:D
689	_expl_primitive:NN	\mutoglu	\etex_mutoglu:D
690	_expl_primitive:NN	\lastlinefit	\etex_lastlinefit:D
691	_expl_primitive:NN	\interlinepenalties	\etex_interlinepenalties:D
692	_expl_primitive:NN	\clubpenalties	\etex_clubpenalties:D
693	_expl_primitive:NN	\widowpenalties	\etex_widowpenalties:D
694	_expl_primitive:NN	\displaywidowpenalties	\etex_displaywidowpenalties:D
695	_expl_primitive:NN	\middle	\etex_middle:D
696	_expl_primitive:NN	\savingshyphcodes	\etex_savingshyphcodes:D
697	_expl_primitive:NN	\savingsvdiscards	\etex_savingsvdiscards:D
698	_expl_primitive:NN	\pagediscards	\etex_pagediscards:D

699	<code>_expl_primitive:NN</code>	<code>\splitdiscards</code>	<code>\etex_splitdiscards:D</code>
700	<code>_expl_primitive:NN</code>	<code>\TeXeTstate</code>	<code>\etex_TeXeTstate:D</code>
701	<code>_expl_primitive:NN</code>	<code>\beginL</code>	<code>\etex_beginL:D</code>
702	<code>_expl_primitive:NN</code>	<code>\endL</code>	<code>\etex_endL:D</code>
703	<code>_expl_primitive:NN</code>	<code>\beginR</code>	<code>\etex_beginR:D</code>
704	<code>_expl_primitive:NN</code>	<code>\endR</code>	<code>\etex_endR:D</code>
705	<code>_expl_primitive:NN</code>	<code>\predisplaydirection</code>	<code>\etex_predisplaydirection:D</code>
706	<code>_expl_primitive:NN</code>	<code>\everyeof</code>	<code>\etex_everyeof:D</code>
707	<code>_expl_primitive:NN</code>	<code>\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.

708	<code>_expl_primitive:NN</code>	<code>\pdfcreationdate</code>	<code>\pdfTEX_pdfcreationdate:D</code>
709	<code>_expl_primitive:NN</code>	<code>\pdfcolorstack</code>	<code>\pdfTEX_pdfcolorstack:D</code>
710	<code>_expl_primitive:NN</code>	<code>\pdfcompresslevel</code>	<code>\pdfTEX_pdfcompresslevel:D</code>
711	<code>_expl_primitive:NN</code>	<code>\pdfdecimaldigits</code>	<code>\pdfTEX_pdfdecimaldigits:D</code>
712	<code>_expl_primitive:NN</code>	<code>\pdfhorigin</code>	<code>\pdfTEX_pdfhorigin:D</code>
713	<code>_expl_primitive:NN</code>	<code>\pdfinfo</code>	<code>\pdfTEX_pdfinfo:D</code>
714	<code>_expl_primitive:NN</code>	<code>\pdflastxform</code>	<code>\pdfTEX_pdflastxform:D</code>
715	<code>_expl_primitive:NN</code>	<code>\pdfliteral</code>	<code>\pdfTEX_pdfliteral:D</code>
716	<code>_expl_primitive:NN</code>	<code>\pdfminorversion</code>	<code>\pdfTEX_pdfminorversion:D</code>
717	<code>_expl_primitive:NN</code>	<code>\pdfobjcompresslevel</code>	<code>\pdfTEX_pdfobjcompresslevel:D</code>
718	<code>_expl_primitive:NN</code>	<code>\pdfoutput</code>	<code>\pdfTEX_pdfoutput:D</code>
719	<code>_expl_primitive:NN</code>	<code>\pdfrefxform</code>	<code>\pdfTEX_pdfrefxform:D</code>
720	<code>_expl_primitive:NN</code>	<code>\pdfrestore</code>	<code>\pdfTEX_pdfrestore:D</code>
721	<code>_expl_primitive:NN</code>	<code>\pdfsave</code>	<code>\pdfTEX_pdfsave:D</code>
722	<code>_expl_primitive:NN</code>	<code>\pdfsetmatrix</code>	<code>\pdfTEX_pdfsetmatrix:D</code>
723	<code>_expl_primitive:NN</code>	<code>\pdfpkresolution</code>	<code>\pdfTEX_pdfpkresolution:D</code>
724	<code>_expl_primitive:NN</code>	<code>\pdfTEXrevision</code>	<code>\pdfTEX_pdfTEXrevision:D</code>
725	<code>_expl_primitive:NN</code>	<code>\pdfvorigin</code>	<code>\pdfTEX_pdfvorigin:D</code>
726	<code>_expl_primitive:NN</code>	<code>\pdfxform</code>	<code>\pdfTEX_pdfxform:D</code>

While these are not.

727	<code>_expl_primitive:NN</code>	<code>\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`.

728	<code>_expl_primitive:NN</code>	<code>\XeTeXversion</code>	<code>\xetex_XeTeXversion:D</code>
-----	-----------------------------------	----------------------------	-------------------------------------

Primitives from LuaTeX.

729	<code>_expl_primitive:NN</code>	<code>\catcodetable</code>	<code>\luaTeX_catcodetable:D</code>
730	<code>_expl_primitive:NN</code>	<code>\directlua</code>	<code>\luaTeX_directlua:D</code>
731	<code>_expl_primitive:NN</code>	<code>\initcatcodetable</code>	<code>\luaTeX_initcatcodetable:D</code>
732	<code>_expl_primitive:NN</code>	<code>\latelua</code>	<code>\luaTeX_latelua:D</code>
733	<code>_expl_primitive:NN</code>	<code>\luaTeXversion</code>	<code>\luaTeX_luaTeXversion:D</code>
734	<code>_expl_primitive:NN</code>	<code>\savecatcodetable</code>	<code>\luaTeX_savecatcodetable:D</code>

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.

```

735 \__expl_primitive:NN \bodydir          \luatex_bodydir:D
736 \__expl_primitive:NN \mathdir         \luatex_mathdir:D
737 \__expl_primitive:NN \pagedir         \luatex_pagedir:D
738 \__expl_primitive:NN \pardir          \luatex_pardir:D
739 \__expl_primitive:NN \textdir         \luatex_textdir:D

```

The job is done: close the group (using the primitive renamed!).

```

740 \tex_endgroup:D
      LATEX 2ε will have moved a few primitives, so these are sorted out.
741 <*package>
742 \tex_let:D \tex_end:D                @@end
743 \tex_let:D \tex_everydisplay:D       \frozen@everydisplay
744 \tex_let:D \tex_everymath:D          \frozen@everymath
745 \tex_let:D \tex_hyphen:D              @hyph
746 \tex_let:D \tex_input:D               @input
747 \tex_let:D \tex_italiccorrection:D   @italiccorr
748 \tex_let:D \tex_underline:D           @underline

```

That is also true for the luatex package for L^AT_EX 2_ε.

```

749 \tex_let:D \luatex_catcodetable:D     \luatexcatcodetable
750 \tex_let:D \luatex_initcatcodetable:D \luatexinitcatcodetable
751 \tex_let:D \luatex_latelua:D          \luatexlatelua
752 \tex_let:D \luatex_savecatcodetable:D \luatexsavecatcodetable

```

Which also covers those slightly odd ones.

```

753 \tex_let:D \luatex_bodydir:D          \luatexbodydir
754 \tex_let:D \luatex_mathdir:D          \luatexmathdir
755 \tex_let:D \luatex_pagedir:D          \luatexpagedir
756 \tex_let:D \luatex_pardir:D           \luatexpardir
757 \tex_let:D \luatex_textdir:D          \luatextextdir
758 </package>
759 </initex | package>

```

3 l3basics implementation

```

760 <*initex | package>
761 <*package>
762 \ProvidesExplPackage
763   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
764 \__expl_package_check:
765 </package>

```

3.1 Renaming some TeX primitives (again)

Having given all the TeX primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but do a few now, just to get started.²

```

\if_true: Then some conditionals.
\if_false: 766 \tex_let:D \if_true:          \tex_iftrue:D
\or:       767 \tex_let:D \if_false:        \tex_iffalse:D
\else:     768 \tex_let:D \or:              \tex_or:D
\fi:       769 \tex_let:D \else:            \tex_else:D
\reverse_if:N 770 \tex_let:D \fi:            \tex_fi:D
\if:w      771 \tex_let:D \reverse_if:N      \etex_unless:D
\if_charcode:w 772 \tex_let:D \if:w          \tex_if:D
\if_catcode:w 773 \tex_let:D \if_charcode:w   \tex_if:D
\if_meaning:w 774 \tex_let:D \if_catcode:w    \tex_ifcat:D
              775 \tex_let:D \if_meaning:w    \tex_ifx:D

```

(End definition for \if_true: and others. These functions are documented on page 24.)

```

\if_mode_math: TeX lets us detect some if its modes.
\if_mode_horizontal: 776 \tex_let:D \if_mode_math:      \tex_ifmmode:D
\if_mode_vertical:   777 \tex_let:D \if_mode_horizontal: \tex_ifhmode:D
\if_mode_inner:      778 \tex_let:D \if_mode_vertical:   \tex_ifvmode:D
                    779 \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 780 \tex_let:D \if_cs_exist:N      \etex_ifdefined:D
\cs:w         781 \tex_let:D \if_cs_exist:w      \etex_ifcsname:D
\cs_end:      782 \tex_let:D \cs:w              \tex_csname:D
              783 \tex_let:D \cs_end:            \tex_endcsname:D

```

(End definition for \if_cs_exist:N and others. These functions are documented on page 17.)

```

\exp_after:wN The three \exp_ functions are used in the l3expan module where they are described.
\exp_not:N    784 \tex_let:D \exp_after:wN      \tex_expandafter:D
\exp_not:n    785 \tex_let:D \exp_not:N        \tex_noexpand:D
              786 \tex_let:D \exp_not:n          \etex_unexpanded:D

```

(End definition for \exp_after:wN, \exp_not:N, and \exp_not:n. These functions are documented on page 33.)

```

\token_to_meaning:N Examining a control sequence or token.
\token_to_str:N     787 \tex_let:D \token_to_meaning:N \tex_meaning:D
\cs_meaning:N       788 \tex_let:D \token_to_str:N   \tex_string:D
                    789 \tex_let:D \cs_meaning:N     \tex_meaning:D

```

(End definition for \token_to_meaning:N, \token_to_str:N, and \cs_meaning:N. These functions are documented on page 16.)

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.

`\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:`

```

790 \tex_let:D \scan_stop:      \tex_relax:D
791 \tex_let:D \group_begin:    \tex_begingroup:D
792 \tex_let:D \group_end:      \tex_endgroup:D

```

(End definition for `\scan_stop:`, `\group_begin:`, and `\group_end:`. These functions are documented on page 9.)

`\if_int_compare:w` For integers.

`__int_to_roman:w`

```

793 \tex_let:D \if_int_compare:w \tex_ifnum:D
794 \tex_let:D \__int_to_roman:w \tex_romannumeral:D

```

(End definition for `\if_int_compare:w` and `__int_to_roman:w`. These functions are documented on page 74.)

`\group_insert_after:N` Adding material after the end of a group.

```
795 \tex_let:D \group_insert_after:N \tex_aftergroup:D
```

(End definition for `\group_insert_after:N`. This function is documented on page 9.)

`\exp_args:Nc` Discussed in `l3expan`, but needed much earlier.

`\exp_args:cc`

```

796 \tex_long:D \tex_def:D \exp_args:Nc #1#2
797 { \exp_after:wN #1 \cs:w #2 \cs_end: }
798 \tex_long:D \tex_def:D \exp_args:cc #1#2
799 { \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:Nc`) are not defined at that point yet, but will be defined before those variants are used. The `\cs_meaning:c` command must check for an undefined control sequence to avoid defining it mistakenly.

```

\token_to_str:c
\cs_meaning:c
800 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
801 \tex_long:D \tex_def:D \cs_meaning:c #1
802 {
803   \if_cs_exist:w #1 \cs_end:
804     \exp_after:wN \use_i:nn
805   \else:
806     \exp_after:wN \use_ii:nn
807   \fi:
808   { \exp_args:Nc \cs_meaning:N {#1} }
809   { \tl_to_str:n {undefined} }
810 }
811 \tex_let:D \token_to_meaning:c = \cs_meaning:c

```

(End definition for `\token_to_meaning:c`, `\token_to_str:c`, and `\cs_meaning:c`. These functions are documented on page ??.)

3.2 Defining some constants

`\c_minus_one` We need the constants `\c_minus_one` and `\c_sixteen` now for writing information to the log and the terminal and `\c_zero` which is used by some functions in the `l3alloc` module.
`\c_zero`
`\c_sixteen` The rest are defined in the `l3int` module – at least for the ones that can be defined
`\c_six` with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the `l3int` module is
`\c_seven` required but it can't be used until the allocation has been set up properly! The actual
`\c_twelve` 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`.

```

812 <*package>
813 \tex_let:D \c_minus_one \m@ne
814 </package>
815 <*initex>
816 \tex_countdef:D \c_minus_one = 10 ~
817 \c_minus_one = -1 ~
818 </initex>
819 \tex_chardef:D \c_sixteen = 16 ~
820 \tex_chardef:D \c_zero = 0 ~
821 \tex_chardef:D \c_six = 6 ~
822 \tex_chardef:D \c_seven = 7 ~
823 \tex_chardef:D \c_twelve = 12 ~

```

(End definition for `\c_minus_one`, `\c_zero`, and `\c_sixteen`. These functions are documented on page 73.)

`\c_max_register_int` This is here as this particular integer is needed both in package mode and to bootstrap `l3alloc`, and is documented in `l3int`.

```

824 \etex_ifdefined:D \luatex luatexversion:D
825 \tex_chardef:D \c_max_register_int = 65 535 ~
826 \tex_else:D
827 \tex_mathchardef:D \c_max_register_int = 32 767 ~
828 \tex_fi:D

```

(End definition for `\c_max_register_int`. This variable is documented on page 73.)

3.3 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

`\cs_set_nopar:Npn` All assignment functions in \LaTeX 3 should be naturally protected; after all, the \TeX 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`

```

829 \tex_let:D \cs_set_nopar:Npn \tex_def:D
830 \tex_let:D \cs_set_nopar:Npx \tex_edef:D
831 \etex_protected:D \cs_set_nopar:Npn \cs_set:Npn
832 { \tex_long:D \cs_set_nopar:Npn }
833 \etex_protected:D \cs_set_nopar:Npn \cs_set:Npx
834 { \tex_long:D \cs_set_nopar:Npx }
835 \etex_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npn
836 { \etex_protected:D \cs_set_nopar:Npn }
837 \etex_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npx

```

```

838 { \etex_protected:D \cs_set_nopar:Npx }
839 \cs_set_protected_nopar:Npn \cs_set_protected:Npn
840 { \etex_protected:D \tex_long:D \cs_set_nopar:Npn }
841 \cs_set_protected_nopar:Npn \cs_set_protected:Npx
842 { \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
843 \tex_let:D \cs_gset_nopar:Npn \tex_gdef:D
844 \tex_let:D \cs_gset_nopar:Npx \tex_xdef:D
845 \cs_set_protected_nopar:Npn \cs_gset:Npn
846 { \tex_long:D \cs_gset_nopar:Npn }
847 \cs_set_protected_nopar:Npn \cs_gset:Npx
848 { \tex_long:D \cs_gset_nopar:Npx }
849 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npn
850 { \etex_protected:D \cs_gset_nopar:Npn }
851 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npx
852 { \etex_protected:D \cs_gset_nopar:Npx }
853 \cs_set_protected_nopar:Npn \cs_gset_protected:Npn
854 { \etex_protected:D \tex_long:D \cs_gset_nopar:Npn }
855 \cs_set_protected_nopar:Npn \cs_gset_protected:Npx
856 { \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.

```

857 \cs_set_nopar:Npn \l__exp_internal_tl { }

```

(End definition for `\l__exp_internal_tl`. This variable is documented on page 34.)

\use:c This macro grabs its argument and returns a csname from it.

```

858 \cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }

```

(End definition for `\use:c`. This function is documented on page 17.)

\use:x Fully expands its argument and passes it to the input stream. Uses the reserved `\l__exp_internal_tl` which will be set up in `l3expan`.

```

859 \cs_set_protected:Npn \use:x #1
860 {
861   \cs_set_nopar:Npx \l__exp_internal_tl {#1}
862   \l__exp_internal_tl
863 }

```

(End definition for `\use:x`. This function is documented on page 20.)

`\use:n` These macros grab their arguments and returns them back to the input (with outer braces removed).

```
\use:nnn 864 \cs_set:Npn \use:n #1 {#1}
\use:nnnn 865 \cs_set:Npn \use:nn #1#2 {#1#2}
866 \cs_set:Npn \use:nnn #1#2#3 {#1#2#3}
867 \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 868 \cs_set:Npn \use_i:nn #1#2 {#1}
869 \cs_set:Npn \use_ii:nn #1#2 {#2}
```

(End definition for `\use_i:nn` and `\use_ii:nn`. These functions are documented on page 19.)

`\use_i:nnnn` We also need something for picking up arguments from a longer list.

```
\use_ii:nnnn 870 \cs_set:Npn \use_i:nnnn #1#2#3 {#1}
\use_iii:nnnn 871 \cs_set:Npn \use_ii:nnnn #1#2#3 {#2}
\use_i_ii:nnnn 872 \cs_set:Npn \use_iii:nnnn #1#2#3 {#3}
\use_i:nnnnn 873 \cs_set:Npn \use_i_ii:nnnn #1#2#3 {#1#2}
\use_ii:nnnnn 874 \cs_set:Npn \use_i:nnnnn #1#2#3#4 {#1}
\use_iii:nnnnn 875 \cs_set:Npn \use_ii:nnnnn #1#2#3#4 {#2}
\use_iv:nnnnn 876 \cs_set:Npn \use_iii:nnnnn #1#2#3#4 {#3}
877 \cs_set:Npn \use_iv:nnnnn #1#2#3#4 {#4}
```

(End definition for `\use_i:nnnn` and others. These functions are documented on page 19.)

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.

```
\use_none_delimit_by_q_stop:w 878 \cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
\use_none_delimit_by_q_recursion_stop:w 879 \cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
880 \cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }
```

(End definition for `\use_none_delimit_by_q_nil:w`, `\use_none_delimit_by_q_stop:w`, and `\use_none_delimit_by_q_recursion_stop:w`. These functions are documented on page 47.)

`\use_i_delimit_by_q_nil:nw` Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

```
\use_i_delimit_by_q_stop:nw 881 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
\use_i_delimit_by_q_recursion_stop:nw 882 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
883 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw #1#2 \q_recursion_stop {#1}
```

(End definition for `\use_i_delimit_by_q_nil:nw`, `\use_i_delimit_by_q_stop:nw`, and `\use_i_delimit_by_q_recursion_stop:nw`. These functions are documented on page 47.)

3.5 Gobbling tokens from input

`\use_none:n` To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of `n`'s following the `:` in the name. Although we could define functions to remove ten arguments or more using separate calls of `\use_none:nnnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once will take care of gobbling all the tokens in one go.

<code>\use_none:n</code>	884	<code>\cs_set:Npn \use_none:n</code>	<code>#1</code>	<code>{ }</code>
<code>\use_none:nn</code>	885	<code>\cs_set:Npn \use_none:nn</code>	<code>#1#2</code>	<code>{ }</code>
<code>\use_none:nnn</code>	886	<code>\cs_set:Npn \use_none:nnn</code>	<code>#1#2#3</code>	<code>{ }</code>
<code>\use_none:nnnn</code>	887	<code>\cs_set:Npn \use_none:nnnn</code>	<code>#1#2#3#4</code>	<code>{ }</code>
<code>\use_none:nnnnn</code>	888	<code>\cs_set:Npn \use_none:nnnnn</code>	<code>#1#2#3#4#5</code>	<code>{ }</code>
<code>\use_none:nnnnnn</code>	889	<code>\cs_set:Npn \use_none:nnnnnn</code>	<code>#1#2#3#4#5#6</code>	<code>{ }</code>
<code>\use_none:nnnnnnn</code>	890	<code>\cs_set:Npn \use_none:nnnnnnn</code>	<code>#1#2#3#4#5#6#7</code>	<code>{ }</code>
<code>\use_none:nnnnnnnn</code>	891	<code>\cs_set:Npn \use_none:nnnnnnnn</code>	<code>#1#2#3#4#5#6#7#8</code>	<code>{ }</code>
<code>\use_none:nnnnnnnnn</code>	892	<code>\cs_set:Npn \use_none:nnnnnnnnn</code>	<code>#1#2#3#4#5#6#7#8#9</code>	<code>{ }</code>

(End definition for `\use_none:n` and others. These functions are documented on page ??.)

3.6 Conditional processing and definitions

Underneath any predicate function (`_p`) or other conditional forms (TF, etc.) is a built-in logic saying that it after all of the testing and processing must return the *state* this leaves `TEX` in. Therefore, a simple user interface could be something like

```

\if_meaning:w #1#2
  \prg_return_true:
\else:
  \if_meaning:w #1#3
    \prg_return_true:
  \else:
    \prg_return_false:
  \fi:
\fi:

```

Usually, a `TEX` programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the `TEX` programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

`\prg_return_true:` The idea here is that `__int_to_roman:w` will expand fully any `\else:` and the `\fi:` that are waiting to be discarded, before reaching the `\c_zero` which will leave the expansion null. The code can then leave either the first or second argument in the input stream. This means that all of the branching code has to contain at least two tokens: see how the logical tests are actually implemented to see this.

```

893 \cs_set_nopar:Npn \prg_return_true:
894 { \exp_after:wN \use_i:nn \__int_to_roman:w }
895 \cs_set_nopar:Npn \prg_return_false:
896 { \exp_after:wN \use_ii:nn \__int_to_roman:w }

```

An extended state space could be implemented by including a more elaborate function in place of `\use_i:nn/\use_ii:nn`. Provided two arguments are absorbed then the code will work.

(End definition for `\prg_return_true:` and `\prg_return_false:`. These functions are documented on page 37.)

```
\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
  \prg_set_protected_conditional:Npnn
  \prg_new_protected_conditional:Npnn
  \prg_generate_conditional_parm:nnNpnn
```

The user functions for the types using parameter text from the programmer. The various functions only differ by which function is used for the assignment. For those `Npnn` type functions, we must grab the parameter text, reading everything up to a left brace before continuing. Then split the base function into name and signature, and feed $\{\langle name \rangle\}$ $\{\langle signature \rangle\}$ $\langle boolean \rangle$ $\{\langle set \text{ or } new \rangle\}$ $\{\langle maybe \text{ protected} \rangle\}$ $\{\langle parameters \rangle\}$ $\{\text{TF}, \dots\}$ $\{\langle code \rangle\}$ to the auxiliary function responsible for defining all conditionals.

```
897 \cs_set_protected_nopar:Npn \prg_set_conditional:Npnn
898   { \prg_generate_conditional_parm:nnNpnn { set } { } }
899 \cs_set_protected_nopar:Npn \prg_new_conditional:Npnn
900   { \prg_generate_conditional_parm:nnNpnn { new } { } }
901 \cs_set_protected_nopar:Npn \prg_set_protected_conditional:Npnn
902   { \prg_generate_conditional_parm:nnNpnn { set } { _protected } }
903 \cs_set_protected_nopar:Npn \prg_new_protected_conditional:Npnn
904   { \prg_generate_conditional_parm:nnNpnn { new } { _protected } }
905 \cs_set_protected:Npn \prg_generate_conditional_parm:nnNpnn #1#2#3#4#
906   {
907     \cs_split_function:NN #3 \prg_generate_conditional:nnNnnnnn
908     {#1} {#2} {#4}
909   }
```

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 35.)

```
\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
  \prg_set_protected_conditional:Nnn
  \prg_new_protected_conditional:Nnn
  \prg_generate_conditional_count:nnNnn
  \prg_generate_conditional_count:nnNnnnn
```

The user functions for the types automatically inserting the correct parameter text based on the signature. The various functions only differ by which function is used for the assignment. Split the base function into name and signature. The second auxiliary generates the parameter text from the number of letters in the signature. Then feed $\{\langle name \rangle\}$ $\{\langle signature \rangle\}$ $\langle boolean \rangle$ $\{\langle set \text{ or } new \rangle\}$ $\{\langle maybe \text{ protected} \rangle\}$ $\{\langle parameters \rangle\}$ $\{\text{TF}, \dots\}$ $\{\langle code \rangle\}$ to the auxiliary function responsible for defining all conditionals. If the $\langle signature \rangle$ has more than 9 letters, the definition is aborted since \TeX macros have at most 9 arguments. The erroneous case where the function name contains no colon is captured later.

```
910 \cs_set_protected_nopar:Npn \prg_set_conditional:Nnn
911   { \prg_generate_conditional_count:nnNnn { set } { } }
912 \cs_set_protected_nopar:Npn \prg_new_conditional:Nnn
913   { \prg_generate_conditional_count:nnNnn { new } { } }
914 \cs_set_protected_nopar:Npn \prg_set_protected_conditional:Nnn
915   { \prg_generate_conditional_count:nnNnn { set } { _protected } }
916 \cs_set_protected_nopar:Npn \prg_new_protected_conditional:Nnn
917   { \prg_generate_conditional_count:nnNnn { new } { _protected } }
918 \cs_set_protected:Npn \prg_generate_conditional_count:nnNnn #1#2#3
919   {
920     \cs_split_function:NN #3 \prg_generate_conditional_count:nnNnnnn
```

```

921     {#1} {#2}
922   }
923   \cs_set_protected:Npn \__prg_generate_conditional_count:nnNnnnn #1#2#3#4#5
924   {
925     \__cs_parm_from_arg_count:nnF
926     { \__prg_generate_conditional:nnNnnnnn {#1} {#2} #3 {#4} {#5} }
927     { \tl_count:n {#2} }
928     {
929       \__msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
930       { \token_to_str:c { #1 : #2 } }
931       { \tl_count:n {#2} }
932       \use_none:nn
933     }
934   }

```

(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.

```

935   \cs_set_protected:Npn \__prg_generate_conditional:nnNnnnnn #1#2#3#4#5#6#7#8
936   {
937     \if_meaning:w \c_false_bool #3
938     \__msg_kernel_error:nnx { kernel } { missing-colon }
939     { \token_to_str:c {#1} }
940     \exp_after:wN \use_none:nn
941     \fi:
942     \use:x
943     {
944       \exp_not:N \__prg_generate_conditional:nnnnnnnw
945       \exp_not:n { {#4} {#5} {#1} {#2} {#6} {#8} }
946       \etex_detokenize:D {#7}
947       \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
948     }
949   }

```

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.

```

950   \cs_set_protected:Npn \__prg_generate_conditional:nnnnnnnw #1#2#3#4#5#6#7 ,
951   {
952     \if_meaning:w \q_recursion_tail #7

```

```

953     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
954   \fi:
955   \use:c { __prg_generate_ #7 _form:wnnnnnn }
956     \tl_if_empty:nF {#7}
957     {
958       \__msg_kernel_error:nxxx
959       { kernel } { conditional-form-unknown }
960       {#7} { \token_to_str:c { #3 : #4 } }
961     }
962     \use_none:nnnnnnn
963   \q_stop
964   {#1} {#2} {#3} {#4} {#5} {#6}
965   \__prg_generate_conditional:nnnnnnw {#1} {#2} {#3} {#4} {#5} {#6}
966 }

```

(End definition for __prg_generate_conditional:nnNnnnnn and __prg_generate_conditional:nnnnnnw.)

__prg_generate_p_form:wnnnnnnn
 __prg_generate_TF_form:wnnnnnnn
 __prg_generate_T_form:wnnnnnnn
 __prg_generate_F_form:wnnnnnnn

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.

```

967 \cs_set_protected:Npn \__prg_generate_p_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
968 {
969   \if_meaning:w \scan_stop: #3 \scan_stop:
970     \exp_after:wN \use_i:nn
971   \else:
972     \exp_after:wN \use_ii:nn
973   \fi:
974   {
975     \exp_args:cc { cs_ #2 #3 :Npn } { #4 _p: #5 } #6
976     { #7 \c_zero \c_true_bool \c_false_bool }
977   }
978   {
979     \__msg_kernel_error:nmx { kernel } { protected-predicate }
980     { \token_to_str:c { #4 _p: #5 } }
981   }
982 }
983 \cs_set_protected:Npn \__prg_generate_T_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
984 {
985   \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 T } #6
986   { #7 \c_zero \use:n \use_none:n }
987 }
988 \cs_set_protected:Npn \__prg_generate_F_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
989 {
990   \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 F } #6
991   { #7 \c_zero { } }
992 }

```

```

993 \cs_set_protected:Npn \__prg_generate_TF_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
994 {
995   \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 TF } #6
996   { #7 \c_zero }
997 }
(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

```

998 \cs_set_protected_nopar:Npn \prg_set_eq_conditional:NNn
999 { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
1000 \cs_set_protected_nopar:Npn \prg_new_eq_conditional:NNn
1001 { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
1002 \cs_set_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
1003 {
1004   \use:x
1005   {
1006     \exp_not:N \__prg_set_eq_conditional:nnNnnNNw
1007     \__cs_split_function:NN #2 \prg_do_nothing:
1008     \__cs_split_function:NN #3 \prg_do_nothing:
1009     \exp_not:N #1
1010     \etex_detokenize:D {#4}
1011     \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
1012   }
1013 }

```

(End definition for \prg_set_eq_conditional:NNn and \prg_new_eq_conditional:NNn. These functions are documented on page 37.)

_prg_set_eq_conditional:nnNnnNNw Split the function to be defined, and setup a manual clist loop over argument #6 of the
 _prg_set_eq_conditional_loop:nnnnNw first auxiliary. The second auxiliary receives twice three arguments coming from splitting
 _prg_set_eq_conditional_p_form:nnn the function to be defined and the function to copy. Make sure that both functions
 _prg_set_eq_conditional_TF_form:nnn contained a colon, otherwise we don't know how to build conditionals, hence abort. Call
 _prg_set_eq_conditional_T_form:nnn the looping macro, with arguments $\{\langle name_1 \rangle\} \{\langle signature_1 \rangle\} \{\langle name_2 \rangle\} \{\langle signature_2 \rangle\}$
 _prg_set_eq_conditional_F_form:nnn $\langle copying function \rangle$ and followed by the comma list. At each step in the loop, make sure
 that the conditional form we copy is defined, and copy it, otherwise abort.

```

1014 \cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
1015 {
1016   \if_meaning:w \c_false_bool #3
1017   \__msg_kernel_error:nnx { kernel } { missing-colon }
1018   { \token_to_str:c {#1} }
1019   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1020   \fi:
1021   \if_meaning:w \c_false_bool #6
1022   \__msg_kernel_error:nnx { kernel } { missing-colon }
1023   { \token_to_str:c {#4} }
1024   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1025   \fi:
1026   \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}

```

```

1027 }
1028 \cs_set_protected:Npn \__prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
1029 {
1030   \if_meaning:w \q_recursion_tail #6
1031   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1032   \fi:
1033   \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
1034   \tl_if_empty:nF {#6}
1035   {
1036     \__msg_kernel_error:nnxx
1037     { kernel } { conditional-form-unknown }
1038     {#6} { \token_to_str:c { #1 : #2 } }
1039   }
1040   \use_none:nnnnnn
1041   \q_stop
1042   #5 {#1} {#2} {#3} {#4}
1043   \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
1044 }
1045 \cs_set:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \q_stop #2#3#4#5#6
1046 {
1047   \__chk_if_exist_cs:c { #5 _p : #6 }
1048   #2 { #3 _p : #4 } { #5 _p : #6 }
1049 }
1050 \cs_set:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \q_stop #2#3#4#5#6
1051 {
1052   \__chk_if_exist_cs:c { #5 : #6 TF }
1053   #2 { #3 : #4 TF } { #5 : #6 TF }
1054 }
1055 \cs_set:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \q_stop #2#3#4#5#6
1056 {
1057   \__chk_if_exist_cs:c { #5 : #6 T }
1058   #2 { #3 : #4 T } { #5 : #6 T }
1059 }
1060 \cs_set:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \q_stop #2#3#4#5#6
1061 {
1062   \__chk_if_exist_cs:c { #5 : #6 F }
1063   #2 { #3 : #4 F } { #5 : #6 F }
1064 }

```

(End definition for `__prg_set_eq_conditional:nnNnnNNw` and `__prg_set_eq_conditional_loop:nnnnNw`. These functions are documented on page 37.)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either TT or TF to be tested using `\if:w` but this was later changed to 00 and 01, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

`\c_true_bool` Here are the canonical boolean values.

`\c_false_bool` 1065 `\tex_chardef:D \c_true_bool = 1 ~`

```
1066 \tex_chardef:D \c_false_bool = 0 ~
```

(End definition for `\c_true_bool` and `\c_false_bool`. These variables are documented on page 21.)

3.7 Dissecting a control sequence

```
\cs_to_str:N This converts a control sequence into the character string of its name, removing the
__cs_to_str:N leading escape character. This turns out to be a non-trivial matter as there are different
__cs_to_str:w cases:
```

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of the `\escapechar` is negative;
- when the escape character is a space.

One approach to solve this is to test how many tokens result from `\token_to_str:N \a`. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, `\token_to_str:N\` yields the escape character itself and a space. The character codes are different, thus the `\if:w` test is false, and TeX reads `__cs_to_str:N` after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial `__int_to_roman:w`. The second case is that the escape character is not printable. Then the `\if:w` test is unfinished after reading a the space from `\token_to_str:N\` , and the auxiliary `__cs_to_str:w` is expanded, feeding `-` as a second character for the test; the test is false, and TeX skips to `\fi:`, then performs `\token_to_str:N`, and stops the `__int_to_roman:w` with `\c_zero`. The last case is that the escape character is itself a space. In this case, the `\if:w` test is true, and the auxiliary `__cs_to_str:w` comes into play, inserting `-__int_value:w`, which expands `\c_zero` to the character 0. The initial `__int_to_roman:w` then sees 0, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the argument of `__int_to_roman:w`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```
1067 \cs_set:nopar:Npn \cs_to_str:N
1068 {
1069   \__int_to_roman:w
1070   \if:w \token_to_str:N \ \__cs_to_str:w \fi:
1071   \exp_after:wN \__cs_to_str:N \token_to_str:N
1072 }
1073 \cs_set:Npn \__cs_to_str:N #1 { \c_zero }
1074 \cs_set:Npn \__cs_to_str:w #1 \__cs_to_str:N
1075 { - \__int_value:w \fi: \exp_after:wN \c_zero }
```

(End definition for `\cs_to_str:N`. This function is documented on page 18.)


```

\__cs_split_function:NN
\__cs_split_function_auxi:w
\__cs_split_function_auxii:w

```

This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean $\langle true \rangle$ or $\langle false \rangle$ is returned with $\langle true \rangle$ for when there is a colon in the function and $\langle false \rangle$ if there is not. Lastly, the second argument of `__cs_split_function:NN` is supposed to be a function taking three variables, one for name, one for signature, and one for the boolean. For example, `__cs_split_function:NN \foo_bar:cnx \use_i:nnn` as input becomes `\use_i:nnn {foo_bar} {cnx} \c_true_bool`.

We can't use a literal `:` because it has the wrong catcode here, so it's transformed from `@` with `\tex_lowercase:D`.

First ensure that we actually get a properly evaluated string by expanding `\cs_to_str:N` twice. If the function contained a colon, the auxiliary takes as `#1` the function name, delimited by the first colon, then the signature `#2`, delimited by `\q_mark`, then `\c_true_bool` as `#3`, and `#4` cleans up until `\q_stop`. Otherwise, the `#1` contains the function name and `\q_mark \c_true_bool`, `#2` is empty, `#3` is `\c_false_bool`, and `#4` cleans up. In both cases, `#5` is the $\langle processor \rangle$. The second auxiliary trims the trailing `\q_mark` from the function name if present (that is, if the original function had no colon).

```

1076 \group_begin:
1077 \tex_lccode:D '\@ = '\: \scan_stop:
1078 \tex_catcode:D '\@ = 12 ~
1079 \tex_lowercase:D
1080 {
1081   \group_end:
1082   \cs_set:Npn \__cs_split_function:NN #1
1083     {
1084       \exp_after:wN \exp_after:wN
1085       \exp_after:wN \__cs_split_function_auxi:w
1086       \cs_to_str:N #1 \q_mark \c_true_bool
1087       @ \q_mark \c_false_bool
1088       \q_stop
1089     }
1090   \cs_set:Npn \__cs_split_function_auxi:w #1 @ #2 \q_mark #3#4 \q_stop #5
1091     { \__cs_split_function_auxii:w #5 #1 \q_mark \q_stop {#2} #3 }
1092   \cs_set:Npn \__cs_split_function_auxii:w #1#2 \q_mark #3 \q_stop
1093     { #1 {#2} }
1094 }

```

(End definition for `__cs_split_function:NN`. This function is documented on page 25.)

```

\__cs_get_function_name:N
\__cs_get_function_signature:N

```

Simple wrappers.

```

1095 \cs_set:Npn \__cs_get_function_name:N #1
1096 { \__cs_split_function:NN #1 \use_i:nnn }
1097 \cs_set:Npn \__cs_get_function_signature:N #1
1098 { \__cs_split_function:NN #1 \use_ii:nnn }

```

(End definition for `__cs_get_function_name:N` and `__cs_get_function_signature:N`.)

3.8 Exist or free

A control sequence is said to *exist* (to be used) if has an entry in the hash table and its meaning is different from the primitive `\relax` token. A control sequence is said to be

free (to be defined) if it does not already exist.

\cs_if_exist_p:N Two versions for checking existence. For the N form we firstly check for `\scan_stop:` and
\cs_if_exist_p:c then if it is in the hash table. There is no problem when inputting something like `\else:`
\cs_if_exist:N \overline{TF} or `\fi:` as T_EX will only ever skip input in case the token tested against is `\scan_stop:`.
\cs_if_exist:c \overline{TF}

```

1099 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1100 {
1101   \if_meaning:w #1 \scan_stop:
1102     \prg_return_false:
1103   \else:
1104     \if_cs_exist:N #1
1105       \prg_return_true:
1106     \else:
1107       \prg_return_false:
1108     \fi:
1109   \fi:
1110 }

```

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.

```

1111 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1112 {
1113   \if_cs_exist:w #1 \cs_end:
1114     \exp_after:wN \use_i:nn
1115   \else:
1116     \exp_after:wN \use_ii:nn
1117   \fi:
1118   {
1119     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1120     \prg_return_false:
1121   \else:
1122     \prg_return_true:
1123   \fi:
1124   }
1125   \prg_return_false:
1126 }

```

(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.

```

1127 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
1128 {
1129   \if_meaning:w #1 \scan_stop:
1130     \prg_return_true:
1131   \else:
1132     \if_cs_exist:N #1
1133       \prg_return_false:

```

```

1134     \else:
1135         \prg_return_true:
1136     \fi:
1137 \fi:
1138 }
1139 \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1140 {
1141     \if_cs_exist:w #1 \cs_end:
1142     \exp_after:wN \use_i:nn
1143 \else:
1144     \exp_after:wN \use_ii:nn
1145 \fi:
1146 {
1147     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1148     \prg_return_true:
1149 \else:
1150     \prg_return_false:
1151 \fi:
1152 }
1153 { \prg_return_true: }
1154 }

```

(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.

```

1155 \cs_set:Npn \cs_if_exist_use:NTF #1#2
1156 { \cs_if_exist:NTF #1 { #1 #2 } }
1157 \cs_set:Npn \cs_if_exist_use:NF #1
1158 { \cs_if_exist:NTF #1 { #1 } }
1159 \cs_set:Npn \cs_if_exist_use:NT #1 #2
1160 { \cs_if_exist:NTF #1 { #1 #2 } { } }
1161 \cs_set:Npn \cs_if_exist_use:N #1
1162 { \cs_if_exist:NTF #1 { #1 } { } }
1163 \cs_set:Npn \cs_if_exist_use:cTF #1#2
1164 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
1165 \cs_set:Npn \cs_if_exist_use:cF #1
1166 { \cs_if_exist:cTF {#1} { \use:c {#1} } }
1167 \cs_set:Npn \cs_if_exist_use:cT #1#2
1168 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
1169 \cs_set:Npn \cs_if_exist_use:c #1
1170 { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }

```

(End definition for \cs_if_exist_use:N and \cs_if_exist_use:c. These functions are documented on page ??.)

3.9 Defining and checking (new) functions

We provide two kinds of functions that can be used to define control sequences. On the one hand we have functions that check if their argument doesn't already exist, they are called `\..._new`. The second type of defining functions doesn't check if the argument is already defined.

Before we can define them, we need some auxiliary macros that allow us to generate error messages. The definitions here are only temporary, they will be redefined later on.

`\iow_log:x` We define a routine to write only to the log file. And a similar one for writing to both
`\iow_term:x` the log file and the terminal. These will be redefined later by `l3io`.

```
1171 \cs_set_protected_nopar:Npn \iow_log:x
1172 { \tex_immediate:D \tex_write:D \c_minus_one }
1173 \cs_set_protected_nopar:Npn \iow_term:x
1174 { \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.

```
1175 \cs_set_protected:Npn \__msg_kernel_error:nxxx #1#2#3#4
1176 {
1177   \tex_errmessage:D
1178   {
1179     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
1180     Argh,~internal~LaTeX3~error! ^^J ^^J
1181     Module ~ #1 , ~ message-name~"#2": ^^J
1182     Arguments~'#3'~and~'#4' ^^J ^^J
1183     This~is~one~for~The~LaTeX3~Project:~bailing~out
1184   }
1185   \tex_end:D
1186 }
1187 \cs_set_protected:Npn \__msg_kernel_error:nxx #1#2#3
1188 { \__msg_kernel_error:nxxx {#1} {#2} {#3} { } }
1189 \cs_set_protected:Npn \__msg_kernel_error:nn #1#2
1190 { \__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.

```
1191 \cs_set_nopar:Npn \msg_line_context:
1192 { on~line~ \tex_the:D \tex_inputlineno:D }
```

(End definition for `\msg_line_context:`. This function is documented on page 142.)

`__chk_if_free_cs:N` This command is called by `\cs_new_nopar:Npn` and `\cs_new_eq:NN` etc. to make sure
`__chk_if_free_cs:c` that the argument sequence is not already in use. If it is, an error is signalled. It checks if `<cname>` 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!

```

1193 \cs_set_protected:Npn \__chk_if_free_cs:N #1
1194 {
1195   \cs_if_free:NF #1
1196   {
1197     \__msg_kernel_error:nxxx { kernel } { command-already-defined }
1198     { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1199   }
1200 }
1201 <*package>
1202 \tex_ifodd:D \l@expl@log@functions@bool
1203 \cs_set_protected:Npn \__chk_if_free_cs:N #1
1204 {
1205   \cs_if_free:NF #1
1206   {
1207     \__msg_kernel_error:nxxx { kernel } { command-already-defined }
1208     { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1209   }
1210   \iow_log:x { Defining~\token_to_str:N #1~ \msg_line_context: }
1211 }
1212 \fi:
1213 </package>
1214 \cs_set_protected_nopar:Npn \__chk_if_free_cs:c
1215 { \exp_args:Nc \__chk_if_free_cs:N }
(End definition for \__chk_if_free_cs:N and \__chk_if_free_cs:c.)

```

__chk_if_exist_cs:N This function issues an error message when the control sequence in its argument does not exist.

__chk_if_exist_cs:c

```

1216 \cs_set_protected:Npn \__chk_if_exist_cs:N #1
1217 {
1218   \cs_if_exist:NF #1
1219   {
1220     \__msg_kernel_error:nxx { kernel } { command-not-defined }
1221     { \token_to_str:N #1 }
1222   }
1223 }
1224 \cs_set_protected_nopar:Npn \__chk_if_exist_cs:c
1225 { \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
1226 \cs_set:Npn \__cs_tmp:w #1#2
1227 {
1228   \cs_set_protected:Npn #1 ##1
1229   {
1230     \__chk_if_free_cs:N ##1
1231     #2 ##1
1232   }

```

```

1233 }
1234 \__cs_tmp:w \cs_new_nopar:Npn \cs_gset_nopar:Npn
1235 \__cs_tmp:w \cs_new_nopar:Npx \cs_gset_nopar:Npx
1236 \__cs_tmp:w \cs_new:Npn \cs_gset:Npn
1237 \__cs_tmp:w \cs_new:Npx \cs_gset:Npx
1238 \__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1239 \__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1240 \__cs_tmp:w \cs_new_protected:Npn \cs_gset_protected:Npn
1241 \__cs_tmp:w \cs_new_protected:Npx \cs_gset_protected:Npx

```

(End definition for \cs_new_nopar:Npn and others. These functions are documented on page ??.)

\cs_set_nopar:cpn Like \cs_set_nopar:Npn and \cs_new_nopar:Npn, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the c stands for csname argument, see the expansion module). Global versions are also provided.

\cs_set_nopar:cpn \cs_set_nopar:cpn⟨string⟩⟨rep-text⟩ will turn ⟨string⟩ into a csname and then assign ⟨rep-text⟩ to it by using \cs_set_nopar:Npn. This means that there might be a parameter string between the two arguments.

```

1242 \cs_set:Npn \__cs_tmp:w #1#2
1243 { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
1244 \__cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
1245 \__cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
1246 \__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1247 \__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1248 \__cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
1249 \__cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

(End definition for \cs_set_nopar:cpn and others. These functions are documented on page ??.)

\cs_set:cpn Variants of the \cs_set:Npn versions which make a csname out of the first arguments.

\cs_set:cpx We may also do this globally.

```

\cs_gset:cpn 1250 \__cs_tmp:w \cs_set:cpn \cs_set:Npn
\cs_gset:cpx 1251 \__cs_tmp:w \cs_set:cpx \cs_set:Npx
\cs_new:cpn 1252 \__cs_tmp:w \cs_gset:cpn \cs_gset:Npn
\cs_new:cpx 1253 \__cs_tmp:w \cs_gset:cpx \cs_gset:Npx
1254 \__cs_tmp:w \cs_new:cpn \cs_new:Npn
1255 \__cs_tmp:w \cs_new:cpx \cs_new:Npx

```

(End definition for \cs_set:cpn and others. These functions are documented on page ??.)

\cs_set_protected_nopar:cpn Variants of the \cs_set_protected_nopar:Npn versions which make a csname out of the first arguments. We may also do this globally.

```

\cs_gset_protected_nopar:cpn 1256 \__cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
\cs_gset_protected_nopar:cpx 1257 \__cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
\cs_new_protected_nopar:cpn 1258 \__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
\cs_new_protected_nopar:cpx 1259 \__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1260 \__cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
1261 \__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 arguments. We may also do this globally.
`\cs_set_protected:cpx`
`\cs_gset_protected:cpn` 1262 `__cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn`
`\cs_gset_protected:cpx` 1263 `__cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx`
`\cs_new_protected:cpn` 1264 `__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn`
`\cs_new_protected:cpx` 1265 `__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx`
1266 `__cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn`
1267 `__cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx`
(End definition for \cs_set_protected:cpn and others. These functions are documented on page ??.)

3.11 Copying definitions

`\cs_set_eq:NN` These macros allow us to copy the definition of a control sequence to another control sequence.
`\cs_set_eq:cN` The = sign allows us to define funny char tokens like = itself or `_` with this function.
`\cs_set_eq:Nc` For the definition of `\c_space_char{~}` to work we need the ~ after the =.
`\cs_set_eq:cc`
`\cs_gset_eq:NN` `\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While
`\cs_gset_eq:cN` `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it
`\cs_gset_eq:Nc` long in order to throw an “already defined” error rather than “runaway argument”.
`\cs_gset_eq:cc` 1268 `\cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }`
`\cs_new_eq:NN` 1269 `\cs_new_protected_nopar:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }`
`\cs_new_eq:cN` 1270 `\cs_new_protected_nopar:Npn \cs_set_eq:Nc { \exp_args:Nnc \cs_set_eq:NN }`
`\cs_new_eq:Nc` 1271 `\cs_new_protected_nopar:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }`
`\cs_new_eq:cc` 1272 `\cs_new_protected_nopar:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }`
1273 `\cs_new_protected_nopar:Npn \cs_gset_eq:Nc { \exp_args:Nnc \cs_gset_eq:NN }`
1274 `\cs_new_protected_nopar:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }`
1275 `\cs_new_protected_nopar:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }`
1276 `\cs_new_protected:Npn \cs_new_eq:NN #1`
1277 `{`
1278 `_chk_if_free_cs:N #1`
1279 `\tex_global:D \cs_set_eq:NN #1`
1280 `}`
1281 `\cs_new_protected_nopar:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }`
1282 `\cs_new_protected_nopar:Npn \cs_new_eq:Nc { \exp_args:Nnc \cs_new_eq:NN }`
1283 `\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 Undefined 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.
1284 `\cs_new_protected:Npn \cs_undefine:N #1`
1285 `{ \cs_gset_eq:NN #1 \tex_undefined:D }`
1286 `\cs_new_protected:Npn \cs_undefine:c #1`
1287 `{`

```

1288 \if_cs_exist:w #1 \cs_end:
1289 \exp_after:wN \use:n
1290 \else:
1291 \exp_after:wN \use_none:n
1292 \fi:
1293 { \cs_gset_eq:cN {#1} \tex_undefined:D }
1294 }

```

(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` `_cs_parm_from_arg_count_test:nnF` L^AT_EX3 provides shorthands to define control sequences and conditionals with a simple parameter text, derived directly from the signature, or more generally from knowing the number of arguments, between 0 and 9. This function expands to its first argument, untouched, followed by a brace group containing the parameter text `{#1...#n}`, where n is the result of evaluating the second argument (as described in `\int_eval:n`). If the second argument gives a result outside the range $[0, 9]$, the third argument is returned instead, normally an error message. Some of the functions use here are not defined yet, but will be defined before this function is called.

```

1295 \cs_set_protected:Npn \_cs_parm_from_arg_count:nnF #1#2
1296 {
1297   \exp_args:Nx \_cs_parm_from_arg_count_test:nnF
1298   {
1299     \exp_after:wN \exp_not:n
1300     \if_case:w \_int_eval:w #2 \_int_eval_end:
1301       { }
1302       \or: { ##1 }
1303       \or: { ##1##2 }
1304       \or: { ##1##2##3 }
1305       \or: { ##1##2##3##4 }
1306       \or: { ##1##2##3##4##5 }
1307       \or: { ##1##2##3##4##5##6 }
1308       \or: { ##1##2##3##4##5##6##7 }
1309       \or: { ##1##2##3##4##5##6##7##8 }
1310       \or: { ##1##2##3##4##5##6##7##8##9 }
1311       \else: { \c_false_bool }
1312     \fi:
1313   }
1314   {#1}
1315 }
1316 \cs_set_protected:Npn \_cs_parm_from_arg_count_test:nnF #1#2
1317 {
1318   \if_meaning:w \c_false_bool #1
1319     \exp_after:wN \use_ii:nn
1320   \else:
1321     \exp_after:wN \use_i:nn
1322   \fi:
1323   { #2 {#1} }
1324 }

```


(End definition for `_cs_parm_from_arg_count:nnF`. This function is documented on page ??.)

3.14 Defining functions from a given number of arguments

`_cs_count_signature:N` 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.

```

1325 \cs_new:Npn \_cs_count_signature:N #1
1326 { \int_eval:n { \_cs_split_function:NN #1 \_cs_count_signature:nnN } }
1327 \cs_new:Npn \_cs_count_signature:nnN #1#2#3
1328 {
1329   \if_meaning:w \c_true_bool #3
1330     \tl_count:n {#2}
1331   \else:
1332     \c_minus_one
1333   \fi:
1334 }
1335 \cs_new_nopar:Npn \_cs_count_signature:c
1336 { \exp_args:Nc \_cs_count_signature:N }
```

(End definition for `_cs_count_signature:N` and `_cs_count_signature:c`. These functions are documented on page ??.)

`\cs_generate_from_arg_count:NNnn` We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since \TeX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```

1337 \cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
1338 {
1339   \_cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
1340   {
1341     \_msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
1342     { \token_to_str:N #1 } { \int_eval:n {#3} }
1343   }
1344   {#4}
1345 }
```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```

1346 \cs_new_protected_nopar:Npn \cs_generate_from_arg_count:cNnn
1347 { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
1348 \cs_new_protected_nopar:Npn \cs_generate_from_arg_count:Ncnn
1349 { \exp_args:NNc \cs_generate_from_arg_count:NNnn }
```

(End definition for `\cs_generate_from_arg_count:NNnn`, `\cs_generate_from_arg_count:cNnn`, and `\cs_generate_from_arg_count:Ncnn`. These functions are documented on page ??.)

3.15 Using the signature to define functions

We can now combine some of the tools we have to provide a simple interface for defining functions. We define some simpler functions with user interface `\cs_set:Nn \foo_bar:nn {#1,#2}`, *i.e.*, the number of arguments is read from the signature.

We want to define `\cs_set:Nn` as

```

\cs_set:Nn \cs_set:Nx \cs_set_protected:Npn \cs_set:Nn #1#2
\cs_set_nopar:Nn {
\cs_set_nopar:Nx \cs_generate_from_arg_count:NNnn #1 \cs_set:Npn
\cs_set_protected:Nn { \__cs_count_signature:N #1 } {#2}
\cs_set_protected:Nx }
\cs_set_protected_nopar:Nx
\cs_gset:Nn \cs_gset:Nx
\cs_gset_nopar:Nn \cs_gset_nopar:Nx
\cs_gset_protected:Nn \cs_new_protected_nopar:cpx { cs_ #1 : #2 }
\cs_gset_protected:Nx {
\exp_not:N \__cs_generate_from_signature:NNn
\exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
}
\cs_new:Nn \cs_new:Nx
\cs_new_nopar:Nn \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
\cs_new_nopar:Nx {
\__cs_split_function:NN #2 \__cs_generate_from_signature:nnNNNn
#1 #2
}
\cs_new_protected:Nn \cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
\cs_new_protected:Nx {
\bool_if:NTF #3
{
\cs_generate_from_arg_count:NNnn
#5 #4 { \tl_count:n {#2} } {#6}
}
{
\__msg_kernel_error:nnx { kernel } { missing-colon }
{ \token_to_str:N #5 }
}
}
}

```

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.

Then we define the 24 variants beginning with N.

```

1375 \__cs_tmp:w { set } { Nn } { Npn }
1376 \__cs_tmp:w { set } { Nx } { Npx }
1377 \__cs_tmp:w { set_nopar } { Nn } { Npn }
1378 \__cs_tmp:w { set_nopar } { Nx } { Npx }
1379 \__cs_tmp:w { set_protected } { Nn } { Npn }
1380 \__cs_tmp:w { set_protected } { Nx } { Npx }

```

```

1381 \_cs_tmp:w { set_protected_nopar } { Nn } { Npn }
1382 \_cs_tmp:w { set_protected_nopar } { Nx } { Npx }
1383 \_cs_tmp:w { gset } { Nn } { Npn }
1384 \_cs_tmp:w { gset } { Nx } { Npx }
1385 \_cs_tmp:w { gset_nopar } { Nn } { Npn }
1386 \_cs_tmp:w { gset_nopar } { Nx } { Npx }
1387 \_cs_tmp:w { gset_protected } { Nn } { Npn }
1388 \_cs_tmp:w { gset_protected } { Nx } { Npx }
1389 \_cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
1390 \_cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
1391 \_cs_tmp:w { new } { Nn } { Npn }
1392 \_cs_tmp:w { new } { Nx } { Npx }
1393 \_cs_tmp:w { new_nopar } { Nn } { Npn }
1394 \_cs_tmp:w { new_nopar } { Nx } { Npx }
1395 \_cs_tmp:w { new_protected } { Nn } { Npn }
1396 \_cs_tmp:w { new_protected } { Nx } { Npx }
1397 \_cs_tmp:w { new_protected_nopar } { Nn } { Npn }
1398 \_cs_tmp:w { new_protected_nopar } { Nx } { Npx }

```

(End definition for \cs_set:Nn and others. These functions are documented on page ??.)

```

\cs_set:cn The 24 c variants simply use \exp_args:Nc.
\cs_set:cx
\cs_set_nopar:cn
\cs_set_nopar:cx
\cs_set_protected:cn
\cs_set_protected:cx
\cs_set_protected_nopar:cn
\cs_set_protected_nopar:cx
\cs_gset:cn
\cs_gset:cx
\cs_gset_nopar:cn
\cs_gset_nopar:cx
\cs_gset_protected:cn
\cs_gset_protected:cx
\cs_gset_protected_nopar:cn
\cs_gset_protected_nopar:cx
\cs_new:cn
\cs_new:cx
\cs_new_nopar:cn
\cs_new_nopar:cx
\cs_new_protected:cn
\cs_new_protected:cx
\cs_new_protected_nopar:cn
\cs_new_protected_nopar:cx

```

```

1399 \cs_set:Npn \_cs_tmp:w #1#2
1400 {
1401   \cs_new_protected_nopar:cpx { cs_ #1 : c #2 }
1402   {
1403     \exp_not:N \exp_args:Nc
1404     \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:
1405   }
1406 }
1407 \_cs_tmp:w { set } { n }
1408 \_cs_tmp:w { set } { x }
1409 \_cs_tmp:w { set_nopar } { n }
1410 \_cs_tmp:w { set_nopar } { x }
1411 \_cs_tmp:w { set_protected } { n }
1412 \_cs_tmp:w { set_protected } { x }
1413 \_cs_tmp:w { set_protected_nopar } { n }
1414 \_cs_tmp:w { set_protected_nopar } { x }
1415 \_cs_tmp:w { gset } { n }
1416 \_cs_tmp:w { gset } { x }
1417 \_cs_tmp:w { gset_nopar } { n }
1418 \_cs_tmp:w { gset_nopar } { x }
1419 \_cs_tmp:w { gset_protected } { n }
1420 \_cs_tmp:w { gset_protected } { x }
1421 \_cs_tmp:w { gset_protected_nopar } { n }
1422 \_cs_tmp:w { gset_protected_nopar } { x }
1423 \_cs_tmp:w { new } { n }
1424 \_cs_tmp:w { new } { x }
1425 \_cs_tmp:w { new_nopar } { n }
1426 \_cs_tmp:w { new_nopar } { x }
1427 \_cs_tmp:w { new_protected } { n }

```

```

1428 \__cs_tmp:w { new_protected } { x }
1429 \__cs_tmp:w { new_protected_nopar } { n }
1430 \__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
\cs_if_eq_p:Nc
\cs_if_eq_p:cc
\cs_if_eq:NNTF
\cs_if_eq:cNTF
\cs_if_eq:NcTF
\cs_if_eq:ccTF
1431 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
1432 {
1433   \if_meaning:w #1#2
1434     \prg_return_true: \else: \prg_return_false: \fi:
1435 }
1436 \cs_new_nopar:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
1437 \cs_new_nopar:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
1438 \cs_new_nopar:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }
1439 \cs_new_nopar:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
1440 \cs_new_nopar:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
1441 \cs_new_nopar:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
1442 \cs_new_nopar:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNT }
1443 \cs_new_nopar:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
1444 \cs_new_nopar:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
1445 \cs_new_nopar:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
1446 \cs_new_nopar:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
1447 \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.

```

1448 \cs_new_protected:Npn \__kernel_register_show:N #1
1449 {
1450   \cs_if_exist:NTF #1
1451     { \tex_showthe:D \use:n {#1} }
1452     {
1453       \__msg_kernel_error:nx { kernel } { variable-not-defined }
1454       { \token_to_str:N #1 }
1455     }
1456 }
1457 \cs_new_protected_nopar:Npn \__kernel_register_show:c
1458 { \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.

```

1459 \group_begin:
1460   \tex_lccode:D ‘? = ‘: \scan_stop:
1461   \tex_catcode:D ‘? = 12 \scan_stop:
1462   \tex_lowercase:D
1463   {
1464     \group_end:
1465     \cs_new_protected:Npn \cs_show:N #1
1466     {
1467       \__msg_show_variable:n
1468       {
1469         > ~ \token_to_str:N #1 =
1470         \exp_after:wN \__cs_show:www \cs_meaning:N #1
1471         \use_none:nn ? \prg_do_nothing:
1472       }
1473     }
1474     \cs_new:Npn \__cs_show:www #1 ? { #1 ? \\\ }
1475   }
1476   \cs_new_protected_nopar:Npn \cs_show:c
1477   { \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
1478 \cs_new_eq:NN \luatex_if_engine:T \use_none:n
1479 \cs_new_eq:NN \luatex_if_engine:F \use:n
1480 \cs_new_eq:NN \luatex_if_engine:TF \use_ii:nn
1481 \cs_new_eq:NN \pdfTeX_if_engine:T \use:n
1482 \cs_new_eq:NN \pdfTeX_if_engine:F \use_none:n
1483 \cs_new_eq:NN \pdfTeX_if_engine:TF \use_i:nn
1484 \cs_new_eq:NN \xetex_if_engine:T \use_none:n
1485 \cs_new_eq:NN \xetex_if_engine:F \use:n
1486 \cs_new_eq:NN \xetex_if_engine:TF \use_ii:nn
1487 \cs_new_eq:NN \luatex_if_engine_p: \c_false_bool
1488 \cs_new_eq:NN \pdfTeX_if_engine_p: \c_true_bool
1489 \cs_new_eq:NN \xetex_if_engine_p: \c_false_bool
1490 \cs_if_exist:NT \xetex_XeTeXversion:D
1491 {
1492   \cs_gset_eq:NN \pdfTeX_if_engine:T \use_none:n
1493   \cs_gset_eq:NN \pdfTeX_if_engine:F \use:n
1494   \cs_gset_eq:NN \pdfTeX_if_engine:TF \use_ii:nn
1495   \cs_gset_eq:NN \xetex_if_engine:T \use:n
1496   \cs_gset_eq:NN \xetex_if_engine:F \use_none:n
1497   \cs_gset_eq:NN \xetex_if_engine:TF \use_i:nn
1498   \cs_gset_eq:NN \pdfTeX_if_engine_p: \c_false_bool

```

```

1499     \cs_gset_eq:NN \xetex_if_engine_p: \c_true_bool
1500   }
1501   \cs_if_exist:NT \luatex_directlua:D
1502   {
1503     \cs_gset_eq:NN \luatex_if_engine:T \use:n
1504     \cs_gset_eq:NN \luatex_if_engine:F \use_none:n
1505     \cs_gset_eq:NN \luatex_if_engine:TF \use_i:nn
1506     \cs_gset_eq:NN \pdfTeX_if_engine:T \use_none:n
1507     \cs_gset_eq:NN \pdfTeX_if_engine:F \use:n
1508     \cs_gset_eq:NN \pdfTeX_if_engine:TF \use_ii:nn
1509     \cs_gset_eq:NN \luatex_if_engine_p: \c_true_bool
1510     \cs_gset_eq:NN \pdfTeX_if_engine_p: \c_false_bool
1511   }

```

(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!

```

1512 \cs_new_nopar:Npn \prg_do_nothing: { }

```

(End definition for `\prg_do_nothing:`. This function is documented on page 9.)

3.20 String comparisons

`\str_if_eq_p:nn` Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore make life a bit clearer. The `nn` and `xx` versions are created directly as this is most efficient. These should eventually move somewhere else.

`\str_if_eq_x_p:nn`
`\str_if_eq:nnTF`
`\str_if_eq_x:nnTF`

```

1513 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
1514 {
1515   \if_int_compare:w \pdfTeX_strcmp:D { \exp_not:n {#1} } { \exp_not:n {#2} }
1516     = \c_zero
1517   \prg_return_true: \else: \prg_return_false: \fi:
1518 }
1519 \prg_new_conditional:Npnn \str_if_eq_x:nn #1#2 { p , T , F , TF }
1520 {
1521   \if_int_compare:w \pdfTeX_strcmp:D {#1} {#2} = \c_zero
1522   \prg_return_true: \else: \prg_return_false: \fi:
1523 }

```

(End definition for `\str_if_eq:nn` and `\str_if_eq_x:nn`. These functions are documented on page 22.)

`__str_if_eq_x_return:nn` It turns out that we often need to compare a token list with the result of applying some function to it, and return with `\prg_return_true/false:`. This test is similar to `\str_if_eq:nnTF`, but hard-coded for speed.

```

1524 \cs_new:Npn __str_if_eq_x_return:nn #1 #2
1525 {
1526   \if_int_compare:w \pdfTeX_strcmp:D {#1} {#2} = \c_zero
1527   \prg_return_true:
1528   \else:

```

```

1529     \prg_return_false:
1530     \fi:
1531 }
(End definition for \_str_if_eq_x_return:nn.)

```

```

\str_case:nnn No calculations for strings, otherwise no surprises.
\str_case_x:nnn
\_prg_case_end:nw
  \_str_case:nw
  \_str_case_x:nw
\_str_case_end:nw
1532 \cs_new:Npn \str_case:nnn #1#2#3
1533 {
1534   \tex_romannumeral:D
1535   \_str_case:nw {#1} #2 {#1} {#3} \q_recursion_stop
1536 }
1537 \cs_new:Npn \_str_case:nw #1#2#3
1538 {
1539   \str_if_eq:nnTF {#1} {#2}
1540   { \_str_case_end:nw {#3} }
1541   { \_str_case:nw {#1} }
1542 }
1543 \cs_new:Npn \str_case_x:nnn #1#2#3
1544 {
1545   \tex_romannumeral:D
1546   \_str_case_x:nw {#1} #2 {#1} {#3} \q_recursion_stop
1547 }
1548 \cs_new:Npn \_str_case_x:nw #1#2#3
1549 {
1550   \str_if_eq_x:nnTF {#1} {#2}
1551   { \_str_case_end:nw {#3} }
1552   { \_str_case_x:nw {#1} }
1553 }

```

Here, #1 will be the code needed, #2 will be any remaining case or cases, and the \c_zero stops the \romannumeral.

```

1554 \cs_new:Npn \_prg_case_end:nw #1#2 \q_recursion_stop { \c_zero #1 }
1555 \cs_new_eq:NN \_str_case_end:nw \_prg_case_end:nw
(End definition for \str_case:nnn and \str_case_x:nnn. These functions are documented on page 25.)

```

3.21 Breaking out of mapping functions

_prg_break_point:Nn In inline mappings, the nesting level must be reset at the end of the mapping, even when 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.

```

1556 \cs_new_eq:NN \_prg_break_point:Nn \use_ii:nn
1557 \cs_new:Npn \_prg_map_break:Nn #1#2#3 \_prg_break_point:Nn #4#5
1558 {
1559   #5
1560   \if_meaning:w #1 #4
1561   \exp_after:wN \use_iii:nnn

```

```

1562 \fi:
1563 \__prg_map_break:Nn #1 {#2}
1564 }

```

(End definition for __prg_break_point:Nn and __prg_map_break:Nn. These functions are documented on page 43.)

__prg_break_point: Very simple analogues of __prg_break_point:Nn and __prg_map_break:Nn, for use
__prg_break: in fast short-term recursions which are not mappings, do not need to support nesting,
__prg_break:n and in which nothing has to be done at the end of the loop.

```

1565 \cs_new_eq:NN \__prg_break_point: \prg_do_nothing:
1566 \cs_new:Npn \__prg_break: #1 \__prg_break_point: { }
1567 \cs_new:Npn \__prg_break:n #1#2 \__prg_break_point: {#1}

```

(End definition for __prg_break_point:. This function is documented on page ??.)

3.22 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

```

1568 <*deprecated>
1569 \cs_new_eq:NN \cs_gnew_nopar:Npn \cs_new_nopar:Npn
1570 \cs_new_eq:NN \cs_gnew:Npn \cs_new:Npn
1571 \cs_new_eq:NN \cs_gnew_protected_nopar:Npn \cs_new_protected_nopar:Npn
1572 \cs_new_eq:NN \cs_gnew_protected:Npn \cs_new_protected:Npn
1573 \cs_new_eq:NN \cs_gnew_nopar:Npx \cs_new_nopar:Npx
1574 \cs_new_eq:NN \cs_gnew:Npx \cs_new:Npx
1575 \cs_new_eq:NN \cs_gnew_protected_nopar:Npx \cs_new_protected_nopar:Npx
1576 \cs_new_eq:NN \cs_gnew_protected:Npx \cs_new_protected:Npx
1577 \cs_new_eq:NN \cs_gnew_nopar:cpn \cs_new_nopar:cpn
1578 \cs_new_eq:NN \cs_gnew:cpn \cs_new:cpn
1579 \cs_new_eq:NN \cs_gnew_protected_nopar:cpn \cs_new_protected_nopar:cpn
1580 \cs_new_eq:NN \cs_gnew_protected:cpn \cs_new_protected:cpn
1581 \cs_new_eq:NN \cs_gnew_nopar:cpx \cs_new_nopar:cpx
1582 \cs_new_eq:NN \cs_gnew:cpx \cs_new:cpx
1583 \cs_new_eq:NN \cs_gnew_protected_nopar:cpx \cs_new_protected_nopar:cpx
1584 \cs_new_eq:NN \cs_gnew_protected:cpx \cs_new_protected:cpx
1585 </deprecated>
1586 <*deprecated>
1587 \cs_new_eq:NN \cs_gnew_eq:NN \cs_new_eq:NN
1588 \cs_new_eq:NN \cs_gnew_eq:cN \cs_new_eq:cN
1589 \cs_new_eq:NN \cs_gnew_eq:Nc \cs_new_eq:Nc
1590 \cs_new_eq:NN \cs_gnew_eq:cc \cs_new_eq:cc
1591 </deprecated>
1592 <*deprecated>
1593 \cs_new_eq:NN \cs_gundefine:N \cs_undefine:N
1594 \cs_new_eq:NN \cs_gundefine:c \cs_undefine:c
1595 </deprecated>
1596 <*deprecated>
1597 \cs_new_eq:NN \group_execute_after:N \group_insert_after:N
1598 </deprecated>

```


Deprecated 2011-09-06, for removal by 2011-12-31.

```
\c_pdftex_is_engine_bool
\c_luatex_is_engine_bool
\c_xetex_is_engine_bool
```

Predicates are better

```
1599 <*deprecated>
1600 \cs_new_eq:NN \c_luatex_is_engine_bool \luatex_if_engine_p:
1601 \cs_new_eq:NN \c_pdftex_is_engine_bool \pdftex_if_engine_p:
1602 \cs_new_eq:NN \c_xetex_is_engine_bool \xetex_if_engine_p:
1603 </deprecated>
(End definition for \c_pdftex_is_engine_bool, \c_luatex_is_engine_bool, and \c_xetex_is_engine_bool.
These variables are documented on page ??.)
```

```
\use_i_after_fi:nw
\use_i_after_else:nw
\use_i_after_or:nw
\use_i_after_orelse:nw
```

These functions return the first argument after ending the conditional. This is rather specialized, and we want to de-emphasize the use of primitive T_EX conditionals.

```
1604 <*deprecated>
1605 \cs_set:Npn \use_i_after_fi:nw #1 \fi: { \fi: #1 }
1606 \cs_set:Npn \use_i_after_else:nw #1 \else: #2 \fi: { \fi: #1 }
1607 \cs_set:Npn \use_i_after_or:nw #1 \or: #2 \fi: { \fi: #1 }
1608 \cs_set:Npn \use_i_after_orelse:nw #1#2#3 \fi: { \fi: #1 }
1609 </deprecated>
(End definition for \use_i_after_fi:nw and others. These functions are documented on page ??.)
Deprecated 2011-09-07, for removal by 2011-12-31.
```

```
\cs_set_eq:NwN
```

```
1610 <*deprecated>
1611 \tex_let:D \cs_set_eq:NwN \tex_let:D
1612 </deprecated>
(End definition for \cs_set_eq:NwN. This function is documented on page ??.)
Deprecated 2012-06-05 for removal after 2012-12-31.
```

```
\str_if_eq_p:xx
\str_if_eq:xxTF
```

Not really true x-type expansion

```
1613 <*deprecated>
1614 \cs_new_eq:NN \str_if_eq_p:xx \str_if_eq_x_p:nn
1615 \cs_new_eq:NN \str_if_eq:xxT \str_if_eq_x:nnT
1616 \cs_new_eq:NN \str_if_eq:xxF \str_if_eq_x:nnF
1617 \cs_new_eq:NN \str_if_eq:xxTF \str_if_eq_x:nnTF
1618 </deprecated>
(End definition for \str_if_eq:xx. These functions are documented on page ??.)
```

```
\chk_if_free_cs:N
```

```
1619 <*deprecated>
1620 \cs_new_eq:NN \chk_if_free_cs:N \__chk_if_free_cs:N
1621 </deprecated>
(End definition for \chk_if_free_cs:N. This function is documented on page ??.)
1622 </initex | package>
```

4 l3expan implementation

```
1623 <*initex | package>
```

```
1624 <@@=exp>
```

We start by ensuring that the required packages are loaded.

```
1625 <*package>
```

```
1626 \ProvidesExplPackage
```

```
1627   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

```
1628   \__expl_package_check:
```

```
1629 </package>
```

`\exp_after:wN` These are defined in l3basics.

`\exp_not:N` (End definition for `\exp_after:wN`. This function is documented on page 33.)

`\exp_not:n`

4.1 General expansion

In this section a general mechanism for defining functions to handle argument handling is defined. These general expansion functions are expandable unless `x` is used. (Any version of `x` is going to have to use one of the L^AT_EX3 names for `\cs_set_nopar:Npx` at some point, and so is never going to be expandable.)

The definition of expansion functions with this technique happens in section 4.3. In section 4.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

`\l__exp_internal_tl` This scratch token list variable is defined in l3basics, as it is needed “early”. This is just a reminder that that is the case!

(End definition for `\l__exp_internal_tl`. This variable is documented on page 34.)

This code uses internal functions with names that start with `\::` to perform the expansions. All macros are `long` as this turned out to be desirable since the tokens undergoing expansion may be arbitrary user input.

An argument manipulator `\::<Z>` always has signature `#1\:::#2#3` where `#1` holds the remaining argument manipulations to be performed, `\:::` serves as an end marker for the list of manipulations, `#2` is the carried over result of the previous expansion steps and `#3` is the argument about to be processed. One exception to this rule is `\::p`, which has to grab an argument delimited by a left brace.

`__exp_arg_next:nnn` `#1` is the result of an expansion step, `#2` is the remaining argument manipulations and `#3` is the current result of the expansion chain. This auxiliary function moves `#1` back after `#3` in the input stream and checks if any expansion is left to be done by calling `#2`. In by far the most cases we will require to add a set of braces to the result of an argument manipulation so it is more effective to do it directly here. Actually, so far only the `c` of the final argument manipulation variants does not require a set of braces.

```
1630 \cs_new:Npn \__exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
```

```
1631 \cs_new:Npn \__exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }
```

(End definition for `__exp_arg_next:nnn`. This function is documented on page 34.)

`\:::` The end marker is just another name for the identity function.

```
1632 \cs_new:Npn \::: #1 {#1}
```

(End definition for \::.)

\::n This function is used to skip an argument that doesn't need to be expanded.

```
1633 \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.

```
1634 \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.

```
1635 \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.

```
1636 \cs_new:Npn \::c #1 \:: #2#3
```

```
1637 { \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.

```
1638 \cs_new:Npn \::o #1 \:: #2#3
```

```
1639 { \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 $\text{T}_{\text{E}}\text{X}$ had fully expanded `\cs_set_eq:Nc \aaa { b \l_tmpa_tl b }` into `\cs_set_eq:NN \aaa = \blurb` which then turned out to contain the non-expandable token `\cs_set_eq:NN`. Since the expansion of `\romannumeral -'0` is $\langle null \rangle$, we wind up with a fully expanded list, only $\text{T}_{\text{E}}\text{X}$ has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```
1640 \cs_new:Npn \::f #1 \:: #2#3
```

```
1641 {
```

```
1642   \exp_after:wN \_exp_arg_next:nnn
```

```
1643   \exp_after:wN { \tex_romannumeral:D -'0 #3 }
```

```
1644   {#1} {#2}
```

```
1645 }
```

```
1646 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(End definition for \::f. This function is documented on page 33.)

\::x This function is used to expand an argument fully.

```

1647 \cs_new_protected:Npn \::x #1 \::: #2#3
1648 {
1649   \cs_set_nopar:Npx \l__exp_internal_tl { {#3} }
1650   \exp_after:wN \__exp_arg_next:nnn \l__exp_internal_tl {#1} {#2}
1651 }

```

(End definition for \::x.)

\::v These functions return the value of a register, i.e., one of `tl`, `clist`, `int`, `skip`, `dim`
\::V and `muskip`. The `V` version expects a single token whereas `v` like `c` creates a `csname` from its argument given in braces and then evaluates it as if it was a `V`. The primitive `\romannumeral` sets off an expansion similar to an `f` type expansion, which we will terminate using `\c_zero`. The argument is returned in braces.

```

1652 \cs_new:Npn \::V #1 \::: #2#3
1653 {
1654   \exp_after:wN \__exp_arg_next:nnn
1655   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #3 }
1656   {#1} {#2}
1657 }
1658 \cs_new:Npn \::v # 1\::: #2#3
1659 {
1660   \exp_after:wN \__exp_arg_next:nnn
1661   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:c {#3} }
1662   {#1} {#2}
1663 }

```

(End definition for \::v. This function is documented on page 34.)

`__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:`.

```

1664 \cs_new:Npn \__exp_eval_register:N #1
1665 {
1666   \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.

```

1667   \if_meaning:w \scan_stop: #1
1668   \__exp_eval_error_msg:w
1669   \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`.

```

1670     \else:
1671       \exp_after:wN \use_i_ii:nnn
1672     \fi:
1673     \exp_after:wN \c_zero \tex_the:D #1
1674   }
1675   \cs_new:Npn \__exp_eval_register:c #1
1676     { \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}

1677 \cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
1678 {
1679   \fi:
1680   \fi:
1681   \_msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
1682   \c_zero
1683 }

```

(End definition for `__exp_eval_register:N` and `__exp_eval_register:c`. These functions are documented on page ??.)

4.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable and therefore allow to prefix them with `\tex_global:D` for example.

```

\exp_args:No Those lovely runs of expansion!
\exp_args:NNo
\exp_args:NNNo
1684 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
1685 \cs_new:Npn \exp_args:NNo #1#2#3
1686   { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
1687 \cs_new:Npn \exp_args:NNNo #1#2#3#4
1688   { \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }

```

(End definition for `\exp_args:No`. This function is documented on page 31.)

```

\exp_args:Nc In l3basics.
\exp_args:cc (End definition for \exp_args:Nc and \exp_args:cc. These functions are documented on page ??.)

```

`\exp_args:Nnc` Here are the functions that turn their argument into csnames but are expandable.

```

\exp_args:Ncc 1689 \cs_new:Npn \exp_args:Nnc #1#2#3
\exp_args:Nccc 1690 { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
1691 \cs_new:Npn \exp_args:Ncc #1#2#3
1692 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
1693 \cs_new:Npn \exp_args:Nccc #1#2#3#4
1694 {
1695   \exp_after:wN #1
1696   \cs:w #2 \exp_after:wN \cs_end:
1697   \cs:w #3 \exp_after:wN \cs_end:
1698   \cs:w #4 \cs_end:
1699 }

```

(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`

```

1700 \cs_new:Npn \exp_args:Nf #1#2
1701 { \exp_after:wN #1 \exp_after:wN { \tex_romannumeral:D -'0 #2 } }
1702 \cs_new:Npn \exp_args:Nv #1#2
1703 {
1704   \exp_after:wN #1 \exp_after:wN
1705   { \tex_romannumeral:D \__exp_eval_register:c {#2} }
1706 }
1707 \cs_new:Npn \exp_args:NV #1#2
1708 {
1709   \exp_after:wN #1 \exp_after:wN
1710   { \tex_romannumeral:D \__exp_eval_register:N #2 }
1711 }

```

(End definition for `\exp_args:Nf`, `\exp_args:Nv`, and `\exp_args:Nv`. These functions are documented on page 30.)

`\exp_args:NNV` Some more hand-tuned function with three arguments. If we forced that an `o` argument
`\exp_args:NNv` always has braces, we could implement `\exp_args:Nco` with less tokens and only two
`\exp_args:NNf` arguments.

```

\exp_args:NVV 1712 \cs_new:Npn \exp_args:NNf #1#2#3
\exp_args:Ncf 1713 {
\exp_args:Nco 1714   \exp_after:wN #1
1715   \exp_after:wN #2
1716   \exp_after:wN { \tex_romannumeral:D -'0 #3 }
1717 }
1718 \cs_new:Npn \exp_args:NNv #1#2#3
1719 {
1720   \exp_after:wN #1
1721   \exp_after:wN #2
1722   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:c {#3} }
1723 }
1724 \cs_new:Npn \exp_args:NNV #1#2#3
1725 {
1726   \exp_after:wN #1

```

```

1727     \exp_after:wN #2
1728     \exp_after:wN { \tex_romannumeral:D \_exp_eval_register:N #3 }
1729   }
1730   \cs_new:Npn \exp_args:Nco #1#2#3
1731   {
1732     \exp_after:wN #1
1733     \cs:w #2 \exp_after:wN \cs_end:
1734     \exp_after:wN {#3}
1735   }
1736   \cs_new:Npn \exp_args:Ncf #1#2#3
1737   {
1738     \exp_after:wN #1
1739     \cs:w #2 \exp_after:wN \cs_end:
1740     \exp_after:wN { \tex_romannumeral:D -'0 #3 }
1741   }
1742   \cs_new:Npn \exp_args:NVV #1#2#3
1743   {
1744     \exp_after:wN #1
1745     \exp_after:wN { \tex_romannumeral:D \exp_after:wN
1746       \_exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
1747     \exp_after:wN { \tex_romannumeral:D \_exp_eval_register:N #3 }
1748   }

```

(End definition for \exp_args:NNV and others. These functions are documented on page ??.)

\exp_args:Ncco A few more that we can hand-tune.

```

\exp_args:NcNc 1749 \cs_new:Npn \exp_args:NNNV #1#2#3#4
\exp_args:NcNo 1750 {
\exp_args:NNNV 1751   \exp_after:wN #1
1752   \exp_after:wN #2
1753   \exp_after:wN #3
1754   \exp_after:wN { \tex_romannumeral:D \_exp_eval_register:N #4 }
1755 }
1756 \cs_new:Npn \exp_args:NcNc #1#2#3#4
1757 {
1758   \exp_after:wN #1
1759   \cs:w #2 \exp_after:wN \cs_end:
1760   \exp_after:wN #3
1761   \cs:w #4 \cs_end:
1762 }
1763 \cs_new:Npn \exp_args:NcNo #1#2#3#4
1764 {
1765   \exp_after:wN #1
1766   \cs:w #2 \exp_after:wN \cs_end:
1767   \exp_after:wN #3
1768   \exp_after:wN {#4}
1769 }
1770 \cs_new:Npn \exp_args:Ncco #1#2#3#4
1771 {
1772   \exp_after:wN #1
1773   \cs:w #2 \exp_after:wN \cs_end:

```

```

1774 \cs:w #3 \exp_after:wN \cs_end:
1775 \exp_after:wN {#4}
1776 }

```

(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`

```

1777 \cs_new_protected_nopar:Npn \exp_args:Nx { \::x \::: }

```

(End definition for `\exp_args:Nx`. This function is documented on page 30.)

`\exp_args:Nnc` Here are the actual function definitions, using the helper functions above.

```

\exp_args:Nnc 1778 \cs_new_nopar:Npn \exp_args:Nnc { \::n \::c \::: }
\exp_args:Nfo 1779 \cs_new_nopar:Npn \exp_args:Nfo { \::f \::o \::: }
\exp_args:Nff 1780 \cs_new_nopar:Npn \exp_args:Nff { \::f \::f \::: }
\exp_args:Nnf 1781 \cs_new_nopar:Npn \exp_args:Nnf { \::n \::f \::: }
\exp_args:Nno 1782 \cs_new_nopar:Npn \exp_args:Nno { \::n \::o \::: }
\exp_args:NnV 1783 \cs_new_nopar:Npn \exp_args:NnV { \::n \::V \::: }
\exp_args:Noo 1784 \cs_new_nopar:Npn \exp_args:Noo { \::o \::o \::: }
\exp_args:Nof 1785 \cs_new_nopar:Npn \exp_args:Nof { \::o \::f \::: }
\exp_args:Noc 1786 \cs_new_nopar:Npn \exp_args:Noc { \::o \::c \::: }
\exp_args:NNx 1787 \cs_new_protected_nopar:Npn \exp_args:NNx { \::N \::x \::: }
\exp_args:Ncx 1788 \cs_new_protected_nopar:Npn \exp_args:Ncx { \::c \::x \::: }
\exp_args:Nnx 1789 \cs_new_protected_nopar:Npn \exp_args:Nnx { \::n \::x \::: }
\exp_args:Nox 1790 \cs_new_protected_nopar:Npn \exp_args:Nox { \::o \::x \::: }
\exp_args:Nxo 1791 \cs_new_protected_nopar:Npn \exp_args:Nxo { \::x \::o \::: }
\exp_args:Nxx 1792 \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:NNno 1793 \cs_new_nopar:Npn \exp_args:NNno { \::N \::n \::o \::: }
\exp_args:NNoo 1794 \cs_new_nopar:Npn \exp_args:NNoo { \::N \::o \::o \::: }
\exp_args:NNnc 1795 \cs_new_nopar:Npn \exp_args:NNnc { \::n \::n \::c \::: }
\exp_args:Nooo 1796 \cs_new_nopar:Npn \exp_args:Nooo { \::n \::n \::o \::: }
\exp_args:NNnx 1797 \cs_new_nopar:Npn \exp_args:NNnx { \::o \::o \::o \::: }
\exp_args:NNox 1798 \cs_new_protected_nopar:Npn \exp_args:NNnx { \::N \::n \::x \::: }
\exp_args:NNox 1799 \cs_new_protected_nopar:Npn \exp_args:NNox { \::N \::o \::x \::: }
\exp_args:NNnx 1800 \cs_new_protected_nopar:Npn \exp_args:NNnx { \::n \::n \::x \::: }
\exp_args:Nnox 1801 \cs_new_protected_nopar:Npn \exp_args:Nnox { \::n \::o \::x \::: }
\exp_args:Nccx 1802 \cs_new_protected_nopar:Npn \exp_args:Nccx { \::c \::c \::x \::: }
\exp_args:Ncnx 1803 \cs_new_protected_nopar:Npn \exp_args:Ncnx { \::c \::n \::x \::: }
\exp_args:Noox 1804 \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

There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

\__exp_arg_last_unbraced:nn
\::f_unbraced
\::o_unbraced
\::V_unbraced
\::v_unbraced
\::x_unbraced
1805 \cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
1806 \cs_new:Npn \::f_unbraced \::: #1#2
1807 {
1808   \exp_after:wN \__exp_arg_last_unbraced:nn
1809   \exp_after:wN { \tex_romannumeral:D -'0 #2 } {#1}
1810 }
1811 \cs_new:Npn \::o_unbraced \::: #1#2
1812 { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
1813 \cs_new:Npn \::V_unbraced \::: #1#2
1814 {
1815   \exp_after:wN \__exp_arg_last_unbraced:nn
1816   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #2 } {#1}
1817 }
1818 \cs_new:Npn \::v_unbraced \::: #1#2
1819 {
1820   \exp_after:wN \__exp_arg_last_unbraced:nn
1821   \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:c {#2} } {#1}
1822 }
1823 \cs_new_protected:Npn \::x_unbraced \::: #1#2
1824 {
1825   \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
1826   \l__exp_internal_tl
1827 }

```

(End definition for __exp_arg_last_unbraced:nn. This function is documented on page ??.)

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```

\exp_last_unbraced:Nv
\exp_last_unbraced:Nv
\exp_last_unbraced:Nf
\exp_last_unbraced:Nv
\exp_last_unbraced:Nco
\exp_last_unbraced:NcV
\exp_last_unbraced:NNV
\exp_last_unbraced:NNo
\exp_last_unbraced:NNNV
\exp_last_unbraced:NNNo
\exp_last_unbraced:Nno
\exp_last_unbraced:Noo
\exp_last_unbraced:Nfo
\exp_last_unbraced:NnNo
\exp_last_unbraced:Nx
1828 \cs_new:Npn \exp_last_unbraced:Nv #1#2
1829 { \exp_after:wN #1 \tex_romannumeral:D \__exp_eval_register:N #2 }
1830 \cs_new:Npn \exp_last_unbraced:Nv #1#2
1831 { \exp_after:wN #1 \tex_romannumeral:D \__exp_eval_register:c {#2} }
1832 \cs_new:Npn \exp_last_unbraced:Nco #1#2 { \exp_after:wN #1 #2 }
1833 \cs_new:Npn \exp_last_unbraced:Nf #1#2
1834 { \exp_after:wN #1 \tex_romannumeral:D -'0 #2 }
1835 \cs_new:Npn \exp_last_unbraced:Nco #1#2#3
1836 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
1837 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
1838 {
1839   \exp_after:wN #1
1840   \cs:w #2 \exp_after:wN \cs_end:
1841   \tex_romannumeral:D \__exp_eval_register:N #3
1842 }
1843 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3
1844 {
1845   \exp_after:wN #1

```

```

1846 \exp_after:wN #2
1847 \tex_romannumeral:D \exp_eval_register:N #3
1848 }
1849 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
1850 { \exp_after:wN #1 \exp_after:wN #2 #3 }
1851 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
1852 {
1853 \exp_after:wN #1
1854 \exp_after:wN #2
1855 \exp_after:wN #3
1856 \tex_romannumeral:D \exp_eval_register:N #4
1857 }
1858 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
1859 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
1860 \cs_new_nopar:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \:: }
1861 \cs_new_nopar:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \:: }
1862 \cs_new_nopar:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \:: }
1863 \cs_new_nopar:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \:: }
1864 \cs_new_protected_nopar:Npn \exp_last_unbraced:Nx { \::x_unbraced \:: }

```

(End definition for `\exp_last_unbraced:NV`. This function is documented on page 32.)

`\exp_last_two_unbraced:Noo`
`\exp_last_two_unbraced:noN`

If #2 is a single token then this can be implemented as

```

\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
{ \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }

```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```

1865 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
1866 { \exp_after:wN \exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
1867 \cs_new:Npn \exp_last_two_unbraced:noN #1#2#3
1868 { \exp_after:wN #3 #2 #1 }

```

(End definition for `\exp_last_two_unbraced:Noo`. This function is documented on page 32.)

4.5 Preventing expansion

`\exp_not:o`
`\exp_not:c`
`\exp_not:f`
`\exp_not:V`
`\exp_not:v`

```

1869 \cs_new:Npn \exp_not:o #1 { \etex_unexpanded:D \exp_after:wN {#1} }
1870 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
1871 \cs_new:Npn \exp_not:f #1
1872 { \etex_unexpanded:D \exp_after:wN { \tex_romannumeral:D -'0 #1 } }
1873 \cs_new:Npn \exp_not:V #1
1874 {
1875 \etex_unexpanded:D \exp_after:wN
1876 { \tex_romannumeral:D \exp_eval_register:N #1 }
1877 }
1878 \cs_new:Npn \exp_not:v #1
1879 {
1880 \etex_unexpanded:D \exp_after:wN

```

```

1881     { \tex_romannumeral:D \__exp_eval_register:c {#1} }
1882   }

```

(End definition for `\exp_not:o`. This function is documented on page 33.)

4.6 Defining function variants

```

1883 \@@=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.

```

1884 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
1885 {
1886   \__chk_if_exist_cs:N #1
1887   \__cs_generate_variant:N #1
1888   \exp_after:wN \__cs_split_function:NN
1889   \exp_after:wN #1
1890   \exp_after:wN \__cs_generate_variant:nnNN
1891   \exp_after:wN #1
1892   \etex_detokenize:D {#2} , \scan_stop: , \q_recursion_stop
1893 }

```

(End definition for `\cs_generate_variant:Nn`. This function is documented on page 28.)

```

\__cs_generate_variant:N
\__cs_generate_variant:ww
\__cs_generate_variant:wwNw

```

The goal here is to pick up protected parent functions. There are four cases: the parent function can be a primitive or a macro, and can be expandable or not. For non-expandable primitives, all variants should be protected; skipping the `\else:` branch is safe because all primitive T_EX conditionals are expandable.

The other case where variants should be protected is when the parent function is a protected macro: then `protected` appears in the meaning before the first occurrence of `macro`. The `ww` auxiliary removes everything in the meaning string after the first `ma`. We use `ma` rather than the full `macro` because the meaning of the `\firstmark` primitive (and four others) can contain an arbitrary string after a leading `firstmark:`. Then, look for `pr` in the part we extracted: no need to look for anything longer: the only strings we can have are an empty string, `\long_`, `\protected_`, `\protected\long_`, `\first`, `\top`, `\bot`, `\splittop`, or `\splitbot`, with `\` replaced by the appropriate escape character. If `pr` appears in the part before `ma`, the first `\q_mark` is taken as an argument of the `wwNw` auxiliary, and `#3` is `\cs_new_protected_nopar:Npx`, otherwise it is `\cs_new_nopar:Npx`.

```

1894 \group_begin:
1895   \tex_catcode:D '\M = 12 \scan_stop:
1896   \tex_catcode:D '\A = 12 \scan_stop:
1897   \tex_catcode:D '\P = 12 \scan_stop:
1898   \tex_catcode:D '\R = 12 \scan_stop:
1899   \tex_lowercase:D

```

```

1900 {
1901   \group_end:
1902   \cs_new_protected:Npn \__cs_generate_variant:N #1
1903   {
1904     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
1905     \cs_set_eq:NN \__cs_tmp:w \cs_new_protected_nopar:Npx
1906     \else:
1907       \exp_after:wN \__cs_generate_variant:ww
1908       \token_to_meaning:N #1 MA \q_mark
1909       \q_mark \cs_new_protected_nopar:Npx
1910       PR
1911       \q_mark \cs_new_nopar:Npx
1912       \q_stop
1913     \fi:
1914   }
1915   \cs_new_protected:Npn \__cs_generate_variant:ww #1 MA #2 \q_mark
1916   { \__cs_generate_variant:wwNw #1 }
1917   \cs_new_protected:Npn \__cs_generate_variant:wwNw
1918   #1 PR #2 \q_mark #3 #4 \q_stop
1919   {
1920     \cs_set_eq:NN \__cs_tmp:w #3
1921   }
1922 }

```

(End definition for __cs_generate_variant:N. This function is documented on page 28.)

__cs_generate_variant:nnNN

- #1 : Base name.
- #2 : Base signature.
- #3 : Boolean.
- #4 : Base function.

If the boolean is \c_false_bool, the base function has no colon and we abort with an error; otherwise, set off a loop through the desired variant forms. The original function is retained as #4 for efficiency.

```

1923 \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
1924 {
1925   \if_meaning:w \c_false_bool #3
1926   \__msg_kernel_error:nnx { kernel } { missing-colon }
1927   { \token_to_str:c {#1} }
1928   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1929   \fi:
1930   \__cs_generate_variant:Nnnw #4 {#1}{#2}
1931 }

```

(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`.

```

1932 \cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
1933 {
1934   \if_meaning:w \scan_stop: #4
1935   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1936   \fi:
1937   \use:x
1938   {
1939     \exp_not:N \__cs_generate_variant:wwNN
1940     \__cs_generate_variant_loop:nNwN { }
1941     #4
1942     \__cs_generate_variant_loop_end:nwwwNNnn
1943     \q_mark
1944     #3 ~
1945     { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
1946     { }
1947     \q_stop
1948     \exp_not:N #1 {#2} {#4}
1949   }
1950   \__cs_generate_variant:Nnnw #1 {#2} {#3}
1951 }
```

(End definition for `__cs_generate_variant:Nnnw`.)

<pre> __cs_generate_variant_loop:nNwN __cs_generate_variant_loop_same:w __cs_generate_variant_loop_end:nwwwNNnn __cs_generate_variant_loop_long:wNNnn __cs_generate_variant_loop_invalid:NNwNNnn </pre>	<pre> #1 : Last few (consecutive) letters common between the base and variant (in fact, __- cs_generate_variant_same:N <letter> for each letter). #2 : Next variant letter. #3 : Remainder of variant form. #4 : Next base letter. </pre>
--	--

The first argument is populated by `__cs_generate_variant_loop_same:w` when a variant letter and a base letter match. It is flushed into the input stream whenever the two letters are different: if the loop ends before, the argument is dropped, which means that trailing common letters are ignored.

The case where the two letters are different is only allowed with a base letter of `N` or `n`. Otherwise, call `__cs_generate_variant_loop_invalid:NNwNNnn` to remove the end of the loop, get arguments at the end of the loop, and place an appropriate error message as a second argument of `__cs_generate_variant:wwNN`. If the letters are distinct and the base letter is indeed `n` or `N`, leave in the input stream whatever argument was collected, and the next variant letter `#2`, then loop by calling `__cs_generate_variant_loop:nNwN`.

The loop can stop in three ways.

- If the end of the variant form is encountered first, `#2` is `__cs_generate_variant_loop_end:nwwwNNnn` (expanded by the conditional `\if:w`), which inserts some tokens to end the conditional; grabs the *<base name>* as `#7`, the *<variant signature>* `#8`, the *<next base letter>* `#1` and the part `#3` of the base signature that wasn't read yet; and combines those into the *<new function>* to be defined.
- If the end of the base form is encountered first, `#4` is `~{}\fi:` which ends the conditional (with an empty expansion), followed by `__cs_generate_variant_loop_long:wNNnn`, which places an error as the second argument of `__cs_generate_variant:wwNN`.
- The loop can be interrupted early if the requested expansion is unavailable, namely when the variant and base letters differ and the base is neither `n` nor `N`. Again, an error is placed as the second argument of `__cs_generate_variant:wwNN`.

Note that if the variant form has the same length as the base form, `#2` is as described in the first point, and `#4` as described in the second point above. The `__cs_generate_variant_loop_end:nwwwNNnn` breaking function takes the empty brace group in `#4` as its first argument: this empty brace group produces the correct signature for the full variant.

```

1952 \cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \q_mark #4
1953 {
1954   \if:w #2 #4
1955     \exp_after:wN \__cs_generate_variant_loop_same:w
1956   \else:
1957     \if:w N #4 \else:
1958       \if:w n #4 \else:
1959         \__cs_generate_variant_loop_invalid:NNwNNnn #4#2
1960       \fi:
1961     \fi:

```

```

1962     \fi:
1963     #1
1964     \prg_do_nothing:
1965     #2
1966     \__cs_generate_variant_loop:nNwN { } #3 \q_mark
1967 }
1968 \cs_new:Npn \__cs_generate_variant_loop_same:w
1969 #1 \prg_do_nothing: #2#3#4
1970 {
1971     #3 { #1 \__cs_generate_variant_same:N #2 }
1972 }
1973 \cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
1974 #1#2 \q_mark #3 ~ #4 \q_stop #5#6#7#8
1975 {
1976     \scan_stop: \scan_stop: \fi:
1977     \exp_not:N \q_mark
1978     \exp_not:N \q_stop
1979     \exp_not:N #6
1980     \exp_not:c { #7 : #8 #1 #3 }
1981 }
1982 \cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \q_stop #2#3#4#5
1983 {
1984     \exp_not:n
1985     {
1986         \q_mark
1987         \__msg_kernel_error:nxxx { kernel } { variant-too-long }
1988         {#5} { \token_to_str:N #3 }
1989         \use_none:nnnn
1990         \q_stop
1991         #3
1992         #3
1993     }
1994 }
1995 \cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
1996 #1#2 \fi: \fi: \fi: #3 \q_stop #4#5#6#7
1997 {
1998     \fi: \fi: \fi:
1999     \exp_not:n
2000     {
2001         \q_mark
2002         \__msg_kernel_error:nxxxx { kernel } { invalid-variant }
2003         {#7} { \token_to_str:N #5 } {#1} {#2}
2004         \use_none:nnnn
2005         \q_stop
2006         #5
2007         #5
2008     }
2009 }

```

(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.

```

2010 \cs_new:Npn \_cs_generate_variant_same:N #1
2011 {
2012   \if:w N #1
2013     N
2014   \else:
2015     \if:w p #1
2016       p
2017     \else:
2018       n
2019     \fi:
2020   \fi:
2021 }

```

(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.

```

2022 \cs_new_protected:Npn \_cs_generate_variant:wwNN
2023   #1 \q_mark #2 \q_stop #3#4
2024 {
2025   #2
2026   \cs_if_free:NTF #4
2027   {
2028     \group_begin:
2029       \_cs_generate_internal_variant:n {#1}
2030       \_cs_tmp:w #4 { \exp_not:c { exp_args:N #1 } \exp_not:N #3 }
2031     \group_end:
2032   }
2033   {
2034     \iow_log:x
2035     {
2036       Variant~\token_to_str:N #4~%
2037       already~defined;~ not~ changing~ it~on~line~%
2038       \tex_the:D \tex_inputlineno:D
2039     }
2040   }
2041 }

```

(End definition for _cs_generate_variant:wwNN.)

`_cs_generate_internal_variant: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.

```

2042 \group_begin:

```



```

2043 \tex_catcode:D '\X = 12 \scan_stop:
2044 \tex_lccode:D '\N = '\N \scan_stop:
2045 \tex_lowercase:D
2046 {
2047   \group_end:
2048   \cs_new_protected:Npn \__cs_generate_internal_variant:n #1
2049   {
2050     \__cs_generate_internal_variant:wnnNwnn
2051     #1 \q_mark
2052     { \cs_set_eq:NN \__cs_tmp:w \cs_new_protected_nopar:Npx }
2053     \cs_new_protected_nopar:cpx
2054     X \q_mark
2055     { }
2056     \cs_new_nopar:cpx
2057     \q_stop
2058     { exp_args:N #1 }
2059     { \__cs_generate_internal_variant_loop:n #1 { : \use_i:nn } }
2060   }
2061   \cs_new_protected:Npn \__cs_generate_internal_variant:wnnNwnn
2062   #1 X #2 \q_mark #3 #4 #5 \q_stop #6 #7
2063   {
2064     #3
2065     \cs_if_free:cT {#6} { #4 {#6} {#7} }
2066   }
2067 }

```

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.

```

2068 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
2069 {
2070   \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
2071   \__cs_generate_internal_variant_loop:n
2072 }

```

(End definition for `__cs_generate_internal_variant:n`. This function is documented on page ??.)

4.7 Variants which cannot be created earlier

`\str_if_eq_p:Vn` These cannot come earlier as they need `\cs_generate_variant:Nn`.

```

\str_if_eq_p:on 2073 \cs_generate_variant:Nn \str_if_eq_p:nn { V , o }
\str_if_eq_p:nV 2074 \cs_generate_variant:Nn \str_if_eq_p:nn { nV , no , VV }
\str_if_eq_p:no 2075 \cs_generate_variant:Nn \str_if_eq:nnT { V , o }
\str_if_eq_p:VV 2076 \cs_generate_variant:Nn \str_if_eq:nnT { nV , no , VV }
\str_if_eq:VnTF 2077 \cs_generate_variant:Nn \str_if_eq:nnF { V , o }
\str_if_eq:onTF 2078 \cs_generate_variant:Nn \str_if_eq:nnF { nV , no , VV }
\str_if_eq:nVTF 2079 \cs_generate_variant:Nn \str_if_eq:nnTF { V , o }
\str_if_eq:noTF 2080 \cs_generate_variant:Nn \str_if_eq:nnTF { nV , no , VV }
\str_if_eq:VVTF 2081 \cs_generate_variant:Nn \str_case:nnn { o }
\str_case:onn

```

(End definition for `\str_if_eq:Vn` and others. These functions are documented on page ??.)

```
2082 </initex | package>
```

5 l3prg implementation

The following test files are used for this code: `m3prg001.lvt`, `m3prg002.lvt`, `m3prg003.lvt`.

```
2083 <*initex | package>
2084 <*package>
2085 \ProvidesExplPackage
2086   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
2087   \__expl_package_check:
2088 </package>
```

5.1 Primitive conditionals

`\if_bool:N` Those two primitive TeX conditionals are synonyms. They should not be used outside the kernel code.

```
2089 \tex_let:D \if_bool:N          \tex_ifodd:D
2090 \tex_let:D \if_predicate:w      \tex_ifodd:D
```

(End definition for `\if_bool:N`. This function is documented on page 42.)

5.2 Defining a set of conditional functions

These are all defined in `l3basics`, as they are needed “early”. This is just a reminder that that is the case!

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 37.)

5.3 The boolean data type

```
2091 <@@=bool>
```

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```
2092 \cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
2093 \cs_generate_variant:Nn \bool_new:N { c }
```

(End definition for `\bool_new:N` and `\bool_new:c`. These functions are documented on page ??.)

Setting is already pretty easy.

```
2094 \cs_new_protected:Npn \bool_set_true:N #1
2095   { \cs_set_eq:NN #1 \c_true_bool }
2096 \cs_new_protected:Npn \bool_set_false:N #1
2097   { \cs_set_eq:NN #1 \c_false_bool }
2098 \cs_new_protected:Npn \bool_gset_true:N #1
2099   { \cs_gset_eq:NN #1 \c_true_bool }
2100 \cs_new_protected:Npn \bool_gset_false:N #1
```

```

2101 { \cs_gset_eq:NN #1 \c_false_bool }
2102 \cs_generate_variant:Nn \bool_set_true:N { c }
2103 \cs_generate_variant:Nn \bool_set_false:N { c }
2104 \cs_generate_variant:Nn \bool_gset_true:N { c }
2105 \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 2106 \cs_new_eq:NN \bool_set_eq:NN \cs_set_eq:NN
\bool_set_eq:Nc 2107 \cs_new_eq:NN \bool_set_eq:Nc \cs_set_eq:Nc
\bool_set_eq:cc 2108 \cs_new_eq:NN \bool_set_eq:cN \cs_set_eq:cN
\bool_gset_eq:NN 2109 \cs_new_eq:NN \bool_set_eq:cc \cs_set_eq:cc
\bool_gset_eq:cN 2110 \cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
\bool_gset_eq:Nc 2111 \cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
\bool_gset_eq:cN 2112 \cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
\bool_gset_eq:cc 2113 \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 2114 \cs_new_protected:Npn \bool_set:Nn #1#2
\bool_gset:cn 2115 { \tex_chardef:D #1 = \bool_if_p:n {#2} }
2116 \cs_new_protected:Npn \bool_gset:Nn #1#2
2117 { \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2} }
2118 \cs_generate_variant:Nn \bool_set:Nn { c }
2119 \cs_generate_variant:Nn \bool_gset:Nn { c }

```

(End definition for `\bool_set:Nn` and `\bool_set:cn`. These functions are documented on page ??.)

`\bool_if_p:N` Straight forward here. We could optimize here if we wanted to as the boolean can just
`\bool_if_p:c` be input directly.

```

\bool_if:N $\underline{TF}$  2120 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
\bool_if:c $\underline{TF}$  2121 {
2122   \if_meaning:w \c_true_bool #1
2123   \prg_return_true:
2124   \else:
2125   \prg_return_false:
2126   \fi:
2127 }
2128 \cs_generate_variant:Nn \bool_if_p:N { c }
2129 \cs_generate_variant:Nn \bool_if:NT { c }
2130 \cs_generate_variant:Nn \bool_if:NF { c }
2131 \cs_generate_variant:Nn \bool_if:NTF { c }

```

(End definition for `\bool_if:N` and `\bool_if:c`. These functions are documented on page ??.)

`\bool_show:N` Show the truth value of the boolean, as true or false. We use `_msg_show_variable:n`
`\bool_show:c` to get a better output; this function requires its argument to start with `>~`.

```

\bool_show:n 2132 \cs_new_protected:Npn \bool_show:N #1
2133 {

```

```

2134     \bool_if_exist:NTF #1
2135     { \bool_show:n {#1} }
2136     {
2137         \__msg_kernel_error:nxx { kernel } { variable-not-defined }
2138         { \token_to_str:N #1 }
2139     }
2140 }
2141 \cs_new_protected:Npn \bool_show:n #1
2142 {
2143     \bool_if:nTF {#1}
2144     { \__msg_show_variable:n { > ~ true } }
2145     { \__msg_show_variable:n { > ~ false } }
2146 }
2147 \cs_generate_variant:Nn \bool_show:N { c }

```

(End definition for `\bool_show:N`, `\bool_show:c`, and `\bool_show:n`. These functions are documented on page 38.)

`\l_tmpa_bool` A few booleans just if you need them.

```

\l_tmpb_bool 2148 \bool_new:N \l_tmpa_bool
\g_tmpa_bool 2149 \bool_new:N \l_tmpb_bool
\g_tmpb_bool 2150 \bool_new:N \g_tmpa_bool
2151 \bool_new:N \g_tmpb_bool

```

(End definition for `\l_tmpa_bool` and others. These variables are documented on page 38.)

`\bool_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\bool_if_exist_p:c 2152 \prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N { TF , T , F , p }
\bool_if_exist:NTF 2153 \prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c { TF , T , F , p }
\bool_if_exist:cTF

```

(End definition for `\bool_if_exist:N` and `\bool_if_exist:c`. These functions are documented on page ??.)

5.4 Boolean expressions

`\bool_if_p:n` Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with `(and)` for grouping, `!` for logical “Not”, `&&` for logical “And” and `||` for logical “Or”. We shall use the terms Not, And, Or, Open and Close for these operations.

Any expression is terminated by a Close operation. Evaluation happens from left to right in the following manner using a `GetNext` function:

- If an Open is seen, start evaluating a new expression using the `Eval` function and call `GetNext` again.
- If a Not is seen, remove the `!` and call a `GetNotNext` function, which eventually reverses the logic compared to `GetNext`.
- If none of the above, reinsert the token found (this is supposed to be a predicate function) in front of an `Eval` function, which evaluates it to the boolean value `<true>` or `<false>`.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

$\langle true \rangle$ And Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

$\langle false \rangle$ And Current truth value is false, logical And seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return $\langle false \rangle$.

$\langle true \rangle$ Or Current truth value is true, logical Or seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return $\langle true \rangle$.

$\langle false \rangle$ Or Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

$\langle true \rangle$ Close Current truth value is true, Close seen, return $\langle true \rangle$.

$\langle false \rangle$ Close Current truth value is false, Close seen, return $\langle false \rangle$.

We introduce an additional Stop operation with the same semantics as the Close operation.

$\langle true \rangle$ Stop Current truth value is true, return $\langle true \rangle$.

$\langle false \rangle$ Stop Current truth value is false, return $\langle false \rangle$.

The reasons for this follow below.

```

2154 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
2155 {
2156   \if_predicate:w \bool_if_p:n {#1}
2157   \prg_return_true:
2158   \else:
2159   \prg_return_false:
2160   \fi:
2161 }
```

(End definition for `\bool_if:n`. These functions are documented on page 39.)

```

\bool_if_p:n
\_bool_if_left_parentheses:www
\_bool_if_right_parentheses:www
\_bool_if_or:www
```

First issue a `\group_align_safe_begin:` as we are using `&&` as syntax shorthand for the And operation and we need to hide it for T_EX. This will be closed at the end of the expression parsing (see S below).

Minimal (“short-circuit”) evaluation of boolean expressions requires skipping to the end of the current parenthesized group when $\langle true \rangle ||$ is seen, but to the next `||` or closing parenthesis when $\langle false \rangle \&\&$ is seen. To avoid having separate functions for the two cases, we transform the boolean expression by doubling each parenthesis and adding parenthesis around each `||`. This ensures that `&&` will bind tighter than `||`.

The replacement is done in three passes, for left and right parentheses and for `||`. At each pass, the part of the expression that has been transformed is stored before `\q_nil`, the rest lies until the first `\q_mark`, followed by an empty brace group. A trailing marker ensures that the auxiliaries’ delimited arguments will not run-away. As long as

the delimiter matches inside the expression, material is moved before `\q_nil` and we continue. Afterwards, the trailing marker is taken as a delimiter, `#4` is the next auxiliary, immediately followed by a new `\q_nil` delimiter, which indicates that nothing has been treated at this pass. The last step calls `__bool_if_parse:NNNww` which cleans up and triggers the evaluation of the expression itself.

```

2162 \cs_new:Npn \bool_if_p:n #1
2163 {
2164   \group_align_safe_begin:
2165   \__bool_if_left_parentheses:wwwn \q_nil
2166   #1 \q_mark { }
2167   ( \q_mark { \__bool_if_right_parentheses:wwwn \q_nil }
2168   ) \q_mark { \__bool_if_or:wwwn \q_nil }
2169   || \q_mark \__bool_if_parse:NNNww
2170   \q_stop
2171 }
2172 \cs_new:Npn \__bool_if_left_parentheses:wwwn #1 \q_nil #2 ( #3 \q_mark #4
2173 { #4 \__bool_if_left_parentheses:wwwn #1 #2 (( \q_nil #3 \q_mark {#4} }
2174 \cs_new:Npn \__bool_if_right_parentheses:wwwn #1 \q_nil #2 ) #3 \q_mark #4
2175 { #4 \__bool_if_right_parentheses:wwwn #1 #2 )) \q_nil #3 \q_mark {#4} }
2176 \cs_new:Npn \__bool_if_or:wwwn #1 \q_nil #2 || #3 \q_mark #4
2177 { #4 \__bool_if_or:wwwn #1 #2 )||( \q_nil #3 \q_mark {#4} }

```

(End definition for `\bool_if_p:n`. This function is documented on page 39.)

`__bool_if_parse:NNNww` After removing extra tokens from the transformation phase, start evaluating. At the end, we will need to finish the special `align_safe` group before finally returning a `\c_true_bool` or `\c_false_bool` as there might otherwise be something left in front in the input stream. For this we call the `Stop` operation, denoted simply by a `S` following the last `Close` operation.

```

2178 \cs_new:Npn \__bool_if_parse:NNNww #1#2#3#4 \q_mark #5 \q_stop
2179 {
2180   \__bool_get_next:NN \use_i:nn (( #4 )) S
2181 }

```

(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.

```

2182 \cs_new:Npn \__bool_get_next:NN #1#2
2183 {
2184   \use:c
2185   {
2186     __bool_
2187     \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
2188     :Nw
2189   }
2190   #1 #2

```

2191 }
 (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.

2192 \cs_new:cpn { _bool_!:Nw } #1#2
 2193 { \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.

2194 \cs_new:cpn { _bool_(:Nw } #1#2
 2195 {
 2196 \exp_after:wN _bool_choose:NNN \exp_after:wN #1
 2197 _int_value:w _bool_get_next:NN \use_i:nn
 2198 }
 (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.

2199 \cs_new:cpn { _bool_p:Nw } #1
 2200 { \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.

2201 \cs_new:Npn _bool_choose:NNN #1#2#3
 2202 {
 2203 \use:c
 2204 {
 2205 _bool_ #3 _
 2206 #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: }
 2207 :w
 2208 }
 2209 }
 (End definition for _bool_choose:NNN.)

bool)_0:w Closing a group is just about returning the result. The Stop operation is similar except it closes the special alignment group before returning the boolean.

bool)_1:w
 _bool_S_0:w 2210 \cs_new_nopar:cpn { _bool_)_0:w } { \c_false_bool }
 _bool_S_1:w 2211 \cs_new_nopar:cpn { _bool_)_1:w } { \c_true_bool }
 2212 \cs_new_nopar:cpn { _bool_S_0:w } { \group_align_safe_end: \c_false_bool }
 2213 \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` 2214 `\cs_new_nopar:cpn { __bool_&_1:w } & { __bool_get_next:NN \use_i:nn }`
 2215 `\cs_new_nopar:cpn { __bool_|_0:w } | { __bool_get_next:NN \use_i:nn }`

(End definition for `__bool_&_1:w`. This function is documented on page 39.)

`__bool_&_0:w` When the truth value has already been decided, we have to throw away the remainder
`__bool_|_1:w` of the current group as we are doing minimal evaluation. This is slightly tricky as there
`__bool_eval_skip_to_end_auxi:Nw` are no braces so we have to play match the `()` manually.
`__bool_eval_skip_to_end_auxii:Nw`

2216 `\cs_new_nopar:cpn { __bool_&_0:w } & { __bool_eval_skip_to_end_auxi:Nw \c_false_bool }`
 2217 `\cs_new_nopar:cpn { __bool_|_1:w } | { __bool_eval_skip_to_end_auxi:Nw \c_true_bool }`
`__bool_eval_skip_to_end_auxiii:Nw`

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.

```
2218 %% (
2219 \cs_new:Npn \__bool_eval_skip_to_end_auxi:Nw #1#2 )
2220 {
2221   \__bool_eval_skip_to_end_auxii:Nw #1#2 ( % )
2222   \q_no_value \q_stop
2223   {#2}
2224 }
```

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

```
2225 \cs_new:Npn \__bool_eval_skip_to_end_auxii:Nw #1#2 ( #3#4 \q_stop #5 % )
2226 {
2227   \quark_if_no_value:NTF #3
2228   {#1}
2229   { \__bool_eval_skip_to_end_auxiii:Nw #1 #5 }
2230 }
```

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!

```
2231 \cs_new:Npn \__bool_eval_skip_to_end_auxiii:Nw #1#2 ( #3 )
2232 { % (
2233   \__bool_eval_skip_to_end_auxi:Nw #1#3 )
2234 }
```

(End definition for __bool_&_0:w. This function is documented on page 39.)

\bool_not_p:n The Not variant just reverses the outcome of \bool_if_p:n. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```
2235 \cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! ( #1 ) } }
```

(End definition for \bool_not_p:n. This function is documented on page 40.)

\bool_xor_p:nn Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```
2236 \cs_new:Npn \bool_xor_p:nn #1#2
2237 {
2238   \int_compare:nNnTF { \bool_if_p:n {#1} } = { \bool_if_p:n {#2} }
2239   \c_false_bool
2240   \c_true_bool
2241 }
```

(End definition for \bool_xor_p:nn. This function is documented on page 40.)

5.5 Logical loops

\bool_while_do:Nn A while loop where the boolean is tested before executing the statement. The “while” version executes the code as long as the boolean is true; the “until” version executes the code as long as the boolean is false.

\bool_while_do:cn

```

2242 \cs_new:Npn \bool_while_do:Nn #1#2
2243 { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
2244 \cs_new:Npn \bool_until_do:Nn #1#2
2245 { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
2246 \cs_generate_variant:Nn \bool_while_do:Nn { c }
2247 \cs_generate_variant:Nn \bool_until_do:Nn { c }

```

(End definition for \bool_while_do:Nn and \bool_while_do:cn. These functions are documented on page ??.)

\bool_do_while:Nn A do-while loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

\bool_do_while:cn

\bool_do_until:Nn

\bool_do_until:cn

```

2248 \cs_new:Npn \bool_do_while:Nn #1#2
2249 { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
2250 \cs_new:Npn \bool_do_until:Nn #1#2
2251 { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
2252 \cs_generate_variant:Nn \bool_do_while:Nn { c }
2253 \cs_generate_variant:Nn \bool_do_until:Nn { c }

```

(End definition for \bool_do_while:Nn and \bool_do_while:cn. These functions are documented on page ??.)

\bool_while_do:nn Loop functions with the test either before or after the first body expansion.

\bool_do_while:nn

\bool_until_do:nn

\bool_do_until:nn

```

2254 \cs_new:Npn \bool_while_do:nn #1#2
2255 {
2256   \bool_if:nT {#1}
2257   {
2258     #2
2259     \bool_while_do:nn {#1} {#2}
2260   }
2261 }
2262 \cs_new:Npn \bool_do_while:nn #1#2
2263 {
2264   #2
2265   \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
2266 }
2267 \cs_new:Npn \bool_until_do:nn #1#2
2268 {
2269   \bool_if:nF {#1}
2270   {
2271     #2
2272     \bool_until_do:nn {#1} {#2}
2273   }
2274 }
2275 \cs_new:Npn \bool_do_until:nn #1#2
2276 {

```

(End definition for `\bool_while_do:nn` and others. These functions are documented on page 40.)

5.6 Producing n copies

2280 $\langle @@=\text{prg} \rangle$

```

\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:n}`. 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.

```

2281 \cs_new:Npn \prg_replicate:nn #1
2282 {
2283   \__int_to_roman:w
2284     \exp_after:wN \__prg_replicate_first:N
2285     \__int_value:w \__int_eval:w #1 \__int_eval_end:
2286   \cs_end:
2287 }
2288 \cs_new:Npn \__prg_replicate:N #1
2289 { \cs:w \__prg_replicate_#1 :n \__prg_replicate:N }
2290 \cs_new:Npn \__prg_replicate_first:N #1
2291 { \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.

```
2292 \cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
2293 \cs_new:cpn { __prg_replicate_0:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } }
2294 \cs_new:cpn { __prg_replicate_1:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1 }
2295 \cs_new:cpn { __prg_replicate_2:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1#1 }
2296 \cs_new:cpn { __prg_replicate_3:n } #1
```

Users shouldn't ask for something to be replicated once or even not at all but...

End definition for \prg_replicate:nn. This function is documented on page 41.)

5.7 Detecting T_EX's mode

`\mode_if_vertical_p:` For testing vertical mode. Strikes me here on the bus with David, that as long as
`\mode_if_vertical:TF` we are just talking about returning true and false states, we can just use the primitive
conditionals for this and gobbling the `\c_zero` in the input stream. However this requires
knowledge of the implementation so we keep things nice and clean and use the return
statements.

```
2325 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
2326 { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }
```

End definition for \mode_if_vertical:. These functions are documented on page 41.)

`\mode_if_horizontal_p:` For testing horizontal mode.

```

\mode_if_horizontal:TF 2327 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
2328 { \if mode horizontal: \prg_return true: \else: \prg_return false: \fi: }

```

End definition for \mode_if_horizontal:. These functions are documented on page 41.)

`\mode_if_inner_p:` For testing inner mode.
`\mode_if_inner:TF`

```

2329 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
2330 { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_inner:`. These functions are documented on page 41.)

`\mode_if_math_p:` For testing math mode. At the beginning of an alignment cell, the programmer should
`\mode_if_math:TF` insert `\scan_align_safe_stop:` before the test.

```

2331 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
2332 { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_math:`. These functions are documented on page 41.)

5.8 Internal programming functions

`\group_align_safe_begin:` `\group_align_safe_end:` T_EX’s alignment structures present many problems. As Knuth says himself in *T_EX: The Program*: “It’s sort of a miracle whenever `\halign` or `\valign` work, [...]” One problem relates to commands that internally issues a `\cr` but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a `&` with category code 4 we will get some sort of weird error message because the underlying `\futurelet` will store the token at the end of the alignment template. This could be a `&4` giving a message like `! Misplaced \cr.` or even worse: it could be the `\endtemplate` token causing even more trouble! To solve this we have to open a special group so that T_EX still thinks it’s on safe ground but at the same time we don’t want to introduce any brace group that may find its way to the output. The following functions help with this by using code documented only in Appendix D of *The T_EXbook*... We place the `\if_false: { \fi: }` part at that place so that the successive expansions of `\group_align_safe_begin/end:` are always brace balanced.

```

2333 \cs_new_nopar:Npn \group_align_safe_begin:
2334 { \if_int_compare:w \if_false: { \fi: ' } = \c_zero \fi: }
2335 \cs_new_nopar:Npn \group_align_safe_end:
2336 { \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.

```

2337 \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:.`)

```
2338 <@@=prg>
```

```
\__prg_variable_get_scope:N
\__prg_variable_get_scope:w
\__prg_variable_get_type:N
\__prg_variable_get_type:w
```

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.

```
2339 \group_begin:
2340   \tex_lccode:D '* = 'g \scan_stop:
2341   \tex_catcode:D '* = \c_twelve
2342   \tl_to_lowercase:n
2343   {
2344     \group_end:
2345     \cs_new:Npn \__prg_variable_get_scope:N #1
2346     {
2347       \exp_after:wN \exp_after:wN
2348       \exp_after:wN \__prg_variable_get_scope:w
2349       \cs_to_str:N #1 \exp_stop_f: \q_stop
2350     }
2351     \cs_new:Npn \__prg_variable_get_scope:w #1#2 \q_stop
2352     { \token_if_eq_meaning:NNT * #1 { g } }
2353   }
2354 \group_begin:
2355   \tex_lccode:D '* = ' _ \scan_stop:
2356   \tex_catcode:D '* = \c_twelve
2357   \tl_to_lowercase:n
2358   {
2359     \group_end:
2360     \cs_new:Npn \__prg_variable_get_type:N #1
```

```

2361     {
2362         \exp_after:wN \__prg_variable_get_type:w
2363         \token_to_str:N #1 * a \q_stop
2364     }
2365     \cs_new:Npn \__prg_variable_get_type:w #1 * #2#3 \q_stop
2366     {
2367         \token_if_eq_meaning:NNTF a #2
2368         {#1}
2369         { \__prg_variable_get_type:w #2#3 \q_stop }
2370     }
2371 }

```

(End definition for `__prg_variable_get_scope:N`. This function is documented on page 42.)

`\g__prg_map_int` A nesting counter for mapping.

```

2372 \int_new:N \g__prg_map_int

```

(End definition for `\g__prg_map_int`. This variable is documented on page 43.)

`__prg_break_point:Nn` These are defined in `l3basics`, as they are needed “early”. This is just a reminder that that is the case!

`__prg_map_break:Nn`

(End definition for `__prg_break_point:Nn`. This function is documented on page 43.)

`__prg_break_point:` Also done in `l3basics` as in format mode these are needed within `l3alloc`.

`__prg_break:`

(End definition for `__prg_break_point:.` This function is documented on page ??.)

`__prg_break:n`

5.9 Deprecated functions

These were deprecated on 2012-02-08, and will be removed entirely by 2012-05-31.

`\prg_define_quicksort:nnn` #1 is the name, #2 and #3 are the tokens enclosing the argument. For the somewhat strange `<clist>` type which doesn’t enclose the items but uses a separator we define it by hand afterwards. When doing the first pass, the algorithm wraps all elements in braces and then uses a generic quicksort which works on token lists.

As an example

```

\prg_define_quicksort:nnn{seq}{\seq_elt:w}{\seq_elt_end:w}

```

defines the user function `\seq_quicksort:n` and furthermore expects to use the two functions `\seq_quicksort_compare:nnTF` which compares the items and `\seq_quicksort_function:n` which is placed before each sorted item. It is up to the programmer to define these functions when needed. For the `seq` type a sequence is a token list variable, so one additionally has to define

```

\cs_set_nopar:Npn \seq_quicksort:N{\exp_args:No\seq_quicksort:n}

```

For details on the implementation see “Sorting in TeX’s Mouth” by Bernd Raichle. Firstly we define the function for parsing the initial list and then the braced list afterwards.

```

2373 <*deprecated>
2374 \cs_new_protected:Npn \prg_define_quicksort:nnn #1#2#3 {

```

```

2375 \cs_set:cpx{#1_quicksort:n}##1{
2376   \exp_not:c{#1_quicksort_start_partition:w} ##1
2377   \exp_not:n{#2\q_nil#3\q_stop}
2378 }
2379 \cs_set:cpx{#1_quicksort_braced:n}##1{
2380   \exp_not:c{#1_quicksort_start_partition_braced:n} ##1
2381   \exp_not:N\q_nil\exp_not:N\q_stop
2382 }
2383 \cs_set:cpx {#1_quicksort_start_partition:w} #2 ##1 #3{
2384   \exp_not:N \quark_if_nil:nTF {##1}\exp_not:N \use_none_delimit_by_q_stop:w
2385   \exp_not:c{#1_quicksort_do_partition_i:nnnw} {##1}{-}{-}
2386 }
2387 \cs_set:cpx {#1_quicksort_start_partition_braced:n} ##1 {
2388   \exp_not:N \quark_if_nil:nTF {##1}\exp_not:N \use_none_delimit_by_q_stop:w
2389   \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn} {##1}{-}{-}
2390 }
2391 </deprecated>

```

Now for doing the partitions.

```

2392 <*deprecated>
2393 \cs_set:cpx {#1_quicksort_do_partition_i:nnnw} ##1##2##3 #2 ##4 #3 {
2394   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2395   {
2396     \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2397     \exp_not:c{#1_quicksort_partition_greater_ii:nnnn}
2398     \exp_not:c{#1_quicksort_partition_less_ii:nnnn}
2399   }
2400   {##1}{##2}{##3}{##4}
2401 }
2402 \cs_set:cpx {#1_quicksort_do_partition_i_braced:nnnn} ##1##2##3##4 {
2403   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2404   {
2405     \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2406     \exp_not:c{#1_quicksort_partition_greater_ii_braced:nnnn}
2407     \exp_not:c{#1_quicksort_partition_less_ii_braced:nnnn}
2408   }
2409   {##1}{##2}{##3}{##4}
2410 }
2411 \cs_set:cpx {#1_quicksort_do_partition_ii:nnnw} ##1##2##3 #2 ##4 #3 {
2412   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2413   {
2414     \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2415     \exp_not:c{#1_quicksort_partition_less_i:nnnn}
2416     \exp_not:c{#1_quicksort_partition_greater_i:nnnn}
2417   }
2418   {##1}{##2}{##3}{##4}
2419 }
2420 \cs_set:cpx {#1_quicksort_do_partition_ii_braced:nnnn} ##1##2##3##4 {
2421   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2422   {

```



```

2423     \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2424     \exp_not:c{#1_quicksort_partition_less_i_braced:nnnn}
2425     \exp_not:c{#1_quicksort_partition_greater_i_braced:nnnn}
2426   }
2427   {##1}{##2}{##3}{##4}
2428 }
2429 </deprecated>

```

This part of the code handles the two branches in each sorting. Again we will also have to do it braced.

```

2430 <*deprecated>
2431 \cs_set:cpx {#1_quicksort_partition_less_i:nnnn} ##1##2##3##4{
2432   \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##2}{##4}{##3}}
2433 \cs_set:cpx {#1_quicksort_partition_less_ii:nnnn} ##1##2##3##4{
2434   \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##3}{##4}}
2435 \cs_set:cpx {#1_quicksort_partition_greater_i:nnnn} ##1##2##3##4{
2436   \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##4}{##2}{##3}}
2437 \cs_set:cpx {#1_quicksort_partition_greater_ii:nnnn} ##1##2##3##4{
2438   \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##4}{##3}}
2439 \cs_set:cpx {#1_quicksort_partition_less_i_braced:nnnn} ##1##2##3##4{
2440   \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##2}{##4}{##3}}
2441 \cs_set:cpx {#1_quicksort_partition_less_ii_braced:nnnn} ##1##2##3##4{
2442   \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2}{##3}{##4}}
2443 \cs_set:cpx {#1_quicksort_partition_greater_i_braced:nnnn} ##1##2##3##4{
2444   \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##4}{##2}{##3}}
2445 \cs_set:cpx {#1_quicksort_partition_greater_ii_braced:nnnn} ##1##2##3##4{
2446   \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2}{##4}{##3}}
2447 </deprecated>

```

Finally, the big kahuna! This is where the sub-lists are sorted.

```

2448 <*deprecated>
2449 \cs_set:cpx {#1_do_quicksort_braced:nnnnw} ##1##2##3##4\q_stop {
2450   \exp_not:c{#1_quicksort_braced:n}{##2}
2451   \exp_not:c{#1_quicksort_function:n}{##1}
2452   \exp_not:c{#1_quicksort_braced:n}{##3}
2453 }
2454 }
2455 </deprecated>

```

(End definition for \prg_define_quicksort:nnn.)

\prg_quicksort:n A simple version. Sorts a list of tokens, uses the function `\prg_quicksort_compare:nnTF` to compare items, and places the function `\prg_quicksort_function:n` in front of each of them.

```

2456 <*deprecated>
2457 \prg_define_quicksort:nnn {prg}{}{ }
2458 </deprecated>

```

(End definition for \prg_quicksort:n. This function is documented on page ??.)

\prg_quicksort_function:n
\prg_quicksort_compare:nnTF

```

2459 <*deprecated>

```

```

2460 \cs_set:Npn \prg_quicksort_function:n {\ERROR}
2461 \cs_set:Npn \prg_quicksort_compare:nnTF {\ERROR}
2462 \deprecated
(End definition for \prg_quicksort_function:n. This function is documented on page ??.)
These were deprecated on 2011-05-27 and will be removed entirely by 2011-08-31.

```

```

\prg_new_map_functions:Nn As we have restructured the structured variables, these are no longer needed.
\prg_set_map_functions:Nn
2463 \*deprecated
2464 \cs_new_protected:Npn \prg_new_map_functions:Nn #1#2 { \deprecated }
2465 \cs_new_protected:Npn \prg_set_map_functions:Nn #1#2 { \deprecated }
2466 \deprecated
(End definition for \prg_new_map_functions:Nn. This function is documented on page ??.)
Deprecated 2012-06-03 for removal after 2012-12-31.

```

```

\prg_case_int:nnn Moved to more sensible modules.
\prg_case_str:nnn
\prg_case_str:onn
\prg_case_str:xxn
\prg_case_tl:Nnn
\prg_case_tl:cnn
2467 \*deprecated
2468 \cs_new_eq:NN \prg_case_int:nnn \int_case:nnn
2469 \cs_new_eq:NN \prg_case_str:nnn \str_case:nnn
2470 \cs_new_eq:NN \prg_case_str:onn \str_case:onn
2471 \cs_new_eq:NN \prg_case_str:xxn \str_case:x:nnn
2472 \cs_new_eq:NN \prg_case_tl:Nnn \tl_case:Nnn
2473 \cs_new_eq:NN \prg_case_tl:cnn \tl_case:cnn
2474 \deprecated
(End definition for \prg_case_int:nnn and others. These functions are documented on page ??.)
Deprecated 2012-06-04 for removal after 2012-12-31.

```

```

\prg_stepwise_function:nnnN
\prg_stepwise_inline:nnnn
\prg_stepwise_variable:nnnNn
2475 \*deprecated
2476 \cs_new_eq:NN \prg_stepwise_function:nnnN \int_step_function:nnnN
2477 \cs_new_eq:NN \prg_stepwise_inline:nnnn \int_step_inline:nnnn
2478 \cs_new_eq:NN \prg_stepwise_variable:nnnNn \int_step_variable:nnnNn
2479 \deprecated
(End definition for \prg_stepwise_function:nnnN, \prg_stepwise_inline:nnnn, and \prg_stepwise_variable:nnnNn.
These functions are documented on page ??.)
2480 \initex | package)

```

6 l3quark implementation

The following test files are used for this code: m3quark001.lvt.

```

2481 \*initex | package)
2482 \*package)
2483 \ProvidesExplPackage
2484 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
2485 \__expl_package_check:
2486 \package)

```

6.1 Quarks

`\quark_new:N` Allocate a new quark.

```
2487 \cs_new_protected:Npn \quark_new:N #1 { \tl_const:Nn #1 {#1} }
```

(End definition for `\quark_new:N`. This function is documented on page 45.)

`\q_nil` Some “public” quarks. `\q_stop` is an “end of argument” marker, `\q_nil` is a empty value and `\q_no_value` marks an empty argument.

`\q_mark`

`\q_no_value`

`\q_stop`

```
2488 \quark_new:N \q_nil
2489 \quark_new:N \q_mark
2490 \quark_new:N \q_no_value
2491 \quark_new:N \q_stop
```

(End definition for `\q_nil` and others. These variables are documented on page 45.)

`\q_recursion_tail`

`\q_recursion_stop`

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.

```
2492 \quark_new:N \q_recursion_tail
2493 \quark_new:N \q_recursion_stop
```

(End definition for `\q_recursion_tail` and `\q_recursion_stop`. These variables are documented on page 46.)

`\quark_if_recursion_tail_stop:N`
`\quark_if_recursion_tail_stop_do:Nn`

When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapper up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```
2494 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
2495 {
2496   \if_meaning:w \q_recursion_tail #1
2497   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2498   \fi:
2499 }
2500 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
2501 {
2502   \if_meaning:w \q_recursion_tail #1
2503   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2504   \else:
2505     \exp_after:wN \use_none:n
2506   \fi:
2507 }
```

(End definition for `\quark_if_recursion_tail_stop:N`. This function is documented on page 46.)

`\quark_if_recursion_tail_stop:n` The same idea applies when testing multiple tokens, but here we just compare the token
`\quark_if_recursion_tail_stop:o` list to `\q_recursion_tail` as a string.
`\quark_if_recursion_tail_stop_do:nn`

```

2508 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
2509 {
2510   \if_int_compare:w \pdfTeX_strcmp:D
2511     { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
2512     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2513   \fi:
2514 }
2515 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
2516 {
2517   \if_int_compare:w \pdfTeX_strcmp:D
2518     { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
2519     \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2520   \else:
2521     \exp_after:wN \use_none:n
2522   \fi:
2523 }
2524 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
2525 \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
`__quark_if_recursion_tail_break:nN` using #2.

```

2526 \cs_new:Npn \__quark_if_recursion_tail_break:NN #1#2
2527 {
2528   \if_meaning:w \q_recursion_tail #1
2529   \exp_after:wN #2
2530   \fi:
2531 }
2532 \cs_new:Npn \__quark_if_recursion_tail_break:nN #1#2
2533 {
2534   \if_int_compare:w \pdfTeX_strcmp:D
2535     { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
2536     \exp_after:wN #2
2537   \fi:
2538 }

```

(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:N TF` `\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

```

2539 \prg_new_conditional:Nnn \quark_if_nil:N { p, T , F , TF }
2540 {
2541   \if_meaning:w \q_nil #1
2542   \prg_return_true:

```

⁴It may still loop in special circumstances however!

```

2543     \else:
2544         \prg_return_false:
2545     \fi:
2546 }
2547 \prg_new_conditional:Nnn \quark_if_no_value:N { p, T , F , TF }
2548 {
2549     \if_meaning:w \q_no_value #1
2550     \prg_return_true:
2551 } \else:
2552     \prg_return_false:
2553 \fi:
2554 }
2555 \cs_generate_variant:Nn \quark_if_no_value_p:N { c }
2556 \cs_generate_variant:Nn \quark_if_no_value:NT { c }
2557 \cs_generate_variant:Nn \quark_if_no_value:NF { c }
2558 \cs_generate_variant:Nn \quark_if_no_value:NTF { c }

```

(End definition for \quark_if_nil:N. These functions are documented on page ??.)

\quark_if_nil_p:n These are essentially \str_if_eq:nn tests but done directly.

\quark_if_nil_p:V 2559 \prg_new_conditional:Nnn \quark_if_nil:n { p, T , F , TF }

\quark_if_nil_p:o 2560 {

\quark_if_nil:nTF 2561 \if_int_compare:w \pdfTeX_strcmp:D

\quark_if_nil:VTF 2562 { \exp_not:N \q_nil } { \exp_not:n {#1} } = \c_zero

\quark_if_nil:oTF 2563 \prg_return_true:

\quark_if_no_value_p:n 2564 \else:

\quark_if_no_value:nTF 2565 \prg_return_false:

2566 \fi:

2567 }

2568 \prg_new_conditional:Nnn \quark_if_no_value:n { p, T , F , TF }

2569 {

2570 \if_int_compare:w \pdfTeX_strcmp:D

2571 { \exp_not:N \q_no_value } { \exp_not:n {#1} } = \c_zero

2572 \prg_return_true:

2573 \else:

2574 \prg_return_false:

2575 \fi:

2576 }

2577 \cs_generate_variant:Nn \quark_if_nil_p:n { V , o }

2578 \cs_generate_variant:Nn \quark_if_nil:nTF { V , o }

2579 \cs_generate_variant:Nn \quark_if_nil:nT { V , o }

2580 \cs_generate_variant:Nn \quark_if_nil:nF { V , o }

(End definition for \quark_if_nil:n, \quark_if_nil:V, and \quark_if_nil:o. These functions are documented on page 45.)

\q__tl_act_mark These private quarks are needed by l3tl, but that is loaded before the quark module,

\q__tl_act_stop hence their definition is deferred.

```

2581 \quark_new:N \q__tl_act_mark
2582 \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

2583 <@@=scan>

`\g__scan_marks_tl` The list of all scan marks currently declared.

2584 `\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.

```
2585 \cs_new_protected:Npn \__scan_new:N #1
2586 {
2587   \tl_if_in:NnTF \g__scan_marks_tl { #1 }
2588   {
2589     \__msg_kernel_error:nxx { kernel } { scanmark-already-defined }
2590     { \token_to_str:N #1 }
2591   }
2592   {
2593     \tl_gput_right:Nn \g__scan_marks_tl {#1}
2594     \cs_new_eq:NN #1 \scan_stop:
2595   }
2596 }
```

(End definition for `__scan_new:N`.)

`\s__stop` We only declare one scan mark here, more can be defined by specific modules.

2597 `__scan_new:N \s__stop`

(End definition for `\s__stop`. This variable is documented on page 48.)

`__use_none_delimit_by_s__stop:w` Similar to `\use_none_delimit_by_q_stop:w`.

2598 `\cs_new:Npn __use_none_delimit_by_s__stop:w #1 \s__stop { }`

(End definition for `__use_none_delimit_by_s__stop:w`.)

6.3 Deprecated quark functions

`\quark_if_recursion_tail_break:N` It's not clear what breaking function we should be using here, so I'm picking one somewhat arbitrarily.
`\quark_if_recursion_tail_break:n`

```
2599 \cs_new:Npn \quark_if_recursion_tail_break:N #1
2600 { \__quark_if_recursion_tail_break:NN #1 \prg_break: }
2601 \cs_new:Npn \quark_if_recursion_tail_break:n #1
2602 { \__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 ??.)

2603 </initex | package>

7 l3token implementation

```

2604 <*initex | package>
2605 <@@=token>
2606 <*package>
2607 \ProvidesExplPackage
2608   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
2609   \_expl_package_check:
2610 </package>

```

7.1 Character tokens

Category code changes.

```

\char_set_catcode:nn
\char_value_catcode:n
\char_show_value_catcode:n
2611 \cs_new_protected:Npn \char_set_catcode:nn #1#2
2612   { \tex_catcode:D #1 = \_int_eval:w #2 \_int_eval_end: }
2613 \cs_new:Npn \char_value_catcode:n #1
2614   { \tex_the:D \tex_catcode:D \_int_eval:w #1 \_int_eval_end: }
2615 \cs_new_protected:Npn \char_show_value_catcode:n #1
2616   { \tex_showthe:D \tex_catcode:D \_int_eval:w #1 \_int_eval_end: }

```

(End definition for `\char_set_catcode:nn`. This function is documented on page 51.)

```

\char_set_catcode_escape:N
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N
2617 \cs_new_protected:Npn \char_set_catcode_escape:N #1
2618   { \char_set_catcode:nn { '#1 } \c_zero }
2619 \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
2620   { \char_set_catcode:nn { '#1 } \c_one }
2621 \cs_new_protected:Npn \char_set_catcode_group_end:N #1
2622   { \char_set_catcode:nn { '#1 } \c_two }
2623 \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
2624   { \char_set_catcode:nn { '#1 } \c_three }
2625 \cs_new_protected:Npn \char_set_catcode_alignment:N #1
2626   { \char_set_catcode:nn { '#1 } \c_four }
2627 \cs_new_protected:Npn \char_set_catcode_end_line:N #1
2628   { \char_set_catcode:nn { '#1 } \c_five }
2629 \cs_new_protected:Npn \char_set_catcode_parameter:N #1
2630   { \char_set_catcode:nn { '#1 } \c_six }
2631 \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
2632   { \char_set_catcode:nn { '#1 } \c_seven }
2633 \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
2634   { \char_set_catcode:nn { '#1 } \c_eight }
2635 \cs_new_protected:Npn \char_set_catcode_ignore:N #1
2636   { \char_set_catcode:nn { '#1 } \c_nine }
2637 \cs_new_protected:Npn \char_set_catcode_space:N #1
2638   { \char_set_catcode:nn { '#1 } \c_ten }
2639 \cs_new_protected:Npn \char_set_catcode_letter:N #1
2640   { \char_set_catcode:nn { '#1 } \c_eleven }
2641 \cs_new_protected:Npn \char_set_catcode_other:N #1
2642   { \char_set_catcode:nn { '#1 } \c_twelve }
2643 \cs_new_protected:Npn \char_set_catcode_active:N #1

```

```

2644 { \char_set_catcode:nn { '#1 } \c_thirteen }
2645 \cs_new_protected:Npn \char_set_catcode_comment:N #1
2646 { \char_set_catcode:nn { '#1 } \c_fourteen }
2647 \cs_new_protected:Npn \char_set_catcode_invalid:N #1
2648 { \char_set_catcode:nn { '#1 } \c_fifteen }

```

(End definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 50.)

```

\char_set_catcode_escape:n
  \char_set_catcode_group_begin:n 2649 \cs_new_protected:Npn \char_set_catcode_escape:n #1
  \char_set_catcode_group_end:n    2650 { \char_set_catcode:nn {#1} \c_zero }
  \char_set_catcode_math_toggle:n  2651 \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
  \char_set_catcode_alignment:n    2652 { \char_set_catcode:nn {#1} \c_one }
\char_set_catcode_end_line:n       2653 \cs_new_protected:Npn \char_set_catcode_group_end:n #1
  \char_set_catcode_parameter:n    2654 { \char_set_catcode:nn {#1} \c_two }
  \char_set_catcode_math_superscript:n 2655 \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
  \char_set_catcode_math_subscript:n 2656 { \char_set_catcode:nn {#1} \c_three }
  \char_set_catcode_ignore:n       2657 \cs_new_protected:Npn \char_set_catcode_alignment:n #1
  \char_set_catcode_space:n        2658 { \char_set_catcode:nn {#1} \c_four }
  \char_set_catcode_letter:n       2659 \cs_new_protected:Npn \char_set_catcode_end_line:n #1
  \char_set_catcode_other:n        2660 { \char_set_catcode:nn {#1} \c_five }
  \char_set_catcode_active:n       2661 \cs_new_protected:Npn \char_set_catcode_parameter:n #1
  \char_set_catcode_comment:n      2662 { \char_set_catcode:nn {#1} \c_six }
  \char_set_catcode_invalid:n      2663 \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
  \char_set_catcode_invalid:n      2664 { \char_set_catcode:nn {#1} \c_seven }
  \char_set_catcode_invalid:n      2665 \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
  \char_set_catcode_invalid:n      2666 { \char_set_catcode:nn {#1} \c_eight }
  \char_set_catcode_invalid:n      2667 \cs_new_protected:Npn \char_set_catcode_ignore:n #1
  \char_set_catcode_invalid:n      2668 { \char_set_catcode:nn {#1} \c_nine }
  \char_set_catcode_invalid:n      2669 \cs_new_protected:Npn \char_set_catcode_space:n #1
  \char_set_catcode_invalid:n      2670 { \char_set_catcode:nn {#1} \c_ten }
  \char_set_catcode_invalid:n      2671 \cs_new_protected:Npn \char_set_catcode_letter:n #1
  \char_set_catcode_invalid:n      2672 { \char_set_catcode:nn {#1} \c_eleven }
  \char_set_catcode_invalid:n      2673 \cs_new_protected:Npn \char_set_catcode_other:n #1
  \char_set_catcode_invalid:n      2674 { \char_set_catcode:nn {#1} \c_twelve }
  \char_set_catcode_invalid:n      2675 \cs_new_protected:Npn \char_set_catcode_active:n #1
  \char_set_catcode_invalid:n      2676 { \char_set_catcode:nn {#1} \c_thirteen }
  \char_set_catcode_invalid:n      2677 \cs_new_protected:Npn \char_set_catcode_comment:n #1
  \char_set_catcode_invalid:n      2678 { \char_set_catcode:nn {#1} \c_fourteen }
  \char_set_catcode_invalid:n      2679 \cs_new_protected:Npn \char_set_catcode_invalid:n #1
  \char_set_catcode_invalid:n      2680 { \char_set_catcode:nn {#1} \c_fifteen }

```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 50.)

```

\char_set_mathcode:nn Pretty repetitive, but necessary!
\char_value_mathcode:n 2681 \cs_new_protected:Npn \char_set_mathcode:nn #1#2
\char_show_value_mathcode:n 2682 { \tex_mathcode:D #1 = \__int_eval:w #2 \__int_eval_end: }
  \char_set_lccode:nn 2683 \cs_new:Npn \char_value_mathcode:n #1
  \char_value_lccode:n 2684 { \tex_the:D \tex_mathcode:D \__int_eval:w #1 \__int_eval_end: }
\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

```



```

2685 \cs_new_protected:Npn \char_show_value_mathcode:n #1
2686 { \tex_showthe:D \tex_mathcode:D \__int_eval:w #1 \__int_eval_end: }
2687 \cs_new_protected:Npn \char_set_lccode:nn #1#2
2688 { \tex_lccode:D #1 = \__int_eval:w #2 \__int_eval_end: }
2689 \cs_new:Npn \char_value_lccode:n #1
2690 { \tex_the:D \tex_lccode:D \__int_eval:w #1 \__int_eval_end: }
2691 \cs_new_protected:Npn \char_show_value_lccode:n #1
2692 { \tex_showthe:D \tex_lccode:D \__int_eval:w #1 \__int_eval_end: }
2693 \cs_new_protected:Npn \char_set_uccode:nn #1#2
2694 { \tex_uccode:D #1 = \__int_eval:w #2 \__int_eval_end: }
2695 \cs_new:Npn \char_value_uccode:n #1
2696 { \tex_the:D \tex_uccode:D \__int_eval:w #1 \__int_eval_end: }
2697 \cs_new_protected:Npn \char_show_value_uccode:n #1
2698 { \tex_showthe:D \tex_uccode:D \__int_eval:w #1 \__int_eval_end: }
2699 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
2700 { \tex_sfcode:D #1 = \__int_eval:w #2 \__int_eval_end: }
2701 \cs_new:Npn \char_value_sfcode:n #1
2702 { \tex_the:D \tex_sfcode:D \__int_eval:w #1 \__int_eval_end: }
2703 \cs_new_protected:Npn \char_show_value_sfcode:n #1
2704 { \tex_showthe:D \tex_sfcode:D \__int_eval:w #1 \__int_eval_end: }

```

(End definition for `\char_set_mathcode:nn`. This function is documented on page 53.)

7.2 Generic tokens

`\token_to_meaning:N` These are all defined in `l3basics`, as they are needed “early”. This is just a reminder that
`\token_to_meaning:c` that is the case!
`\token_to_str:N` (End definition for `\token_to_meaning:N` and `\token_to_meaning:c`. These functions are documented
`\token_to_str:c` on page ??.)

`\token_new:Nn` Creates a new token.

```

2705 \cs_new_protected:Npn \token_new:Nn #1#2 { \cs_new_eq:NN #1 #2 }

```

(End definition for `\token_new:Nn`. This function is documented on page 53.)

`\c_group_begin_token` We define these useful tokens. We have to do it by hand with the brace tokens for obvious
`\c_group_end_token` reasons.

```

\c_math_toggle_token 2706 \cs_new_eq:NN \c_group_begin_token {
\c_alignment_token 2707 \cs_new_eq:NN \c_group_end_token }
\c_parameter_token 2708 \group_begin:
\c_math_superscript_token 2709 \char_set_catcode_math_toggle:N \*
\c_math_subscript_token 2710 \token_new:Nn \c_math_toggle_token { * }
\c_space_token 2711 \char_set_catcode_alignment:N \*
\c_catcode_letter_token 2712 \token_new:Nn \c_alignment_token { * }
\c_catcode_other_token 2713 \token_new:Nn \c_parameter_token { # }
2714 \token_new:Nn \c_math_superscript_token { ^ }
2715 \char_set_catcode_math_subscript:N \*
2716 \token_new:Nn \c_math_subscript_token { * }
2717 \token_new:Nn \c_space_token { ~ }
2718 \token_new:Nn \c_catcode_letter_token { a }
2719 \token_new:Nn \c_catcode_other_token { 1 }
2720 \group_end:

```

(End definition for `\c_group_begin_token` and others. These functions are documented on page 53.)

`\c_catcode_active_tl` Not an implicit token!

```

2721 \group_begin:
2722   \char_set_catcode_active:N \*
2723   \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
2724 \group_end:

```

(End definition for `\c_catcode_active_tl`. This variable is documented on page 53.)

`\l_char_active_seq` Two sequences for dealing with special characters. The first is characters which may be active, and contains the active characters themselves to allow easy redefinition. The 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`.

`\l_char_special_seq`

```

2725 \seq_new:N \l_char_active_seq
2726 \use:n
2727 {
2728   \group_begin:
2729   \char_set_catcode_active:N \"
2730   \char_set_catcode_active:N \$
2731   \char_set_catcode_active:N &
2732   \char_set_catcode_active:N ^
2733   \char_set_catcode_active:N _
2734   \char_set_catcode_active:N ~
2735   \use:nn
2736   {
2737     \group_end:
2738     \seq_set_split:Nnn \l_char_active_seq { }
2739   }
2740 }
2741 { { " $ & ^ _ ~ } } %$
2742 \seq_new:N \l_char_special_seq
2743 \seq_set_split:Nnn \l_char_special_seq { }
2744 { \ \ " \# \$ \% \& \ \ ^ \_ \{ \} \~ }

```

(End definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 53.)

7.3 Token conditionals

`\token_if_group_begin_p:N` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.
`\token_if_group_begin:NTF`

```

2745 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
2746 {
2747   \if_catcode:w \exp_not:N #1 \c_group_begin_token
2748   \prg_return_true: \else: \prg_return_false: \fi:
2749 }

```

(End definition for `\token_if_group_begin:N`. These functions are documented on page 54.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.
`\token_if_group_end:N \textit{TF}`

```
2750 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
2751 {
2752   \if_catcode:w \exp_not:N #1 \c_group_end_token
2753   \prg_return_true: \else: \prg_return_false: \fi:
2754 }
```

(End definition for `\token_if_group_end:N`. These functions are documented on page 54.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.
`\token_if_math_toggle:N \textit{TF}`

```
2755 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
2756 {
2757   \if_catcode:w \exp_not:N #1 \c_math_toggle_token
2758   \prg_return_true: \else: \prg_return_false: \fi:
2759 }
```

(End definition for `\token_if_math_toggle:N`. These functions are documented on page 54.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.
`\token_if_alignment:N \textit{TF}`

```
2760 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
2761 {
2762   \if_catcode:w \exp_not:N #1 \c_alignment_token
2763   \prg_return_true: \else: \prg_return_false: \fi:
2764 }
```

(End definition for `\token_if_alignment:N`. These functions are documented on page 54.)

`\token_if_parameter_p:N` Check if token is a parameter token. We use the constant `\c_parameter_token` for this.
`\token_if_parameter:N \textit{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.

```
2765 \group_begin:
2766 \cs_set_eq:NN \c_parameter_token \scan_stop:
2767 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
2768 {
2769   \if_catcode:w \exp_not:N #1 \c_parameter_token
2770   \prg_return_true: \else: \prg_return_false: \fi:
2771 }
2772 \group_end:
```

(End definition for `\token_if_parameter:N`. These functions are documented on page 55.)

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.
`\token_if_math_superscript:N \textit{TF}`

```
2773 \prg_new_conditional:Npnn \token_if_math_superscript:N #1 { p , T , F , TF }
2774 {
2775   \if_catcode:w \exp_not:N #1 \c_math_superscript_token
2776   \prg_return_true: \else: \prg_return_false: \fi:
2777 }
```

(End definition for `\token_if_math_superscript:N`. These functions are documented on page 55.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token` for this.

```

2778 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
2779 {
2780   \if_catcode:w \exp_not:N #1 \c_math_subscript_token
2781   \prg_return_true: \else: \prg_return_false: \fi:
2782 }

```

(End definition for \token_if_math_subscript:N. These functions are documented on page 55.)

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.

```

2783 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
2784 {
2785   \if_catcode:w \exp_not:N #1 \c_space_token
2786   \prg_return_true: \else: \prg_return_false: \fi:
2787 }

```

(End definition for \token_if_space:N. These functions are documented on page 55.)

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.

```

2788 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
2789 {
2790   \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
2791   \prg_return_true: \else: \prg_return_false: \fi:
2792 }

```

(End definition for \token_if_letter:N. These functions are documented on page 55.)

`\token_if_other_p:N` Check if token is an other char token. We use the constant `\c_catcode_other_token` for this.

```

2793 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
2794 {
2795   \if_catcode:w \exp_not:N #1 \c_catcode_other_token
2796   \prg_return_true: \else: \prg_return_false: \fi:
2797 }

```

(End definition for \token_if_other:N. These functions are documented on page 55.)

`\token_if_active_p:N` Check if token is an active char token. We use the constant `\c_catcode_active_tl` for this. A technical point is that `\c_catcode_active_tl` is in fact a macro expanding to `\exp_not:N *`, where `*` is active.

```

2798 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
2799 {
2800   \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
2801   \prg_return_true: \else: \prg_return_false: \fi:
2802 }

```

(End definition for \token_if_active:N. These functions are documented on page 55.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.
`\token_if_eq_meaning:NNTF`

```

2803 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
2804 {
2805     \if_meaning:w #1 #2
2806     \prg_return_true: \else: \prg_return_false: \fi:
2807 }

```

(End definition for `\token_if_eq_meaning:NN`. These functions are documented on page 56.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.
`\token_if_eq_catcode:NNTF`

```

2808 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
2809 {
2810     \if_catcode:w \exp_not:N #1 \exp_not:N #2
2811     \prg_return_true: \else: \prg_return_false: \fi:
2812 }

```

(End definition for `\token_if_eq_catcode:NN`. These functions are documented on page 55.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.
`\token_if_eq_charcode:NNTF`

```

2813 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
2814 {
2815     \if_charcode:w \exp_not:N #1 \exp_not:N #2
2816     \prg_return_true: \else: \prg_return_false: \fi:
2817 }

```

(End definition for `\token_if_eq_charcode:NN`. These functions are documented on page 55.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` will always output something like
`\token_if_macro:NTF` `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`.
`_token_if_macro_p:w` However, this can fail the five `\...mark` primitives, whose meaning has the form
`\...mark:<user material>`. The problem is that the `<user material>` can contain `->`.

However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first `:`, and analyse what is left. However, macros can have any combination of `\long`, `\protected` or `\outer` (not used in L^AT_EX3) before the string `macro:.` We thus only select the part of the meaning between the first `ma` and the first following `:`. If this string is `cro`, then we have a macro. If the string is `rk`, then we have a mark. The string can also be `cro parameter character` for a colon with a weird category code (namely the usual category code of `#`). Otherwise, it is empty.

This relies on the fact that `\long`, `\protected`, `\outer` cannot contain `ma`, regardless of the escape character, even if the escape character is `m...`

Both `ma` and `:` must be of category code 12 (other), and we achieve using the standard lowercasing technique.

```

2818 \group_begin:
2819 \char_set_catcode_other:N \M
2820 \char_set_catcode_other:N \A
2821 \char_set_lccode:nn { '\; } { '\: }
2822 \char_set_lccode:nn { '\T } { '\T }
2823 \char_set_lccode:nn { '\F } { '\F }
2824 \tl_to_lowercase:n
2825 {

```

```

2826 \group_end:
2827 \prg_new_conditional:Npnn \token_if_macro:N #1 { p , T , F , TF }
2828 {
2829   \exp_after:wN \__token_if_macro_p:w
2830   \token_to_meaning:N #1 MA; \q_stop
2831 }
2832 \cs_new:Npn \__token_if_macro_p:w #1 MA #2 ; #3 \q_stop
2833 {
2834   \if_int_compare:w \pdfTeX_strcmp:D { #2 } { cro } = \c_zero
2835     \prg_return_true:
2836   \else:
2837     \prg_return_false:
2838   \fi:
2839 }
2840 }

```

(End definition for `\token_if_macro:N`. These functions are documented on page 56.)

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. This follows the same pattern as
`\token_if_cs:NTF` for `\token_if_letter:N` etc. We use `\scan_stop:` for this.

```

2841 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
2842 {
2843   \if_catcode:w \exp_not:N #1 \scan_stop:
2844     \prg_return_true: \else: \prg_return_false: \fi:
2845 }

```

(End definition for `\token_if_cs:N`. These functions are documented on page 56.)

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that T_EX will temporarily convert `\exp_not:N`
`\token_if_expandable:N` `\token` into `\scan_stop:` if `\token` is expandable. An undefined token is not
 considered as expandable. No problem nesting the conditionals, since the third #1 is only
 skipped if it is non-expandable (hence not part of T_EX's conditional apparatus).

```

2846 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
2847 {
2848   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
2849   \prg_return_false:
2850 \else:
2851   \if_cs_exist:N #1
2852     \prg_return_true:
2853   \else:
2854     \prg_return_false:
2855   \fi:
2856 \fi:
2857 }

```

(End definition for `\token_if_expandable:N`. These functions are documented on page 56.)

`\token_if_chardef_p:N` Most of these functions have to check the meaning of the token in question so we need to
`\token_if_mathchardef_p:N` do some checkups on which characters are output by `\token_to_meaning:N`. As usual,
`\token_if_dim_register_p:N` these characters have catcode 12 so we must do some serious substitutions in the code
`\token_if_int_register_p:N` below...

```

\token_if_muskip_register_p:N
\token_if_skip_register_p:N
\token_if_toks_register_p:N
\token_if_long_macro_p:N
\token_if_protected_macro_p:N
\token_if_protected_long_macro_p:N
\token_if_chardef:N\TF
\token_if_mathchardef:N\TF
\token_if_dim_register:N\TF
\token_if_int_register:N\TF

```

```

2858 \group_begin:
2859 \char_set_lccode:nn { 'T } { 'T }
2860 \char_set_lccode:nn { 'F } { 'F }
2861 \char_set_lccode:nn { 'X } { 'n }
2862 \char_set_lccode:nn { 'Y } { 't }
2863 \char_set_lccode:nn { 'Z } { 'd }
2864 \tl_map_inline:nn { A C E G H I K L M O P R S U X Y Z R " }
2865   { \char_set_catcode:nn { '#1 } \c_twelve }

```

We convert the token list to lower case and restore the catcode and lowercase code changes.

```

2866 \tl_to_lowercase:n
2867 {
2868   \group_end:

```

First up is checking if something has been defined with `\chardef` or `\mathchardef`. This is easy since \TeX thinks of such tokens as hexadecimal so it stores them as `\char"<hex number>` or `\mathchar"<hex number>`. Grab until the first occurrence of `char"`, and compare what precedes with `\` or `\math`. In fact, the escape character may not be a backslash, so we compare with the result of converting some other control sequence to a string, namely `\char` or `\mathchar` (the auxiliary adds the `char` back).

```

2869 \prg_new_conditional:Npnn \token_if_chardef:N #1 { p , T , F , TF }
2870 {
2871   \__str_if_eq_x_return:nn
2872   {
2873     \exp_after:wN \__token_if_chardef:w
2874     \token_to_meaning:N #1 CHAR" \q_stop
2875   }
2876   { \token_to_str:N \char }
2877 }
2878 \prg_new_conditional:Npnn \token_if_mathchardef:N #1 { p , T , F , TF }
2879 {
2880   \__str_if_eq_x_return:nn
2881   {
2882     \exp_after:wN \__token_if_chardef:w
2883     \token_to_meaning:N #1 CHAR" \q_stop
2884   }
2885   { \token_to_str:N \mathchar }
2886 }
2887 \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`.

```

2888 \prg_new_conditional:Npnn \token_if_dim_register:N #1 { p , T , F , TF }
2889 {
2890   \if_meaning:w \tex_dimen:D #1
2891   \prg_return_false:
2892   \else:
2893     \if_meaning:w \tex_dimendef:D #1
2894     \prg_return_false:

```

```

2895         \else:
2896         \__str_if_eq_x_return:nn
2897         {
2898         \exp_after:wN \__token_if_dim_register:w
2899         \token_to_meaning:N #1 ZIMEX \q_stop
2900         }
2901         { \token_to_str:N \ }
2902         \fi:
2903         \fi:
2904     }
2905     \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`.

```

2906     \prg_new_conditional:Npnn \token_if_int_register:N #1 { p , T , F , TF }
2907     {
2908         % \token_if_chardef:NTF #1 { \prg_return_true: }
2909         % {
2910         %     \token_if_mathchardef:NTF #1 { \prg_return_true: }
2911         %     {
2912         \if_meaning:w \tex_count:D #1
2913         \prg_return_false:
2914         \else:
2915         \if_meaning:w \tex_countdef:D #1
2916         \prg_return_false:
2917         \else:
2918         \__str_if_eq_x_return:nn
2919         {
2920         \exp_after:wN \__token_if_int_register:w
2921         \token_to_meaning:N #1 COUXY \q_stop
2922         }
2923         { \token_to_str:N \ }
2924         \fi:
2925         \fi:
2926         %     }
2927         % }
2928     }
2929     \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.

```

2930     \prg_new_conditional:Npnn \token_if_muskip_register:N #1 { p , T , F , TF }
2931     {
2932         \if_meaning:w \tex_muskip:D #1
2933         \prg_return_false:
2934         \else:
2935         \if_meaning:w \tex_muskipdef:D #1
2936         \prg_return_false:
2937         \else:
2938         \__str_if_eq_x_return:nn
2939         {

```



```

2940         \exp_after:wN \__token_if_muskip_register:w
2941         \token_to_meaning:N #1 MUSKIP \q_stop
2942     }
2943     { \token_to_str:N \ }
2944     \fi:
2945     \fi:
2946 }
2947 \cs_new:Npn \__token_if_muskip_register:w #1 MUSKIP #2 \q_stop { #1 ~ }

```

Skip registers.

```

2948 \prg_new_conditional:Npnn \token_if_skip_register:N #1 { p , T , F , TF }
2949 {
2950     \if_meaning:w \tex_skip:D #1
2951     \prg_return_false:
2952 }else:
2953     \if_meaning:w \tex_skipdef:D #1
2954     \prg_return_false:
2955 }else:
2956     \__str_if_eq_x_return:nn
2957     {
2958         \exp_after:wN \__token_if_skip_register:w
2959         \token_to_meaning:N #1 SKIP \q_stop
2960     }
2961     { \token_to_str:N \ }
2962     \fi:
2963     \fi:
2964 }
2965 \cs_new:Npn \__token_if_skip_register:w #1 SKIP #2 \q_stop { #1 ~ }

```

Toks registers.

```

2966 \prg_new_conditional:Npnn \token_if_toks_register:N #1 { p , T , F , TF }
2967 {
2968     \if_meaning:w \tex_toks:D #1
2969     \prg_return_false:
2970 }else:
2971     \if_meaning:w \tex_toksdef:D #1
2972     \prg_return_false:
2973 }else:
2974     \__str_if_eq_x_return:nn
2975     {
2976         \exp_after:wN \__token_if_toks_register:w
2977         \token_to_meaning:N #1 YOKS \q_stop
2978     }
2979     { \token_to_str:N \ }
2980     \fi:
2981     \fi:
2982 }
2983 \cs_new:Npn \__token_if_toks_register:w #1 YOKS #2 \q_stop { #1 ~ }

```

Protected macros.

```

2984 \prg_new_conditional:Npnn \token_if_protected_macro:N #1

```

```

2985 { p , T , F , TF }
2986 {
2987   \__str_if_eq_x_return:nn
2988   {
2989     \exp_after:wN \__token_if_protected_macro:w
2990     \token_to_meaning:N #1 PROYECY EZ~MACRO \q_stop
2991   }
2992   { \token_to_str:N \ }
2993 }
2994 \cs_new:Npn \__token_if_protected_macro:w
2995 #1 PROYECY EZ~MACRO #2 \q_stop { #1 ~ }

```

Long macros and protected long macros share an auxiliary.

```

2996 \prg_new_conditional:Npnn \token_if_long_macro:N #1 { p , T , F , TF }
2997 {
2998   \__str_if_eq_x_return:nn
2999   {
3000     \exp_after:wN \__token_if_long_macro:w
3001     \token_to_meaning:N #1 LOXG~MACRO \q_stop
3002   }
3003   { \token_to_str:N \ }
3004 }
3005 \prg_new_conditional:Npnn \token_if_protected_long_macro:N #1
3006 { p , T , F , TF }
3007 {
3008   \__str_if_eq_x_return:nn
3009   {
3010     \exp_after:wN \__token_if_long_macro:w
3011     \token_to_meaning:N #1 LOXG~MACRO \q_stop
3012   }
3013   { \token_to_str:N \protected \token_to_str:N \ }
3014 }
3015 \cs_new:Npn \__token_if_long_macro:w #1 LOXG~MACRO #2 \q_stop { #1 ~ }

```

Finally the `\tl_to_lowercase:n` ends!

```

3016 }

```

(End definition for `\token_if_chardef:N` and others. These functions are documented on page 56.)

```

\token_if_primitive_p:N
\token_if_primitive:N 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.

```

3017 \tex_chardef:D \c_token_A_int = 'A ~ %
3018 \group_begin:
3019 \char_set_catcode_other:N \;
3020 \char_set_lccode:nn { '\; } { '\: }
3021 \char_set_lccode:nn { '\T } { '\T }
3022 \char_set_lccode:nn { '\F } { '\F }
3023 \tl_to_lowercase:n {
3024   \group_end:
3025   \prg_new_conditional:Npnn \token_if_primitive:N #1 { p , T , F , TF }
3026   {
3027     \token_if_macro:NTF #1
3028     \prg_return_false:
3029     {
3030       \exp_after:wN \__token_if_primitive:NNw
3031       \token_to_meaning:N #1 ; ; ; \q_stop #1
3032     }
3033   }
3034   \cs_new:Npn \__token_if_primitive:NNw #1#2 #3 ; #4 \q_stop
3035   {
3036     \tl_if_empty:oTF { \__token_if_primitive_space:w #3 ~ }
3037     { \__token_if_primitive_loop:N #3 ; \q_stop }
3038     { \__token_if_primitive_nullfont:N }
3039   }
3040 }
3041 \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
3042 \cs_new:Npn \__token_if_primitive_nullfont:N #1
3043 {
3044   \if_meaning:w \tex_nullfont:D #1
3045   \prg_return_true:
3046   \else:
3047     \prg_return_false:
3048   \fi:
3049 }
3050 \cs_new:Npn \__token_if_primitive_loop:N #1
3051 {
3052   \if_int_compare:w '#1 < \c_token_A_int %

```

```

3053     \exp_after:wN \_token_if_primitive:Nw
3054     \exp_after:wN #1
3055   \else:
3056     \exp_after:wN \_token_if_primitive_loop:N
3057   \fi:
3058 }
3059 \cs_new:Npn \_token_if_primitive:Nw #1 #2 \q_stop
3060 {
3061   \if:w : #1
3062     \exp_after:wN \_token_if_primitive_undefined:N
3063   \else:
3064     \prg_return_false:
3065     \exp_after:wN \use_none:n
3066   \fi:
3067 }
3068 \cs_new:Npn \_token_if_primitive_undefined:N #1
3069 {
3070   \if_cs_exist:N #1
3071     \prg_return_true:
3072   \else:
3073     \prg_return_false:
3074   \fi:
3075 }

```

(End definition for `\token_if_primitive:N`. These functions are documented on page 57.)

7.4 Peeking ahead at the next token

```

3076 <@@=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` 3077 \cs_new_eq:NN \l_peek_token ?

3078 \cs_new_eq:NN \g_peek_token ?

(End definition for `\l_peek_token`. This function is documented on page 58.)

`\l__peek_search_token` The token to search for as an implicit token: cf. `\l__peek_search_tl`.

3079 \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`.

```

3080 \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 3081 \cs_new_nopar:Npn \__peek_true:w { }
\__peek_false:w    3082 \cs_new_nopar:Npn \__peek_true_aux:w { }
\__peek_tmp:w      3083 \cs_new_nopar:Npn \__peek_false:w { }
                   3084 \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`

```

3085 \cs_new_protected_nopar:Npn \peek_after:Nw
3086 { \tex_futurelet:D \l__peek_token }
3087 \cs_new_protected_nopar:Npn \peek_gafter:Nw
3088 { \tex_global:D \tex_futurelet:D \g__peek_token }
(End definition for \peek_after:Nw. This function is documented on page 58.)

```

`__peek_true_remove:w` A function to remove the next token and then regain control.

```

3089 \cs_new_protected:Npn \__peek_true_remove:w
3090 {
3091   \group_align_safe_end:
3092   \tex_afterassignment:D \__peek_true_aux:w
3093   \cs_set_eq:NN \__peek_tmp:w
3094 }
(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.

```

3095 \cs_new_protected:Npn \__peek_token_generic:NNTF #1#2#3#4
3096 {
3097   \cs_set_eq:NN \l__peek_search_token #2
3098   \tl_set:Nn \l__peek_search_tl {#2}
3099   \cs_set_nopar:Npx \__peek_true:w
3100   {
3101     \exp_not:N \group_align_safe_end:
3102     \exp_not:n {#3}
3103   }
3104   \cs_set_nopar:Npx \__peek_false:w
3105   {
3106     \exp_not:N \group_align_safe_end:
3107     \exp_not:n {#4}
3108   }
3109   \group_align_safe_begin:
3110   \peek_after:Nw #1
3111 }
3112 \cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3

```

```

3113 { \__peek_token_generic:NNTF #1 #2 {#3} { } }
3114 \cs_new_protected:Npn \__peek_token_generic:NNTF #1#2#3
3115 { \__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.

```

3116 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF #1#2#3#4
3117 {
3118   \cs_set_eq:NN \l__peek_search_token #2
3119   \tl_set:Nn \l__peek_search_tl {#2}
3120   \cs_set_eq:NN \__peek_true:w \__peek_true_remove:w
3121   \cs_set_nopar:Npx \__peek_true_aux:w { \exp_not:n {#3} }
3122   \cs_set_nopar:Npx \__peek_false:w
3123   {
3124     \exp_not:N \group_align_safe_end:
3125     \exp_not:n {#4}
3126   }
3127   \group_align_safe_begin:
3128   \peek_after:Nw #1
3129 }
3130 \cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
3131 { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
3132 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF #1#2#3
3133 { \__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.

```

3134 \cs_new_nopar:Npn \__peek_execute_branches_meaning:
3135 {
3136   \if_meaning:w \l_peek_token \l__peek_search_token
3137   \exp_after:wN \__peek_true:w
3138   \else:
3139     \exp_after:wN \__peek_false:w
3140   \fi:
3141 }

```

(End definition for __peek_execute_branches_meaning:. This function is documented on page ??.)

__peek_execute_branches_catcode:

__peek_execute_branches_charcode:

__peek_execute_branches_catcode_aux:

__peek_execute_branches_catcode_auxii:N

__peek_execute_branches_catcode_auxiii:

The catcode and charcode tests are very similar, and in order to use the same auxiliaries we do something a little bit odd, firing \if_catcode:w and \if_charcode:w before finding the operands for those tests, which will only be given in the auxii:N and auxiii: auxiliaries. For our purposes, three kinds of tokens may follow the peeking function:

- control sequences which are not equal to a non-active character token (e.g., macro, primitive);
- active characters which are not equal to a non-active character token (e.g., macro, primitive);
- explicit non-active character tokens, or control sequences or active characters set equal to a non-active character token.

The first two cases are not distinguishable simply using \TeX 's `\futurelet`, because we can only access the `\meaning` of tokens in that way. In those cases, detected thanks to a comparison with `\scan_stop:`, we grab the following token, and compare it explicitly with the explicit search token stored in `\l__peek_search_tl`. The `\exp_not:N` prevents outer macros (coming from non- \LaTeX 3 code) from blowing up. In the third case, `\l_peek_token` is good enough for the test, and we compare it again with the explicit search token. Just like the peek token, the search token may be of any of the three types above, hence the need to use the explicit token that was given to the peek function.

```

3142 \cs_new_nopar:Npn \__peek_execute_branches_catcode:
3143 { \if_catcode:w \__peek_execute_branches_catcode_aux: }
3144 \cs_new_nopar:Npn \__peek_execute_branches_charcode:
3145 { \if_charcode:w \__peek_execute_branches_catcode_aux: }
3146 \cs_new_nopar:Npn \__peek_execute_branches_catcode_aux:
3147 {
3148     \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
3149     \exp_after:wN \exp_after:wN
3150     \exp_after:wN \__peek_execute_branches_catcode_auxii:N
3151     \exp_after:wN \exp_not:N
3152     \else:
3153     \exp_after:wN \__peek_execute_branches_catcode_auxiii:
3154     \fi:
3155 }
3156 \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
3157 {
3158     \exp_not:N #1
3159     \exp_after:wN \exp_not:N \l__peek_search_tl
3160     \exp_after:wN \__peek_true:w
3161     \else:
3162     \exp_after:wN \__peek_false:w
3163     \fi:
3164     #1
3165 }
3166 \cs_new_nopar:Npn \__peek_execute_branches_catcode_auxiii:
3167 {
3168     \exp_not:N \l_peek_token
3169     \exp_after:wN \exp_not:N \l__peek_search_tl
3170     \exp_after:wN \__peek_true:w
3171     \else:
3172     \exp_after:wN \__peek_false:w
3173     \fi:
3174 }

```

(End definition for `__peek_execute_branches_catcode:` and `__peek_execute_branches_charcode:`. These functions are documented on page ??.)

`__peek_ignore_spaces_execute_branches:` This function removes one space token at a time, and calls `__peek_execute_branches:` when encountering the first non-space token. We directly use the primitive meaning test rather than `\token_if_eq_meaning:NNTF` because `\l_peek_token` may be an outer macro (coming from non- \LaTeX 3 packages). Spaces are removed using a side-effect of f-expansion: `\tex_romannumeral:D -'0` removes one space.

```

3175 \cs_new_protected_nopar:Npn \__peek_ignore_spaces_execute_branches:
3176 {
3177   \if_meaning:w \l_peek_token \c_space_token
3178     \exp_after:wN \peek_after:Nw
3179     \exp_after:wN \__peek_ignore_spaces_execute_branches:
3180     \tex_romannumeral:D -‘0
3181   \else:
3182     \exp_after:wN \__peek_execute_branches:
3183   \fi:
3184 }

```

(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.

```

3185 \group_begin:
3186 \cs_set:Npn \__peek_def:nnnn #1#2#3#4
3187 {
3188   \__peek_def:nnnnn {#1} {#2} {#3} {#4} { TF }
3189   \__peek_def:nnnnn {#1} {#2} {#3} {#4} { T }
3190   \__peek_def:nnnnn {#1} {#2} {#3} {#4} { F }
3191 }
3192 \cs_set:Npn \__peek_def:nnnnn #1#2#3#4#5
3193 {
3194   \cs_new_protected_nopar:cpx { #1 #5 }
3195   {
3196     \tl_if_empty:nF {#2}
3197       { \exp_not:n { \cs_set_eq:NN \__peek_execute_branches: #2 } }
3198     \exp_not:c { #3 #5 }
3199     \exp_not:n {#4}
3200   }
3201 }

```

(End definition for __peek_def:nnnn. This function is documented on page ??.)

\peek_catcode:N~~TF~~ With everything in place the definitions can take place. First for category codes.
\peek_catcode_ignore_spaces:N~~TF~~
\peek_catcode_remove:N~~TF~~
\peek_catcode_remove_ignore_spaces:N~~TF~~

```

3202 \__peek_def:nnnn { peek_catcode:N }
3203 { }
3204 { __peek_token_generic:NN }
3205 { \__peek_execute_branches_catcode: }
3206 \__peek_def:nnnn { peek_catcode_ignore_spaces:N }
3207 { \__peek_execute_branches_catcode: }
3208 { __peek_token_generic:NN }
3209 { \__peek_ignore_spaces_execute_branches: }
3210 \__peek_def:nnnn { peek_catcode_remove:N }
3211 { }
3212 { __peek_token_remove_generic:NN }
3213 { \__peek_execute_branches_catcode: }
3214 \__peek_def:nnnn { peek_catcode_remove_ignore_spaces:N }

```



```

3215 { \__peek_execute_branches_catcode: }
3216 { __peek_token_remove_generic:NN }
3217 { \__peek_ignore_spaces_execute_branches: }

```

(End definition for \peek_catcode:NTF and others. These functions are documented on page 59.)

\peek_charcode:NTF
\peek_charcode_ignore_spaces:NTF
\peek_charcode_remove:NTF
\peek_charcode_remove_ignore_spaces:NTF

Then for character codes.

```

3218 \__peek_def:nnnn { peek_charcode:N }
3219 { }
3220 { __peek_token_generic:NN }
3221 { \__peek_execute_branches_charcode: }
3222 \__peek_def:nnnn { peek_charcode_ignore_spaces:N }
3223 { \__peek_execute_branches_charcode: }
3224 { __peek_token_generic:NN }
3225 { \__peek_ignore_spaces_execute_branches: }
3226 \__peek_def:nnnn { peek_charcode_remove:N }
3227 { }
3228 { __peek_token_remove_generic:NN }
3229 { \__peek_execute_branches_charcode: }
3230 \__peek_def:nnnn { peek_charcode_remove_ignore_spaces:N }
3231 { \__peek_execute_branches_charcode: }
3232 { __peek_token_remove_generic:NN }
3233 { \__peek_ignore_spaces_execute_branches: }

```

(End definition for \peek_charcode:NTF and others. These functions are documented on page 60.)

\peek_meaning:NTF
\peek_meaning_ignore_spaces:NTF
\peek_meaning_remove:NTF
\peek_meaning_remove_ignore_spaces:NTF

Finally for meaning, with the group closed to remove the temporary definition functions.

```

3234 \__peek_def:nnnn { peek_meaning:N }
3235 { }
3236 { __peek_token_generic:NN }
3237 { \__peek_execute_branches_meaning: }
3238 \__peek_def:nnnn { peek_meaning_ignore_spaces:N }
3239 { \__peek_execute_branches_meaning: }
3240 { __peek_token_generic:NN }
3241 { \__peek_ignore_spaces_execute_branches: }
3242 \__peek_def:nnnn { peek_meaning_remove:N }
3243 { }
3244 { __peek_token_remove_generic:NN }
3245 { \__peek_execute_branches_meaning: }
3246 \__peek_def:nnnn { peek_meaning_remove_ignore_spaces:N }
3247 { \__peek_execute_branches_meaning: }
3248 { __peek_token_remove_generic:NN }
3249 { \__peek_ignore_spaces_execute_branches: }
3250 \group_end:

```

(End definition for \peek_meaning:NTF and others. These functions are documented on page 60.)

7.5 Decomposing a macro definition

\token_get_prefix_spec:N
\token_get_arg_spec:N
\token_get_replacement_spec:N
__peek_get_prefix_arg_replacement:wN

We sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the

argument specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token `\scan_stop:` is returned instead.

```

3251 \exp_args:Nno \use:nn
3252 { \cs_new:Npn \__peek_get_prefix_arg_replacement:wN #1 }
3253 { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
3254 { #4 {#1} {#2} {#3} }
3255 \cs_new:Npn \token_get_prefix_spec:N #1
3256 {
3257   \token_if_macro:NTF #1
3258   {
3259     \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3260     \token_to_meaning:N #1 \q_stop \use_i:nnn
3261   }
3262   { \scan_stop: }
3263 }
3264 \cs_new:Npn \token_get_arg_spec:N #1
3265 {
3266   \token_if_macro:NTF #1
3267   {
3268     \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3269     \token_to_meaning:N #1 \q_stop \use_ii:nnn
3270   }
3271   { \scan_stop: }
3272 }
3273 \cs_new:Npn \token_get_replacement_spec:N #1
3274 {
3275   \token_if_macro:NTF #1
3276   {
3277     \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3278     \token_to_meaning:N #1 \q_stop \use_iii:nnn
3279   }
3280   { \scan_stop: }
3281 }

```

(End definition for `\token_get_prefix_spec:N`. This function is documented on page 61.)

7.6 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

```

\char_set_catcode:w Primitives renamed.
\char_set_mathcode:w
\char_set_lccode:w 3282 <*/deprecated>
\char_set_uccode:w 3283 \cs_new_eq:NN \char_set_catcode:w \tex_catcode:D
\char_set_sfcode:w 3284 \cs_new_eq:NN \char_set_mathcode:w \tex_mathcode:D
3285 \cs_new_eq:NN \char_set_lccode:w \tex_lccode:D
3286 \cs_new_eq:NN \char_set_uccode:w \tex_uccode:D
3287 \cs_new_eq:NN \char_set_sfcode:w \tex_sfcode:D
3288 </deprecated>

```

(End definition for `\char_set_catcode:w`. This function is documented on page ??.)

```

\char_value_catcode:w More w functions we should not have.
\char_show_value_catcode:w 3289 <*deprecated>
\char_value_mathcode:w 3290 \cs_new_nopar:Npn \char_value_catcode:w { \tex_the:D \char_set_catcode:w }
\char_show_value_mathcode:w 3291 \cs_new_nopar:Npn \char_show_value_catcode:w
\char_value_lccode:w 3292 { \tex_showthe:D \char_set_catcode:w }
\char_show_value_lccode:w 3293 \cs_new_nopar:Npn \char_value_mathcode:w { \tex_the:D \char_set_mathcode:w }
\char_value_uccode:w 3294 \cs_new_nopar:Npn \char_show_value_mathcode:w
\char_show_value_uccode:w 3295 { \tex_showthe:D \char_set_mathcode:w }
\char_value_sfcode:w 3296 \cs_new_nopar:Npn \char_value_lccode:w { \tex_the:D \char_set_lccode:w }
\char_show_value_sfcode:w 3297 \cs_new_nopar:Npn \char_show_value_lccode:w
3298 { \tex_showthe:D \char_set_lccode:w }
3299 \cs_new_nopar:Npn \char_value_uccode:w { \tex_the:D \char_set_uccode:w }
3300 \cs_new_nopar:Npn \char_show_value_uccode:w
3301 { \tex_showthe:D \char_set_uccode:w }
3302 \cs_new_nopar:Npn \char_value_sfcode:w { \tex_the:D \char_set_sfcode:w }
3303 \cs_new_nopar:Npn \char_show_value_sfcode:w
3304 { \tex_showthe:D \char_set_sfcode:w }
3305 </deprecated>
(End definition for \char_value_catcode:w. This function is documented on page ??.)

```

```

\peek_after:NN The second argument here must be w.
\peek_gafter:NN 3306 <*deprecated>
3307 \cs_new_eq:NN \peek_after:NN \peek_after:Nw
3308 \cs_new_eq:NN \peek_gafter:NN \peek_gafter:Nw
3309 </deprecated>
(End definition for \peek_after:NN. This function is documented on page ??.)
Functions deprecated 2011-05-28 for removal by 2011-08-31.

```

```

\c_alignment_tab_token
\c_math_shift_token 3310 <*deprecated>
\c_letter_token 3311 \cs_new_eq:NN \c_alignment_tab_token \c_alignment_token
\c_other_char_token 3312 \cs_new_eq:NN \c_math_shift_token \c_math_toggle_token
3313 \cs_new_eq:NN \c_letter_token \c_catcode_letter_token
3314 \cs_new_eq:NN \c_other_char_token \c_catcode_other_token
3315 </deprecated>
(End definition for \c_alignment_tab_token. This function is documented on page ??.)

```

```

\c_active_char_token An odd one: this was never a token!
3316 <*deprecated>
3317 \cs_new_eq:NN \c_active_char_token \c_catcode_active_tl
3318 </deprecated>
(End definition for \c_active_char_token. This function is documented on page ??.)

```

```

\char_make_escape:N Two renames in one block!
\char_make_group_begin:N 3319 <*deprecated>
\char_make_group_end:N 3320 \cs_new_eq:NN \char_make_escape:N \char_set_catcode_escape:N
\char_make_math_toggle:N 3321 \cs_new_eq:NN \char_make_begin_group:N \char_set_catcode_group_begin:N
\char_make_alignment:N 3322 \cs_new_eq:NN \char_make_end_group:N \char_set_catcode_group_end:N
\char_make_end_line:N
\char_make_parameter:N
\char_make_math_superscript:N
\char_make_math_subscript:N
\char_make_ignore:N
\char_make_space:N
\char_make_letter:N
\char_make_other:N
\char_make_active:N
\char_make_comment:N

```

```

3323 \cs_new_eq:NN \char_make_math_shift:N \char_set_catcode_math_toggle:N
3324 \cs_new_eq:NN \char_make_alignment_tab:N \char_set_catcode_alignment:N
3325 \cs_new_eq:NN \char_make_end_line:N \char_set_catcode_end_line:N
3326 \cs_new_eq:NN \char_make_parameter:N \char_set_catcode_parameter:N
3327 \cs_new_eq:NN \char_make_math_superscript:N
3328 \char_set_catcode_math_superscript:N
3329 \cs_new_eq:NN \char_make_math_subscript:N
3330 \char_set_catcode_math_subscript:N
3331 \cs_new_eq:NN \char_make_ignore:N \char_set_catcode_ignore:N
3332 \cs_new_eq:NN \char_make_space:N \char_set_catcode_space:N
3333 \cs_new_eq:NN \char_make_letter:N \char_set_catcode_letter:N
3334 \cs_new_eq:NN \char_make_other:N \char_set_catcode_other:N
3335 \cs_new_eq:NN \char_make_active:N \char_set_catcode_active:N
3336 \cs_new_eq:NN \char_make_comment:N \char_set_catcode_comment:N
3337 \cs_new_eq:NN \char_make_invalid:N \char_set_catcode_invalid:N
3338 \cs_new_eq:NN \char_make_escape:n \char_set_catcode_escape:n
3339 \cs_new_eq:NN \char_make_begin_group:n \char_set_catcode_group_begin:n
3340 \cs_new_eq:NN \char_make_end_group:n \char_set_catcode_group_end:n
3341 \cs_new_eq:NN \char_make_math_shift:n \char_set_catcode_math_toggle:n
3342 \cs_new_eq:NN \char_make_alignment_tab:n \char_set_catcode_alignment:n
3343 \cs_new_eq:NN \char_make_end_line:n \char_set_catcode_end_line:n
3344 \cs_new_eq:NN \char_make_parameter:n \char_set_catcode_parameter:n
3345 \cs_new_eq:NN \char_make_math_superscript:n
3346 \char_set_catcode_math_superscript:n
3347 \cs_new_eq:NN \char_make_math_subscript:n
3348 \char_set_catcode_math_subscript:n
3349 \cs_new_eq:NN \char_make_ignore:n \char_set_catcode_ignore:n
3350 \cs_new_eq:NN \char_make_space:n \char_set_catcode_space:n
3351 \cs_new_eq:NN \char_make_letter:n \char_set_catcode_letter:n
3352 \cs_new_eq:NN \char_make_other:n \char_set_catcode_other:n
3353 \cs_new_eq:NN \char_make_active:n \char_set_catcode_active:n
3354 \cs_new_eq:NN \char_make_comment:n \char_set_catcode_comment:n
3355 \cs_new_eq:NN \char_make_invalid:n \char_set_catcode_invalid:n
3356 </deprecated>

```

(End definition for \char_make_escape:N and others. These functions are documented on page ??.)

```

\token_if_alignment_tab_p:N
\token_if_alignment_tab:NTF
\token_if_math_shift_p:N
\token_if_math_shift:NTF
\token_if_other_char_p:N
\token_if_other_char:NTF
\token_if_active_char_p:N
\token_if_active_char:NTF

```

```

3357 <*deprecated>
3358 \cs_new_eq:NN \token_if_alignment_tab_p:N \token_if_alignment_p:N
3359 \cs_new_eq:NN \token_if_alignment_tab:NT \token_if_alignment:NT
3360 \cs_new_eq:NN \token_if_alignment_tab:NF \token_if_alignment:NF
3361 \cs_new_eq:NN \token_if_alignment_tab:NTF \token_if_alignment:NTF
3362 \cs_new_eq:NN \token_if_math_shift_p:N \token_if_math_toggle_p:N
3363 \cs_new_eq:NN \token_if_math_shift:NT \token_if_math_toggle:NT
3364 \cs_new_eq:NN \token_if_math_shift:NF \token_if_math_toggle:NF
3365 \cs_new_eq:NN \token_if_math_shift:NTF \token_if_math_toggle:NTF
3366 \cs_new_eq:NN \token_if_other_char_p:N \token_if_other_p:N
3367 \cs_new_eq:NN \token_if_other_char:NT \token_if_other:NT
3368 \cs_new_eq:NN \token_if_other_char:NF \token_if_other:NF
3369 \cs_new_eq:NN \token_if_other_char:NTF \token_if_other:NTF

```

```

3370 \cs_new_eq:NN \token_if_active_char_p:N \token_if_active_p:N
3371 \cs_new_eq:NN \token_if_active_char:NT \token_if_active:NT
3372 \cs_new_eq:NN \token_if_active_char:NF \token_if_active:NF
3373 \cs_new_eq:NN \token_if_active_char:NTF \token_if_active:NTF
3374 \deprecated
(End definition for \token_if_alignment_tab:N. These functions are documented on page ??.)
3375 \initex | package)

```

8 l3int implementation

```

3376 \*initex | package)
3377 \@@=int)

The following test files are used for this code: m3int001,m3int002,m3int03.
3378 \*package)
3379 \ProvidesExplPackage
3380 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
3381 \__expl_package_check:
3382 \package)

```

__int_to_roman:w Done in l3basics.
if_int_compare:w (End definition for __int_to_roman:w. This function is documented on page 74.)

__int_value:w Here are the remaining primitives for number comparisons and expressions.
__int_eval:w 3383 \cs_new_eq:NN __int_value:w \tex_number:D
__int_eval_end: 3384 \cs_new_eq:NN __int_eval:w \etex_numexpr:D
if_int_odd:w 3385 \cs_new_eq:NN __int_eval_end: \tex_relax:D
if_case:w 3386 \cs_new_eq:NN if_int_odd:w \tex_ifodd:D
3387 \cs_new_eq:NN if_case:w \tex_ifcase:D
(End definition for __int_value:w. This function is documented on page 74.)

8.1 Integer expressions

int_eval:n Wrapper for __int_eval:w. Can be used in an integer expression or directly in the input stream. In format mode, there is already a definition in l3alloc for bootstrapping, which is therefore corrected to the “real” version here.

```

3388 \*initex)
3389 \cs_set:Npn int_eval:n #1 { __int_value:w __int_eval:w #1 __int_eval_end: }
3390 \initex)
3391 \*package)
3392 \cs_new:Npn int_eval:n #1 { __int_value:w __int_eval:w #1 __int_eval_end: }
3393 \package)

(End definition for int_eval:n. This function is documented on page 62.)

```

`\int_abs:n` Functions for min, max, and absolute value with only one evaluation. The absolute value is obtained by removing a leading sign if any. All three functions expand in two steps.

`__int_abs:N`

`\int_max:nn`

`\int_min:nn`

`__int_maxmin:wwN`

```

3394 \cs_new:Npn \int_abs:n #1
3395 {
3396   \__int_value:w \exp_after:wN \__int_abs:N
3397   \int_use:N \__int_eval:w #1 \__int_eval_end:
3398   \exp_stop_f:
3399 }
3400 \cs_new:Npn \__int_abs:N #1
3401 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
3402 \cs_set:Npn \int_max:nn #1#2
3403 {
3404   \__int_value:w \exp_after:wN \__int_maxmin:wwN
3405   \int_use:N \__int_eval:w #1 \exp_after:wN ;
3406   \int_use:N \__int_eval:w #2 ;
3407   >
3408   \exp_stop_f:
3409 }
3410 \cs_set:Npn \int_min:nn #1#2
3411 {
3412   \__int_value:w \exp_after:wN \__int_maxmin:wwN
3413   \int_use:N \__int_eval:w #1 \exp_after:wN ;
3414   \int_use:N \__int_eval:w #2 ;
3415   <
3416   \exp_stop_f:
3417 }
3418 \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
3419 {
3420   \if_int_compare:w #1 #3 #2 ~
3421   #1
3422   \else:
3423   #2
3424   \fi:
3425 }

```

(End definition for `\int_abs:n`. This function is documented on page 63.)

`\int_div_truncate:nn` As `__int_eval:w` rounds the result of a division we also provide a version that truncates the result. We use an auxiliary to make sure numerator and denominator are only evaluated once: this comes in handy when those are more expressions are expensive to evaluate (e.g., `\tl_count:n`). If the numerator `#1#2` is 0, then we divide 0 by the denominator (this ensures that 0/0 is correctly reported as an error). Otherwise, shift the numerator `#1#2` towards 0 by $(| \#3\#4 | - 1)/2$, which we round away from zero. It turns out that this quantity exactly compensates the difference between ϵ -TeX's rounding and the truncating behaviour that we want. The details are thanks to Heiko Oberdiek: getting things right in all cases is not so easy.

`\int_div_round:nn`

`\int_mod:nn`

`__int_div_truncate:NwNw`

`__int_mod:ww`

```

3426 \cs_new:Npn \int_div_truncate:nn #1#2
3427 {
3428   \int_use:N \__int_eval:w

```

```

3429     \exp_after:wN \__int_div_truncate:NwNw
3430     \int_use:N \__int_eval:w #1 \exp_after:wN ;
3431     \int_use:N \__int_eval:w #2 ;
3432     \__int_eval_end:
3433   }
3434   \cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
3435   {
3436     \if_meaning:w 0 #1
3437       \c_zero
3438     \else:
3439       (
3440         #1#2
3441         \if_meaning:w - #1 + \else: - \fi:
3442         ( \if_meaning:w - #3 - \fi: #3#4 - \c_one ) / \c_two
3443       )
3444     \fi:
3445     / #3#4
3446   }

```

For the sake of completeness:

```

3447   \cs_new:Npn \int_div_round:nn #1#2
3448   { \__int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }

```

Finally there's the modulus operation.

```

3449   \cs_new:Npn \int_mod:nn #1#2
3450   {
3451     \__int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
3452     \__int_value:w \__int_eval:w #1 \exp_after:wN ;
3453     \__int_value:w \__int_eval:w #2 ;
3454     \__int_eval_end:
3455   }
3456   \cs_new:Npn \__int_mod:ww #1; #2;
3457   { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }

```

(End definition for `\int_div_truncate:nn`. This function is documented on page 63.)

8.2 Creating and initialising integers

`\int_new:N` Two ways to do this: one for the format and one for the L^AT_EX 2_ε package.

```

\int_new:c 3458 <*package>
3459   \cs_new_protected:Npn \int_new:N #1
3460   {
3461     \__chk_if_free_cs:N #1
3462     \newcount #1
3463   }
3464   </package>
3465   \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_constdef:Nw
\c__max_constdef_int

```

```

3466 \cs_new_protected:Npn \int_const:Nn #1#2
3467 {
3468   \int_compare:nNnTF {#2} > \c_minus_one
3469   {
3470     \int_compare:nNnTF {#2} > \c__max_constdef_int
3471     {
3472       \int_new:N #1
3473       \int_gset:Nn #1 {#2}
3474     }
3475     {
3476       \__chk_if_free_cs:N #1
3477       \tex_global:D \__int_constdef:Nw #1 =
3478       \__int_eval:w #2 \__int_eval_end:
3479     }
3480   }
3481   {
3482     \int_new:N #1
3483     \int_gset:Nn #1 {#2}
3484   }
3485 }
3486 \cs_generate_variant:Nn \int_const:Nn { c }
3487 \pdfTeX_if_engine:TF
3488 {
3489   \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
3490   \tex_mathchardef:D \c__max_constdef_int 32 767 ~
3491 }
3492 {
3493   \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D
3494   \tex_chardef:D \c__max_constdef_int 1 114 111 ~
3495 }

```

(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 3496 \cs_new_protected:Npn \int_zero:N #1 { #1 = \c_zero }
\int_gzero:N 3497 \cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero }
\int_gzero:c 3498 \cs_generate_variant:Nn \int_zero:N { c }
3499 \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 3500 \cs_new_protected:Npn \int_zero_new:N #1
\int_gzero_new:N 3501 { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
\int_gzero_new:c 3502 \cs_new_protected:Npn \int_gzero_new:N #1
3503 { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
3504 \cs_generate_variant:Nn \int_zero_new:N { c }
3505 \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 3506 \cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
\int_set_eq:Nc 3507 \cs_generate_variant:Nn \int_set_eq:NN { c }
\int_set_eq:cc 3508 \cs_generate_variant:Nn \int_set_eq:NN { Nc , cc }
\int_gset_eq:NN 3509 \cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\int_gset_eq:cN 3510 \cs_generate_variant:Nn \int_gset_eq:NN { c }
\int_gset_eq:Nc 3511 \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 3512 \prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N { TF , T , F , p }
\int_if_exist:NTF 3513 \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 3514 \cs_new_protected:Npn \int_add:Nn #1#2
\int_gadd:Nn 3515 { \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
\int_gadd:cn 3516 \cs_new_protected:Npn \int_sub:Nn #1#2
\int_sub:Nn 3517 { \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
\int_sub:cn 3518 \cs_new_protected_nopar:Npn \int_gadd:Nn
\int_gsub:Nn 3519 { \tex_global:D \int_add:Nn }
\int_gsub:cn 3520 \cs_new_protected_nopar:Npn \int_gsub:Nn
3521 { \tex_global:D \int_sub:Nn }
3522 \cs_generate_variant:Nn \int_add:Nn { c }
3523 \cs_generate_variant:Nn \int_gadd:Nn { c }
3524 \cs_generate_variant:Nn \int_sub:Nn { c }
3525 \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 3526 \cs_new_protected:Npn \int_incr:N #1
\int_gincr:N 3527 { \tex_advance:D #1 \c_one }
\int_gincr:c 3528 \cs_new_protected:Npn \int_decr:N #1
\int_decr:N 3529 { \tex_advance:D #1 \c_minus_one }
\int_decr:c 3530 \cs_new_protected_nopar:Npn \int_gincr:N
\int_gdecr:N 3531 { \tex_global:D \int_incr:N }
\int_gdecr:c 3532 \cs_new_protected_nopar:Npn \int_gdecr:N
3533 { \tex_global:D \int_decr:N }
3534 \cs_generate_variant:Nn \int_incr:N { c }
3535 \cs_generate_variant:Nn \int_decr:N { c }
3536 \cs_generate_variant:Nn \int_gincr:N { c }
3537 \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
3538 \cs_new_protected:Npn \int_set:Nn #1#2
3539 { #1 ~ \__int_eval:w #2\__int_eval_end: }
3540 \cs_new_protected_nopar:Npn \int_gset:Nn { \tex_global:D \int_set:Nn }
3541 \cs_generate_variant:Nn \int_set:Nn { c }
3542 \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
3543 \cs_new_eq:NN \int_use:N \tex_the:D
3544 \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. `__prg_compare_error:NNw` 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.

```
3545 \cs_new_protected_nopar:Npn \__prg_compare_error:
3546 {
3547   \if_int_compare:w \c_zero \c_zero \fi:
3548   =
3549   \__prg_compare_error:
3550 }
3551 \cs_new:Npn \__prg_compare_error:Nw
3552 #1#2 \q_stop
3553 {
3554   { }
3555   \c_zero \fi:
3556   \msg_kernel_expandable_error:nnn
3557     { kernel } { unknown-comparison } {#1}
3558   \prg_return_false:
3559 }
```

(End definition for `__prg_compare_error:` and `__prg_compare_error:NNw`.)

`\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:NNw` 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.

```

3560 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
3561 {
3562     \exp_after:wN \__int_compare:w
3563     \int_use:N \__int_eval:w #1 \__prg_compare_error:
3564 }
3565 \cs_new:Npn \__int_compare:w #1 \__prg_compare_error:
3566 {
3567     \exp_after:wN \if_false: \__int_value:w
3568     \__int_compare:Nw #1 e { = nd_ } \q_stop
3569 }

```

The goal here is to find an *<operand>* and a *<comparison>*. The *<operand>* is already evaluated, but we cannot yet grab it as an argument. To access the following relation symbol, we remove the number by applying `__int_to_roman:w`, after making sure that the argument becomes non-positive: its roman numeral representation is then empty. Then probe the first two tokens with `__int_compare:NNw` to determine the relation symbol, building a control sequence from it. All the extended forms have an extra `=` hence the test for that as a second token. If the relation symbol is unknown, then the control sequence is turned by \TeX into `\scan_stop:`, ignored thanks to `\unexpanded`, and `__prg_compare_error:Nw` raises an error.

```

3570 \cs_new:Npn \__int_compare:Nw #1#2 \q_stop
3571 {
3572     \exp_after:wN \__int_compare:NNw
3573     \__int_to_roman:w - 0 #2 \q_mark
3574     #1#2 \q_stop
3575 }
3576 \cs_new:Npn \__int_compare:NNw #1#2#3 \q_mark
3577 {
3578     \etex_unexpanded:D
3579     \use:c { __int_compare_ #1 \if_meaning:w = #2 = \fi: :NNw }

```

```

3580     \__prg_compare_error:Nw #1
3581 }

```

When the last $\langle operand \rangle$ is seen, $__int_compare:NNw$ receives e and $=nd_$ as arguments, hence calling $__int_compare_end_=:NNw$ to end the loop: return the result of the last comparison (involving the operand that we just found). When a normal relation is found, the appropriate auxiliary calls $__int_compare:nnN$ where #1 is $_if_int_compare:w$ or $_reverse_if:N$ $_if_int_compare:w$, #2 is the $\langle operand \rangle$, and #3 is one of $<$, $=$, or $>$. As announced earlier, we leave the $\langle operand \rangle$ for the previous conditional. If this conditional is true the result of the test is known, so we remove all tokens and return `false`. Otherwise, we apply the conditional #1 to the $\langle operand \rangle$ #2 and the comparison #3, and call $__int_compare:Nw$ to look for additional operands, after evaluating the following expression.

```

3582 \cs_new:cpn { \_\_int\_compare\_end\_=:NNw } #1#2#3 e #4 \q\_stop
3583 {
3584     {#3} \exp\_stop\_f:
3585     \prg\_return\_false: \else: \prg\_return\_true: \fi:
3586 }
3587 \cs_new:Npn \_\_int\_compare:nnN #1#2#3
3588 {
3589     {#2} \exp\_stop\_f:
3590     \prg\_return\_false: \exp\_after:wN \use\_none\_delimit\_by\_q\_stop:w
3591     \fi:
3592     #1 #2 #3 \exp\_after:wN \_\_int\_compare:Nw \_\_int\_value:w \_\_int\_eval:w
3593 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding $__prg_compare_error:Nw \langle token \rangle$ responsible for error detection.

```

3594 \cs_new:cpn { \_\_int\_compare\_=:NNw } #1#2#3 =
3595 { \_\_int\_compare:nnN { \_reverse\_if:N \_if\_int\_compare:w } {#3} = }
3596 \cs_new:cpn { \_\_int\_compare\_<:NNw } #1#2#3 <
3597 { \_\_int\_compare:nnN { \_reverse\_if:N \_if\_int\_compare:w } {#3} < }
3598 \cs_new:cpn { \_\_int\_compare\_>:NNw } #1#2#3 >
3599 { \_\_int\_compare:nnN { \_reverse\_if:N \_if\_int\_compare:w } {#3} > }
3600 \cs_new:cpn { \_\_int\_compare\_=:NNw } #1#2#3 ==
3601 { \_\_int\_compare:nnN { \_reverse\_if:N \_if\_int\_compare:w } {#3} = }
3602 \cs_new:cpn { \_\_int\_compare\_!=:NNw } #1#2#3 !=
3603 { \_\_int\_compare:nnN { \_if\_int\_compare:w } {#3} = }
3604 \cs_new:cpn { \_\_int\_compare\_<=:NNw } #1#2#3 <=
3605 { \_\_int\_compare:nnN { \_if\_int\_compare:w } {#3} > }
3606 \cs_new:cpn { \_\_int\_compare\_>=:NNw } #1#2#3 >=
3607 { \_\_int\_compare:nnN { \_if\_int\_compare:w } {#3} < }

```

(End definition for $_int_compare:n$. These functions are documented on page 66.)

$_int_compare_p:nNn$
 $_int_compare:nNnTF$

More efficient but less natural in typing.

```

3608 \prg\_new\_conditional:Npnn \_int\_compare:nNn #1#2#3 { p , T , F , TF }
3609 {
3610     \_if\_int\_compare:w \_\_int\_eval:w #1 #2 \_\_int\_eval:w #3 \_\_int\_eval\_end:

```

```

3611     \prg_return_true:
3612   \else:
3613     \prg_return_false:
3614   \fi:
3615 }

```

(End definition for `\int_compare:nNn`. These functions are documented on page 65.)

`\int_case:nnn` For integer cases, the first task to fully expand the check condition. After that, a loop is started to compare each possible value and stop if the test is true. The tested value is put at the end to ensure that there is necessarily a match, which will fire the “else” pathway. The leading `\romannumeral` triggers an expansion which is then stopped in `__int_case_end:nw`.

```

3616 \cs_new:Npn \int_case:nnn #1
3617 {
3618   \tex_romannumeral:D
3619   \exp_args:Nf \__int_case:nnn { \int_eval:n {#1} }
3620 }
3621 \cs_new:Npn \__int_case:nnn #1#2#3
3622 { \__int_case:nw {#1} #2 {#1} {#3} \q_recursion_stop }
3623 \cs_new:Npn \__int_case:nw #1#2#3
3624 {
3625   \int_compare:nNnTF {#1} = {#2}
3626   { \__int_case_end:nw {#3} }
3627   { \__int_case:nw {#1} }
3628 }
3629 \cs_new_eq:NN \__int_case_end:nw \__prg_case_end:nw

```

(End definition for `\int_case:nnn`. This function is documented on page 67.)

`\int_if_odd:p:n` A predicate function.

```

\int_if_odd:nTF 3630 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
\int_if_even:p:n 3631 {
\int_if_even:nTF 3632   \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
3633     \prg_return_true:
3634   \else:
3635     \prg_return_false:
3636   \fi:
3637 }
3638 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
3639 {
3640   \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
3641     \prg_return_false:
3642   \else:
3643     \prg_return_true:
3644   \fi:
3645 }

```

(End definition for `\int_if_odd:n`. These functions are documented on page 67.)

8.6 Integer expression loops

`\int_while_do:nn` These are quite easy given the above functions. The `while` versions test first and then execute the body. The `do_while` does it the other way round.

```
\int_until_do:nn
\int_do_while:nn
\int_do_until:nn
3646 \cs_new:Npn \int_while_do:nn #1#2
3647 {
3648   \int_compare:nT {#1}
3649   {
3650     #2
3651     \int_while_do:nn {#1} {#2}
3652   }
3653 }
3654 \cs_new:Npn \int_until_do:nn #1#2
3655 {
3656   \int_compare:nF {#1}
3657   {
3658     #2
3659     \int_until_do:nn {#1} {#2}
3660   }
3661 }
3662 \cs_new:Npn \int_do_while:nn #1#2
3663 {
3664   #2
3665   \int_compare:nT {#1}
3666   { \int_do_while:nn {#1} {#2} }
3667 }
3668 \cs_new:Npn \int_do_until:nn #1#2
3669 {
3670   #2
3671   \int_compare:nF {#1}
3672   { \int_do_until:nn {#1} {#2} }
3673 }
```

(End definition for `\int_while_do:nn`. This function is documented on page 68.)

`\int_while_do:nNnn` As above but not using the more natural syntax.

```
\int_until_do:nNnn
\int_do_while:nNnn
\int_do_until:nNnn
3674 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
3675 {
3676   \int_compare:nNnT {#1} #2 {#3}
3677   {
3678     #4
3679     \int_while_do:nNnn {#1} #2 {#3} {#4}
3680   }
3681 }
3682 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
3683 {
3684   \int_compare:nNnF {#1} #2 {#3}
3685   {
3686     #4
3687     \int_until_do:nNnn {#1} #2 {#3} {#4}
```

```

3688     }
3689   }
3690   \cs_new:Npn \int_do_while:nNnn #1#2#3#4
3691   {
3692     #4
3693     \int_compare:nNnT {#1} #2 {#3}
3694     { \int_do_while:nNnn {#1} #2 {#3} {#4} }
3695   }
3696   \cs_new:Npn \int_do_until:nNnn #1#2#3#4
3697   {
3698     #4
3699     \int_compare:nNnF {#1} #2 {#3}
3700     { \int_do_until:nNnn {#1} #2 {#3} {#4} }
3701   }

```

(End definition for `\int_while_do:nNnn`. This function is documented on page 67.)

8.7 Integer step functions

`\int_step_function:nnnN` 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).

`__int_step:NnnnN`

```

3702   \cs_new:Npn \int_step_function:nnnN #1#2#3#4
3703   {
3704     \int_compare:nNnTF {#2} > \c_zero
3705     { \exp_args:Nnf \__int_step:NnnnN > }
3706     {
3707       \int_compare:nNnTF {#2} = \c_zero
3708       {
3709         \__msg_kernel_expandable_error:nnn { kernel } { zero-step } {#4}
3710         \use_none:nnnn
3711       }
3712       { \exp_args:Nnf \__int_step:NnnnN < }
3713     }
3714     { \int_eval:n {#1} } {#2} {#3} #4
3715   }
3716   \cs_new:Npn \__int_step:NnnnN #1#2#3#4#5
3717   {
3718     \int_compare:nNnF {#2} #1 {#4}
3719     {
3720       #5 {#2}
3721       \exp_args:Nnf \__int_step:NnnnN
3722       #1 { \int_eval:n { #2 + #3 } } {#3} {#4} #5
3723     }
3724   }

```

(End definition for `\int_step_function:nnnN`. This function is documented on page 69.)

`\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.

```

3725 \cs_new_protected_nopar:Npn \int_step_inline:nnnn
3726 {
3727   \int_gincr:N \g__prg_map_int
3728   \exp_args:NNc \__int_step:NNnnnn
3729   \cs_gset_nopar:Npn
3730   { __prg_map_ \int_use:N \g__prg_map_int :w }
3731 }
3732 \cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
3733 {
3734   \int_gincr:N \g__prg_map_int
3735   \exp_args:NNc \__int_step:NNnnnn
3736   \cs_gset_nopar:Npx
3737   { __prg_map_ \int_use:N \g__prg_map_int :w }
3738   {#1}{#2}{#3}
3739   {
3740     \tl_set:Nn \exp_not:N #4 {##1}
3741     \exp_not:n {#5}
3742   }
3743 }
3744 \cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
3745 {
3746   #1 #2 ##1 {#6}
3747   \int_step_function:nnnN {#3} {#4} {#5} #2
3748   \__prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__prg_map_int }
3749 }

```

(End definition for `\int_step_inline:nnnn`. This function is documented on page 69.)

8.8 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```

3750 \cs_new:Npn \int_to_arabic:n #1 { \int_eval:n {#1} }

```

(End definition for `\int_to_arabic:n`. This function is documented on page 69.)

`\int_to_symbols:nnn`
`__int_to_symbols:nnnn`

For conversion of integers to arbitrary symbols the method is in general as follows. The input number (`#1`) is compared to the total number of symbols available at each place (`#2`). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an `f`-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

```

3751 \cs_new:Npn \int_to_symbols:nnn #1#2#3
3752 {
3753   \int_compare:nNnTF {#1} > {#2}
3754   {

```



```

3755 \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
3756 {
3757   \int_case:nnn
3758     { 1 + \int_mod:nn { #1 - 1 } {#2} }
3759     {#3} { }
3760 }
3761 {#1} {#2} {#3}
3762 }
3763 { \int_case:nnn {#1} {#3} { } }
3764 }
3765 \cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
3766 {
3767   \exp_args:Nf \int_to_symbols:nnn
3768   { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
3769   #1
3770 }

```

(End definition for `\int_to_symbols:nnn`. This function is documented on page 70.)

`\int_to_alph:n` These both use the above function with input functions that make sense for the alphabet
`\int_to_Alph:n` in English.

```

3771 \cs_new:Npn \int_to_alph:n #1
3772 {
3773   \int_to_symbols:nnn {#1} { 26 }
3774   {
3775     { 1 } { a }
3776     { 2 } { b }
3777     { 3 } { c }
3778     { 4 } { d }
3779     { 5 } { e }
3780     { 6 } { f }
3781     { 7 } { g }
3782     { 8 } { h }
3783     { 9 } { i }
3784     { 10 } { j }
3785     { 11 } { k }
3786     { 12 } { l }
3787     { 13 } { m }
3788     { 14 } { n }
3789     { 15 } { o }
3790     { 16 } { p }
3791     { 17 } { q }
3792     { 18 } { r }
3793     { 19 } { s }
3794     { 20 } { t }
3795     { 21 } { u }
3796     { 22 } { v }
3797     { 23 } { w }
3798     { 24 } { x }
3799     { 25 } { y }

```

```

3800         { 26 } { z }
3801     }
3802 }
3803 \cs_new:Npn \int_to_Alph:n #1
3804 {
3805     \int_to_symbols:nnn {#1} { 26 }
3806     {
3807         { 1 } { A }
3808         { 2 } { B }
3809         { 3 } { C }
3810         { 4 } { D }
3811         { 5 } { E }
3812         { 6 } { F }
3813         { 7 } { G }
3814         { 8 } { H }
3815         { 9 } { I }
3816         { 10 } { J }
3817         { 11 } { K }
3818         { 12 } { L }
3819         { 13 } { M }
3820         { 14 } { N }
3821         { 15 } { O }
3822         { 16 } { P }
3823         { 17 } { Q }
3824         { 18 } { R }
3825         { 19 } { S }
3826         { 20 } { T }
3827         { 21 } { U }
3828         { 22 } { V }
3829         { 23 } { W }
3830         { 24 } { X }
3831         { 25 } { Y }
3832         { 26 } { Z }
3833     }
3834 }

```

(End definition for `\int_to_alph:n` and `\int_to_Alph:n`. These functions are documented on page 70.)

\int_to_base:nn Converting from base ten (#1) to a second base (#2) starts with computing #1: if it is a complicated calculation, we shouldn't perform it twice. Then check the sign, store it, either - or `\c_empty_tl`, and feed the absolute value to the next auxiliary function.

```

\__int_to_base:nn
\__int_to_base:nnN
\__int_to_base:nnnN
\__int_to_letter:n
3835 \cs_new:Npn \int_to_base:nn #1
3836 { \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } }
3837 \cs_new:Npn \__int_to_base:nn #1#2
3838 {
3839     \int_compare:nNnTF {#1} < \c_zero
3840     { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
3841     { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
3842 }

```

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.

```

3843 \cs_new:Npn \__int_to_base:nnN #1#2#3
3844 {
3845   \int_compare:nNnTF {#1} < {#2}
3846   { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
3847   {
3848     \exp_args:Nf \__int_to_base:nnnN
3849     { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
3850     {#1}
3851     {#2}
3852     #3
3853   }
3854 }
3855 \cs_new:Npn \__int_to_base:nnnN #1#2#3#4
3856 {
3857   \exp_args:Nf \__int_to_base:nnN
3858   { \int_div_truncate:nn {#2} {#3} }
3859   {#3}
3860   #4
3861   #1
3862 }

```

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use `\int_case:nnn`, 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:`.

```

3863 \cs_new:Npn \__int_to_letter:n #1
3864 {
3865   \exp_after:wN \exp_after:wN
3866   \if_case:w \__int_eval:w #1 - \c_ten \__int_eval_end:
3867   A
3868   \or: B
3869   \or: C
3870   \or: D
3871   \or: E
3872   \or: F
3873   \or: G
3874   \or: H
3875   \or: I
3876   \or: J
3877   \or: K
3878   \or: L

```

```

3879     \or: M
3880     \or: N
3881     \or: O
3882     \or: P
3883     \or: Q
3884     \or: R
3885     \or: S
3886     \or: T
3887     \or: U
3888     \or: V
3889     \or: W
3890     \or: X
3891     \or: Y
3892     \or: Z
3893     \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
3894     \fi:
3895 }

```

(End definition for \int_to_base:nn. This function is documented on page 71.)

```

\int_to_binary:n Wrappers around the generic function.
\int_to_hexadecimal:n 3896 \cs_new:Npn \int_to_binary:n #1
\int_to_octal:n       3897 { \int_to_base:nn {#1} { 2 } }
                     3898 \cs_new:Npn \int_to_hexadecimal:n #1
                     3899 { \int_to_base:nn {#1} { 16 } }
                     3900 \cs_new:Npn \int_to_octal:n #1
                     3901 { \int_to_base:nn {#1} { 8 } }

```

(End definition for \int_to_binary:n, \int_to_hexadecimal:n, and \int_to_octal:n. These functions are documented on page 71.)

```

\int_to_roman:n The \__int_to_roman:w primitive creates tokens of category code 12 (other). Usually,
\int_to_Roman:n what is actually wanted is letters. The approach here is to convert the output of the
                  primitive into letters using appropriate control sequence names. That keeps everything
                  expandable. The loop will be terminated by the conversion of the Q.
\__int_to_roman:N
\__int_to_roman:N
\__int_to_roman_i:w 3902 \cs_new:Npn \int_to_roman:n #1
\__int_to_roman_v:w 3903 {
\__int_to_roman_x:w 3904   \exp_after:wN \__int_to_roman:N
\__int_to_roman_l:w 3905   \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_roman_c:w 3906 }
\__int_to_roman_d:w 3907 \cs_new:Npn \__int_to_roman:N #1
\__int_to_roman_m:w 3908 {
\__int_to_roman_Q:w 3909   \use:c { __int_to_roman_ #1 :w }
\__int_to_Roman_i:w 3910   \__int_to_roman:N
\__int_to_Roman_v:w 3911 }
\__int_to_Roman_x:w 3912 \cs_new:Npn \int_to_Roman:n #1
\__int_to_Roman_l:w 3913 {
\__int_to_Roman_c:w 3914   \exp_after:wN \__int_to_Roman_aux:N
\__int_to_Roman_d:w 3915   \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_Roman_m:w 3916 }
\__int_to_Roman_Q:w 3917 \cs_new:Npn \__int_to_Roman_aux:N #1

```

```

3918 {
3919   \use:c { __int_to_Roman_ #1 :w }
3920   \__int_to_Roman_aux:N
3921 }
3922 \cs_new_nopar:Npn \__int_to_roman_i:w { i }
3923 \cs_new_nopar:Npn \__int_to_roman_v:w { v }
3924 \cs_new_nopar:Npn \__int_to_roman_x:w { x }
3925 \cs_new_nopar:Npn \__int_to_roman_l:w { l }
3926 \cs_new_nopar:Npn \__int_to_roman_c:w { c }
3927 \cs_new_nopar:Npn \__int_to_roman_d:w { d }
3928 \cs_new_nopar:Npn \__int_to_roman_m:w { m }
3929 \cs_new_nopar:Npn \__int_to_roman_Q:w #1 { }
3930 \cs_new_nopar:Npn \__int_to_Roman_i:w { I }
3931 \cs_new_nopar:Npn \__int_to_Roman_v:w { V }
3932 \cs_new_nopar:Npn \__int_to_Roman_x:w { X }
3933 \cs_new_nopar:Npn \__int_to_Roman_l:w { L }
3934 \cs_new_nopar:Npn \__int_to_Roman_c:w { C }
3935 \cs_new_nopar:Npn \__int_to_Roman_d:w { D }
3936 \cs_new_nopar:Npn \__int_to_Roman_m:w { M }
3937 \cs_new:Npn \__int_to_Roman_Q:w #1 { }

```

(End definition for `\int_to_roman:n` and `\int_to_Roman:n`. These functions are documented on page [71](#).)

8.9 Converting from other formats to integers

```

__int_get_sign:n
__int_get_digits:n
\__int_get_sign_and_digits:nNNN
\__int_get_sign_and_digits:oNNN

```

Finding a number and its sign requires dealing with an arbitrary list of + and - symbols. This is done by working through token by token until there is something else at the start of the input. The sign of the input is tracked by the first Boolean used by the auxiliary function.

```

3938 \cs_new:Npn \__int_get_sign:n #1
3939 {
3940   \__int_get_sign_and_digits:nNNN {#1}
3941   \c_true_bool \c_true_bool \c_false_bool
3942 }
3943 \cs_new:Npn \__int_get_digits:n #1
3944 {
3945   \__int_get_sign_and_digits:nNNN {#1}
3946   \c_true_bool \c_false_bool \c_true_bool
3947 }

```

The auxiliary loops through, finding sign tokens and removing them. The sign itself is carried through as a flag.

```

3948 \cs_new:Npn \__int_get_sign_and_digits:nNNN #1#2#3#4
3949 {
3950   \exp_args:Nf \tl_if_head_eq_charcode:nNTF {#1} -
3951   {
3952     \bool_if:NTF #2
3953     {
3954       \__int_get_sign_and_digits:oNNN

```

```

3955         { \use_none:n #1 } \c_false_bool #3#4
3956     }
3957     {
3958         \__int_get_sign_and_digits:oNNN
3959         { \use_none:n #1 } \c_true_bool #3#4
3960     }
3961 }
3962 {
3963     \exp_args:Nf \tl_if_head_eq_charcode:nNTF {#1} +
3964     { \__int_get_sign_and_digits:oNNN { \use_none:n #1 } #2#3#4 }
3965     {
3966         \bool_if:NT #3 { \bool_if:NF #2 - }
3967         \bool_if:NT #4 {#1}
3968     }
3969 }
3970 }
3971 \cs_generate_variant:Nn \__int_get_sign_and_digits:nNNN { o }

```

(End definition for `__int_get_sign:n`. This function is documented on page ??.)

\int_from_alph:n 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
3972 \cs_new:Npn \int_from_alph:n #1
3973 {
3974     \int_eval:n
3975     {
3976         \__int_get_sign:n {#1}
3977         \exp_args:Nf \__int_from_alph:n { \__int_get_digits:n {#1} }
3978     }
3979 }
3980 \cs_new:Npn \__int_from_alph:n #1
3981 { \__int_from_alph:nN { 0 } #1 \q_nil }
3982 \cs_new:Npn \__int_from_alph:nN #1#2
3983 {
3984     \quark_if_nil:NNTF #2
3985     {#1}
3986     {
3987         \exp_args:Nf \__int_from_alph:nN
3988         { \int_eval:n { #1 * 26 + \__int_from_alph:N #2 } }
3989     }
3990 }
3991 \cs_new:Npn \__int_from_alph:N #1
3992 { \int_eval:n { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } } }

```

(End definition for `\int_from_alph:n`. This function is documented on page 71.)

\int_from_base:nn 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
3993 \cs_new:Npn \int_from_base:nn #1#2
3994 {

```

```

3995     \int_eval:n
3996     {
3997         \__int_get_sign:n {#1}
3998         \exp_args:Nf \__int_from_base:nn
3999         { \__int_get_digits:n {#1} } {#2}
4000     }
4001 }
4002 \cs_new:Npn \__int_from_base:nn #1#2
4003 { \__int_from_base:nnN { 0 } { #2 } #1 \q_nil }
4004 \cs_new:Npn \__int_from_base:nnN #1#2#3
4005 {
4006     \quark_if_nil:NTF #3
4007     {#1}
4008     {
4009         \exp_args:Nf \__int_from_base:nnN
4010         { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
4011         {#2}
4012     }
4013 }

```

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.

```

4014 \cs_new:Npn \__int_from_base:N #1
4015 {
4016     \int_compare:nNnTF { '#1 } < { 58 }
4017     {#1}
4018     {
4019         \int_eval:n
4020         { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
4021     }
4022 }

```

(End definition for `\int_from_base:nn`. This function is documented on page 72.)

```

\int_from_binary:n
\int_from_hexadecimal:n
\int_from_octal:n

```

Wrappers around the generic function.

```

4023 \cs_new:Npn \int_from_binary:n #1
4024 { \int_from_base:nn {#1} \c_two }
4025 \cs_new:Npn \int_from_hexadecimal:n #1
4026 { \int_from_base:nn {#1} \c_sixteen }
4027 \cs_new:Npn \int_from_octal:n #1
4028 { \int_from_base:nn {#1} \c_eight }

```

(End definition for `\int_from_binary:n`, `\int_from_hexadecimal:n`, and `\int_from_octal:n`. These functions are documented on page 72.)

```

\c__int_from_roman_i_int
\c__int_from_roman_v_int
\c__int_from_roman_x_int
\c__int_from_roman_l_int
\c__int_from_roman_c_int
\c__int_from_roman_d_int
\c__int_from_roman_m_int
\c__int_from_roman_I_int
\c__int_from_roman_V_int
\c__int_from_roman_X_int
\c__int_from_roman_L_int
\c__int_from_roman_C_int
\c__int_from_roman_D_int
\c__int_from_roman_M_int

```

Constants used to convert from Roman numerals to integers.

```

4029 \int_const:cn { c__int_from_roman_i_int } { 1 }
4030 \int_const:cn { c__int_from_roman_v_int } { 5 }
4031 \int_const:cn { c__int_from_roman_x_int } { 10 }
4032 \int_const:cn { c__int_from_roman_l_int } { 50 }
4033 \int_const:cn { c__int_from_roman_c_int } { 100 }
4034 \int_const:cn { c__int_from_roman_d_int } { 500 }

```

```

4035 \int_const:cn { c__int_from_roman_m_int } { 1000 }
4036 \int_const:cn { c__int_from_roman_I_int } { 1 }
4037 \int_const:cn { c__int_from_roman_V_int } { 5 }
4038 \int_const:cn { c__int_from_roman_X_int } { 10 }
4039 \int_const:cn { c__int_from_roman_L_int } { 50 }
4040 \int_const:cn { c__int_from_roman_C_int } { 100 }
4041 \int_const:cn { c__int_from_roman_D_int } { 500 }
4042 \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`

```

4043 \cs_new:Npn \int_from_roman:n #1
4044 {
4045   \tl_if_blank:nF {#1}
4046   {
4047     \exp_after:wN \__int_from_roman_end:w
4048     \__int_value:w \__int_eval:w
4049     \__int_from_roman:NN #1 Q \q_stop
4050   }
4051 }
4052 \cs_new:Npn \__int_from_roman:NN #1#2
4053 {
4054   \str_if_eq:nnTF {#1} { Q }
4055   {#1#2}
4056   {
4057     \str_if_eq:nnTF {#2} { Q }
4058     {
4059       \int_if_exist:cF { c__int_from_roman_ #1 _int }
4060       { \__int_from_roman_clean_up:w }
4061       +
4062       \use:c { c__int_from_roman_ #1 _int }
4063       #2
4064     }
4065     {
4066       \int_if_exist:cF { c__int_from_roman_ #1 _int }
4067       { \__int_from_roman_clean_up:w }
4068       \int_if_exist:cF { c__int_from_roman_ #2 _int }
4069       { \__int_from_roman_clean_up:w }
4070       \int_compare:nNnTF
4071       { \use:c { c__int_from_roman_ #1 _int } }
4072       <
4073       { \use:c { c__int_from_roman_ #2 _int } }
4074       {
4075         + \use:c { c__int_from_roman_ #2 _int }
4076         - \use:c { c__int_from_roman_ #1 _int }
4077         \__int_from_roman:NN
4078       }
4079     }

```



```

4080         + \use:c { c__int_from_roman_ #1 _int }
4081         \__int_from_roman:NN #2
4082     }
4083 }
4084 }
4085 }
4086 \cs_new:Npn \__int_from_roman_end:w #1 Q #2 \q_stop
4087 { \tl_if_empty:nTF {#2} {#1} {#2} }
4088 \cs_new:Npn \__int_from_roman_clean_up:w #1 Q { + 0 Q -1 }

```

(End definition for `\int_from_roman:n`. This function is documented on page 72.)

8.10 Viewing integer

```

\int_show:N
\int_show:c

```

```

4089 \cs_new_eq:NN \int_show:N \__kernel_register_show:N
4090 \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.

```

4091 \cs_new_protected:Npn \int_show:n #1
4092 { \etex_showtokens:D \exp_after:wN { \int_use:N \__int_eval:w #1 } }

```

(End definition for `\int_show:n`. This function is documented on page 72.)

8.11 Constant integers

`\c_minus_one` This is needed early, and so is in `l3basics`
(End definition for `\c_minus_one`. This variable is documented on page 73.)

`\c_zero` Again, one in `l3basics` for obvious reasons.
(End definition for `\c_zero`. This variable is documented on page 73.)

`\c_six` Once again, in `l3basics`.
(End definition for `\c_six` and `\c_seven`. These functions are documented on page 73.)

`\c_twelve`
`\c_one`
`\c_sixteen`
`\c_two` Low-number values not previously defined.

```

4093 \int_const:Nn \c_one      { 1 }
4094 \int_const:Nn \c_two      { 2 }
4095 \int_const:Nn \c_three    { 3 }
4096 \int_const:Nn \c_four     { 4 }
4097 \int_const:Nn \c_five     { 5 }
4098 \int_const:Nn \c_eight    { 8 }
4099 \int_const:Nn \c_nine     { 9 }
4100 \int_const:Nn \c_ten       { 10 }
4101 \int_const:Nn \c_eleven    { 11 }
4102 \int_const:Nn \c_thirteen { 13 }
4103 \int_const:Nn \c_fourteen { 14 }
4104 \int_const:Nn \c_fifteen  { 15 }

```

(End definition for `\c_one` and others. These variables are documented on page 73.)

`\c_thirty_two` One middling value.

```
4105 \int_const:Nn \c_thirty_two { 32 }
```

(End definition for `\c_thirty_two`. This variable is documented on page 73.)

`\c_two_hundred_fifty_five` Two classic mid-range integer constants.

```
\c_two_hundred_fifty_six 4106 \int_const:Nn \c_two_hundred_fifty_five { 255 }
4107 \int_const:Nn \c_two_hundred_fifty_six { 256 }
```

(End definition for `\c_two_hundred_fifty_five` and `\c_two_hundred_fifty_six`. These variables are documented on page 73.)

`\c_one_hundred` Simple runs of powers of ten.

```
\c_one_thousand 4108 \int_const:Nn \c_one_hundred { 100 }
\c_ten_thousand 4109 \int_const:Nn \c_one_thousand { 1000 }
4110 \int_const:Nn \c_ten_thousand { 10000 }
```

(End definition for `\c_one_hundred`, `\c_one_thousand`, and `\c_ten_thousand`. These variables are documented on page 73.)

`\c_max_int` The largest number allowed is $2^{31} - 1$

```
4111 \int_const:Nn \c_max_int { 2 147 483 647 }
```

(End definition for `\c_max_int`. This variable is documented on page 73.)

8.12 Scratch integers

`\l_tmpa_int` We provide two local and two global scratch counters, maybe we need more or less.

```
\l_tmpb_int 4112 \int_new:N \l_tmpa_int
\g_tmpa_int 4113 \int_new:N \l_tmpb_int
\g_tmpb_int 4114 \int_new:N \g_tmpa_int
4115 \int_new:N \g_tmpb_int
```

(End definition for `\l_tmpa_int` and `\l_tmpb_int`. These functions are documented on page 73.)

8.13 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

`\int_convert_from_base_ten:nn` Some simple renames.

```
\int_convert_to_symbols:nnn 4116 <deprecated>
\int_convert_to_base_ten:nn 4117 \cs_new_eq:NN \int_convert_from_base_ten:nn \int_to_base:nn
4118 \cs_new_eq:NN \int_convert_to_symbols:nnn \int_to_symbols:nnn
4119 \cs_new_eq:NN \int_convert_to_base_ten:nn \int_from_base:nn
4120 </deprecated>
```

(End definition for `\int_convert_from_base_ten:nn`. This function is documented on page ??.)

`\int_to_symbol:n` This is rather too tied to L^AT_EX 2_ε.

`\int_to_symbol_math:n`

`\int_to_symbol_text:n`

```

4121 <*deprecated>
4122 \cs_new_nopar:Npn \int_to_symbol:n
4123 {
4124   \scan_align_safe_stop:
4125   \mode_if_math:TF
4126     { \int_to_symbol_math:n }
4127     { \int_to_symbol_text:n }
4128 }
4129 \cs_new:Npn \int_to_symbol_math:n #1
4130 {
4131   \int_to_symbols:nnn {#1} { 9 }
4132   {
4133     { 1 } { * }
4134     { 2 } { \dagger }
4135     { 3 } { \ddagger }
4136     { 4 } { \mathsection }
4137     { 5 } { \mathparagraph }
4138     { 6 } { \| }
4139     { 7 } { ** }
4140     { 8 } { \dagger \dagger }
4141     { 9 } { \ddagger \ddagger }
4142   }
4143 }
4144 \cs_new:Npn \int_to_symbol_text:n #1
4145 {
4146   \int_to_symbols:nnn {#1} { 9 }
4147   {
4148     { 1 } { \textasteriskcentered }
4149     { 2 } { \textdagger }
4150     { 3 } { \textdaggerdbl }
4151     { 4 } { \textsection }
4152     { 5 } { \textparagraph }
4153     { 6 } { \textbardbl }
4154     { 7 } { \textasteriskcentered \textasteriskcentered }
4155     { 8 } { \textdagger \textdagger }
4156     { 9 } { \textdaggerdbl \textdaggerdbl }
4157   }
4158 }
4159 </deprecated>

```

(End definition for `\int_to_symbol:n`. This function is documented on page ??.)

`\if_num:w` Deprecated 2012-05-30 for removal after 2012-11-30.

```

4160 <*deprecated>
4161 \cs_new_eq:NN \if_num:w \if_int_compare:w
4162 </deprecated>

```

(End definition for `\if_num:w`. This function is documented on page ??.)

`\l_tmpc_int` Deprecated 2012-07-04 for removal after 2012-12-31.

```

4163 <*deprecated>
4164 \int_new:N \l_tmpc_int
4165 </deprecated>
(End definition for \l_tmpc_int. This variable is documented on page ??.)

\int_eval:w Deprecated 2012-07-13 for removal after 2012-12-31.
\int_eval_end: 4166 <*deprecated>
4167 \cs_new_eq:NN \int_eval:w \__int_eval:w
4168 \cs_new_eq:NN \int_eval_end: \__int_eval_end:
4169 </deprecated>
(End definition for \int_eval:w and \int_eval_end:. These functions are documented on page ??.)

\int_value:w Deprecated 2012-07-14 for removal after 2012-12-31.
4170 <*deprecated>
4171 \cs_new_eq:NN \int_value:w \__int_value:w
4172 </deprecated>
(End definition for \int_value:w. This function is documented on page ??.)
4173 </initex | package>

```

9 l3skip implementation

```

4174 <*initex | package>
4175 <@@=dim>
4176 <*package>
4177 \ProvidesExplPackage
4178 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
4179 \__expl_package_check:
4180 </package>

```

9.1 Length primitives renamed

```

\if_dim:w Primitives renamed.
\__dim_eval:w 4181 \cs_new_eq:NN \if_dim:w \tex_ifdim:D
\__dim_eval_end: 4182 \cs_new_eq:NN \__dim_eval:w \etex_dimexpr:D
4183 \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 <dim> registers ...
\dim_new:c 4184 <*package>
4185 \cs_new_protected:Npn \dim_new:N #1
4186 {
4187   \__chk_if_free_cs:N #1
4188   \newdimen #1
4189 }
4190 </package>
4191 \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 4192 \cs_new_protected:Npn \dim_const:Nn #1
4193 {
4194   \dim_new:N #1
4195   \dim_gset:Nn #1
4196 }
4197 \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 4198 \cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_dim }
\dim_gzero:N 4199 \cs_new_protected:Npn \dim_gzero:N { \tex_global:D \dim_zero:N }
\dim_gzero:c 4200 \cs_generate_variant:Nn \dim_zero:N { c }
4201 \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 4202 \cs_new_protected:Npn \dim_zero_new:N #1
\dim_gzero_new:N 4203 { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
\dim_gzero_new:c 4204 \cs_new_protected:Npn \dim_gzero_new:N #1
4205 { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
4206 \cs_generate_variant:Nn \dim_zero_new:N { c }
4207 \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 4208 \prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N { TF , T , F , p }
\dim_if_exist:NTF 4209 \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 4210 \cs_new_protected:Npn \dim_set:Nn #1#2
\dim_gset:Nn 4211 { #1 ~ \_dim_eval:w #2 \_dim_eval_end: }
\dim_gset:cn 4212 \cs_new_protected:Npn \dim_gset:Nn { \tex_global:D \dim_set:Nn }
4213 \cs_generate_variant:Nn \dim_set:Nn { c }
4214 \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 4215 \cs_new_protected:Npn \dim_set_eq:NN #1#2 { #1 = #2 }
\dim_set_eq:Nc 4216 \cs_generate_variant:Nn \dim_set_eq:NN { c }
\dim_set_eq:cc 4217 \cs_generate_variant:Nn \dim_set_eq:NN { Nc , cc }
\dim_gset_eq:NN 4218 \cs_new_protected:Npn \dim_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\dim_gset_eq:cN 4219 \cs_generate_variant:Nn \dim_gset_eq:NN { c }
\dim_gset_eq:Nc 4220 \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 4221 \cs_new_protected:Npn \dim_add:Nn #1#2
\dim_gadd:Nn 4222 { \tex_advance:D #1 by \__dim_eval:w #2 \__dim_eval_end: }
\dim_gadd:cn 4223 \cs_new_protected:Npn \dim_gadd:Nn { \tex_global:D \dim_add:Nn }
\dim_sub:Nn 4224 \cs_generate_variant:Nn \dim_add:Nn { c }
\dim_sub:cn 4225 \cs_generate_variant:Nn \dim_gadd:Nn { c }
\dim_gsub:Nn 4226 \cs_new_protected:Npn \dim_sub:Nn #1#2
\dim_gsub:cn 4227 { \tex_advance:D #1 by - \__dim_eval:w #2 \__dim_eval_end: }
4228 \cs_new_protected:Npn \dim_gsub:Nn { \tex_global:D \dim_sub:Nn }
4229 \cs_generate_variant:Nn \dim_sub:Nn { c }
4230 \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 4231 \cs_new:Npn \dim_abs:n #1
\dim_max:nn 4232 {
\dim_min:nn 4233   \exp_after:wN \__dim_abs:N
\__dim_maxmin:wwN 4234   \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
4235 }
4236 \cs_new:Npn \__dim_abs:N #1
4237 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
4238 \cs_set:Npn \dim_max:nn #1#2
4239 {
4240   \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
4241   \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
4242   \dim_use:N \__dim_eval:w #2 ;
4243   >
4244   \__dim_eval_end:
4245 }
4246 \cs_set:Npn \dim_min:nn #1#2
4247 {
4248   \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
4249   \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
4250   \dim_use:N \__dim_eval:w #2 ;
4251   <
4252   \__dim_eval_end:
4253 }
4254 \cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
4255 {
4256   \if_dim:w #1 #3 #2 ~
4257   #1
4258   \else:
4259   #2
4260   \fi:

```

```
4261 }
```

(End definition for `\dim_abs:n`. This function is documented on page 77.)

`\dim_ratio:nn` With dimension expressions, something like `10 pt * (5 pt / 10 pt)` will not work. Instead, the ratio part needs to be converted to an integer expression. Using `__int_value:w` forces everything into `sp`, avoiding any decimal parts.

```
4262 \cs_new:Npn \dim_ratio:nn #1#2
4263 { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
4264 \cs_new:Npn \__dim_ratio:n #1
4265 { \__int_value:w \__dim_eval:w #1 \__dim_eval_end: }
```

(End definition for `\dim_ratio:nn`. This function is documented on page 78.)

9.5 Dimension expression conditionals

`\dim_compare_p:nNn` Simple comparison.

```
\dim_compare:nNnTF
4266 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
4267 {
4268   \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
4269   \prg_return_true: \else: \prg_return_false: \fi:
4270 }
```

(End definition for `\dim_compare:nNn`. These functions are documented on page 78.)

`\dim_compare_p:n` This code is adapted from the `\int_compare:nTF` function. First make sure that there is at least one relation operator, by evaluating a dimension expression with a trailing `__prg_compare_error:`. Just like for integers, the looping auxiliary `__dim_compare:wNN` closes a primitive conditional and opens a new one. It is actually easier to grab a dimension operand than an integer one, because once evaluated, dimensions all end with `pt` (with category other). Thus we do not need specific auxiliaries for the three “simple” relations `<`, `=`, and `>`.

```
\dim_compare:nTF
\__dim_compare:w
\__dim_compare:wNN
\__dim_compare_=w
\__dim_compare_!w
\__dim_compare<w
\__dim_compare>w
4271 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
4272 {
4273   \exp_after:wN \__dim_compare:w
4274   \dim_use:N \__dim_eval:w #1 \__prg_compare_error:
4275 }
4276 \cs_new:Npn \__dim_compare:w #1 \__prg_compare_error:
4277 {
4278   \exp_after:wN \if_false: \tex_romannumeral:D -‘0
4279   \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \q_stop
4280 }
4281 \exp_args:Nno \use:nn
4282 { \cs_new:Npn \__dim_compare:wNN #1 }
4283 { \tl_to_str:n {pt} }
4284 #2#3
4285 {
4286   \if_meaning:w = #3
4287   \use:c { __dim_compare_#2:w }
4288   \fi:
4289   #1 pt \exp_stop_f:
```

```

4290     \prg_return_false:
4291     \exp_after:wN \use_none_delimit_by_q_stop:w
4292     \fi:
4293     \reverse_if:N \if_dim:w #1 pt #2
4294     \exp_after:wN \__dim_compare:wNN
4295     \dim_use:N \__dim_eval:w #3
4296   }
4297   \cs_new:cpn { __dim_compare_ ! :w }
4298     #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
4299   \cs_new:cpn { __dim_compare_ = :w }
4300     #1 \__dim_eval:w = { #1 \__dim_eval:w }
4301   \cs_new:cpn { __dim_compare_ < :w }
4302     #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
4303   \cs_new:cpn { __dim_compare_ > :w }
4304     #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
4305   \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \q_stop
4306     { #1 \prg_return_false: \else: \prg_return_true: \fi: }

```

(End definition for \dim_compare:n. These functions are documented on page 79.)

\dim_case:nnn The dimension function is the same as the int version, so there is not much to say here.

```

\__dim_case_aux:nnn 4307 \cs_new:Npn \dim_case:nnn #1
\__dim_case_aux:nw 4308 {
\__dim_case_end:nw 4309   \tex_romannumeral:D
4310   \exp_args:Nf \__dim_case_aux:nnn { \dim_eval:n {#1} }
4311 }
4312 \cs_new:Npn \__dim_case_aux:nnn #1#2#3
4313 { \__dim_case_aux:nw {#1} #2 {#1} {#3} \q_recursion_stop }
4314 \cs_new:Npn \__dim_case_aux:nw #1#2#3
4315 {
4316   \dim_compare:nNnTF {#1} = {#2}
4317     { \__dim_case_end:nw {#3} }
4318     { \__dim_case_aux:nw {#1} }
4319 }
4320 \cs_new_eq:NN \__dim_case_end:nw \__prg_case_end:nw

```

(End definition for \dim_case:nnn. This function is documented on page 80.)

9.6 Dimension expression loops

\dim_while_do:nn while_do and do_while functions for dimensions. Same as for the int type only the names have changed.

```

\dim_while_do:nn 4321 \cs_set:Npn \dim_while_do:nn #1#2
\dim_until_do:nn 4322 {
\dim_do_while:nn 4323   \dim_compare:nT {#1}
\dim_do_until:nn 4324     {
4325       #2
4326       \dim_while_do:nn {#1} {#2}
4327     }
4328   }
4329 \cs_set:Npn \dim_until_do:nn #1#2

```



```

4330 {
4331   \dim_compare:nF {#1}
4332   {
4333     #2
4334     \dim_until_do:nn {#1} {#2}
4335   }
4336 }
4337 \cs_set:Npn \dim_do_while:nn #1#2
4338 {
4339   #2
4340   \dim_compare:nT {#1}
4341   { \dim_do_while:nn {#1} {#2} }
4342 }
4343 \cs_set:Npn \dim_do_until:nn #1#2
4344 {
4345   #2
4346   \dim_compare:nF {#1}
4347   { \dim_do_until:nn {#1} {#2} }
4348 }

```

(End definition for \dim_while_do:nn. This function is documented on page 81.)

\dim_while_do:nNnn **\dim_until_do:nNnn** **\dim_do_while:nNnn** **\dim_do_until:nNnn** while_do and do_while functions for dimensions. Same as for the int type only the names have changed.

```

4349 \cs_set:Npn \dim_while_do:nNnn #1#2#3#4
4350 {
4351   \dim_compare:nNnT {#1} #2 {#3}
4352   {
4353     #4
4354     \dim_while_do:nNnn {#1} #2 {#3} {#4}
4355   }
4356 }
4357 \cs_set:Npn \dim_until_do:nNnn #1#2#3#4
4358 {
4359   \dim_compare:nNnF {#1} #2 {#3}
4360   {
4361     #4
4362     \dim_until_do:nNnn {#1} #2 {#3} {#4}
4363   }
4364 }
4365 \cs_set:Npn \dim_do_while:nNnn #1#2#3#4
4366 {
4367   #4
4368   \dim_compare:nNnT {#1} #2 {#3}
4369   { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
4370 }
4371 \cs_set:Npn \dim_do_until:nNnn #1#2#3#4
4372 {
4373   #4
4374   \dim_compare:nNnF {#1} #2 {#3}

```

```

4375     { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
4376   }

```

(End definition for `\dim_while_do:nNnn`. This function is documented on page 80.)

9.7 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

4377 \cs_new:Npn \dim_eval:n #1
4378 { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_eval:n`. This function is documented on page 81.)

`__dim_strip_bp:n`

```

4379 \cs_new:Npn \__dim_strip_bp:n #1
4380 { \__dim_strip_pt:n { 0.996 26 \__dim_eval:w #1 \__dim_eval_end: } }

```

(End definition for `__dim_strip_bp:n`.)

`__dim_strip_pt:n` A function which comes up often enough to deserve a place in the kernel. The idea here is that the input is assumed to be in `pt`, but can be given in other units, while the output is the value of the dimension in `pt` but with no units given. This is used a lot by low-level manipulations.

`__dim_strip_pt:w`

```

4381 \cs_new:Npn \__dim_strip_pt:n #1
4382 {
4383   \exp_after:wN
4384   \__dim_strip_pt:w \dim_use:N \__dim_eval:w #1 \__dim_eval_end: \q_stop
4385 }
4386 \use:x
4387 {
4388   \cs_new:Npn \exp_not:N \__dim_strip_pt:w
4389     ##1 . ##2 \tl_to_str:n { pt } ##3 \exp_not:N \q_stop
4390   {
4391     ##1
4392     \exp_not:N \int_compare:nNnT {##2} > \c_zero
4393     { . ##2 }
4394   }
4395 }

```

(End definition for `__dim_strip_pt:n`. This function is documented on page 89.)

`\dim_use:N` Accessing a $\langle dim \rangle$.

```

\dim_use:c 4396 \cs_new_eq:NN \dim_use:N \tex_the:D
4397 \cs_generate_variant:Nn \dim_use:N { c }

```

(End definition for `\dim_use:N` and `\dim_use:c`. These functions are documented on page ??.)

9.8 Viewing dim variables

`\dim_show:N` Diagnostics.

```
\dim_show:c 4398 \cs_new_eq:NN \dim_show:N \__kernel_register_show:N
4399 \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.

```
4400 \cs_new_protected:Npn \dim_show:n #1
4401 { \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 4402 \dim_const:Nn \c_zero_dim { 0 pt }
4403 \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 4404 \dim_new:N \l_tmpa_dim
\g_tmpa_dim 4405 \dim_new:N \l_tmpb_dim
\g_tmpb_dim 4406 \dim_new:N \g_tmpa_dim
4407 \dim_new:N \g_tmpb_dim
(End definition for \l_tmpa_dim and \l_tmpb_dim. These functions are documented on page 82.)
```

9.11 Creating and initialising skip variables

`\skip_new:N` Allocation of a new internal registers.

```
\skip_new:c 4408 <*package>
4409 \cs_new_protected:Npn \skip_new:N #1
4410 {
4411   \__chk_if_free_cs:N #1
4412   \newskip #1
4413 }
4414 </package>
4415 \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 4416 \cs_new_protected:Npn \skip_const:Nn #1
4417 {
4418   \skip_new:N #1
4419   \skip_gset:Nn #1
4420 }
4421 \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 4422 \cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }
\skip_gzero:N 4423 \cs_new_protected:Npn \skip_gzero:N { \tex_global:D \skip_zero:N }
\skip_gzero:c 4424 \cs_generate_variant:Nn \skip_zero:N { c }
4425 \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 4426 \cs_new_protected:Npn \skip_zero_new:N #1
\skip_gzero_new:N 4427 { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\skip_gzero_new:c 4428 \cs_new_protected:Npn \skip_gzero_new:N #1
4429 { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
4430 \cs_generate_variant:Nn \skip_zero_new:N { c }
4431 \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 4432 \prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N { TF , T , F , p }
\skip_if_exist:NTF 4433 \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 4434 \cs_new_protected:Npn \skip_set:Nn #1#2
\skip_gset:Nn 4435 { #1 ~ \etex_glueexpr:D #2 \scan_stop: }
\skip_gset:cn 4436 \cs_new_protected:Npn \skip_gset:Nn { \tex_global:D \skip_set:Nn }
4437 \cs_generate_variant:Nn \skip_set:Nn { c }
4438 \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 4439 \cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
\skip_set_eq:Nc 4440 \cs_generate_variant:Nn \skip_set_eq:NN { c }
\skip_set_eq:cc 4441 \cs_generate_variant:Nn \skip_set_eq:NN { Nc , cc }
\skip_gset_eq:NN 4442 \cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\skip_gset_eq:cN 4443 \cs_generate_variant:Nn \skip_gset_eq:NN { c }
\skip_gset_eq:Nc 4444 \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 4445 \cs_new_protected:Npn \skip_add:Nn #1#2
\skip_gadd:Nn 4446 { \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }
\skip_gadd:cn 4447 \cs_new_protected:Npn \skip_gadd:Nn { \tex_global:D \skip_add:Nn }
\skip_sub:Nn 4448 \cs_generate_variant:Nn \skip_add:Nn { c }
\skip_sub:cn 4449 \cs_generate_variant:Nn \skip_gadd:Nn { c }
\skip_gsub:Nn 4450 \cs_new_protected:Npn \skip_sub:Nn #1#2
\skip_gsub:cn 4451 { \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }
4452 \cs_new_protected:Npn \skip_gsub:Nn { \tex_global:D \skip_sub:Nn }
4453 \cs_generate_variant:Nn \skip_sub:Nn { c }
4454 \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:n` Comparing skips means doing two expansions to make strings, and then testing them.
`\skip_if_eq:nnTF` As a result, only equality is tested.

```

4455 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
4456 {
4457   \if_int_compare:w
4458     \pdfTeX_strcmp:D { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
4459     = \c_zero
4460     \prg_return_true:
4461   \else:
4462     \prg_return_false:
4463   \fi:
4464 }

```

(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.

```

4465 \cs_set_protected:Npn \__cs_tmp:w #1
4466 {
4467   \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
4468   {
4469     \exp_after:wN \__skip_if_finite:wwNw
4470     \skip_use:N \etex_glueexpr:D ##1 ; \prg_return_false:
4471     #1 ; \prg_return_true: \q_stop
4472   }
4473   \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \q_stop {##3}
4474 }
4475 \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.

```
4476 \cs_new:Npn \skip_eval:n #1
4477 { \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 4478 \cs_new_eq:NN \skip_use:N \tex_the:D
4479 \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 4480 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
\skip_horizontal:n 4481 \cs_new:Npn \skip_horizontal:n #1
\skip_vertical:N 4482 { \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop: }
\skip_vertical:c 4483 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
\skip_vertical:n 4484 \cs_new:Npn \skip_vertical:n #1
4485 { \skip_vertical:N \etex_glueexpr:D #1 \scan_stop: }
4486 \cs_generate_variant:Nn \skip_horizontal:N { c }
4487 \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 4488 \cs_new_eq:NN \skip_show:N \__kernel_register_show:N
4489 \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.

```
4490 \cs_new_protected:Npn \skip_show:n #1
4491 { \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 4492 \skip_const:Nn \c_zero_skip { \c_zero_dim }
4493 \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` 4494 `\skip_new:N \l_tmpa_skip`
`\g_tmpa_skip` 4495 `\skip_new:N \l_tmpb_skip`
`\g_tmpb_skip` 4496 `\skip_new:N \g_tmpa_skip`
4497 `\skip_new:N \g_tmpb_skip`

(End definition for `\l_tmpa_skip` and `\l_tmpb_skip`. These functions are documented on page 85.)

9.19 Creating and initialising muskip variables

`\muskip_new:N` And then we add muskips.

`\muskip_new:c` 4498 `*package`
4499 `\cs_new_protected:Npn \muskip_new:N #1`
4500 `{`
4501 `__chk_if_free_cs:N #1`
4502 `\newmuskip #1`
4503 `}`
4504 `\</package>`
4505 `\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` 4506 `\cs_new_protected:Npn \muskip_const:Nn #1`
4507 `{`
4508 `\muskip_new:N #1`
4509 `\muskip_gset:Nn #1`
4510 `}`
4511 `\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` 4512 `\cs_new_protected:Npn \muskip_zero:N #1`
`\muskip_gzero:N` 4513 `{ #1 \c_zero_muskip }`
`\muskip_gzero:c` 4514 `\cs_new_protected:Npn \muskip_gzero:N { \tex_global:D \muskip_zero:N }`
4515 `\cs_generate_variant:Nn \muskip_zero:N { c }`
4516 `\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` 4517 `\cs_new_protected:Npn \muskip_zero_new:N #1`
`\muskip_gzero_new:N` 4518 `{ \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }`
`\muskip_gzero_new:c` 4519 `\cs_new_protected:Npn \muskip_gzero_new:N #1`
4520 `{ \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }`
4521 `\cs_generate_variant:Nn \muskip_zero_new:N { c }`
4522 `\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` 4523 `\prg_new_eq_conditional:Nn \muskip_if_exist:N \cs_if_exist:N { TF , T , F , p }`
`\muskip_if_exist:N $\textcolor{red}{TF}$` 4524 `\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` 4525 `\cs_new_protected:Npn \muskip_set:Nn #1#2`
`\muskip_gset:Nn` 4526 `{ #1 ~ \etex_muexpr:D #2 \scan_stop: }`
`\muskip_gset:cn` 4527 `\cs_new_protected:Npn \muskip_gset:Nn { \tex_global:D \muskip_set:Nn }`
4528 `\cs_generate_variant:Nn \muskip_set:Nn { c }`
4529 `\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` 4530 `\cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }`
`\muskip_set_eq:Nc` 4531 `\cs_generate_variant:Nn \muskip_set_eq:NN { c }`
`\muskip_set_eq:cc` 4532 `\cs_generate_variant:Nn \muskip_set_eq:NN { Nc , cc }`
`\muskip_gset_eq:NN` 4533 `\cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }`
`\muskip_gset_eq:cN` 4534 `\cs_generate_variant:Nn \muskip_gset_eq:NN { c }`
`\muskip_gset_eq:Nc` 4535 `\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` 4536 `\cs_new_protected:Npn \muskip_add:Nn #1#2`
`\muskip_gadd:Nn` 4537 `{ \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }`
`\muskip_gadd:cn` 4538 `\cs_new_protected:Npn \muskip_gadd:Nn { \tex_global:D \muskip_add:Nn }`
`\muskip_sub:Nn` 4539 `\cs_generate_variant:Nn \muskip_add:Nn { c }`
`\muskip_sub:cn` 4540 `\cs_generate_variant:Nn \muskip_gadd:Nn { c }`
`\muskip_gsub:Nn` 4541 `\cs_new_protected:Npn \muskip_sub:Nn #1#2`
`\muskip_gsub:cn` 4542 `{ \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }`
4543 `\cs_new_protected:Npn \muskip_gsub:Nn { \tex_global:D \muskip_sub:Nn }`
4544 `\cs_generate_variant:Nn \muskip_sub:Nn { c }`
4545 `\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.
4546 `\cs_new:Npn \muskip_eval:n #1`
4547 `{ \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` 4548 `\cs_new_eq:NN \muskip_use:N \tex_the:D`
4549 `\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

```

4550 \cs_new_eq:NN \muskip_show:N \__kernel_register_show:N
4551 \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 T_EX 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.

```

4552 \cs_new_protected:Npn \muskip_show:n #1
4553 { \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

```

4554 \muskip_const:Nn \c_zero_muskip { 0 mu }
4555 \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

```

4556 \muskip_new:N \l_tmpa_muskip
4557 \muskip_new:N \l_tmpb_muskip
4558 \muskip_new:N \g_tmpa_muskip
4559 \muskip_new:N \g_tmpb_muskip

```

(End definition for \l_tmpa_muskip and \l_tmpb_muskip. These functions are documented on page 88.)

9.25 Deprecated functions

Deprecated on 2012-05-10, for removal by 2012-08-31.

\skip_if_infinite_glue_p:n Reverse of `\skip_if_finite:nTF`.
\skip_if_infinite_glue:nTF

```

4560 \*deprecated
4561 \prg_new_conditional:Npnn \skip_if_infinite_glue:n #1 { p , T , F , TF }
4562 { \skip_if_finite:nTF {#1} \prg_return_false: \prg_return_true: }
4563 \*deprecated

```

(End definition for \skip_if_infinite_glue:n. These functions are documented on page ??.)
 Deprecated 2012-06-03 for removal after 2012-12-31.

\prg_case_dim:nnn Moved here, was in `l3prg` but load order means we define it here now.

```

4564 \*deprecated
4565 \cs_new_eq:NN \prg_case_dim:nnn \dim_case:nnn
4566 \*deprecated

```

(End definition for \prg_case_dim:nnn. This function is documented on page ??.)

```

\dim_eval:w
\dim_eval_end:
4567 \*deprecated
4568 \cs_new_eq:NN \dim_eval:w \__dim_eval:w
4569 \cs_new_eq:NN \dim_eval_end: \__dim_eval_end:
4570 \*deprecated
(End definition for \dim_eval:w and \dim_eval_end:. These functions are documented on page ??.)

\dim_set_max:Nn
\dim_set_max:cn
\dim_set_min:Nn
\dim_set_min:cn
\dim_gset_max:Nn
\dim_gset_max:cn
\dim_gset_min:Nn
\dim_gset_min:cn
\__dim_set_max:NNNn
4571 \*deprecated
4572 \cs_new_protected_nopar:Npn \dim_set_max:Nn
4573 { \__dim_set_max:NNNn < \dim_set:Nn }
4574 \cs_new_protected_nopar:Npn \dim_gset_max:Nn
4575 { \__dim_set_max:NNNn < \dim_gset:Nn }
4576 \cs_new_protected_nopar:Npn \dim_set_min:Nn
4577 { \__dim_set_max:NNNn > \dim_set:Nn }
4578 \cs_new_protected_nopar:Npn \dim_gset_min:Nn
4579 { \__dim_set_max:NNNn > \dim_gset:Nn }
4580 \cs_new_protected:Npn \__dim_set_max:NNNn #1#2#3#4
4581 { \dim_compare:nNnT {#3} #1 {#4} { #2 #3 {#4} } }
4582 \cs_generate_variant:Nn \dim_set_max:Nn { c }
4583 \cs_generate_variant:Nn \dim_gset_max:Nn { c }
4584 \cs_generate_variant:Nn \dim_set_min:Nn { c }
4585 \cs_generate_variant:Nn \dim_gset_min:Nn { c }
4586 \*deprecated
(End definition for \dim_set_max:Nn and \dim_set_max:cn. These functions are documented on page ??.)

4587 \*initex | package)

```

10 l3tl implementation

```

4588 \*initex | package)
4589 \@@=tl
4590 \*package)
4591 \ProvidesExplPackage
4592 {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
4593 \__expl_package_check:
4594 \*package)

```

A token list variable is a \TeX macro that holds tokens. By using the $\varepsilon\text{-TeX}$ primitive \unexpanded inside a \TeX \edef it is possible to store any tokens, including $\#$, in this way.

10.1 Functions

\tl_new:N Creating new token list variables is a case of checking for an existing definition and doing the definition.

```

\tl_new:c
4595 \cs_new_protected:Npn \tl_new:N #1
4596 {

```

```

4597     \_chk_if_free_cs:N #1
4598     \cs_gset_eq:NN #1 \c_empty_tl
4599   }
4600 \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 4601 \cs_new_protected:Npn \tl_const:Nn #1#2
\tl_const:cn 4602 {
\tl_const:cx 4603   \_chk_if_free_cs:N #1
4604   \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
4605 }
4606 \cs_new_protected:Npn \tl_const:Nx #1#2
4607 {
4608   \_chk_if_free_cs:N #1
4609   \cs_gset_nopar:Npx #1 {#2}
4610 }
4611 \cs_generate_variant:Nn \tl_const:Nn { c }
4612 \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 4613 \cs_new_protected:Npn \tl_clear:N #1
\tl_gclear:N 4614 { \tl_set_eq:NN #1 \c_empty_tl }
\tl_gclear:c 4615 \cs_new_protected:Npn \tl_gclear:N #1
4616 { \tl_gset_eq:NN #1 \c_empty_tl }
4617 \cs_generate_variant:Nn \tl_clear:N { c }
4618 \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 4619 \cs_new_protected:Npn \tl_clear_new:N #1
\tl_gclear_new:c 4620 { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
4621 \cs_new_protected:Npn \tl_gclear_new:N #1
4622 { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
4623 \cs_generate_variant:Nn \tl_clear_new:N { c }
4624 \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 4625 \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN
\tl_set_eq:cN 4626 \cs_new_eq:NN \tl_set_eq:cN \cs_set_eq:cN
\tl_set_eq:cc 4627 \cs_new_eq:NN \tl_set_eq:Nc \cs_set_eq:Nc
\tl_gset_eq:NN 4628 \cs_new_eq:NN \tl_set_eq:cc \cs_set_eq:cc
\tl_gset_eq:Nc 4629 \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
\tl_gset_eq:cN 4630 \cs_new_eq:NN \tl_gset_eq:cN \cs_gset_eq:cN
\tl_gset_eq:cc

```

```

4631 \cs_new_eq:NN \tl_gset_eq:Nc \cs_gset_eq:Nc
4632 \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 4633 \cs_new_protected:Npn \tl_concat:NNN #1#2#3
\tl_gconcat:NNN 4634 { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
\tl_gconcat:ccc 4635 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
4636 { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
4637 \cs_generate_variant:Nn \tl_concat:NNN { ccc }
4638 \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 4639 \prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
\tl_if_exist:NTF 4640 \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`.

```

4641 \tl_const:Nn \c_empty_tl { }

```

(End definition for `\c_empty_tl`. This variable is documented on page 102.)

`\c_job_name_tl` Inherited from the L^AT_EX3 name for the primitive: this needs to actually contain the text of the job name rather than the name of the primitive, of course. Lua_TE_X does not quote file names containing spaces, whereas pdf_TE_X and X_T_TE_X do. So there may be a correction to make in the Lua_TE_X case.

```

4642 <*initex>
4643 \luatex_if_engine:T
4644 {
4645   \tex_everyjob:D \exp_after:wN
4646   {
4647     \tex_the:D \tex_everyjob:D
4648     \lua_now_x:n
4649     { dofile ( assert ( kpse.find_file ("lualatexquotejobname.lua" ) ) ) }
4650   }
4651 }
4652 \tex_everyjob:D \exp_after:wN
4653 {
4654   \tex_the:D \tex_everyjob:D
4655   \tl_const:Nx \c_job_name_tl { \tex_jobname:D }
4656 }
4657 </initex>
4658 <*package>
4659 \tl_const:Nx \c_job_name_tl { \tex_jobname:D }
4660 </package>

```

(End definition for `\c_job_name_tl`. This variable is documented on page 102.)

`\c_space_tl` A space as a token list (as opposed to as a character).

```
4661 \tl_const:Nn \c_space_tl { ~ }
```

(End definition for `\c_space_tl`. This variable is documented on page 102.)

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.

```
4662 \cs_new_protected:Npn \tl_set:Nn #1#2
4663 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
4664 \cs_new_protected:Npn \tl_set:No #1#2
4665 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
4666 \cs_new_protected:Npn \tl_set:Nx #1#2
4667 { \cs_set_nopar:Npx #1 {#2} }
4668 \cs_new_protected:Npn \tl_gset:Nn #1#2
4669 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
4670 \cs_new_protected:Npn \tl_gset:No #1#2
4671 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }
4672 \cs_new_protected:Npn \tl_gset:Nx #1#2
```

```
4673 { \cs_gset_nopar:Npx #1 {#2} }
4674 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
4675 \cs_generate_variant:Nn \tl_set:Nx { c }
4676 \cs_generate_variant:Nn \tl_set:Nn { c , co , cV , cv , cf }
4677 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
4678 \cs_generate_variant:Nn \tl_gset:Nx { c }
4679 \cs_generate_variant:Nn \tl_gset:Nn { c , co , cV , cv , cf }
\tl_gset:cv
```

(End definition for `\tl_set:Nn` and others. These functions are documented on page ??.)

```
\tl_gset:co
\tl_put_left:Nn
\tl_gset:cf
\tl_put_left:Nv
\tl_gset:cx
\tl_put_left:No
\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
\tl_gput_left:cV
\tl_gput_left:co
\tl_gput_left:cx
```

```
4680 \cs_new_protected:Npn \tl_put_left:Nn #1#2
4681 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
4682 \cs_new_protected:Npn \tl_put_left:Nv #1#2
4683 { \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4684 \cs_new_protected:Npn \tl_put_left:No #1#2
4685 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4686 \cs_new_protected:Npn \tl_put_left:Nx #1#2
4687 { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
4688 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
4689 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
4690 \cs_new_protected:Npn \tl_gput_left:Nv #1#2
4691 { \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4692 \cs_new_protected:Npn \tl_gput_left:No #1#2
4693 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4694 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
4695 { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
4696 \cs_generate_variant:Nn \tl_put_left:Nn { c }
4697 \cs_generate_variant:Nn \tl_put_left:Nv { c }
```

Adding to the left is done directly to gain a little performance.

```

4698 \cs_generate_variant:Nn \tl_put_left:No { c }
4699 \cs_generate_variant:Nn \tl_put_left:Nx { c }
4700 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
4701 \cs_generate_variant:Nn \tl_gput_left:NV { c }
4702 \cs_generate_variant:Nn \tl_gput_left:No { c }
4703 \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 4704 \cs_new_protected:Npn \tl_put_right:Nn #1#2
\tl_put_right:No 4705 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\tl_put_right:Nx 4706 \cs_new_protected:Npn \tl_put_right:NV #1#2
\tl_put_right:cn 4707 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\tl_put_right:cV 4708 \cs_new_protected:Npn \tl_put_right:No #1#2
\tl_put_right:co 4709 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\tl_put_right:cx 4710 \cs_new_protected:Npn \tl_put_right:Nx #1#2
\tl_gput_right:Nn 4711 { \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 } }
\tl_gput_right:NV 4712 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
\tl_gput_right:No 4713 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\tl_gput_right:Nx 4714 \cs_new_protected:Npn \tl_gput_right:NV #1#2
\tl_gput_right:cn 4715 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\tl_gput_right:cV 4716 \cs_new_protected:Npn \tl_gput_right:No #1#2
\tl_gput_right:co 4717 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\tl_gput_right:cx 4718 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
4719 { \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
4720 \cs_generate_variant:Nn \tl_put_right:Nn { c }
4721 \cs_generate_variant:Nn \tl_put_right:NV { c }
4722 \cs_generate_variant:Nn \tl_put_right:No { c }
4723 \cs_generate_variant:Nn \tl_put_right:Nx { c }
4724 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
4725 \cs_generate_variant:Nn \tl_gput_right:NV { c }
4726 \cs_generate_variant:Nn \tl_gput_right:No { c }
4727 \cs_generate_variant:Nn \tl_gput_right:Nx { c }

```

(End definition for `\tl_put_right:Nn` and others. These functions are documented on page ??.)

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.

```

4728 \group_begin:
4729 \tex_lccode:D '\A = '\@ \scan_stop:
4730 \tex_lccode:D '\B = '\@ \scan_stop:
4731 \tex_catcode:D '\A = 8 \scan_stop:
4732 \tex_catcode:D '\B = 3 \scan_stop:
4733 \tex_lowercase:D
4734 {
4735 \group_end:
4736 \tl_const:Nn \c__tl_rescan_marker_tl { A B }
4737 }

```

(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”.

```

4738 \cs_new_protected_nopar:Npn \tl_set_rescan:Nnn
4739 { \__tl_set_rescan:NNnn \tl_set:Nn }
4740 \cs_new_protected_nopar:Npn \tl_gset_rescan:Nnn
4741 { \__tl_set_rescan:NNnn \tl_gset:Nn }
4742 \cs_new_protected_nopar:Npn \tl_rescan:nn
4743 { \__tl_set_rescan:NNnn \prg_do_nothing: \use:n }
4744 \cs_new_protected:Npn \__tl_set_rescan:NNnn #1#2#3#4
4745 {
4746   \group_begin:
4747   \exp_args:No \etex_everyeof:D { \c__tl_rescan_marker_tl \exp_not:N }
4748   \tex_endlinechar:D \c_minus_one
4749   \tex_newlinechar:D \c_minus_one
4750   #3
4751   \use:x
4752   {
4753     \group_end:
4754     #1 \exp_not:N #2
4755     {
4756       \exp_after:wN \__tl_rescan:w
4757       \exp_after:wN \prg_do_nothing:
4758       \etex_scantokens:D {#4}
4759     }
4760   }
4761 }
4762 \use:x
4763 {
4764   \cs_new:Npn \exp_not:N \__tl_rescan:w ##1
4765   \c__tl_rescan_marker_tl
4766   { \exp_not:N \exp_not:o { ##1 } }
4767 }
4768 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno , Nnx }
4769 \cs_generate_variant:Nn \tl_set_rescan:Nnn { c , cno , cnx }
4770 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno , Nnx }
4771 \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`

```

4772 \cs_new_protected:Npn \tl_to_lowercase:n #1
4773 { \tex_lowercase:D {#1} }
4774 \cs_new_protected:Npn \tl_to_uppercase:n #1
4775 { \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` $\langle function \rangle$, $\langle \text{tl} \rangle \text{set:Nx}$, $\langle \text{tl var} \rangle$, $\langle search\ tokens \rangle$, $\langle replacement\ tokens \rangle$.
`\tl_greplace_all:Nnn`
`\tl_greplace_all:cnn`
`\tl_replace_once:Nnn`
`\tl_replace_once:cnn`
`\tl_greplace_once:Nnn`
`\tl_greplace_once:cnn`
`__tl_replace:NNNnn`
`__tl_replace:w`
`__tl_replace_all:`
`__tl_replace_once:`
`__tl_replace_once_end:w`

```

4776 \cs_new_protected_nopar:Npn \tl_replace_once:Nnn
4777 { \__tl_replace:NNNnn \__tl_replace_once: \tl_set:Nx }
4778 \cs_new_protected_nopar:Npn \tl_greplace_once:Nnn
4779 { \__tl_replace:NNNnn \__tl_replace_once: \tl_gset:Nx }
4780 \cs_new_protected_nopar:Npn \tl_replace_all:Nnn
4781 { \__tl_replace:NNNnn \__tl_replace_all: \tl_set:Nx }
4782 \cs_new_protected_nopar:Npn \tl_greplace_all:Nnn
4783 { \__tl_replace:NNNnn \__tl_replace_all: \tl_gset:Nx }
4784 \cs_generate_variant:Nn \tl_replace_once:Nnn { c }
4785 \cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
4786 \cs_generate_variant:Nn \tl_replace_all:Nnn { c }
4787 \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 $\langle search\ tokens \rangle$ by $\langle replacement\ tokens \rangle$. To avoid runaway arguments, we expand something like `__tl_tmp:w $\langle token\ list \rangle$ \q_mark $\langle search\ tokens \rangle$ \q_stop`, repeating until the end. How do we detect that we have reached the last occurrence of $\langle search\ tokens \rangle$? The last replacement is characterized by the fact that the argument of `__tl_tmp:w` contains `\q_mark`. In the code below, `__tl_replace:w` takes an argument delimited by `\q_mark`, and removes the following token. Before we reach the end, this gobbles `\q_mark \use_none_delimit_by_q_stop:w` which appear in the definition of `__tl_tmp:w`, and leaves the $\langle replacement\ tokens \rangle$, 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.

```

4788 \cs_new_protected:Npn \__tl_replace:NNNnn #1#2#3#4#5
4789 {
4790   \tl_if_empty:nTF {#4}
4791   {
4792     \__msg_kernel_error:nxx { kernel } { empty-search-pattern }
4793     { \tl_to_str:n {#5} }
4794   }
4795   {

```



```

4796 \group_align_safe_begin:
4797 \cs_set:Npx \__tl_tmp:w ##1##2 #4
4798 {
4799   ##2
4800   \exp_not:N \q_mark
4801   \exp_not:N \use_none_delimit_by_q_stop:w
4802   \exp_not:n { \exp_not:n {#5} }
4803   ##1
4804 }
4805 \group_align_safe_end:
4806 #2 #3
4807 {
4808   \exp_after:wN #1
4809   #3 \q_mark #4 \q_stop
4810 }
4811 }
4812 }
4813 \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.

```

4814 \cs_new_nopar:Npn \__tl_replace_all:
4815 {
4816   \exp_after:wN \__tl_replace:w
4817   \__tl_tmp:w \__tl_replace_all: \prg_do_nothing:
4818 }
4819 \cs_new_nopar:Npn \__tl_replace_once:
4820 {
4821   \exp_after:wN \__tl_replace:w
4822   \__tl_tmp:w { \__tl_replace_once_end:w \prg_do_nothing: } \prg_do_nothing:
4823 }
4824 \cs_new:Npn \__tl_replace_once_end:w #1 \q_mark #2 \q_stop
4825 { \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.

```

4826 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
4827 { \tl_replace_once:Nnn #1 {#2} { } }
4828 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
4829 { \tl_greplace_once:Nnn #1 {#2} { } }
4830 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
4831 \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 ??.)

```

\relax\tl_remove_all:Nn Removal is just a special case of replacement.
\relax\tl_remove_all:cn 4832 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
\relax\tl_gremove_all:Nn 4833 { \tl_replace_all:Nnn #1 {#2} { } }
\relax\tl_gremove_all:cn 4834 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
4835 { \tl_greplace_all:Nnn #1 {#2} { } }
4836 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
4837 \cs_generate_variant:Nn \tl_gremove_all:Nn { c }

```

10.7 Token list conditionals

`\tl_if_blank_p:n` TeX skips spaces when reading a non-delimited arguments. Thus, a *token list* is blank if and only if `\use_none:n <token list> ?` is empty. For performance reasons, we hard-code the emptiness test done in `\tl_if_empty:n(TF)`: convert to harmless characters with `\tl_to_str:n`, and then use `\if_meaning:w \q_nil ... \q_nil`. Note that converting to a string is done after reading the delimited argument for `\use_none:n`. The similar construction `\exp_after:wN \use_none:n \tl_to_str:n {<token list>} ?` would fail if the token list contains the control sequence `\`, while `\escapechar` is a space or is unprintable.

```

4838 \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
4839 { \__tl_if_empty_return:o { \use_none:n #1 ? } }
4840 \cs_generate_variant:Nn \tl_if_blank_p:n { V }
4841 \cs_generate_variant:Nn \tl_if_blank:nT { V }
4842 \cs_generate_variant:Nn \tl_if_blank:nF { V }
4843 \cs_generate_variant:Nn \tl_if_blank:nTF { V }
4844 \cs_generate_variant:Nn \tl_if_blank_p:n { o }
4845 \cs_generate_variant:Nn \tl_if_blank:nT { o }
4846 \cs_generate_variant:Nn \tl_if_blank:nF { o }
4847 \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).

```

\relax\tl_if_empty_p:c 4848 \prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
\relax\tl_if_empty:NTF 4849 {
\relax\tl_if_empty:cTF 4850   \if_meaning:w #1 \c_empty_tl
4851   \prg_return_true:
4852   \else:
4853   \prg_return_false:
4854   \fi:
4855 }
4856 \cs_generate_variant:Nn \tl_if_empty_p:N { c }
4857 \cs_generate_variant:Nn \tl_if_empty:N { c }
4858 \cs_generate_variant:Nn \tl_if_empty:N { c }
4859 \cs_generate_variant:Nn \tl_if_empty:N { c }

```

(End definition for \tl_if_empty:N and \tl_if_empty:c. These functions are documented on page ??.)

\tl_if_empty_p:n It would be tempting to just use \if_meaning:w \q_nil #1 \q_nil as a test since this works really well. However, it fails on a token list starting with \q_nil of course but more troubling is the case where argument is a complete conditional such as \if_true: a \else: b \fi: because then \if_true: is used by \if_meaning:w, the test turns out false, the \else: executes the false branch, the \fi: ends it and the \q_nil at the end starts executing... A safer route is to convert the entire token list into harmless characters first and then compare that. This way the test will even accept \q_nil as the first token.

```

4860 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
4861 {
4862   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil \tl_to_str:n {#1} \q_nil
4863   \prg_return_true:
4864   \else:
4865     \prg_return_false:
4866   \fi:
4867 }
4868 \cs_generate_variant:Nn \tl_if_empty_p:n { V }
4869 \cs_generate_variant:Nn \tl_if_empty:nTF { V }
4870 \cs_generate_variant:Nn \tl_if_empty:nT { V }
4871 \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 conditionals on token lists, which mostly reduce to testing if a given token list is empty after applying a simple function to it. The test for emptiness is based on \tl_if_empty:n(TF), but the expansion is hard-coded for efficiency, as this auxiliary function is used in many places. Note that this works because \tl_to_str:n expands tokens that follow until reading a catcode 1 (begin-group) token.

```

4872 \cs_new:Npn \__tl_if_empty_return:o #1
4873 {
4874   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
4875   \tl_to_str:n \exp_after:wN {#1} \q_nil
4876   \prg_return_true:
4877   \else:
4878     \prg_return_false:
4879   \fi:
4880 }
4881 \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
4882 { \__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.

```

\tl_if_eq_p:Nc 4883 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
\tl_if_eq_p:cN 4884 {
\tl_if_eq_p:cc 4885   \if_meaning:w #1 #2
\tl_if_eq:NNTF 4886   \prg_return_true:
\tl_if_eq:NcTF
\tl_if_eq:cNTF
\tl_if_eq:ccTF

```

```

4887 \else:
4888 \prg_return_false:
4889 \fi:
4890 }
4891 \cs_generate_variant:Nn \tl_if_eq_p:NN { Nc , c , cc }
4892 \cs_generate_variant:Nn \tl_if_eq:NNTF { Nc , c , cc }
4893 \cs_generate_variant:Nn \tl_if_eq:NNT { Nc , c , cc }
4894 \cs_generate_variant:Nn \tl_if_eq:NNF { Nc , c , cc }

```

(End definition for \tl_if_eq:NN and others. These functions are documented on page ??.)

\tl_if_eq:nnTF A simple store and compare routine.

```

\l__tl_internal_a_tl 4895 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
\l__tl_internal_b_tl 4896 {
4897 \group_begin:
4898 \tl_set:Nn \l__tl_internal_a_tl {#1}
4899 \tl_set:Nn \l__tl_internal_b_tl {#2}
4900 \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
4901 \group_end:
4902 \prg_return_true:
4903 \else:
4904 \group_end:
4905 \prg_return_false:
4906 \fi:
4907 }
4908 \tl_new:N \l__tl_internal_a_tl
4909 \tl_new:N \l__tl_internal_b_tl

```

(End definition for \tl_if_eq:nn. This function is documented on page ??.)

\tl_if_in:NnTF See \tl_if_in:nn(TF) for further comments. Here we simply expand the token list
\tl_if_in:cnTF variable and pass it to \tl_if_in:nn(TF).

```

4910 \cs_new_protected_nopar:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
4911 \cs_new_protected_nopar:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
4912 \cs_new_protected_nopar:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
4913 \cs_generate_variant:Nn \tl_if_in:NnT { c }
4914 \cs_generate_variant:Nn \tl_if_in:NnF { c }
4915 \cs_generate_variant:Nn \tl_if_in:NnTF { c }

```

(End definition for \tl_if_in:NnTF and \tl_if_in:cnTF. These functions are documented on page ??.)

\tl_if_in:nnTF Once more, the test relies on \tl_to_str:n for robustness. The function __tl_tmp:w
\tl_if_in:VnTF removes tokens until the first occurrence of #2. If this does not appear in #1, then the
\tl_if_in:onTF final #2 is removed, leaving an empty token list. Otherwise some tokens remain, and the
\tl_if_in:noTF test is false. See \tl_if_empty:n(TF) for details on the emptiness test.

Special care is needed to treat correctly cases like \tl_if_in:nnTF {a state}{states}, where #1#2 contains #2 before the end. To cater for this case, we insert {}{} between the two token lists. This marker may not appear in #2 because of TeX limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the test work also for empty arguments.

```

4916 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }

```

```

4917 {
4918   \cs_set:Npn \__tl_tmp:w ##1 #2 { }
4919   \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
4920   { \prg_return_false: } { \prg_return_true: }
4921 }
4922 \cs_generate_variant:Nn \tl_if_in:nnT { V , o , no }
4923 \cs_generate_variant:Nn \tl_if_in:nnF { V , o , no }
4924 \cs_generate_variant:Nn \tl_if_in:nnTF { V , o , no }

```

(End definition for \tl_if_in:nnTF and others. These functions are documented on page ??.)

\tl_if_single_p:N Expand the token list and feed it to \tl_if_single:n.

\tl_if_single:N \overline{TF}

```

4925 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
4926 \cs_new:Npn \tl_if_single:NT { \exp_args:No \tl_if_single:nT }
4927 \cs_new:Npn \tl_if_single:NF { \exp_args:No \tl_if_single:nF }
4928 \cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }

```

(End definition for \tl_if_single:N. These functions are documented on page 95.)

\tl_if_single_p:n A token list has exactly one item if it is a single token or a single brace group, surrounded by optional explicit spaces. The naive version of this test would do \use_none:n #1, and test if the result is empty. However, this will fail when the token list is empty. Furthermore, it does not allow optional trailing spaces.

\tl_if_single:n \overline{TF}

```

4929 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
4930 { \__str_if_eq_x_return:nn { \exp_not:o { \use_none:nn #1 ?? } } {?} }

```

(End definition for \tl_if_single:n. These functions are documented on page 95.)

\tl_case:Nnn

\tl_case:cnn

__tl_case:Nw

__tl_case_end:nw

```

4931 \cs_new:Npn \tl_case:Nnn #1#2#3
4932 {
4933   \tex_romannumeral:D
4934   \__tl_case:Nw #1 #2 #1 {#3} \q_recursion_stop
4935 }
4936 \cs_new:Npn \__tl_case:Nw #1#2#3
4937 {
4938   \tl_if_eq:NNTF #1 #2
4939   { \__tl_case_end:nw {#3} }
4940   { \__tl_case:Nw #1 }
4941 }
4942 \cs_generate_variant:Nn \tl_case:Nnn { c }
4943 \cs_new_eq:NN \__tl_case_end:nw \__prg_case_end:nw

```

(End definition for \tl_case:Nnn and \tl_case:cnn. These functions are documented on page ??.)

10.8 Mapping to token lists

\tl_map_function:nN Expandable loop macro for token lists. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker will be read immediately and the loop terminated.

\tl_map_function:NN

\tl_map_function:cN

__tl_map_function:Nn

```

4944 \cs_new:Npn \tl_map_function:nN #1#2
4945 {

```

```

4946     \_tl_map_function:Nn #2 #1
4947     \q_recursion_tail
4948     \_prg_break_point:Nn \tl_map_break: { }
4949   }
4950   \cs_new_nopar:Npn \tl_map_function:NN
4951   { \exp_args:No \tl_map_function:nN }
4952   \cs_new:Npn \__tl_map_function:Nn #1#2
4953   {
4954     \_quark_if_recursion_tail_break:nN {#2} \tl_map_break:
4955     #1 {#2} \_tl_map_function:Nn #1
4956   }
4957   \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
\tl_map_inline:Nn \g__prg_map_int to make them nestable. We can also make use of __tl_map_-
\tl_map_inline:cn function:Nn from before.

```

4958   \cs_new_protected:Npn \tl_map_inline:nn #1#2
4959   {
4960     \int_gincr:N \g__prg_map_int
4961     \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
4962     \exp_args:Nc \_tl_map_function:Nn
4963     { __prg_map_ \int_use:N \g__prg_map_int :w }
4964     #1 \q_recursion_tail
4965     \_prg_break_point:Nn \tl_map_break: { \int_gdecr:N \g__prg_map_int }
4966   }
4967   \cs_new_protected:Npn \tl_map_inline:Nn
4968   { \exp_args:No \tl_map_inline:nn }
4969   \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:cn \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
__tl_map_variable:Nnn {

```

4970   {
4971     \_tl_map_variable:Nnn #2 {#3} #1
4972     \q_recursion_tail
4973     \_prg_break_point:Nn \tl_map_break: { }
4974   }
4975   \cs_new_protected_nopar:Npn \tl_map_variable:NNn
4976   { \exp_args:No \tl_map_variable:nNn }
4977   \cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
4978   {
4979     \tl_set:Nn #1 {#3}
4980     \_quark_if_recursion_tail_break:NN #1 \tl_map_break:
4981     \use:n {#2}
4982     \_tl_map_variable:Nnn #1 {#2}
4983   }
4984   \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`

```
4986 \cs_new_nopar:Npn \tl_map_break:
4987 { \__prg_map_break:Nn \tl_map_break: { } }
4988 \cs_new_nopar:Npn \tl_map_break:n
4989 { \__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.

```
4990 \cs_new_eq:NN \tl_to_str:n \etex_detokenize:D
```

(End definition for `\tl_to_str:n`. This function is documented on page 97.)

`\tl_to_str:N` These functions return the replacement text of a token list as a string.

`\tl_to_str:c`

```
4991 \cs_new:Npn \tl_to_str:N #1 { \etex_detokenize:D \exp_after:wN {#1} }
4992 \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.

```
4993 \cs_new:Npn \tl_use:N #1
4994 {
4995   \tl_if_exist:NTF #1 {#1}
4996   { \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#1} }
4997 }
4998 \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 to ensure it works on an empty list.

```
\tl_count:N 4999 \cs_new:Npn \tl_count:n #1
\tl_count:c 5000 {
\__tl_count:n 5001   \int_eval:n
5002     { 0 \tl_map_function:nN {#1} \__tl_count:n }
5003   }
5004 \cs_new:Npn \tl_count:N #1
5005 {
5006   \int_eval:n
5007   { 0 \tl_map_function:NN #1 \__tl_count:n }
5008 }
5009 \cs_new:Npn \__tl_count:n #1 { + \c_one }
5010 \cs_generate_variant:Nn \tl_count:n { V , o }
5011 \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`
`__tl_reverse_items:wn`

```

5012 \cs_new:Npn \tl_reverse_items:n #1
5013 {
5014   \__tl_reverse_items:nwNwn #1 ?
5015   \q_mark \__tl_reverse_items:nwNwn
5016   \q_mark \__tl_reverse_items:wn
5017   \q_stop { }
5018 }
5019 \cs_new:Npn \__tl_reverse_items:nwNwn #1 #2 \q_mark #3 #4 \q_stop #5
5020 {
5021   #3 #2
5022   \q_mark \__tl_reverse_items:nwNwn
5023   \q_mark \__tl_reverse_items:wn
5024   \q_stop { {#1} #5 }
5025 }
5026 \cs_new:Npn \__tl_reverse_items:wn #1 \q_stop #2
5027 { \exp_not:o { \use_none:n #2 } }

```

(End definition for `\tl_reverse_items:n`. This function is documented on page 98.)

`\tl_trim_spaces:n`

Trimming spaces from around the input is deferred to an internal function whose first argument is the token list to trim, augmented by an initial `\q_mark`, and whose second argument is a *continuation*, which will receive as a braced argument `\use_none:n \q_mark` *(trimmed token list)*. In the case at hand, we take `\exp_not:o` as our continuation, so that space trimming will behave correctly within an x-type expansion.

`\tl_trim_spaces:N`

`\tl_trim_spaces:c`

`\tl_gtrim_spaces:N`

`\tl_gtrim_spaces:c`

```

5028 \cs_new:Npn \tl_trim_spaces:n #1
5029 { \__tl_trim_spaces:nn { \q_mark #1 } \exp_not:o }
5030 \cs_new_protected:Npn \tl_trim_spaces:N #1
5031 { \tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
5032 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
5033 { \tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
5034 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
5035 \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:n`

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*.

`__tl_trim_spaces_auxi:w`
`__tl_trim_spaces_auxii:w`
`__tl_trim_spaces_auxiii:w`
`__tl_trim_spaces_auxiv:w`

```

5036 \cs_set:Npn \__tl_tmp:w #1

```



```

5037 {
5038   \cs_new:Npn \__tl_trim_spaces:nn ##1
5039   {
5040     \__tl_trim_spaces_auxi:w
5041     ##1
5042     \q_nil
5043     \q_mark #1 { }
5044     \q_mark \__tl_trim_spaces_auxii:w
5045     \__tl_trim_spaces_auxiii:w
5046     #1 \q_nil
5047     \__tl_trim_spaces_auxiv:w
5048     \q_stop
5049   }
5050   \cs_new:Npn \__tl_trim_spaces_auxi:w ##1 \q_mark #1 ##2 \q_mark ##3
5051   {
5052     ##3
5053     \__tl_trim_spaces_auxi:w
5054     \q_mark
5055     ##2
5056     \q_mark #1 {##1}
5057   }
5058   \cs_new:Npn \__tl_trim_spaces_auxii:w
5059   \__tl_trim_spaces_auxi:w \q_mark \q_mark ##1
5060   {
5061     \__tl_trim_spaces_auxiii:w
5062     ##1
5063   }
5064   \cs_new:Npn \__tl_trim_spaces_auxiii:w ##1 #1 \q_nil ##2
5065   {
5066     ##2
5067     ##1 \q_nil
5068     \__tl_trim_spaces_auxiii:w
5069   }
5070   \cs_new:Npn \__tl_trim_spaces_auxiv:w ##1 \q_nil ##2 \q_stop ##3
5071   { ##3 { \use_none:n ##1 } }
5072 }
5073 \__tl_tmp:w { ~ }

```

(End definition for __tl_trim_spaces:nn. This function is documented on page 103.)

10.11 Token by token changes

\q___tl_act_mark The \tl_act functions may be applied to any token list. Hence, we use two private quarks, to allow any token, even quarks, in the token list. Only \q___tl_act_mark and \q___tl_act_stop may not appear in the token lists manipulated by __tl_act:NNNnn functions. The quarks are effectively defined in l3quark.

(End definition for \q___tl_act_mark and \q___tl_act_stop. These variables are documented on page ??.)

<pre> __tl_act:NNNnn __tl_act_output:n __tl_act_reverse_output:n __tl_act_loop:w __tl_act_normal:NwnNNN __tl_act_group:nwnNNN __tl_act_space:wwnNNN __tl_act_end:w </pre>	<p>To help control the expansion, __tl_act:NNNnn should always be proceeded by \romannumeral and ends by producing \c_zero once the result has been obtained. Then</p>
---	---

loop over tokens, groups, and spaces in #5. The marker `\q___tl_act_mark` is used both to avoid losing outer braces and to detect the end of the token list more easily. The result is stored as an argument for the dummy function `__tl_act_result:n`.

```

5074 \cs_new:Npn \__tl_act:NNNnn #1#2#3#4#5
5075 {
5076   \group_align_safe_begin:
5077   \__tl_act_loop:w #5 \q___tl_act_mark \q___tl_act_stop
5078   {#4} #1 #2 #3
5079   \__tl_act_result:n { }
5080 }

```

In the loop, we check how the token list begins and act accordingly. In the “normal” case, we may have reached `\q___tl_act_mark`, the end of the list. Then leave `\c_zero` and the result in the input stream, to terminate the expansion of `\romannumeral`. Otherwise, apply the relevant function to the “arguments”, #3 and to the head of the token list. Then repeat the loop. The scheme is the same if the token list starts with a group or with a space. Some extra work is needed to make `__tl_act_space:wnnNNN` gobble the space.

```

5081 \cs_new:Npn \__tl_act_loop:w #1 \q___tl_act_stop
5082 {
5083   \tl_if_head_is_N_type:nTF {#1}
5084   { \__tl_act_normal:NwnNNN }
5085   {
5086     \tl_if_head_is_group:nTF {#1}
5087     { \__tl_act_group:nwnNNN }
5088     { \__tl_act_space:wnnNNN }
5089   }
5090   #1 \q___tl_act_stop
5091 }
5092 \cs_new:Npn \__tl_act_normal:NwnNNN #1 #2 \q___tl_act_stop #3#4
5093 {
5094   \if_meaning:w \q___tl_act_mark #1
5095   \exp_after:wN \__tl_act_end:wn
5096   \fi:
5097   #4 {#3} #1
5098   \__tl_act_loop:w #2 \q___tl_act_stop
5099   {#3} #4
5100 }
5101 \cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
5102 { \group_align_safe_end: \c_zero #2 }
5103 \cs_new:Npn \__tl_act_group:nwnNNN #1 #2 \q___tl_act_stop #3#4#5
5104 {
5105   #5 {#3} {#1}
5106   \__tl_act_loop:w #2 \q___tl_act_stop
5107   {#3} #4 #5
5108 }
5109 \exp_last_unbraced:NNo
5110 \cs_new:Npn \__tl_act_space:wnnNNN \c_space_tl #1 \q___tl_act_stop #2#3#4#5
5111 {

```

```

5112     #5 {#2}
5113     \__tl_act_loop:w #1 \q__tl_act_stop
5114     {#2} #3 #4 #5
5115 }

```

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.

```

5116 \cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
5117 { #2 \__tl_act_result:n { #3 #1 } }
5118 \cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
5119 { #2 \__tl_act_result:n { #1 #3 } }

```

(End definition for `__tl_act:NNNnn`. This function is documented on page ??.)

```

\tl_reverse:n
\tl_reverse:o
\tl_reverse:V
\__tl_reverse_normal:nN
\__tl_reverse_group_preserve:nn
\__tl_reverse_space:n

```

The goal here is to reverse without losing spaces nor braces. This is done using the general internal function `__tl_act:NNNnn`. Spaces and “normal” tokens are output on the left of the current output. Grouped tokens are output to the left but without any reversal within the group. All of the internal functions here drop one argument: this is needed by `__tl_act:NNNnn` when changing case (to record which direction the change is in), but not when reversing the tokens.

```

5120 \cs_new:Npn \tl_reverse:n #1
5121 {
5122   \etex_unexpanded:D \exp_after:wN
5123   {
5124     \tex_romannumeral:D
5125     \__tl_act:NNNnn
5126     \__tl_reverse_normal:nN
5127     \__tl_reverse_group_preserve:nn
5128     \__tl_reverse_space:n
5129     { }
5130     {#1}
5131   }
5132 }
5133 \cs_generate_variant:Nn \tl_reverse:n { o , V }
5134 \cs_new:Npn \__tl_reverse_normal:nN #1#2
5135 { \__tl_act_reverse_output:n {#2} }
5136 \cs_new:Npn \__tl_reverse_group_preserve:nn #1#2
5137 { \__tl_act_reverse_output:n { {#2} } }
5138 \cs_new:Npn \__tl_reverse_space:n #1
5139 { \__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
\tl_reverse:c
\tl_greverse:N
\tl_greverse:c

```

This reverses the list, leaving `\exp_stop_f:` in front, which stops the f-expansion.

```

5140 \cs_new_protected:Npn \tl_reverse:N #1
5141 { \tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
5142 \cs_new_protected:Npn \tl_greverse:N #1
5143 { \tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
5144 \cs_generate_variant:Nn \tl_reverse:N { c }
5145 \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:nw grabbing is exactly the marker, or there is anything else to deal with. Is there is, there
\tl_head:w is a loop. If not, tidy up and leave the item in the output stream. More detail in
\tl_tail:N http://tex.stackexchange.com/a/70168.
\tl_tail:n 5146 \cs_new:Npn \tl_head:n #1
\tl_tail:V 5147 {
\tl_tail:v 5148   \etex_unexpanded:D
\tl_tail:f 5149   \if_false: { \fi: \__tl_head_auxi:nw #1 { } \q_stop }
5150 }
5151 \cs_new:Npn \__tl_head_auxi:nw #1#2 \q_stop
5152 { \exp_after:wN \__tl_head_auxii:nw \exp_after:wN { \if_false: } \fi: {#1} }
5153 \cs_new:Npn \__tl_head_auxii:nw #1
5154 {
5155   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
5156   \tl_to_str:n \exp_after:wN { \use_none:n #1 } \q_nil
5157   \exp_after:wN \use_i:nn
5158   \else:
5159     \exp_after:wN \use_ii:nn
5160   \fi:
5161   {#1}
5162   { \if_false: { \fi: \__tl_head_auxi:nw #1 } }
5163 }
5164 \cs_generate_variant:Nn \tl_head:n { V , v , f }
5165 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
5166 \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.

```

5167 \cs_new:Npn \tl_tail:n #1
5168 {
5169   \etex_unexpanded:D
5170   \tl_if_blank:nTF {#1}

```

```

5171         { { } }
5172         { \exp_after:wN { \use_none:n #1 } }
5173     }
5174 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
5175 \cs_new_nopar:Npn \tl_tail:N { \exp_args:No \tl_tail:n }

```

(End definition for `\tl_head:N` and others. These functions are documented on page ??.)

`\str_head:n` After `\tl_to_str:n`, we have a list of character tokens, all with category code 12, except the space, which has category code 10. Directly using `\tl_head:w` would thus lose leading spaces. Instead, we take an argument delimited by an explicit space, and then only use `\tl_head:w`. If the string started with a space, then the argument of `__str_head:w` is empty, and the function correctly returns a space character. Otherwise, it returns the first token of `#1`, which is the first token of the string. If the string is empty, we return an empty result.

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`.

```

5176 \cs_new:Npn \str_head:n #1
5177 {
5178     \exp_after:wN \__str_head:w
5179     \tl_to_str:n {#1}
5180     { { } } ~ \q_stop
5181 }
5182 \cs_new:Npn \__str_head:w #1 ~ %
5183 { \tl_head:w #1 { ~ } }
5184 \cs_new:Npn \str_tail:n #1
5185 {
5186     \exp_after:wN \__str_tail:w
5187     \reverse_if:N \if_charcode:w
5188         \scan_stop: \tl_to_str:n {#1} X X \q_stop
5189 }
5190 \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 100.)

`\tl_if_head_eq_meaning_p:nN` Accessing the first token of a token list is tricky in three cases: when it has category code 1 (begin-group token), when it is an explicit space, with category code 10 and character code 32, or when the token list is empty (obviously).

`\tl_if_head_eq_meaning:nNTF` Forgetting temporarily about this issue we would use the following test in `\tl_if_head_eq_charcode:nN`. Here, `\tl_head:w` yields the first token of the token list, then passed to `\exp_not:N`.

```

\tl_if_head_eq_charcode:nNTF
\tl_if_head_eq_charcode:p:nN
\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
\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.

```

5191 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
5192 {
5193   \if_charcode:w
5194     \exp_not:N #2
5195     \tl_if_head_is_N_type:nTF { #1 ? }
5196     {
5197       \exp_after:wN \exp_not:N
5198       \tl_head:w #1 { ? \use_none:nn } \q_stop
5199     }
5200     { \str_head:n {#1} }
5201     \prg_return_true:
5202   \else:
5203     \prg_return_false:
5204   \fi:
5205 }
5206 \cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN { f }
5207 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF { f }
5208 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT { f }
5209 \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.

```

5210 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
5211 {
5212   \if_catcode:w
5213     \exp_not:N #2
5214     \tl_if_head_is_N_type:nTF { #1 ? }
5215     {
5216       \exp_after:wN \exp_not:N
5217       \tl_head:w #1 { ? \use_none:nn } \q_stop
5218     }
5219     {
5220       \tl_if_head_is_group:nTF {#1}
5221       { \c_group_begin_token }
5222       { \c_space_token }
5223     }
5224     \prg_return_true:
5225   \else:

```

```

5226     \prg_return_false:
5227     \fi:
5228 }

```

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.

```

5229 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
5230 {
5231     \tl_if_head_is_N_type:nTF { #1 ? }
5232     { \__tl_if_head_eq_meaning_normal:nN }
5233     { \__tl_if_head_eq_meaning_special:nN }
5234     {#1} #2
5235 }
5236 \cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
5237 {
5238     \exp_after:wN \if_meaning:w
5239     \tl_head:w #1 { ?? \use_none:nnn } \q_stop #2
5240     \prg_return_true:
5241     \else:
5242     \prg_return_false:
5243     \fi:
5244 }
5245 \cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
5246 {
5247     \if_charcode:w \str_head:n {#1} \exp_not:N #2
5248     \exp_after:wN \use:n
5249     \else:
5250     \prg_return_false:
5251     \exp_after:wN \use_none:n
5252     \fi:
5253     {
5254         \if_catcode:w \exp_not:N #2
5255         \tl_if_head_is_group:nTF {#1}
5256         { \c_group_begin_token }
5257         { \c_space_token }
5258         \prg_return_true:
5259         \else:
5260         \prg_return_false:
5261         \fi:
5262     }
5263 }

```

(End definition for `\tl_if_head_eq_meaning:nN`. These functions are documented on page 100.)

`\tl_if_head_is_N_type_p:n` The first token of a token list can be either an N-type argument, a begin-group token
`\tl_if_head_is_N_type:nTF` (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”.

```

5264 \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
5265 {
5266   \__str_if_eq_x_return:nn
5267   { \exp_not:o { \use:n #1 { } } }
5268   { \exp_not:n { #1 { } } }
5269 }

```

(End definition for `\tl_if_head_is_N_type:n`. These functions are documented on page 101.)

`\tl_if_head_is_group_p:n` Pass the first token of `#1` through `\token_to_str:N`, then check for the brace balance.
`\tl_if_head_is_group:nTF` The extra ? caters for an empty argument.⁵

```

5270 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
5271 {
5272   \if_catcode:w *
5273   \exp_after:wN \use_none:n
5274   \exp_after:wN {
5275     \exp_after:wN {
5276       \token_to_str:N #1 ?
5277     }
5278   }
5279   *
5280   \prg_return_false:
5281   \else:
5282     \prg_return_true:
5283   \fi:
5284 }

```

(End definition for `\tl_if_head_is_group:n`. These functions are documented on page 101.)

`\tl_if_head_is_space_p:n` If the first token of the token list is an explicit space, i.e., a character token with character code 32 and category code 10, then this test will be `true`. It is `false` if the token list is empty, if the first token is an implicit space token, such as `\c_space_token`, or any token other than an explicit space. The slightly convoluted approach with `\romannumeral` ensures that each expansion step gives a balanced token list.
`\tl_if_head_is_space:nTF`
`__tl_if_head_is_space:w`

```

5285 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
5286 {
5287   \tex_romannumeral:D \if_false: { \fi:
5288     \__tl_if_head_is_space:w ? #1 ? ~ }
5289 }
5290 \cs_new:Npn \__tl_if_head_is_space:w #1 ~
5291 {
5292   \tl_if_empty:oTF { \use_none:n #1 }
5293   { \exp_after:wN \c_zero \exp_after:wN \prg_return_true: }
5294   { \exp_after:wN \c_zero \exp_after:wN \prg_return_false: }

```

⁵Bruno: this could be made faster, but we don’t: if we hope to ever have an e-type argument, we need all brace “tricks” to happen in one step of expansion, keeping the token list brace balanced at all times.


```

5295     \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
5296   }

```

(End definition for `\tl_if_head_is_space:n`. These functions are documented on page 101.)

10.13 Viewing token lists

`\tl_show:N` Showing token list variables is done after checking that the variable is defined (see `__kernel_register_show:N`).

```

5297 \cs_new_protected:Npn \tl_show:N #1
5298 {
5299   \tl_if_exist:NTF #1
5300   { \cs_show:N #1 }
5301   {
5302     \__msg_kernel_error:nxx { kernel } { variable-not-defined }
5303     { \token_to_str:N #1 }
5304   }
5305 }
5306 \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.

```

5307 \cs_new_protected:Npn \tl_show:n #1
5308 { \__msg_show_variable:n { > ~ \tl_to_str:n {#1} } }

```

(End definition for `\tl_show:n`. This function is documented on page 102.)

10.14 Scratch token lists

`\g_tmpa_tl` Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

```

5309 \tl_new:N \g_tmpa_tl
5310 \tl_new:N \g_tmpb_tl

```

(End definition for `\g_tmpa_tl` and `\g_tmpb_tl`. These variables are documented on page 102.)

`\l_tmpa_tl` These are local temporary token list variables. Be sure not to assume that the value you put into them will survive for long—see discussion above.

```

5311 \tl_new:N \l_tmpa_tl
5312 \tl_new:N \l_tmpb_tl

```

(End definition for `\l_tmpa_tl` and `\l_tmpb_tl`. These variables are documented on page 102.)

10.15 Deprecated functions

`\tl_new:Nn` Use either `\tl_const:Nn` or `\tl_new:N`.

```

\tl_new:cn      5313 <*deprecated>
\tl_new:Nx      5314 \cs_new_protected:Npn \tl_new:Nn #1#2
                  {
                    5315   \tl_new:N #1
                    5316   \tl_gset:Nn #1 {#2}
                  }
                  5317
                  5318 }
                  5319 \cs_generate_variant:Nn \tl_new:Nn { c }
                  5320 \cs_generate_variant:Nn \tl_new:Nn { Nx }
                  5321 </deprecated>

```

(End definition for `\tl_new:Nn`, `\tl_new:cn`, and `\tl_new:Nx`. These functions are documented on page ??.)

`\tl_gset:Nc` This was useful once, but nowadays does not make much sense.

```

\tl_set:Nc      5322 <*deprecated>
                  5323 \cs_new_protected_nopar:Npn \tl_gset:Nc
                  5324   { \tex_global:D \tl_set:Nc }
                  5325 \cs_new_protected:Npn \tl_set:Nc #1#2
                  5326   { \tl_set:No #1 { \cs:w #2 \cs_end: } }
                  5327 </deprecated>

```

(End definition for `\tl_gset:Nc`. This function is documented on page ??.)

`\tl_replace_in:Nnn` These are renamed.

```

\tl_replace_in:cn      5328 <*deprecated>
\tl_replace_in:Nnn     5329 \cs_new_eq:NN \tl_replace_in:Nnn \tl_replace_once:Nnn
\tl_greplace_in:Nnn    5330 \cs_new_eq:NN \tl_replace_in:cn \tl_replace_once:cn
\tl_greplace_in:cn     5331 \cs_new_eq:NN \tl_greplace_in:Nnn \tl_greplace_once:Nnn
\tl_replace_all_in:Nnn 5332 \cs_new_eq:NN \tl_greplace_in:cn \tl_greplace_once:cn
\tl_replace_all_in:cn  5333 \cs_new_eq:NN \tl_replace_all_in:Nnn \tl_replace_all:Nnn
\tl_greplace_all_in:Nnn 5334 \cs_new_eq:NN \tl_replace_all_in:cn \tl_replace_all:cn
\tl_greplace_all_in:cn 5335 \cs_new_eq:NN \tl_greplace_all_in:Nnn \tl_greplace_all:Nnn
\tl_greplace_all_in:cn 5336 \cs_new_eq:NN \tl_greplace_all_in:cn \tl_greplace_all:cn
\tl_greplace_all_in:cn 5337 </deprecated>

```

(End definition for `\tl_replace_in:Nnn` and `\tl_replace_in:cn`. These functions are documented on page ??.)

`\tl_remove_in:Nn` Also renamed.

```

\tl_remove_in:cn      5338 <*deprecated>
\tl_remove_in:Nn      5339 \cs_new_eq:NN \tl_remove_in:Nn \tl_remove_once:Nn
\tl_gremove_in:Nn     5340 \cs_new_eq:NN \tl_remove_in:cn \tl_remove_once:cn
\tl_gremove_in:cn     5341 \cs_new_eq:NN \tl_gremove_in:Nn \tl_gremove_once:Nn
\tl_remove_all_in:Nn  5342 \cs_new_eq:NN \tl_gremove_in:cn \tl_gremove_once:cn
\tl_remove_all_in:cn  5343 \cs_new_eq:NN \tl_remove_all_in:Nn \tl_remove_all:Nn
\tl_gremove_all_in:Nn 5344 \cs_new_eq:NN \tl_remove_all_in:cn \tl_remove_all:cn
\tl_gremove_all_in:cn 5345 \cs_new_eq:NN \tl_gremove_all_in:Nn \tl_gremove_all:Nn
\tl_gremove_all_in:cn 5346 \cs_new_eq:NN \tl_gremove_all_in:cn \tl_gremove_all:cn
\tl_gremove_all_in:cn 5347 </deprecated>

```

(End definition for `\tl_remove_in:Nn` and `\tl_remove_in:cn`. These functions are documented on page ??.)

`\tl_elt_count:n` Another renaming job.

```

\tl_elt_count:V 5348 \*deprecated
\tl_elt_count:o 5349 \cs_new_eq:NN \tl_elt_count:n \tl_count:n
\tl_elt_count:N 5350 \cs_new_eq:NN \tl_elt_count:V \tl_count:V
\tl_elt_count:c 5351 \cs_new_eq:NN \tl_elt_count:o \tl_count:o
                  5352 \cs_new_eq:NN \tl_elt_count:N \tl_count:N
                  5353 \cs_new_eq:NN \tl_elt_count:c \tl_count:c
                  5354 \</deprecated>

```

(End definition for `\tl_elt_count:n`, `\tl_elt_count:V`, and `\tl_elt_count:o`. These functions are documented on page ??.)

`\tl_head_i:n` Two renames, and a few that are rather too specialised.

```

\tl_head_i:w 5355 \*deprecated
\tl_head_iii:n 5356 \cs_new_eq:NN \tl_head_i:n \tl_head:n
\tl_head_iii:f 5357 \cs_new_eq:NN \tl_head_i:w \tl_head:w
\tl_head_iii:w 5358 \cs_new:Npn \tl_head_iii:n #1 { \tl_head_iii:w #1 \q_stop }
                  5359 \cs_generate_variant:Nn \tl_head_iii:n { f }
                  5360 \cs_new:Npn \tl_head_iii:w #1#2#3#4 \q_stop {#1#2#3}
                  5361 \</deprecated>

```

(End definition for `\tl_head_i:n`. This function is documented on page ??.)

Deprecated on 2012-05-13 for removal by 2012-08-31.

`\tl_length_tokens:n`

```

5362 \*deprecated
5363 \cs_new_eq:NN \tl_length_tokens:n \tl_count_tokens:n
5364 \</deprecated>

```

(End definition for `\tl_length_tokens:n`. This function is documented on page ??.)

Deprecated 2012-05-13 for removal by 2012-11-31.

`\tl_length:N` Renames.

```

\tl_length:c 5365 \*deprecated
\tl_length:n 5366 \cs_new_eq:NN \tl_length:N \tl_count:n
\tl_length:V 5367 \cs_new_eq:NN \tl_length:c \tl_count:c
\tl_length:o 5368 \cs_new_eq:NN \tl_length:n \tl_count:n
                  5369 \cs_new_eq:NN \tl_length:V \tl_count:V
                  5370 \cs_new_eq:NN \tl_length:o \tl_count:o
                  5371 \</deprecated>

```

(End definition for `\tl_length:N` and others. These functions are documented on page ??.)

Deprecated 2012-06-05 for removal after 2012-12-31.

`\tl_if_empty_p:x` We can test expandably the emptiness of an expanded token list thanks to the primitive
`\tl_if_empty:xTF` `\pdfstrcmp` which expands its argument: a token list is empty if and only if its string representation is empty.

```

5372 \*deprecated
5373 \prg_new_conditional:Npnn \tl_if_empty:x #1 { p , T , F , TF }
5374 { \__str_if_eq_x_return:nn { } {#1} }
5375 \</deprecated>

```

(End definition for `\tl_if_empty:x`. These functions are documented on page ??.)

Deprecated 2012-07-08 for removal after 2012-10-31.

```

\tl_if_head_group_p:n
\tl_if_head_group:nTF
\tl_if_head_N_type_p:n
\tl_if_head_N_type:nTF
\tl_if_head_space_p:n
\tl_if_head_space:nTF
5376 < *deprecated >
5377 \prg_new_eq_conditional:NNn \tl_if_head_group:n \tl_if_head_is_group:n
5378 { p , T , F , TF }
5379 \prg_new_eq_conditional:NNn \tl_if_head_N_type:n \tl_if_head_is_N_type:n
5380 { p , T , F , TF }
5381 \prg_new_eq_conditional:NNn \tl_if_head_space:n \tl_if_head_is_space:n
5382 { p , T , F , TF }
5383 < /deprecated >

```

(End definition for `\tl_if_head_group:n`. These functions are documented on page ??.)

`\tl_tail:w` Deprecated 2012-09-01 for removal after 2012-12-31. This is broken as it will strip braces from a case such as `a{bc}`.

```

5384 < *deprecated >
5385 \cs_new:Npn \tl_tail:w #1#2 \q_stop {#2}
5386 < /deprecated >

```

(End definition for `\tl_tail:w`. This function is documented on page ??.)

```

5387 < /initex | package >

```

11 l3seq implementation

The following test files are used for this code: `m3seq002,m3seq003`.

```

5388 < *initex | package >
5389 < @@=seq >
5390 < *package >
5391 \ProvidesExplPackage
5392 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
5393 \__expl_package_check:
5394 < /package >

```

A sequence is a control sequence whose top-level expansion is of the form “`__seq_item:n {<item1>} ... __seq_item:n {<itemn>}`”. An earlier implementation used the structure “`\seq_elt:w <item1> \seq_elt_end: ... \seq_elt:w <itemn> \seq_elt_end:`”. This allows rapid searching using a delimited function, but is not suitable for items containing `{`, `}` and `#` tokens, and also leads to the loss of surrounding braces around items.

`__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.

```

5395 \cs_new:Npn \__seq_item:n
5396 {
5397   \__msg_kernel_expandable_error:nn { kernel } { misused-sequence }
5398   \use_none:n
5399 }

```

(End definition for _seq_item:n.)

\l__seq_internal_a_tl Scratch space for various internal uses.

\l__seq_internal_b_tl 5400 \tl_new:N \l__seq_internal_a_tl
5401 \tl_new:N \l__seq_internal_b_tl

(End definition for \l__seq_internal_a_tl and \l__seq_internal_b_tl. These variables are documented on page ??.)

\c_empty_seq Simply copy the empty token list.

5402 \cs_new_eq:NN \c_empty_seq \c_empty_tl

(End definition for \c_empty_seq. This variable is documented on page 112.)

11.1 Allocation and initialisation

\seq_new:N Internally, sequences are just token lists.

\seq_new:c 5403 \cs_new_eq:NN \seq_new:N \tl_new:N
5404 \cs_new_eq:NN \seq_new:c \tl_new:c

(End definition for \seq_new:N and \seq_new:c. These functions are documented on page ??.)

\seq_clear:N Clearing sequences is just the same as clearing token lists.

\seq_clear:c 5405 \cs_new_eq:NN \seq_clear:N \tl_clear:N
\seq_gclear:N 5406 \cs_new_eq:NN \seq_clear:c \tl_clear:c
\seq_gclear:c 5407 \cs_new_eq:NN \seq_gclear:N \tl_gclear:N
5408 \cs_new_eq:NN \seq_gclear:c \tl_gclear:c

(End definition for \seq_clear:N and \seq_clear:c. These functions are documented on page ??.)

\seq_clear_new:N Once again a copy from the token list functions.

\seq_clear_new:c 5409 \cs_new_eq:NN \seq_clear_new:N \tl_clear_new:N
\seq_gclear_new:N 5410 \cs_new_eq:NN \seq_clear_new:c \tl_clear_new:c
\seq_gclear_new:c 5411 \cs_new_eq:NN \seq_gclear_new:N \tl_gclear_new:N
5412 \cs_new_eq:NN \seq_gclear_new:c \tl_gclear_new:c

(End definition for \seq_clear_new:N and \seq_clear_new:c. These functions are documented on page ??.)

\seq_set_eq:NN Once again, these are simple copies from the token list functions.

\seq_set_eq:cN 5413 \cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN
\seq_set_eq:Nc 5414 \cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc
\seq_set_eq:cc 5415 \cs_new_eq:NN \seq_set_eq:cN \tl_set_eq:cN
\seq_gset_eq:NN 5416 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
\seq_gset_eq:cN 5417 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
\seq_gset_eq:Nc 5418 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
\seq_gset_eq:cN 5419 \cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
5420 \cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc

(End definition for \seq_set_eq:NN and others. These functions are documented on page ??.)

```

\seq_set_split:Nnn
\seq_set_split:NnV
\seq_gset_split:Nnn
\seq_gset_split:NnV
\__seq_set_split:NNnn
\__seq_set_split_auxi:w
\__seq_set_split_auxii:w
\__seq_set_split_end:

```

The goal is to split a given token list at a 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.

```

5421 \cs_new_protected_nopar:Npn \seq_set_split:Nnn
5422 { \__seq_set_split:NNnn \tl_set:Nx }
5423 \cs_new_protected_nopar:Npn \seq_gset_split:Nnn
5424 { \__seq_set_split:NNnn \tl_gset:Nx }
5425 \cs_new_protected:Npn \__seq_set_split:NNnn #1 #2 #3 #4
5426 {
5427   \tl_if_empty:nTF {#3}
5428   { #1 #2 { \tl_map_function:nN {#4} \__seq_wrap_item:n } }
5429   {
5430     \tl_set:Nn \l__seq_internal_a_tl
5431     {
5432       \__seq_set_split_auxi:w \prg_do_nothing:
5433       #4
5434       \__seq_set_split_end:
5435     }
5436     \tl_replace_all:Nnn \l__seq_internal_a_tl { #3 }
5437     {
5438       \__seq_set_split_end:
5439       \__seq_set_split_auxi:w \prg_do_nothing:
5440     }
5441     \tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
5442     #1 #2 { \l__seq_internal_a_tl }
5443   }
5444 }
5445 \cs_new:Npn \__seq_set_split_auxi:w #1 \__seq_set_split_end:
5446 {
5447   \exp_not:N \__seq_set_split_auxii:w
5448   \exp_args:No \tl_trim_spaces:n {#1}
5449   \exp_not:N \__seq_set_split_end:
5450 }
5451 \cs_new:Npn \__seq_set_split_auxii:w #1 \__seq_set_split_end:
5452 { \__seq_wrap_item:n {#1} }
5453 \cs_generate_variant:Nn \seq_set_split:Nnn { NnV }
5454 \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

```

Concatenating sequences is easy.

```

5455 \cs_new_eq:NN \seq_concat:NNN \tl_concat:NNN

```

```

5456 \cs_new_eq:NN \seq_gconcat:NNN \tl_gconcat:NNN
5457 \cs_new_eq:NN \seq_concat:ccc \tl_concat:ccc
5458 \cs_new_eq:NN \seq_gconcat:ccc \tl_gconcat:ccc

```

(End definition for \seq_concat:NNN and \seq_concat:ccc. These functions are documented on page ??.)

\seq_if_exist_p:N Copies of the cs functions defined in l3basics.

```

\seq_if_exist_p:c      5459 \prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N { TF , T , F , p }
\seq_if_exist:NTF      5460 \prg_new_eq_conditional:NNn \seq_if_exist:c \cs_if_exist:c { TF , T , F , p }
\seq_if_exist:cTF

```

(End definition for \seq_if_exist:N and \seq_if_exist:c. These functions are documented on page ??.)

11.2 Appending data to either end

\seq_put_left:Nn The code here is just a wrapper for adding to token lists.

```

\seq_put_left:NV      5461 \cs_new_protected:Npn \seq_put_left:Nn #1#2
\seq_put_left:Nv      5462 { \tl_put_left:Nn #1 { \__seq_item:n {#2} } }
\seq_put_left:No      5463 \cs_new_protected:Npn \seq_put_right:Nn #1#2
\seq_put_left:Nx      5464 { \tl_put_right:Nn #1 { \__seq_item:n {#2} } }
\seq_put_left:cn      5465 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
\seq_put_left:cV      5466 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
\seq_put_left:cV      5467 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
\seq_put_left:cV      5468 \cs_generate_variant:Nn \seq_put_right:Nn { c , cV , cv , co , cx }
\seq_put_left:co
\seq_put_left:cx

```

(End definition for \seq_put_left:Nn and others. These functions are documented on page ??.)

\seq_gput_left:Nn The same for global addition.

```

\seq_gput_left:NV      5469 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
\seq_gput_left:Nv      5470 { \tl_gput_left:Nn #1 { \__seq_item:n {#2} } }
\seq_gput_left:Nv      5471 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
\seq_gput_left:No      5472 { \tl_gput_right:Nn #1 { \__seq_item:n {#2} } }
\seq_gput_left:Nx      5473 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
\seq_gput_left:cn      5474 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }
\seq_gput_left:cV      5475 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
\seq_gput_left:cV      5476 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
\seq_gput_left:cV
\seq_gput_left:co
\seq_gput_left:cx

```

(End definition for \seq_gput_left:Nn and others. These functions are documented on page ??.)

\seq_gput_right:Nn

\seq_gput_right:NV

\seq_gput_right:Nv

__seq_wrap_item:n

\seq_gput_right:No

\seq_gput_right:Nx

\seq_gput_right:cn

\seq_gput_right:cV

\l__seq_remove_seq

\seq_gput_right:cV

\seq_gput_right:co

\seq_gput_right:cx

11.3 Modifying sequences

This function converts its argument to a proper sequence item in an x-expansion context.

```

5477 \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.

```

5478 \seq_new:N \l__seq_remove_seq

```

(End definition for \l__seq_remove_seq. This variable is documented on page ??.)

```

\seq_remove_duplicates:N
\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
__seq_remove_duplicates:NN

```

Removing duplicates means making a new list then copying it.

```

5479 \cs_new_protected:Npn \seq_remove_duplicates:N
5480 { __seq_remove_duplicates:NN \seq_set_eq:NN }
5481 \cs_new_protected:Npn \seq_gremove_duplicates:N
5482 { __seq_remove_duplicates:NN \seq_gset_eq:NN }
5483 \cs_new_protected:Npn __seq_remove_duplicates:NN #1#2
5484 {
5485   \seq_clear:N \l__seq_remove_seq
5486   \seq_map_inline:Nn #2
5487   {
5488     \seq_if_in:NnF \l__seq_remove_seq {##1}
5489     { \seq_put_right:Nn \l__seq_remove_seq {##1} }
5490   }
5491   #1 #2 \l__seq_remove_seq
5492 }
5493 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
5494 \cs_generate_variant:Nn \seq_gremove_duplicates:N { c }

```

(End definition for \seq_remove_duplicates:N and \seq_remove_duplicates:c. These functions are documented on page ??.)

```

\seq_remove_all:Nn
\seq_remove_all:cn
\seq_gremove_all:Nn
\seq_gremove_all:cn
__seq_remove_all_aux:NNn

```

The idea of the code here is to avoid a relatively expensive addition of items one at a time to an intermediate sequence. The approach taken is therefore similar to that in __seq_pop_right_aux:NNN, using a “flexible” x-type expansion to do most of the work. As \tl_if_eq:nnT is not expandable, a two-part strategy is needed. First, the x-type expansion uses \str_if_eq:nnT to find potential matches. If one is found, the expansion is halted and the necessary set up takes place to use the \tl_if_eq:NNT test. The x-type is started again, including all of the items copied already. This will happen repeatedly until the entire sequence has been scanned. The code is set up to avoid needing and intermediate scratch list: the lead-off x-type expansion (#1 #2 {#2}) will ensure that nothing is lost.

```

5495 \cs_new_protected:Npn \seq_remove_all:Nn
5496 { __seq_remove_all_aux:NNn \tl_set:Nx }
5497 \cs_new_protected:Npn \seq_gremove_all:Nn
5498 { __seq_remove_all_aux:NNn \tl_gset:Nx }
5499 \cs_new_protected:Npn __seq_remove_all_aux:NNn #1#2#3
5500 {
5501   __seq_push_item_def:n
5502   {
5503     \str_if_eq:nnT {##1} {#3}
5504     {
5505       \if_false: { \fi: }
5506       \tl_set:Nn \l__seq_internal_b_tl {##1}
5507       #1 #2
5508       { \if_false: } \fi:
5509       \exp_not:o {#2}
5510       \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
5511       { \use_none:nn }
5512     }
5513     __seq_wrap_item:n {##1}

```



```

5514     }
5515     \tl_set:Nn \l__seq_internal_a_tl {#3}
5516     #1 #2 {#2}
5517     \__seq_pop_item_def:
5518   }
5519   \cs_generate_variant:Nn \seq_remove_all:Nn { c }
5520   \cs_generate_variant:Nn \seq_gremove_all:Nn { c }

```

(End definition for `\seq_remove_all:Nn` and `\seq_remove_all:cn`. These functions are documented on page ??.)

11.4 Sequence conditionals

`\seq_if_empty_p:N` Simple copies from the token list variable material.

```

\seq_if_empty_p:c 5521 \prg_new_eq_conditional:NNn \seq_if_empty:N \tl_if_empty:N
\seq_if_empty:NTF 5522 { p , T , F , TF }
\seq_if_empty:cTF 5523 \prg_new_eq_conditional:NNn \seq_if_empty:c \tl_if_empty:c
5524 { p , T , F , TF }

```

(End definition for `\seq_if_empty:N` and `\seq_if_empty:c`. These functions are documented on page ??.)

`\seq_if_in:NnTF` The approach here is to define `__seq_item:n` to compare its argument with the test sequence. If the two items are equal, the mapping is terminated and `\group_end: \prg_return_true:` is inserted after skipping over the rest of the recursion. On the other hand, if there is no match then the loop will break returning `\prg_return_false:`. Everything is inside a group so that `__seq_item:n` is preserved in nested situations.

```

\seq_if_in:cnTF 5525 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
\seq_if_in:cVTF 5526 { T , F , TF }
\seq_if_in:cVTF 5527 {
\seq_if_in:coTF 5528   \group_begin:
\seq_if_in:cxTF 5529   \tl_set:Nn \l__seq_internal_a_tl {#2}
\__seq_if_in: 5530   \cs_set_protected:Npn \__seq_item:n ##1
5531   {
5532     \tl_set:Nn \l__seq_internal_b_tl {##1}
5533     \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
5534       \exp_after:wN \__seq_if_in:
5535     \fi:
5536   }
5537   #1
5538   \group_end:
5539   \prg_return_false:
5540   \__prg_break_point:
5541 }
5542 \cs_new_nopar:Npn \__seq_if_in:
5543 { \__prg_break:n { \group_end: \prg_return_true: } }
5544 \cs_generate_variant:Nn \seq_if_in:NnT { NV , Nv , No , Nx }
5545 \cs_generate_variant:Nn \seq_if_in:NnT { c , cV , cv , co , cx }
5546 \cs_generate_variant:Nn \seq_if_in:NnF { NV , Nv , No , Nx }
5547 \cs_generate_variant:Nn \seq_if_in:NnF { c , cV , cv , co , cx }
5548 \cs_generate_variant:Nn \seq_if_in:NnTF { NV , Nv , No , Nx }

```

```
5549 \cs_generate_variant:Nn \seq_if_in:NnTF { c , cV , cv , co , cx }
(End definition for \seq_if_in:Nn and others. These functions are documented on page ??.)
```

11.5 Recovering data from sequences

`__seq_pop:NNNN` The two pop functions share their emptiness tests. We also use a common emptiness test
`__seq_pop_TF:NNNN` for all branching get and pop functions.

```
5550 \cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
5551 {
5552   \if_meaning:w #3 \c_empty_seq
5553   \tl_set:Nn #4 { \q_no_value }
5554   \else:
5555     #1#2#3#4
5556   \fi:
5557 }
5558 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
5559 {
5560   \if_meaning:w #3 \c_empty_seq
5561   % \tl_set:Nn #4 { \q_no_value }
5562   \prg_return_false:
5563   \else:
5564     #1#2#3#4
5565     \prg_return_true:
5566   \fi:
5567 }
```

(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 removing the `__seq_item:n` at the start. We first append a `\q_no_value` item to
`__seq_get_left:NnwN` cover the case of an empty sequence

```
5568 \cs_new_protected:Npn \seq_get_left:NN #1#2
5569 {
5570   \tl_set:Nx #2
5571   {
5572     \exp_after:wN \__seq_get_left:Nnw
5573     #1 \__seq_item:n { \q_no_value } \q_stop
5574   }
5575 }
5576 \cs_new:Npn \__seq_get_left:Nnw \__seq_item:n #1#2 \q_stop
5577 { \exp_not:n {#1} }
5578 \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` `\cs_new_protected_nopar:Npn \seq_pop_left:NN`
`__seq_pop_left:NNN`
`__seq_pop_left:NnwNNN`

```

5580 { \_seq_pop:NNNN \_seq_pop_left:NNN \tl_set:Nn }
5581 \cs_new_protected_nopar:Npn \seq_gpop_left:NN
5582 { \_seq_pop:NNNN \_seq_pop_left:NNN \tl_gset:Nn }
5583 \cs_new_protected:Npn \_seq_pop_left:NNN #1#2#3
5584 { \exp_after:wN \_seq_pop_left:NnwNNN #2 \q_stop #1#2#3 }
5585 \cs_new_protected:Npn \_seq_pop_left:NnwNNN \_seq_item:n #1#2 \q_stop #3#4#5
5586 {
5587   #3 #4 {#2}
5588   \tl_set:Nn #5 {#1}
5589 }
5590 \cs_generate_variant:Nn \seq_pop_left:NN { c }
5591 \cs_generate_variant:Nn \seq_gpop_left:NN { c }

```

(End definition for \seq_pop_left:NN and \seq_pop_left:cN. These functions are documented on page ??.)

`\seq_get_right:NN`
`\seq_get_right:cN`
`_seq_get_right_loop:nn`

First prepend \q_no_value, then take two arguments at a time. Apart from the right-hand end of the sequence, this be a brace group followed by _seq_item:n. The \use_none:nn removes both of those. At the end of the sequence, the two question marks are taken by \use_none:nn, and the assignment is placed before the right-most item. The \afterassignment primitive places \use_none:n to get rid of a trailing _seq_get_right_loop:nn.

```

5592 \cs_new_protected:Npn \seq_get_right:NN #1#2
5593 {
5594   \exp_after:wN \_seq_get_right_loop:nn
5595   \exp_after:wN \q_no_value
5596   #1
5597   {
5598     ??
5599     \tex_afterassignment:D \use_none:n
5600     \tl_set:Nn #2
5601   }
5602 }
5603 \cs_new_protected:Npn \_seq_get_right_loop:nn #1#2
5604 {
5605   \use_none:nn #2 {#1}
5606   \_seq_get_right_loop:nn
5607 }
5608 \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`
`\seq_pop_right:cN`
`\seq_gpop_right:NN`
`\seq_gpop_right:cN`
`_seq_pop_right_aux:NNN`
`_seq_pop_right_loop:nn`

The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a “flexible length” way to set a token list variable. This is supplied by the { \if_false: } \fi: ... \if_false: { \fi: } construct. Using an x-type expansion and a “non-expanding” definition for _seq_item:n, the left-most $n - 1$ entries in a sequence of n items will be stored back in the sequence. That needs a loop of unknown length, hence using the strange \if_false: way of including brackets. When the last item of the sequence is reached, the closing bracket for the assignment is inserted, and \tl_set:Nn #3 is inserted

in front of the final entry. This therefore does the pop assignment. The trailing looping macro is removed by placing a `\use_none:n` using the `\afterassignment` primitive.

```

5609 \cs_new_protected_nopar:Npn \seq_pop_right:NN
5610 { \__seq_pop:NNNN \__seq_pop_right_aux:NNN \tl_set:Nx }
5611 \cs_new_protected_nopar:Npn \seq_gpop_right:NN
5612 { \__seq_pop:NNNN \__seq_pop_right_aux:NNN \tl_gset:Nx }
5613 \cs_new_protected:Npn \__seq_pop_right_aux:NNN #1#2#3
5614 {
5615   \cs_set_eq:NN \seq_tmp:w \__seq_item:n
5616   \cs_set_eq:NN \__seq_item:n \scan_stop:
5617   #1 #2
5618   { \if_false: } \fi:
5619   \exp_after:wN \exp_after:wN
5620   \exp_after:wN \__seq_pop_right_loop:nn
5621   \exp_after:wN \use_none:n
5622   #2
5623   {
5624     \if_false: { \fi: }
5625     \tex_afterassignment:D \use_none:n
5626     \tl_set:Nx #3
5627   }
5628   \cs_set_eq:NN \__seq_item:n \seq_tmp:w
5629 }
5630 \cs_new:Npn \__seq_pop_right_loop:nn #1#2
5631 {
5632   #2 { \exp_not:n {#1} }
5633   \__seq_pop_right_loop:nn
5634 }
5635 \cs_generate_variant:Nn \seq_pop_right:NN { c }
5636 \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_get_left:cNTF` `\seq_pop_TF:NNNN` is left unused.

```

\seq_get_right:NNTF 5637 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
\seq_get_right:cNTF 5638 { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
5639 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
5640 { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
5641 \cs_generate_variant:Nn \seq_get_left:NNT { c }
5642 \cs_generate_variant:Nn \seq_get_left:NNF { c }
5643 \cs_generate_variant:Nn \seq_get_left:NNTF { c }
5644 \cs_generate_variant:Nn \seq_get_right:NNT { c }
5645 \cs_generate_variant:Nn \seq_get_right:NNF { c }
5646 \cs_generate_variant:Nn \seq_get_right:NNTF { c }

```

(End definition for `\seq_get_left:NN` and `\seq_get_left:cN`. These functions are documented on page ??.)

`\seq_pop_left:NNTF` More or less the same for popping.

```

\seq_pop_left:cNTF
\seq_gpop_left:NNTF
\seq_gpop_left:cNTF
\seq_pop_right:NNTF
\seq_pop_right:cNTF
\seq_gpop_right:NNTF
\seq_gpop_right:cNTF

```

```

5647 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 { T , F , TF }
5648 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
5649 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 { T , F , TF }
5650 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
5651 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2 { T , F , TF }
5652 { \__seq_pop_TF:NNNN \__seq_pop_right_aux:NNN \tl_set:Nx #1 #2 }
5653 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2 { T , F , TF }
5654 { \__seq_pop_TF:NNNN \__seq_pop_right_aux:NNN \tl_gset:Nx #1 #2 }
5655 \cs_generate_variant:Nn \seq_pop_left:NNT { c }
5656 \cs_generate_variant:Nn \seq_pop_left:NNF { c }
5657 \cs_generate_variant:Nn \seq_pop_left:NNTF { c }
5658 \cs_generate_variant:Nn \seq_gpop_left:NNT { c }
5659 \cs_generate_variant:Nn \seq_gpop_left:NNF { c }
5660 \cs_generate_variant:Nn \seq_gpop_left:NNTF { c }
5661 \cs_generate_variant:Nn \seq_pop_right:NNT { c }
5662 \cs_generate_variant:Nn \seq_pop_right:NNF { c }
5663 \cs_generate_variant:Nn \seq_pop_right:NNTF { c }
5664 \cs_generate_variant:Nn \seq_gpop_right:NNT { c }
5665 \cs_generate_variant:Nn \seq_gpop_right:NNF { c }
5666 \cs_generate_variant:Nn \seq_gpop_right:NNTF { c }

```

(End definition for `\seq_pop_left:NN` and `\seq_pop_left:cN`. These functions are documented on page ??.)

11.6 Mapping to sequences

`\seq_map_break:` To break a function, the special token `__prg_break_point:Nn` is used to find the end of the code. Any ending code is then inserted before the return value of `\seq_map_break:n` is inserted.

```

5667 \cs_new_nopar:Npn \seq_map_break:
5668 { \__prg_map_break:Nn \seq_map_break: { } }
5669 \cs_new_nopar:Npn \seq_map_break:n
5670 { \__prg_map_break:Nn \seq_map_break: }

```

(End definition for `\seq_map_break:`. This function is documented on page 109.)

`\seq_map_function:NN` The idea here is to apply the code of #2 to each item in the sequence without altering the definition of `__seq_item:n`. This is done as by noting that every odd token in the sequence must be `__seq_item:n`, which can be gobbled by `\use_none:n`. At the end of the loop, #2 is instead `? \seq_map_break:`, which therefore breaks the loop without needing to do a (relatively-expensive) quark test.

```

5671 \cs_new:Npn \seq_map_function:NN #1#2
5672 {
5673   \exp_after:wN \__seq_map_function:NNn \exp_after:wN #2 #1
5674   { ? \seq_map_break: } { }
5675   \__prg_break_point:Nn \seq_map_break: { }
5676 }
5677 \cs_new:Npn \__seq_map_function:NNn #1#2#3
5678 {
5679   \use_none:n #2

```

```

5680     #1 {#3}
5681     \__seq_map_function:NNn #1
5682   }
5683   \cs_generate_variant:Nn \seq_map_function:NN { c }

```

(End definition for \seq_map_function:NN and \seq_map_function:cN. These functions are documented on page ??.)

```

\__seq_push_item_def:n
\__seq_push_item_def:x
\__seq_push_item_def:
\__seq_pop_item_def:

```

The definition of __seq_item:n needs to be saved and restored at various points within the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

```

5684 \cs_new_protected:Npn \__seq_push_item_def:n
5685 {
5686   \__seq_push_item_def:
5687   \cs_gset:Npn \__seq_item:n ##1
5688 }
5689 \cs_new_protected:Npn \__seq_push_item_def:x
5690 {
5691   \__seq_push_item_def:
5692   \cs_gset:Npx \__seq_item:n ##1
5693 }
5694 \cs_new_protected:Npn \__seq_push_item_def:
5695 {
5696   \int_gincr:N \g__prg_map_int
5697   \cs_gset_eq:cN { __prg_map_ \int_use:N \g__prg_map_int :w }
5698   \__seq_item:n
5699 }
5700 \cs_new_protected_nopar:Npn \__seq_pop_item_def:
5701 {
5702   \cs_gset_eq:Nc \__seq_item:n
5703   { __prg_map_ \int_use:N \g__prg_map_int :w }
5704   \int_gdecr:N \g__prg_map_int
5705 }

```

(End definition for __seq_push_item_def:n and __seq_push_item_def:x. These functions are documented on page 112.)

```

\seq_map_inline:Nn
\seq_map_inline:cn

```

The idea here is that __seq_item:n is already “applied” to each item in a sequence, and so an in-line mapping is just a case of redefining __seq_item:n.

```

5706 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
5707 {
5708   \__seq_push_item_def:n {#2}
5709   #1
5710   \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
5711 }
5712 \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
\seq_map_variable:Ncn
\seq_map_variable:cNn
\seq_map_variable:ccn

```

This is just a specialised version of the in-line mapping function, using an x-type expansion for the code set up so that the number of # tokens required is as expected.

```

5713 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
5714 {
5715   \__seq_push_item_def:x
5716   {
5717     \tl_set:Nn \exp_not:N #2 {##1}
5718     \exp_not:n {#3}
5719   }
5720   #1
5721   \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
5722 }
5723 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
5724 \cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }

```

(End definition for \seq_map_variable:NNn and others. These functions are documented on page ??.)

\seq_count:N Counting the items in a sequence is done using the same approach as for other count functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics.

\seq_count:c
__seq_count:n

```

5725 \cs_new:Npn \seq_count:N #1
5726 {
5727   \int_eval:n
5728   {
5729     0
5730     \seq_map_function:NN #1 \__seq_count:n
5731   }
5732 }
5733 \cs_new:Npn \__seq_count:n #1 { + \c_one }
5734 \cs_generate_variant:Nn \seq_count:N { c }

```

(End definition for \seq_count:N and \seq_count:c. These functions are documented on page ??.)

11.7 Using sequences

\seq_use:Nnnn See \clist_use:Nnnn for a general explanation. The main difference is that we use _seq_item:n as a delimiter rather than commas. We also need to add __seq_item:n at various places.

\seq_use:cnnn

__seq_use:NnNnn

__seq_use:nwwwwnwn

__seq_use:nwwn

\seq_use:Nn

\seq_use:cn

```

5735 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
5736 {
5737   \seq_if_exist:NTF #1
5738   {
5739     \int_case:nnn { \seq_count:N #1 }
5740     {
5741       { 0 } { }
5742       { 1 } { \exp_after:wN \__seq_use:NnNnn #1 \__seq_item:n { } { } }
5743       { 2 } { \exp_after:wN \__seq_use:NnNnn #1 {#2} }
5744     }
5745     {
5746       \exp_after:wN \__seq_use:nwwwwnwn
5747       \exp_after:wN { \exp_after:wN } #1 \__seq_item:n
5748       \q_mark { \__seq_use:nwwwwnwn {#3} }

```

```

5749         \q_mark { \__seq_use:nwnn {#4} }
5750         \q_stop { }
5751     }
5752 }
5753 { \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#1} }
5754 }
5755 \cs_generate_variant:Nn \seq_use:Nnnn { c }
5756 \cs_new:Npn \__seq_use:NnNnn \__seq_item:n #1 \__seq_item:n #2#3
5757 { \exp_not:n { #1 #3 #2 } }
5758 \cs_new:Npn \__seq_use:nwwwnwnn
5759 #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
5760 \q_mark #6#7 \q_stop #8
5761 {
5762 #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
5763 \q_mark {#6} #7 \q_stop { #8 #1 #2 }
5764 }
5765 \cs_new:Npn \__seq_use:nwnn #1 \__seq_item:n #2 #3 \q_stop #4
5766 { \exp_not:n { #4 #1 #2 } }
5767 \cs_new:Npn \seq_use:Nn #1#2
5768 { \seq_use:Nnnn #1 {#2} {#2} {#2} }
5769 \cs_generate_variant:Nn \seq_use:Nn { c }

```

(End definition for \seq_use:Nnnn and \seq_use:cnnn. These functions are documented on page ??.)

11.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

\seq_push:Nn Pushing to a sequence is the same as adding on the left.

```

\seq_push:NV 5770 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
\seq_push:Nv 5771 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
\seq_push:No 5772 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
\seq_push:Nx 5773 \cs_new_eq:NN \seq_push:No \seq_put_left:No
\seq_push:cn 5774 \cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx
\seq_push:cV 5775 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
\seq_push:cV 5776 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
\seq_push:cv 5777 \cs_new_eq:NN \seq_push:cv \seq_put_left:cv
\seq_push:co 5778 \cs_new_eq:NN \seq_push:co \seq_put_left:co
\seq_push:cx 5779 \cs_new_eq:NN \seq_push:cx \seq_put_left:cx
\seq_gpush:Nn 5780 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
\seq_gpush:NV 5781 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
\seq_gpush:Nv 5782 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
\seq_gpush:No 5783 \cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
\seq_gpush:Nx 5784 \cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
\seq_gpush:cn 5785 \cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
\seq_gpush:cV 5786 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
\seq_gpush:cV 5787 \cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv
\seq_gpush:cv 5788 \cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
\seq_gpush:co 5789 \cs_new_eq:NN \seq_gpush:cx \seq_gput_left:cx
\seq_gpush:cx

```

(End definition for \seq_push:Nn and others. These functions are documented on page ??.)

`\seq_get:NN` In most cases, getting items from the stack does not need to specify that this is from the left. So alias are provided.

`\seq_get:cN`

`\seq_pop:NN` 5790 `\cs_new_eq:NN \seq_get:NN \seq_get_left:NN`

`\seq_pop:cN` 5791 `\cs_new_eq:NN \seq_get:cN \seq_get_left:cN`

`\seq_gpop:NN` 5792 `\cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN`

`\seq_gpop:cN` 5793 `\cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN`

5794 `\cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN`

5795 `\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` 5796 `\prg_new_eq_conditional:NNn \seq_get:NN \seq_get_left:NN { T , F , TF }`

`\seq_pop:NNTF` 5797 `\prg_new_eq_conditional:NNn \seq_get:cN \seq_get_left:cN { T , F , TF }`

`\seq_pop:cNTF` 5798 `\prg_new_eq_conditional:NNn \seq_pop:NN \seq_pop_left:NN { T , F , TF }`

`\seq_gpop:NNTF` 5799 `\prg_new_eq_conditional:NNn \seq_pop:cN \seq_pop_left:cN { T , F , TF }`

`\seq_gpop:cNTF` 5800 `\prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }`

5801 `\prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }`

(End definition for `\seq_get:NN` and `\seq_get:cN`. These functions are documented on page ??.)

11.9 Viewing sequences

`\seq_show:N` Apply the general `_msg_show_variable:Nnn`.

`\seq_show:c` 5802 `\cs_new_protected:Npn \seq_show:N #1`

5803 `{`

5804 `_msg_show_variable:Nnn #1 { seq }`

5805 `{ \seq_map_function:NN #1 _msg_show_item:n }`

5806 `}`

5807 `\cs_generate_variant:Nn \seq_show:N { c }`

(End definition for `\seq_show:N` and `\seq_show:c`. These functions are documented on page ??.)

11.10 Scratch sequences

`\l_tmpa_seq` Temporary comma list variables.

`\l_tmpb_seq` 5808 `\seq_new:N \l_tmpa_seq`

`\g_tmpa_seq` 5809 `\seq_new:N \l_tmpb_seq`

`\g_tmpb_seq` 5810 `\seq_new:N \g_tmpa_seq`

5811 `\seq_new:N \g_tmpb_seq`

(End definition for `\l_tmpa_seq` and others. These variables are documented on page 112.)

11.11 Deprecated interfaces

A few functions which are no longer documented: these were moved here on or before 2011-04-20, and will be removed entirely by 2011-07-20.

`\seq_top:NN` These are old stack functions.

`\seq_top:cN`

```

5812 <*deprecated>
5813 \cs_new_eq:NN \seq_top:NN \seq_get_left:NN
5814 \cs_new_eq:NN \seq_top:cN \seq_get_left:cN
5815 </deprecated>

```

(End definition for `\seq_top:NN` and `\seq_top:cN`. These functions are documented on page ??.)

`\seq_display:N` An older name for `\seq_show:N`.

`\seq_display:c`

```

5816 <*deprecated>
5817 \cs_new_eq:NN \seq_display:N \seq_show:N
5818 \cs_new_eq:NN \seq_display:c \seq_show:c
5819 </deprecated>

```

(End definition for `\seq_display:N` and `\seq_display:c`. These functions are documented on page ??.)

Deprecated 2012-05-13 for removal by 2012-11-30.

`\seq_length:N`

`\seq_length:c`

```

5820 <*deprecated>
5821 \cs_new_eq:NN \seq_length:N \seq_count:N
5822 \cs_new_eq:NN \seq_length:c \seq_count:c
5823 </deprecated>

```

(End definition for `\seq_length:N` and `\seq_length:c`. These functions are documented on page ??.)

Deprecated 2012-05-23 for removal by 2012-08-30.

`\seq_use:N` A simple short cut for a mapping.

`\seq_use:c`

```

5824 <*deprecated>
5825 \cs_new:Npn \seq_use:N #1 { \seq_use:Nn #1 { } }
5826 \cs_generate_variant:Nn \seq_use:N { c }
5827 </deprecated>

```

(End definition for `\seq_use:N` and `\seq_use:c`. These functions are documented on page ??.)

5828 </initex | package>

12 l3clist implementation

The following test files are used for this code: `m3clist002`.

```

5829 <*initex | package>
5830 <@@=clist>
5831 <*package>
5832 \ProvidesExplPackage
5833   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
5834   \_expl_package_check:
5835 </package>

```

`\c_empty_clist` An empty comma list is simply an empty token list.

```

5836 \cs_new_eq:NN \c_empty_clist \c_empty_tl

```

(End definition for `\c_empty_clist`. This variable is documented on page 121.)

`\l__clist_internal_clist` Scratch space for various internal uses. This comma list variable cannot be declared as such because it comes before `\clist_new:N`

5837 `\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.

5838 `\cs_new_protected:Npn __clist_tmp:w { }`
(End definition for __clist_tmp:w.)

12.1 Allocation and initialisation

`\clist_new:N` Internally, comma lists are just token lists.

`\clist_new:c` 5839 `\cs_new_eq:NN \clist_new:N \tl_new:N`
5840 `\cs_new_eq:NN \clist_new:c \tl_new:c`
(End definition for \clist_new:N and \clist_new:c. These functions are documented on page ??.)

`\clist_clear:N` Clearing comma lists is just the same as clearing token lists.

`\clist_clear:c` 5841 `\cs_new_eq:NN \clist_clear:N \tl_clear:N`
`\clist_gclear:N` 5842 `\cs_new_eq:NN \clist_clear:c \tl_clear:c`
`\clist_gclear:c` 5843 `\cs_new_eq:NN \clist_gclear:N \tl_gclear:N`
5844 `\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` 5845 `\cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N`
`\clist_gclear_new:N` 5846 `\cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c`
`\clist_gclear_new:c` 5847 `\cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N`
5848 `\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` 5849 `\cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN`
`\clist_set_eq:Nc` 5850 `\cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc`
`\clist_set_eq:cc` 5851 `\cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN`
`\clist_gset_eq:NN` 5852 `\cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc`
`\clist_gset_eq:cN` 5853 `\cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN`
`\clist_gset_eq:Nc` 5854 `\cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc`
`\clist_gset_eq:cN` 5855 `\cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN`
`\clist_gset_eq:cc` 5856 `\cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc`
(End definition for \clist_set_eq:NN and others. These functions are documented on page ??.)

`\clist_concat:NNN` Concatenating comma lists is not quite as easy as it seems, as there needs to be the correct addition of a comma to the output. So a little work to do.

`\clist_concat:ccc`

`\clist_gconcat:NNN`

`\clist_gconcat:ccc`

`__clist_concat:NNNN`

```

5857 \cs_new_protected_nopar:Npn \clist_concat:NNN
5858 { \__clist_concat:NNNN \tl_set:Nx }
5859 \cs_new_protected_nopar:Npn \clist_gconcat:NNN
5860 { \__clist_concat:NNNN \tl_gset:Nx }
5861 \cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
5862 {
5863   #1 #2
5864   {
5865     \exp_not:o #3
5866     \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
5867     \exp_not:o #4
5868   }
5869 }
5870 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
5871 \cs_generate_variant:Nn \clist_gconcat:NNN { ccc }

```

(End definition for `\clist_concat:NNN` and `\clist_concat:ccc`. These functions are documented on page ??.)

`\clist_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

`\clist_if_exist_p:c`

`\clist_if_exist:N \overline{TF}`

`\clist_if_exist:c \overline{TF}`

```

5872 \prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N { TF , T , F , p }
5873 \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 ??.)

12.2 Removing spaces around items

`_clist_trim_spaces_generic:nw` 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 $\langle trimmed\ item \rangle$.

`__clist_trim_spaces_generic:nn`

```

5874 \cs_new:Npn \__clist_trim_spaces_generic:nw #1#2 ,
5875 {
5876   \_tl_trim_spaces:nn {#2}
5877   { \exp_args:No \__clist_trim_spaces_generic:nn } {#1}
5878 }
5879 \cs_new:Npn \__clist_trim_spaces_generic:nn #1#2 { #2 {#1} }

```

(End definition for `__clist_trim_spaces_generic:nw`. This function is documented on page ??.)

`__clist_trim_spaces:n` 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.

`__clist_trim_spaces:nn`

```

5880 \cs_new:Npn \__clist_trim_spaces:n #1
5881 {

```

```

5882     \_clist_trim_spaces_generic:nw
5883     { \_clist_trim_spaces:nn { } }
5884     \q_mark #1 ,
5885     \q_recursion_tail, \q_recursion_stop
5886 }
5887 \cs_new:Npn \_clist_trim_spaces:nn #1 #2
5888 {
5889     \quark_if_recursion_tail_stop:n {#2}
5890     \tl_if_empty:nTF {#2}
5891     {
5892         \_clist_trim_spaces_generic:nw
5893         { \_clist_trim_spaces:nn {#1} } \q_mark
5894     }
5895     {
5896         #1 \exp_not:n {#2}
5897         \_clist_trim_spaces_generic:nw
5898         { \_clist_trim_spaces:nn { , } } \q_mark
5899     }
5900 }

```

(End definition for _clist_trim_spaces:n. This function is documented on page ??.)

12.3 Adding data to comma lists

```

\clist_set:Nn
\clist_set:NV 5901 \cs_new_protected:Npn \clist_set:Nn #1#2
\clist_set:No 5902 { \tl_set:Nx #1 { \_clist_trim_spaces:n {#2} } }
\clist_set:Nx 5903 \cs_new_protected:Npn \clist_gset:Nn #1#2
\clist_set:cn 5904 { \tl_gset:Nx #1 { \_clist_trim_spaces:n {#2} } }
\clist_set:cV 5905 \cs_generate_variant:Nn \clist_set:Nn { NV , No , Nx , c , cV , co , cx }
\clist_set:co 5906 \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 ??.)

Comma lists cannot hold empty values: there are therefore a couple of sanity checks to avoid accumulating commas.

```

\clist_gset:Nn
\clist_put_left:Nn
\clist_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
\clist_gput_left:Nn
\clist_gput_left:NV
\clist_gput_left:No
\clist_gput_left:Nx
\clist_gput_left:cn
\clist_gput_left:cV
\clist_gput_left:co
\clist_gput_left:cx
\_clist_put_left:NNNn

```

```

5907 \cs_new_protected_nopar:Npn \clist_put_left:Nn
5908 { \_clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
5909 \cs_new_protected_nopar:Npn \clist_gput_left:Nn
5910 { \_clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
5911 \cs_new_protected:Npn \_clist_put_left:NNNn #1#2#3#4
5912 {
5913     #2 \l__clist_internal_clist {#4}
5914     #1 #3 \l__clist_internal_clist #3
5915 }
5916 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
5917 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
5918 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
5919 \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

```

```

5920 \cs_new_protected_nopar:Npn \clist_put_right:Nn
5921 { \__clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
5922 \cs_new_protected_nopar:Npn \clist_gput_right:Nn
5923 { \__clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
5924 \cs_new_protected:Npn \__clist_put_right:NNNn #1#2#3#4
5925 {
5926     #2 \l__clist_internal_clist {#4}
5927     #1 #3 #3 \l__clist_internal_clist
5928 }
5929 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
5930 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
5931 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
5932 \cs_generate_variant:Nn \clist_gput_right:Nn { c , cV , co , cx }

```

(End definition for \clist_put_right:Nn and others. These functions are documented on page ??.)

12.4 Comma lists as stacks

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma.

```

\clist_get:NN
\clist_get:cN
\__clist_get:wN

```

```

5933 \cs_new_protected:Npn \clist_get:NN #1#2
5934 {
5935     \if_meaning:w #1 \c_empty_clist
5936     \tl_set:Nn #2 { \q_no_value }
5937     \else:
5938     \exp_after:wN \__clist_get:wN #1 , \q_stop #2
5939     \fi:
5940 }
5941 \cs_new_protected:Npn \__clist_get:wN #1 , #2 \q_stop #3
5942 { \tl_set:Nn #3 {#1} }
5943 \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.

```

5944 \cs_new_protected_nopar:Npn \clist_pop:NN
5945 { \__clist_pop:NNN \tl_set:Nx }
5946 \cs_new_protected_nopar:Npn \clist_gpop:NN
5947 { \__clist_pop:NNN \tl_gset:Nx }
5948 \cs_new_protected:Npn \__clist_pop:NNN #1#2#3
5949 {
5950     \if_meaning:w #2 \c_empty_clist
5951     \tl_set:Nn #3 { \q_no_value }
5952     \else:
5953     \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3

```

```

5954     \fi:
5955   }
5956   \cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \q_stop #3#4#5
5957   {
5958     \tl_set:Nn #5 {#1}
5959     #3 #4
5960     {
5961       \__clist_pop:wN \prg_do_nothing:
5962       #2 \exp_not:o
5963       , \q_mark \use_none:n
5964       \q_stop
5965     }
5966   }
5967   \cs_new:Npn \__clist_pop:wN #1 , \q_mark #2 #3 \q_stop { #2 {#1} }
5968   \cs_generate_variant:Nn \clist_pop:NN { c }
5969   \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 5970 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }

\clist_pop:NNTF 5971 {

\clist_pop:cNTF 5972 \if_meaning:w #1 \c_empty_clist

\clist_gpop:NNTF 5973 \prg_return_false:

\clist_gpop:cNTF 5974 \else:

__clist_pop_TF:NNN 5975 \exp_after:wN __clist_get:wN #1 , \q_stop #2

5976 \prg_return_true:

5977 \fi:

5978 }

5979 \cs_generate_variant:Nn \clist_get:NNT { c }

5980 \cs_generate_variant:Nn \clist_get:NNF { c }

5981 \cs_generate_variant:Nn \clist_get:NNTF { c }

5982 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }

5983 { __clist_pop_TF:NNN \tl_set:Nx #1 #2 }

5984 \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }

5985 { __clist_pop_TF:NNN \tl_gset:Nx #1 #2 }

5986 \cs_new_protected:Npn __clist_pop_TF:NNN #1#2#3

5987 {

5988 \if_meaning:w #2 \c_empty_clist

5989 \prg_return_false:

5990 \else:

5991 \exp_after:wN __clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3

5992 \prg_return_true:

5993 \fi:

5994 }

5995 \cs_generate_variant:Nn \clist_pop:NNT { c }

5996 \cs_generate_variant:Nn \clist_pop:NNF { c }

5997 \cs_generate_variant:Nn \clist_pop:NNTF { c }

5998 \cs_generate_variant:Nn \clist_gpop:NNT { c }

5999 \cs_generate_variant:Nn \clist_gpop:NNF { c }

6000 \cs_generate_variant:Nn \clist_gpop:NNTF { c }

(End definition for `\clist_get:NN` and `\clist_get:cN`. These functions are documented on page ??.)

`\clist_push:Nn` Pushing to a comma list is the same as adding on the left.

<code>\clist_push:NV</code>	6001	<code>\cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn</code>
<code>\clist_push:No</code>	6002	<code>\cs_new_eq:NN \clist_push:NV \clist_put_left:NV</code>
<code>\clist_push:Nx</code>	6003	<code>\cs_new_eq:NN \clist_push:No \clist_put_left:No</code>
<code>\clist_push:cn</code>	6004	<code>\cs_new_eq:NN \clist_push:Nx \clist_put_left:Nx</code>
<code>\clist_push:cV</code>	6005	<code>\cs_new_eq:NN \clist_push:cn \clist_put_left:cn</code>
<code>\clist_push:co</code>	6006	<code>\cs_new_eq:NN \clist_push:cV \clist_put_left:cV</code>
<code>\clist_push:cx</code>	6007	<code>\cs_new_eq:NN \clist_push:co \clist_put_left:co</code>
<code>\clist_gpush:Nn</code>	6008	<code>\cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn</code>
<code>\clist_gpush:NV</code>	6009	<code>\cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV</code>
<code>\clist_gpush:No</code>	6010	<code>\cs_new_eq:NN \clist_gpush:No \clist_gput_left:No</code>
<code>\clist_gpush:Nx</code>	6011	<code>\cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx</code>
<code>\clist_gpush:cn</code>	6012	<code>\cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn</code>
<code>\clist_gpush:cV</code>	6013	<code>\cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV</code>
<code>\clist_gpush:co</code>	6014	<code>\cs_new_eq:NN \clist_gpush:co \clist_gput_left:co</code>
<code>\clist_gpush:cx</code>	6015	<code>\cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx</code>

(End definition for `\clist_push:Nn` and others. These functions are documented on page ??.)

12.5 Modifying comma lists

`\l__clist_internal_remove_clist` An internal comma list for the removal routines.

6017 `\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.

<code>\clist_remove_duplicates:c</code>	6018	<code>\cs_new_protected:Npn \clist_remove_duplicates:N</code>
<code>\clist_gremove_duplicates:N</code>	6019	<code>{ __clist_remove_duplicates:NN \clist_set_eq:NN }</code>
<code>\clist_gremove_duplicates:c</code>	6020	<code>\cs_new_protected:Npn \clist_gremove_duplicates:N</code>
<code>__clist_remove_duplicates:NN</code>	6021	<code>{ __clist_remove_duplicates:NN \clist_gset_eq:NN }</code>
	6022	<code>\cs_new_protected:Npn __clist_remove_duplicates:NN #1#2</code>
	6023	<code>{</code>
	6024	<code>\clist_clear:N \l__clist_internal_remove_clist</code>
	6025	<code>\clist_map_inline:Nn #2</code>
	6026	<code>{</code>
	6027	<code>\clist_if_in:NnF \l__clist_internal_remove_clist {##1}</code>
	6028	<code>{ \clist_put_right:Nn \l__clist_internal_remove_clist {##1} }</code>
	6029	<code>}</code>
	6030	<code>#1 #2 \l__clist_internal_remove_clist</code>
	6031	<code>}</code>
	6032	<code>\cs_generate_variant:Nn \clist_remove_duplicates:N { c }</code>
	6033	<code>\cs_generate_variant:Nn \clist_gremove_duplicates:N { c }</code>

(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.

```

6034 \cs_new_protected:Npn \clist_remove_all:Nn
6035 { \__clist_remove_all:NNn \tl_set:Nx }
6036 \cs_new_protected:Npn \clist_gremove_all:Nn
6037 { \__clist_remove_all:NNn \tl_gset:Nx }
6038 \cs_new_protected:Npn \__clist_remove_all:NNn #1#2#3
6039 {
6040   \cs_set:Npn \__clist_tmp:w ##1 , #3 ,
6041   {
6042     ##1
6043     , \q_mark , \use_none_delimit_by_q_stop:w ,
6044     \__clist_remove_all:
6045   }
6046   #1 #2
6047   {
6048     \exp_after:wN \__clist_remove_all:
6049     #2 , \q_mark , #3 , \q_stop
6050   }
6051   \clist_if_empty:NF #2
6052   {
6053     #1 #2
6054     {
6055       \exp_args:No \exp_not:o
6056       { \exp_after:wN \use_none:n #2 }
6057     }
6058   }
6059 }
6060 \cs_new:Npn \__clist_remove_all:
6061 { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
6062 \cs_new:Npn \__clist_remove_all:w #1 , \q_mark , #2 , { \exp_not:n {#1} }
6063 \cs_generate_variant:Nn \clist_remove_all:Nn { c }
6064 \cs_generate_variant:Nn \clist_gremove_all:Nn { c }

```

(End definition for `\clist_remove_all:Nn` and `\clist_remove_all:cn`. These functions are documented on page ??.)

12.6 Comma list conditionals

`\clist_if_empty_p:N` Simple copies from the token list variable material.
`\clist_if_empty_p:c` 6065 `\prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N { p , T , F , TF }`
`\clist_if_empty:NTF` 6066 `\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_in:NnTF` See description of the `\tl_if_in:Nn` function for details. We simply surround the comma list, and the item, with commas.
`\clist_if_in:NVTF`
`\clist_if_in:NoTF` 6067 `\prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }`
`\clist_if_in:cnTF` 6068 `{`
`\clist_if_in:cVTF` 6069 `\exp_args:No __clist_if_in_return:nn #1 {#2}`
`\clist_if_in:coTF` 6070 `}`
`\clist_if_in:nnTF` 6071 `\prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T , F , TF }`
`\clist_if_in:nVTF` 6072 `{`
`\clist_if_in:noTF` 6073 `\clist_set:Nn \l__clist_internal_clist {#1}`
`__clist_if_in_return:nn` 6074 `\exp_args:No __clist_if_in_return:nn \l__clist_internal_clist {#2}`
6075 `}`
6076 `\cs_new_protected:Npn __clist_if_in_return:nn #1#2`
6077 `{`
6078 `\cs_set:Npn __clist_tmp:w ##1 ,#2, { }`
6079 `\tl_if_empty:oTF`
6080 `{ __clist_tmp:w ,#1, {} {} ,#2, }`
6081 `{ \prg_return_false: } { \prg_return_true: }`
6082 `}`
6083 `\cs_generate_variant:Nn \clist_if_in:NnT { NV , No }`
6084 `\cs_generate_variant:Nn \clist_if_in:NnT { c , cV , co }`
6085 `\cs_generate_variant:Nn \clist_if_in:NnF { NV , No }`
6086 `\cs_generate_variant:Nn \clist_if_in:NnF { c , cV , co }`
6087 `\cs_generate_variant:Nn \clist_if_in:NnTF { NV , No }`
6088 `\cs_generate_variant:Nn \clist_if_in:NnTF { c , cV , co }`
6089 `\cs_generate_variant:Nn \clist_if_in:nnT { nV , no }`
6090 `\cs_generate_variant:Nn \clist_if_in:nnF { nV , no }`
6091 `\cs_generate_variant:Nn \clist_if_in:nnTF { nV , no }`
(End definition for `\clist_if_in:Nn` and others. These functions are documented on page ??.)

12.7 Mapping to comma lists

`\clist_map_function:NN` If the variable is empty, the mapping is skipped (otherwise, that comma-list would be
`\clist_map_function:cN` seen as consisting of one empty item). Then loop over the comma-list, grabbing one
`__clist_map_function:Nw` comma-delimited item at a time. The end is marked by `\q_recursion_tail`. The auxiliary function `__clist_map_function:Nw` is used directly in `\clist_map_inline:Nn`. Change with care.

```
6092 \cs_new:Npn \clist_map_function:NN #1#2
6093 {
6094   \clist_if_empty:NF #1
6095   {
6096     \exp_last_unbraced:NNo \__clist_map_function:Nw #2 #1
```

```

6097         , \q_recursion_tail ,
6098         \__prg_break_point:Nn \clist_map_break: { }
6099     }
6100 }
6101 \cs_new:Npn \__clist_map_function:Nw #1#2 ,
6102 {
6103     \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
6104     #1 {#2}
6105     \__clist_map_function:Nw #1
6106 }
6107 \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
__clist_map_function_n:Nn
__clist_map_unbrace:Nw

The n-type mapping function is a bit more awkward, since spaces must be trimmed from each item. Space trimming is again based on __clist_trim_spaces_generic:nw. The auxiliary __clist_map_function_n:Nn receives as arguments the function, and the result of removing leading and trailing spaces from the item which lies until the next comma. Empty items are ignored, then one level of braces is removed by __clist_map_unbrace:Nw.

```

6108 \cs_new:Npn \clist_map_function:nN #1#2
6109 {
6110     \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #2 }
6111     \q_mark #1, \q_recursion_tail,
6112     \__prg_break_point:Nn \clist_map_break: { }
6113 }
6114 \cs_new:Npn \__clist_map_function_n:Nn #1 #2
6115 {
6116     \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
6117     \tl_if_empty:nF {#2} { \__clist_map_unbrace:Nw #1 #2, }
6118     \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #1 }
6119     \q_mark
6120 }
6121 \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
\clist_map_inline:cn
\clist_map_inline:nN

Inline mapping is done by creating a suitable function “on the fly”: this is done globally to avoid any issues with TeX’s groups. We use a different function for each level of nesting.

Since the mapping is non-expandable, we can perform the space-trimming needed by the n version simply by storing the comma-list in a variable. We don’t need a different comma-list for each nesting level: the comma-list is expanded before the mapping starts.

```

6122 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
6123 {
6124     \clist_if_empty:NF #1
6125     {
6126         \int_gincr:N \g__prg_map_int
6127         \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}

```

```

6128         \exp_last_unbraced:Nco \__clist_map_function:Nw
6129         { __prg_map_ \int_use:N \g__prg_map_int :w }
6130         #1 , \q_recursion_tail ,
6131         \__prg_break_point:Nn \clist_map_break:
6132         { \int_gdecr:N \g__prg_map_int }
6133     }
6134 }
6135 \cs_new_protected:Npn \clist_map_inline:nn #1
6136 {
6137     \clist_set:Nn \l__clist_internal_clist {#1}
6138     \clist_map_inline:Nn \l__clist_internal_clist
6139 }
6140 \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
\clist_map_variable:cNn
\clist_map_variable:nNn
__clist_map_variable:Nnw

As for other comma-list mappings, filter out the case of an empty list. Same approach as \clist_map_function:Nn, additionally we store each item in the given variable. As for inline mappings, space trimming for the n variant is done by storing the comma list in a variable.

```

6141 \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
6142 {
6143     \clist_if_empty:NF #1
6144     {
6145         \exp_args:Nno \use:nn
6146         { \__clist_map_variable:Nnw #2 {#3} }
6147         #1
6148         , \q_recursion_tail , \q_recursion_stop
6149         \__prg_break_point:Nn \clist_map_break: { }
6150     }
6151 }
6152 \cs_new_protected:Npn \clist_map_variable:nNn #1
6153 {
6154     \clist_set:Nn \l__clist_internal_clist {#1}
6155     \clist_map_variable:NNn \l__clist_internal_clist
6156 }
6157 \cs_new_protected:Npn \__clist_map_variable:Nnw #1#2#3,
6158 {
6159     \tl_set:Nn #1 {#3}
6160     \quark_if_recursion_tail_stop:N #1
6161     \use:n {#2}
6162     \__clist_map_variable:Nnw #1 {#2}
6163 }
6164 \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:
\clist_map_break:n

The break statements use the general __prg_map_break:Nn mechanism.

```

6165 \cs_new_nopar:Npn \clist_map_break:

```

```

6166 { \prg_map_break:Nn \clist_map_break: { } }
6167 \cs_new_nopar:Npn \clist_map_break:n
6168 { \prg_map_break:Nn \clist_map_break: }

```

(End definition for `\clist_map_break:` and `\clist_map_break:n`. These functions are documented on page 118.)

`\clist_count:N` Counting the items in a comma list is done using the same approach as for other token count functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics. In the case of an `n`-type comma-list, we could of course use `\clist_map_function:nN`, but that is very slow, because it carefully removes spaces. Instead, we loop manually, and skip blank items (but not `{}`, hence the extra spaces).

```

6169 \cs_new:Npn \clist_count:N #1
6170 {
6171   \int_eval:n
6172   {
6173     0
6174     \clist_map_function:NN #1 \__clist_count:n
6175   }
6176 }
6177 \cs_generate_variant:Nn \clist_count:N { c }
6178 \cs_new:Npx \clist_count:n #1
6179 {
6180   \exp_not:N \int_eval:n
6181   {
6182     0
6183     \exp_not:N \__clist_count:w \c_space_tl
6184     #1 \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
6185   }
6186 }
6187 \cs_new:Npn \__clist_count:n #1 { + \c_one }
6188 \cs_new:Npx \__clist_count:w #1 ,
6189 {
6190   \exp_not:n { \exp_args:Nf \quark_if_recursion_tail_stop:n } {#1}
6191   \exp_not:N \tl_if_blank:nF {#1} { + \c_one }
6192   \exp_not:N \__clist_count:w \c_space_tl
6193 }

```

(End definition for `\clist_count:N`, `\clist_count:c`, and `\clist_count:n`. These functions are documented on page ??.)

12.8 Using comma lists

`\clist_use:Nnnn` First check that the variable exists. Then count the items in the comma list. If it has none, output nothing. If it has one item, output that item, brace stripped (note that space-trimming has already been done when the comma list was assigned). If it has two, place the *<separator between two>* in the middle.

`\clist_use:cn` 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 `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*.

```

6194 \cs_new:Npn \clist_use:Nnnn #1#2#3#4
6195 {
6196   \clist_if_exist:NTF #1
6197   {
6198     \int_case:nnn { \clist_count:N #1 }
6199     {
6200       { 0 } { }
6201       { 1 } { \exp_after:wN \__clist_use:wnn #1 , , { } }
6202       { 2 } { \exp_after:wN \__clist_use:wnn #1 , {#2} }
6203     }
6204     {
6205       \exp_after:wN \__clist_use:nwwwnwn
6206       \exp_after:wN { \exp_after:wN } #1 ,
6207       \q_mark , { \__clist_use:nwwwnwn {#3} }
6208       \q_mark , { \__clist_use:nwnn {#4} }
6209       \q_stop { }
6210     }
6211   }
6212   { \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#1} }
6213 }
6214 \cs_generate_variant:Nn \clist_use:Nnnn { c }
6215 \cs_new:Npn \__clist_use:wnn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
6216 \cs_new:Npn \__clist_use:nwwwnwn
6217   #1#2 , #3 , #4 , #5 \q_mark , #6#7 \q_stop #8
6218   { #6 {#3} , {#4} , #5 \q_mark , {#6} #7 \q_stop { #8 #1 #2 } }
6219 \cs_new:Npn \__clist_use:nwnn #1#2 , #3 \q_stop #4
6220   { \exp_not:n { #4 #1 #2 } }
6221 \cs_new:Npn \clist_use:Nn #1#2
6222   { \clist_use:Nnnn #1 {#2} {#2} {#2} }
6223 \cs_generate_variant:Nn \clist_use:Nn { c }

```

(End definition for `\clist_use:Nnnn` and `\clist_use:cnnn`. These functions are documented on page ??.)

12.9 Viewing comma lists

`\clist_show:N` Apply the general `__msg_show_variable:Nnn`. In the case of an n-type comma-list, **`\clist_show:c`** first store it in a scratch variable, then show that variable: The message takes care of omitting its name.

```

6224 \cs_new_protected:Npn \clist_show:N #1
6225 {

```

```

6226     \_msg_show_variable:Nnn #1 { clist }
6227     { \clist_map_function:NN #1 \_msg_show_item:n }
6228   }
6229   \cs_new_protected:Npn \clist_show:n #1
6230   {
6231     \clist_set:Nn \l__clist_internal_clist {#1}
6232     \clist_show:N \l__clist_internal_clist
6233   }
6234   \cs_generate_variant:Nn \clist_show:N { c }

```

(End definition for \clist_show:N and \clist_show:c. These functions are documented on page 121.)

12.10 Scratch comma lists

`\l_tmpa_clist` Temporary comma list variables.

```

\l_tmpb_clist 6235 \clist_new:N \l_tmpa_clist
\g_tmpa_clist 6236 \clist_new:N \l_tmpb_clist
\g_tmpb_clist 6237 \clist_new:N \g_tmpa_clist
               6238 \clist_new:N \g_tmpb_clist

```

(End definition for \l_tmpa_clist and \l_tmpb_clist. These functions are documented on page 121.)

12.11 Deprecated interfaces

Deprecated on 2011-05-27, for removal by 2011-08-31.

`\clist_top:NN` These are old stack functions.

```

\clist_top:cN 6239 {*deprecated}
               6240 \cs_new_eq:NN \clist_top:NN \clist_get:NN
               6241 \cs_new_eq:NN \clist_top:cN \clist_get:cN
               6242 {/deprecated}

```

(End definition for \clist_top:NN and \clist_top:cN. These functions are documented on page ??.)

`\clist_remove_element:Nn` An older name for `\clist_remove_all:Nn`.

```

\clist_gremove_element:Nn 6243 {*deprecated}
                          6244 \cs_new_eq:NN \clist_remove_element:Nn \clist_remove_all:Nn
                          6245 \cs_new_eq:NN \clist_gremove_element:Nn \clist_gremove_all:Nn
                          6246 {/deprecated}

```

(End definition for \clist_remove_element:Nn and \clist_gremove_element:Nn. These functions are documented on page ??.)

`\clist_display:N` An older name for `\clist_show:N`.

```

\clist_display:c 6247 {*deprecated}
                 6248 \cs_new_eq:NN \clist_display:N \clist_show:N
                 6249 \cs_new_eq:NN \clist_display:c \clist_show:c
                 6250 {/deprecated}

```

(End definition for \clist_display:N and \clist_display:c. These functions are documented on page ??.)

Deprecated on 2011-09-05, for removal by 2011-12-31.

`\clist_trim_spaces:N` Since clist items are now always stripped from their surrounding spaces, it is redundant to provide these functions. The `_clist_trim_spaces:n` function is now internal, deprecated for use outside the kernel.

```

\clist_trim_spaces:c
\clist_gtrim_spaces:N
\clist_gtrim_spaces:c
6251 < *deprecated >
6252 \cs_new_protected:Npn \clist_trim_spaces:N #1 { \clist_set:No #1 {#1} }
6253 \cs_new_protected:Npn \clist_gtrim_spaces:N #1 { \clist_gset:No #1 {#1} }
6254 \cs_generate_variant:Nn \clist_trim_spaces:N { c }
6255 \cs_generate_variant:Nn \clist_gtrim_spaces:N { c }
6256 < /deprecated >

```

(End definition for `\clist_trim_spaces:N` and others. These functions are documented on page ??.)

Deprecated on 2012-05-10, for removal by 2012-08-31.

`\clist_if_eq_p:NN` Simple copies from the token list variable material.

```

\clist_if_eq_p:Nc
\clist_if_eq_p:cN
\clist_if_eq_p:cc
\clist_if_eq:NNTF
\clist_if_eq:NcTF
\clist_if_eq:cNTF
\clist_if_eq:ccTF
6257 < *deprecated >
6258 \prg_new_eq_conditional:NNn \clist_if_eq:NN \tl_if_eq:NN { p , T , F , TF }
6259 \prg_new_eq_conditional:NNn \clist_if_eq:Nc \tl_if_eq:Nc { p , T , F , TF }
6260 \prg_new_eq_conditional:NNn \clist_if_eq:cN \tl_if_eq:cN { p , T , F , TF }
6261 \prg_new_eq_conditional:NNn \clist_if_eq:cc \tl_if_eq:cc { p , T , F , TF }
6262 < /deprecated >

```

(End definition for `\clist_if_eq:NN` and others. These functions are documented on page ??.)

Deprecated 2012-05-13 for removal by 2012-11-31.

`\clist_length:N`
`\clist_length:c`
`\clist_length:n`

```

6263 < *deprecated >
6264 \cs_new_eq:NN \clist_length:N \clist_count:N
6265 \cs_new_eq:NN \clist_length:n \clist_count:c
6266 \cs_new_eq:NN \clist_length:c \clist_count:n
6267 < /deprecated >

```

(End definition for `\clist_length:N`, `\clist_length:c`, and `\clist_length:n`. These functions are documented on page ??.)

Deprecated 2012-05-19 for removal by 2012-11-31.

`\clist_use:N`
`\clist_use:c`

```

6268 < *deprecated >
6269 \cs_new_eq:NN \clist_use:N \tl_use:N
6270 \cs_new_eq:NN \clist_use:c \tl_use:c
6271 < /deprecated >

```

(End definition for `\clist_use:N` and `\clist_use:c`. These functions are documented on page ??.)

```

6272 < /initex | package >

```


13 l3prop implementation

The following test files are used for this code: `m3prop001`, `m3prop002`, `m3prop003`, `m3prop004`, `m3show001`.

```

6273 <*initex | package>
6274 <@@=prop>
6275 <*package>
6276 \ProvidesExplPackage
6277   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
6278   \__expl_package_check:
6279 </package>

```

A property list is a macro whose top-level expansion is for the form

```

\s__prop <key1> \s__prop {<value1>}
...
\s__prop <keyn> \s__prop {<valuen>}

```

\s__prop A private scan mark is used as a marker surrounding each key.

```

6280 \__scan_new:N \s__prop
(End definition for \s__prop.)

```

\l__prop_internal_tl Token list used to store the new key–value pair inserted by `\prop_put:Nnn` and friends.

```

6281 \tl_new:N \l__prop_internal_tl
(End definition for \l__prop_internal_tl. This variable is documented on page 127.)

```

\c_empty_prop An empty prop is an empty token list.

```

6282 \cs_new_eq:NN \c_empty_prop \c_empty_tl
(End definition for \c_empty_prop. This variable is documented on page 127.)

```

13.1 Allocation and initialisation

\prop_new:N Internally, property lists are just token lists.

```

\prop_new:c
6283 \cs_new_eq:NN \prop_new:N \tl_new:N
6284 \cs_new_eq:NN \prop_new:c \tl_new: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
6285 \cs_new_eq:NN \prop_clear:N \tl_clear:N
\prop_gclear:N
6286 \cs_new_eq:NN \prop_clear:c \tl_clear:c
\prop_gclear:c
6287 \cs_new_eq:NN \prop_gclear:N \tl_gclear:N
6288 \cs_new_eq:NN \prop_gclear:c \tl_gclear:c
(End definition for \prop_clear:N and \prop_clear:c. These functions are documented on page ??.)

```

```

\prop_clear_new:N
\prop_clear_new:c
\prop_gclear_new:N
\prop_gclear_new:c

```

Once again a simple copy from the token list functions.

```

6289 \cs_new_eq:NN \prop_clear_new:N \tl_clear_new:N
6290 \cs_new_eq:NN \prop_clear_new:c \tl_clear_new:c
6291 \cs_new_eq:NN \prop_gclear_new:N \tl_gclear_new:N
6292 \cs_new_eq:NN \prop_gclear_new:c \tl_gclear_new:c

```

(End definition for `\prop_clear_new:N` and `\prop_clear_new:c`. These functions are documented on page ??.)

```

\prop_set_eq:NN
\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

```

Once again, these are simply copies from the token list functions.

```

6293 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
6294 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
6295 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
6296 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
6297 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
6298 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
6299 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
6300 \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
\l_tmpb_prop
\g_tmpa_prop
\g_tmpb_prop

```

We can now initialize the scratch variables.

```

6301 \prop_new:N \l_tmpa_prop
6302 \prop_new:N \l_tmpb_prop
6303 \prop_new:N \g_tmpa_prop
6304 \prop_new:N \g_tmpb_prop

```

(End definition for `\l_tmpa_prop` and `\l_tmpb_prop`. These functions are documented on page 127.)

13.2 Accessing data in property lists

```

\__prop_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 \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.

If the $\langle \textit{key} \rangle$ is not there, then the $\langle \textit{function} \rangle$ is `\use_ii:nn`, which keeps the $\langle \textit{false code} \rangle$.

```

6305 \cs_new_protected:Npn \__prop_split:NnTF #1#2
6306 { \exp_args:NNo \__prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }

```

```

6307 \cs_new_protected:Npn \__prop_split_aux:NnTF #1#2#3#4
6308 {
6309   \cs_set:Npn \__prop_split_aux:w ##1
6310     \s__prop #2 \s__prop ##2 ##3 \q_mark ##4 ##5 \q_stop
6311     { ##4 {#3} {#4} }
6312   \exp_after:wN \__prop_split_aux:w #1 \q_mark \use_i:nn
6313     \s__prop #2 \s__prop { } \q_mark \use_ii:nn \q_stop
6314 }
6315 \cs_new:Npn \__prop_split_aux:w { }

```

(End definition for __prop_split:NnTF. This function is documented on page 127.)

\prop_remove:Nn Deleting from a property starts by splitting the list. If the key is present in the property list, the returned value is ignored. If the key is missing, nothing happens.

```

\prop_remove:NV
\prop_remove:cn
\prop_remove:cV
\prop_gremove:Nn
\prop_gremove:NV
\prop_gremove:cn
\prop_gremove:cV
6316 \cs_new_protected:Npn \prop_remove:Nn #1#2
6317 {
6318   \__prop_split:NnTF #1 {#2}
6319   { \tl_set:Nn #1 { ##1 ##3 } }
6320   { }
6321 }
6322 \cs_new_protected:Npn \prop_gremove:Nn #1#2
6323 {
6324   \__prop_split:NnTF #1 {#2}
6325   { \tl_gset:Nn #1 { ##1 ##3 } }
6326   { }
6327 }
6328 \cs_generate_variant:Nn \prop_remove:Nn { NV }
6329 \cs_generate_variant:Nn \prop_remove:Nn { c , cV }
6330 \cs_generate_variant:Nn \prop_gremove:Nn { NV }
6331 \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
6332 \cs_new_protected:Npn \prop_get:NnN #1#2#3
6333 {
6334   \__prop_split:NnTF #1 {#2}
6335   { \tl_set:Nn #3 {##2} }
6336   { \tl_set:Nn #3 { \q_no_value } }
6337 }
6338 \cs_generate_variant:Nn \prop_get:NnN { NV , No }
6339 \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
6340 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
\prop_gpop:NnN
6341 {
\prop_gpop:NoN
6342   \__prop_split:NnTF #1 {#2}
\prop_gpop:cnN
\prop_gpop:coN

```

```

6343     {
6344         \tl_set:Nn #3 {##2}
6345         \tl_set:Nn #1 { ##1 ##3 }
6346     }
6347     { \tl_set:Nn #3 { \q_no_value } }
6348 }
6349 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
6350 {
6351     \__prop_split:NnTF #1 {#2}
6352     {
6353         \tl_set:Nn #3 {##2}
6354         \tl_gset:Nn #1 { ##1 ##3 }
6355     }
6356     { \tl_set:Nn #3 { \q_no_value } }
6357 }
6358 \cs_generate_variant:Nn \prop_pop:NnN { No }
6359 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
6360 \cs_generate_variant:Nn \prop_gpop:NnN { No }
6361 \cs_generate_variant:Nn \prop_gpop:NnN { c , co }

```

(End definition for \prop_pop:NnN and others. These functions are documented on page ??.)

\prop_pop:NnTF Popping an item from a property list, keeping track of whether the key was present or not, is implemented as a conditional. If the key was missing, neither the property list, nor the token list are altered. Otherwise, \prg_return_true: is used after the assignments.

\prop_pop:cnNTF

\prop_gpop:NnN

\prop_gpop:cnNTF

```

6362 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
6363 {
6364     \__prop_split:NnTF #1 {#2}
6365     {
6366         \tl_set:Nn #3 {##2}
6367         \tl_set:Nn #1 { ##1 ##3 }
6368         \prg_return_true:
6369     }
6370     { \prg_return_false: }
6371 }
6372 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
6373 {
6374     \__prop_split:NnTF #1 {#2}
6375     {
6376         \tl_set:Nn #3 {##2}
6377         \tl_gset:Nn #1 { ##1 ##3 }
6378         \prg_return_true:
6379     }
6380     { \prg_return_false: }
6381 }
6382 \cs_generate_variant:Nn \prop_pop:NnNT { c }
6383 \cs_generate_variant:Nn \prop_pop:NnNF { c }
6384 \cs_generate_variant:Nn \prop_pop:NnNTF { c }
6385 \cs_generate_variant:Nn \prop_gpop:NnNT { c }
6386 \cs_generate_variant:Nn \prop_gpop:NnNF { c }

```

6387 \cs_generate_variant:Nn \prop_gpop:NnNTF { c }
 (End definition for \prop_pop:NnNTF and others. These functions are documented on page ??.)

\prop_put:Nnn Since the branches of __prop_split:NnTF are used as the replacement text of an internal macro, and since the *<key>* and new *<value>* may contain arbitrary tokens, it is not safe to include them in the argument of __prop_split:NnTF. We thus start by storing in \l__prop_internal_tl tokens which (after x-expansion) encode the key–value pair. This variable can safely be used in __prop_split:NnTF. If the *<key>* was absent, append the new key–value to the list. Otherwise concatenate the extracts ##1 and ##3 with the new key–value pair \l__prop_internal_tl. The updated entry is placed at the same spot as the original *<key>* in the property list, preserving the order of entries.

```
6388 \cs_new_protected_nopar:Npn \prop_put:Nnn
6389 { \__prop_put:NNNnn \tl_set:Nx \tl_put_right:Nx }
6390 \cs_new_protected_nopar:Npn \prop_gput:Nnn
6391 { \__prop_put:NNNnn \tl_gset:Nx \tl_gput_right:Nx }
6392 \cs_new_protected:Npn \__prop_put:NNNnn #1#2#3#4#5
6393 {
6394   \tl_set:Nn \l__prop_internal_tl
6395     { \s__prop \tl_to_str:n {#4} \s__prop { \exp_not:n {#5} } }
6396   \__prop_split:NnTF #3 {#4}
6397     { #1 #3 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } }
6398     { #2 #3 { \l__prop_internal_tl } }
6399 }
6400 \cs_generate_variant:Nn \prop_put:Nnn
6401 { NnV , Nno , Nnx , NV , NVV , No , Noo }
6402 \cs_generate_variant:Nn \prop_gput:Nnn
6403 { c , cnV , cno , cnx , cV , cVV , co , coo }
6404 \cs_generate_variant:Nn \prop_gput:Nnn
6405 { NnV , Nno , Nnx , NV , NVV , No , Noo }
6406 \cs_generate_variant:Nn \prop_gput:Nnn
6407 { 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.

```
6408 \cs_new_protected_nopar:Npn \prop_put_if_new:Nnn
6409 { \__prop_put_if_new:NNnn \tl_put_right:Nx }
6410 \cs_new_protected_nopar:Npn \prop_gput_if_new:Nnn
6411 { \__prop_put_if_new:NNnn \tl_gput_right:Nx }
6412 \cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4
6413 {
6414   \tl_set:Nn \l__prop_internal_tl
6415     { \s__prop \tl_to_str:n {#3} \s__prop \exp_not:n { {#4} } }
6416   \__prop_split:NnTF #2 {#3}
6417     { }
6418     { #1 #2 { \l__prop_internal_tl } }
6419 }
```

6420 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }

6421 \cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }

(End definition for \prop_put_if_new:Nnn and \prop_gput_if_new:cnn. These functions are documented on page ??.)

13.3 Property list conditionals

\prop_if_exist_p:N

Copies of the cs functions defined in l3basics.

\prop_if_exist_p:c

6422 \prg_new_eq_conditional:NNn \prop_if_exist:N \cs_if_exist:N { TF , T , F , p }

\prop_if_exist:N \overline{TF}

6423 \prg_new_eq_conditional:NNn \prop_if_exist:c \cs_if_exist:c { TF , T , F , p }

\prop_if_exist:c \overline{TF}

(End definition for \prop_if_exist:N and \prop_if_exist:c. These functions are documented on page ??.)

\prop_if_empty_p:N

Same test as for token lists.

\prop_if_empty_p:c

6424 \prg_new_eq_conditional:NNn \prop_if_empty:N \tl_if_empty:N

\prop_if_empty:N \overline{TF}

6425 { p , T , F , TF }

\prop_if_empty:c \overline{TF}

6426 \prg_new_eq_conditional:NNn \prop_if_empty:c \tl_if_empty:c

6427 { p , T , F , TF }

(End definition for \prop_if_empty:N and \prop_if_empty:c. These functions are documented on page ??.)

\prop_if_in_p:Nn

Testing expandably if a key is in a property list requires to go through the key–value pairs one by one. This is rather slow, and a faster test would be

\prop_if_in_p:Nv

\prop_if_in_p:No

\prop_if_in_p:cn

\prop_if_in_p:cV

\prop_if_in_p:co

\prop_if_in:Nn \overline{TF}

\prop_if_in:Nv \overline{TF}

\prop_if_in:No \overline{TF}

\prop_if_in:cn \overline{TF}

\prop_if_in:cV \overline{TF}

\prop_if_in:co \overline{TF}

__prop_if_in:nwn

__prop_if_in:N

```
\prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
{
  \@@_split:NnTF #1 {#2}
  {
    \prg_return_true:
    \use_none:nnn
  }
  { \prg_return_false: }
}
```

but __prop_split:NnTF is non-expandable.

Instead, the key is compared to each key in turn using \str_if_eq_x:nn, which is expandable. To terminate the mapping, we append to the property list the key that is searched for. This second \tl_to_str:n is not expanded at the start, but only when included in the \str_if_eq_x:nn. It cannot make the breaking mechanism choke, because the arbitrary token list material is enclosed in braces. When ending, we test the next token: it is either \s__prop or \q_recursion_tail in the case of a missing key. 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.

6428 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }

6429 {

6430 \exp_last_unbraced:Noo __prop_if_in:nwn

6431 { \tl_to_str:n {#2} } #1

6432 \s__prop \tl_to_str:n {#2} \s__prop { }

```

6433     \q_recursion_tail
6434     \__prg_break_point:
6435 }
6436 \cs_new:Npn \__prop_if_in:nwn #1 \s__prop #2 \s__prop #3
6437 {
6438     \str_if_eq_x:nnTF {#1} {#2}
6439     { \__prop_if_in:N }
6440     { \__prop_if_in:nwn {#1} }
6441 }
6442 \cs_new:Npn \__prop_if_in:N #1
6443 {
6444     \if_meaning:w \s__prop #1
6445     \prg_return_true:
6446     \else:
6447     \prg_return_false:
6448     \fi:
6449     \__prg_break:
6450 }
6451 \cs_generate_variant:Nn \prop_if_in_p:Nn { NV , No }
6452 \cs_generate_variant:Nn \prop_if_in_p:Nn { c , cV , co }
6453 \cs_generate_variant:Nn \prop_if_in:NnT { NV , No }
6454 \cs_generate_variant:Nn \prop_if_in:NnT { c , cV , co }
6455 \cs_generate_variant:Nn \prop_if_in:NnF { NV , No }
6456 \cs_generate_variant:Nn \prop_if_in:NnF { c , cV , co }
6457 \cs_generate_variant:Nn \prop_if_in:NnTF { NV , No }
6458 \cs_generate_variant:Nn \prop_if_in:NnTF { c , cV , co }

```

(End definition for `\prop_if_in:Nn` and others. These functions are documented on page ??.)

13.4 Recovering values from property lists with branching

`\prop_get:NnTF` Getting the value corresponding to a key, keeping track of whether the key was present or not, is implemented as a conditional (with side effects). If the key was absent, the token list is not altered.

```

\prop_get:NVNTF
\prop_get:NoNTF
\prop_get:cnNTF
\prop_get:cVNTF
\prop_get:coNTF
6459 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
6460 {
6461     \__prop_split:NnTF #1 {#2}
6462     {
6463         \tl_set:Nn #3 {##2}
6464         \prg_return_true:
6465     }
6466     { \prg_return_false: }
6467 }
6468 \cs_generate_variant:Nn \prop_get:NnNT { NV , No }
6469 \cs_generate_variant:Nn \prop_get:NnNF { NV , No }
6470 \cs_generate_variant:Nn \prop_get:NnNTF { NV , No }
6471 \cs_generate_variant:Nn \prop_get:NnNT { c , cV , co }
6472 \cs_generate_variant:Nn \prop_get:NnNF { c , cV , co }
6473 \cs_generate_variant:Nn \prop_get:NnNTF { c , cV , co }

```

(End definition for `\prop_get:NnNTF` and others. These functions are documented on page ??.)

13.5 Mapping to property lists

`\prop_map_function:NN` The fastest way to do a recursion here is to use an `\if_meaning:w` test: the keys are strings, and thus cannot match the marker `\q_recursion_tail`. A special case to note is when the key #2 is empty: then `\q_recursion_tail` is compared to `\exp_after:wN`, also different.

```

6474 \cs_new:Npn \prop_map_function:NN #1#2
6475 {
6476   \exp_last_unbraced:NNo \__prop_map_function:Nwn #2
6477   #1 \s__prop \q_recursion_tail \s__prop { }
6478   \__prg_break_point:Nn \prop_map_break: { }
6479 }
6480 \cs_new:Npn \__prop_map_function:Nwn #1 \s__prop #2 \s__prop #3
6481 {
6482   \if_meaning:w \q_recursion_tail #2
6483   \exp_after:wN \prop_map_break:
6484   \fi:
6485   #1 {#2} {#3}
6486   \__prop_map_function:Nwn #1
6487 }
6488 \cs_generate_variant:Nn \prop_map_function:NN { Nc }
6489 \cs_generate_variant:Nn \prop_map_function:NN { c , cc }

```

(End definition for `\prop_map_function:NN` and others. These functions are documented on page ??.)

`\prop_map_inline:Nn` Mapping in line requires a nesting level counter.

```

\prop_map_inline:cn
6490 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
6491 {
6492   \int_gincr:N \g__prg_map_int
6493   \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1##2 {#2}
6494   \exp_last_unbraced:Nco \__prop_map_function:Nwn
6495   { __prg_map_ \int_use:N \g__prg_map_int :w }
6496   #1
6497   \s__prop \q_recursion_tail \s__prop { }
6498   \__prg_break_point:Nn \prop_map_break: { \int_gdecr:N \g__prg_map_int }
6499 }
6500 \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
6501 \cs_new_nopar:Npn \prop_map_break:
6502 { \__prg_map_break:Nn \prop_map_break: { } }
6503 \cs_new_nopar:Npn \prop_map_break:n
6504 { \__prg_map_break:Nn \prop_map_break: }

```

(End definition for `\prop_map_break:.` This function is documented on page 126.)

13.6 Viewing property lists

\prop_show:N Apply the general `_msg_show_variable:Nnn`. Contrarily to sequences and comma lists, we use `_msg_show_item:nn` to format both the key and the value for each pair.

\prop_show:c

```
6505 \cs_new_protected:Npn \prop_show:N #1
6506 {
6507   \_msg_show_variable:Nnn #1 { prop }
6508   { \prop_map_function:NN #1 \_msg_show_item:nn }
6509 }
6510 \cs_generate_variant:Nn \prop_show:N { c }
```

(End definition for `\prop_show:N` and `\prop_show:c`. These functions are documented on page ??.)

13.7 Deprecated interfaces

Deprecated on 2011-05-27, for removal by 2011-08-31.

\prop_display:N An older name for `\prop_show:N`.

\prop_display:c

```
6511 <*deprecated>
6512 \cs_new_eq:NN \prop_display:N \prop_show:N
6513 \cs_new_eq:NN \prop_display:c \prop_show:c
6514 </deprecated>
```

(End definition for `\prop_display:N` and `\prop_display:c`. These functions are documented on page ??.)

\prop_gget:NnN Getting globally is no longer supported: this is a conceptual change, so the necessary code for the transition is provided directly.

\prop_gget:NVN

\prop_gget:cnN

\prop_gget:cVN

\prop_gget_aux:Nnnn

```
6515 <*deprecated>
6516 \tl_new:N \l__prop_internal_tl
6517 \cs_new_protected:Npn \prop_gget:NnN #1#2#3
6518 {
6519   \prop_get:NnN #1 {#2} \l__prop_internal_tl
6520   \tl_gset_eq:NN #3 \l__prop_internal_tl
6521 }
6522 \cs_generate_variant:Nn \prop_gget:NnN { NV }
6523 \cs_generate_variant:Nn \prop_gget:NnN { c , cV }
6524 </deprecated>
```

(End definition for `\prop_gget:NnN` and others. These functions are documented on page ??.)

\prop_get_gdel:NnN This name seems very odd.

```
6525 <*deprecated>
6526 \cs_new_eq:NN \prop_get_gdel:NnN \prop_gpop:NnN
6527 </deprecated>
```

(End definition for `\prop_get_gdel:NnN`. This function is documented on page ??.)

\prop_if_in:ccTF A hang-over from an ancient implementation

```
6528 <*deprecated>
6529 \cs_generate_variant:Nn \prop_if_in:NnT { cc }
6530 \cs_generate_variant:Nn \prop_if_in:NnF { cc }
6531 \cs_generate_variant:Nn \prop_if_in:NnTF { cc }
6532 </deprecated>
```

(End definition for `\prop_if_in:ccTF`. This function is documented on page ??.)

`\prop_gput:ccx` Another one.

```
6533 <*deprecated>
6534 \cs_generate_variant:Nn \prop_gput:Nnn { ccx }
6535 </deprecated>
```

(End definition for `\prop_gput:ccx`. This function is documented on page ??.)

`\prop_if_eq_p:NN` These ones do no even make sense!

```
\prop_if_eq_p:Nc 6536 <*deprecated>
\prop_if_eq_p:cN 6537 \prg_new_eq_conditional:NNn \prop_if_eq:NN \tl_if_eq:NN { p , T , F , TF }
\prop_if_eq_p:cc 6538 \prg_new_eq_conditional:NNn \prop_if_eq:cN \tl_if_eq:cN { p , T , F , TF }
\prop_if_eq:NNTF 6539 \prg_new_eq_conditional:NNn \prop_if_eq:Nc \tl_if_eq:Nc { p , T , F , TF }
\prop_if_eq:NcTF 6540 \prg_new_eq_conditional:NNn \prop_if_eq:cc \tl_if_eq:cc { p , T , F , TF }
\prop_if_eq:cNTF 6541 </deprecated>
```

`\prop_if_eq:ccTF` (End definition for `\prop_if_eq:NN` and others. These functions are documented on page ??.)

Deprecated on 2012-05-12, for removal by 2012-11-30.

```
\prop_del:Nn
\prop_del:NV 6542 <*deprecated>
\prop_del:cn 6543 \cs_new_eq:NN \prop_del:Nn \prop_remove:Nn
\prop_del:cV 6544 \cs_new_eq:NN \prop_del:NV \prop_remove:NV
\prop_gdel:Nn 6545 \cs_new_eq:NN \prop_del:cn \prop_remove:cn
\prop_gdel:NV 6546 \cs_new_eq:NN \prop_del:cV \prop_remove:cV
\prop_gdel:cn 6547 \cs_new_eq:NN \prop_gdel:Nn \prop_gremove:Nn
\prop_gdel:cV 6548 \cs_new_eq:NN \prop_gdel:NV \prop_gremove:NV
6549 \cs_new_eq:NN \prop_gdel:cn \prop_gremove:cn
6550 \cs_new_eq:NN \prop_gdel:cV \prop_gremove:cV
6551 </deprecated>
```

(End definition for `\prop_del:Nn` and others. These functions are documented on page ??.)

```
6552 </initex | package>
```

14 l3box implementation

```
6553 <*initex | package>
6554 <@@=box>
6555 <*package>
6556 \ProvidesExplPackage
6557 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
6558 \__expl_package_check:
6559 </package>
```

The code in this module is very straight forward so I'm not going to comment it very extensively.

14.1 Creating and initialising boxes

The following test files are used for this code: `m3box001.lvt`.

`\box_new:N` Defining a new $\langle box \rangle$ register: remember that box 255 is not generally available.

```
\box_new:c
6560 <*package>
6561 \cs_new_protected:Npn \box_new:N #1
6562 {
6563   \__chk_if_free_cs:N #1
6564   \newbox #1
6565 }
6566 </package>
6567 \cs_generate_variant:Nn \box_new:N { c }
```

`\box_clear:N` Clear a $\langle box \rangle$ register.

```
\box_clear:c
6568 \cs_new_protected:Npn \box_clear:N #1
\box_gclear:N
6569 { \box_set_eq:NN #1 \c_empty_box }
\box_gclear:c
6570 \cs_new_protected:Npn \box_gclear:N #1
6571 { \box_gset_eq:NN #1 \c_empty_box }
6572 \cs_generate_variant:Nn \box_clear:N { c }
6573 \cs_generate_variant:Nn \box_gclear:N { c }
```

`\box_clear_new:N` Clear or new.

```
\box_clear_new:c
6574 \cs_new_protected:Npn \box_clear_new:N #1
\box_gclear_new:N
6575 { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
\box_gclear_new:c
6576 \cs_new_protected:Npn \box_gclear_new:N #1
6577 { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
6578 \cs_generate_variant:Nn \box_clear_new:N { c }
6579 \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
6580 \cs_new_protected:Npn \box_set_eq:NN #1#2
\box_set_eq:Nc
6581 { \tex_setbox:D #1 \tex_copy:D #2 }
\box_set_eq:cc
6582 \cs_new_protected:Npn \box_gset_eq:NN
\box_gset_eq:NN
6583 { \tex_global:D \box_set_eq:NN }
\box_gset_eq:cN
6584 \cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }
\box_gset_eq:Nc
6585 \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
6586 \cs_new_protected:Npn \box_set_eq_clear:NN #1#2
\box_set_eq_clear:Nc
6587 { \tex_setbox:D #1 \tex_box:D #2 }
\box_set_eq_clear:cc
6588 \cs_new_protected:Npn \box_gset_eq_clear:NN
\box_gset_eq_clear:NN
6589 { \tex_global:D \box_set_eq_clear:NN }
\box_gset_eq_clear:cN
6590 \cs_generate_variant:Nn \box_set_eq_clear:NN { c , Nc , cc }
\box_gset_eq_clear:Nc
6591 \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
6592 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N { TF , T , F , p }
\box_if_exist:N $\text{\textit{TF}}$ 
6593 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c { TF , T , F , p }
\box_if_exist:c $\text{\textit{TF}}$ 
```

14.2 Measuring and setting box dimensions

`\box_ht:N` Accessing the height, depth, and width of a $\langle box \rangle$ register.

```

\box_ht:c      6594 \cs_new_eq:NN \box_ht:N \tex_ht:D
\box_dp:N      6595 \cs_new_eq:NN \box_dp:N \tex_dp:D
\box_dp:c      6596 \cs_new_eq:NN \box_wd:N \tex_wd:D
\box_wd:N      6597 \cs_generate_variant:Nn \box_ht:N { c }
\box_wd:c      6598 \cs_generate_variant:Nn \box_dp:N { c }
               6599 \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 6600 \cs_new_protected:Npn \box_set_dp:Nn #1#2
               6601 { \box_dp:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\box_set_dp:cn 6602 \cs_new_protected:Npn \box_set_ht:Nn #1#2
               6603 { \box_ht:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\box_set_wd:Nn 6604 \cs_new_protected:Npn \box_set_wd:Nn #1#2
               6605 { \box_wd:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\box_set_wd:cn 6606 \cs_generate_variant:Nn \box_set_ht:Nn { c }
               6607 \cs_generate_variant:Nn \box_set_dp:Nn { c }
               6608 \cs_generate_variant:Nn \box_set_wd:Nn { c }

```

14.3 Using boxes

`\box_use_clear:N` Using a $\langle box \rangle$. These are just T_EX primitives with meaningful names.

```

\box_use_clear:c 6609 \cs_new_eq:NN \box_use_clear:N \tex_box:D
\box_use:N       6610 \cs_new_eq:NN \box_use:N \tex_copy:D
\box_use:c       6611 \cs_generate_variant:Nn \box_use_clear:N { c }
               6612 \cs_generate_variant:Nn \box_use:N { c }

```

`\box_move_left:nn` Move box material in different directions.

```

\box_move_right:nn 6613 \cs_new_protected:Npn \box_move_left:nn #1#2
\box_move_up:nn    6614 { \tex_moveleft:D \__dim_eval:w #1 \__dim_eval_end: #2 }
\box_move_down:nn  6615 \cs_new_protected:Npn \box_move_right:nn #1#2
               6616 { \tex_moveright:D \__dim_eval:w #1 \__dim_eval_end: #2 }
               6617 \cs_new_protected:Npn \box_move_up:nn #1#2
               6618 { \tex_raise:D \__dim_eval:w #1 \__dim_eval_end: #2 }
               6619 \cs_new_protected:Npn \box_move_down:nn #1#2
               6620 { \tex_lower:D \__dim_eval:w #1 \__dim_eval_end: #2 }

```

14.4 Box conditionals

`\if_hbox:N` The primitives for testing if a $\langle box \rangle$ is empty/void or which type of box it is.

```

\if_vbox:N      6621 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
\if_box_empty:N 6622 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
               6623 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D

```

```

\box_if_horizontal_p:N
\box_if_horizontal_p:c
\box_if_horizontal:N $\overline{TF}$ 
\box_if_horizontal:c $\overline{TF}$ 
\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_vertical:N $\overline{TF}$ 
\box_if_vertical:c $\overline{TF}$ 

```

```

6624 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
6625 { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
6626 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
6627 { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
6628 \cs_generate_variant:Nn \box_if_horizontal_p:N { c }
6629 \cs_generate_variant:Nn \box_if_horizontal:NT { c }
6630 \cs_generate_variant:Nn \box_if_horizontal:NF { c }
6631 \cs_generate_variant:Nn \box_if_horizontal:NTF { c }
6632 \cs_generate_variant:Nn \box_if_vertical_p:N { c }
6633 \cs_generate_variant:Nn \box_if_vertical:NT { c }
6634 \cs_generate_variant:Nn \box_if_vertical:NF { c }
6635 \cs_generate_variant:Nn \box_if_vertical:NTF { c }

```

$\backslash\box_if_empty_p:N$ Testing if a $\langle box \rangle$ is empty/void.

```

\box_if_empty_p:c
\box_if_empty:N $\overline{TF}$ 
\box_if_empty:c $\overline{TF}$ 

```

```

6636 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
6637 { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
6638 \cs_generate_variant:Nn \box_if_empty_p:N { c }
6639 \cs_generate_variant:Nn \box_if_empty:NT { c }
6640 \cs_generate_variant:Nn \box_if_empty:NF { c }
6641 \cs_generate_variant:Nn \box_if_empty:NTF { c }

```

(End definition for $\backslash\box_new:N$ and $\backslash\box_new:c$. These functions are documented on page ??.)

14.5 The last box inserted

$\backslash\box_set_to_last:N$ Set a box to the previous box.

```

\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

```

```

6642 \cs_new_protected:Npn \box_set_to_last:N #1
6643 { \tex_setbox:D #1 \tex_lastbox:D }
6644 \cs_new_protected:Npn \box_gset_to_last:N
6645 { \tex_global:D \box_set_to_last:N }
6646 \cs_generate_variant:Nn \box_set_to_last:N { c }
6647 \cs_generate_variant:Nn \box_gset_to_last:N { c }

```

(End definition for $\backslash\box_set_to_last:N$ and $\backslash\box_set_to_last:c$. These functions are documented on page ??.)

14.6 Constant boxes

$\backslash\c_empty_box$ A box we never use.

```

6648 \box_new:N \c_empty_box

```

(End definition for $\backslash\c_empty_box$. This variable is documented on page 131.)

14.7 Scratch boxes

$\backslash\l_tmpa_box$ Scratch boxes.

```

\l_tmpb_box
\g_tmpa_box
\g_tmpb_box

```

```

6649 \box_new:N \l_tmpa_box
6650 \box_new:N \l_tmpb_box
6651 \box_new:N \g_tmpa_box
6652 \box_new:N \g_tmpb_box

```

(End definition for $\backslash\l_tmpa_box$ and others. These variables are documented on page 131.)

14.8 Viewing box contents

TeX's `\tex_showbox:D` is not really that helpful in many cases, and it is also inconsistent with other L^AT_EX3 show functions as it does not actually shows material in the terminal. So we provide a richer set of functionality.

`\box_show:N` Essentially a wrapper around the internal function.
`\box_show:c` 6653 `\cs_new_protected:Npn \box_show:N #1`
`\box_show:Nnn` 6654 `{ \box_show:Nnn #1 \c_max_int \c_max_int }`
`\box_show:cnn` 6655 `\cs_generate_variant:Nn \box_show:N { c }`
6656 `\cs_new_protected_nopar:Npn \box_show:Nnn`
6657 `{ _box_show:NNnn \c_one }`
6658 `\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
`\box_log:c` interaction mode. For that, the ϵ -TeX extensions are needed.
`\box_log:Nnn` 6659 `\cs_new_protected:Npn \box_log:N #1`
`\box_log:cnn` 6660 `{ \box_log:Nnn #1 \c_max_int \c_max_int }`
6661 `\cs_generate_variant:Nn \box_log:N { c }`
6662 `\cs_new_protected:Npn \box_log:Nnn #1#2#3`
6663 `{`
6664 `\use:x`
6665 `{`
6666 `\etex_interactionmode:D \c_zero`
6667 `_box_show:NNnn \c_zero \exp_not:N #1`
6668 `{ \int_eval:n {#2} } { \int_eval:n {#3} }`
6669 `\etex_interactionmode:D`
6670 `= \tex_the:D \etex_interactionmode:D \scan_stop:`
6671 `}`
6672 `}`
6673 `\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.

6674 `\cs_new_protected:Npn _box_show:NNnn #1#2#3#4`
6675 `{`
6676 `\group_begin:`
6677 `\int_set:Nn \tex_showboxbreadth:D {#3}`
6678 `\int_set:Nn \tex_showboxdepth:D {#4}`
6679 `\int_set_eq:NN \tex_tracingonline:D #1`
6680 `\box_if_exist:NTF #2`
6681 `{ \tex_showbox:D \use:n {#2} }`
6682 `{`
6683 `_msg_kernel_error:nxx { kernel } { variable-not-defined }`
6684 `{ \token_to_str:N #2 }`
6685 `}`

```

6686     \group_end:
6687   }
(End definition for \_box_show:Nnn.)

```

14.9 Horizontal mode boxes

\hbox:n (The test suite for this command, and others in this file, is *m3box002.lvt*.)
Put a horizontal box directly into the input stream.

```

6688 \cs_new_protected:Npn \hbox:n { \tex_hbox:D \scan_stop: }
(End definition for \hbox:n. This function is documented on page 132.)

```

```

\hbox_set:Nn
\hbox_set:cn 6689 \cs_new_protected:Npn \hbox_set:Nn #1#2 { \tex_setbox:D #1 \tex_hbox:D {#2} }
\hbox_gset:Nn 6690 \cs_new_protected:Npn \hbox_gset:Nn { \tex_global:D \hbox_set:Nn }
\hbox_gset:cn 6691 \cs_generate_variant:Nn \hbox_set:Nn { c }
6692 \cs_generate_variant:Nn \hbox_gset:Nn { c }
(End definition for \hbox_set:Nn and \hbox_set:cn. These functions are documented on page ??.)

```

\hbox_set_to_wd:Nnn Storing material in a horizontal box with a specified width.
\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn

```

6693 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
6694 { \tex_setbox:D #1 \tex_hbox:D to \_dim_eval:w #2 \_dim_eval_end: {#3} }
6695 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn
6696 { \tex_global:D \hbox_set_to_wd:Nnn }
6697 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
6698 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }
(End definition for \hbox_set_to_wd:Nnn and \hbox_set_to_wd:cnn. These functions are documented
on page ??.)

```

\hbox_set:Nw Storing material in a horizontal box. This type is useful in environment definitions.
\hbox_set:cw
\hbox_gset:Nw
\hbox_gset:cw
\hbox_set_end:
\hbox_gset_end:

```

6699 \cs_new_protected:Npn \hbox_set:Nw #1
6700 { \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token }
6701 \cs_new_protected:Npn \hbox_gset:Nw
6702 { \tex_global:D \hbox_set:Nw }
6703 \cs_generate_variant:Nn \hbox_set:Nw { c }
6704 \cs_generate_variant:Nn \hbox_gset:Nw { c }
6705 \cs_new_eq:NN \hbox_set_end: \c_group_end_token
6706 \cs_new_eq:NN \hbox_gset_end: \c_group_end_token
(End definition for \hbox_set:Nw and \hbox_set:cw. These functions are documented on page 133.)

```

\hbox_set_inline_begin:N Renamed September 2011.
\hbox_set_inline_begin:c
\hbox_gset_inline_begin:N
\hbox_gset_inline_begin:c
\hbox_set_inline_end:
\hbox_gset_inline_end:

```

6707 \cs_new_eq:NN \hbox_set_inline_begin:N \hbox_set:Nw
6708 \cs_new_eq:NN \hbox_set_inline_begin:c \hbox_set:cw
6709 \cs_new_eq:NN \hbox_set_inline_end: \hbox_set_end:
6710 \cs_new_eq:NN \hbox_gset_inline_begin:N \hbox_gset:Nw
6711 \cs_new_eq:NN \hbox_gset_inline_begin:c \hbox_gset:cw
6712 \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`

```

6713 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
6714 { \tex_hbox:D to \__dim_eval:w #1 \__dim_eval_end: {#2} }
6715 \cs_new_protected:Npn \hbox_to_zero:n #1 { \tex_hbox:D to \c_zero_dim {#1} }

```

(End definition for `\hbox_to_wd:nn`. This function is documented on page 132.)

`\hbox_overlap_left:n` Put a zero-sized box with the contents pushed against one side (which makes it stick out
`\hbox_overlap_right:n` on the other) directly into the input stream.

```

6716 \cs_new_protected:Npn \hbox_overlap_left:n #1
6717 { \hbox_to_zero:n { \tex_hss:D #1 } }
6718 \cs_new_protected:Npn \hbox_overlap_right:n #1
6719 { \hbox_to_zero:n { #1 \tex_hss:D } }

```

(End definition for `\hbox_overlap_left:n` and `\hbox_overlap_right:n`. These functions are documented on page 132.)

`\hbox_unpack:N` Unpacking a box and if requested also clear it.
`\hbox_unpack:c`
`\hbox_unpack_clear:N`
`\hbox_unpack_clear:c`

```

6720 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
6721 \cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
6722 \cs_generate_variant:Nn \hbox_unpack:N { c }
6723 \cs_generate_variant:Nn \hbox_unpack_clear:N { c }

```

(End definition for `\hbox_unpack:N` and `\hbox_unpack:c`. These functions are documented on page ??.)

14.10 Vertical mode boxes

TeX ends these boxes directly with the internal `end_graf` routine. This means that there is no `\par` at the end of vertical boxes unless we insert one.

`\vbox:n` The following test files are used for this code: `m3box003.lvt`.

`\vbox_top:n` The following test files are used for this code: `m3box003.lvt`.
Put a vertical box directly into the input stream.

```

6724 \cs_new_protected:Npn \vbox:n #1 { \tex_vbox:D { #1 \par } }
6725 \cs_new_protected:Npn \vbox_top:n #1 { \tex_vtop:D { #1 \par } }

```

(End definition for `\vbox:n`. This function is documented on page 133.)

`\vbox_to_ht:nn` Put a vertical box directly into the input stream.
`\vbox_to_zero:n`

```

6726 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
6727 { \tex_vbox:D to \__dim_eval:w #1 \__dim_eval_end: { #2 \par } }
6728 \cs_new_protected:Npn \vbox_to_zero:n #1
6729 { \tex_vbox:D to \c_zero_dim { #1 \par } }

```

(End definition for `\vbox_to_ht:nn` and `\vbox_to_zero:n`. These functions are documented on page 134.)

`\vbox_set:Nn` Storing material in a vertical box with a natural height.
`\vbox_set:cn`
`\vbox_gset:Nn`
`\vbox_gset:cn`

```

6730 \cs_new_protected:Npn \vbox_set:Nn #1#2
6731 { \tex_setbox:D #1 \tex_vbox:D { #2 \par } }
6732 \cs_new_protected:Npn \vbox_gset:Nn { \tex_global:D \vbox_set:Nn }
6733 \cs_generate_variant:Nn \vbox_set:Nn { c }
6734 \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.

```

6735 \cs_new_protected:Npn \vbox_set_top:Nn #1#2
6736 { \tex_setbox:D #1 \tex_vtop:D { #2 \par } }
6737 \cs_new_protected:Npn \vbox_gset_top:Nn
6738 { \tex_global:D \vbox_set_top:Nn }
6739 \cs_generate_variant:Nn \vbox_set_top:Nn { c }
6740 \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.

```

6741 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
6742 { \tex_setbox:D #1 \tex_vbox:D to \_dim_eval:w #2 \_dim_eval_end: { #3 \par } }
6743 \cs_new_protected:Npn \vbox_gset_to_ht:Nnn
6744 { \tex_global:D \vbox_set_to_ht:Nnn }
6745 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }
6746 \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.

```

6747 \cs_new_protected:Npn \vbox_set:Nw #1
6748 { \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }
6749 \cs_new_protected:Npn \vbox_gset:Nw
6750 { \tex_global:D \vbox_set:Nw }
6751 \cs_generate_variant:Nn \vbox_set:Nw { c }
6752 \cs_generate_variant:Nn \vbox_gset:Nw { c }
6753 \cs_new_protected:Npn \vbox_set_end:
6754 {
6755   \par
6756   \c_group_end_token
6757 }
6758 \cs_new_eq:NN \vbox_gset_end: \vbox_set_end:

```

(End definition for `\vbox_set:Nw` and `\vbox_set:cw`. These functions are documented on page 134.)

`\vbox_set_inline_begin:N` Renamed September 2011.

```

6759 \cs_new_eq:NN \vbox_set_inline_begin:N \vbox_set:Nw
6760 \cs_new_eq:NN \vbox_set_inline_begin:c \vbox_set:cw
6761 \cs_new_eq:NN \vbox_set_inline_end: \vbox_set_end:
6762 \cs_new_eq:NN \vbox_gset_inline_begin:N \vbox_gset:Nw
6763 \cs_new_eq:NN \vbox_gset_inline_begin:c \vbox_gset:cw
6764 \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.

```

\lcs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
\lcs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D
\lcs_generate_variant:Nn \vbox_unpack:N { c }
\lcs_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.

```

\lcs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
{ \tex_setbox:D #1 \tex_vsplit:D #2 to \__dim_eval:w #3 \__dim_eval_end: }

```

(End definition for `\vbox_set_split_to_ht:NNn`. This function is documented on page 134.)

14.11 Deprecated functions

`\l_last_box` Deprecated 2011-11-13, for removal by 2012-02-28.

```

\*deprecated
\lcs_new_eq:NN \l_last_box \tex_lastbox:D
\*deprecated

```

(End definition for `\l_last_box`. This variable is documented on page ??.)

```

\*initex | package

```

15 l3coffins Implementation

```

\*initex | package
\@@=coffin
\*package
\ProvidesExplPackage
{ \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
\__expl_package_check:
\*package

```

15.1 Coffins: data structures and general variables

`\l__coffin_internal_box` Scratch variables.

```

\l__coffin_internal_dim
\l__coffin_internal_tl

```

```

\box_new:N \l__coffin_internal_box
\dim_new:N \l__coffin_internal_dim
\tl_new:N \l__coffin_internal_tl

```

(End definition for `\l__coffin_internal_box`. This function is documented on page ??.)

`\c__coffin_corners_prop` The “corners”; of a coffin define the real content, as opposed to the \TeX bounding box. They all start off in the same place, of course.

```

\prop_new:N \c__coffin_corners_prop
\prop_put:Nnn \c__coffin_corners_prop { tl } { { 0 pt } { 0 pt } }
\prop_put:Nnn \c__coffin_corners_prop { tr } { { 0 pt } { 0 pt } }
\prop_put:Nnn \c__coffin_corners_prop { bl } { { 0 pt } { 0 pt } }
\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.

```

6790 \prop_new:N \c__coffin_poles_prop
6791 \tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 0 pt } { 1000 pt } }
6792 \prop_put:Nno \c__coffin_poles_prop { l } { \l__coffin_internal_tl }
6793 \prop_put:Nno \c__coffin_poles_prop { hc } { \l__coffin_internal_tl }
6794 \prop_put:Nno \c__coffin_poles_prop { r } { \l__coffin_internal_tl }
6795 \tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 1000 pt } { 0 pt } }
6796 \prop_put:Nno \c__coffin_poles_prop { b } { \l__coffin_internal_tl }
6797 \prop_put:Nno \c__coffin_poles_prop { vc } { \l__coffin_internal_tl }
6798 \prop_put:Nno \c__coffin_poles_prop { t } { \l__coffin_internal_tl }
6799 \prop_put:Nno \c__coffin_poles_prop { B } { \l__coffin_internal_tl }
6800 \prop_put:Nno \c__coffin_poles_prop { H } { \l__coffin_internal_tl }
6801 \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.

```

6802 \fp_new:N \l__coffin_slope_x_fp
6803 \fp_new:N \l__coffin_slope_y_fp

```

(End definition for `\l__coffin_slope_x_fp`. This function is documented on page ??.)

`\l__coffin_error_bool` For propagating errors so that parts of the code can work around them.

```

6804 \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.

```

6805 \dim_new:N \l__coffin_offset_x_dim
6806 \dim_new:N \l__coffin_offset_y_dim

```

(End definition for `\l__coffin_offset_x_dim`. This function is documented on page ??.)

`\l__coffin_pole_a_tl` Needed for finding the intersection of two poles.

```

6807 \tl_new:N \l__coffin_pole_a_tl
6808 \tl_new:N \l__coffin_pole_b_tl

```

(End definition for `\l__coffin_pole_a_tl`. This function is documented on page ??.)

`\l__coffin_x_dim` For calculating intersections and so forth.

```

\l__coffin_y_dim
6809 \dim_new:N \l__coffin_x_dim
\l__coffin_x_prime_dim
6810 \dim_new:N \l__coffin_y_dim
\l__coffin_y_prime_dim
6811 \dim_new:N \l__coffin_x_prime_dim
6812 \dim_new:N \l__coffin_y_prime_dim

```

(End definition for `\l__coffin_x_dim`. This function is documented on page ??.)

15.2 Basic coffin functions

There are a number of basic functions needed for creating coffins and placing material in them. This all relies on the following data structures.

\coffin_if_exist_p:N Several of the higher-level coffin functions will give multiple errors if the coffin does not exist. A cleaner way to handle this is provided here: both the box and the coffin structure are checked.

\coffin_if_exist_p:c

\coffin_if_exist:N \underline{T}

\coffin_if_exist:c \underline{T}

```

6813 \prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
6814 {
6815     \cs_if_exist:NTF #1
6816     {
6817         \cs_if_exist:cTF { l__coffin_poles_ \__int_value:w #1 _prop }
6818         { \prg_return_true: }
6819         { \prg_return_false: }
6820     }
6821     { \prg_return_false: }
6822 }
6823 \cs_generate_variant:Nn \coffin_if_exist_p:N { c }
6824 \cs_generate_variant:Nn \coffin_if_exist:N $\underline{T}$  { c }
6825 \cs_generate_variant:Nn \coffin_if_exist:N $\underline{F}$  { c }
6826 \cs_generate_variant:Nn \coffin_if_exist:N $\underline{TF}$  { c }

```

(End definition for \coffin_if_exist:N and \coffin_if_exist:c. These functions are documented on page ??.)

__coffin_if_exist:N \underline{T} 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.

```

6827 \cs_new_protected:Npn \__coffin_if_exist:N $\underline{T}$  #1#2
6828 {
6829     \coffin_if_exist:N $\underline{TF}$  #1
6830     { #2 }
6831     {
6832         \__msg_kernel_error:nxx { kernel } { unknown-coffin }
6833         { \token_to_str:N #1 }
6834     }
6835 }

```

(End definition for __coffin_if_exist:N \underline{T} . This function is documented on page ??.)

\coffin_clear:N Clearing coffins means emptying the box and resetting all of the structures.

\coffin_clear:c

```

6836 \cs_new_protected:Npn \coffin_clear:N #1
6837 {
6838     \__coffin_if_exist:N $\underline{T}$  #1
6839     {
6840         \box_clear:N #1
6841         \__coffin_reset_structure:N #1
6842     }
6843 }
6844 \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.

```

6845 \cs_new_protected:Npn \coffin_new:N #1
6846 {
6847   \box_new:N #1
6848   \prop_clear_new:c { l__coffin_corners_ } __int_value:w #1 _prop }
6849   \prop_clear_new:c { l__coffin_poles_ } __int_value:w #1 _prop }
6850   \prop_gset_eq:cN { l__coffin_corners_ } __int_value:w #1 _prop }
6851   \c__coffin_corners_prop
6852   \prop_gset_eq:cN { l__coffin_poles_ } __int_value:w #1 _prop }
6853   \c__coffin_poles_prop
6854 }
6855 \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.

```

6856 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
6857 {
6858   \__coffin_if_exist:NT #1
6859   {
6860     \hbox_set:Nn #1
6861     {
6862       \color_group_begin:
6863       \color_ensure_current:
6864       #2
6865       \color_group_end:
6866     }
6867     \__coffin_reset_structure:N #1
6868     \__coffin_update_poles:N #1
6869     \__coffin_update_corners:N #1
6870   }
6871 }
6872 \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 *TeX by Topic* for discussion of the `\vtop` primitive, used to do the measuring).

```

6873 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
6874 {

```

```

6875     \_coffin_if_exist:NT #1
6876     {
6877         \vbox_set:Nn #1
6878         {
6879             \dim_set:Nn \tex_hsize:D {#2}
6880 <*package>
6881             \dim_set_eq:NN \linewidth \tex_hsize:D
6882             \dim_set_eq:NN \columnwidth \tex_hsize:D
6883 </package>
6884             \color_group_begin:
6885             #3
6886             \color_group_end:
6887         }
6888         \_coffin_reset_structure:N #1
6889         \_coffin_update_poles:N #1
6890         \_coffin_update_corners:N #1
6891         \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
6892         \_coffin_set_pole:Nnx #1 { T }
6893         {
6894             { 0 pt }
6895             { \dim_eval:n { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box } }
6896             { 1000 pt }
6897             { 0 pt }
6898         }
6899         \box_clear:N \l__coffin_internal_box
6900     }
6901 }
6902 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }

```

(End definition for \vcoffin_set:Nnn and \vcoffin_set:cnn. These functions are documented on page ??.)

\hcoffin_set:Nw These are the “begin”/“end” versions of the above: watch the grouping!
\hcoffin_set:cw
\hcoffin_set_end:

```

6903 \cs_new_protected:Npn \hcoffin_set:Nw #1
6904 {
6905     \_coffin_if_exist:NT #1
6906     {
6907         \hbox_set:Nw #1 \color_group_begin: \color_ensure_current:
6908         \cs_set_protected_nopar:Npn \hcoffin_set_end:
6909         {
6910             \color_group_end:
6911             \hbox_set_end:
6912             \_coffin_reset_structure:N #1
6913             \_coffin_update_poles:N #1
6914             \_coffin_update_corners:N #1
6915         }
6916     }
6917 }
6918 \cs_new_protected_nopar:Npn \hcoffin_set_end: { }
6919 \cs_generate_variant:Nn \hcoffin_set:Nw { c }

```

(End definition for `\hcoffin_set:Nw` and `\hcoffin_set:cw`. These functions are documented on page 137.)

`\vcoffin_set:Nnw` The same for vertical coffins.

```

6920 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
6921 {
6922   \__coffin_if_exist:NT #1
6923   {
6924     \vbox_set:Nw #1
6925     \dim_set:Nn \tex_hsize:D {#2}
6926     <*package>
6927       \dim_set_eq:NN \linewidth \tex_hsize:D
6928       \dim_set_eq:NN \columnwidth \tex_hsize:D
6929     </package>
6930     \color_group_begin: \color_ensure_current:
6931     \cs_set_protected:Npn \vcoffin_set_end:
6932     {
6933       \color_group_end:
6934       \vbox_set_end:
6935       \__coffin_reset_structure:N #1
6936       \__coffin_update_poles:N #1
6937       \__coffin_update_corners:N #1
6938       \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
6939       \__coffin_set_pole:Nnx #1 { T }
6940       {
6941         { 0 pt }
6942         {
6943           \dim_eval:n { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
6944         }
6945         { 1000 pt }
6946         { 0 pt }
6947       }
6948       \box_clear:N \l__coffin_internal_box
6949     }
6950   }
6951 }
6952 \cs_new_protected_nopar:Npn \vcoffin_set_end: { }
6953 \cs_generate_variant:Nn \vcoffin_set:Nnw { c }

```

(End definition for `\vcoffin_set:Nnw` and `\vcoffin_set:cw`. These functions are documented on page 137.)

`\coffin_set_eq:NN` Setting two coffins equal is just a wrapper around other functions.

```

6954 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
6955 {
6956   \__coffin_if_exist:NT #1
6957   {
6958     \box_set_eq:NN #1 #2
6959     \__coffin_set_eq_structure:NN #1 #2
6960   }
6961 }

```

```
6962 \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

```
6963 \coffin_new:N \c_empty_coffin
6964 \hbox_set:Nn \c_empty_coffin { }
6965 \coffin_new:N \l__coffin_aligned_coffin
6966 \coffin_new:N \l__coffin_aligned_internal_coffin
(End definition for \c_empty_coffin. This function is documented on page ??.)
```

\l_tmpa_coffin The usual scratch space.

\l_tmpb_coffin

```
6967 \coffin_new:N \l_tmpa_coffin
6968 \coffin_new:N \l_tmpb_coffin
(End definition for \l_tmpa_coffin and \l_tmpb_coffin. These variables are documented on page 139.)
```

15.3 Measuring coffins

\coffin_dp:N Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

\coffin_dp:c

\coffin_ht:N

\coffin_ht:c

\coffin_wd:N

\coffin_wd:c

```
6969 \cs_new_eq:NN \coffin_dp:N \box_dp:N
6970 \cs_new_eq:NN \coffin_dp:c \box_dp:c
6971 \cs_new_eq:NN \coffin_ht:N \box_ht:N
6972 \cs_new_eq:NN \coffin_ht:c \box_ht:c
6973 \cs_new_eq:NN \coffin_wd:N \box_wd:N
6974 \cs_new_eq:NN \coffin_wd:c \box_wd:c
(End definition for \coffin_dp:N and others. These functions are documented on page ??.)
```

15.4 Coffins: handle and pole management

__coffin_get_pole:NnN A simple wrapper around the recovery of a coffin pole, with some error checking and recovery built-in.

```
6975 \cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3
6976 {
6977   \prop_get:cnNF
6978     { l__coffin_poles_ \__int_value:w #1 _prop } {#2} #3
6979     {
6980       \_msg_kernel_error:nxxx { kernel } { unknown-coffin-pole }
6981       {#2} { \token_to_str:N #1 }
6982       \tl_set:Nn #3 { { 0 pt } { 0 pt } { 0 pt } { 0 pt } }
6983     }
6984 }
```

(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.

```

6985 \cs_new_protected:Npn \__coffin_reset_structure:N #1
6986 {
6987   \prop_set_eq:cN { l__coffin_corners_ \__int_value:w #1 _prop }
6988   \c__coffin_corners_prop
6989   \prop_set_eq:cN { l__coffin_poles_ \__int_value:w #1 _prop }
6990   \c__coffin_poles_prop
6991 }

```

(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`

```

6992 \cs_new_protected:Npn \__coffin_set_eq_structure:NN #1#2
6993 {
6994   \prop_set_eq:cc { l__coffin_corners_ \__int_value:w #1 _prop }
6995   { l__coffin_corners_ \__int_value:w #2 _prop }
6996   \prop_set_eq:cc { l__coffin_poles_ \__int_value:w #1 _prop }
6997   { l__coffin_poles_ \__int_value:w #2 _prop }
6998 }
6999 \cs_new_protected:Npn \__coffin_gset_eq_structure:NN #1#2
7000 {
7001   \prop_gset_eq:cc { l__coffin_corners_ \__int_value:w #1 _prop }
7002   { l__coffin_corners_ \__int_value:w #2 _prop }
7003   \prop_gset_eq:cc { l__coffin_poles_ \__int_value:w #1 _prop }
7004   { l__coffin_poles_ \__int_value:w #2 _prop }
7005 }

```

(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.

```

7006 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
7007 {
7008   \__coffin_if_exist:NT #1
7009   {
7010     \__coffin_set_pole:Nnx #1 {#2}
7011     {
7012       { 0 pt } { \dim_eval:n {#3} }
7013       { 1000 pt } { 0 pt }
7014     }
7015   }
7016 }
7017 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
7018 {
7019   \__coffin_if_exist:NT #1
7020   {
7021     \__coffin_set_pole:Nnx #1 {#2}
7022     {
7023       { \dim_eval:n {#3} } { 0 pt }

```

```

7024         { 0 pt } { 1000 pt }
7025     }
7026 }
7027 }
7028 \cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
7029 { \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } {#2} {#3} }
7030 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
7031 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
7032 \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.

```

7033 \cs_new_protected:Npn \__coffin_update_corners:N #1
7034 {
7035     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tl }
7036     { { 0 pt } { \dim_use:N \box_ht:N #1 } }
7037     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tr }
7038     { { \dim_use:N \box_wd:N #1 } { \dim_use:N \box_ht:N #1 } }
7039     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { bl }
7040     { { 0 pt } { \dim_eval:n { - \box_dp:N #1 } } }
7041     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { br }
7042     { { \dim_use:N \box_wd:N #1 } { \dim_eval:n { - \box_dp:N #1 } } }
7043 }

```

(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.

```

7044 \cs_new_protected:Npn \__coffin_update_poles:N #1
7045 {
7046     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { hc }
7047     {
7048         { \dim_eval:n { 0.5 \box_wd:N #1 } }
7049         { 0 pt } { 0 pt } { 1000 pt }
7050     }
7051     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { r }
7052     {
7053         { \dim_use:N \box_wd:N #1 }
7054         { 0 pt } { 0 pt } { 1000 pt }
7055     }
7056     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { vc }
7057     {
7058         { 0 pt }
7059         { \dim_eval:n { ( \box_ht:N #1 - \box_dp:N #1 ) / 2 } }
7060         { 1000 pt }
7061         { 0 pt }

```

```

7062     }
7063     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { t }
7064     {
7065         { 0 pt }
7066         { \dim_use:N \box_ht:N #1 }
7067         { 1000 pt }
7068         { 0 pt }
7069     }
7070     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { b }
7071     {
7072         { 0 pt }
7073         { \dim_eval:n { - \box_dp:N #1 } }
7074         { 1000 pt }
7075         { 0 pt }
7076     }
7077 }

```

(End definition for `__coffin_update_poles:N`. This function is documented on page ??.)

15.5 Coffins: calculation of pole intersections

```

\__coffin_calculate_intersection:Nnn
\__coffin_calculate_intersection:nnnnnnnn
\__coffin_calculate_intersection_aux: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.

```

7078 \cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
7079 {
7080     \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
7081     \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
7082     \bool_set_false:N \l__coffin_error_bool
7083     \exp_last_two_unbraced:Noo
7084     \__coffin_calculate_intersection:nnnnnnnn
7085     \l__coffin_pole_a_tl \l__coffin_pole_b_tl
7086     \bool_if:NT \l__coffin_error_bool
7087     {
7088         \__msg_kernel_error:nn { kernel } { no-pole-intersection }
7089         \dim_zero:N \l__coffin_x_dim
7090         \dim_zero:N \l__coffin_y_dim
7091     }
7092 }

```

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.

```

7093 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
7094 #1#2#3#4#5#6#7#8
7095 {
7096     \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.

```

7097      {
7098          \dim_set:Nn \l__coffin_x_dim {#1}
7099          \dim_compare:nNnTF {#7} = \c_zero_dim
7100              { \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.

```

7101      {
7102          \dim_compare:nNnTF {#8} = \c_zero_dim
7103              { \dim_set:Nn \l__coffin_y_dim {#6} }
7104              {
7105                  \__coffin_calculate_intersection_aux:nnnnnN
7106                      {#1} {#5} {#6} {#7} {#8} \l__coffin_y_dim
7107              }
7108      }
7109  }

```

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.

```

7110      {
7111          \dim_compare:nNnTF {#4} = \c_zero_dim
7112              {
7113                  \dim_set:Nn \l__coffin_y_dim {#2}
7114                  \dim_compare:nNnTF {#8} = { \c_zero_dim }
7115                  { \bool_set_true:N \l__coffin_error_bool }
7116                  {
7117                      \dim_compare:nNnTF {#7} = \c_zero_dim
7118                      { \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.

```

7119      {
7120          \__coffin_calculate_intersection_aux:nnnnnN
7121              {#2} {#6} {#5} {#8} {#7} \l__coffin_x_dim
7122      }
7123  }
7124  }

```

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.

```

7125     {
7126         \dim_compare:nNnTF {#7} = \c_zero_dim
7127         {
7128             \dim_set:Nn \l__coffin_x_dim {#5}
7129             \__coffin_calculate_intersection_aux:nnnnnN
7130             {#5} {#1} {#2} {#3} {#4} \l__coffin_y_dim
7131         }
7132         {
7133             \dim_compare:nNnTF {#8} = \c_zero_dim
7134             {
7135                 \dim_set:Nn \l__coffin_x_dim {#6}
7136                 \__coffin_calculate_intersection_aux:nnnnnN
7137                 {#6} {#2} {#1} {#4} {#3} \l__coffin_x_dim
7138             }

```

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.

```

7139     {
7140         \fp_set:Nn \l__coffin_slope_x_fp
7141         { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
7142         \fp_set:Nn \l__coffin_slope_y_fp
7143         { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
7144         \fp_compare:nNnTF
7145         \l__coffin_slope_x_fp = \l__coffin_slope_y_fp
7146         { \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.

```

7147     {
7148         \dim_set:Nn \l__coffin_x_dim
7149         {
7150             \fp_to_dim:n
7151             {
7152                 (
7153                     \dim_to_fp:n {#1} * \l__coffin_slope_x_fp
7154                     - ( \dim_to_fp:n {#5} * \l__coffin_slope_y_fp )
7155                     - \dim_to_fp:n {#2}
7156                     + \dim_to_fp:n {#6}
7157                 )
7158                 /
7159                 ( \l__coffin_slope_x_fp - \l__coffin_slope_y_fp )
7160             }
7161         }

```

```

7162         \_coffin_calculate_intersection_aux:nnnnnN
7163         { \l__coffin_x_dim }
7164         {#5} {#6} {#8} {#7} \l__coffin_y_dim
7165     }
7166 }
7167 }
7168 }
7169 }
7170 }

```

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.

```

7171 \cs_new_protected:Npn \_coffin_calculate_intersection_aux:nnnnnN #1#2#3#4#5#6
7172 {
7173     \dim_set:Nn #6
7174     {
7175         \fp_to_dim:n
7176         {
7177             \dim_to_fp:n {#4} *
7178             ( \dim_to_fp:n {#1} - \dim_to_fp:n {#2} ) /
7179             \dim_to_fp:n {#5}
7180             + \dim_to_fp:n {#3}
7181         }
7182     }
7183 }

```

(End definition for _coffin_calculate_intersection:Nnn. This function is documented on page ??.)

15.6 Aligning and typesetting of coffins

`\coffin_join:NnnNnnnn`
`\coffin_join:cnnNnnnn`
`\coffin_join: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.

```

7184 \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
7185 {
7186     \_coffin_align:NnnNnnnnN
7187     #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.

```

7188     \hbox_set:Nn \l__coffin_aligned_coffin
7189     {
7190         \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
7191         { \tex_kern:D -\l__coffin_offset_x_dim }

```

```

7192     \hbox_unpack:N \l__coffin_aligned_coffin
7193     \dim_set:Nn \l__coffin_internal_dim
7194         { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
7195     \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
7196         { \tex_kern:D -\l__coffin_internal_dim }
7197 }

```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```

7198     \__coffin_reset_structure:N \l__coffin_aligned_coffin
7199     \prop_clear:c
7200     { \l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _ prop }
7201     \__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.

```

7202     \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
7203     {
7204         \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
7205         \__coffin_offset_poles:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
7206         \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
7207         \__coffin_offset_corners:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
7208     }
7209     {
7210         \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
7211         \__coffin_offset_poles:Nnn #4
7212             { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
7213         \__coffin_offset_corners:Nnn #1 { 0 pt } { 0 pt }
7214         \__coffin_offset_corners:Nnn #4
7215             { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
7216     }
7217     \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
7218     \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
7219 }
7220 \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 structure updates.

`\coffin_attach:Nnncnnnnn`

`\coffin_attach_mark:NnnNnnnn`

```

7221 \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
7222 {
7223     \__coffin_align:NnnNnnnnN
7224     #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
7225     \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
7226     \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
7227     \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
7228     \__coffin_reset_structure:N \l__coffin_aligned_coffin

```

```

7229 \prop_set_eq:cc
7230 { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
7231 { l__coffin_corners_ \__int_value:w #1 _prop }
7232 \__coffin_update_poles:N \l__coffin_aligned_coffin
7233 \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
7234 \__coffin_offset_poles:Nnn #4
7235 { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
7236 \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
7237 \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
7238 }
7239 \cs_new_protected:Npn \coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
7240 {
7241 \__coffin_align:NnnNnnnnN
7242 #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
7243 \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
7244 \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
7245 \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
7246 \box_set_eq:NN #1 \l__coffin_aligned_coffin
7247 }
7248 \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.

```

7249 \cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
7250 {
7251 \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
7252 \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
7253 \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
7254 \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
7255 \dim_set:Nn \l__coffin_offset_x_dim
7256 { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
7257 \dim_set:Nn \l__coffin_offset_y_dim
7258 { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
7259 \hbox_set:Nn \l__coffin_aligned_internal_coffin
7260 {
7261 \box_use:N #1
7262 \tex_kern:D -\box_wd:N #1
7263 \tex_kern:D \l__coffin_offset_x_dim
7264 \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
7265 }
7266 \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
7267 }

```

(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.

```

7268 \cs_new_protected:Npn \_coffin_offset_poles:Nnn #1#2#3
7269 {
7270   \prop_map_inline:cn { l__coffin_poles_ \_int_value:w #1 _prop }
7271   { \_coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
7272 }
7273 \cs_new_protected:Npn \_coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
7274 {
7275   \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
7276   \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
7277   \tl_if_in:nnTF {#2} { - }
7278   { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
7279   { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
7280   \exp_last_unbraced:NNo \_coffin_set_pole:Nnx \l__coffin_aligned_coffin
7281   { \l__coffin_internal_tl }
7282   {
7283     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
7284     {#5} {#6}
7285   }
7286 }

```

(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.

```

7287 \cs_new_protected:Npn \_coffin_offset_corners:Nnn #1#2#3
7288 {
7289   \prop_map_inline:cn { l__coffin_corners_ \_int_value:w #1 _prop }
7290   { \_coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
7291 }
7292 \cs_new_protected:Npn \_coffin_offset_corner:Nnnnn #1#2#3#4#5#6
7293 {
7294   \prop_put:cnx
7295   { l__coffin_corners_ \_int_value:w \l__coffin_aligned_coffin _prop }
7296   { #1 - #2 }
7297   {
7298     { \dim_eval:n { #3 + #5 } }
7299     { \dim_eval:n { #4 + #6 } }
7300   }
7301 }

```

(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.

```

7302 \cs_new_protected:Npn \_coffin_update_vertical_poles:NNN #1#2#3
7303 {
7304   \_coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
7305   \_coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
7306   \exp_last_two_unbraced:Noo \_coffin_update_T:nnnnnnnnN
7307   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
7308   \_coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
7309   \_coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
7310   \exp_last_two_unbraced:Noo \_coffin_update_B:nnnnnnnnN
7311   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
7312 }
7313 \cs_new_protected:Npn \_coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
7314 {
7315   \dim_compare:nNnTF {#2} < {#6}
7316   {
7317     \_coffin_set_pole:Nnx #9 { T }
7318     { { 0 pt } {#6} { 1000 pt } { 0 pt } }
7319   }
7320   {
7321     \_coffin_set_pole:Nnx #9 { T }
7322     { { 0 pt } {#2} { 1000 pt } { 0 pt } }
7323   }
7324 }
7325 \cs_new_protected:Npn \_coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
7326 {
7327   \dim_compare:nNnTF {#2} < {#6}
7328   {
7329     \_coffin_set_pole:Nnx #9 { B }
7330     { { 0 pt } {#2} { 1000 pt } { 0 pt } }
7331   }
7332   {
7333     \_coffin_set_pole:Nnx #9 { B }
7334     { { 0 pt } {#6} { 1000 pt } { 0 pt } }
7335   }
7336 }

```

(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.

```

7337 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
7338 {
7339   \hbox_unpack:N \c_empty_box
7340   \_coffin_align:NnnNnnnnN \c_empty_coffin { H } { 1 }
7341   #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
7342   \box_use:N \l__coffin_aligned_coffin

```

```

7343 }
7344 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }

```

(End definition for \coffin_typeset:Nnnnn and \coffin_typeset:cnnnn. These functions are documented on page ??.)

15.7 Coffin diagnostics

`\l__coffin_display_coffin` Used for printing coffins with data structures attached.

```

\l__coffin_display_coord_coffin 7345 \coffin_new:N \l__coffin_display_coffin
\l__coffin_display_pole_coffin 7346 \coffin_new:N \l__coffin_display_coord_coffin
7347 \coffin_new:N \l__coffin_display_pole_coffin

```

(End definition for \l__coffin_display_coffin. This function is documented on page ??.)

`\l__coffin_display_handles_prop` This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

```

7348 \prop_new:N \l__coffin_display_handles_prop
7349 \prop_put:Nnn \l__coffin_display_handles_prop { tl }
7350 { { b } { r } { -1 } { 1 } }
7351 \prop_put:Nnn \l__coffin_display_handles_prop { thc }
7352 { { b } { hc } { 0 } { 1 } }
7353 \prop_put:Nnn \l__coffin_display_handles_prop { tr }
7354 { { b } { l } { 1 } { 1 } }
7355 \prop_put:Nnn \l__coffin_display_handles_prop { vcl }
7356 { { vc } { r } { -1 } { 0 } }
7357 \prop_put:Nnn \l__coffin_display_handles_prop { vhc }
7358 { { vc } { hc } { 0 } { 0 } }
7359 \prop_put:Nnn \l__coffin_display_handles_prop { vcr }
7360 { { vc } { l } { 1 } { 0 } }
7361 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
7362 { { t } { r } { -1 } { -1 } }
7363 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
7364 { { t } { hc } { 0 } { -1 } }
7365 \prop_put:Nnn \l__coffin_display_handles_prop { br }
7366 { { t } { l } { 1 } { -1 } }
7367 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }
7368 { { t } { r } { -1 } { -1 } }
7369 \prop_put:Nnn \l__coffin_display_handles_prop { Thc }
7370 { { t } { hc } { 0 } { -1 } }
7371 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
7372 { { t } { l } { 1 } { -1 } }
7373 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
7374 { { vc } { r } { -1 } { 1 } }
7375 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
7376 { { vc } { hc } { 0 } { 1 } }
7377 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
7378 { { vc } { l } { 1 } { 1 } }
7379 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
7380 { { b } { r } { -1 } { -1 } }
7381 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }

```

```

7382 { { b } { hc } { 0 } { -1 } }
7383 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
7384 { { 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.

```

7385 \dim_new:N \l__coffin_display_offset_dim
7386 \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.

```

7387 \dim_new:N \l__coffin_display_x_dim
7388 \dim_new:N \l__coffin_display_y_dim
(End definition for \l__coffin_display_x_dim. This function 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.

```

7389 \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.

```

7390 \tl_new:N \l__coffin_display_font_tl
7391 <*initex>
7392 \tl_set:Nn \l__coffin_display_font_tl { } % TODO
7393 </initex>
7394 <*package>
7395 \tl_set:Nn \l__coffin_display_font_tl { \sfamily \tiny }
7396 </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.

```

7397 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
7398 {
7399   \hcoffin_set:Nn \l__coffin_display_pole_coffin
7400   {
7401     <*initex>
7402     \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
7403     </initex>
7404     <*package>
7405     \color {#4}
7406     \rule { 1 pt } { 1 pt }
7407     </package>
7408   }

```

```

7409 \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
7410 \l__coffin_display_pole_coffin { hc } { vc } { 0 pt } { 0 pt }
7411 \hcoffin_set:Nn \l__coffin_display_coord_coffin
7412 {
7413 <*initex>
7414 % TODO
7415 </initex>
7416 <*package>
7417 \color {#4}
7418 </package>
7419 \l__coffin_display_font_tl
7420 ( \tl_to_str:n { #2 , #3 } )
7421 }
7422 \prop_get:NnN \l__coffin_display_handles_prop
7423 { #2 #3 } \l__coffin_internal_tl
7424 \quark_if_no_value:NTF \l__coffin_internal_tl
7425 {
7426 \prop_get:NnN \l__coffin_display_handles_prop
7427 { #3 #2 } \l__coffin_internal_tl
7428 \quark_if_no_value:NTF \l__coffin_internal_tl
7429 {
7430 \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
7431 \l__coffin_display_coord_coffin { l } { vc }
7432 { 1 pt } { 0 pt }
7433 }
7434 {
7435 \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
7436 \l__coffin_internal_tl #1 {#2} {#3}
7437 }
7438 }
7439 {
7440 \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
7441 \l__coffin_internal_tl #1 {#2} {#3}
7442 }
7443 }
7444 \cs_new_protected:Npn \__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
7445 {
7446 \coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
7447 \l__coffin_display_coord_coffin {#1} {#2}
7448 { #3 \l__coffin_display_offset_dim }
7449 { #4 \l__coffin_display_offset_dim }
7450 }
7451 \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.

```

7452 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
7453 {
7454   \hcoffin_set:Nn \l__coffin_display_pole_coffin
7455   {
7456     (*initex)
7457     \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
7458   }
7459   (*package)
7460   \color {#2}
7461   \rule { 1 pt } { 1 pt }
7462 }
7463 }
7464 \prop_set_eq:Nc \l__coffin_display_poles_prop
7465 { l__coffin_poles_ \__int_value:w #1 _prop }
7466 \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
7467 \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
7468 \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
7469 { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
7470 \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
7471 \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
7472 { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
7473 \coffin_set_eq:NN \l__coffin_display_coffin #1
7474 \prop_map_inline:Nn \l__coffin_display_poles_prop
7475 {
7476   \prop_remove:Nn \l__coffin_display_poles_prop {##1}
7477   \__coffin_display_handles_aux:nnnnnn {##1} ##2 {##2}
7478 }
7479 \box_use:N \l__coffin_display_coffin
7480 }

```

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.

```

7481 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
7482 {
7483   \prop_map_inline:Nn \l__coffin_display_poles_prop
7484   {
7485     \bool_set_false:N \l__coffin_error_bool
7486     \__coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2
7487     \bool_if:NF \l__coffin_error_bool
7488     {
7489       \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
7490       \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
7491       \__coffin_display_attach:Nnnnn
7492       \l__coffin_display_pole_coffin { hc } { vc }
7493       { 0 pt } { 0 pt }
7494       \hcoffin_set:Nn \l__coffin_display_coord_coffin
7495       {
7496         (*initex)

```

```

7497                                     % TODO
7498 </initex>
7499 <*package>
7500                                     \color {#6}
7501 </package>
7502                                     \l__coffin_display_font_tl
7503                                     ( \tl_to_str:n { #1 , ##1 } )
7504                                 }
7505 \prop_get:NnN \l__coffin_display_handles_prop
7506 { #1 ##1 } \l__coffin_internal_tl
7507 \quark_if_no_value:NTF \l__coffin_internal_tl
7508 {
7509     \prop_get:NnN \l__coffin_display_handles_prop
7510     { ##1 #1 } \l__coffin_internal_tl
7511     \quark_if_no_value:NTF \l__coffin_internal_tl
7512     {
7513         \__coffin_display_attach:Nnnnn
7514         \l__coffin_display_coord_coffin { 1 } { vc }
7515         { 1 pt } { 0 pt }
7516     }
7517     {
7518         \exp_last_unbraced:No
7519         \__coffin_display_handles_aux:nnnn
7520         \l__coffin_internal_tl
7521     }
7522 }
7523 {
7524     \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
7525     \l__coffin_internal_tl
7526 }
7527 }
7528 }
7529 }
7530 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
7531 {
7532     \__coffin_display_attach:Nnnnn
7533     \l__coffin_display_coord_coffin {#1} {#2}
7534     { #3 \l__coffin_display_offset_dim }
7535     { #4 \l__coffin_display_offset_dim }
7536 }
7537 \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.

```

7538 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
7539 {
7540     \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
7541     \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
7542     \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }

```

```

7543 \dim_set:Nn \l__coffin_offset_x_dim
7544 { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
7545 \dim_set:Nn \l__coffin_offset_y_dim
7546 { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
7547 \hbox_set:Nn \l__coffin_aligned_coffin
7548 {
7549   \box_use:N \l__coffin_display_coffin
7550   \tex_kern:D -\box_wd:N \l__coffin_display_coffin
7551   \tex_kern:D \l__coffin_offset_x_dim
7552   \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
7553 }
7554 \box_set_ht:Nn \l__coffin_aligned_coffin
7555 { \box_ht:N \l__coffin_display_coffin }
7556 \box_set_dp:Nn \l__coffin_aligned_coffin
7557 { \box_dp:N \l__coffin_display_coffin }
7558 \box_set_wd:Nn \l__coffin_aligned_coffin
7559 { \box_wd:N \l__coffin_display_coffin }
7560 \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
7561 }

```

(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.

```

7562 \cs_new_protected:Npn \coffin_show_structure:N #1
7563 {
7564   \__coffin_if_exist:NT #1
7565   {
7566     \__msg_show_variable:Nnn #1 { coffins }
7567     {
7568       \prop_map_function:cN
7569       { l__coffin_poles_ \__int_value:w #1 _prop }
7570       \__msg_show_item_unbraced:nn
7571     }
7572   }
7573 }
7574 \cs_generate_variant:Nn \coffin_show_structure:N { c }

```

(End definition for `\coffin_show_structure:N` and `\coffin_show_structure:c`. These functions are documented on page ??.)

15.8 Messages

```

7575 \__msg_kernel_new:nnnn { kernel } { no-pole-intersection }
7576 { No~intersection~between~coffin~poles. }
7577 {
7578   \c_msg_coding_error_text_tl
7579   LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
7580   but~they~do~not~have~a~unique~meeting~point:~
7581   the~value~(0~pt,~0~pt)~will~be~used.

```



```

7582 }
7583 \_msg_kernel_new:nnnn { kernel } { unknown-coffin }
7584 { Unknown~coffin~'#1'. }
7585 { The~coffin~'#1'~was~never~defined. }
7586 \_msg_kernel_new:nnnn { kernel } { unknown-coffin-pole }
7587 { Pole~'#1'~unknown~for~coffin~'#2'. }
7588 {
7589   \c_msg_coding_error_text_tl
7590   LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
7591   but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
7592 }
7593 \_msg_kernel_new:nnn { kernel } { show-coffins }
7594 {
7595   Size~of~coffin~\token_to_str:N #1 : \\
7596   > ~ ht~==~\dim_use:N \box_ht:N #1 \\
7597   > ~ dp~==~\dim_use:N \box_dp:N #1 \\
7598   > ~ wd~==~\dim_use:N \box_wd:N #1 \\
7599   Poles~of~coffin~\token_to_str:N #1 :
7600 }
7601 </initex | package>

```

16 l3color Implementation

```

7602 <*initex | package>
7603 <*package>
7604 \ProvidesExplPackage
7605   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
7606   \_expl_package_check:
7607 </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.

```

7608 \cs_new_eq:NN \color_group_begin: \group_begin:
7609 \cs_new_protected_nopar:Npn \color_group_end:
7610 {
7611   \tex_par:D
7612   \group_end:
7613 }

```

(End definition for `\color_group_begin:` and `\color_group_end:`. These functions are documented on page 140.)

\color_ensure_current: A driver-independent wrapper for setting the foreground color to the current color “now”.

```

7614 <*initex>
7615 \cs_new_protected_nopar:Npn \color_ensure_current:
7616 { \_driver_color_ensure_current: }
7617 </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.

```

7618 <*package>
7619 \cs_new_protected_nopar:Npn \color_ensure_current: { }
7620 \AtBeginDocument
7621 {
7622   \cs_if_exist:NTF \__driver_color_ensure_current:
7623   {
7624     \cs_set_protected_nopar:Npn \color_ensure_current:
7625     { \__driver_color_ensure_current: }
7626   }
7627   {
7628     \cs_if_exist:NT \set@color
7629     { \cs_set_protected_nopar:Npn \color_ensure_current: { \set@color } }
7630   }
7631 }
7632 </package>

```

(End definition for `\color_ensure_current:`. This function is documented on page 140.)

```

7633 </initex | package>

```

17 l3msg implementation

```

7634 <*initex | package>
7635 <@@=msg>
7636 <*package>
7637 \ProvidesExplPackage
7638 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
7639 \__expl_package_check:
7640 </package>

```

`\l__msg_internal_tl` A general scratch for the module.

```

7641 \tl_new:N \l__msg_internal_tl

```

(End definition for `\l__msg_internal_tl`. This variable is documented on page ??.)

17.1 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

`\c__msg_text_prefix_tl` Locations for the text of messages.

```

7642 \tl_const:Nn \c__msg_text_prefix_tl { msg-text~>~ }
7643 \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

```

7644 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
7645 {
7646   \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
7647   { \prg_return_true: } { \prg_return_false: }
7648 }

```

(End definition for \msg_if_exist:nn. These functions are documented on page 142.)

__chk_if_free_msg:nn This auxiliary is similar to **__chk_if_free_cs:N**, and is used when defining messages with **\msg_new:nnnn**. It could be inlined in **\msg_new:nnnn**, but the experimental **l3trace** module needs to disable this check when reloading a package with the extra tracing information.

```

7649 \cs_new_protected:Npn \__chk_if_free_msg:nn #1#2
7650 {
7651   \msg_if_exist:nnT {#1} {#2}
7652   {
7653     \__msg_kernel_error:nxxx { kernel } { message-already-defined }
7654     {#1} {#2}
7655   }
7656 }
7657 <*package>
7658 \tex_ifodd:D \l@expl@log@functions@bool
7659 \cs_gset_protected:Npn \__chk_if_free_msg:nn #1#2
7660 {
7661   \msg_if_exist:nnT {#1} {#2}
7662   {
7663     \__msg_kernel_error:nxxx { kernel } { message-already-defined }
7664     {#1} {#2}
7665   }
7666   \iow_log:x { Defining~message~ #1 / #2 ~\msg_line_context: }
7667 }
7668 \fi:
7669 </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_gset:nnnn
\msg_set:nnnn
\msg_set:nnnn

```

7670 \cs_new_protected:Npn \msg_new:nnnn #1#2
7671 {
7672   \__chk_if_free_msg:nn {#1} {#2}
7673   \msg_gset:nnnn {#1} {#2}
7674 }
7675 \cs_new_protected:Npn \msg_new:nnn #1#2#3
7676 { \msg_new:nnnn {#1} {#2} {#3} { } }
7677 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
7678 {
7679   \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
7680   ##1##2##3##4 {#3}
7681   \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }

```

```

7682     ##1##2##3##4 {#4}
7683   }
7684   \cs_new_protected:Npn \msg_set:nnn #1#2#3
7685   { \msg_set:nnnn {#1} {#2} {#3} { } }
7686   \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
7687   {
7688     \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
7689     ##1##2##3##4 {#3}
7690     \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
7691     ##1##2##3##4 {#4}
7692   }
7693   \cs_new_protected:Npn \msg_gset:nnn #1#2#3
7694   { \msg_gset:nnnn {#1} {#2} {#3} { } }

```

(End definition for \msg_new:nnnn and \msg_new:nnn. These functions are documented on page ??.)

17.2 Messages: support functions and text

\c_msg_coding_error_text_tl
 \c_msg_continue_text_tl
 \c_msg_critical_text_tl
 \c_msg_fatal_text_tl
 \c_msg_help_text_tl
 \c_msg_no_info_text_tl
 \c_msg_on_line_text_tl
 \c_msg_return_text_tl
 \c_msg_trouble_text_tl

Simple pieces of text for messages.

```

7695 \tl_const:Nn \c_msg_coding_error_text_tl
7696 {
7697   This-is-a-coding-error.
7698   \\ \\
7699 }
7700 \tl_const:Nn \c_msg_continue_text_tl
7701 { Type~<return>~to~continue }
7702 \tl_const:Nn \c_msg_critical_text_tl
7703 { Reading~the~current~file~will~stop }
7704 \tl_const:Nn \c_msg_fatal_text_tl
7705 { This-is-a-fatal-error:~LaTeX~will~abort }
7706 \tl_const:Nn \c_msg_help_text_tl
7707 { For~immediate~help~type~H~<return> }
7708 \tl_const:Nn \c_msg_no_info_text_tl
7709 {
7710   LaTeX~does~not~know~anything~more~about~this~error,~sorry.
7711   \c_msg_return_text_tl
7712 }
7713 \tl_const:Nn \c_msg_on_line_text_tl { on~line }
7714 \tl_const:Nn \c_msg_return_text_tl
7715 {
7716   \\ \\
7717   Try~typing~<return>~to~proceed.
7718   \\
7719   If~that~doesn't~work,~type~X~<return>~to~quit.
7720 }
7721 \tl_const:Nn \c_msg_trouble_text_tl
7722 {
7723   \\ \\
7724   More~errors~will~almost~certainly~follow: \\
7725   the~LaTeX~run~should~be~aborted.

```

```
7726 }
```

(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.

```
7727 \cs_new_nopar:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
7728 \cs_gset_nopar:Npn \msg_line_context:
7729 {
7730   \c_msg_on_line_text_tl
7731   \c_space_tl
7732   \msg_line_number:
7733 }
```

(End definition for `\msg_line_number:` and `\msg_line_context:`. These functions are documented on page 142.)

17.3 Showing messages: low level mechanism

`\msg_interrupt:nnn` The low-level interruption macro is rather opaque, unfortunately. Depending on the availability of more information there is a choice of how to set up the further help. We feed the extra help text and the message itself to a wrapping auxiliary, in this order because we must first setup TeX's `\errhelp` register before issuing an `\errmessage`.

```
7734 \cs_new_protected:Npn \msg_interrupt:nnn #1#2#3
7735 {
7736   \tl_if_empty:nTF {#3}
7737   {
7738     \__msg_interrupt_wrap:nn { \ \c_msg_no_info_text_tl }
7739     {#1 \ \ \ \ #2 \ \ \ \ \c_msg_continue_text_tl }
7740   }
7741   {
7742     \__msg_interrupt_wrap:nn { \ \ #3 }
7743     {#1 \ \ \ \ #2 \ \ \ \ \c_msg_help_text_tl }
7744   }
7745 }
```

(End definition for `\msg_interrupt:nnn`. This function is documented on page 146.)

`__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.

```
7746 \cs_new_protected:Npn \__msg_interrupt_wrap:nn #1#2
7747 {
7748   \iow_wrap:nnnN {#1} { | ~ } { } \__msg_interrupt_more_text:n
7749   \iow_wrap:nnnN {#2} { ! ~ } { } \__msg_interrupt_text:n
7750 }
7751 \cs_new_protected:Npn \__msg_interrupt_more_text:n #1
7752 {
```

```

7753 \exp_args:Nx \tex_errhelp:D
7754 {
7755 |,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
7756 #1 \iow_newline:
7757 |.....
7758 }
7759 }

```

(End definition for `_msg_interrupt_wrap:nn`. This function is documented on page 146.)

`_msg_interrupt_text:n` The business end of the process starts by producing some visual separation of the message from the main part of the log. The error message needs to be printed with everything made “invisible”: 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.

```

7760 \group_begin:
7761 \char_set_lccode:nn {'\} {'\ }
7762 \char_set_lccode:nn {'\} {'\ }
7763 \char_set_lccode:nn {'&} {'!\}
7764 \char_set_catcode_active:N \&
7765 \tl_to_lowercase:n
7766 {
7767 \group_end:
7768 \cs_new_protected:Npn \_msg_interrupt_text:n #1
7769 {
7770 \iow_term:x
7771 {
7772 \iow_newline:
7773 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
7774 \iow_newline:
7775 !
7776 }
7777 \group_begin:
7778 \cs_set_protected_nopar:Npn &
7779 {
7780 \tex_errmessage:D
7781 {
7782 #1
7783 \use_none:n
7784 { ..... }
7785 }
7786 }
7787 \exp_after:wN
7788 \group_end:
7789 &
7790 }

```

```

7791 }
(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.

```

7792 \cs_new_protected:Npn \msg_log:n #1
7793 {
7794   \iow_log:n { ..... }
7795   \iow_wrap:nnnN { . ~ #1 } { . ~ } { } \iow_log:n
7796   \iow_log:n { ..... }
7797 }
7798 \cs_new_protected:Npn \msg_term:n #1
7799 {
7800   \iow_term:n { ***** }
7801   \iow_wrap:nnnN { * ~ #1 } { * ~ } { } \iow_term:n
7802   \iow_term:n { ***** }
7803 }
(End definition for \msg_log:n. This function is documented on page 147.)

```

17.4 Displaying messages

L^AT_EX is handling error messages and so the T_EX ones are disabled. This is already done by the L^AT_EX_{2 ϵ} kernel, so to avoid messing up any deliberate change by a user this is only set in format mode.

```

7804 <*initex>
7805 \int_gset_eq:NN \tex_errorcontextlines:D \c_minus_one
7806 </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`

```

7807 \cs_new:Npn \msg_fatal_text:n #1 { Fatal~#1~error }
7808 \cs_new:Npn \msg_critical_text:n #1 { Critical~#1~error }
7809 \cs_new:Npn \msg_error_text:n #1 { #1~error }
7810 \cs_new:Npn \msg_warning_text:n #1 { #1~warning }
7811 \cs_new:Npn \msg_info_text:n #1 { #1~info }
(End definition for \msg_fatal_text:n and others. These functions are documented on page 143.)

```

`\msg_see_documentation_text:n` Contextual footer information. The L^AT_EX module only comprises L^AT_EX₃ code, so we refer to the L^AT_EX₃ documentation rather than simply “L^AT_EX”.

```

7812 \cs_new:Npn \msg_see_documentation_text:n #1
7813 {
7814   \ \ See~the~
7815   \str_if_eq:nnTF {#1} { LaTeX } { LaTeX3 } {#1} ~
7816   documentation~for~further~information.
7817 }
(End definition for \msg_see_documentation_text:n. This function is documented on page 143.)

```

_msg_class_new:nn

```

7818 \group_begin:
7819 \cs_set_protected:Npn \_msg\_class\_new:nn #1#2
7820 {
7821   \prop_new:c { l\_msg\_redirect\_ #1\_prop }
7822   \cs_new_protected:cpn { \_msg\_ #1\_code:nnnnnn } ##1##2##3##4##5##6 {#2}
7823   \cs_new_protected:cpn { msg\_ #1 :nnnnnn } ##1##2##3##4##5##6
7824   {
7825     \use:x
7826     {
7827       \exp_not:n { \_msg\_use:nnnnnn {#1} {##1} {##2} }
7828       { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
7829       { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
7830     }
7831   }
7832   \cs_new_protected:cpx { msg\_ #1 :nnnnnn } ##1##2##3##4##5
7833   { \exp_not:c { msg\_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
7834   \cs_new_protected:cpx { msg\_ #1 :nnnn } ##1##2##3##4
7835   { \exp_not:c { msg\_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
7836   \cs_new_protected:cpx { msg\_ #1 :nnn } ##1##2##3
7837   { \exp_not:c { msg\_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
7838   \cs_new_protected:cpx { msg\_ #1 :nn } ##1##2
7839   { \exp_not:c { msg\_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
7840   \cs_new_protected:cpx { msg\_ #1 :nnxxxx } ##1##2##3##4##5##6
7841   {
7842     \use:x
7843     {
7844       \exp_not:N \exp_not:n
7845       { \exp_not:c { msg\_ #1 :nnnnnn } {##1} {##2} }
7846       {##3} {##4} {##5} {##6}
7847     }
7848   }
7849   \cs_new_protected:cpx { msg\_ #1 :nnxxx } ##1##2##3##4##5
7850   { \exp_not:c { msg\_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
7851   \cs_new_protected:cpx { msg\_ #1 :nnxx } ##1##2##3##4
7852   { \exp_not:c { msg\_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
7853   \cs_new_protected:cpx { msg\_ #1 :nnx } ##1##2##3
7854   { \exp_not:c { msg\_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
7855 }

```

(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
 \msg_fatal:nnnn
 \msg_fatal:nnn
 \msg_fatal:nn
 \msg_fatal:nnxxxx
 \msg_fatal:nnxxx
 \msg_fatal:nnxx
 \msg_fatal:nnx

```

7856 \_msg\_class\_new:nn { fatal }
7857 {
7858   \msg_interrupt:nnn
7859   { \msg_fatal_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_fatal_text_tl }
7865     \tex_end:D
7866 }

```

(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 7867 \__msg_class_new:nn { critical }
\msg_critical:nnnnn 7868 {
\msg_critical:nnnn 7869   \msg_interrupt:nnn
\msg_critical:nn 7870   { \msg_critical_text:n {#1} : ~ "#2" }
\msg_critical:nnxxx 7871   {
\msg_critical:nnxxx 7872     \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
\msg_critical:nnxx 7873     \msg_see_documentation_text:n {#1}
\msg_critical:nnx 7874   }
7875   { \c_msg_critical_text_tl }
7876   \tex_endinput:D
7877 }

```

(End definition for `\msg_critical:nnnnnn` and others. These functions are documented on page ??.)

`\msg_error:nnnnnn` For an error, the interrupt routine is called. We check if there is a “more text” by comparing that control sequence with a permanently empty text.

```

\msg_error:nnnnnn 7878 \__msg_class_new:nn { error }
\msg_error:nnnnn 7879 {
\msg_error:nnnn 7880   \__msg_error:cnnnnn
\msg_error:nnxxx 7881   { \c__msg_more_text_prefix_tl #1 / #2 }
\msg_error:nnxxx 7882   {#3} {#4} {#5} {#6}
\msg_error:nnxx 7883   {
\msg_error:nnx 7884     \msg_interrupt:nnn
7885     { \msg_error_text:n {#1} : ~ "#2" }
7886     {
7887       \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
7888       \msg_see_documentation_text:n {#1}
7889     }
7890   }
7891 }
7892 \cs_new_protected:Npn \__msg_error:cnnnnn #1#2#3#4#5#6
7893 {
7894   \cs_if_eq:cNTF {#1} \__msg_no_more_text:nnnn
7895   { #6 { } }
7896   { #6 { \use:c {#1} {#2} {#3} {#4} {#5} } }
7897 }
7898 \cs_new:Npn \__msg_no_more_text:nnnn #1#2#3#4 { }

```

(End definition for `\msg_error:nnnnnn` and others. These functions are documented on page ??.)

`\msg_warning:nnnnnn` Warnings are printed to the terminal.

```

\msg_warning:nnnnnn 7899 \__msg_class_new:nn { warning }
\msg_warning:nnnn 7900 {
\msg_warning:nnn 7901   \msg_term:n
\msg_warning:nn
\msg_warning:nnxxx
\msg_warning:nnxxx
\msg_warning:nnxx
\msg_warning:nnx

```


`\l__msg_redirect_prop` For redirection of individually-named messages

7930 `\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 `{}`.

7931 `\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.

7932 `\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.

```

7933 \cs_new_protected:Npn \__msg_use:nnnnnnn #1#2#3#4#5#6#7
7934 {
7935   \msg_if_exist:nnTF {#2} {#3}
7936   {
7937     \__msg_class_chk_exist:nT {#1}
7938     {
7939       \tl_set:Nn \l__msg_current_class_tl {#1}
7940       \cs_set_protected_nopar:Npx \__msg_use_code:
7941       {
7942         \exp_not:n
7943         {
7944           \use:c { __msg_ \l__msg_class_tl _code:nnnnnnn }
7945           {#2} {#3} {#4} {#5} {#6} {#7}
7946         }
7947       }
7948       \__msg_use_redirect_name:n { #2 / #3 }
7949     }
7950   }
7951   { \__msg_kernel_error:nnxx { kernel } { message-unknown } {#2} {#3} }
7952 }
7953 \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.

```

7954 \cs_new_protected:Npn \__msg_use_redirect_name:n #1
7955 {
7956   \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
```

```

7957     { \_msg_use_code: }
7958     {
7959         \seq_clear:N \l__msg_hierarchy_seq
7960         \_msg_use_hierarchy:nwN { }
7961         #1 \q_mark \_msg_use_hierarchy:nwN
7962         / \q_mark \use_none_delimit_by_q_stop:w
7963         \q_stop
7964         \_msg_use_redirect_module:n { }
7965     }
7966 }
7967 \cs_new_protected:Npn \_msg_use_hierarchy:nwN #1#2 / #3 \q_mark #4
7968 {
7969     \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
7970     #4 { #1 / #2 } #3 \q_mark #4
7971 }

```

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`.

```

7972 \cs_new_protected:Npn \_msg_use_redirect_module:n #1
7973 {
7974     \seq_map_inline:Nn \l__msg_hierarchy_seq
7975     {
7976         \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
7977         {##1} \l__msg_class_tl
7978         {
7979             \seq_map_break:n
7980             {
7981                 \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
7982                 { \_msg_use_code: }
7983                 {
7984                     \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
7985                     \_msg_use_redirect_module:n {##1}
7986                 }
7987             }
7988         }
7989         {
7990             \str_if_eq:nnT {##1} {#1}
7991             {
7992                 \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
7993                 \seq_map_break:n { \_msg_use_code: }
7994             }

```

```

7995         }
7996     }
7997 }

```

(End definition for `_msg_use:nnnnnnn`. This function is documented on page ??.)

`\msg_redirect_name:nnn` Named message will always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```

7998 \cs_new_protected:Npn \msg\_redirect\_name:nnn #1#2#3
7999 {
8000     \tl\_if\_empty:nTF {#3}
8001     { \prop\_remove:Nn \l\_msg\_redirect\_prop { / #1 / #2 } }
8002     {
8003         \\_msg\_class\_chk\_exist:nT {#3}
8004         { \prop\_put:Nnn \l\_msg\_redirect\_prop { / #1 / #2 } {#3} }
8005     }
8006 }

```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page 146.)

`\msg_redirect_class:nn` If the target class is empty, eliminate the corresponding redirection. Otherwise, add the
`\msg_redirect_module:nnn` 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:nn
8007 \cs\_new\_protected\_nopar:Npn \msg\_redirect\_class:nn
8008 { \\_msg\_redirect:nnn { } }
8009 \cs\_new\_protected:Npn \msg\_redirect\_module:nnn #1
8010 { \\_msg\_redirect:nnn { / #1 } }
8011 \cs\_new\_protected:Npn \\_msg\_redirect:nnn #1#2#3
8012 {
8013     \\_msg\_class\_chk\_exist:nT {#2}
8014     {
8015         \tl\_if\_empty:nTF {#3}
8016         { \prop\_remove:cn { l\_msg\_redirect\_ #2\_prop } {#1} }
8017         {
8018             \\_msg\_class\_chk\_exist:nT {#3}
8019             {
8020                 \prop\_put:cn { l\_msg\_redirect\_ #2\_prop } {#1} {#3}
8021                 \tl\_set:Nn \l\_msg\_current\_class\_tl {#2}
8022                 \seq\_clear:N \l\_msg\_class\_loop\_seq
8023                 \\_msg\_redirect\_loop\_chk:nnn {#2} {#3} {#1}
8024             }
8025         }
8026     }
8027 }

```

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.

```

8028 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
8029 {
8030   \seq_put_right:Nn \l__msg_class_loop_seq {#1}
8031   \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
8032   {
8033     \str_if_eq:x:nnF { \l__msg_class_tl } {#1}
8034     {
8035       \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
8036       {
8037         \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
8038         \__msg_kernel_warning:nnxxx { kernel } { message-redirect-loop }
8039         { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
8040         { \seq_item:Nn \l__msg_class_loop_seq { \c_two } }
8041         {#3}
8042         {
8043           \seq_map_function:NN \l__msg_class_loop_seq
8044             \__msg_redirect_loop_list:n
8045             { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
8046         }
8047       }
8048       { \__msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
8049     }
8050   }
8051 }
8052 \cs_generate_variant:Nn \__msg_redirect_loop_chk:nnn { o }
8053 \cs_new:Npn \__msg_redirect_loop_list:n #1 { {#1} ~ => ~ }

```

(End definition for \msg_redirect_class:nn and \msg_redirect_module:nnn. These functions are documented on page 146.)

17.5 Kernel-specific functions

The kernel needs some messages of its own. These are created using pre-built functions. Two functions are provided: one more general and one which only has the short text part.

```

\__msg_kernel_new:nnnn
\__msg_kernel_new:nnn
\__msg_kernel_set:nnnn
\__msg_kernel_set:nnn
8054 \cs_new_protected:Npn \__msg_kernel_new:nnnn #1#2
8055 { \msg_new:nnnn { LaTeX } { #1 / #2 } }
8056 \cs_new_protected:Npn \__msg_kernel_new:nnn #1#2
8057 { \msg_new:nnn { LaTeX } { #1 / #2 } }
8058 \cs_new_protected:Npn \__msg_kernel_set:nnnn #1#2
8059 { \msg_set:nnnn { LaTeX } { #1 / #2 } }
8060 \cs_new_protected:Npn \__msg_kernel_set:nnn #1#2
8061 { \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.

```

8062 \group_begin:
8063 \cs_set_protected:Npn \_msg_kernel_class_new:nN #1
8064 { \_msg_kernel_class_new_aux:nN { kernel_ #1 } }
8065 \cs_set_protected:Npn \_msg_kernel_class_new_aux:nN #1#2
8066 {
8067   \cs_new_protected:cpn { \_msg_ #1 :nnnnnn } ##1##2##3##4##5##6
8068   {
8069     \use:x
8070     {
8071       \exp_not:n { #2 { LaTeX } { ##1 / ##2 } }
8072       { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
8073       { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
8074     }
8075   }
8076   \cs_new_protected:cpx { \_msg_ #1 :nnnnn } ##1##2##3##4##5
8077   { \exp_not:c { \_msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
8078   \cs_new_protected:cpx { \_msg_ #1 :nnnn } ##1##2##3##4
8079   { \exp_not:c { \_msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
8080   \cs_new_protected:cpx { \_msg_ #1 :nnn } ##1##2##3
8081   { \exp_not:c { \_msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
8082   \cs_new_protected:cpx { \_msg_ #1 :nn } ##1##2
8083   { \exp_not:c { \_msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
8084   \cs_new_protected:cpx { \_msg_ #1 :nnxxxx } ##1##2##3##4##5##6
8085   {
8086     \use:x
8087     {
8088       \exp_not:N \exp_not:n
8089       { \exp_not:c { \_msg_ #1 :nnnnnn } {##1} {##2} }
8090       {##3} {##4} {##5} {##6}
8091     }
8092   }
8093   \cs_new_protected:cpx { \_msg_ #1 :nnxxx } ##1##2##3##4##5
8094   { \exp_not:c { \_msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
8095   \cs_new_protected:cpx { \_msg_ #1 :nnxx } ##1##2##3##4
8096   { \exp_not:c { \_msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
8097   \cs_new_protected:cpx { \_msg_ #1 :nnx } ##1##2##3
8098   { \exp_not:c { \_msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
8099 }

```

(End definition for `_msg_kernel_class_new:nN`. This function is documented on page ??.)

`_msg_kernel_fatal:nnnnnn`
`_msg_kernel_fatal:nnnnn`
`_msg_kernel_fatal:nnnn`
`_msg_kernel_fatal:nnnn`
`_msg_kernel_fatal:nn`
`_msg_kernel_fatal:nn`
`_msg_kernel_fatal:nnxxxx`
`_msg_kernel_fatal:nnxxxx`
`_msg_kernel_fatal:nnxxx`
`_msg_kernel_fatal:nnx`
`_msg_kernel_error:nnnnnn`
`_msg_kernel_error:nnnnn`
`_msg_kernel_error:nnnn`
`_msg_kernel_error:nnnn`
`_msg_kernel_error:nn`
`_msg_kernel_error:nn`
`_msg_kernel_error:nnxxxx`

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.

```

8100 \_msg_kernel_class_new:nN { fatal } \_msg_fatal_code:nnnnnn
8101 \cs_undefine:N \_msg_kernel_error:nnxx

```

```

8102 \cs_undefine:N \__msg_kernel_error:nnx
8103 \cs_undefine:N \__msg_kernel_error:nn
8104 \__msg_kernel_class_new:nN { error } \__msg_error_code:nnnnnn

```

(End definition for __msg_kernel_fatal:nnnnnn and others. These functions are documented on page ??.)

```

\__msg_kernel_warning:nnnnnn
\__msg_kernel_warning:nnnnn
\__msg_kernel_warning:nnnn
\__msg_kernel_warning:nnn
\__msg_kernel_warning:nn
\__msg_kernel_warning:nnxxx
\__msg_kernel_warning:nnxxx
\__msg_kernel_warning:nnxxx
\__msg_kernel_warning:nnxx
\__msg_kernel_warning:nnxx
\__msg_kernel_warning:nnx
\__msg_kernel_info:nnnnnn
\__msg_kernel_info:nnnnn
\__msg_kernel_info:nnnn
\__msg_kernel_info:nnn
\__msg_kernel_info:nn
\__msg_kernel_info:nnxxx
\__msg_kernel_info:nnxxx
\__msg_kernel_info:nnxx
\__msg_kernel_info:nnx

```

Kernel messages which can be redirected simply use the machinery for normal messages, with the module name “LaTeX”.

```

8105 \__msg_kernel_class_new:nN { warning } \msg_warning:nnxxxx
8106 \__msg_kernel_class_new:nN { info } \msg_info:nnxxxx

```

(End definition for __msg_kernel_warning:nnnnnn and others. These functions are documented on page ??.)

End the group to eliminate __msg_kernel_class_new:nN.

```

8107 \group_end:

```

Error messages needed to actually implement the message system itself.

```

8108 \__msg_kernel_new:nnnn { kernel } { message-already-defined }
8109 { Message~'#2'~for-module~'#1'~already-defined. }
8110 {
8111   \c_msg_coding_error_text_tl
8112   LaTeX~was~asked~to~define~a~new~message~called~'#2'\
8113   by~the~module~'#1':~this~message~already~exists.
8114   \c_msg_return_text_tl
8115 }
8116 \__msg_kernel_new:nnnn { kernel } { message-unknown }
8117 { Unknown~message~'#2'~for-module~'#1'. }
8118 {
8119   \c_msg_coding_error_text_tl
8120   LaTeX~was~asked~to~display~a~message~called~'#2'\
8121   by~the~module~'#1':~this~message~does~not~exist.
8122   \c_msg_return_text_tl
8123 }
8124 \__msg_kernel_new:nnnn { kernel } { message-class-unknown }
8125 { Unknown~message~class~'#1'. }
8126 {
8127   LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\
8128   this~was~never~defined.
8129   \c_msg_return_text_tl
8130 }
8131 \__msg_kernel_new:nnnn { kernel } { message-redirect-loop }
8132 {
8133   Message~redirection~loop~caused~by~ {#1} ~>~ {#2}
8134   \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
8135 }
8136 {
8137   Adding~the~message~redirection~ {#1} ~>~ {#2}
8138   \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
8139   created~an~infinite~loop\\\\
8140   \iow_indent:n { #4 \\\ }
8141 }

```


Messages for earlier kernel modules.

```

8142 \_msg_kernel_new:nnnn { kernel } { bad-number-of-arguments }
8143 { Function~'~#1'~cannot-be-defined-with~#2~arguments. }
8144 {
8145   \c_msg_coding_error_text_tl
8146   LaTeX~has~been~asked~to~define~a~function~'~#1'~with~
8147   #2~arguments.~
8148   TeX~allows~between~0~and~9~arguments~for~a~single~function.
8149 }
8150 \_msg_kernel_new:nnnn { kernel } { command-already-defined }
8151 { Control~sequence~#1~already~defined. }
8152 {
8153   \c_msg_coding_error_text_tl
8154   LaTeX~has~been~asked~to~create~a~new~control~sequence~'~#1'~
8155   but~this~name~has~already~been~used~elsewhere. \\ \\
8156   The~current~meaning~is:\\
8157   \ \ #2
8158 }
8159 \_msg_kernel_new:nnnn { kernel } { command-not-defined }
8160 { Control~sequence~#1~undefined. }
8161 {
8162   \c_msg_coding_error_text_tl
8163   LaTeX~has~been~asked~to~use~a~command~#1,~but~this~has~not~
8164   been~defined~yet.
8165 }
8166 \_msg_kernel_new:nnnn { kernel } { empty-search-pattern }
8167 { Empty~search~pattern. }
8168 {
8169   \c_msg_coding_error_text_tl
8170   LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'~#1':~that~
8171   would~lead~to~an~infinite~loop!
8172 }
8173 \_msg_kernel_new:nnnn { kernel } { out-of-registers }
8174 { No~room~for~a~new~#1. }
8175 {
8176   TeX~only~supports~\int\_use:N \c\_max\_register\_int \
8177   of~each~type.~All~the~#1~registers~have~been~used.~
8178   This~run~will~be~aborted~now.
8179 }
8180 \_msg_kernel_new:nnnn { kernel } { missing-colon }
8181 { Function~'~#1'~contains~no~':'. }
8182 {
8183   \c_msg_coding_error_text_tl
8184   Code~level~functions~must~contain~': '~to~separate~the~
8185   argument~specification~from~the~function~name.~This~is~
8186   needed~when~defining~conditionals~or~variants,~or~when~building~a~
8187   parameter~text~from~the~number~of~arguments~of~the~function.
8188 }
8189 \_msg_kernel_new:nnnn { kernel } { protected-predicate }

```

```

8190 { Predicate~'#1'~must~be~expandable. }
8191 {
8192   \c_msg_coding_error_text_tl
8193   LaTeX~has~been~asked~to~define~'#1'~as~a~protected~predicate..
8194   Only~expandable~tests~can~have~a~predicate~version.
8195 }
8196 \_msg_kernel_new:nnnn { kernel } { conditional-form-unknown }
8197 { Conditional~form~'#1'~for~function~'#2'~unknown. }
8198 {
8199   \c_msg_coding_error_text_tl
8200   LaTeX~has~been~asked~to~define~the~conditional~form~'#1'~of~
8201   the~function~'#2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.
8202 }
8203 \_msg_kernel_new:nnnn { kernel } { scanmark-already-defined }
8204 { Scan~mark~'#1'~already~defined. }
8205 {
8206   \c_msg_coding_error_text_tl
8207   LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~
8208   but~this~name~has~already~been~used~for~a~scan~mark.
8209 }
8210 \_msg_kernel_new:nnnn { kernel } { variable-not-defined }
8211 { Variable~#1~undefined. }
8212 {
8213   \c_msg_coding_error_text_tl
8214   LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
8215   been~defined~yet.
8216 }
8217 \_msg_kernel_new:nnnn { kernel } { variant-too-long }
8218 { Variant~form~'#1'~longer~than~base~signature~of~'#2'. }
8219 {
8220   \c_msg_coding_error_text_tl
8221   LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
8222   with~a~signature~starting~with~'#1',~but~that~is~longer~than~
8223   the~signature~(part~after~the~colon)~of~'#2'.
8224 }
8225 \_msg_kernel_new:nnnn { kernel } { invalid-variant }
8226 { Variant~form~'#1'~invalid~for~base~form~'#2'. }
8227 {
8228   \c_msg_coding_error_text_tl
8229   LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
8230   with~a~signature~starting~with~'#1',~but~cannot~change~an~argument~
8231   from~type~'#3'~to~type~'#4'.
8232 }

```

Some errors only appear in expandable settings, hence don't need a "more-text" argument.

```

8233 \_msg_kernel_new:nnn { kernel } { bad-variable }
8234 { Erroneous~variable~#1~used! }
8235 \_msg_kernel_new:nnn { kernel } { misused-sequence }
8236 { A~sequence~was~misused. }

```

```

8237 \_msg_kernel_new:nnn { kernel } { negative-replication }
8238 { Negative~argument~for~\prg_replicate:nn. }
8239 \_msg_kernel_new:nnn { kernel } { unknown-comparison }
8240 { Relation~symbol~'~#1'~unknown:~use~=,~<,~>,~==,~!=,~<=,~>=. }
8241 \_msg_kernel_new:nnn { kernel } { zero-step }
8242 { Zero~step~size~for~step~function~#1. }

Messages used by the “show” functions.

8243 \_msg_kernel_new:nnn { kernel } { show-clist }
8244 {
8245   The~comma~list~
8246   \str_if_eq:nnF {#1} { \l__clist_internal_clist } { \token_to_str:N #1~}
8247   \clist_if_empty:NTF #1
8248   { is~empty }
8249   { contains~the~items~(without~outer~braces): }
8250 }
8251 \_msg_kernel_new:nnn { kernel } { show-prop }
8252 {
8253   The~property~list~\token_to_str:N #1~
8254   \prop_if_empty:NTF #1
8255   { is~empty }
8256   { contains~the~pairs~(without~outer~braces): }
8257 }
8258 \_msg_kernel_new:nnn { kernel } { show-seq }
8259 {
8260   The~sequence~\token_to_str:N #1~
8261   \seq_if_empty:NTF #1
8262   { is~empty }
8263   { contains~the~items~(without~outer~braces): }
8264 }
8265 \_msg_kernel_new:nnn { kernel } { show-no-stream }
8266 { No~ #1 ~streams~are~open }
8267 \_msg_kernel_new:nnn { kernel } { show-open-streams }
8268 { The~following~ #1 ~streams~are~in~use: }

```

17.6 Expandable errors

`_msg_expandable_error:n` In expansion only context, we cannot use the normal means of reporting errors. Instead, we feed \TeX an undefined control sequence, `\LaTeX3 error:.` It is thus interrupted, and shows the context, which thanks to the odd-looking `\use:n` is

```

<argument> \LaTeX3 error:
                The error message.

```

In other words, \TeX is processing the argument of `\use:n`, which is `\LaTeX3 error: <error message>`. Then `_msg_expandable_error:w` cleans up. In fact, there is an extra subtlety: if the user inserts tokens for error recovery, they should be kept. Thus we also use an odd space character (with category code 7) and keep tokens until that space character, dropping everything else until `\q_stop`. The `\c_zero` prevents losing braces around the user-inserted text if any, and stops the expansion of `\romannumeral`.

```

8269 \group_begin:
8270 \char_set_catcode_math_superscript:N \^
8271 \char_set_lccode:nn { '^ } { '\ }
8272 \char_set_lccode:nn { 'L } { 'L }
8273 \char_set_lccode:nn { 'T } { 'T }
8274 \char_set_lccode:nn { 'X } { 'X }
8275 \tl_to_lowercase:n
8276 {
8277   \cs_new:Npx \_msg_expandable_error:n #1
8278   {
8279     \exp_not:n
8280     {
8281       \tex_romannumeral:D
8282       \exp_after:wN \exp_after:wN
8283       \exp_after:wN \_msg_expandable_error:w
8284       \exp_after:wN \exp_after:wN
8285       \exp_after:wN \c_zero
8286     }
8287     \exp_not:N \use:n { \exp_not:c { LaTeX3-error: } ^ #1 } ^
8288   }
8289   \cs_new:Npn \_msg_expandable_error:w #1 ^ #2 ^ { #1 }
8290 }
8291 \group_end:

```

(End definition for `_msg_expandable_error:n`. This function is documented on page 149.)

`_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`.

```

8292 \cs_new:Npn \_msg_kernel_expandable_error:nnnnnn #1#2#3#4#5#6
8293 {
8294   \exp_args:Nf \_msg_expandable_error:n
8295   {
8296     \exp_args:NNc \exp_after:wN \exp_stop_f:
8297     { \c_@_msg_text_prefix_tl LaTeX / #1 / #2 }
8298     {#3} {#4} {#5} {#6}
8299   }
8300 }
8301 \cs_new:Npn \_msg_kernel_expandable_error:nnnnn #1#2#3#4#5
8302 {
8303   \_msg_kernel_expandable_error:nnnnnn
8304   {#1} {#2} {#3} {#4} {#5} { }
8305 }
8306 \cs_new:Npn \_msg_kernel_expandable_error:nnnn #1#2#3#4
8307 {
8308   \_msg_kernel_expandable_error:nnnnnn
8309   {#1} {#2} {#3} {#4} { } { }
8310 }
8311 \cs_new:Npn \_msg_kernel_expandable_error:nnn #1#2#3
8312 {
8313   \_msg_kernel_expandable_error:nnnnnn

```

```

8314     {#1} {#2} {#3} { } { } { }
8315   }
8316   \cs_new:Npn \__msg_kernel_expandable_error:nn #1#2
8317   {
8318     \__msg_kernel_expandable_error:nnnnnn
8319     {#1} {#2} { } { } { } { }
8320   }

```

(End definition for `__msg_kernel_expandable_error:nnnnnn` and others. These functions are documented on page ??.)

17.7 Showing variables

Functions defined in this section are used for diagnostic functions in `l3clist`, `l3file`, `l3prop`, `l3seq`, `xtemplate`

`__msg_term:nnnnnn` Print the text of a message to the terminal without formatting: short cuts around `\iow_wrap:nnnN`.

```

\__msg_term:nnnnnV
\__msg_term:nnnnn
\__msg_term:nnn
\__msg_term:nn
8321 \cs_new_protected:Npn \__msg_term:nnnnnn #1#2#3#4#5#6
8322 {
8323   \iow_wrap:nnnN
8324   { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
8325   { } { } \iow_term:n
8326 }
8327 \cs_generate_variant:Nn \__msg_term:nnnnnn { nnnnnV }
8328 \cs_new_protected:Npn \__msg_term:nnnnn #1#2#3#4#5
8329 { \__msg_term:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
8330 \cs_new_protected:Npn \__msg_term:nnn #1#2#3
8331 { \__msg_term:nnnnnn {#1} {#2} {#3} { } { } { } }
8332 \cs_new_protected:Npn \__msg_term:nn #1#2
8333 { \__msg_term:nnnnnn {#1} {#2} { } { } { } { } }

```

(End definition for `__msg_term:nnnnnn` and `__msg_term:nnnnnV`. These functions are documented on page ??.)

`__msg_show_variable:Nnn` 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.

```

8334 \cs_new_protected:Npn \__msg_show_variable:Nnn #1#2#3
8335 {
8336   \cs_if_exist:NTF #1
8337   {
8338     \__msg_term:nnn { LaTeX / kernel } { show- #2 } {#1}
8339     \__msg_show_variable:n {#3}
8340   }
8341   {
8342     \__msg_kernel_error:nnx { kernel } { variable-not-defined }
8343     { \token_to_str:N #1 }
8344   }
8345 }
8346 \cs_new_protected:Npn \__msg_show_variable:n #1
8347 { \iow_wrap:nnnN {#1} { } { } \__msg_show_variable_aux:n }
8348 \cs_new_protected:Npn \__msg_show_variable_aux:n #1
8349 {
8350   \tl_if_empty:nTF {#1}
8351   { \tl_clear:N \l__msg_internal_tl }
8352   { \tl_set:Nf \l__msg_internal_tl { \__msg_show_variable_aux:w #1 } }
8353   \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
8354   { \exp_after:wN \l__msg_internal_tl }
8355 }
8356 \cs_new:Npn \__msg_show_variable_aux:w #1 > { }

```

(End definition for `__msg_show_variable:Nnn`. This function is documented on page 149.)

`__msg_show_item:n`
`__msg_show_item:nn`
`__msg_show_item_unbraced:nn`

Each item in the variable is formatted using one of the following functions.

```

8357 \cs_new:Npn \__msg_show_item:n #1
8358 {
8359   \> \ \ \ { \tl_to_str:n {#1} \}
8360 }
8361 \cs_new:Npn \__msg_show_item:nn #1#2
8362 {
8363   \> \ \ \ { \tl_to_str:n {#1} \}
8364   \ \ => \ \ \ { \tl_to_str:n {#2} \}
8365 }
8366 \cs_new:Npn \__msg_show_item_unbraced:nn #1#2
8367 {
8368   \> \ \ \ { \tl_to_str:n {#1}
8369   \ \ => \ \ \ { \tl_to_str:n {#2}
8370   }

```

(End definition for `__msg_show_item:n`. This function is documented on page 149.)

17.8 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

```

\msg_class_new:nn This is only ever used in a set fashion.
8371 \*deprecated
8372 \cs_new_eq:NN \msg_class_new:nn \msg_class_set:nn
8373 \deprecated
(End definition for \msg_class_new:nn. This function is documented on page ??.)

\msg_trace:nnxxxx The performance here is never going to be good enough for tracing code, so let's be
\msg_trace:nnxxx realistic.
\msg_trace:nnxx
\msg_trace:nnx
\msg_trace:nn
8374 \*deprecated
8375 \cs_new_eq:NN \msg_trace:nnxxxx \msg_log:nnxxxx
8376 \cs_new_eq:NN \msg_trace:nnxxx \msg_log:nnxxx
8377 \cs_new_eq:NN \msg_trace:nnxx \msg_log:nnxx
8378 \cs_new_eq:NN \msg_trace:nnx \msg_log:nnx
8379 \cs_new_eq:NN \msg_trace:nn \msg_log:nn
8380 \deprecated
(End definition for \msg_trace:nnxxxx and others. These functions are documented on page ??.)

\msg_generic_new:nnn These were all too low-level.
\msg_generic_new:nn
\msg_generic_set:nnn
\msg_generic_set:nn
\msg_direct_interrupt:xxxxx
\msg_direct_log:xx
\msg_direct_term:xx
8381 \*deprecated
8382 \cs_new_protected:Npn \msg_generic_new:nnn #1#2#3 { \deprecated }
8383 \cs_new_protected:Npn \msg_generic_new:nn #1#2 { \deprecated }
8384 \cs_new_protected:Npn \msg_generic_set:nnn #1#2#3 { \deprecated }
8385 \cs_new_protected:Npn \msg_generic_set:nn #1#2 { \deprecated }
8386 \cs_new_protected:Npn \msg_direct_interrupt:xxxxx #1#2#3#4#5 { \deprecated }
8387 \cs_new_protected:Npn \msg_direct_log:xx #1#2 { \deprecated }
8388 \cs_new_protected:Npn \msg_direct_term:xx #1#2 { \deprecated }
8389 \deprecated
(End definition for \msg_generic_new:nnn. This function is documented on page ??.)

\__msg_kernel_bug:x
\c_msg_kernel_bug_text_tl
\c_msg_kernel_bug_more_text_tl
8390 \*deprecated
8391 \cs_set_protected:Npn \__msg_kernel_bug:x #1
8392 {
8393   \msg_interrupt:nnn { \c_msg_kernel_bug_text_tl }
8394   {
8395     #1
8396     \msg_see_documentation_text:n { LaTeX3 }
8397   }
8398   { \c_msg_kernel_bug_more_text_tl }
8399 }
8400 \tl_const:Nn \c_msg_kernel_bug_text_tl
8401 { This~is~a~LaTeX~bug:~check~coding! }
8402 \tl_const:Nn \c_msg_kernel_bug_more_text_tl
8403 {
8404   There~is~a~coding~bug~somewhere~around~here. \\
8405   This~probably~needs~examining~by~an~expert.
8406   \c_msg_return_text_tl
8407 }
8408 \deprecated

```

(End definition for _msg_kernel_bug:x. This function is documented on page ??.)

Deprecated on 2012-06-28, for removal by 2012-12-31.

```
\msg_newline: New lines are printed in the same way as for low-level file writing.
\msg_two_newlines: 8409 \*deprecated
                   8410 \cs_new_nopar:Npn \msg_newline: { ^^J }
                   8411 \cs_new_nopar:Npn \msg_two_newlines: { ^^J ^^J }
                   8412 \</deprecated>
(End definition for \msg_newline: and \msg_two_newlines:. These functions are documented on page
??.)
```

```
\msg_log:x These were all misnamed.
\msg_term:x 8413 \*deprecated
\msg_interrupt:xxx 8414 \cs_generate_variant:Nn \msg_log:n { x }
                   8415 \cs_generate_variant:Nn \msg_term:n { x }
                   8416 \cs_generate_variant:Nn \msg_interrupt:nnn { xxx }
                   8417 \</deprecated>
(End definition for \msg_log:x and \msg_term:x. These functions are documented on page ??.)
Deprecated on 2012-06-29, for removal by 2012-12-31.
```

\msg_class_set:nn Setting up a message class does two tasks. Any existing redirection is cleared, and the various message functions are created to simply use the code stored for the message.

```
8418 \*deprecated
8419 \cs_new_protected:Npn \msg_class_set:nn #1#2
8420 {
8421   \cs_if_exist:cTF { __msg_ #1 _code:nnnnnn }
8422   \cs_set_protected:cpn
8423   \cs_new_protected:cpn
8424   { __msg_ #1 _code:nnnnnn } ##1##2##3##4##5##6 {#2}
8425   \prop_clear_new:c { l__msg_redirect_ #1 _prop }
8426   \cs_set_protected_nopar:cpn { msg_ #1 :nnxxxx }
8427   { \__msg_use:nnnnnn {#1} }
8428   \cs_set_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
8429   { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
8430   \cs_set_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
8431   { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
8432   \cs_set_protected:cpx { msg_ #1 :nnx } ##1##2##3
8433   { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
8434   \cs_set_protected:cpx { msg_ #1 :nn } ##1##2
8435   { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} { } { } { } { } }
8436 }
8437 \</deprecated>
```

(End definition for \msg_class_set:nn. This function is documented on page ??.)

```
8438 \</initex | package>
```


18 l3keys Implementation

```

8439 <*initex | package>
8440 <*package>
8441 \ProvidesExplPackage
8442   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
8443   \__expl_package_check:
8444 </package>

```

18.1 Low-level interface

```

8445 <@@=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.

```

8446 \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`

```

8447 \tl_new:N \l__keyval_key_tl

```

```

8448 \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`

```

8449 \tl_new:N \l__keyval_sanitise_tl

```

```

8450 \tl_new:N \l__keyval_parse_tl

```

(End definition for `\l__keyval_sanitise_tl`. This function is documented on page ??.)

`__keyval_parse:n` The parsing function first deals with the category codes for = and , , so that there are no odd events. The input is then handed off to the element by element system.

```

8451 \group_begin:
8452   \char_set_catcode_active:n { '=' }
8453   \char_set_catcode_active:n { ',' }
8454   \char_set_lccode:nn { '\8 } { '=' }
8455   \char_set_lccode:nn { '\9 } { ',' }
8456   \tl_to_lowercase:n
8457   {
8458     \group_end:
8459     \cs_new_protected:Npn \__keyval_parse:n #1
8460     {
8461       \group_begin:
8462       \tl_clear:N \l__keyval_sanitise_tl
8463       \tl_set:Nn \l__keyval_sanitise_tl {#1}
8464       \tl_replace_all:Nnn \l__keyval_sanitise_tl { = } { 8 }
8465       \tl_replace_all:Nnn \l__keyval_sanitise_tl { , } { 9 }
8466       \tl_clear:N \l__keyval_parse_tl
8467       \exp_after:wN \__keyval_parse_elt:w \exp_after:wN
8468       \q_nil \l__keyval_sanitise_tl 9 \q_recursion_tail 9 \q_recursion_stop

```

```

8469         \exp_after:wN \group_end:
8470         \l__keyval_parse_tl
8471     }
8472 }

```

(End definition for __keyval_parse:n. This function is documented on page ??.)

__keyval_parse_elt:w Each item to be parsed will have \q_nil added to the front. Hence the blank test here can always be used to find a totally empty argument. If this is the case, the system loops round. If there is something to parse, there is a check for the end of the input before handing off.

```

8473 \cs_new_protected:Npn \__keyval_parse_elt:w #1 ,
8474 {
8475     \tl_if_blank:oTF { \use_none:n #1 }
8476     { \__keyval_parse_elt:w \q_nil }
8477     {
8478         \quark_if_recursion_tail_stop:o { \use_ii:nn #1 }
8479         \__keyval_split_key_value:w #1 == \q_stop
8480         \__keyval_parse_elt:w \q_nil
8481     }
8482 }

```

(End definition for __keyval_parse_elt:w. This function is documented on page ??.)

__keyval_split_key_value:w The key and value are handled separately. First the key is grabbed and saved as \l__keyval_key_tl. Then a check is need to see if there is a value at all: if not then the key name is simply added to the output. If there is a value then there is a check to ensure that there was only one = in the input (remembering some extra ones are around at the moment to prevent errors). All being well, there is an hand-off to find the value: the \q_nil is there to prevent loss of braces.

__keyval_split_key_value:wTF

```

8483 \cs_new_protected:Npn \__keyval_split_key_value:w #1 = #2 \q_stop
8484 {
8485     \__keyval_split_key:n {#1}
8486     \str_if_eq:nnTF {#2} { = }
8487     {
8488         \tl_put_right:Nx \l__keyval_parse_tl
8489         {
8490             \exp_not:c
8491             { \__keyval_key_no_value_elt_ \int_use:N \g__keyval_level_int :n }
8492             { \exp_not:o \l__keyval_key_tl }
8493         }
8494     }
8495     {
8496         \__keyval_split_key_value:wTF #2 \q_no_value \q_stop
8497         { \__keyval_split_value:w \q_nil #2 }
8498         { \__msg_kernel_error:nn { kernel } { misplaced-equals-sign } }
8499     }
8500 }
8501 \cs_new:Npn \__keyval_split_key_value:wTF #1 = #2#3 \q_stop
8502 { \tl_if_head_eq_meaning:nNTF {#3} \q_no_value }

```

(End definition for `__keyval_split_key_value:w`. This function is documented on page ??.)

```

\__keyval_split_key:n There are two possible cases here. The first case is that #1 is surrounded by braces,
\__keyval_split_key:w in which case the \use_none:nnn #1 \q_nil \q_nil will yield \q_nil. There, we can
remove the leading \q_nil, the braces and any spaces around the outside with \use_ii:nnn. On the other hand, if there are no braces then the second branch removes the
leading \q_nil and any surrounding spaces. (This code does not have to cover the case
with no key, as that's already taken out above.)

8503 \cs_new_protected:Npn \__keyval_split_key:n #1
8504 {
8505   \quark_if_nil:oTF { \use_none:nnn #1 \q_nil \q_nil }
8506   { \tl_set:Nx \l__keyval_key_tl { \exp_not:o { \use_ii:nnn #1 \q_nil } } }
8507   { \__keyval_split_key:w #1 \q_stop }
8508 }
8509 \cs_new_protected:Npn \__keyval_split_key:w \q_nil #1 \q_stop
8510 { \tl_set:Nx \l__keyval_key_tl { \tl_trim_spaces:n {#1} } }

```

(End definition for `__keyval_split_key:n`. This function is documented on page ??.)

`__keyval_split_value:w` Here the value has to be separated from the equals signs and the leading `\q_nil` added in to keep the brace levels. First the processing function can be added to the output list. If there is no value, setting `\l__keyval_value_tl` with three groups removed will leave nothing at all, and so an empty group can be added to the parsed list. On the other hand, if the value is entirely contained within a set of braces then `\l__keyval_value_tl` will contain `\q_nil` only. In that case, strip off the leading quark using `\use_ii:nnn`, which also deals with any spaces.

```

8511 \cs_new_protected:Npn \__keyval_split_value:w #1 = =
8512 {
8513   \tl_put_right:Nx \l__keyval_parse_tl
8514   {
8515     \exp_not:c
8516     { __keyval_key_value_elt_ \int_use:N \g__keyval_level_int :nn }
8517     { \exp_not:o \l__keyval_key_tl }
8518   }
8519   \tl_set:Nx \l__keyval_value_tl
8520   { \exp_not:o { \use_none:nnn #1 \q_nil \q_nil } }
8521   \tl_if_empty:NTF \l__keyval_value_tl
8522   { \tl_put_right:Nn \l__keyval_parse_tl { { } } }
8523   {
8524     \quark_if_nil:NTF \l__keyval_value_tl
8525     {
8526       \tl_put_right:Nx \l__keyval_parse_tl
8527       { { \exp_not:o { \use_ii:nnn #1 \q_nil } } }
8528     }
8529     { \__keyval_split_value_aux:w #1 \q_stop }
8530   }
8531 }

```

A similar idea to the key code: remove the spaces from each end and deal with one set of braces.

```

8532 \cs_new_protected:Npn \__keyval_split_value_aux:w \q_nil #1 \q_stop
8533 {
8534   \tl_set:Nx \l__keyval_value_tl { \tl_trim_spaces:n {#1} }
8535   \tl_put_right:Nx \l__keyval_parse_tl
8536     { { \exp_not:o \l__keyval_value_tl } }
8537 }

```

(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.

```

8538 \cs_new_protected:Npn \keyval_parse:NNn #1#2#3
8539 {
8540   \int_gincr:N \g__keyval_level_int
8541   \cs_gset_eq:cN
8542     { __keyval_key_no_value_elt_ \int_use:N \g__keyval_level_int :n } #1
8543   \cs_gset_eq:cN
8544     { __keyval_key_value_elt_ \int_use:N \g__keyval_level_int :nn } #2
8545   \__keyval_parse:n {#3}
8546   \int_gdecr:N \g__keyval_level_int
8547 }

```

(End definition for `\keyval_parse:NNn`. This function is documented on page 160.)

One message for the low level parsing system.

```

8548 \__msg_kernel_new:nnnn { kernel } { misplaced-equals-sign }
8549 { Misplaced~equals~sign~in~key~value~input~\msg_line_number: }
8550 {
8551   LaTeX-is-attempting-to-parse-some-key-value-input-but-found-
8552   two-equals-signs-not-separated-by-a-comma.
8553 }

```

18.2 Constants and variables

```

8554 <@@=keys>

```

`\c__keys_code_root_tl` The prefixes for the code and variables of the keys themselves.

```

\c__keys_vars_root_tl 8555 \tl_const:Nn \c__keys_code_root_tl { key-code->~ }
8556 \tl_const:Nn \c__keys_vars_root_tl { key-var->~ }

```

(End definition for `\c__keys_code_root_tl` and `\c__keys_vars_root_tl`. These variables are documented on page ??.)

`\c__keys_props_root_tl` The prefix for storing properties.

```

8557 \tl_const:Nn \c__keys_props_root_tl { key-prop->~ }

```

(End definition for `\c__keys_props_root_tl`. This variable is documented on page ??.)

`\c__keys_value_forbidden_tl` Two marker token lists.

```

\c__keys_value_required_tl 8558 \tl_const:Nn \c__keys_value_forbidden_tl { forbidden }
8559 \tl_const:Nn \c__keys_value_required_tl { required }

```

(End definition for `\c__keys_value_forbidden_tl` and `\c__keys_value_required_tl`. These variables are 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`

```

8560 \int_new:N \l_keys_choice_int
8561 \tl_new:N \l_keys_choice_tl

```

(End definition for \l_keys_choice_int and \l_keys_choice_tl. These variables are documented on page 156.)

`\l_keys_key_tl` The name of a key itself: needed when setting keys.

```

8562 \tl_new:N \l_keys_key_tl

```

(End definition for \l_keys_key_tl. This variable is documented on page 158.)

`\l__keys_module_tl` The module for an entire set of keys.

```

8563 \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.

```

8564 \bool_new:N \l_keys_no_value_bool

```

(End definition for \l_keys_no_value_bool. This variable is documented on page ??.)

`\l_keys_path_tl` The “path” of the current key is stored here: this is available to the programmer and so is public.

```

8565 \tl_new:N \l_keys_path_tl

```

(End definition for \l_keys_path_tl. This variable is documented on page 158.)

`\l_keys_property_tl` The “property” begin set for a key at definition time is stored here.

```

8566 \tl_new:N \l_keys_property_tl

```

(End definition for \l_keys_property_tl. This variable is documented on page ??.)

`\l_keys_unknown_clist` Used when setting only known keys to store those left over.

```

8567 \tl_new:N \l_keys_unknown_clist

```

(End definition for \l_keys_unknown_clist. This variable is documented on page ??.)

`\l_keys_value_tl` The value given for a key: may be empty if no value was given.

```

8568 \tl_new:N \l_keys_value_tl

```

(End definition for \l_keys_value_tl. This variable is documented on page 158.)

18.3 The key defining mechanism

`\keys_define:nn` The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting. The module is set removing any leading / (which is not needed here).

```

8569 \cs_new_protected:Npn \keys_define:nn
8570 { \__keys_define:onn \l__keys_module_tl }
8571 \cs_new_protected:Npn \__keys_define:nnn #1#2#3
8572 {
8573   \tl_set:Nx \l__keys_module_tl { \tl_to_str:n {#2} }
8574   \keyval_parse:NnN \__keys_define_elt:n \__keys_define_elt:nn {#3}
8575   \tl_set:Nn \l__keys_module_tl {#1}
8576 }
8577 \cs_generate_variant:Nn \__keys_define:nnn { o }

```

(End definition for `\keys_define:nn`. This function is documented on page 151.)

`__keys_define_elt:n` The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

```

8578 \cs_new_protected:Npn \__keys_define_elt:n #1
8579 {
8580   \bool_set_true:N \l__keys_no_value_bool
8581   \__keys_define_elt_aux:nn {#1} { }
8582 }
8583 \cs_new_protected:Npn \__keys_define_elt:nn #1#2
8584 {
8585   \bool_set_false:N \l__keys_no_value_bool
8586   \__keys_define_elt_aux:nn {#1} {#2}
8587 }
8588 \cs_new_protected:Npn \__keys_define_elt_aux:nn #1#2
8589 {
8590   \__keys_property_find:n {#1}
8591   \cs_if_exist:cTF { \c__keys_props_root_tl \l__keys_property_tl }
8592   { \__keys_define_key:n {#2} }
8593   {
8594     \str_if_eq_x:nnF { \l__keys_property_tl } { .abort: }
8595     {
8596       \__msg_kernel_error:nnxx { kernel } { property-unknown }
8597       { \l__keys_property_tl } { \l__keys_path_tl }
8598     }
8599   }
8600 }

```

(End definition for `__keys_define_elt:n`. This function is documented on page ??.)

`__keys_property_find:n` Searching for a property means finding the last . in the input, and storing the text before and after it. Everything is turned into strings, so there is no problem using an x-type expansion.

```

8601 \cs_new_protected:Npn \__keys_property_find:n #1
8602 {

```

```

8603     \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / }
8604     \tl_if_in:nnTF {#1} { . }
8605     { \__keys_property_find:w #1 \q_stop }
8606     {
8607         \__msg_kernel_error:nxx { kernel } { key-no-property } {#1}
8608         \tl_set:Nn \l__keys_property_tl { .abort: }
8609     }
8610 }
8611 \cs_new_protected:Npn \__keys_property_find:w #1 . #2 \q_stop
8612 {
8613     \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl \tl_to_str:n {#1} }
8614     \tl_if_in:nnTF {#2} { . }
8615     {
8616         \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl . }
8617         \__keys_property_find:w #2 \q_stop
8618     }
8619     { \tl_set:Nn \l__keys_property_tl { . #2 } }
8620 }

```

(End definition for `__keys_property_find:n`. This function is documented on page ??.)

`__keys_define_key:n`
`__keys_define_key:w`

Two possible cases. If there is a value for the key, then just use the function. If not, then a check to make sure there is no need for a value with the property. If there should be one then complain, otherwise execute it. There is no need to check for a : as if it is missing the earlier tests will have failed.

```

8621 \cs_new_protected:Npn \__keys_define_key:n #1
8622 {
8623     \bool_if:NTF \l__keys_no_value_bool
8624     {
8625         \exp_after:wN \__keys_define_key:w
8626         \l__keys_property_tl \q_stop
8627         { \use:c { \c__keys_props_root_tl \l__keys_property_tl } }
8628         {
8629             \__msg_kernel_error:nxxx { kernel }
8630             { property-requires-value } { \l__keys_property_tl }
8631             { \l_keys_path_tl }
8632         }
8633     }
8634     { \use:c { \c__keys_props_root_tl \l__keys_property_tl } {#1} }
8635 }
8636 \cs_new_protected:Npn \__keys_define_key:w #1 : #2 \q_stop
8637 { \tl_if_empty:nTF {#2} }

```

(End definition for `__keys_define_key:n`. This function is documented on page ??.)

18.4 Turning properties into actions

`__keys_bool_set:Nn`
`__keys_bool_set:cn`

Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```

8638 \cs_new_protected:Npn \__keys_bool_set:Nn #1#2

```

```

8639 {
8640   \bool_if_exist:NF #1 { \bool_new:N #1 }
8641   \__keys_choice_make:
8642   \__keys_cmd_set:nx { \l_keys_path_tl / true }
8643   { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
8644   \__keys_cmd_set:nx { \l_keys_path_tl / false }
8645   { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
8646   \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
8647   {
8648     \__msg_kernel_error:nxx { kernel } { boolean-values-only }
8649     { \l_keys_key_tl }
8650   }
8651   \__keys_default_set:n { true }
8652 }
8653 \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
8654 \cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
8655 {
8656   \bool_if_exist:NF #1 { \bool_new:N #1 }
8657   \__keys_choice_make:
8658   \__keys_cmd_set:nx { \l_keys_path_tl / true }
8659   { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
8660   \__keys_cmd_set:nx { \l_keys_path_tl / false }
8661   { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
8662   \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
8663   {
8664     \__msg_kernel_error:nxx { kernel } { boolean-values-only }
8665     { \l_keys_key_tl }
8666   }
8667   \__keys_default_set:n { true }
8668 }
8669 \cs_generate_variant:Nn \__keys_bool_set_inverse:Nn { c }
(End definition for \__keys_bool_set_inverse:Nn and \__keys_bool_set_inverse:cn.)

```

__keys_choice_make: To make a choice from a key, two steps: set the code, and set the unknown key.

```

8670 \cs_new_protected_nopar:Npn \__keys_choice_make:
8671 {
8672   \__keys_cmd_set:nn { \l_keys_path_tl }
8673   { \__keys_choice_find:n {##1} }
8674   \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
8675   {
8676     \__msg_kernel_error:nxxx { kernel } { key-choice-unknown }
8677     { \l_keys_path_tl } {##1}
8678   }
8679 }
(End definition for \__keys_choice_make:.)

```


`__keys_choices_make:nn` Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

```

8680 \cs_new_protected:Npn \__keys_choices_make:nn #1#2
8681 {
8682   \__keys_choice_make:
8683   \int_zero:N \l_keys_choice_int
8684   \clist_map_inline:nn {#1}
8685   {
8686     \int_incr:N \l_keys_choice_int
8687     \__keys_cmd_set:nx { \l_keys_path_tl / ##1 }
8688     {
8689       \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
8690       \int_set:Nn \exp_not:N \l_keys_choice_int
8691       { \int_use:N \l_keys_choice_int }
8692       \exp_not:n {#2}
8693     }
8694   }
8695 }

```

(End definition for __keys_choices_make:nn.)

`__keys_cmd_set:nn` Creating a new command means tidying up the properties and then making the internal
`__keys_cmd_set:nx` function which actually does the work.

```

8696 \cs_new_protected:Npn \__keys_cmd_set:nn #1#2
8697 {
8698   \__keys_cmd_set:n {#1}
8699   \cs_set:cpn { \c__keys_code_root_tl #1 } ##1 {#2}
8700 }
8701 \cs_generate_variant:Nn \__keys_cmd_set:nn { nx , Vn , Vo }
8702 \cs_new_protected:Npn \__keys_cmd_set:n #1
8703 {
8704   \tl_clear_new:c { \c__keys_vars_root_tl #1 .default }
8705   \tl_set:cn { \c__keys_vars_root_tl #1 .default } { \q_no_value }
8706   \tl_clear_new:c { \c__keys_vars_root_tl #1 .req }
8707 }

```

(End definition for __keys_cmd_set:nn and others. These functions are documented on page ??.)

`__keys_default_set:n` Setting a default value is easy.

```

8708 \cs_new_protected:Npn \__keys_default_set:n #1
8709 { \tl_set:cn { \c__keys_vars_root_tl \l_keys_path_tl .default } {#1} }

```

(End definition for __keys_default_set:n.)

`__keys_initialise:nn` 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.

```

8710 \cs_new_protected:Npn \__keys_initialise:n #1
8711 { \exp_after:wN \__keys_initialise:wn \l_keys_path_tl \q_stop {#1} }
8712 \cs_new_protected:Npn \__keys_initialise:wn #1 / #2 \q_stop #3
8713 { \keys_set:nn {#1} { #2 = {#3} } }

```

(End definition for `__keys_initialise:n`. This function is documented on page ??.)

`__keys_meta_make:n` To create a meta-key, simply set up to pass data through.

```

\__keys_meta_make:nn
8714 \cs_new_protected:Npn \__keys_meta_make:n #1
8715 {
8716   \__keys_cmd_set:Vo \l_keys_path_tl
8717   { \exp_after:wN \keys_set:nn \exp_after:wN { \l__keys_module_tl } {#1} }
8718 }
8719 \cs_new_protected:Npn \__keys_meta_make:nn #1#2
8720 { \__keys_cmd_set:Vn \l_keys_path_tl { \keys_set:nn {#1} {#2} } }
```

(End definition for `__keys_meta_make:n`. This function is documented on page ??.)

`__keys_multichoice_find:n` Choices where several values can be selected are very similar to normal exclusive choices.

`__keys_multichoice_make:` There is just a slight change in implementation to map across a comma-separated list.

`__keys_multichoices_make:nn` This then requires that the appropriate set up takes place elsewhere.

```

8721 \cs_new:Npn \__keys_multichoice_find:n #1
8722 { \clist_map_function:nN {#1} \__keys_choice_find:n }
8723 \cs_new_protected_nopar:Npn \__keys_multichoice_make:
8724 {
8725   \__keys_cmd_set:nn { \l_keys_path_tl }
8726   { \__keys_multichoice_find:n {##1} }
8727   \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
8728   {
8729     \__msg_kernel_error:nxxx { kernel } { key-choice-unknown }
8730     { \l_keys_path_tl } {##1}
8731   }
8732 }
8733 \cs_new_protected:Npn \__keys_multichoices_make:nn #1#2
8734 {
8735   \__keys_multichoice_make:
8736   \int_zero:N \l_keys_choice_int
8737   \clist_map_inline:nn {#1}
8738   {
8739     \int_incr:N \l_keys_choice_int
8740     \__keys_cmd_set:nx { \l_keys_path_tl / ##1 }
8741     {
8742       \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
8743       \int_set:Nn \exp_not:N \l_keys_choice_int
8744       { \int_use:N \l_keys_choice_int }
8745       \exp_not:n {#2}
8746     }
8747   }
8748 }
```

(End definition for `__keys_multichoice_find:n`. This function is documented on page ??.)

`__keys_value_requirement:n` Values can be required or forbidden by having the appropriate marker set.

```

8749 \cs_new_protected:Npn \__keys_value_requirement:n #1
8750 {
8751   \tl_set_eq:cc
```

```

8752     { \c__keys_vars_root_tl \l_keys_path_tl .req }
8753     { c__keys_value_ #1 _tl }
8754   }

```

(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.

__keys_variable_set:cnmN

```

8755 \cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
8756 {
8757   \use:c { #2_if_exist:NF } #1 { \use:c { #2_new:N } #1 }
8758   \__keys_cmd_set:nx { \l_keys_path_tl }
8759   { \exp_not:c { #2 _ #3 set:N #4 } \exp_not:N #1 \exp_not:n { {##1} } }
8760 }
8761 \cs_generate_variant:Nn \__keys_variable_set:NnnN { c }

```

(End definition for __keys_variable_set:NnnN and __keys_variable_set:cnmN.)

18.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

Importantly, while key properties have “normal” argument specs, the underlying code always supplies one braced argument to these. As such, argument expansion is handled by hand rather than using the standard tools. This shows up particularly for the two-argument properties, where things would otherwise go badly wrong.

.bool_set:N One function for this.

```

8762 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:N } #1
8763 { \__keys_bool_set:Nn #1 { } }
8764 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:c } #1
8765 { \__keys_bool_set:cn {#1} { } }
8766 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:N } #1
8767 { \__keys_bool_set:Nn #1 { g } }
8768 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:c } #1
8769 { \__keys_bool_set:cn {#1} { g } }

```

(End definition for .bool_set:N and .bool_set:c. These functions are documented on page 152.)

.bool_set_inverse:N One function for this.

```

8770 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:N } #1
8771 { \__keys_bool_set_inverse:Nn #1 { } }
8772 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:c } #1
8773 { \__keys_bool_set_inverse:cn {#1} { } }
8774 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:N } #1
8775 { \__keys_bool_set_inverse:Nn #1 { g } }
8776 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:c } #1
8777 { \__keys_bool_set_inverse:cn {#1} { g } }

```

(End definition for .bool_set_inverse:N and .bool_set_inverse:c. These functions are documented on page 152.)

.choice: Making a choice is handled internally, as it is also needed by **.generate_choices:n**.

```

8778 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .choice: }
8779 { \__keys_choice_make: }

```

(End definition for **.choice:**. This function is documented on page 152.)

.choices:nn For auto-generation of a series of mutually-exclusive choices. Here, #1 will consist of two separate arguments, hence the slightly odd-looking implementation.

.choices:Vn

.choices:on

.choices:xn

```

8780 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:nn } #1
8781 { \__keys_choices_make:nn #1 }
8782 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:Vn } #1
8783 { \exp_args:NV \__keys_choices_make:nn #1 }
8784 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:on } #1
8785 { \exp_args:No \__keys_choices_make:nn #1 }
8786 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:xn } #1
8787 { \exp_args:Nx \__keys_choices_make:nn #1 }

```

(End definition for **.choices:nn** and others. These functions are documented on page 152.)

.code:n Creating code is simply a case of passing through to the underlying set function.

```

8788 \cs_new_protected:cpn { \c__keys_props_root_tl .code:n } #1
8789 { \__keys_cmd_set:nn { \l_keys_path_tl } {#1} }

```

(End definition for **.code:n**. This function is documented on page 152.)

.clist_set:N

.clist_set:c

.clist_gset:N

.clist_gset:c

```

8790 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:N } #1
8791 { \__keys_variable_set:NnnN #1 { clist } { } n }
8792 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:c } #1
8793 { \__keys_variable_set:cnnN {#1} { clist } { } n }
8794 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:N } #1
8795 { \__keys_variable_set:NnnN #1 { clist } { g } n }
8796 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:c } #1
8797 { \__keys_variable_set:cnnN {#1} { clist } { g } n }

```

(End definition for **.clist_set:N** and **.clist_set:c**. These functions are documented on page 152.)

.default:n Expansion is left to the internal functions.

.default:V

.default:o

.default:x

```

8798 \cs_new_protected:cpn { \c__keys_props_root_tl .default:n } #1
8799 { \__keys_default_set:n {#1} }
8800 \cs_new_protected:cpn { \c__keys_props_root_tl .default:V } #1
8801 { \exp_args:NV \__keys_default_set:n #1 }
8802 \cs_new_protected:cpn { \c__keys_props_root_tl .default:o } #1
8803 { \exp_args:No \__keys_default_set:n {#1} }
8804 \cs_new_protected:cpn { \c__keys_props_root_tl .default:x } #1
8805 { \exp_args:Nx \__keys_default_set:n {#1} }

```

(End definition for **.default:n** and others. These functions are documented on page 153.)

```
.dim_set:N Setting a variable is very easy: just pass the data along.
.dim_set:c 8806 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:N } #1
.dim_gset:N 8807 { \__keys_variable_set:NnnN #1 { dim } { } n }
.dim_gset:c 8808 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:c } #1
8809 { \__keys_variable_set:cnnN {#1} { dim } { } n }
8810 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:N } #1
8811 { \__keys_variable_set:NnnN #1 { dim } { g } n }
8812 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
8813 { \__keys_variable_set:cnnN {#1} { dim } { g } n }
(End definition for .dim_set:N and .dim_set:c. These functions are documented on page 153.)
```

```
.fp_set:N Setting a variable is very easy: just pass the data along.
.fp_set:c 8814 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:N } #1
.fp_gset:N 8815 { \__keys_variable_set:NnnN #1 { fp } { } n }
.fp_gset:c 8816 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:c } #1
8817 { \__keys_variable_set:cnnN {#1} { fp } { } n }
8818 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:N } #1
8819 { \__keys_variable_set:NnnN #1 { fp } { g } n }
8820 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
8821 { \__keys_variable_set:cnnN {#1} { fp } { g } n }
(End definition for .fp_set:N and .fp_set:c. These functions are documented on page 153.)
```

```
.initial:n The standard hand-off approach.
.initial:N 8822 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:n } #1
.initial:o 8823 { \__keys_initialise:n {#1} }
.initial:x 8824 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:N } #1
8825 { \exp_args:NV \__keys_initialise:n #1 }
8826 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:o } #1
8827 { \exp_args:No \__keys_initialise:n {#1} }
8828 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
8829 { \exp_args:Nx \__keys_initialise:n {#1} }
(End definition for .initial:n and others. These functions are documented on page 153.)
```

```
.int_set:N Setting a variable is very easy: just pass the data along.
.int_set:c 8830 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:N } #1
.int_gset:N 8831 { \__keys_variable_set:NnnN #1 { int } { } n }
.int_gset:c 8832 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:c } #1
8833 { \__keys_variable_set:cnnN {#1} { int } { } n }
8834 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:N } #1
8835 { \__keys_variable_set:NnnN #1 { int } { g } n }
8836 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
8837 { \__keys_variable_set:cnnN {#1} { int } { g } n }
(End definition for .int_set:N and .int_set:c. These functions are documented on page 153.)
```

```
.meta:n Making a meta is handled internally.
8838 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:n } #1
8839 { \__keys_meta_make:n {#1} }
(End definition for .meta:n. This function is documented on page 153.)
```

.meta:nn Meta with path: potentially lots of variants, but for the moment no so many defined.

```
8840 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:nn } #1
8841 { \__keys_meta_make:nn #1 }
```

(End definition for .meta:nn. This function is documented on page 154.)

.multichoice: The same idea as .choice: and .choices:nn, but where more than one choice is allowed.

```
8842 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .multichoice: }
8843 { \__keys_multichoice_make: }
8844 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:nn } #1
8845 { \__keys_multichoices_make:nn #1 }
8846 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:Vn } #1
8847 { \exp_args:NV \__keys_multichoices_make:nn #1 }
8848 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:on } #1
8849 { \exp_args:No \__keys_multichoices_make:nn #1 }
8850 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:xn } #1
8851 { \exp_args:Nx \__keys_multichoices_make:nn #1 }
```

(End definition for .multichoice:. This function is documented on page 154.)

.skip_set:N Setting a variable is very easy: just pass the data along.

```
8852 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:N } #1
8853 { \__keys_variable_set:NnnN #1 { skip } { } n }
8854 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:c } #1
8855 { \__keys_variable_set:cnnN {#1} { skip } { } n }
8856 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:N } #1
8857 { \__keys_variable_set:NnnN #1 { skip } { g } n }
8858 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:c } #1
8859 { \__keys_variable_set:cnnN {#1} { skip } { g } n }
```

(End definition for .skip_set:N and .skip_set:c. These functions are documented on page 154.)

.tl_set:N Setting a variable is very easy: just pass the data along.

```
8860 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:N } #1
8861 { \__keys_variable_set:NnnN #1 { tl } { } n }
8862 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:c } #1
8863 { \__keys_variable_set:cnnN {#1} { tl } { } n }
8864 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:N } #1
8865 { \__keys_variable_set:NnnN #1 { tl } { } x }
8866 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:c } #1
8867 { \__keys_variable_set:cnnN {#1} { tl } { } x }
8868 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:N } #1
8869 { \__keys_variable_set:NnnN #1 { tl } { g } n }
8870 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:c } #1
8871 { \__keys_variable_set:cnnN {#1} { tl } { g } n }
8872 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:N } #1
8873 { \__keys_variable_set:NnnN #1 { tl } { g } x }
8874 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:c } #1
8875 { \__keys_variable_set:cnnN {#1} { tl } { g } x }
```

(End definition for .tl_set:N and .tl_set:c. These functions are documented on page 154.)

`.value_forbidden:` These are very similar, so both call the same function.

```
.value_required:
8876 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .value_forbidden: }
8877 { \__keys_value_requirement:n { forbidden } }
8878 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .value_required: }
8879 { \__keys_value_requirement:n { required } }
```

(End definition for `.value_forbidden:`. This function is documented on page 154.)

18.6 Setting keys

`\keys_set:nn` A simple wrapper again.

```
\keys_set:nV
\keys_set:nv
\keys_set:no
\__keys_set:nnn
\__keys_set:onn
8880 \cs_new_protected:Npn \keys_set:nn
8881 { \__keys_set:onn { \l__keys_module_tl } }
8882 \cs_new_protected:Npn \__keys_set:nnn #1#2#3
8883 {
8884   \tl_set:Nx \l__keys_module_tl { \tl_to_str:n {#2} }
8885   \keyval_parse:NNn \__keys_set_elt:n \__keys_set_elt:nn {#3}
8886   \tl_set:Nn \l__keys_module_tl {#1}
8887 }
8888 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
8889 \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`

```
\keys_set_known:nVN
\keys_set_known:nvN
\keys_set_known:noN
\__keys_set_known:nnnN
\__keys_set_known:onnN
8890 \cs_new_protected:Npn \keys_set_known:nnN
8891 { \__keys_set_known:onnN { \l__keys_module_tl } }
8892 \cs_new_protected:Npn \__keys_set_known:nnnN #1#2#3#4
8893 {
8894   \tl_set:Nx \l__keys_module_tl { \tl_to_str:n {#2} }
8895   \clist_clear:N \l__keys_unknown_clist
8896   \cs_set_eq:NN \__keys_execute_unknown: \__keys_execute_unknown_alt:
8897   \keyval_parse:NNn \__keys_set_elt:n \__keys_set_elt:nn {#3}
8898   \cs_set_eq:NN \__keys_execute_unknown: \__keys_execute_unknown_std:
8899   \tl_set:Nn \l__keys_module_tl {#1}
8900   \cs_set_eq:NN #4 \l__keys_unknown_clist
8901 }
8902 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
8903 \cs_generate_variant:Nn \__keys_set_known:nnnN { o }
```

(End definition for `\keys_set_known:nnN` and others. These functions are documented on page ??.)

`__keys_set_elt:n` A shared system once again. First, set the current path and add a default if needed.

`__keys_set_elt:nn` There are then checks to see if the a value is required or forbidden. If everything passes,

`__keys_set_elt_aux:nn` move on to execute the code.

```
8904 \cs_new_protected:Npn \__keys_set_elt:n #1
8905 {
8906   \bool_set_true:N \l__keys_no_value_bool
8907   \__keys_set_elt_aux:nn {#1} { }
8908 }
8909 \cs_new_protected:Npn \__keys_set_elt:nn #1#2
```

```

8910 {
8911     \bool_set_false:N \l__keys_no_value_bool
8912     \__keys_set_elt_aux:nn {#1} {#2}
8913 }
8914 \cs_new_protected:Npn \__keys_set_elt_aux:nn #1#2
8915 {
8916     \tl_set:Nx \l_keys_key_tl { \tl_to_str:n {#1} }
8917     \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / \l_keys_key_tl }
8918     \__keys_value_or_default:n {#2}
8919     \bool_if:nTF
8920     {
8921         \__keys_if_value_p:n { required } &&
8922         \l__keys_no_value_bool
8923     }
8924     {
8925         \__msg_kernel_error:nnx { kernel } { value-required }
8926         { \l_keys_path_tl }
8927     }
8928     {
8929         \bool_if:nTF
8930         {
8931             \__keys_if_value_p:n { forbidden } &&
8932             ! \l__keys_no_value_bool
8933         }
8934         {
8935             \__msg_kernel_error:nnxx { kernel } { value-forbidden }
8936             { \l_keys_path_tl } { \l_keys_value_tl }
8937         }
8938         { \__keys_execute: }
8939     }
8940 }

```

(End definition for __keys_set_elt:n and __keys_set_elt:nn. These functions are documented on page ??.)

__keys_value_or_default:n If a value is given, return it as #1, otherwise send a default if available.

```

8941 \cs_new_protected:Npn \__keys_value_or_default:n #1
8942 {
8943     \tl_set:Nn \l_keys_value_tl {#1}
8944     \bool_if:NT \l__keys_no_value_bool
8945     {
8946         \quark_if_no_value:cF { \c__keys_vars_root_tl \l_keys_path_tl .default }
8947         {
8948             \cs_if_exist:cT { \c__keys_vars_root_tl \l_keys_path_tl .default }
8949             {
8950                 \tl_set_eq:Nc \l_keys_value_tl
8951                 { \c__keys_vars_root_tl \l_keys_path_tl .default }
8952             }
8953         }
8954     }
8955 }

```


(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.

```

8956 \prg_new_conditional:Npnn \_keys_if_value:n #1 { p }
8957 {
8958   \tl_if_eq:ccTF { c\_keys_value_ #1 _tl }
8959   { \c_keys_vars_root_tl \l_keys_path_tl .req }
8960   { \prg_return_true: }
8961   { \prg_return_false: }
8962 }

```

(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.

```

\_keys_execute_unknown:
\_keys_execute_unknown_std:
\_keys_execute_unknown_alt:
\_keys_execute:nn
8963 \cs_new_protected_nopar:Npn \_keys_execute:
8964 { \_keys_execute:nn { \l_keys_path_tl } { \_keys_execute_unknown: } }
8965 \cs_new_protected_nopar:Npn \_keys_execute_unknown:
8966 {
8967   \_keys_execute:nn { \l_keys_module_tl / unknown }
8968   {
8969     \_msg_kernel_error:nxxx { kernel } { key-unknown }
8970     { \l_keys_path_tl } { \l_keys_module_tl }
8971   }
8972 }
8973 \cs_new_eq:NN \_keys_execute_unknown_std: \_keys_execute_unknown:
8974 \cs_new_protected_nopar:Npn \_keys_execute_unknown_alt:
8975 {
8976   \clist_put_right:Nx \l_keys_unknown_clist
8977   {
8978     \exp_not:o \l_keys_key_tl
8979     \bool_if:NF \l_keys_no_value_bool
8980     { = { \exp_not:o \l_keys_value_tl } }
8981   }
8982 }
8983 \cs_new:Npn \_keys_execute:nn #1#2
8984 {
8985   \cs_if_exist:cTF { \c_keys_code_root_tl #1 }
8986   {
8987     \exp_args:Nc \exp_args:No { \c_keys_code_root_tl #1 }
8988     \l_keys_value_tl
8989   }
8990   {#2}
8991 }

```

(End definition for `_keys_execute:`. This function is documented on page ??.)

`_keys_choice_find:n` Executing a choice has two parts. First, try the choice given, then if that fails call the `unknown` key. That will exist, as it is created when a choice is first made. So there is no need for any escape code.

```

8992 \cs_new:Npn \__keys_choice_find:n #1
8993 {
8994     \__keys_execute:nn { \l_keys_path_tl / \tl_to_str:n {#1} }
8995     { \__keys_execute:nn { \l_keys_path_tl / unknown } { } }
8996 }
(End definition for \__keys_choice_find:n.)

```

18.7 Utilities

\keys_if_exist_p:nn A utility for others to see if a key exists.

```

\keys_if_exist:nnTF
8997 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
8998 {
8999     \cs_if_exist:cTF { \c__keys_code_root_tl #1 / #2 }
9000     { \prg_return_true: }
9001     { \prg_return_false: }
9002 }
(End definition for \keys_if_exist:nn. These functions are documented on page 159.)

```

Just an alternative view on `\keys_if_exist:nn(TF)`.

```

\keys_if_choice_exist_p:nnn
\keys_if_choice_exist:nnnTF
9003 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3 { p , T , F , TF }
9004 {
9005     \cs_if_exist:cTF { \c__keys_code_root_tl #1 / #2 / #3 }
9006     { \prg_return_true: }
9007     { \prg_return_false: }
9008 }
(End definition for \keys_if_choice_exist:nnn. These functions are documented on page 159.)

```

\keys_show:nn Showing a key is just a question of using the correct name.

```

9009 \cs_new_protected:Npn \keys_show:nn #1#2
9010 { \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 159.)

```

18.8 Messages

For when there is a need to complain.

```

9011 \__msg_kernel_new:nnnn { kernel } { boolean-values-only }
9012 { Key~'#1'~accepts~boolean~values~only. }
9013 { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
9014 \__msg_kernel_new:nnnn { kernel } { choice-unknown }
9015 { Choice~'#2'~unknown~for~key~'#1'. }
9016 {
9017     The~key~'#1'~takes~a~limited~number~of~values.\\
9018     The~input~given,~'#2',~is~not~on~the~list~accepted.
9019 }
9020 \__msg_kernel_new:nnnn { kernel } { key-choice-unknown }
9021 { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
9022 {
9023     The~key~'#1'~only~accepts~predefined~values,~and~'#2'~is~not~one~of~these.

```

```

9024 }
9025 \_msg_kernel_new:nnnn { kernel } { key-no-property }
9026 { No~property~given~in~definition~of~key~'#1'. }
9027 {
9028   \c_msg_coding_error_text_tl
9029   Inside~\keys_define:nn  each~key~name~
9030   needs~a~property:  \ \ \
9031   \iow_indent:n { #1 .<property> } \ \ \
9032   LaTeX~did~not~find~a~'. ' ~to~indicate~the~start~of~a~property.
9033 }
9034 \_msg_kernel_new:nnnn { kernel } { key-unknown }
9035 { The~key~'#1'~is~unknown~and~is~being~ignored. }
9036 {
9037   The~module~'#2'~does~not~have~a~key~called~'#1'. \ \
9038   Check~that~you~have~spelled~the~key~name~correctly.
9039 }
9040 \_msg_kernel_new:nnnn { kernel } { property-requires-value }
9041 { The~property~'#1'~requires~a~value. }
9042 {
9043   \c_msg_coding_error_text_tl
9044   LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'. \ \
9045   No~value~was~given~for~the~property,~and~one~is~required.
9046 }
9047 \_msg_kernel_new:nnnn { kernel } { property-unknown }
9048 { The~key~property~'#1'~is~unknown. }
9049 {
9050   \c_msg_coding_error_text_tl
9051   LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
9052   this~property~is~not~defined.
9053 }
9054 \_msg_kernel_new:nnnn { kernel } { value-forbidden }
9055 { The~key~'#1'~does~not~taken~a~value. }
9056 {
9057   The~key~'#1'~should~be~given~without~a~value. \ \
9058   LaTeX~will~ignore~the~given~value~'#2'.
9059 }
9060 \_msg_kernel_new:nnnn { kernel } { value-required }
9061 { The~key~'#1'~requires~a~value. }
9062 {
9063   The~key~'#1'~must~have~a~value. \ \
9064   No~value~was~present:~the~key~will~be~ignored.
9065 }

```

18.9 Deprecated functions

Deprecated on 2013-07-09.

`_keys_choice_code_store:n` The code for making multiple choices is stored in a token list.
`_keys_choice_code_store:x` 9066 \cs_new_protected:Npn _keys_choice_code_store:n #1

```

9067 {
9068   \cs_if_exist:cF
9069   { \c__keys_vars_root_tl \l_keys_path_tl .choice~code }
9070   {
9071     \tl_new:c
9072     { \c__keys_vars_root_tl \l_keys_path_tl .choice~code }
9073   }
9074   \tl_set:cn { \c__keys_vars_root_tl \l_keys_path_tl .choice~code }
9075   {#1}
9076 }
9077 \cs_generate_variant:Nn \__keys_choice_code_store:n { x }
(End definition for \__keys_choice_code_store:n and \__keys_choice_code_store:x.)

```

.choice_code:n Storing the code for choices

```

.choice_code:x
9078 \cs_new_protected:cpn { \c__keys_props_root_tl .choice_code:n } #1
9079 { \__keys_choice_code_store:n {#1} }
9080 \cs_new_protected:cpn { \c__keys_props_root_tl .choice_code:x } #1
9081 { \__keys_choice_code_store:x {#1} }

```

(End definition for .choice_code:n and .choice_code:x. These functions are documented on page ??.)

__keys_choices_generate:n Creating multiple-choices means setting up the “indicator” code, then applying whatever the user wanted.

__keys_choices_generate_aux:n

```

9082 \cs_new_protected:Npn \__keys_choices_generate:n #1
9083 {
9084   \cs_if_exist:cTF
9085   { \c__keys_vars_root_tl \l_keys_path_tl .choice~code }
9086   {
9087     \__keys_choice_make:
9088     \int_zero:N \l_keys_choice_int
9089     \clist_map_function:nN {#1} \__keys_choices_generate_aux:n
9090   }
9091   {
9092     \__msg_kernel_error:nxx { kernel }
9093     { generate-choices-before-code } { \l_keys_path_tl }
9094   }
9095 }
9096 \cs_new_protected:Npn \__keys_choices_generate_aux:n #1
9097 {
9098   \int_incr:N \l_keys_choice_int
9099   \__keys_cmd_set:nx { \l_keys_path_tl / #1 }
9100   {
9101     \tl_set:Nn \exp_not:N \l_keys_choice_tl {#1}
9102     \int_set:Nn \exp_not:N \l_keys_choice_int
9103     { \int_use:N \l_keys_choice_int }
9104     \exp_not:v
9105     { \c__keys_vars_root_tl \l_keys_path_tl .choice~code }
9106   }
9107 }
9108 \__msg_kernel_new:nnnn { kernel } { generate-choices-before-code }

```

```

9109 { No~code~available~to~generate~choices~for~key~'#1'. }
9110 {
9111   \c_msg_coding_error_text_tl
9112   Before~using~.generate_choices:n~the~code~should~be~defined~
9113   with~'.choice_code:n'~or~'.choice_code:x'.
9114 }

```

(End definition for __keys_choices_generate:n. This function is documented on page ??.)

.generate_choices:n Making choices is easy.

```

9115 \cs_new_protected:cpn { \c__keys_props_root_tl .generate_choices:n } #1
9116 { \__keys_choices_generate:n {#1} }

```

(End definition for .generate_choices:n. This function is documented on page ??.)

Deprecated on 2011-05-27, for removal by 2011-08-31.

```

\KV_process_space_removal_sanitize:NNn
\KV_process_space_removal_no_sanitize:NNn
\KV_process_no_space_removal_no_sanitize:NNn

```

There is just one function for this now.

```

9117 <*deprecated>
9118 \cs_new_eq:NN \KV_process_space_removal_sanitize:NNn \keyval_parse:NNn
9119 \cs_new_eq:NN \KV_process_space_removal_no_sanitize:NNn \keyval_parse:NNn
9120 \cs_new_eq:NN \KV_process_no_space_removal_no_sanitize:NNn \keyval_parse:NNn
9121 </deprecated>

```

(End definition for \KV_process_space_removal_sanitize:NNn. This function is documented on page ??.)

Internal material for removal by 2012-12-31.

```

9122 <*deprecated>
9123 \cs_new_eq:NN \c_keys_code_root_tl \c__keys_code_root_tl
9124 </deprecated>
9125 </initex | package>

```

19 l3file implementation

The following test files are used for this code: m3file001.

```

9126 <*initex | package>
9127 <@@=file>
9128 <*package>
9129 \ProvidesExplPackage
9130   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
9131   \__expl_package_check:
9132 </package>

```

19.1 File operations

`\g_file_current_name_tl` The name of the current file should be available at all times. For the format the file name needs to be picked up at the start of the file. In package mode the current file name is collected from L^AT_EX 2_ε.

```

9133 \tl_new:N \g_file_current_name_tl
9134 <*initex>
9135 \tex_everyjob:D \exp_after:wN
9136 {
9137   \tex_the:D \tex_everyjob:D
9138   \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
9139 }
9140 </initex>
9141 <*package>
9142 \tl_gset_eq:NN \g_file_current_name_tl \@currname
9143 </package>

```

(End definition for `\g_file_current_name_tl`. This variable is documented on page 161.)

`\g__file_stack_seq` The input list of files is stored as a sequence stack.

```

9144 \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`.

```

9145 \seq_new:N \g__file_record_seq
9146 <*initex>
9147 \tex_everyjob:D \exp_after:wN
9148 {
9149   \tex_the:D \tex_everyjob:D
9150   \seq_gput_right:NV \g__file_record_seq \g_file_current_name_tl
9151 }
9152 </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.

```

9153 \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.

```

9154 \tl_new:N \l__file_internal_name_tl

```

(End definition for `\l__file_internal_name_tl`. This variable is documented on page 166.)

`\l__file_search_path_seq` The current search path.

```

9155 \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.

```

9156 <*package>
9157 \seq_new:N \l_file_saved_search_path_seq
9158 </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.

```

9159 <*package>
9160 \seq_new:N \l_file_internal_seq
9161 </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.

```

9162 \cs_new_protected:Npn \__file_name_sanitiz:nn #1#2
9163 {
9164   \group_begin:
9165     \seq_map_inline:Nn \l_char_active_seq
9166       { \cs_set_nopar:Npx ##1 { \token_to_str:N ##1 } }
9167     \tl_set:Nx \l_file_internal_name_tl {#1}
9168     \tl_set:Nx \l_file_internal_name_tl
9169       { \tl_to_str:N \l_file_internal_name_tl }
9170     \tl_if_in:NnT \l_file_internal_name_tl { ~ }
9171     {
9172       \__msg_kernel_error:nxx { kernel } { space-in-file-name }
9173       { \l_file_internal_name_tl }
9174       \tl_remove_all:Nn \l_file_internal_name_tl { ~ }
9175     }
9176     \use:x
9177     {
9178       \group_end:
9179       \exp_not:n {#2} { \l_file_internal_name_tl }
9180     }
9181   }
(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 T_EX 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.

```

9182 \cs_new_protected:Npn \file_add_path:nN #1
9183   { \__file_name_sanitiz:nn {#1} { \__file_add_path:nN } }
9184 \cs_new_protected:Npn \__file_add_path:nN #1#2
9185   {
9186     \__ior_open:Nn \g__file_internal_ior {#1}
9187     \ior_if_eof:NTF \g__file_internal_ior
9188       { \__file_add_path_search:nN {#1} #2 }

```

```

9189     { \tl_set:Nn #2 {#1} }
9190     \ior_close:N \g__file_internal_ior
9191   }
9192   \cs_new_protected:Npn \__file_add_path_search:nN #1#2
9193   {
9194     \tl_set:Nn #2 { \q_no_value }
9195     <*package>
9196     \cs_if_exist:NT \input@path
9197     {
9198       \seq_set_eq:NN \l__file_saved_search_path_seq \l__file_search_path_seq
9199       \seq_set_split:NnV \l__file_internal_seq { , } \input@path
9200       \seq_concat:NNN \l__file_search_path_seq
9201       \l__file_search_path_seq \l__file_internal_seq
9202     }
9203     </package>
9204     \seq_map_inline:Nn \l__file_search_path_seq
9205     {
9206       \__ior_open:Nn \g__file_internal_ior { ##1 #1 }
9207       \ior_if_eof:NF \g__file_internal_ior
9208       {
9209         \tl_set:Nx #2 { ##1 #1 }
9210         \seq_map_break:
9211       }
9212     }
9213     <*package>
9214     \cs_if_exist:NT \input@path
9215     { \seq_set_eq:NN \l__file_search_path_seq \l__file_saved_search_path_seq }
9216     </package>
9217   }

```

(End definition for \file_add_path:nN. This function is documented on page 161.)

\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.

```

9218   \prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
9219   {
9220     \file_add_path:nN {#1} \l__file_internal_name_tl
9221     \quark_if_no_value:NTF \l__file_internal_name_tl
9222     { \prg_return_false: }
9223     { \prg_return_true: }
9224   }

```

(End definition for \file_if_exist:nTF. This function is documented on page 161.)

\file_input:n Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the \g__file_stack_seq, and add it to the file list, either \g__file_record_seq, or \@filelist in package mode.

```

\__file_input:n\__file_input:V
\__file_input_aux:n
\__file_input_aux:o
9225   \cs_new_protected:Npn \file_input:n #1
9226   {
9227     \file_add_path:nN {#1} \l__file_internal_name_tl

```



```

9228 \quark_if_no_value:NTF \l__file_internal_name_tl
9229 {
9230   \__file_name_sanitiz:nn {#1}
9231   { \__msg_kernel_error:nxx { kernel } { file-not-found } }
9232 }
9233 { \__file_input:V \l__file_internal_name_tl }
9234 }
9235 \cs_new_protected:Npn \__file_input:n #1
9236 {
9237   \tl_if_in:nnTF {#1} { . }
9238   { \__file_input_aux:n {#1} }
9239   { \__file_input_aux:o { \tl_to_str:n { #1 . tex } } }
9240 }
9241 \cs_generate_variant:Nn \__file_input:n { V }
9242 \cs_new_protected:Npn \__file_input_aux:n #1
9243 {
9244   <*initex>
9245   \seq_gput_right:Nn \g__file_record_seq {#1}
9246   </initex>
9247   <*package>
9248   \clist_if_exist:NTF \@filelist
9249   { \@addtofilelist {#1} }
9250   { \seq_gput_right:Nn \g__file_record_seq {#1} }
9251   </package>
9252   \seq_gpush:No \g__file_stack_seq \g_file_current_name_tl
9253   \tl_gset:Nn \g_file_current_name_tl {#1}
9254   \tex_input:D #1 \c_space_tl
9255   \seq_gpop:NN \g__file_stack_seq \l__file_internal_tl
9256   \tl_gset_eq:NN \g_file_current_name_tl \l__file_internal_tl
9257 }
9258 \cs_generate_variant:Nn \__file_input_aux:n { o }

```

(End definition for \file_input:n. This function is documented on page 161.)

\file_path_include:n Wrapper functions to manage the search path.

```

9259 \cs_new_protected:Npn \file_path_include:n #1
9260 { \__file_name_sanitiz:nn {#1} { \__file_path_include:n } }
9261 \cs_new_protected:Npn \__file_path_include:n #1
9262 {
9263   \seq_if_in:NnF \l__file_search_path_seq {#1}
9264   { \seq_put_right:Nn \l__file_search_path_seq {#1} }
9265 }
9266 \cs_new_protected:Npn \file_path_remove:n #1
9267 {
9268   \__file_name_sanitiz:nn {#1}
9269   { \seq_remove_all:Nn \l__file_search_path_seq }
9270 }

```

(End definition for \file_path_include:n. This function is documented on page 162.)

\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.

```

9271 \cs_new_protected_nopar:Npn \file_list:
9272 {
9273   \seq_set_eq:NN \l__file_internal_seq \g__file_record_seq
9274   \cs_new_protected_nopar:Npn \@filelist
9275   {
9276     \clist_map_inline:Nn \@filelist
9277     {
9278       \seq_put_right:No \l__file_internal_seq
9279       { \tl_to_str:n {##1} }
9280     }
9281   }
9282 }
9283 \package
9284 \seq_remove_duplicates:N \l__file_internal_seq
9285 \iow_log:n { *-File~List~* }
9286 \seq_map_inline:Nn \l__file_internal_seq { \iow_log:n {##1} }
9287 \iow_log:n { ***** }
9288 }

```

(End definition for `\file_list`:. This function is documented on page 162.)

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`.

```

9289 \package
9290 \AtBeginDocument
9291 {
9292   \clist_map_inline:Nn \@filelist
9293   { \seq_gput_right:No \g__file_record_seq { \tl_to_str:n {##1} } }
9294 }
9295 \package

```

19.2 Input operations

```

9296 \@@=ior

```

19.2.1 Variables and constants

`\c_term_ior` Reading from the terminal (with a prompt) is done using a positive but non-existent stream number. Unlike writing, there is no concept of reading from the log.

```

9297 \cs_new_eq:NN \c_term_ior \c_sixteen

```

(End definition for `\c_term_ior`. This variable is documented on page 166.)

`\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!

```

9298 \seq_new:N \g__ior_streams_seq
9299 \package
9300 \seq_gset_split:Nnn \g__ior_streams_seq { , }

```

```

9301 { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 }
9302 </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.

```

9303 \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.

```

9304 \prop_new:N \g__ior_streams_prop
9305 <*package>
9306 \prop_gput:Nnn \g__ior_streams_prop { 0 } { LaTeX2e~reserved }
9307 </package>

```

(End definition for `\g__ior_streams_prop`. This variable is documented on page ??.)

19.2.2 Stream management

`\ior_new:N` Reserving a new stream is done by defining the name as equal to using the terminal.

```

\ior_new:c
9308 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c_term_ior }
9309 \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`

```

9310 \cs_new_protected:Npn \ior_open:Nn #1#2
9311 { \__file_name_sanitize:nn {#2} { \__ior_open_aux:Nn #1 } }
9312 \cs_generate_variant:Nn \ior_open:Nn { c }
9313 \cs_new_protected:Npn \__ior_open_aux:Nn #1#2
9314 {
9315   \file_add_path:nN {#2} \l__file_internal_name_tl
9316   \quark_if_no_value:NTF \l__file_internal_name_tl
9317     { \__msg_kernel_error:nmx { kernel } { file-not-found } {#2} }
9318     { \__ior_open:No #1 \l__file_internal_name_tl }
9319 }

```

(End definition for `\ior_open:Nn` and `\ior_open:cn`. These functions are documented on page ??.)

`\ior_open:NnTF` Much the same idea for opening a read with a conditional, except the auxiliary function does not issue an error if the file is not found.

`\ior_open:cnTF`

`__ior_open_aux:NnTF`

```

9320 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
9321 { \__file_name_sanitize:nn {#2} { \__ior_open_aux:NnTF #1 } }
9322 \cs_generate_variant:Nn \ior_open:NnT { c }
9323 \cs_generate_variant:Nn \ior_open:NnF { c }
9324 \cs_generate_variant:Nn \ior_open:NnTF { c }
9325 \cs_new_protected:Npn \__ior_open_aux:NnTF #1#2
9326 {
9327   \file_add_path:nN {#2} \l__file_internal_name_tl

```

```

9328 \quark_if_no_value:NTF \l__file_internal_name_tl
9329 { \prg_return_false: }
9330 {
9331   \__ior_open:No #1 \l__file_internal_name_tl
9332   \prg_return_true:
9333 }
9334 }

```

(End definition for `\ior_open:Nn` and `\ior_open:cn`. These functions are documented on page ??.)

`__ior_open:Nn` The stream allocation itself uses the fact that there is a list of all of those available, so allocation is simply a question of using the number at the top of the list. In package **`__ior_open:No`** 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).

`__ior_open_stream:Nn`

```

9335 \cs_new_protected:Npn \__ior_open:Nn #1#2
9336 {
9337   \ior_close:N #1
9338   \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
9339   { \__ior_open_stream:Nn #1 {#2} }
9340 <*initex>
9341   { \__msg_kernel_fatal:nn { kernel } { input-streams-exhausted } }
9342 </initex>
9343 <*package>
9344   {
9345     \newread #1
9346     \tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
9347     \__ior_open_stream:Nn #1 {#2}
9348   }
9349 </package>
9350 }
9351 \cs_generate_variant:Nn \__ior_open:Nn { No }
9352 \cs_new_protected:Npn \__ior_open_stream:Nn #1#2
9353 {
9354   \tex_global:D \tex_chardef:D #1 = \l__ior_stream_tl \scan_stop:
9355   \prop_gput:NVn \g__ior_streams_prop #1 {#2}
9356   \tex_openin:D #1 #2 \scan_stop:
9357 }

```

(End definition for `__ior_open:Nn` and `__ior_open:No`. These functions are documented on page ??.)

`\ior_close:N` Closing a stream means getting rid of it at the T_EX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range [0, 15]), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.

`\ior_close:c`

```

9358 \cs_new_protected:Npn \ior_close:N #1
9359 {
9360   \int_compare:nT { \c_minus_one < #1 < \c_sixteen }
9361   {

```

```

9362         \tex_closein:D #1
9363         \prop_gremove:NV \g__ior_streams_prop #1
9364         \seq_if_in:NVF \g__ior_streams_seq #1
9365         { \seq_gpush:NV \g__ior_streams_seq #1 }
9366         \cs_gset_eq:NN #1 \c_term_ior
9367     }
9368 }
9369 \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`.

```

9370 \cs_new_protected_nopar:Npn \ior_list_streams:
9371 { \__ior_list_streams:Nn \g__ior_streams_prop { input } }
9372 \cs_new_protected:Npn \__ior_list_streams:Nn #1#2
9373 {
9374     \__msg_term:nnn { LaTeX / kernel }
9375     { \prop_if_empty:NTF #1 { show-no-stream } { show-open-streams } }
9376     {#2}
9377     \__msg_show_variable:n
9378     { \prop_map_function:NN #1 \__msg_show_item_unbraced:nn }
9379 }

```

(End definition for `\ior_list_streams:.` This function is documented on page 163.)

19.2.3 Reading input

`\if_eof:w` The primitive conditional

```

9380 \cs_new_eq:NN \if_eof:w \tex_ifeof:D

```

(End definition for `\if_eof:w`.)

`\ior_if_eof_p:N` To test if some particular input stream is exhausted the following conditional is provided.

`\ior_if_eof:N \overline{TF}`

```

9381 \prg_new_conditional:Nnn \ior_if_eof:N { p , T , F , TF }
9382 {
9383     \cs_if_exist:NTF #1
9384     {
9385         \if_int_compare:w #1 = \c_sixteen
9386         \prg_return_true:
9387     }
9388     \else:
9389         \if_eof:w #1
9390         \prg_return_true:
9391     }
9392     \else:
9393         \prg_return_false:
9394     }
9395     \fi:
9396 }

```

(End definition for `\ior_if_eof:N`. These functions are documented on page 164.)

`\ior_get:NN` And here we read from files.

```
9397 \cs_new_protected:Npn \ior_get:NN #1#2
9398 { \tex_read:D #1 to #2 }
```

(End definition for `\ior_get:NN`. This function is documented on page 163.)

`\ior_get_str:NN` Reading as strings is a more complicated wrapper, as we wish to remove the endline character.

```
9399 \cs_new_protected:Npn \ior_get_str:NN #1#2
9400 {
9401   \use:x
9402   {
9403     \int_set_eq:NN \tex_endlinechar:D \c_minus_one
9404     \exp_not:n { \etex_readline:D #1 to #2 }
9405     \int_set:Nn \tex_endlinechar:D { \int_use:N \tex_endlinechar:D }
9406   }
9407 }
```

(End definition for `\ior_get_str:NN`. This function is documented on page 163.)

`\g__file_internal_ior` Needed by the higher-level code, but cannot be created until here.

```
9408 \ior_new:N \g__file_internal_ior
```

(End definition for `\g__file_internal_ior`. This variable is documented on page ??.)

19.3 Output operations

```
9409 <@@=iow>
```

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

19.3.1 Variables and constants

`\c_log_iow` Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`)
`\c_term_iow` and to both the terminal and transcript file (`\c_term_iow`).

```
9410 \cs_new_eq:NN \c_log_iow \c_minus_one
9411 \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 166.)

`\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!

```
9412 \seq_new:N \g__iow_streams_seq
9413 <*initex>
9414 \seq_gset_eq:NN \g__iow_streams_seq \g__ior_streams_seq
9415 </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.

```
9416 \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.

```

9417 \prop_new:N \g__iow_streams_prop
9418 <*package>
9419 \prop_put:Nnn \g__iow_streams_prop { 0 } { LaTeX2e-reserved }
9420 \prop_put:Nnn \g__iow_streams_prop { 1 } { LaTeX2e-reserved }
9421 \prop_put:Nnn \g__iow_streams_prop { 2 } { LaTeX2e-reserved }
9422 </package>

```

(End definition for `\g__iow_streams_prop`. This variable is documented on page ??.)

19.4 Stream management

`\iow_new:N` Reserving a new stream is done by defining the name as equal to writing to the terminal:
`\iow_new:c` odd but at least consistent.

```

9423 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
9424 \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
9425 \cs_new_protected:Npn \iow_open:Nn #1#2
9426 { \__file_name_sanitize:nn {#2} { \__iow_open:Nn #1 } }
9427 \cs_generate_variant:Nn \iow_open:Nn { c }
9428 \cs_new_protected:Npn \__iow_open:Nn #1#2
9429 {
9430   \iow_close:N #1
9431   \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
9432   { \__iow_open_stream:Nn #1 {#2} }
9433 <*initex>
9434   { \__msg_kernel_fatal:nn { kernel } { output-streams-exhausted } }
9435 </initex>
9436 <*package>
9437   {
9438     \newwrite #1
9439     \tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
9440     \__iow_open_stream:Nn #1 {#2}
9441   }
9442 </package>
9443 }
9444 \cs_generate_variant:Nn \__iow_open:Nn { No }
9445 \cs_new_protected:Npn \__iow_open_stream:Nn #1#2
9446 {
9447   \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
9448   \prop_gput:NVn \g__iow_streams_prop #1 {#2}
9449   \tex_immediate:D \tex_openout:D #1 #2 \scan_stop:
9450 }

```

(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

```

9451 \cs_new_protected:Npn \iow_close:N #1
9452 {
9453   \int_compare:nT { \c_minus_one < #1 < \c_sixteen }
9454   {
9455     \tex_immediate:D \tex_closeout:D #1
9456     \prop_gremove:NV \g__iow_streams_prop #1
9457     \seq_if_in:NVF \g__iow_streams_seq #1
9458     { \seq_gpush:NV \g__iow_streams_seq #1 }
9459     \cs_gset_eq:NN #1 \c_term_ior
9460   }
9461 }
9462 \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.

```

9463 \cs_new_protected_nopar:Npn \iow_list_streams:
9464 { \__iow_list_streams:Nn \g__iow_streams_prop { output } }
9465 \cs_new_eq:NN \__iow_list_streams:Nn \__ior_list_streams:Nn

```

(End definition for \iow_list_streams:. This function is documented on page ??.)

19.4.1 Deferred writing

\iow_shipout_x:Nn
\iow_shipout_x:Nx

First the easy part, this is the primitive, which expects its argument to be braced.

```

9466 \cs_new_protected:Npn \iow_shipout_x:Nn #1#2
9467 { \tex_write:D #1 {#2} }
9468 \cs_generate_variant:Nn \iow_shipout_x:Nn { Nx }

```

(End definition for \iow_shipout_x:Nn and \iow_shipout_x:Nx. These functions are documented on page ??.)

\iow_shipout:Nn
\iow_shipout:Nx

With ϵ -TeX available deferred writing without expansion is easy.

```

9469 \cs_new_protected:Npn \iow_shipout:Nn #1#2
9470 { \tex_write:D #1 { \exp_not:n {#2} } }
9471 \cs_generate_variant:Nn \iow_shipout:Nn { Nx }

```

(End definition for \iow_shipout:Nn and \iow_shipout:Nx. These functions are documented on page ??.)

19.4.2 Immediate writing

\iow_now:Nn
\iow_now:Nx

This routine writes the second argument onto the output stream without expansion. If this stream isn't open, the output goes to the terminal instead. If the first argument is no output stream at all, we get an internal error. We don't use the expansion done by `\write` to get the Nx variant, because it differs in subtle ways from x-expansion, namely, macro parameter characters would not need to be doubled.

```

9472 \cs_new_protected:Npn \iow_now:Nn #1#2
9473 { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
9474 \cs_generate_variant:Nn \iow_now:Nn { Nx }

```


(End definition for `\iow_now:Nn` and `\iow_now:Nx`. These functions are documented on page ??.)

```

\iow_log:n Writing to the log and the terminal directly are relatively easy.
\iow_log:x 9475 \cs_set_protected_nopar:Npn \iow_log:x { \iow_now:Nx \c_log_iow }
\iow_term:n 9476 \cs_new_protected_nopar:Npn \iow_log:n { \iow_now:Nn \c_log_iow }
\iow_term:x 9477 \cs_set_protected_nopar:Npn \iow_term:x { \iow_now:Nx \c_term_iow }
           9478 \cs_new_protected_nopar:Npn \iow_term:n { \iow_now:Nn \c_term_iow }

```

(End definition for `\iow_log:n` and `\iow_log:x`. These functions are documented on page ??.)

19.4.3 Special characters for writing

`\iow_newline:` Global variable holding the character that forces a new line when something is written to an output stream

```
9479 \cs_new_nopar:Npn \iow_newline: { ^^J }
```

(End definition for `\iow_newline:`. This function is documented on page 165.)

`\iow_char:N` Function to write any escaped char to an output stream.

```
9480 \cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(End definition for `\iow_char:N`. This function is documented on page 164.)

19.4.4 Hard-wrapping lines to a character count

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

`\l_iow_line_count_int` This is the “raw” number of characters in a line which can be written to the terminal. The standard value is the line length typically used by `TeXLive` and `MikTeX`.

```
9481 \int_new:N \l_iow_line_count_int
9482 \int_set:Nn \l_iow_line_count_int { 78 }
```

(End definition for `\l_iow_line_count_int`. This variable is documented on page 166.)

`\l__iow_target_count_int` This stores the target line count: the full number of characters in a line, minus any part for a leader at the start of each line.

```
9483 \int_new:N \l__iow_target_count_int
```

(End definition for `\l__iow_target_count_int`.)

`\l_iow_current_line_int` These store the number of characters in the line and word currently being constructed, and the current indentation, respectively.

```

\l_iow_current_word_int 9484 \int_new:N \l__iow_current_line_int
\l_iow_current_indentation_int 9485 \int_new:N \l__iow_current_word_int
                             9486 \int_new:N \l__iow_current_indentation_int

```

(End definition for `\l_iow_current_line_int`, `\l__iow_current_word_int`, and `\l__iow_current_indentation_int`.)

`\l__iow_current_line_tl` These hold the current line of text and current word, and a number of spaces for indentation, respectively.

```

\l__iow_current_word_tl 9487 \tl_new:N \l__iow_current_line_tl
\l__iow_current_indentation_tl 9488 \tl_new:N \l__iow_current_word_tl
                             9489 \tl_new:N \l__iow_current_indentation_tl

```

(End definition for `\l__iow_current_line_tl`, `\l__iow_current_word_tl`, and `\l__iow_current_indentation_tl`.)

`\l__iow_wrap_tl` Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.

```
9490 \tl_new:N \l__iow_wrap_tl
(End definition for \l__iow_wrap_tl.)
```

`\l__iow_newline_tl` The token list inserted to produce the new line, with the *run-on text*.

```
9491 \tl_new:N \l__iow_newline_tl
(End definition for \l__iow_newline_tl.)
```

`\l__iow_line_start_bool` Boolean to avoid adding a space at the beginning of forced newlines, and to know when to add the indentation.

```
9492 \bool_new:N \l__iow_line_start_bool
(End definition for \l__iow_line_start_bool.)
```

`\c_catcode_other_space_tl` Lowercase a character with category code 12 to produce an “other” space. We can do everything within the group, because `\tl_const:Nn` defines its argument globally.

```
9493 \group_begin:
9494   \char_set_catcode_other:N \*
9495   \char_set_lccode:nn {'\*} {'\ }
9496   \tl_to_lowercase:n { \tl_const:Nn \c_catcode_other_space_tl { * } }
9497 \group_end:
(End definition for \c_catcode_other_space_tl. This function is documented on page 166.)
```

`\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
9498 \group_begin:
9499   \int_set_eq:NN \tex_escapechar:D \c_minus_one
9500   \tl_const:Nx \c__iow_wrap_marker_tl
9501     { \tl_to_str:n { ^^I ^^O ^^W ^^_ ^^W ^^R ^^A ^^P } }
9502 \group_end:
9503 \tl_map_inline:nn
9504   { { end } { newline } { indent } { unindent } }
9505   {
9506     \tl_const:cx { c__iow_wrap_ #1 _marker_tl }
9507     {
9508       \c_catcode_other_space_tl
9509       \c__iow_wrap_marker_tl
9510       \c_catcode_other_space_tl
9511       #1
9512       \c_catcode_other_space_tl
9513     }
9514   }
```

(End definition for `\c__iow_wrap_marker_tl`. This function is documented on page 166.)

`\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.

```

9515 \cs_new_protected:Npn \iow_indent:n #1 { }
9516 \cs_new:Npx \__iow_indent:n #1
9517 {
9518   \c__iow_wrap_indent_marker_tl
9519   #1
9520   \c__iow_wrap_unindent_marker_tl
9521 }

```

(End definition for `\iow_indent:n`. This function is documented on page 165.)

`\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.

```

9522 \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
9523 {
9524   \group_begin:
9525     \int_set_eq:NN \tex_escapechar:D \c_minus_one
9526     \cs_set_nopar:Npx \{ { \token_to_str:N \{ }
9527     \cs_set_nopar:Npx \# { \token_to_str:N \# }
9528     \cs_set_nopar:Npx \} { \token_to_str:N \} }
9529     \cs_set_nopar:Npx \% { \token_to_str:N \% }
9530     \cs_set_nopar:Npx \~ { \token_to_str:N \~ }
9531     \int_set:Nn \tex_escapechar:D { 92 }
9532     \cs_set_eq:NN \ \ \c__iow_wrap_newline_marker_tl
9533     \cs_set_eq:NN \_ \c_catcode_other_space_tl
9534     \cs_set_eq:NN \iow_indent:n \__iow_indent:n
9535     #3
9536   <*initex>
9537     \tl_set:Nx \l__iow_wrap_tl {#1}
9538   </initex>
9539   <*package>
9540     \protected@edef \l__iow_wrap_tl {#1}
9541   </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.

```

9542 \tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
9543 \tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
9544 \tl_replace_all:Nnn \l__iow_newline_tl { ~ } { \c_space_tl }
9545 \int_set:Nn \l__iow_target_count_int
9546 { \l__iow_line_count_int - \tl_count:N \l__iow_newline_tl + \c_one }
9547 \int_zero:N \l__iow_current_indentation_int
9548 \tl_clear:N \l__iow_current_indentation_tl
9549 \int_zero:N \l__iow_current_line_int
9550 \tl_clear:N \l__iow_current_line_tl
9551 \bool_set_true:N \l__iow_line_start_bool
9552 \use:x
9553 {
9554   \exp_not:n { \tl_clear:N \l__iow_wrap_tl }
9555   \__iow_wrap_loop:w
9556   \tl_to_str:N \l__iow_wrap_tl
9557   \tl_to_str:N \c__iow_wrap_end_marker_tl
9558   \c_space_tl \c_space_tl
9559   \exp_not:N \q_stop
9560 }
9561 \exp_args:NNo \group_end:
9562 #4 \l__iow_wrap_tl
9563 }

```

(End definition for \iow_wrap:nnnN. This function is documented on page 165.)

__iow_wrap_loop:w The loop grabs one word in the input, and checks whether it is the special marker, or a normal word.

```

9564 \cs_new_protected:Npn \__iow_wrap_loop:w #1 ~ %
9565 {
9566   \tl_set:Nn \l__iow_current_word_tl {#1}
9567   \tl_if_eq:NNTF \l__iow_current_word_tl \c__iow_wrap_marker_tl
9568   { \__iow_wrap_special:w }
9569   { \__iow_wrap_word: }
9570 }

```

(End definition for __iow_wrap_loop:w.)

__iow_wrap_word: For a normal word, update the line count, then test if the current word would fit in the current line, and call the appropriate function. If the word fits in the current line, __iow_wrap_word_fits: add it to the line, preceded by a space unless it is the first word of the line. Otherwise, __iow_wrap_word_newline: 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.

```

9571 \cs_new_protected_nopar:Npn \__iow_wrap_word:
9572 {
9573   \int_set:Nn \l__iow_current_word_int
9574   { \__str_count_ignore_spaces:N \l__iow_current_word_tl }
9575   \int_add:Nn \l__iow_current_line_int { \l__iow_current_word_int }
9576   \int_compare:nNnTF \l__iow_current_line_int < \l__iow_target_count_int
9577   { \__iow_wrap_word_fits: }
9578   { \__iow_wrap_word_newline: }
9579   \__iow_wrap_loop:w

```

```

9580 }
9581 \cs_new_protected_nopar:Npn \__iow_wrap_word_fits:
9582 {
9583   \bool_if:NTF \l__iow_line_start_bool
9584   {
9585     \bool_set_false:N \l__iow_line_start_bool
9586     \tl_put_right:Nx \l__iow_current_line_tl
9587       { \l__iow_current_indentation_tl \l__iow_current_word_tl }
9588     \int_add:Nn \l__iow_current_line_int
9589       { \l__iow_current_indentation_int }
9590   }
9591   {
9592     \tl_put_right:Nx \l__iow_current_line_tl
9593       { ~ \l__iow_current_word_tl }
9594     \int_incr:N \l__iow_current_line_int
9595   }
9596 }
9597 \cs_new_protected_nopar:Npn \__iow_wrap_word_newline:
9598 {
9599   \tl_put_right:Nx \l__iow_wrap_tl
9600     { \l__iow_current_line_tl \l__iow_newline_tl }
9601   \int_set:Nn \l__iow_current_line_int
9602     {
9603       \l__iow_current_word_int
9604       + \l__iow_current_indentation_int
9605     }
9606   \tl_set:Nx \l__iow_current_line_tl
9607     { \l__iow_current_indentation_tl \l__iow_current_word_tl }
9608 }

```

(End definition for `__iow_wrap_word:`. This function is documented on page 165.)

<code>__iow_wrap_special:w</code> <code>__iow_wrap_newline:w</code> <code>__iow_wrap_indent:w</code> <code>__iow_wrap_unindent:w</code> <code>__iow_wrap_end:w</code>	<p>When the “special” marker is encountered, read what operation to perform, as a space-delimited argument, perform it, and remember to loop. In fact, to avoid spurious spaces when two special actions follow each other, we look ahead for another copy of the marker. Forced newlines are almost identical to those caused by overflow, except that here the word is empty. To indent more, add four spaces to the start of the indentation token list. To reduce indentation, rebuild the indentation token list using <code>\prg_replicate:nn</code>. At the end, we simply save the last line (without the run-on text), and prevent the loop.</p>
--	---

```

9609 \cs_new_protected:Npn \__iow_wrap_special:w #1 ~ #2 ~ #3 ~ %
9610 {
9611   \use:c { __iow_wrap_#1: }
9612   \str_if_eq_x:nnTF { #2~#3 } { ~ \c__iow_wrap_marker_tl }
9613     { \__iow_wrap_special:w }
9614     { \__iow_wrap_loop:w #2 ~ #3 ~ }
9615 }
9616 \cs_new_protected_nopar:Npn \__iow_wrap_newline:
9617 {
9618   \tl_put_right:Nx \l__iow_wrap_tl
9619     { \l__iow_current_line_tl \l__iow_newline_tl }

```

```

9620 \int_zero:N \l__iow_current_line_int
9621 \tl_clear:N \l__iow_current_line_tl
9622 \bool_set_true:N \l__iow_line_start_bool
9623 }
9624 \cs_new_protected_nopar:Npx \__iow_wrap_indent:
9625 {
9626 \int_add:Nn \l__iow_current_indentation_int \c_four
9627 \tl_put_right:Nx \exp_not:N \l__iow_current_indentation_tl
9628 { \c_space_tl \c_space_tl \c_space_tl \c_space_tl }
9629 }
9630 \cs_new_protected_nopar:Npn \__iow_wrap_unindent:
9631 {
9632 \int_sub:Nn \l__iow_current_indentation_int \c_four
9633 \tl_set:Nx \l__iow_current_indentation_tl
9634 { \prg_replicate:nn \l__iow_current_indentation_int { ~ } }
9635 }
9636 \cs_new_protected_nopar:Npn \__iow_wrap_end:
9637 {
9638 \tl_put_right:Nx \l__iow_wrap_tl
9639 { \l__iow_current_line_tl }
9640 \use_none_delimit_by_q_stop:w
9641 }

```

(End definition for `__iow_wrap_special:w`. This function is documented on page 165.)

```

\__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.

```

9642 \cs_new_nopar:Npn \__str_count_ignore_spaces:N
9643 { \exp_args:No \__str_count_ignore_spaces:n }
9644 \cs_new:Npn \__str_count_ignore_spaces:n #1
9645 {
9646 \__int_value:w \__int_eval:w
9647 \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
9648 { X8 } { X7 } { X6 } { X5 } { X4 } { X3 } { X2 } { X1 } { X0 } \q_stop
9649 \__int_eval_end:
9650 }
9651 \cs_new:Npn \__str_count_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
9652 {
9653 \if_catcode:w X #9
9654 \exp_after:wN \use_none_delimit_by_q_stop:w
9655 \else:
9656 9 +
9657 \exp_after:wN \__str_count_loop:NNNNNNNNN
9658 \fi:
9659 }

```

(End definition for `__str_count_ignore_spaces:N`. This function is documented on page 165.)

19.5 Messages

```

9660 \__msg_kernel_new:nnnn { kernel } { file-not-found }

```

```

9661 { File~'#1'~not~found. }
9662 {
9663   The~requested~file~could~not~be~found~in~the~current~directory,~
9664   in~the~TeX~search~path~or~in~the~LaTeX~search~path.
9665 }
9666 \_msg_kernel_new:nnnn { kernel } { input-streams-exhausted }
9667 { Input~streams~exhausted }
9668 {
9669   TeX~can~only~open~up~to~16~input~streams~at~one~time.\\
9670   All~16~are~currently~in~use,~and~something~wanted~to~open~
9671   another~one.
9672 }
9673 \_msg_kernel_new:nnnn { kernel } { output-streams-exhausted }
9674 { Output~streams~exhausted }
9675 {
9676   TeX~can~only~open~up~to~16~output~streams~at~one~time.\\
9677   All~16~are~currently~in~use,~and~something~wanted~to~open~
9678   another~one.
9679 }
9680 \_msg_kernel_new:nnnn { kernel } { space-in-file-name }
9681 { Space~in~file~name~'#1'. }
9682 {
9683   Spaces~are~not~permitted~in~files~loaded~by~LaTeX: \\
9684   Further~errors~may~follow!
9685 }

```

19.6 Deprecated functions

Deprecated on 2012-06-28, for removal by 2012-12-31.

`\iow_wrap:xnnnN` This was renamed and one unneeded argument was removed.

```

9686 \*deprecated
9687 \cs_new_protected:Npn \iow_wrap:xnnnN #1#2#3#4#5
9688 { \iow_wrap:nnnN {#1} {#2} {#4} #5 }
9689 \*deprecated

```

(End definition for `\iow_wrap:xnnnN`. This function is documented on page ??.)

Deprecated on 2012-06-24, for removal by 2012-12-31.

`\l_iow_line_length_int` Simple rename. Here we copy the `TeX` register.

```

9690 \*deprecated
9691 \cs_new_eq:NN \l_iow_line_length_int \l_iow_line_count_int
9692 \*deprecated

```

(End definition for `\l_iow_line_length_int`. This variable is documented on page ??.)

`\ior_to:NN` The local variants are renames, while the global variants are deprecated and add the `TeX` primitive `\global`.

`\ior_gto:NN`

`\ior_str_to:NN`

`\ior_str_gto:NN`

```

9693 \cs_new_eq:NN \ior_to:NN \ior_get:NN
9694 \cs_new_protected_nopar:Npn \ior_gto:NN { \tex_global:D \ior_to:NN }
9695 \cs_new_eq:NN \ior_str_to:NN \ior_get_str:NN
9696 \cs_new_protected_nopar:Npn \ior_str_gto:NN { \tex_global:D \ior_str_to:NN }

```

(End definition for \ior_to:NN and others. These functions are documented on page ??.)

Deprecated on 2012-02-10, for removal by 2012-05-31.

\iow_now_when_avail:Nn For writing only if the stream requested is open at all.

```
\iow_now_when_avail:Nx
9697 \*deprecated
9698 \cs_new_protected:Npn \iow_now_when_avail:Nn #1
9699 { \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nn #1 } }
9700 \cs_new_protected:Npn \iow_now_when_avail:Nx #1
9701 { \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nx #1 } }
9702 \*deprecated
```

(End definition for \iow_now_when_avail:Nn and \iow_now_when_avail:Nx. These functions are documented on page ??.)

Deprecated on 2011-05-27, for removal by 2011-08-31.

\iow_now_buffer_safe:Nn This is much more easily done using the wrapping system: there is an expansion there,
\iow_now_buffer_safe:Nx so a bit of a hack is needed.

```
9703 \*deprecated
9704 \cs_new_protected:Npn \iow_now_buffer_safe:Nn #1#2
9705 { \iow_wrap:xnnnN { \exp_not:n {#2} } { } \c_zero { } \iow_now:Nn #1 }
9706 \cs_new_protected:Npn \iow_now_buffer_safe:Nx #1#2
9707 { \iow_wrap:xnnnN {#2} { } \c_zero { } \iow_now:Nn #1 }
9708 \*deprecated
```

(End definition for \iow_now_buffer_safe:Nn and \iow_now_buffer_safe:Nx. These functions are documented on page ??.)

\ior_open_streams: Slightly misleading names.

```
\iow_open_streams:
9709 \*deprecated
9710 \cs_new_eq:NN \ior_open_streams: \ior_list_streams:
9711 \cs_new_eq:NN \iow_open_streams: \iow_list_streams:
9712 \*deprecated
```

(End definition for \ior_open_streams:. This function is documented on page ??.)

```
9713 \*initex | package)
```

20 l3fp implementation

```
9714 \*package
9715 \ProvidesExplPackage
9716 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
9717 \__expl_package_check:
9718 \*package
```

21 l3fp-aux implementation

```
9719 \*initex | package)
9720 \@@=fp)
```


22 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 $\pm \backslash c_fp_max_exponent_int = \pm 10000$ inclusive.

Additionally, positive and negative floating point numbers may only be stored with $1000 \leq \langle X_1 \rangle < 10000$. This requirement is necessary in order to preserve accuracy and speed.

22.1 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.

```
9721 \cs_new:Npn \__fp_use_none_stop_f:n #1 { \exp_stop_f: }
(End definition for \__fp_use_none_stop_f:n.)
```

`__fp_use_s:n` Those functions place a semicolon after one or two arguments (typically digits).

```
\__fp_use_s:nn 9722 \cs_new:Npn \__fp_use_s:n #1 { #1; }
9723 \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`

```
9724 \cs_new:Npn \__fp_use_none_until_s:w #1; { }
9725 \cs_new:Npn \__fp_use_i_until_s:nw #1#2; { #1 }
9726 \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.

```
9727 \cs_new:Npn \__fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }
(End definition for \__fp_reverse_args:Nww.)
```

22.2 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.

```

9728 \__scan_new:N \s__fp
9729 \cs_new_protected:Npn \__fp_chk:w #1 ;
9730 {
9731   \msg_kernel_error:nnx { kernel } { misused-fp }
9732   { \fp_to_tl:n { \s__fp \__fp_chk:w #1 ; } }
9733 }
(End definition for \s__fp and \__fp_chk:w.)

```

`\s__fp_mark` Aliases of `\tex_relax:D`, used to terminate expressions.

```

\s__fp_stop
9734 \__scan_new:N \s__fp_mark
9735 \__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
9736 \__scan_new:N \s__fp_invalid
9737 \__scan_new:N \s__fp_underflow
9738 \__scan_new:N \s__fp_overflow
9739 \__scan_new:N \s__fp_division
9740 \__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`, `exact`, describing how the floating point was created. We define the floating points here as “exact”.

```

9741 \tl_const:Nn \c_zero_fp { \s__fp \__fp_chk:w 0 0 \s__fp_exact ; }
9742 \tl_const:Nn \c_minus_zero_fp { \s__fp \__fp_chk:w 0 2 \s__fp_exact ; }
9743 \tl_const:Nn \c_inf_fp { \s__fp \__fp_chk:w 2 0 \s__fp_exact ; }
9744 \tl_const:Nn \c_minus_inf_fp { \s__fp \__fp_chk:w 2 2 \s__fp_exact ; }
9745 \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.

```
9746 \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`

```
9747 \cs_new:Npn \__fp_zero_fp:N #1 { \s__fp \__fp_chk:w 0 #1 \s__fp_underflow ; }
9748 \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.

```
9749 \cs_new:Npn \__fp_min_fp:N #1
9750 {
9751   \s__fp \__fp_chk:w 1 #1
9752   { \int_eval:n { - \c__fp_max_exponent_int } }
9753   {1000} {0000} {0000} {0000} ;
9754 }
9755 \cs_new:Npn \__fp_max_fp:N #1
9756 {
9757   \s__fp \__fp_chk:w 1 #1
9758   { \int_use:N \c__fp_max_exponent_int }
9759   {9999} {9999} {9999} {9999} ;
9760 }
(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.

```
9761 \cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
9762 {
9763   \if_meaning:w 1 #1
9764   \exp_after:wN \__fp_use_ii_until_s:nnw
9765   \else:
9766   \exp_after:wN \__fp_use_i_until_s:nw
9767   \exp_after:wN 0
9768   \fi:
9769 }
(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.

```
9770 \cs_new:Npn \__fp_neg_sign:N #1
9771 { \__int_eval:w \c_two - #1 \__int_eval_end: }
(End definition for \__fp_neg_sign:N.)
```

22.3 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`.

```

9772 \cs_new:Npn \__fp_sanitize:Nw #1 #2;
9773 {
9774   \if_case:w \if_int_compare:w #2 > \c__fp_max_exponent_int \c_one \else:
9775     \if_int_compare:w #2 < - \c__fp_max_exponent_int \c_two \else:
9776     \if_meaning:w 1 #1 \c_three \else: \c_zero \fi: \fi: \fi:
9777   \or: \exp_after:wN \__fp_overflow:w
9778   \or: \exp_after:wN \__fp_underflow:w
9779   \or: \exp_after:wN \__fp_sanitize_zero:w
9780   \fi:
9781   \s__fp \__fp_chk:w 1 #1 {#2}
9782 }
9783 \cs_new:Npn \__fp_sanitize:wN #1; #2 { \__fp_sanitize:Nw #2 #1; }
9784 \cs_new:Npn \__fp_sanitize_zero:w \s__fp \__fp_chk:w #1 #2 #3; { \c_zero_fp }

```

(End definition for `__fp_sanitize:Nw` and `__fp_sanitize:wN`. These functions are documented on page ??.)

22.4 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.

```

9785 \cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
9786 {
9787   \if_meaning:w 1 #1
9788     \exp_after:wN \__fp_exp_after_normal:nNNw
9789   \else:
9790     \exp_after:wN \__fp_exp_after_special:nNNw
9791   \fi:
9792   { }
9793   #1
9794 }
9795 \cs_new:Npn \__fp_exp_after_o:nw #1 \s__fp \__fp_chk:w #2
9796 {
9797   \if_meaning:w 1 #2
9798     \exp_after:wN \__fp_exp_after_normal:nNNw
9799   \else:
9800     \exp_after:wN \__fp_exp_after_special:nNNw
9801   \fi:
9802   { #1 }
9803   #2
9804 }

```

```

9805 \cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
9806 {
9807   \if_meaning:w 1 #2
9808     \exp_after:wN \__fp_exp_after_normal:nNNw
9809   \else:
9810     \exp_after:wN \__fp_exp_after_special:nNNw
9811   \fi:
9812   { \tex_romannumeral:D -'0 #1 }
9813   #2
9814 }

```

(End definition for __fp_exp_after_o:w. This function is documented on page ??.)

__fp_exp_after_special:nNNw Special floating point numbers are easy to jump over since they contain few tokens.

```

9815 \cs_new:Npn \__fp_exp_after_special:nNNw #1#2#3#4;
9816 {
9817   \exp_after:wN \s__fp
9818   \exp_after:wN \__fp_chk:w
9819   \exp_after:wN #2
9820   \exp_after:wN #3
9821   \exp_after:wN #4
9822   \exp_after:wN ;
9823   #1
9824 }

```

(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.

```

9825 \cs_new:Npn \__fp_exp_after_normal:nNNw #1 1 #2 #3 #4#5#6#7;
9826 {
9827   \exp_after:wN \__fp_exp_after_normal:Nwwwww
9828   \exp_after:wN #2
9829   \__int_value:w #3 \exp_after:wN ;
9830   \__int_value:w 1 #4 \exp_after:wN ;
9831   \__int_value:w 1 #5 \exp_after:wN ;
9832   \__int_value:w 1 #6 \exp_after:wN ;
9833   \__int_value:w 1 #7 \exp_after:wN ; #1
9834 }
9835 \cs_new:Npn \__fp_exp_after_normal:Nwwwww
9836   #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
9837   { \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

```

9838 \cs_new:Npn \__fp_exp_after_array_f:w #1
9839 {
9840   \cs:w \__fp_exp_after \__fp_type_from_scan:N #1 _f:nw \cs_end:
9841   { \__fp_exp_after_array_f:w }
9842   #1

```

```

9843 }
9844 \cs_new_eq:NN \__fp_exp_after_stop_f:nw \use_none:nn
(End definition for \__fp_exp_after_array_f:w. This function is documented on page ??.)

```

22.5 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`
`__fp_pack:NNNNNwn`
`\c__fp_trailing_shift_int`
`\c__fp_middle_shift_int`
`\c__fp_leading_shift_int`

This set of shifts allows for computations involving results in the range $[-4 \cdot 10^8, 5 \cdot 10^8 - 1]$. Shifted values all have exactly 9 digits. The `__fp_pack:NNNNNwn` function brings a braced *<continuation>* up through the levels of expansion.

```
9845 \int_const:Nn \c__fp_leading_shift_int { - 5 0000 }
9846 \int_const:Nn \c__fp_middle_shift_int { 5 0000 * 9999 }
9847 \int_const:Nn \c__fp_trailing_shift_int { 5 0000 * 10000 }
9848 \cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }
9849 \cs_new:Npn \__fp_pack:NNNNNwn #1 #2#3#4#5 #6; #7
9850 { + #1#2#3#4#5 ; {#7} {#6} }
```

(End definition for `__fp_pack:NNNNNw` and `__fp_pack:NNNNNwn`. These functions are documented on page ??.)

`__fp_pack_big:NNNNNNw`
`__fp_pack_big:NNNNNNwn`
`\c__fp_big_trailing_shift_int`
`\c__fp_big_middle_shift_int`
`\c__fp_big_leading_shift_int`

This set of shifts allows for computations involving results in the range $[-5 \cdot 10^8, 6 \cdot 10^8 - 1]$ (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper bound is due to T_EX's limit of $2^{31} - 1$ on integers. The shifts are chosen to be roughly the mid-point of 10^9 and 2^{31} , the two bounds on 10-digit integers in T_EX.

```
9851 \int_const:Nn \c__fp_big_leading_shift_int { - 15 2374 }
9852 \int_const:Nn \c__fp_big_middle_shift_int { 15 2374 * 9999 }
9853 \int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 }
9854 \cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
9855 { + #1#2#3#4#5#6 ; {#7} }
9856 \cs_new:Npn \__fp_pack_big:NNNNNNwn #1#2 #3#4#5#6 #7; #8
9857 { + #1#2#3#4#5#6 ; {#8} {#7} }
```

(End definition for `__fp_pack_big:NNNNNNw` and `__fp_pack_big:NNNNNNwn`. These functions are documented on page ??.)

`__fp_pack_Bigg:NNNNNNw`
`\c__fp_Bigg_trailing_shift_int`
`\c__fp_Bigg_middle_shift_int`
`\c__fp_Bigg_leading_shift_int`

This set of shifts allows for computations involving results in the range $[-1 \cdot 10^9, 147483647]$; the end-point is $2^{31} - 1 - 2 \cdot 10^9 \simeq 1.47 \cdot 10^8$. Shifted values all have exactly 10 digits.

```
9858 \int_const:Nn \c__fp_Bigg_leading_shift_int { - 20 0000 }
9859 \int_const:Nn \c__fp_Bigg_middle_shift_int { 20 0000 * 9999 }
9860 \int_const:Nn \c__fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
9861 \cs_new:Npn \__fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
9862 { + #1#2#3#4#5#6 ; {#7} }
```

(End definition for `__fp_pack_Bigg:NNNNNNw`. This function is documented on page ??.)

`_fp_pack_twice_four:wNNNNNNNN`

Grabs two sets of 4 digits and places them before the semi-colon delimiter. Putting several copies of this function before a semicolon will pack more digits since each will take the digits packed by the others in its first argument.

```
9863 \cs_new:Npn \_fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
9864 { #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.

```

9865 \cs_new:Npn \_fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
9866 { #1 {#2#3#4#5#6#7#8#9} ; }
(End definition for \_fp_pack_eight:wNNNNNNNN.)

```

22.6 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 $\langle shift \rangle$.

```

9867 \cs_new:Npn \_fp_decimate:nNnnnn #1
9868 {
9869   \cs:w
9870   \_fp_decimate_
9871   \if_int_compare:w \_int_eval:w #1 > \c_sixteen
9872     tiny
9873   \else:
9874     \tex_romannumeral:D \_int_eval:w #1
9875   \fi:
9876   :Nnnnn
9877 \cs_end:
9878 }

```

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` If the $\langle shift \rangle$ is zero, or too big, life is very easy.

```

\_fp_decimate_tiny:Nnnnn
9879 \cs_new:Npn \_fp_decimate_:Nnnnn #1 #2#3#4#5
9880 { #1 0 {#2#3} {#4#5} ; }
9881 \cs_new:Npn \_fp_decimate_tiny:Nnnnn #1 #2#3#4#5
9882 { #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 *rounding* 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 *rounding* digit. This triggers the f-expansion of `__fp_decimate_pack:nnnnnnnnnw`,⁶ responsible for building two blocks of 8 digits, and removing the rest. For this to work, #3 alternates between braced and unbraced blocks of 4 digits, in such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.

```

9883 \cs_new:Npn \__fp_tmp:w #1 #2 #3
9884 {
9885   \cs_new:cpn { \__fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
9886   {
9887     \exp_after:wN ##1
9888     \__int_value:w
9889     \exp_after:wN \__fp_round_digit:Nw #2 ;
9890     \__fp_decimate_pack:nnnnnnnnnw #3 ;
9891   }
9892 }
9893 \__fp_tmp:w {i} {\use_none:nnn #50} { 0{#2}#3{#4}#5 }
9894 \__fp_tmp:w {ii} {\use_none:nn #5} { 00{#2}#3{#4}#5 }
9895 \__fp_tmp:w {iii} {\use_none:n #5} { 000{#2}#3{#4}#5 }
9896 \__fp_tmp:w {iv} { #5 } { {0000}#2{#3}#4 #5 }
9897 \__fp_tmp:w {v} {\use_none:nnn #4#5} { 0{0000}#2{#3}#4 #5 }
9898 \__fp_tmp:w {vi} {\use_none:nn #4#5} { 00{0000}#2{#3}#4 #5 }
9899 \__fp_tmp:w {vii} {\use_none:n #4#5} { 000{0000}#2{#3}#4 #5 }
9900 \__fp_tmp:w {viii}{ #4#5 } { {0000}0000{#2}#3 #4 #5 }
9901 \__fp_tmp:w {ix} {\use_none:nnn #3#4+#5} { 0{0000}0000{#2}#3 #4 #5 }
9902 \__fp_tmp:w {x} {\use_none:nn #3#4+#5} { 00{0000}0000{#2}#3 #4 #5 }
9903 \__fp_tmp:w {xi} {\use_none:n #3#4+#5} { 000{0000}0000{#2}#3 #4 #5 }
9904 \__fp_tmp:w {xii} { #3#4+#5 } { {0000}0000{0000}#2 #3 #4 #5 }
9905 \__fp_tmp:w {xiii}{\use_none:nnn#2#3+#4#5} { 0{0000}0000{0000}#2 #3 #4 #5 }
9906 \__fp_tmp:w {xiv} {\use_none:nn #2#3+#4#5} { 00{0000}0000{0000}#2 #3 #4 #5 }
9907 \__fp_tmp:w {xv} {\use_none:n #2#3+#4#5} { 000{0000}0000{0000}#2 #3 #4 #5 }
9908 \__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_decimate_pack:nnnnnnnnnw

```

`__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

⁶No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

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.

```

9909 \cs_new:Npn \__fp_decimate_pack:nnnnnnnnnw #1#2#3#4#5
9910 { \__fp_decimate_pack:nnnnnnw { #1#2#3#4#5 } }
9911 \cs_new:Npn \__fp_decimate_pack:nnnnnnw #1 #2#3#4#5#6
9912 { {#1} {#2#3#4#5#6} }
(End definition for \__fp_round_digit:Nw and \__fp_decimate_pack:nnnnnnnnnw.)

```

22.7 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.

```

9913 \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.

```

9914 \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.

```

9915 \cs_new:Npn \__fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 ;
9916 { \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.

```

9917 \cs_new:Npn \__fp_case_return_same_o:w #1 \fi: #2 \s__fp
9918 { \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.

```

9919 \cs_new:Npn \__fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 ; #5 ;
9920 { \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.

```

9921 \cs_new:Npn \__fp_case_return_i_o:ww #1 \fi: #2 \s__fp #3 ; \s__fp #4 ;
9922 { \fi: \__fp_exp_after_o:w \s__fp #3 ; }
9923 \cs_new:Npn \__fp_case_return_ii_o:ww #1 \fi: #2 \s__fp #3 ;
9924 { \fi: \__fp_exp_after_o:w }
(End definition for \__fp_case_return_i_o:ww and \__fp_case_return_ii_o:ww.)

```

22.8 Small integer floating points

`__fp_small_int:wTF` This function tests if its floating point argument is an integer in the range $[-99999999, 99999999]$.
`__fp_small_int_true:wTF` If it is, the result of the conversion is fed as a braced argument to the *⟨true code⟩*. Other-
`__fp_small_int_normal:NnwTF` wise, the *⟨false code⟩* is performed. First filter special cases: neither `nan` nor infinities are
`__fp_small_int_test:NnnwNTF` integers. Normal numbers with a non-positive exponent are never integers. When the ex-
ponent 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.

```

9925 \cs_new:Npn \__fp_small_int:wTF \s__fp \__fp_chk:w #1
9926 {
9927   \if_case:w #1 \exp_stop_f:
9928     \__fp_case_return:nw { \__fp_small_int_true:wTF 0 ; }
9929   \or:   \exp_after:wN \__fp_small_int_normal:NnwTF
9930   \else: \__fp_case_return:nw \use_ii:nn
9931   \fi:
9932 }
9933 \cs_new:Npn \__fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
9934 \cs_new:Npn \__fp_small_int_normal:NnwTF #1#2#3;
9935 {
9936   \if_int_compare:w #2 > \c_zero
9937     \if_int_compare:w #2 > \c_eight
9938       \exp_after:wN \exp_after:wN

```

```

9939         \exp_after:wN \use_iii:nnn
9940     \else:
9941         \__fp_decimate:nNnnnn { \c_sixteen - #2 }
9942         \__fp_small_int_test:NnnwNTF
9943         #3 #1
9944     \fi:
9945 \else:
9946     \exp_after:wN \use_iii:nnn
9947 \fi:
9948 ;
9949 }
9950 \cs_new:Npn \__fp_small_int_test:NnnwNTF #1#2#3#4; #5
9951 {
9952     \if_meaning:w 0 #1
9953         \exp_after:wN \__fp_small_int_true:wTF
9954         \__int_value:w \if_meaning:w 2 #5 - \fi: #3
9955     \else:
9956         \exp_after:wN \use_i:nn
9957     \fi:
9958 }

```

(End definition for __fp_small_int:wTF. This function is documented on page ??.)

22.9 Length of a floating point array

```

\__fp_array_count:n
\__fp_array_count_loop:Nw

```

Count the number of items in an array of floating points. The technique is very similar to `\tl_count:n`, but with the loop built-in. Checking for the end of the loop is done with the `\use_none:n #1` construction.

```

9959 \cs_new:Npn \__fp_array_count:n #1
9960 {
9961     \int_use:N \__int_eval:w \c_zero
9962     \__fp_array_count_loop:Nw #1 { ? \__prg_break: } ;
9963     \__prg_break_point:
9964     \__int_eval_end:
9965 }
9966 \cs_new:Npn \__fp_array_count_loop:Nw #1#2;
9967 { \use_none:n #1 + \c_one \__fp_array_count_loop:Nw }

```

(End definition for __fp_array_count:n. This function is documented on page ??.)

22.10 x-like expansion expandably

```

\__fp_expand:n
\__fp_expand_loop:nwnN

```

This expandable function behaves in a way somewhat similar to `\use:x`, but much less robust. The argument is `f`-expanded, then the leading item (often a single character token) is moved to a storage area after `\s__fp_mark`, and `f`-expansion is applied again, repeating until the argument is empty. The result built one piece at a time is then inserted in the input stream. Note that spaces are ignored by this procedure, unless surrounded with braces. Multiple tokens which do not need expansion can be inserted within braces.

```

9968 \cs_new:Npn \__fp_expand:n #1

```

```

9969 {
9970   \__fp_expand_loop:nwnN { }
9971   #1 \prg_do_nothing:
9972   \s__fp_mark { } \__fp_expand_loop:nwnN
9973   \s__fp_mark { } \__fp_use_i_until:s:nw ;
9974 }
9975 \cs_new:Npn \__fp_expand_loop:nwnN #1#2 \s__fp_mark #3 #4
9976 {
9977   \exp_after:wN #4 \tex_romannumeral:D -'0
9978   #2
9979   \s__fp_mark { #3 #1 } #4
9980 }

```

(End definition for __fp_expand:n. This function is documented on page ??.)

22.11 Messages

Using a floating point directly is an error.

```

9981 \__msg_kernel_new:nnnn { kernel } { misused-fp }
9982 { A~floating~point~with~value~'#1'~was~misused. }
9983 {
9984   To~obtain~the~value~of~a~floating~point~variable,~use~
9985   '\token_to_str:N \fp_to_decimal:N',~
9986   '\token_to_str:N \fp_to_scientific:N',~or~other~
9987   conversion~functions.
9988 }
9989 </initex | package>

```

23 l3fp-traps Implementation

```

9990 <*initex | package>
9991 <@@=fp>

```

Exceptions should be accessed by an `n`-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

23.1 Flags

`\fp_flag_off:n` Function to turn a flag off. Simply undefine it.

```
9992 \cs_new_protected:Npn \fp_flag_off:n #1
9993 { \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 175.)
```

`\fp_flag_on:n` Function to turn a flag on expandably: use \TeX 's automatic assignment to `\scan_stop:.`

```
9994 \cs_new:Npn \fp_flag_on:n #1
9995 { \exp_args:Nc \use_none:n { l__fp_ #1 _flag_token } }
(End definition for \fp_flag_on:n. This function is documented on page 175.)
```

`\fp_if_flag_on_p:n` Returns true if the flag is on, false otherwise.

```
\fp_if_flag_on:nTF
9996 \prg_new_conditional:Npnn \fp_if_flag_on:n #1 { p , T , F , TF }
9997 {
9998   \if_cs_exist:w l__fp_ #1 _flag_token \cs_end:
9999   \prg_return_true:
10000   \else:
10001   \prg_return_false:
10002   \fi:
10003 }
```

(End definition for `\fp_if_flag_on:n`. These functions are documented on page 175.)

`\l_fp_invalid_operation_flag_token`
`\l_fp_division_by_zero_flag_token`
`\l__fp_overflow_flag_token`
`\l__fp_underflow_flag_token` The IEEE standard defines five exceptions. We currently don't support the "inexact" exception.

```
10004 \cs_new_eq:NN \l_fp_invalid_operation_flag_token \tex_undefined:D
10005 \cs_new_eq:NN \l_fp_division_by_zero_flag_token \tex_undefined:D
10006 \cs_new_eq:NN \l__fp_overflow_flag_token \tex_undefined:D
10007 \cs_new_eq:NN \l__fp_underflow_flag_token \tex_undefined:D
```

(End definition for `\l_fp_invalid_operation_flag_token` and others.)

23.2 Traps

Exceptions can be trapped to obtain custom behaviour. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an N -type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives $+\infty$. Currently, the inexact exception is entirely ignored.

The behaviour when an exception occurs is controlled by the definitions of the functions

- `__fp_invalid_operation:nnw`,
- `__fp_invalid_operation_o:Nww`,
- `__fp_invalid_operation_tl_o:nf`,

- `__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`

```

10008 \cs_new_protected:Npn \fp_trap:nn #1#2
10009 {
10010   \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
10011   {
10012     \clist_if_in:nnTF
10013     { invalid_operation , division_by_zero , overflow , underflow }
10014     {#1}
10015     {
10016       \__msg_kernel_error:nnxx { kernel }
10017       { unknown-fpu-trap-type } {#1} {#2}
10018     }
10019     { \__msg_kernel_error:nnx { kernel } { unknown-fpu-exception } {#1} }
10020   }
10021 }
```

(End definition for `\fp_trap:nn`. This function is documented on page 175.)

`__fp_trap_invalid_operation_set_error:`
`__fp_trap_invalid_operation_set_flag:`
`__fp_trap_invalid_operation_set_none:`
`__fp_trap_invalid_operation_set:N`

We provide three types of trapping for invalid operations: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In most cases, the function produces as a result its first argument, possibly with post-expansion.

```

10022 \cs_new_protected_nopar:Npn \__fp_trap_invalid_operation_set_error:
10023 { \__fp_trap_invalid_operation_set:N \prg_do_nothing: }
10024 \cs_new_protected_nopar:Npn \__fp_trap_invalid_operation_set_flag:
10025 { \__fp_trap_invalid_operation_set:N \use_none:nnnnn }
10026 \cs_new_protected_nopar:Npn \__fp_trap_invalid_operation_set_none:
10027 { \__fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
10028 \cs_new_protected:Npn \__fp_trap_invalid_operation_set:N #1
10029 {
10030   \exp_args:Nno \use:n
10031   { \cs_set:Npn \__fp_invalid_operation:nnw ##1##2##3; }
10032   {
10033     #1
10034     \__fp_error:nnfn { invalid } {##2} { \fp_to_tl:n { ##3; } } { }
10035     \fp_flag_on:n { invalid_operation }
10036     ##1
10037   }
```

```

10038 \exp_args:Nno \use:n
10039 { \cs_set:Npn \__fp_invalid_operation_o:Nnw ##1##2; ##3; }
10040 {
10041   #1
10042   \__fp_error:nfn { invalid-ii }
10043   { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
10044   \fp_flag_on:n { invalid_operation }
10045   \exp_after:wN \c_nan_fp
10046 }
10047 \exp_args:Nno \use:n
10048 { \cs_set:Npn \__fp_invalid_operation_tl_o:nf ##1##2 }
10049 {
10050   #1
10051   \__fp_error:nfn { invalid } {##1} {##2} { }
10052   \fp_flag_on:n { invalid_operation }
10053   \exp_after:wN \c_nan_fp
10054 }
10055 }

```

(End definition for `__fp_trap_invalid_operation_set_error:` and others.)

`__fp_trap_division_by_zero_set_error:` We provide three types of trapping for invalid operations and division by zero: either
`__fp_trap_division_by_zero_set_flag:` produce an error and raise the relevant flag; or only raise the flag; or don't even raise the
`__fp_trap_division_by_zero_set_none:` flag. In all cases, the function must produce a result, namely its first argument, $\pm\infty$ or
`__fp_trap_division_by_zero_set:N` NaN.

```

10056 \cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_error:
10057 { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
10058 \cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_flag:
10059 { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
10060 \cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_none:
10061 { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
10062 \cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
10063 {
10064   \exp_args:Nno \use:n
10065   { \cs_set:Npn \__fp_division_by_zero_o:Nnw ##1##2##3; }
10066   {
10067     #1
10068     \__fp_error:nfn { zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
10069     \fp_flag_on:n { division_by_zero }
10070     \exp_after:wN ##1
10071   }
10072   \exp_args:Nno \use:n
10073   { \cs_set:Npn \__fp_division_by_zero_o:NNnw ##1##2##3; ##4; }
10074   {
10075     #1
10076     \__fp_error:nfn { zero-div-ii }
10077     { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
10078     \fp_flag_on:n { division_by_zero }
10079     \exp_after:wN ##1
10080   }

```



```
10081 }
(End definition for \_fp_trap_division_by_zero_set_error: and others.)
```

_fp_trap_overflow_set_error: Just as for invalid operations and division by zero, the three different behaviours are obtained by feeding \prg_do_nothing:, \use_none:nnnnn or \use_none:nnnnnnn to an auxiliary, with a further auxiliary common to overflow and underflow functions. In most cases, the argument of the _fp_overflow:w and _fp_underflow:w functions will be an (almost) normal number (with an exponent outside the allowed range), and the error message thus displays that number together with the result to which it overflowed or underflowed. For extreme cases such as 10×10^{9999} , the exponent would be too large for T_EX, and _fp_overflow:w receives $\pm\infty$ (_fp_underflow:w would receive ± 0); then we cannot do better than simply say an overflow or underflow occurred.

```
10082 \cs_new_protected_nopar:Npn \_fp_trap_overflow_set_error:
10083 { \_fp_trap_overflow_set:N \prg_do_nothing: }
10084 \cs_new_protected_nopar:Npn \_fp_trap_overflow_set_flag:
10085 { \_fp_trap_overflow_set:N \use_none:nnnnn }
10086 \cs_new_protected_nopar:Npn \_fp_trap_overflow_set_none:
10087 { \_fp_trap_overflow_set:N \use_none:nnnnnnn }
10088 \cs_new_protected:Npn \_fp_trap_overflow_set:N #1
10089 { \_fp_trap_overflow_set:NnNn #1 { overflow } \_fp_inf_fp:N { inf } }
10090 \cs_new_protected_nopar:Npn \_fp_trap_underflow_set_error:
10091 { \_fp_trap_underflow_set:N \prg_do_nothing: }
10092 \cs_new_protected_nopar:Npn \_fp_trap_underflow_set_flag:
10093 { \_fp_trap_underflow_set:N \use_none:nnnnn }
10094 \cs_new_protected_nopar:Npn \_fp_trap_underflow_set_none:
10095 { \_fp_trap_underflow_set:N \use_none:nnnnnnn }
10096 \cs_new_protected:Npn \_fp_trap_underflow_set:N #1
10097 { \_fp_trap_overflow_set:NnNn #1 { underflow } \_fp_zero_fp:N { 0 } }
10098 \cs_new_protected:Npn \_fp_trap_overflow_set:NnNn #1#2#3#4
10099 {
10100   \exp_args:Nno \use:n
10101   { \cs_set:cpn { \_fp_ #2 :w } \s_fp \_fp_chk:w ##1##2##3; }
10102   {
10103     #1
10104     \_fp_error:nffn
10105     { flow \if_meaning:w 1 ##1 -to \fi: }
10106     { \fp_to_tl:n { \s_fp \_fp_chk:w ##1##2##3; } }
10107     { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
10108     {#2}
10109     \fp_flag_on:n {#2}
10110     #3 ##2
10111   }
10112 }
```

(End definition for _fp_trap_overflow_set_error: and others. These functions are documented on page 175.)

_fp_invalid_operation:nnw Initialize the two control sequences (to log properly their existence). Then set invalid operations to trigger an error, and division by zero, overflow, and underflow to act silently on their flag.

```
\_fp_invalid_operation_o:Nww
\_fp_invalid_operation_tl_o:nf
\_fp_division_by_zero_o:Nnw
\_fp_division_by_zero_o:NNww
\_fp_overflow:w
\_fp_underflow:w
```

```

10113 \cs_new:Npn \__fp_invalid_operation:nnw #1#2#3; { }
10114 \cs_new:Npn \__fp_invalid_operation_o:Nww #1#2; #3; { }
10115 \cs_new:Npn \__fp_invalid_operation_tl_o:nf #1 #2 { }
10116 \cs_new:Npn \__fp_division_by_zero_o:Nnw #1#2#3; { }
10117 \cs_new:Npn \__fp_division_by_zero_o:NNww #1#2#3; #4; { }
10118 \cs_new:Npn \__fp_overflow:w { }
10119 \cs_new:Npn \__fp_underflow:w { }
10120 \fp_trap:nn { invalid_operation } { error }
10121 \fp_trap:nn { division_by_zero } { flag }
10122 \fp_trap:nn { overflow } { flag }
10123 \fp_trap:nn { underflow } { flag }

```

(End definition for __fp_invalid_operation:nnw and others.)

__fp_invalid_operation_o:nw Convenient short-hands for returning \c_nan_fp for a unary or binary operation, and expanding after.

```

10124 \cs_new_nopar:Npn \__fp_invalid_operation_o:nw
10125 { \__fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } }

```

(End definition for __fp_invalid_operation_o:nw.)

23.3 Errors

__fp_error:nnnn
__fp_error:nnfn
__fp_error:nffn

```

10126 \cs_new:Npn \__fp_error:nnnn #1
10127 { \__msg_kernel_expandable_error:nnnnn { kernel } { fp - #1 } }
10128 \cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff }

```

(End definition for __fp_error:nnnn, __fp_error:nnfn, and __fp_error:nffn.)

23.4 Messages

Some messages.

```

10129 \__msg_kernel_new:nnnn { kernel } { unknown-fpu-exception }
10130 { The~FPU~exception~'#1'~is~not~known:~that~trap~will~never~be~triggered. }
10131 {
10132   The~only~exceptions~to~which~traps~can~be~attached~are \\\
10133   \iow_indent:n
10134   {
10135     * ~ invalid_operation \\\
10136     * ~ division_by_zero \\\
10137     * ~ overflow \\\
10138     * ~ underflow
10139   }
10140 }
10141 \__msg_kernel_new:nnnn { kernel } { unknown-fpu-trap-type }
10142 { The~FPU~trap~type~'#2'~is~not~known. }
10143 {
10144   The~trap~type~must~be~one~of \\\
10145   \iow_indent:n
10146   {

```

```

10147         * ~ error \\
10148         * ~ flag \\
10149         * ~ none
10150     }
10151 }
10152 \_msg_kernel_new:nnn { kernel } { fp-flow }
10153 { An ~ #3 ~ occurred. }
10154 \_msg_kernel_new:nnn { kernel } { fp-flow-to }
10155 { #1 ~ #3 ed ~ to ~ #2 . }
10156 \_msg_kernel_new:nnn { kernel } { fp-zero-div }
10157 { Division-by-zero-in~ #1 (#2) }
10158 \_msg_kernel_new:nnn { kernel } { fp-zero-div-ii }
10159 { Division-by-zero-in~ (#1) #3 (#2) }
10160 \_msg_kernel_new:nnn { kernel } { fp-invalid }
10161 { Invalid-operation~ #1 (#2) }
10162 \_msg_kernel_new:nnn { kernel } { fp-invalid-ii }
10163 { Invalid-operation~ (#1) #3 (#2) }
10164 </initex | package>

```

24 l3fp-round implementation

```

10165 <*initex | package>
10166 <@@=fp>

```

24.1 Rounding tools

Floating point operations often yield a result that cannot be exactly represented in a significand with 16 digits. In that case, we need to round the exact result to a representable number. The IEEE standard defines four rounding modes:

- Round to nearest: round to the representable floating point number whose absolute difference with the exact result is the smallest. If the exact result lies exactly at the mid-point between two consecutive representable floating point numbers, round to the floating point number whose last digit is even.
- Round towards negative infinity: round to the greatest floating point number not larger than the exact result.
- Round towards zero: round to a floating point number with the same sign as the exact result, with the largest absolute value not larger than the absolute value of the exact result.
- Round towards positive infinity: round to the least floating point number not smaller than the exact result.

This is not fully implemented in l3fp yet, and transcendental functions fall back on the “round to nearest” mode. All rounding for basic algebra is done through the functions defined in this module, which can be redefined to change their rounding behaviour (but there is not interface for that yet).

The rounding tools available in this module are many variations on a base function `__fp_round:NNN`, which expands to `\c_zero` or `\c_one` depending on whether the final result should be rounded up or down.

- `__fp_round:NNN` $\langle sign \rangle$ $\langle digit_1 \rangle$ $\langle digit_2 \rangle$ can expand to `\c_zero` or `\c_one`.
- `__fp_round_s:NNNw` $\langle sign \rangle$ $\langle digit_1 \rangle$ $\langle digit_2 \rangle$ $\langle more\ digits \rangle$; can expand to `\c_zero` ; or `\c_one` ;.
- `__fp_round_neg:NNN` $\langle sign \rangle$ $\langle digit_1 \rangle$ $\langle digit_2 \rangle$ can expand to `\c_zero` or `\c_one`.

See implementation comments for details on the syntax.

<code>__fp_round:NNN</code> <code>__fp_round_to_nearest:NNN</code> <code>__fp_round_to_ninf:NNN</code> <code>__fp_round_to_zero:NNN</code> <code>__fp_round_to_pinf:NNN</code>	<p>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 <code>\c_zero</code>, and otherwise to <code>\c_one</code>. Typically used within the scope of an <code>__int_eval:w</code>, to add 1 if needed, and thereby round correctly. The result depends on the rounding mode.</p>
---	--

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.

```

10167 \cs_new:Npn \__fp_round_return_one:
10168 { \exp_after:wN \c_one \tex_romannumeral:D }
10169 \cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
10170 {
10171   \if_meaning:w 2 #1
10172     \if_int_compare:w #3 > \c_zero
10173       \__fp_round_return_one:
10174     \fi:
10175   \fi:
10176   \c_zero
10177 }
10178 \cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { \c_zero }
10179 \cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
10180 {
10181   \if_meaning:w 0 #1
10182     \if_int_compare:w #3 > \c_zero
10183       \__fp_round_return_one:
10184     \fi:
10185   \fi:
10186   \c_zero
10187 }
10188 \cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3

```

```

10189 {
10190   \if_int_compare:w #3 > \c_five
10191   \__fp_round_return_one:
10192 \else:
10193   \if_meaning:w 5 #3
10194   \if_int_odd:w #2 \exp_stop_f:
10195   \__fp_round_return_one:
10196   \fi:
10197 \fi:
10198 \fi:
10199 \c_zero
10200 }
10201 \cs_new_eq:NN \__fp_round:NNN \__fp_round_to_nearest:NNN

```

(End definition for __fp_round:NNN. This function is documented on page ??.)

__fp_round_s:NNNw Similar to __fp_round:NNN, but with an extra semicolon, this function expands to \c_zero ; if rounding *⟨final sign⟩⟨digit⟩.⟨more digits⟩* to an integer truncates, and to \c_one ; otherwise. The *⟨more digits⟩* 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.

```

10202 \cs_new:Npn \__fp_round_s:NNNw #1 #2 #3 #4;
10203 {
10204   \exp_after:wN \__fp_round:NNN
10205   \exp_after:wN #1
10206   \exp_after:wN #2
10207   \int_use:N \__int_eval:w
10208   \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
10209   \if_meaning:w 5 #3 1 \fi:
10210   \exp_stop_f:
10211   \if_int_compare:w \__int_eval:w #4 > \c_zero
10212   1 +
10213   \fi:
10214 \fi:
10215 #3
10216 ;
10217 }

```

(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.

```

10218 \cs_new:Npn \__fp_round_digit:Nw #1 #2;
10219 {
10220   \if_int_odd:w \if_meaning:w 0 #1 \c_one \else:
10221   \if_meaning:w 5 #1 \c_one \else:
10222   \c_zero \fi: \fi:
10223   \if_int_compare:w \__int_eval:w #2 > \c_zero
10224   \__int_eval:w \c_one +

```

```

10225     \fi:
10226     \fi:
10227     #1
10228   }
(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.

```

10229 \cs_new:Npn \_fp_round_to_ninf_neg:NNN #1 #2 #3
10230 {
10231   \if_meaning:w 0 #1
10232     \if_int_compare:w #3 > \c_zero
10233       \_fp_round_return_one:
10234     \fi:
10235   \fi:
10236   \c_zero
10237 }
10238 \cs_new:Npn \_fp_round_to_zero_neg:NNN #1 #2 #3
10239 {
10240   \if_int_compare:w #3 > \c_zero
10241     \_fp_round_return_one:
10242   \fi:
10243   \c_zero
10244 }
10245 \cs_new:Npn \_fp_round_to_pinf_neg:NNN #1 #2 #3
10246 {
10247   \if_meaning:w 2 #1
10248     \if_int_compare:w #3 > \c_zero
10249       \_fp_round_return_one:
10250     \fi:
10251   \fi:
10252   \c_zero
10253 }
10254 \cs_new_eq:NN \_fp_round_to_nearest_neg:NNN \_fp_round_to_nearest:NNN
10255 \cs_new_eq:NN \_fp_round_neg:NNN \_fp_round_to_nearest_neg:NNN
(End definition for \_fp_round_neg:NNN. This function is documented on page ??.)

```

24.2 The round function

```

    \_fp_round:Nww
    \_fp_round:Nwn
10256 \cs_new:Npn \_fp_round:Nww #1#2 ; #3 ;
    \_fp_round_normal:NwNNnw
10257 {
    \_fp_round_normal:NnnwNNnn
10258   \_fp_small_int:wTF #3; { \_fp_round:Nwn #1#2; }
    \_fp_round_pack:Nw
    \_fp_round_normal:NNwNnn
    \_fp_round_normal_end:wwNnn
    \_fp_round_special:NwNnn
    \_fp_round_special_aux:Nw

```

```

10259     {
10260         \__fp_invalid_operation_tl_o:nf
10261         { round } { \__fp_array_to_clist:n { #2; #3; } }
10262     }
10263 }
10264 \cs_new:Npn \__fp_round:Nwn #1 \s__fp \__fp_chk:w #2#3#4; #5
10265 {
10266     \if_meaning:w 1 #2
10267         \exp_after:wN \__fp_round_normal:NwNNnw
10268         \exp_after:wN #1
10269         \__int_value:w #5
10270     \else:
10271         \exp_after:wN \__fp_exp_after_o:w
10272     \fi:
10273     \s__fp \__fp_chk:w #2#3#4;
10274 }
10275 \cs_new:Npn \__fp_round_normal:NwNNnw #1#2 \s__fp \__fp_chk:w 1#3#4#5;
10276 {
10277     \__fp_decimate:nNnnnn { \c_sixteen - #4 - #2 }
10278     \__fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
10279 }
10280 \cs_new:Npn \__fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
10281 {
10282     \exp_after:wN \__fp_round_normal:NNwNnn
10283     \int_use:N \__int_eval:w
10284     \if_int_compare:w #2 > \c_zero
10285         1 \__int_value:w #2
10286         \exp_after:wN \__fp_round_pack:Nw
10287         \int_use:N \__int_eval:w 1#3 +
10288     \else:
10289         \if_int_compare:w #3 > \c_zero
10290             1 \__int_value:w #3 +
10291         \fi:
10292     \fi:
10293     \exp_after:wN #5
10294     \exp_after:wN #6
10295     \use_none:nnnnnnn #3
10296     #1
10297     \__int_eval_end:
10298     0000 0000 0000 0000 ; #6
10299 }
10300 \cs_new:Npn \__fp_round_pack:Nw #1
10301 { \if_meaning:w 2 #1 + \c_one \fi: \__int_eval_end: }
10302 \cs_new:Npn \__fp_round_normal:NNwNnn #1 #2
10303 {
10304     \if_meaning:w 0 #2
10305         \exp_after:wN \__fp_round_special:NwNnn
10306         \exp_after:wN #1
10307     \fi:
10308     \__fp_pack_twice_four:wNNNNNNNN

```

```

10309     \_fp_pack_twice_four:wNNNNNNNN
10310     \_fp_round_normal_end:wwNnn
10311     ; #2
10312 }
10313 \cs_new:Npn \_fp_round_normal_end:wwNnn #1;#2;#3#4#5
10314 {
10315     \exp_after:wN \_fp_exp_after_o:w \tex_romannumeral:D -'0
10316     \_fp_sanitize:Nw #3 #4 ; #1 ;
10317 }
10318 \cs_new:Npn \_fp_round_special:NwwNnn #1#2;#3;#4#5#6
10319 {
10320     \if_meaning:w 0 #1
10321     \_fp_case_return:nw
10322     { \exp_after:wN \_fp_zero_fp:N \exp_after:wN #4 }
10323     \else:
10324     \exp_after:wN \_fp_round_special_aux:Nw
10325     \exp_after:wN #4
10326     \int_use:N \_int_eval:w \c_one
10327     \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
10328     \fi:
10329     ;
10330 }
10331 \cs_new:Npn \_fp_round_special_aux:Nw #1#2;
10332 {
10333     \exp_after:wN \_fp_exp_after_o:w \tex_romannumeral:D -'0
10334     \_fp_sanitize:Nw #1#2; {1000}{0000}{0000}{0000};
10335 }

```

(End definition for `_fp_round:Nww` and `_fp_round:Nwn`. These functions are documented on page ??.)

```

10336 </initex | package>

```

25 l3fp-parse implementation

```

10337 <*initex | package>
10338 <@@=fp>

```

26 Precedences

In order of evaluation (some distinctions are irrelevant for the order of evaluation, but serve as signals).

- 32 Juxtaposition for implicit multiplication.
- 16 Function calls with multiple arguments.
- 15 Function calls expecting exactly one argument.
- 14 Binary `**` and `^` (right to left).

- 12 Unary +, -, ! (right to left).
- 10 Binary *, / and %.
- 9 Binary + and -.
- 7 Comparisons.
- 5 Logical **and**, denoted by &&.
- 4 Logical **or**, denoted by ||.
- 3 Ternary operator ?:, piece ?.
- 2 Ternary operator ?:, piece :.
- 1 Commas, and parentheses accepting commas.
- 0 Parentheses expecting exactly one argument.
- 1 Start and end of the expression.

27 Evaluating an expression

`__fp_parse:n` This **f**-expands to the internal floating point number obtained by evaluating the *floating point expression*. During this 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 TeX errors.⁷

(End definition for `__fp_parse:n`.)

28 Work plan

The task at hand is non-trivial, and some previous failed attempts have shown me that the code ends up giving unreadable logs, so we'd better get it (almost) right the first time. Let us thus first discuss precisely the design before starting to write the code. To simplify matters, we first consider expressions with integers only.

28.1 Storing results

The main issue in parsing expressions expandably is: “where in the input stream should the result be put?”

One option is to place the result at the end of the expression, but this has several drawbacks:

⁷Bruno: describe what happens in cases like `2\c_three = 6`.

- firstly it means that for long expressions we would be reaching all the way to the end of the expression at every step of the calculation, which can be rather expensive;
- secondly, when parsing parenthesized sub-expressions, we would naturally place the result after the corresponding closing parenthesis. But since `__fp_parse:n` does not assume that its argument is expanded, this closing parenthesis may be hidden in a macro, and not present yet, causing havoc.

The other natural option is to store the result at the start of the expression, and carry it as an argument of each macro. This does not really work either: in order to expand what follows on the input stream, we need to skip at each step over all the tokens in the result using `\exp_after:wN`. But this requires adding many `\exp_after:wN` to the result at each step, also an expensive process.

Hence, we need to go for some fine expansion control: the result is stored *before* the start... A toy model that illustrates this idea is to try and add some positive integers which may be hidden within macros, or registers. Assume that one number 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 -'0 \clean:w <stuff>
```

Hitting this construction by one step of expansion expands `\exp_after:wN`, which triggers the primitive `__int_value:w`, which reads an integer, 12345. This integer is unfinished, causing the second `\exp_after:wN` to expand, and trigger the construction `\tex_romannumeral:D -'0`, which f-expands `\clean:w` (see `l3expan.dtx` for an explanation). Assume then that `\clean:w` is such that it expands `<stuff>` to *e.g.*, 333444;. Once `\clean:w` is done expanding, we will obtain essentially

```
\exp_after:wN \add:ww \__int_value:w 12345 ; 333444 ;
```

where in fact `\exp_after:wN` has already been expanded, and `__int_value:w` has already seen 12345. Now, `__int_value:w` sees the `;`, and stops expanding, and we are left with

```
\add:ww 12345 ; 333444 ;
```

which can safely perform the addition by grabbing two arguments delimited by `;`.

On this toy example, we could note that 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 to waste as little as possible of this precious memory.

28.2 Precedence

A major point to keep in mind when parsing expressions is that different operators have different precedence. The true analog of our toy `\clean:w` macro must thus take care of that. For definiteness, let us assume that the operation which prompted `\clean:w` was a multiplication. Then `\clean:w` (expand and) read digits until the number is ended by some operation. If this is `+` or `-`, then the multiplication should be calculated next, so `\clean:w` can simply decide that its job is done. However, if the operator we find is `^`, then this operation must be performed before returning control to the multiplication. This means that we need to `\clean:w` the number following `^`, and perform the calculation, then just end our job.

Hence, each time a number is cleaned, the precedence of the following operation must be compared to that of the previous operation. The process of course has to happen recursively. For instance, $1+2^3*4$ would involve the following steps.

- 1 is cleaned up.
- 2 is cleaned up.
- The precedences of `+` and `^` are compared. Since the latter is higher, the second operand of `^` should be cleaned.
- 3 is cleaned up.
- The precedences of `^` and `*` are compared. Since the former is higher, the cleaning step stops.
- Compute $2^3 = 8$.
- We now have $1+8*4$, and the operation `+` is still looking for a second operand. Clean 8.
- The precedences of `+` and `*` are compared. Since the latter is higher, the second operand of `*` should be cleaned.
- 4 is cleaned up, and the end of the expression is reached.
- Compute $8*4 = 32$.
- We now have $1+8*4$, and the operation `+` is still looking for a second operand. Clean 32, and reach the end of the expression.
- Compute $1+32 = 33$.

Here, there is some (expensive) redundant work: the results of computations should not need to be cleaned again. Thus the true definition is slightly more elaborate.

The precedence of `(` and `)` are defined to be equal, and smaller than the precedence of `+` and `-`, itself smaller than `*` and `/`, smaller, finally, than the power operator `**` (or `^`).

28.3 Infix operators

The implementation that was chosen is slightly wasteful: it causes more nesting than necessary. However, it is simpler to implement and to explain than a slightly optimized variant.

The cornerstone of that method is a pair of functions, `\until` and `\one`, which both take as their first argument the precedence (an integer) of the last operation. The f-expansion of

```
\until <prec> \one <prec> <stuff>
```

is the internal floating point obtained by “cleaning” numbers which follow in the input stream, and performing computations until reaching an operation with a precedence less than or equal to `<prec>`. This is followed by a control sequence of the form `\infix_?`, namely,

```
<floating point> \infix_?
```

where `?` is the operation following that number in the input stream (we thus know that this operation has at most the precedence `<prec>`, otherwise it would have been performed already).

How is that expansion achieved? First, `\one <prec>` reads one `<floating point>` number, and converts it to an internal form, then the following operation, say `*`, is packed in the form `\infix_*`, which is fed the `<prec>`. This function (one per infix operator) compares `<prec>` with the precedence of the operator we just read (here `*`). If `<prec>` is higher, our job is finished, and `\one` leaves `__fp_parse_stop_until:N` so that `\until` knows to stop. Otherwise, `\infix_*` triggers a new pair `\until <prec(*)> \one <prec(*)>`, which produces the second operand `<floating point2 for the multiplication:`

```
\until <prec> <floating point>
... <floating point2

```

The dots are `__fp_parse_apply_binary:NwNwN *`. The boolean tells `\until` that it is not done, and it expands (essentially) to

```
\until <prec> \__fp*_o:ww <floating point> <floating point2

```

making `TEX` expand `__fp*_o:ww` before `\until`. As implemented in `l3fp-basics`, this operation expands what follows its result exactly once. This triggers `\tex_romannumeral:D`, which fully expands `\infix_? <prec>`. This compares the precedence of the next operation, `?`, and `<prec>`, and leaves a boolean (and possibly more things), which is then checked by `\until <prec>` to know if the result of the multiplication is the end of the story, or if `?` should be computed as well before `\until <prec>` ends.

This should be easier to see on an example. To each infix operator, for instance, `*`, is associated the following data:

- a test function, `\infix_*`, which conditionally continues the calculation or waits to be hit again by expansion;

- a function `*` (notation for `_fp_o:ww`) which performs the actual calculation;
- an integer, `*`, which encodes the precedence of the operator.

The token that is currently being expanded is underlined, and in red. Tokens that have not yet been read (and could still be hidden in macros) are in gray.

In a first reading, the distinction between the *precedence* `+`, the operation `+`, and the character token `+` should not matter. It is only required to accommodate for multi-token infix operators such as `**`: indeed, when controlling expansion, we need to skip over those tokens using `\exp_after:wN`, and this only skips one token. Thus `**` needs to be replaced by a single token (either its precedence or its calculating function, depending on the place).

To end the computation cleanly, we add a trailing right parenthesis, and give `(` and `)` the lowest precedence, so that `\until(\one(` reads numbers and performs operations until meeting a right parenthesis. This is discussed more precisely in the next section.

```

\until( \one( 11 + 2**3 * 5 - 9 )
\until( 1 \one( 1 + 2**3 * 5 - 9 )
\until( 11 \one( + 2**3 * 5 - 9 )
\until( 11; \infix_+( 2**3 * 5 - 9 )
\until( 11; F + \until+ \one+ 2**3 * 5 - 9 )
\until( 11; F + \until+ 2 \one+ **3 * 5 - 9 )
\until( 11; F + \until+ 2; \infix_** 3 * 5 - 9 )
\until( 11; F + \until+ 2; F ** \until** \one** 3 * 5 - 9 )
\until( 11; F + \until+ 2; F ** \until** 3 \one** * 5 - 9 )
\until( 11; F + \until+ 2; F ** \until** 3; \infix_*** 5 - 9 )
\until( 11; F + \until+ 2; F ** \until** 3; T \infix_* 5 - 9 )
\until( 11; F + \until+ 2; F ** 3; \infix_* 5 - 9 )
\until( 11; F + \until+ ** 2; 3; \infix_** 5 - 9 )
\until( 11; F + \until+ 8; \infix_*+ 5 - 9 )
\until( 11; F + \until+ 8; F * \until* \one* 5 - 9 )
\until( 11; F + \until+ 8; F * \until* 5 \one* - 9 )
\until( 11; F + \until+ 8; F * \until* 5; \infix_-* 9 )
\until( 11; F + \until+ 8; F * \until* 5; T \infix_- 9 )
\until( 11; F + \until+ 8; F * 5; \infix_- 9 )
\until( 11; F + \until+ * 8; 5; \infix_-+ 9 )
\until( 11; F + \until+ 40; \infix_-+ 9 )
\until( 11; F + \until+ 40; T \infix_- 9 )
\until( 11; F + 40; \infix_- 9 )

```

```

\until( + 11; 40; \infix_-( 9 )
\until( 51; \infix_-( 9 )
\until( 51; F - \until- \one- 9 )
\until( 51; F - \until- 9 \one- )
\until( 51; F - \until- 9; \infix_-
\until( 51; F - \until- 9; T \infix_)
\until( 51; F - 9; \infix_)
\until( - 51; 9; \infix_)(
\until( 42; \infix_)(
\until( 42; T \infix_)
42; \infix_)

```

The only missing step is to clean the output by removing `\infix_)`, and possibly checking that nothing else remains.

28.4 Prefix operators, parentheses, and functions

Prefix operators (typically the 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 small subtlety on precedence explained below. Once that argument is found, its sign can be flipped. A left parenthesis is just a prefix operator which removes the closing parenthesis (with some extra checks).

Detecting prefix operators is done by `\one`. Before looking for a number, it tests the first character. If it is a digit, a dot, or a register, then we have a number. Otherwise, it is put in a function, `\prefix_?` (where `?` is roughly that first character), which is expanded. For instance, with a left parenthesis we would have the following.

```

\one* ( 2 + 3 )
\prefix_(* 2 + 3 )
(* \until( \one( 2 + 3 )
...
(* 5; \infix_)

```

As usual, the `\until`–`\one` pair reads and compute until reaching an operator of precedence at most `(`. Then `(` removes `\infix_)` and looks ahead for the next operation, comparing its precedence with the precedence `*` of the previous operation (in fact, this comparison is done by the relevant `\infix_?` built from the next operation).

To support multi-character function (and constant) names, we may need to put more than one character in the `\prefix_?` construction. See implementation for details.

Note that contrarily to `\infix_?` functions, the `\prefix_?` functions perform no test on their argument (which is once more the previous precedence), since we know that we need a number, and must never stop there.

Functions are implemented as prefix operators with infinitely high precedence, so that their argument is the first number that can possibly be built. For instance, something like the following could happen in a computation

```
\one* sqrt 4 + 3 )
\prefix_sqrt* 4 + 3 )
sqrt* \until∞ \one∞ 4 + 3 )
...
sqrt* 4; \infix_+ 3 )
2; \infix_+* 3 )
```

Lonely example, to be put somewhere: $2+\sin 1 * 3$ is $2 + (\sin(1) \times 3)$.

A further complication arises in the case of the unary $-$ sign: $-3**2$ should be $-(3^2) = -9$, and not $(-3)^2 = 9$. Easy, just give $-$ a lower precedence, equal to that of the infix $+$ and $-$. Unfortunately, this fails in subtle cases such as $3**-2*4$, yielding $3^{-2 \times 4}$ instead of the correct $3^{-2} \times 4$. In fact, a unary $-$ should only perform operations whose precedence is greater than that of the last operation, as well as $-$.⁸ Thus, `\prefix_<prec>` expands to something like

```
- <prec> \until? \one ?
```

where `?` is the maximum of `<prec>` and the precedence of $-$. Once the argument of $-$ is found, $-$ gets its opposite, and leaves it for the previous operation to use.

An example with parentheses.

```
\until( \one( 11 * ( 2 + 3 ) - 9 )
\until( 1 \one( 1 * ( 2 + 3 ) - 9 )
\until( 11 \one( * ( 2 + 3 ) - 9 )
\until( 11; \infix_*( ( 2 + 3 ) - 9 )
\until( 11; F * \until* \one* ( 2 + 3 ) - 9 )
\until( 11; F * \until* \prefix_(* 2 + 3 ) - 9 )
\until( 11; F * \until* (* \until( \one( 2 + 3 ) - 9 )
\until( 11; F * \until* (* \until( 2 \one( + 3 ) - 9 )
\until( 11; F * \until* (* \until( 2; \infix_+( 3 ) - 9 )
\until( 11; F * \until* (* \until( 2; F + \until+ \one+ 3)-9)
\until( 11; F * \until* (* \until( 2; F + \until+ 3 \one+ )-9)
\until( 11; F * \until* (* \until( 2; F + \until+ 3; \infix_)+ -9)
\until( 11; F * \until* (* \until( 2; F + \until+ 3; T \infix_) -9)
\until( 11; F * \until* (* \until( 2; F + 3; \infix_) - 9 )
```

⁸Taking into account the precedence of $-$ itself only matters when it follows a left parenthesis: $(-2*4+3)$ should give $((-8)+3)$, not $(-(8+3))$.

```

\until( 11; F * \until* (* \until( + 2; 3; \infix_)( - 9 )
\until( 11; F * \until* (* \until( 5; \infix_)( - 9 )
\until( 11; F * \until* (* \until( 5; T \infix_)( - 9 )
\until( 11; F * \until* (* 5; \infix_)( - 9 )
\until( 11; F * \until* 5; \infix_- 9 )
\until( 11; F * \until* 5; T \infix_- 9 )
\until( 11; F * 5; \infix_- 9 )
\until( _ 11; 5; \infix_-( 9 )
\until( 55; \infix_-( 9 )
\until( 55; F - \until- \one- 9 )
\until( 55; F - \until- 9 \one- )
\until( 55; F - \until- 9; \infix_- )
\until( 55; F - \until- 9; T \infix_- )
\until( 55; F - 9; \infix_- )
\until( _ 55; 9; \infix_- )
\until( 47; \infix_- )
\until( 47; T \infix_- )
47; \infix_- )

```

The end of this (sub)section was not revised yet

- If it is a sign (- or +), then any following sign will be combined with this initial sign, forming `\prefix_+` or `\prefix_-`.
- If it is a letter, then any following letter is grabbed, forming for instance `\prefix_-sin` or `\prefix_-sinh`.
- Otherwise, only one token⁹ is grabbed, for instance `\prefix_()`.

Functions may take several arguments, possibly an unknown number¹⁰, for instance `round(1.23456,2)`.

- `round` is made into `\prefix_round`, which tries to grab one number using `\one`.
- This builds `\prefix_()`, which uses `\one` to grab one number, calculating as necessary. The comma is given the same precedence as parentheses, and thus ends the calculation of the argument of `round`.
- `round` now has its first argument. It can check whether the argument was closed by `,` or `)`, and branch accordingly.

⁹Some support for multi-character prefix operator may be added in the future, but right now, I don't see a use for it. Perhaps, for including comments inside the computation itself??

¹⁰Keyword argument support may be added later.

- If it was a comma, then the first argument is skipped over, through an expensive set of `\exp_after:wN`, and the second argument can be grabbed. Here it is simply an integer, easier to parse by building upon `\etex_numexpr:D`.
- The closing parenthesis (or another comma) is seen, and the control is given back to `\prefix_round`.

28.5 Type detection

The type of data should be detected by reading the first few tokens, before calling a type-specific function to parse it. Or should the type be obtained after the semicolon which indicates the end of the thing? And placed there?

Also to grab exponents correctly, build `__fp_<abc>:w` when seeing some non-numeric `abc` while still looking to complete a number (or other data). Then, if `__fp_postfix_<type>_<abc>:w` exists, use it.

The internal representation of floating point numbers is quite untypable, and we provide here the tools to convert from a more user-friendly representation to internal floating point numbers, and for various other conversions. Every floating point operation calls those functions to normalize the input, so they must be optimized.

29 Internal representation

Internally, a floating point number $\langle X \rangle$ is a token list containing

```
\s__fp \__fp_chk:w <case> <sign> <body> ;
```

Let us explain each piece separately.

Internal floating point numbers will be used in expressions, and in this context will be subject to f-expansion. They must leave a recognizable mark after f-expansion, to prevent the floating point number from being re-parsed. Thus, `\s__fp` is simply another name for `\relax`.

Since floating point numbers are always accessed by the various operations using f-expansion, we can safely let them be protected: x-expansion will then leave them untouched. However, when used directly without an accessor function, floating points should produce an error. `\s__fp` will do nothing, and `__fp_chk:w` produces an error.

The (decimal part of the) IEEE-754-2008 standard requires the format to be able to represent special floating point numbers besides the usual positive and negative cases. The various possibilities will be distinguished by their $\langle case \rangle$, which is a single digit:¹¹

- 0 zeros: `+0` and `-0`,
- 1 “normal” numbers (positive and negative),
- 2 infinities: `+inf` and `-inf`,
- 3 quiet and signalling `nan`.

¹¹Bruno: 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.

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

$\backslash s_fp \backslash_fp_chk:w \langle case \rangle \langle sign \rangle \backslash s_fp... ;$

where $\backslash s_fp...$ 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

$\backslash s_fp \backslash_fp_chk:w 1 \langle sign \rangle \{\langle exponent \rangle\} \{\langle X_1 \rangle\} \{\langle X_2 \rangle\} \{\langle X_3 \rangle\} \{\langle X_4 \rangle\} ;$

Here, the $\langle exponent \rangle$ is an integer, at most $\backslash c_fp_max_exponent_int = 10000$ 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 $1000 \leq \langle X_1 \rangle \leq 9999$.

30 Internal parsing functions

$\backslash_fp_parse_until:Nw$ Reads the $\langle tokens \rangle$, performing every computation with a precedence higher than $\langle precedence \rangle$, then expands to where the $\langle op \rangle$ is the first operation with a lower precedence, possibly **end**.
(End definition for $\backslash_fp_parse_until:Nw$.)

$\backslash_fp_parse_operand:Nw$ If the following $\langle operation \rangle$ has a precedence higher than $\langle precedence \rangle$, expands to and otherwise expands to
(End definition for $\backslash_fp_parse_operand:Nw$.)

$\backslash_fp_parse_infix_meta\{operation\}:N$ If the $\langle op \rangle$ has a precedence higher than $\langle precedence \rangle$, expands to Otherwise expands to
(End definition for $\backslash_fp_parse_infix_metaoperation:N$.)

30.1 Expansion control

At each step in reading a floating point expression, we wish to perform `f`-expansion. Normally, spaces stop this `f`-expansion. This can be problematic: for instance, the macro `\X` below will not be expanded if we simply do `f`-expansion.

```
\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }
```

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.

Floating point expressions should behave as much as possible like ϵ -TeX-based integer expressions and dimension expressions. In particular, full-expansion should be performed as the expression is read, token by token, forcing the expansion of protected macros, and ignoring spaces.

Full expansion can be done with `\tex_romannumeral:D -'0`. Unfortunately, this expansion is stopped by spaces. Thus using simply this will fail on `\fp_eval:n { 1 + ~ \l_tmpa_fp }` since the floating point variable will not be expanded. 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. We can avoid being stopped by such explicit space characters (and by some braces) if we add `\use:n` after `-'0`.

Testing if a character token `#1` is a digit can be done using

```
\if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
  true code
\else:
  false code
\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.

`__fp_parse_expand:w` This function must always come within a `\romannumeral` expansion. The *tokens* should be the part of the expression that we have not yet read. This requires in particular closing all conditionals properly before expanding.

```
10339 \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.

```
10340 \cs_new:Npn \__fp_parse_return_semicolon:w
10341   #1 \fi: \__fp_parse_expand:w { \fi: ; #1 }
(End definition for \__fp_parse_return_semicolon:w.)
```

30.2 Fp object type

`__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 `_?`.

```

10342 \group_begin:
10343 \char_set_catcode_other:N \S
10344 \char_set_catcode_other:N \F
10345 \char_set_catcode_other:N \P
10346 \char_set_lccode:nn { '\- } { '\_ }
10347 \tl_to_lowercase:n
10348 {
10349   \group_end:
10350   \cs_new:Npn \__fp_type_from_scan:N #1
10351   {
10352     \exp_after:wN \__fp_type_from_scan:w
10353     \token_to_str:N #1 \q_mark S--FP-? \q_mark \q_stop
10354   }
10355   \cs_new:Npn \__fp_type_from_scan:w #1 S--FP #2 \q_mark #3 \q_stop {#2}
10356 }

```

(End definition for `__fp_type_from_scan:N` and `__fp_type_from_scan:w`.)

30.3 Reading digits

`__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` $\langle digits \rangle ; \langle filling\ 0 \rangle ; \langle length \rangle$

`__fp_parse_digits_iv:N` 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. `__fp_parse_digits_iii:N` 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.

```

10357 \cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
10358 {
10359   \cs_new:cpn { __fp_parse_digits_ #1 :N } ##1
10360   {
10361     \if_int_compare:w \c_nine < 1 \token_to_str:N ##1 \exp_stop_f:
10362     \token_to_str:N ##1 \exp_after:wN #2 \tex_romannumeral:D
10363   \else:
10364     \__fp_parse_return_semicolon:w #3 ##1
10365   \fi:
10366   \__fp_parse_expand:w
10367 }
10368 }
10369 \__fp_tmp:w {vii} \__fp_parse_digits_vi:N { 0000000 ; 7 }

```

```

10370 \__fp_tmp:w {vi}    \__fp_parse_digits_v:N    { 000000 ; 6 }
10371 \__fp_tmp:w {v}     \__fp_parse_digits_iv:N   { 00000 ; 5 }
10372 \__fp_tmp:w {iv}    \__fp_parse_digits_iii:N  { 0000 ; 4 }
10373 \__fp_tmp:w {iii}   \__fp_parse_digits_ii:N   { 000 ; 3 }
10374 \__fp_tmp:w {ii}    \__fp_parse_digits_i:N    { 00 ; 2 }
10375 \__fp_tmp:w {i}     \__fp_parse_digits_:N     { 0 ; 1 }
10376 \cs_new_nopar:Npn \__fp_parse_digits_:N { ; ; 0 }
(End definition for \__fp_parse_digits_vii:N and others.)

```

30.4 Parsing one operand

At the start of an expression, or just following a binary operation or a function call, we are looking for an operand. This can be an explicit floating point number, a floating point variable, a \TeX register, a function call such as `sin(3)`, a parenthesized expression, *etc.* We distinguish the various cases by their first token after `f`-expansion:

- `\tex_relax:D` in some form. That can be an internal floating point, a premature end, or an uninitialized register.
- A register. We interpret this as the significand of a floating point number. This is subtly different from unpacking it, for instance, `\c_minus_one**2` gives 1, while `-1**2` gives -1 .
- A digit, or a dot. That marks the start of the significand for a floating point number.
- A letter (lower or upper-case), which starts an identifier, either a constant or a function (possibly unknown).
- `+`, `-`, or `!`, unary operators, which resume looking for a floating point number before acting on it.
- `(`, which makes us parse a subexpression until the matching `)`.
- Other characters such as `'` or `"` may be given a meaning later. Characters such as `*` or `/` have a meaning as infix operators but are not valid when we are looking for an operand: for instance, `3+*4` is not valid.

A category code test separates the first two cases from the others, and they are further distinguished with a meaning test. We then single out digits. Letters are detected using their character code. All other characters are taken care of by building a `csname` from that character and using it to continue parsing. Unknown characters lead to an error.

`__fp_parse_operand:Nw` Function called `\one` at other places. It grabs one operand, and packs the symbol that follows in an `\infix_ csname`. `#1` is the previous *precedence*, and `#2` the first character of the operand (already `f`-expanded).

```

10377 \cs_new:Npn \__fp_parse_operand:Nw #1 #2
10378 {
10379   \if_catcode:w \tex_relax:D #2

```

```

10380 \if_meaning:w \tex_relax:D #2
10381 \exp_after:wN \exp_after:wN
10382 \exp_after:wN \__fp_parse_operand_relax:NN
10383 \else:
10384 \exp_after:wN \exp_after:wN
10385 \exp_after:wN \__fp_parse_operand_register:NN
10386 \fi:
10387 \else:
10388 \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
10389 \exp_after:wN \exp_after:wN
10390 \exp_after:wN \__fp_parse_operand_digit:NN
10391 \else:
10392 \exp_after:wN \exp_after:wN
10393 \exp_after:wN \__fp_parse_operand_other:NN
10394 \fi:
10395 \fi:
10396 #1 #2
10397 }

```

(End definition for __fp_parse_operand:Nw.)

__fp_parse_operand_register:NN
__fp_parse_operand_register_aux:www

Find the exponent following the register #2, then combine the value of #2 (mapping 1pt to 1) with the exponent to produce a floating point number.

```

10398 \group_begin:
10399 \char_set_catcode_other:N \P
10400 \char_set_catcode_other:N \T
10401 \tl_to_lowercase:n
10402 {
10403 \group_end:
10404 \cs_new:Npn \__fp_parse_operand_register:NN #1#2
10405 {
10406 \exp_after:wN \__fp_parse_infix_after_operand:NwN
10407 \exp_after:wN #1
10408 \tex_romannumeral:D -'0
10409 \exp_after:wN \__fp_parse_operand_register_aux:www
10410 \tex_the:D
10411 \exp_after:wN #2
10412 \exp_after:wN P
10413 \exp_after:wN T
10414 \exp_after:wN \q_stop
10415 \__int_value:w \__fp_parse_exponent:N
10416 }
10417 \cs_new:Npn \__fp_parse_operand_register_aux:www #1 PT #2 \q_stop #3 ;
10418 { \__fp_parse:n { #1 e #3 } }
10419 }

```

(End definition for __fp_parse_operand_register:NN and __fp_parse_operand_register_aux:www.)

__fp_parse_operand_relax:NN
__fp_parse_exp_after_f:nw
__fp_parse_exp_after_mark_f:nw
__fp_parse_exp_after_?_f:nw

The second argument is a control sequence equal to \tex_relax:D. There are three cases, dispatched using __fp_type_from_scan:N.

- `\s__fp` starts a floating point number, and we call `__fp_parse_exp_after_f:nw`, which `f`-expands after the floating point.
- `\s__fp_mark` is a premature end, we call `__fp_parse_exp_after_mark_f:nw`, which triggers the appropriate error.
- For a control sequence not containing `\s__fp`, we call `__fp_parse_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_parse_exp_after_⟨type⟩_f:nw`. In all cases, we make sure that the last argument of `__fp_parse_infix:NN` is correctly expanded.

```

10420 \cs_new:Npn \__fp_parse_operand_relax:NN #1#2
10421 {
10422   \cs:w __fp_parse_exp_after \__fp_type_from_scan:N #2 _f:nw \cs_end:
10423   {
10424     \exp_after:wN \__fp_parse_infix:NN
10425     \exp_after:wN #1 \tex_romannumeral:D \__fp_parse_expand:w
10426   }
10427   #2
10428 }
10429 \cs_new_eq:NN \__fp_parse_exp_after_f:nw \__fp_exp_after_f:nw
10430 \cs_new:Npn \__fp_parse_exp_after_mark_f:nw #1
10431 {
10432   \_msg_kernel_expandable_error:nn { kernel } { fp-early-end }
10433   \exp_after:wN \c_nan_fp
10434   \tex_romannumeral:D -'0 #1
10435 }
10436 \cs_new:cpn { __fp_parse_exp_after_?_f:nw } #1#2
10437 {
10438   \_msg_kernel_expandable_error:nnn
10439   { kernel } { bad-variable } {#2}
10440   \exp_after:wN \c_nan_fp
10441   \tex_romannumeral:D -'0 #1
10442 }

```

(End definition for `__fp_parse_operand_relax:NN` and others.)

`__fp_parse_operand_other:NN`

The interesting bit is `__fp_parse_operand_other:NN`. It separates letters from non-letters and builds the appropriate `\prefix` function. If it is not defined (is `\tex_relax:D`), make it a signalling nan. We don't look for an argument, as the unknown “prefix” can also be a (mistyped) constant such as `Inf`.

```

10443 \cs_new:Npn \__fp_parse_operand_other:NN #1 #2
10444 {
10445   \if_int_compare:w
10446     \__int_eval:w \tex_uccode:D '2 / 26 = \c_three
10447     \exp_after:wN \__fp_parse_operand_other_word_aux:Nw
10448     \exp_after:wN #1
10449     \tex_romannumeral:D

```

```

10450         \exp_after:wN \__fp_parse_letters:NN
10451         \exp_after:wN #2
10452         \tex_romannumeral:D
10453     \else:
10454         \exp_after:wN \__fp_parse_operand_other_prefix_aux:NNN
10455         \exp_after:wN #1
10456         \exp_after:wN #2
10457         \cs:w \__fp_parse_prefix_#2:Nw \exp_after:wN \cs_end:
10458         \tex_romannumeral:D
10459     \fi:
10460     \__fp_parse_expand:w
10461 }
10462
10463 \cs_new:Npn \__fp_parse_letters:NN #1#2
10464 {
10465     \exp_after:wN \c_zero
10466     \exp_after:wN #1
10467     \tex_romannumeral:D
10468     \if_int_compare:w
10469         \if_catcode:w \tex_relax:D #2
10470         \c_zero
10471     \else:
10472         \__int_eval:w \tex_uccode:D '#2 / 26
10473     \fi:
10474     = \c_three
10475     \exp_after:wN \__fp_parse_letters:NN
10476     \exp_after:wN #2
10477     \tex_romannumeral:D
10478     \exp_after:wN \__fp_parse_expand:w
10479 \else:
10480     \exp_after:wN \c_zero
10481     \exp_after:wN ;
10482     \exp_after:wN #2
10483 \fi:
10484 }
10485 \cs_new:Npn \__fp_parse_operand_other_word_aux:Nw #1 #2;
10486 {
10487     \cs_if_exist_use:cF { \__fp_parse_word_#2:N }
10488     {
10489         \__msg_kernel_expandable_error:nnn
10490         { kernel } { unknown-fp-word } {#2}
10491         \exp_after:wN \c_nan_fp
10492         \tex_romannumeral:D -'0
10493         \__fp_parse_infix:NN
10494     }
10495     #1
10496 }
10497 \cs_new_eq:NN \s__fp_unknown \tex_relax:D
10498 \cs_new:Npn \__fp_parse_operand_other_prefix_aux:NNN #1#2#3
10499 {

```



```

10500 \if_meaning:w \tex_relax:D #3
10501 \exp_after:wN \__fp_parse_operand_other_prefix_unknown:NNN
10502 \exp_after:wN #2
10503 \fi:
10504 #3 #1
10505 }
10506 \cs_new:Npn \__fp_parse_operand_other_prefix_unknown:NNN #1#2#3
10507 {
10508 \cs_if_exist:cTF { \__fp_parse_infix_#1:N }
10509 {
10510 \__msg_kernel_expandable_error:nnn
10511 { kernel } { fp-missing-number } {#1}
10512 \exp_after:wN \c_nan_fp
10513 \tex_romannumeral:D -‘0
10514 \__fp_parse_infix:NN #3 #1
10515 }
10516 {
10517 \__msg_kernel_expandable_error:nnn
10518 { kernel } { fp-unknown-symbol } {#1}
10519 \__fp_parse_operand:Nw #3
10520 }
10521 }

```

(End definition for `__fp_parse_operand_other:NN`.)

The following forms are accepted:

-
- $\langle floating\ point \rangle$
- $\langle integer \rangle . \langle decimal \rangle e \langle exponent \rangle$

In both cases, $\langle signs \rangle$ is a (possibly empty) string of + and - (with any category code¹²).¹³

In the second form, the $\langle integer \rangle$ is a sequence of digits, whose length is not limited by constraints T_EX’s integer registers. It stops at the first non-digit character. The $\langle decimal \rangle$ part is formed by all digits from the dot (if it exists) until the first non-digit character. The $\langle exponent \rangle$ part has the form $\langle exponent\ sign \rangle \langle exponent\ body \rangle$, where $\langle exponent\ sign \rangle$ is any string of + or -, and $\langle exponent\ body \rangle$ is a string of digits, stopping, as usual, at the first non-digit.

Any missing part will take the appropriate default value.

- A missing $\langle exponent \rangle$ is considered to be zero.
- A number with no dot has zero decimal part.
- An empty $\langle integer \rangle$ part or decimal part is zero.

Border cases:

¹²Bruno: except 1, 2, 4, 10, 13, and those which cannot be tokens (0, 5, 9), so really, just 3, 6, 7, 8, 11, 12.

¹³Bruno: test (and implement) non-other digits.

- `e1` is considered as invalid input, and gives `qnan`.¹⁴ This will be important once parsing expressions is implemented, since `e-1` would be ambiguous otherwise.
- `.e3` and `.` are zero.

Bruno: expansion, not yet. Only f-expansion at the start, and unpacking of registers after signs.

Work-plan.

- Remove any leading sign and build the $\langle sign \rangle$ as we go. If the next character is a letter, go to the “special” branch, discussed later.
- Drop leading zeros.
- If the next character is a dot, drop some more zeros, keeping track of how many were dropped after the dot. Counting those gives $\langle exp_1 \rangle < 0$. Then read the decimal part with the `__fp_from_str_small` functions.
- Otherwise, $\langle exp_1 \rangle = 0$, and first read the integer part, then the decimal part. This is implemented through the more elaborate `__fp_from_str_large` functions.
- Continuing in the same line of expansion, read the exponent $\langle exp_2 \rangle$.
- Finally check that nothing is left.¹⁵

`__fp_parse_operand_digit:NN`

```

10522 \cs_new:Npn \__fp_parse_operand_digit:NN #1
10523 {
10524   \exp_after:wN \__fp_parse_infix_after_operand:NwN
10525   \exp_after:wN #1
10526   \tex_romannumeral:D -‘0
10527   \exp_after:wN \__fp_sanitizewN
10528   \int_use:N \__int_eval:w \c_zero \__fp_parse_trim_zeros:N
10529 }

```

(End definition for `__fp_parse_operand_digit:NN`.)

30.4.1 Trimming leading zeros

`__fp_parse_trim_zeros:N`
`__fp_parse_trim_end:w`

This function expects an already expanded token. It removes any leading zero, then distinguished 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.

```

10530 \cs_new:Npn \__fp_parse_trim_zeros:N #1
10531 {
10532   \if:w 0 #1
10533     \exp_after:wN \__fp_parse_trim_zeros:N

```

¹⁴Bruno: now just gives an error.

¹⁵Bruno: not done yet.

```

10534     \tex_romannumeral:D
10535 \else:
10536   \if:w . #1
10537     \exp_after:wN \__fp_parse_strim_zeros:N
10538     \tex_romannumeral:D
10539   \else:
10540     \__fp_parse_trim_end:w #1
10541   \fi:
10542 \fi:
10543 \__fp_parse_expand:w
10544 }
10545 \cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
10546 {
10547   \fi:
10548   \fi:
10549   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10550     \exp_after:wN \__fp_parse_large:N
10551   \else:
10552     \exp_after:wN \__fp_parse_zero:
10553   \fi:
10554   #1
10555 }

```

(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
 __fp_parse_strim_end:w 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.

```

10556 \cs_new:Npn \__fp_parse_strim_zeros:N #1
10557 {
10558   \if:w 0 #1
10559     - \c_one
10560     \exp_after:wN \__fp_parse_strim_zeros:N
10561     \tex_romannumeral:D
10562   \else:
10563     \__fp_parse_strim_end:w #1
10564   \fi:
10565   \__fp_parse_expand:w
10566 }
10567 \cs_new:Npn \__fp_parse_strim_end:w #1 \fi: \__fp_parse_expand:w
10568 {
10569   \fi:
10570   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10571     \exp_after:wN \__fp_parse_small:N
10572   \else:
10573     \exp_after:wN \__fp_parse_zero:
10574   \fi:
10575   #1

```

```

10576 }
(End definition for \__fp_parse_strim_zeros:N and \__fp_parse_strim_end:w.)

```

30.4.2 Exact zero

`__fp_parse_zero:` After reading a significand of 0, we need to remove any exponent, then put a sign of 1 for `__fp_sanitizewN`, denoting an exact zero.

```

10577 \cs_new:Npn \__fp_parse_zero:
10578 {
10579   \exp_after:wN ; \exp_after:wN 1
10580   \__int_value:w \__fp_parse_exponent:N
10581 }
(End definition for \__fp_parse_zero:.)

```

30.4.3 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.

```

10582 \cs_new:Npn \__fp_parse_small:N #1
10583 {
10584   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
10585   \int_use:N \__int_eval:w 1 \token_to_str:N #1
10586   \exp_after:wN \__fp_parse_small_leading:wwNN
10587   \__int_value:w 1
10588   \exp_after:wN \__fp_parse_digits_vii:N
10589   \tex_romannumeral:D \__fp_parse_expand:w
10590 }
(End definition for \__fp_parse_small:N.)

```

`__fp_parse_small_leading:wwNN` We leave *<digits>* *<zeros>* in the input stream: the functions used to grab digits are such that this constitutes digits 1 through 8 of the significand. Then prepare to pack 8 more digits, with an exponent shift of `\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.

```

10591 \cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
10592 {
10593   #1 #2
10594   \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
10595   \exp_after:wN \c_zero

```

```

10596 \int_use:N \__int_eval:w 1
10597 \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
10598 \token_to_str:N #4
10599 \exp_after:wN \__fp_parse_small_trailing:wwNN
10600 \__int_value:w 1
10601 \exp_after:wN \__fp_parse_digits_vi:N
10602 \tex_romannumeral:D
10603 \else:
10604 0000 0000 \__fp_parse_exponent:Nw #4
10605 \fi:
10606 \__fp_parse_expand:w
10607 }

```

(End definition for __fp_parse_small_leading:wwNN.)

__fp_parse_small_trailing:wwNN

Leave digits 10 to 15 (arguments #1 and #2) in the input stream. If the *<next token>* is a digit, it is the 16th digit, we keep it, then the `small_round` auxiliary considers this digit and all further digits to perform the rounding: the function expands to nothing or to +1. Otherwise, there is no 16-th digit, so we put a 0, and look for an exponent.

```

10608 \cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
10609 {
10610   #1 #2
10611   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
10612   \token_to_str:N #4
10613   \exp_after:wN \__fp_parse_small_round:NN
10614   \exp_after:wN #4
10615   \tex_romannumeral:D
10616   \else:
10617   0 \__fp_parse_exponent:Nw #4
10618   \fi:
10619   \__fp_parse_expand:w
10620 }

```

(End definition for __fp_parse_small_trailing:wwNN.)

__fp_parse_pack_trailing:NNNNNNww
 __fp_parse_pack_leading:NNNNNNww
 __fp_parse_pack_carry:w

Those functions are expanded after all the digits are found, we took care of the rounding, as well as the exponent. The last argument is the exponent. The previous five arguments are 8 digits which we pack in groups of 4, and the argument before that is 1, except in the rare case where rounding lead to a carry, in which case the argument is 2. The `trailing` function has an exponent shift as its first argument, which we add to the exponent found in the `e...` syntax. If the trailing digits cause a carry, the integer expression for the leading digits is incremented (+ `\c_one` in the code below). If the leading digits propagate this carry all the way up, the function `__fp_parse_pack_carry:w` increments the exponent, and changes the significand from 0000... to 1000...: this is simple because such a carry can only occur to give rise to a power of 10.

```

10621 \cs_new:Npn \__fp_parse_pack_trailing:NNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
10622 {
10623   \if_meaning:w 2 #2 + \c_one \fi:
10624   ; #8 + #1 ; {#3#4#5#6} {#7};
10625 }

```

```

10626 \cs_new:Npn \__fp_parse_pack_leading:NNNNNww #1 #2#3#4#5 #6; #7;
10627 {
10628   + #7
10629   \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
10630   ; 0 {#2#3#4#5} {#6}
10631 }
10632 \cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
10633 { \fi: + \c_one ; 0 {1000} }
(End definition for \__fp_parse_pack_trailing:NNNNNww, \__fp_parse_pack_leading:NNNNNww, and
\__fp_parse_pack_carry:w.)

```

30.4.4 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.

```

10634 \cs_new:Npn \__fp_parse_large:N #1
10635 {
10636   \exp_after:wN \__fp_parse_large_leading:wwNN
10637   \__int_value:w 1 \token_to_str:N #1
10638   \exp_after:wN \__fp_parse_digits_vii:N
10639   \tex_romannumeral:D \__fp_parse_expand:w
10640 }
(End definition for \__fp_parse_large:N.)

```

`__fp_parse_large_leading:wwNN` We shift the exponent by the number of digits in #1, namely the target number, 8, minus the *number of zeros* (number of digits missing). Then prepare to pack the 8 first digits. If the *next token* is a digit, read up to 6 more digits (digits 10 to 15). If it is a period, try to grab the end of our 8 first digits, branching to the `small` functions since the number of digit does not affect the exponent anymore. Finally, if this is the end of the significand, insert the *zeros* to complete the 8 first digits, insert 8 more, and look for an exponent.

```

10641 \cs_new:Npn \__fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
10642 {
10643   + \c_eight - #3
10644   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
10645   \int_use:N \__int_eval:w 1 #1
10646   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
10647     \exp_after:wN \__fp_parse_large_trailing:wwNN
10648     \__int_value:w 1 \token_to_str:N #4
10649     \exp_after:wN \__fp_parse_digits_vi:N
10650     \tex_romannumeral:D
10651   \else:
10652     \if:w . #4

```

```

10653         \exp_after:wN \__fp_parse_small_leading:wwNN
10654         \__int_value:w 1
10655         \cs:w
10656             __fp_parse_digits_
10657             \tex_romannumeral:D #3
10658             :N \exp_after:wN
10659         \cs_end:
10660         \tex_romannumeral:D
10661     \else:
10662         #2
10663         \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
10664         \exp_after:wN \c_zero
10665         \__int_value:w 1 0000 0000
10666         \__fp_parse_exponent:Nw #4
10667     \fi:
10668 \fi:
10669 \__fp_parse_expand:w
10670 }

```

(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.

```

10671 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
10672 {
10673     \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
10674     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
10675     \exp_after:wN \c_eight
10676     \int_use:N \__int_eval:w 1 #1 \token_to_str:N #4
10677     \exp_after:wN \__fp_parse_large_round:NN
10678     \exp_after:wN #4
10679     \tex_romannumeral:D
10680 \else:
10681     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
10682     \int_use:N \__int_eval:w \c_seven - #3 \exp_stop_f:
10683     \int_use:N \__int_eval:w 1 #1
10684     \if:w . #4
10685     \exp_after:wN \__fp_parse_small_trailing:wwNN
10686     \__int_value:w 1
10687     \cs:w
10688         __fp_parse_digits_
10689         \tex_romannumeral:D #3
10690     :N \exp_after:wN

```

```

10691         \cs_end:
10692         \tex_romannumeral:D
10693     \else:
10694         #2 0 \__fp_parse_exponent:Nw #4
10695     \fi:
10696 \fi:
10697 \__fp_parse_expand:w
10698 }
(End definition for \__fp_parse_large_trailing:wwNN.)

```

30.4.5 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 — and hopefully that is safe.

`__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.

```

10699 \cs_new:Npn \__fp_parse_exponent:Nw #1 #2 \__fp_parse_expand:w
10700 {
10701     \exp_after:wN ;
10702     \__int_value:w #2 \__fp_parse_exponent:N #1
10703 }
(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. Namely, 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.

```

10704 \cs_new:Npn \__fp_parse_exponent:N #1
10705 {
10706   \if:w e #1
10707     \exp_after:wN \__fp_parse_exponent_aux:N
10708     \tex_romannumeral:D
10709   \else:
10710     0 \__fp_parse_return_semicolon:w #1
10711   \fi:
10712   \__fp_parse_expand:w
10713 }
10714 \cs_new:Npn \__fp_parse_exponent_aux:N #1
10715 {
10716   \if_int_compare:w \if_catcode:w \tex_relax:D #1
10717     \c_zero \else: ' #1 \fi: > '9 \exp_stop_f:
10718     0 \exp_after:wN ; \exp_after:wN e
10719   \else:
10720     \exp_after:wN \__fp_parse_exponent_sign:N
10721   \fi:
10722   #1
10723 }

```

(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).

```

10724 \cs_new:Npn \__fp_parse_exponent_sign:N #1
10725 {
10726   \if:w + \if:w - #1 + \fi: \token_to_str:N #1
10727   \exp_after:wN \__fp_parse_exponent_sign:N
10728   \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
10729   \else:
10730     \exp_after:wN \__fp_parse_exponent_body:N
10731     \exp_after:wN #1
10732   \fi:
10733 }

```

(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).

```

10734 \cs_new:Npn \__fp_parse_exponent_body:N #1
10735 {
10736   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10737   \token_to_str:N #1
10738   \exp_after:wN \__fp_parse_exponent_digits:N
10739   \tex_romannumeral:D
10740   \else:
10741     \__fp_parse_exponent_keep:NTF #1
10742     { \__fp_parse_return_semicolon:w #1 }

```

```

10743     {
10744         \exp_after:wN ;
10745         \tex_romannumeral:D
10746     }
10747     \fi:
10748     \__fp_parse_expand:w
10749 }
(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 don't check for overflow of the exponent, hence there can be a TeX error. It is mostly harmless, except when parsing `0e9876543210`, which should be a valid representation of 0, but is not.

```

10750 \cs_new:Npn \__fp_parse_exponent_digits:N #1
10751 {
10752     \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10753     \token_to_str:N #1
10754     \exp_after:wN \__fp_parse_exponent_digits:N
10755     \tex_romannumeral:D
10756 }else:
10757     \__fp_parse_return_semicolon:w #1
10758 \fi:
10759 \__fp_parse_expand:w
10760 }
(End definition for \__fp_parse_exponent_digits:N.)

```

`__fp_parse_exponent_keep:N` 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.

```

10761 \prg_new_conditional:Npnn \__fp_parse_exponent_keep:N #1 { TF }
10762 {
10763     \if_catcode:w \tex_relax:D #1
10764     \if_meaning:w \tex_relax:D #1
10765     \if_int_compare:w \pdfstrcmp:D { \s__fp } { #1 } = \c_zero
10766     0
10767     \__msg_kernel_expandable_error:nnn
10768     { kernel } { fp-after-e } { floating~point~ }
10769     \prg_return_true:
10770 }else:
10771     0
10772     \__msg_kernel_expandable_error:nnn
10773     { kernel } { bad-variable } { #1 }

```

```

10774         \prg_return_false:
10775     \fi:
10776 \else:
10777     \if_int_compare:w
10778         \pdfTeX_strcmp:D { \__int_value:w #1 } { \tex_the:D #1 }
10779         = \c_zero
10780         \__int_value:w #1
10781     \else:
10782         0
10783         \__msg_kernel_expandable_error:nnn
10784         { kernel } { fp-after-e } { dimension~#1 }
10785     \fi:
10786     \prg_return_false:
10787 \fi:
10788 \else:
10789     0
10790     \__msg_kernel_expandable_error:nnn
10791     { kernel } { fp-missing } { exponent }
10792     \prg_return_true:
10793 \fi:
10794 }

```

(End definition for __fp_parse_exponent_keep:NTF.)

30.4.6 Beyond 16 digits: rounding

__fp_cfs_round_loop:N Used both for __fp_parse_small_round:NN and __fp_parse_large_round:NN. Should appear after a __int_eval:w 0. Reads digits one by one, until reaching a non-digit. Adds +1 for each digit. If all digits found are 0, ends the __int_eval:w by ;\c_zero, otherwise by ;\c_one. This is done by switching the loop to round_up at the first non-zero digit.

```

10795 \cs_new:Npn \__fp_cfs_round_loop:N #1
10796 {
10797     \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
10798     + \c_one
10799     \if:w 0 #1
10800         \exp_after:wN \__fp_cfs_round_loop:N
10801         \tex_romannumeral:D
10802     \else:
10803         \exp_after:wN \__fp_cfs_round_up:N
10804         \tex_romannumeral:D
10805     \fi:
10806 \else:
10807     \__fp_parse_return_semicolon:w \c_zero #1
10808 \fi:
10809 \__fp_parse_expand:w
10810 }
10811 \cs_new:Npn \__fp_cfs_round_up:N #1
10812 {
10813     \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:

```

```

10814         + 1
10815         \exp_after:wN \__fp_cfs_round_up:N
10816         \tex_romannumeral:D
10817     \else:
10818         \__fp_parse_return_semicolon:w \c_one #1
10819     \fi:
10820     \__fp_parse_expand:w
10821 }

```

(End definition for __fp_cfs_round_loop:N.)

__fp_parse_large_round:NN *<digit>* is the digit that we are currently rounding (we only care whether it is even or odd).

The goal is to get \c_zero or \c_one, check for an exponent afterwards, and combine it to the number of digits before the decimal point (which we thus need to keep track of).

```

10822 \cs_new:Npn \__fp_parse_large_round:NN #1#2
10823 {
10824     \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
10825     +
10826     \exp_after:wN \__fp_round_s:NNNw
10827     \exp_after:wN 0
10828     \exp_after:wN #1
10829     \exp_after:wN #2
10830     \int_use:N \__int_eval:w
10831     \exp_after:wN \__fp_parse_large_round_after:wNN
10832     \int_use:N \__int_eval:w \c_one
10833     \exp_after:wN \__fp_cfs_round_loop:N
10834     \else: %^^A could be dot, or e, or other
10835     \exp_after:wN \__fp_parse_large_round_dot_test:NNw
10836     \exp_after:wN #1
10837     \exp_after:wN #2
10838     \fi:
10839 }
10840 \cs_new:Npn \__fp_parse_large_round_dot_test:NNw #1#2
10841 {
10842     \if:w . #2
10843     \exp_after:wN \__fp_parse_small_round:NN
10844     \exp_after:wN #1
10845     \tex_romannumeral:D
10846     \else:
10847     \__fp_parse_exponent:Nw #2
10848     \fi:
10849     \__fp_parse_expand:w
10850 }
10851 \cs_new:Npn \__fp_parse_large_round_after:wNN #1 ; #2 #3
10852 {
10853     \if:w . #3
10854     \exp_after:wN \__fp_parse_large_round_after_aux:wN
10855     \int_use:N \__int_eval:w #1 +
10856     \c_zero * \__int_eval:w \c_zero

```

```

10857         \exp_after:wN \__fp_cfs_round_loop:N
10858         \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
10859     \else:
10860         + #2
10861         \exp_after:wN ;
10862         \int_use:N \__int_eval:w #1 +
10863         \exp_after:wN \__fp_parse_exponent:N
10864         \exp_after:wN #3
10865     \fi:
10866 }
10867 \cs_new:Npn \__fp_parse_large_round_after_aux:wN #1 ; #2
10868 {
10869     + #2
10870     \exp_after:wN ;
10871     \int_use:N \__int_eval:w #1 +
10872     \__fp_parse_exponent:N
10873 }

```

(End definition for __fp_parse_large_round:NN.)

__fp_parse_small_round:NN *<digit>* is the digit that we are currently rounding (we only care whether it is even or odd).

The goal is to get \c_zero or \c_one

```

10874 \cs_new:Npn \__fp_parse_small_round:NN #1#2
10875 {
10876     \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
10877     +
10878     \exp_after:wN \__fp_round_s:NNNw
10879     \exp_after:wN 0
10880     \exp_after:wN #1
10881     \exp_after:wN #2
10882     \int_use:N \__int_eval:w
10883     \exp_after:wN \__fp_parse_small_round_after:wN
10884     \int_use:N \__int_eval:w \c_zero
10885     \exp_after:wN \__fp_cfs_round_loop:N
10886     \tex_romannumeral:D
10887     \else:
10888         \__fp_parse_exponent:Nw #2
10889     \fi:
10890     \__fp_parse_expand:w
10891 }
10892 \cs_new:Npn \__fp_parse_small_round_after:wN #1; #2
10893 {
10894     + #2 \exp_after:wN ;
10895     \__int_value:w \__fp_parse_exponent:N
10896 }

```

(End definition for __fp_parse_small_round:NN.)

30.5 Main functions

`__fp_parse:n` Start a `\romannumeral` expansion so that `__fp_parse:n` expands in two steps. The `__fp_parse_after:ww` `__fp_parse_until:Nw` function will perform computations until reaching an operation with precedence `\c_minus_one` or less. Then check that there was indeed nothing left (this cannot happen), and stop the initial expansion with `\c_zero`.

```

10897 \cs_new:Npn \__fp_parse:n #1
10898 {
10899   \tex_romannumeral:D
10900   \exp_after:wN \__fp_parse_after:ww
10901   \tex_romannumeral:D
10902   \__fp_parse_until:Nw \c_minus_one
10903   \__fp_parse_expand:w #1 \s__fp_mark
10904   \s__fp_stop
10905 }
10906 \cs_new:Npn \__fp_parse_after:ww #1@ #2 \s__fp_stop
10907 {
10908   <assert> \assert_str_eq:nn { #2 } { \__fp_parse_infix_end:N \s__fp_mark }
10909   \c_zero #1
10910 }

```

(End definition for `__fp_parse:n`. This function is documented on page ??.)

`__fp_parse_until:Nw` The `__fp_parse_until` This is just a shorthand which sets up both `__fp_parse_until_test` and `__fp_parse_operand` with the same precedence. Note the trailing `\tex_romannumeral:D`. This function should be used with much care.

```

10911 \cs_new:Npn \__fp_parse_until:Nw #1
10912 {
10913   -'0
10914   \exp_after:wN \__fp_parse_until_test:NwN
10915   \exp_after:wN #1
10916   \tex_romannumeral:D -'0
10917   \exp_after:wN \__fp_parse_operand:Nw
10918   \exp_after:wN #1
10919   \tex_romannumeral:D
10920 }
10921 \cs_new:Npn \__fp_parse_until_test:NwN #1 #2 @ #3 { #3 #1 #2 @ }
10922 \cs_new_eq:NN \__fp_parse_stop_until:N \use_none:n

```

(End definition for `__fp_parse_until:Nw`. This function is documented on page ??.)

`__fp_parse_until_test:NwN` If `<bool>` is true, then `<fp>` is the floating point number that we are looking for (it ends with `;`), and this expands to `<fp>`. If `<bool>` is false, then the input stream actually looks like

`__fp_parse_until_test:NwN <prec> <fp12`

and we must feed `<prec>` to `\infix_?`, and perform `<oper>` on `<fp1 and <fp2: this triggers the expansion of \infix_? <prec>, continuing the computation (or stopping). In that case, the function \until yields`

```

    \_fp_parse_until_test:NwN <prec> <oper> <fp1> <fp2> \tex_romannumeral:D
    -'0 \infix_? <prec>

```

expanding $\langle oper \rangle$ next.

(End definition for `_fp_parse_until_test:NwN`.)

30.6 Main functions

`_fp_parse_infix_after_operand:NwN`

```

10923 \cs_new:Npn \_fp_parse_infix_after_operand:NwN #1 #2;
10924 {
10925   \_fp_exp_after_f:nw { \_fp_parse_infix:NN #1 }
10926   #2;
10927 }
10928 \group_begin:
10929   \char_set_catcode_letter:N \*
10930   \cs_new:Npn \_fp_parse_infix:NN #1 #2
10931   {
10932     \if_catcode:w \tex_relax:D #2
10933     \if_int_compare:w
10934       \pdfstrcmp:D { \s_fp_mark } { #2 }
10935       = \c_zero
10936       \exp_after:wN \exp_after:wN
10937       \exp_after:wN \_fp_parse_infix_end:N
10938     \else:
10939       \exp_after:wN \exp_after:wN
10940       \exp_after:wN \_fp_parse_infix_juxtapose:N
10941     \fi:
10942   \else:
10943     \if_int_compare:w
10944       \__int_eval:w \tex_uccode:D '#2 / 26
10945       = \c_three
10946       \exp_after:wN \exp_after:wN
10947       \exp_after:wN \_fp_parse_infix_juxtapose:N
10948     \else:
10949       \exp_after:wN \_fp_parse_infix_check:NNN
10950       \cs:w
10951         \_fp_parse_infix_#2:N
10952       \exp_after:wN \exp_after:wN \exp_after:wN
10953     \cs_end:
10954     \fi:
10955   \fi:
10956   #1
10957   #2
10958 }
10959 \cs_new:Npn \_fp_parse_infix_check:NNN #1#2#3
10960 {
10961   \if_meaning:w \tex_relax:D #1
10962     \__msg_kernel_expandable_error:nnn { kernel } { fp-missing } { * }

```

```

10963         \exp_after:wN \__fp_parse_infix_*:N
10964         \exp_after:wN #2
10965         \exp_after:wN #3
10966     \else:
10967         \exp_after:wN #1
10968         \exp_after:wN #2
10969         \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
10970     \fi:
10971 }
10972 \group_end:
(End definition for \__fp_parse_infix_after_operand:NwN.)

```

`__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 $\langle operation \rangle$ #4, given the types of the two $\langle operands \rangle$.

```

10973 \cs_new:Npn \__fp_parse_apply_binary:NwNwN #1 #2#3@ #4 #5#6@ #7
10974 {
10975     \exp_after:wN \__fp_parse_until_test:NwN
10976     \exp_after:wN #1
10977     \tex_romannumeral:D -‘0
10978     \cs:w
10979         __fp
10980         \__fp_type_from_scan:N #2
10981         _ #4
10982         \__fp_type_from_scan:N #5
10983         _o:ww
10984     \cs_end:
10985     #2#3 #5#6
10986     \tex_romannumeral:D -‘0 #7 #1
10987 }
(End definition for \__fp_parse_apply_binary:NwNwN.)

```

`__fp_parse_apply_unary_array:NNwN` Here, #2 is *e.g.*, `__fp_sin_o:w`, and expands once after the calculation.¹⁶ The argument #3 may be an array, so either we map through all its items, or we feed all items at once to the custom function.

```

10988 \cs_new:Npn \__fp_parse_apply_unary_array:NNwN #1#2#3@#4
10989 {
10990     #2 #3 @
10991     \tex_romannumeral:D -‘0 #4 #1
10992 }
10993 \cs_new:Npn \__fp_parse_apply_unary:NNwN #1#2#3@#4
10994 {
10995     #2 #3
10996     \tex_romannumeral:D -‘0 #4 #1
10997 }
10998 \cs_new:Npn \__fp_parse_unary_type:N #1
10999 { \__fp_type_from_scan:N #1 _o:w \cs_end: #1 }
(End definition for \__fp_parse_apply_unary_array:NNwN and \__fp_parse_apply_unary:NNwN.)

```

¹⁶Bruno: explain.

30.7 Prefix operators

30.7.1 Identifiers

```

\__fp_parse_word_inf:N A whole bunch of floating point numbers.
\__fp_parse_word_nan:N 11000 \cs_set_protected:Npn \__fp_tmp:w #1 #2
\__fp_parse_word_pi:N 11001 {
\__fp_parse_word_deg:N 11002 \cs_new_nopar:cpn { __fp_parse_word_#1:N }
\__fp_parse_word_em:N 11003 { \exp_after:wN #2 \tex_romannumeral:D -'0 \__fp_parse_infix:NN }
\__fp_parse_word_ex:N 11004 }
\__fp_parse_word_in:N 11005 \__fp_tmp:w { inf } \c_inf_fp
\__fp_parse_word_pt:N 11006 \__fp_tmp:w { nan } \c_nan_fp
\__fp_parse_word_pc:N 11007 \__fp_tmp:w { pi } \c_pi_fp
\__fp_parse_word_cm:N 11008 \__fp_tmp:w { deg } \c_one_degree_fp
\__fp_parse_word_mm:N 11009 \__fp_tmp:w { true } \c_one_fp
\__fp_parse_word_dd:N 11010 \__fp_tmp:w { false } \c_zero_fp
\__fp_parse_word_cc:N 11011 \__fp_tmp:w { pt } \c_one_fp
\__fp_parse_word_nd:N 11012 \cs_set_protected:Npn \__fp_tmp:w #1 #2
\__fp_parse_word_nc:N 11013 {
\__fp_parse_word_bp:N 11014 \cs_new_nopar:cpn { __fp_parse_word_#1:N }
\__fp_parse_word_sp:N 11015 {
11016 \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
11017 \s__fp \__fp_chk:w 10 #2 ;
11018 }
11019 }
11020 \__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
11021 \__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
11022 \__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
11023 \__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
11024 \__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
11025 \__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
11026 \__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
11027 \__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
11028 \__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
11029 \__fp_tmp:w {sp} { {-4} {1525} {8789} {0625} {0000} }
11030 \tl_map_inline:nn { {em} {ex} }
11031 {
11032 \cs_new_nopar:cpn { __fp_parse_word_#1:N }
11033 {
11034 \exp_after:wN \dim_to_fp:n \exp_after:wN
11035 { \dim_use:N \__dim_eval:w 1 #1 \exp_after:wN }
11036 \tex_romannumeral:D -'0 \__fp_parse_infix:NN
11037 }
11038 }

```

(End definition for `__fp_parse_word_inf:N` and others.)

```

\__fp_parse_word_abs:N Unary functions, which are applied to all of their arguments when receiving an array.
\__fp_parse_word_cos:N 11039 \tl_map_inline:nn
\__fp_parse_word_cot:N 11040 { {abs} {cos} {cot} {csc} {exp} {ln} {sec} {sin} {tan} }
\__fp_parse_word_csc:N 11041 {
\__fp_parse_word_exp:N
\__fp_parse_word_ln:N
\__fp_parse_word_sec:N
\__fp_parse_word_sin:N
\__fp_parse_word_tan:N

```

```

11042 \cs_new:cpn { __fp_parse_word_#1:N } ##1
11043 {
11044   \exp_after:wN \__fp_parse_apply_unary:NNwN
11045   \exp_after:wN ##1
11046   \cs:w __fp_ #1 \exp_after:wN \__fp_parse_unary_type:N
11047   \tex_romannumeral:D
11048   \__fp_parse_until:Nw \c_fifteen
11049   \__fp_parse_expand:w
11050 }
11051 }

```

(End definition for __fp_parse_word_abs:N and others.)

__fp_parse_word_max:N Those functions are also unary, but need to mix all of their arguments together.

__fp_parse_word_min:N

```

11052 \cs_set_protected:Npn \__fp_tmp:w #1#2
11053 {
11054   \cs_new:Npn #1 ##1
11055   {
11056     \exp_after:wN \__fp_parse_apply_unary_array:NNwN
11057     \exp_after:wN ##1
11058     \exp_after:wN #2
11059     \tex_romannumeral:D
11060     \__fp_parse_until:Nw \c_sixteen \__fp_parse_expand:w
11061   }
11062 }
11063 \__fp_tmp:w \__fp_parse_word_max:N \__fp_max_o:w
11064 \__fp_tmp:w \__fp_parse_word_min:N \__fp_min_o:w

```

(End definition for __fp_parse_word_max:N and __fp_parse_word_min:N.)

__fp_parse_word_round:N This function expects one or two arguments.

```

11065 \cs_new:Npn \__fp_parse_word_round:N #1#2
11066 {
11067   \if_meaning:w + #2
11068     \__fp_parse_round:Nw \__fp_round_to_pinf:NNN
11069   \else:
11070     \if_meaning:w 0 #2
11071       \__fp_parse_round:Nw \__fp_round_to_zero:NNN
11072     \else:
11073       \if_meaning:w - #2
11074         \__fp_parse_round:Nw \__fp_round_to_ninf:NNN
11075       \fi:
11076     \fi:
11077   \fi:
11078   \exp_after:wN \__fp_parse_apply_round:NNwN
11079   \exp_after:wN #1
11080   \exp_after:wN \__fp_round_to_nearest:NNN
11081   \tex_romannumeral:D
11082   \__fp_parse_until:Nw \c_sixteen \__fp_parse_expand:w #2
11083 }
11084 \cs_new:Npn \__fp_parse_round:Nw

```

```

11085     #1 #2 \__fp_round_to_nearest:NNN #3 \__fp_parse_expand:w #4
11086     { #2 #1 #3 \__fp_parse_expand:w }
11087 \cs_new:Npn \__fp_parse_apply_round:NNwN #1#2#3@#4
11088 {
11089     \if_case:w \__int_eval:w \__fp_array_count:n {#3} - \c_one \__int_eval_end:
11090         \__fp_round:Nwn #2 #3 {0} \tex_romannumeral:D
11091     \or: \__fp_round:Nww #2 #3 \tex_romannumeral:D
11092     \else:
11093         \_msg_kernel_expandable_error:nnnnn
11094         { kernel } { fp-num-args } { round() } { 1 } { 2 }
11095     \exp_after:wN \c_nan_fp \tex_romannumeral:D
11096     \fi:
11097     -'0 #4 #1
11098 }

```

(End definition for __fp_parse_word_round:N.)

30.7.2 Unary minus, plus, not

__fp_parse_prefix+:Nw A unary + does nothing.

```

11099 \cs_new_eq:cN { __fp_parse_prefix+:Nw } \__fp_parse_operand:Nw

```

(End definition for __fp_parse_prefix+:Nw.)

__fp_parse_prefix -:Nw Unary - is harder. Boolean not.

```

\__fp_parse_prefix!:Nw
11100 \cs_set_protected:Npn \__fp_tmp:w #1#2
11101 {
11102     \cs_new:cpn { __fp_parse_prefix_#1:Nw } ##1
11103     {
11104         \exp_after:wN \__fp_parse_apply_unary:NNwN
11105         \exp_after:wN ##1
11106         \cs:w __fp_ #2 \exp_after:wN \__fp_parse_unary_type:N
11107         \tex_romannumeral:D
11108         \if_int_compare:w \c_twelve < ##1
11109             \__fp_parse_until:Nw ##1
11110         \else:
11111             \__fp_parse_until:Nw \c_twelve
11112         \fi:
11113         \__fp_parse_expand:w
11114     }
11115 }
11116 \__fp_tmp:w - { - }
11117 \__fp_tmp:w ! { ! }

```

(End definition for __fp_parse_prefix -:Nw and __fp_parse_prefix!:Nw.)

30.7.3 Other prefixes

__fp_parse_prefix(:Nw

```

11118 \group_begin:
11119 \char_set_catcode_letter:N \)
11120 \cs_new:cpn { __fp_parse_prefix(:Nw } #1

```

```

11121 {
11122   \exp_after:wN \__fp_parse_lparen_after:NwN
11123   \exp_after:wN #1
11124   \tex_romannumeral:D
11125   \if_int_compare:w #1 = \c_sixteen
11126     \__fp_parse_until:Nw \c_one
11127   \else:
11128     \__fp_parse_until:Nw \c_zero
11129   \fi:
11130   \__fp_parse_expand:w
11131 }
11132 \cs_new:Npn \__fp_parse_lparen_after:NwN #1#2@#3
11133 {
11134   \token_if_eq_meaning:NNTF #3 \__fp_parse_infix_):N
11135   {
11136     \__fp_exp_after_array_f:w #2 \s__fp_stop
11137     \exp_after:wN \__fp_parse_infix:NN
11138     \exp_after:wN #1
11139     \tex_romannumeral:D \__fp_parse_expand:w
11140   }
11141   {
11142     \__msg_kernel_expandable_error:nnn { kernel } { fp-missing } { ) }
11143     #2 @ \__fp_parse_stop_until:N #3
11144   }
11145 }
11146 \group_end:
(End definition for \__fp_parse_prefix_(:Nw.)

\__fp_parse_prefix_.:Nw This function is called when a number starts with a dot.
11147 \cs_new:cpn {\__fp_parse_prefix_.:Nw} #1
11148 {
11149   \exp_after:wN \__fp_parse_infix_after_operand:NwN
11150   \exp_after:wN #1
11151   \tex_romannumeral:D -‘0
11152   \exp_after:wN \__fp_sanitize:wN
11153   \int_use:N \__int_eval:w \c_zero \__fp_parse_strim_zeros:N
11154 }
(End definition for \__fp_parse_prefix_.:Nw.)

```

30.8 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`. The latter two are only needed when defining the `\infix` function.

```

11155 \set_protected:Npn \__fp_tmp:w #1#2#3#4
11156 {
11157   \cs_new:Npn #1 ##1
11158   {
11159     \if_int_compare:w ##1 < #3

```

```

11160         \exp_after:wN @
11161         \exp_after:wN \__fp_parse_apply_binary:NwNwN
11162         \exp_after:wN #2
11163         \tex_romannumeral:D
11164         \__fp_parse_until:Nw #4
11165         \exp_after:wN \__fp_parse_expand:w
11166     \else:
11167         \exp_after:wN @
11168         \exp_after:wN \__fp_parse_stop_until:N
11169         \exp_after:wN #1
11170     \fi:
11171 }
11172 }

```

Using the general mechanism for arithmetic operations.

```

\__fp_parse_infix_+:N
\__fp_parse_infix_-:N
\__fp_parse_infix_/:N
\__fp_parse_infix_mul:N
\__fp_parse_infix_and:N
\__fp_parse_infix_or:N

```

```

11173 \group_begin:
11174 \char_set_catcode_other:N \&
11175 \__fp_tmp:w \__fp_parse_infix_juxtapose:N * \c_thirty_two \c_thirty_two
11176 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_ / :N } / \c_ten \c_ten
11177 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_mul:N } * \c_ten \c_ten
11178 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_ - :N } - \c_nine \c_nine
11179 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_ + :N } + \c_nine \c_nine
11180 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_and:N } & \c_five \c_five
11181 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_or:N } | \c_four \c_four
11182 \group_end:

```

(End definition for __fp_parse_infix_+:N and others.)

The power operation must be associative in the opposite order from all others. For this, we reverse the test, hence treating a “previous precedence” of \c_fourteen as less binding than ^.

```

11183 \group_begin:
11184 \char_set_catcode_letter:N ^
11185 \__fp_tmp:w \__fp_parse_infix^:N ^ \c_fifteen \c_fourteen
11186 \cs_new:cpn { \__fp_parse_infix_*:N } #1#2
11187 {
11188     \if:w * #2
11189         \exp_after:wN \__fp_parse_infix^:N
11190         \exp_after:wN #1
11191     \else:
11192         \exp_after:wN \__fp_parse_infix_mul:N
11193         \exp_after:wN #1
11194         \exp_after:wN #2
11195     \fi:
11196 }
11197 \group_end:

```

(End definition for __fp_parse_infix_*:N. This function is documented on page ??.)

```

\__fp_parse_infix_|:Nw
\__fp_parse_infix_&:Nw

```

```

11198 \group_begin:

```

```

11199 \char_set_catcode_letter:N \l
11200 \char_set_catcode_letter:N \&
11201 \cs_new:Npn \__fp_parse_infix_|:N #1#2
11202 {
11203   \if:w | #2
11204     \exp_after:wN \__fp_parse_infix_|:N
11205     \exp_after:wN #1
11206     \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
11207   \else:
11208     \exp_after:wN \__fp_parse_infix_or:N
11209     \exp_after:wN #1
11210     \exp_after:wN #2
11211   \fi:
11212 }
11213 \cs_new:Npn \__fp_parse_infix_&:N #1#2
11214 {
11215   \if:w & #2
11216     \exp_after:wN \__fp_parse_infix_&:N
11217     \exp_after:wN #1
11218     \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
11219   \else:
11220     \exp_after:wN \__fp_parse_infix_and:N
11221     \exp_after:wN #1
11222     \exp_after:wN #2
11223   \fi:
11224 }
11225 \group_end:

```

(End definition for __fp_parse_infix_|:Nw. This function is documented on page ??.)

```

\__fp_parse_infix_<:N
\__fp_parse_infix_=:N 11226 \cs_new:cpn { __fp_parse_infix_<:N } #1
\__fp_parse_infix_>:N 11227 {
\__fp_parse_infix_!:N 11228   \__fp_infix_compare:N #1 \c_one_fp
  \__fp_parse_infix_excl_aux:NN 11229   \c_zero_fp \c_zero_fp \c_zero_fp \c_zero_fp <
  \__fp_parse_infix_excl_error: 11230 }
  \__fp_infix_compare:N 11231 \cs_new:cpn { __fp_parse_infix_=:N } #1
\__fp_parse_compare:NNNNNNw 11232 {
  \__fp_infix_compare:N #1 \c_one_fp
  \__fp_parse_compare_expand:NNNNNNw 11233   \c_zero_fp \c_zero_fp \c_zero_fp \c_zero_fp =
\__fp_parse_compare_end:NNNN 11234 }
  \__fp_compare:wNNNNw 11235 }
11236 \cs_new:cpn { __fp_parse_infix_>:N } #1
11237 {
11238   \__fp_infix_compare:N #1 \c_one_fp
11239   \c_zero_fp \c_zero_fp \c_zero_fp \c_zero_fp >
11240 }
11241 \cs_new:cpn { __fp_parse_infix_!:N } #1
11242 {
11243   \exp_after:wN \__fp_parse_infix_excl_aux:NN
11244   \exp_after:wN #1 \tex_romannumeral:D \__fp_parse_expand:w
11245 }

```

```

11246 \cs_new:Npn \__fp_parse_infix_excl_aux:NN #1#2
11247 {
11248   \__fp_infix_compare:N #1 \c_zero_fp
11249   \c_one_fp \c_one_fp \c_one_fp \c_one_fp #2
11250 }
11251 \cs_new:Npn \__fp_parse_infix_excl_error:
11252 {
11253   \_msg_kernel_expandable_error:nnnn
11254   { kernel } { fp-missing } { = } { ~after~!. }
11255 }
11256 \cs_new:Npn \__fp_infix_compare:N #1
11257 {
11258   \if_int_compare:w #1 < \c_seven
11259     \exp_after:wN \__fp_parse_compare:NNNNNNw
11260     \exp_after:wN \__fp_parse_infix_excl_error:
11261   \else:
11262     \exp_after:wN @
11263     \exp_after:wN \__fp_parse_stop_until:N
11264     \exp_after:wN \__fp_infix_compare:N
11265   \fi:
11266 }
11267 \cs_new:Npn \__fp_parse_compare:NNNNNNw #1#2#3#4#5#6#7
11268 {
11269   \if_case:w
11270     \if_catcode:w \tex_relax:D #7
11271     \c_minus_one
11272   \else:
11273     \__int_eval:w '#7 - '< \__int_eval_end:
11274   \fi:
11275   \__fp_parse_compare_expand:NNNNNNw #2#2#4#5#6
11276   \or: \__fp_parse_compare_expand:NNNNNNw #2#3#2#5#6
11277   \or: \__fp_parse_compare_expand:NNNNNNw #2#3#4#2#6
11278   \or: \__fp_parse_compare_expand:NNNNNNw #2#3#4#5#2
11279   \else: #1 \__fp_parse_compare_end:NNNN #3#4#5#6#7
11280   \fi:
11281 }
11282 \cs_new:Npn \__fp_parse_compare_expand:NNNNNNw #1#2#3#4#5
11283 {
11284   \exp_after:wN \__fp_parse_compare:NNNNNNw
11285   \exp_after:wN \prg_do_nothing:
11286   \exp_after:wN #1
11287   \exp_after:wN #2
11288   \exp_after:wN #3
11289   \exp_after:wN #4
11290   \exp_after:wN #5
11291   \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
11292 }
11293 \cs_new:Npn \__fp_parse_compare_end:NNNN #1#2#3#4#5 \fi:
11294 {
11295   \fi:

```

```

11296 \exp_after:wN @
11297 \exp_after:wN \_fp_parse_apply_compare:NwNNNNwN
11298 \exp_after:wN #1
11299 \exp_after:wN #2
11300 \exp_after:wN #3
11301 \exp_after:wN #4
11302 \tex_romannumeral:D
11303 \_fp_parse_until:Nw \c_seven \_fp_parse_expand:w #5
11304 }
11305 \cs_new:Npn \_fp_parse_apply_compare:NwNNNNwN #1 #2@ #3#4#5#6 #7@ #8
11306 {
11307 \exp_after:wN \_fp_parse_until_test:NwN
11308 \exp_after:wN #1
11309 \tex_romannumeral:D -‘0
11310 \exp_after:wN \exp_after:wN
11311 \exp_after:wN \exp_after:wN
11312 \exp_after:wN \exp_after:wN
11313 \if_case:w \_fp_compare_back:ww #7 #2 \exp_stop_f:
11314 #4
11315 \or: #5
11316 \or: #6
11317 \else: #3
11318 \fi:
11319 \tex_romannumeral:D -‘0 #8 #1
11320 }

```

(End definition for _fp_parse_infix_<:N and others. These functions are documented on page ??.)

```

\_fp_parse_infix_?:N
\_fp_parse_infix_:N

```

```

11321 \group_begin:
11322 \char_set_catcode_letter:N \?
11323 \cs_new:Npn \_fp_parse_infix_?:N #1
11324 {
11325 \if_int_compare:w #1 < \c_three
11326 \exp_after:wN @
11327 \exp_after:wN \_fp_ternary:NwN
11328 \tex_romannumeral:D
11329 \_fp_parse_until:Nw \c_three
11330 \exp_after:wN \_fp_parse_expand:w
11331 \else:
11332 \exp_after:wN @
11333 \exp_after:wN \_fp_parse_stop_until:N
11334 \exp_after:wN \_fp_parse_infix_?:N
11335 \fi:
11336 }
11337 \cs_new:Npn \_fp_parse_infix_:N #1
11338 {
11339 \if_int_compare:w #1 < \c_three
11340 \_msg_kernel_expandable_error:nnnn
11341 { kernel } { fp-missing } { ? } { ~for~?: }
11342 \exp_after:wN @

```



```

11343         \exp_after:wN \__fp_ternary_auxii:NwwN
11344         \tex_romannumeral:D
11345         \__fp_parse_until:Nw \c_two
11346         \exp_after:wN \__fp_parse_expand:w
11347     \else:
11348         \exp_after:wN @
11349         \exp_after:wN \__fp_parse_stop_until:N
11350         \exp_after:wN \__fp_parse_infix_::N
11351     \fi:
11352 }
11353 \group_end:
(End definition for \__fp_parse_infix_?:N and \__fp_parse_infix_::N.)

```

`__fp_parse_infix_):N` This one is a little bit odd: force every previous operator to end, regardless of the precedence. This is very similar to `__fp_parse_infix_end:N`.

```

11354 \group_begin:
11355     \char_set_catcode_letter:N \
11356     \cs_new:Npn \__fp_parse_infix_):N #1
11357     {
11358         \if_int_compare:w #1 < \c_zero
11359             \__msg_kernel_expandable_error:nnn { kernel } { fp-extra } { } }
11360         \exp_after:wN \__fp_parse_infix:NN
11361         \exp_after:wN #1
11362         \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
11363     \else:
11364         \exp_after:wN @
11365         \exp_after:wN \__fp_parse_stop_until:N
11366         \exp_after:wN \__fp_parse_infix_):N
11367     \fi:
11368 }
11369 \group_end:
11370 \cs_new:Npn \__fp_parse_infix_end:N #1
11371     { @ \__fp_parse_stop_until:N \__fp_parse_infix_end:N }
(End definition for \__fp_parse_infix_):N.)

```

`__fp_parse_infix_`

```

:N
11372 \group_begin:
11373     \char_set_catcode_letter:N \,
11374     \cs_new:Npn \__fp_parse_infix_,:N #1
11375     {
11376         \if_int_compare:w #1 > \c_one
11377             \exp_after:wN @
11378             \exp_after:wN \__fp_parse_stop_until:N
11379             \exp_after:wN \__fp_parse_infix_,:N
11380         \else:
11381             \if_int_compare:w #1 = \c_one
11382                 \exp_after:wN \__fp_parse_infix_comma:w
11383                 \tex_romannumeral:D
11384             \else:

```

```

11385         \exp_after:wN \__fp_parse_infix_comma_gobble:w
11386         \tex_romannumeral:D
11387         \fi:
11388         \__fp_parse_until:Nw \c_one
11389         \exp_after:wN \__fp_parse_expand:w
11390         \fi:
11391     }
11392     \cs_new:Npn \__fp_parse_infix_comma:w #1 @
11393     { #1 @ \__fp_parse_stop_until:N }
11394     \cs_new:Npn \__fp_parse_infix_comma_gobble:w #1 @
11395     {
11396         \__msg_kernel_expandable_error:nn { kernel } { fp-extra-comma }
11397         @ \__fp_parse_stop_until:N
11398     }
11399     \group_end:
(End definition for \__fp_parse_infix_ and :N.)

```

31 Messages

```

11400 \__msg_kernel_new:nnn { kernel } { unknown-fp-word }
11401 { Unknown~fp~word~#1. }
11402 \__msg_kernel_new:nnn { kernel } { fp-missing }
11403 { Missing~#1~inserted #2. }
11404 \__msg_kernel_new:nnn { kernel } { fp-extra }
11405 { Extra~#1~ignored. }
11406 \__msg_kernel_new:nnn { kernel } { fp-early-end }
11407 { Premature~end~in~fp~expression. }
11408 \__msg_kernel_new:nnn { kernel } { fp-after-e }
11409 { Cannot~use~#1 after~'e'. }
11410 \__msg_kernel_new:nnn { kernel } { fp-missing-number }
11411 { Missing~number~before~'#1'. }
11412 \__msg_kernel_new:nnn { kernel } { fp-unknown-symbol }
11413 { Unknown~symbol~#1~ignored. }
11414 \__msg_kernel_new:nnn { kernel } { fp-extra-comma }
11415 { Unexpected~comma:~extra~arguments~ignored. }
11416 \__msg_kernel_new:nnn { kernel } { fp-num-args }
11417 { #1~expects~between~#2~and~#3~arguments. }
11418 </initex | package>

```

32 l3fp-logic Implementation

```

11419 <*initex | package>
11420 <@@=fp>

```

32.1 Syntax of internal functions

- `__fp_compare_npos:nwnw {<exp01>} <body1>} {<exp02>} <body2>} ;`
- `__fp_max_o:w <floating point array>`

- `__fp_min_o:w` *⟨floating point array⟩*
- `__fp_!_o:w` *⟨floating point⟩*
- `__fp_&_o:ww` *⟨floating point⟩* *⟨floating point⟩*
- `__fp_|_o:ww` *⟨floating point⟩* *⟨floating point⟩*
- `__fp_ternary:NwwN`, `__fp_ternary_auxi:NwwN`, `__fp_ternary_auxii:NwwN` have to be understood.

32.2 Existence test

`\fp_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.
`\fp_if_exist_p:c` 11421 `\prg_new_eq_conditional:NNn \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }`
`\fp_if_exist:N \underline{TF}` 11422 `\prg_new_eq_conditional:NNn \fp_if_exist:c \cs_if_exist:c { TF , T , F , p }`
`\fp_if_exist:c \underline{TF}` (End definition for `\fp_if_exist:N` and `\fp_if_exist:c`. These functions are documented on page ??.)

32.3 Comparison

`\fp_compare_p:n` Within floating point expressions, comparison operators are treated as operations, so we
`\fp_compare:n \underline{TF}` evaluate #1, then compare with 0.
`__fp_compare_return:w` 11423 `\prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }`
11424 `{`
11425 `\exp_after:wN __fp_compare_return:w`
11426 `\tex_romannumeral:D -‘0 __fp_parse:n {#1}`
11427 `}`
11428 `\cs_new:Npn __fp_compare_return:w \s__fp __fp_chk:w #1#2;`
11429 `{`
11430 `\if_meaning:w 0 #1`
11431 `\prg_return_false:`
11432 `\else:`
11433 `\prg_return_true:`
11434 `\fi:`
11435 `}`
(End definition for `\fp_compare:n`. These functions are documented on page ??.)

`\fp_compare_p:nNn` Evaluate #1 and #3, using an auxiliary to expand both, and feed the two floating point
`\fp_compare:nNn \underline{TF}` numbers swapped to `__fp_compare_back:ww`, defined below. Compare the result with
`__fp_compare_aux:wn` ‘#2-‘=, which is -1 for $<$, 0 for $=$, 1 for $>$ and 2 for $?$.
11436 `\prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }`
11437 `{`
11438 `\if_int_compare:w`
11439 `\exp_after:wN __fp_compare_aux:wn`
11440 `\tex_romannumeral:D -‘0 __fp_parse:n {#1} {#3}`
11441 `= __int_eval:w ‘#2 - ‘= __int_eval_end:`
11442 `\prg_return_true:`
11443 `\else:`
11444 `\prg_return_false:`

```

11445     \fi:
11446   }
11447   \cs_new:Npn \__fp_compare_aux:wn #1; #2
11448   {
11449     \exp_after:wN \__fp_compare_back:ww
11450     \tex_romannumeral:D -'0 \__fp_parse:n {#2} #1;
11451   }

```

(End definition for `\fp_compare:nNn`. These functions are documented on page 172.)

`__fp_compare_back:ww` `__fp_compare_back:ww <y> ; <x> ;`
`__fp_compare_nan:w` Expands (in the same way as `\int_eval:n`) to -1 if $x < y$, 0 if $x = y$, 1 if $x > y$, and 2 otherwise (denoted as $x?y$). If either operand is `nan`, stop the comparison with `__fp_compare_nan:w` returning 2 . If x is negative, swap the outputs 1 and -1 (i.e., $>$ and $<$); we can henceforth assume that $x \geq 0$. If $y \geq 0$, and they have the same type, either they are normal and we compare them with `__fp_compare_npos:nwnw`, or they are equal. If $y \geq 0$, but of a different type, the highest type is a larger number. Finally, if $y \leq 0$, then $x > y$, unless both are zero.

```

11452   \cs_new:Npn \__fp_compare_back:ww
11453   {
11454     \s__fp \__fp_chk:w #1 #2 #3;
11455     \s__fp \__fp_chk:w #4 #5 #6;
11456   }
11457   \__int_value:w
11458   \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
11459   \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
11460   \if_meaning:w 2 #5 - \fi:
11461   \if_meaning:w #2 #5
11462   \if_meaning:w #1 #4
11463   \if_meaning:w 1 #1
11464   \__fp_compare_npos:nwnw #6; #3;
11465   \else:
11466     0
11467   \fi:
11468   \else:
11469     \if_int_compare:w #4 < #1 - \fi: 1
11470   \fi:
11471   \else:
11472     \if_int_compare:w #1#4 = \c_zero
11473     0
11474   \else:
11475     1
11476   \fi:
11477   \fi:
11478   \exp_stop_f:
11479   }
11480   \cs_new:Npn \__fp_compare_nan:w #1 \exp_stop_f: { \c_two }

```

(End definition for `__fp_compare_back:ww` and `__fp_compare_nan:w`.)

`__fp_compare_npos:nwnw` `__fp_compare_npos:nwnw {<expo1>} <body1> ; {<expo2>} <body2> ;`
`__fp_compare_significand:nnnnnnnn`

Within an `_int_value:w ... \exp_stop_f:` construction, this expands to 0 if the two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```

11480 \cs_new:Npn \_fp_compare_npos:nwnw #1#2; #3#4;
11481 {
11482   \if_int_compare:w #1 = #3 \exp_stop_f:
11483     \_fp_compare_significand:nnnnnnnn #2 #4
11484   \else:
11485     \if_int_compare:w #1 < #3 - \fi: 1
11486   \fi:
11487 }
11488 \cs_new:Npn \_fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
11489 {
11490   \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
11491     \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
11492     0
11493   \else:
11494     \if_int_compare:w #3#4 < #7#8 - \fi: 1
11495   \fi:
11496   \else:
11497     \if_int_compare:w #1#2 < #5#6 - \fi: 1
11498   \fi:
11499 }

```

(End definition for `_fp_compare_npos:nwnw`. This function is documented on page ??.)

32.4 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.

```

11500 \cs_new:Npn \fp_do_until:nn #1#2
11501 {
11502   #2
11503   \fp_compare:nF {#1}
11504   { \fp_do_until:nn {#1} {#2} }
11505 }
11506 \cs_new:Npn \fp_do_while:nn #1#2
11507 {
11508   #2
11509   \fp_compare:nT {#1}
11510   { \fp_do_while:nn {#1} {#2} }
11511 }
11512 \cs_new:Npn \fp_until_do:nn #1#2
11513 {
11514   \fp_compare:nF {#1}
11515   {

```

```

11516         #2
11517         \fp_until_do:nn {#1} {#2}
11518     }
11519 }
11520 \cs_new:Npn \fp_while_do:nn #1#2
11521 {
11522     \fp_compare:nT {#1}
11523     {
11524         #2
11525         \fp_while_do:nn {#1} {#2}
11526     }
11527 }

```

(End definition for `\fp_do_until:nn` and others. These functions are documented on page 173.)

`\fp_do_until:nNnn` As above but not using the `nNn` syntax.

```

11528 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
11529 {
11530     #4
11531     \fp_compare:nNnF {#1} #2 {#3}
11532     { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
11533 }
11534 \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
11535 {
11536     #4
11537     \fp_compare:nNnT {#1} #2 {#3}
11538     { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
11539 }
11540 \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
11541 {
11542     \fp_compare:nNnF {#1} #2 {#3}
11543     {
11544         #4
11545         \fp_until_do:nNnn {#1} #2 {#3} {#4}
11546     }
11547 }
11548 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
11549 {
11550     \fp_compare:nNnT {#1} #2 {#3}
11551     {
11552         #4
11553         \fp_while_do:nNnn {#1} #2 {#3} {#4}
11554     }
11555 }

```

(End definition for `\fp_do_until:nNnn` and others. These functions are documented on page 173.)

32.5 Extrema

`__fp_max_o:w` The maximum (minimum) of an array of floating point numbers is computed by reading them sequentially, keeping track of the largest (smallest) number found so far. We start

`__fp_min_o:w`

with $-\infty$ (∞) since every number is larger (smaller) than that. The weird fp-like trailing marker breaks the loop correctly: see the precise definition of `__fp_minmax_loop:Nww`.

```

11556 \cs_new:Npn \__fp_max_o:w #1 @
11557 {
11558   \exp_after:wN \__fp_minmax_loop:Nww
11559   \exp_after:wN \c_minus_one
11560   \c_minus_inf_fp
11561   #1
11562   \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
11563 }
11564 \cs_new:Npn \__fp_min_o:w #1 @
11565 {
11566   \exp_after:wN \__fp_minmax_loop:Nww
11567   \exp_after:wN \c_one
11568   \c_inf_fp
11569   #1
11570   \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
11571 }

```

(End definition for `__fp_max_o:w` and `__fp_min_o:w`.)

`__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.

```

11572 \cs_new:Npn \__fp_minmax_loop:Nww
11573   #1 \s__fp \__fp_chk:w #2#3; \s__fp \__fp_chk:w #4#5;
11574 {
11575   \if_meaning:w 3 #4
11576     \if_meaning:w 3 #2
11577       \__fp_minmax_auxi:ww
11578     \else:
11579       \__fp_minmax_auxii:ww
11580     \fi:
11581   \else:
11582     \if_int_compare:w
11583       \__fp_compare_back:ww
11584       \s__fp \__fp_chk:w #4#5;
11585       \s__fp \__fp_chk:w #2#3;
11586       = #1
11587       \__fp_minmax_auxii:ww
11588     \else:
11589       \__fp_minmax_auxi:ww
11590     \fi:
11591   \fi:
11592   \__fp_minmax_loop:Nww #1
11593   \s__fp \__fp_chk:w #2#3;

```

```

11594     \s__fp \__fp_chk:w #4#5;
11595 }
(End definition for \__fp_minmax_loop:Nww.)

```

```

\__fp_minmax_auxi:ww Keep the first/second number, and remove the other.
\__fp_minmax_auxii:ww
11596 \cs_new:Npn \__fp_minmax_auxi:ww #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
11597 { \fi: \fi: #2 \s__fp #3 ; }
11598 \cs_new:Npn \__fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
11599 { \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.
11600 \cs_new:Npn \__fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3; #4;
11601 { \fi: \__fp_exp_after_o:w \s__fp #3; }
(End definition for \__fp_minmax_break_o:w.)

```

32.6 Boolean operations

`__fp_!_o:w` Return true or false, with two expansions, one to exit the conditional, and one to please l3fp-parse.

```

11602 \cs_new:cpn { __fp_!_o:w } \s__fp \__fp_chk:w #1#2;
11603 {
11604     \if_meaning:w 0 #1
11605     \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
11606     \else:
11607     \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
11608     \fi:
11609 }
(End definition for \__fp_!_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.

```

11610 \group_begin:
11611 \char_set_catcode_letter:N &
11612 \char_set_catcode_letter:N |
11613 \cs_new:Npn \__fp_&_o:ww #1 \s__fp \__fp_chk:w #2#3;
11614 {
11615     \if_meaning:w 0 #2 #1
11616     \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
11617     \fi:
11618     \__fp_exp_after_o:w
11619 }
11620 \cs_new_nopar:Npn \__fp_|_o:ww { \__fp_&_o:ww \else: }

```



```

11621 \group_end:
11622 \cs_new:Npn \__fp_and_return:wNw #1; \fi: #2#3; { \fi: #2 #1; }
(End definition for \__fp_&o:ww. This function is documented on page ??.)

```

32.7 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
11623 \cs_new:Npn \__fp_ternary:NwwN #1 #2@ #3@ #4
11624 {
11625   \if_meaning:w \__fp_parse_infix_:N #4
11626   \__fp_ternary_loop:Nw
11627   #2
11628   \s_fp \__fp_chk:w { \__fp_ternary_loop_break:w } ;
11629   \__fp_ternary_break_point:n { \exp_after:wN \__fp_ternary_auxi:NwwN }
11630   \exp_after:wN #1
11631   \tex_romannumeral:D -'0
11632   \__fp_exp_after_array_f:w #3 \s_fp_stop
11633   \exp_after:wN @
11634   \tex_romannumeral:D
11635   \__fp_parse_until:Nw \c_two
11636   \__fp_parse_expand:w
11637   \else:
11638   \__msg_kernel_expandable_error:nnnn
11639   { kernel } { fp-missing } { : } { ~for~?: }
11640   \exp_after:wN \__fp_parse_until_test:NwN
11641   \exp_after:wN #1
11642   \tex_romannumeral:D -'0
11643   \__fp_exp_after_array_f:w #3 \s_fp_stop
11644   \exp_after:wN #4
11645   \exp_after:wN #1
11646   \fi:
11647   }
11648 \cs_new:Npn \__fp_ternary_loop_break:w #1 \fi: #2 \__fp_ternary_break_point:n #3
11649 {
11650   \c_zero = \c_zero \fi:
11651   \exp_after:wN \__fp_ternary_auxii:NwwN
11652   }
11653 \cs_new:Npn \__fp_ternary_loop:Nw \s_fp \__fp_chk:w #1#2;
11654 {
11655   \if_int_compare:w #1 > \c_zero
11656   \exp_after:wN \__fp_ternary_map_break:
11657   \fi:
11658   \__fp_ternary_loop:Nw
11659   }
11660 \cs_new:Npn \__fp_ternary_map_break: #1 \__fp_ternary_break_point:n #2 {#2}

```

```

11661 \cs_new:Npn \__fp_ternary_auxi:NwwN #1#2@#3@#4
11662 {
11663   \exp_after:wN \__fp_parse_until_test:NwN
11664   \exp_after:wN #1
11665   \tex_romannumeral:D -‘0
11666   \__fp_exp_after_array_f:w #2 \s__fp_stop
11667   #4 #1
11668 }
11669 \cs_new:Npn \__fp_ternary_auxii:NwwN #1#2@#3@#4
11670 {
11671   \exp_after:wN \__fp_parse_until_test:NwN
11672   \exp_after:wN #1
11673   \tex_romannumeral:D -‘0
11674   \__fp_exp_after_array_f:w #3 \s__fp_stop
11675   #4 #1
11676 }

```

(End definition for `__fp_ternary:NwwN`, `__fp_ternary_auxi:NwwN`, and `__fp_ternary_auxii:NwwN`. These functions are documented on page ??.)

```

11677 </initex | package>

```

33 l3fp-basics Implementation

```

11678 <*initex | package>

```

```

11679 <@@=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>.

33.1 Common to several operations

```

\__fp_basics_pack_low:NNNNw
  \__fp_basics_pack_high:NNNNw
  \__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}`.

```

11680 \cs_new:Npn \__fp_basics_pack_low:NNNNw #1 #2#3#4#5 #6;
11681 {
11682   \if_meaning:w 2 #1
11683     + \c_one
11684   \fi:
11685   ; {#2#3#4#5} {#6} ;
11686 }

```

```

11687 \cs_new:Npn \__fp_basics_pack_high:NNNNWw #1 #2#3#4#5 #6;
11688 {
11689     \if_meaning:w 2 #1
11690         \__fp_basics_pack_high_carry:w
11691         \fi:
11692         ; {#2#3#4#5} {#6}
11693     }
11694 \cs_new:Npn \__fp_basics_pack_high_carry:w \fi: ; #1
11695 { \fi: + \c_one ; {1000} }

```

(End definition for __fp_basics_pack_low:NNNNWw, __fp_basics_pack_high:NNNNWw, and __fp_basics_pack_high_carry:w)

```

\__fp_basics_pack_weird_low:NNNNWw
\__fp_basics_pack_weird_high:NNNNNNNNWw

```

I don't fully understand those functions, used for additions and divisions. Hence the name.

```

11696 \cs_new:Npn \__fp_basics_pack_weird_low:NNNNWw #1 #2#3#4 #5;
11697 {
11698     \if_meaning:w 2 #1
11699         + \c_one
11700         \fi:
11701         \__int_eval_end:
11702         #2#3#4; {#5} ;
11703     }
11704 \cs_new:Npn \__fp_basics_pack_weird_high:NNNNNNNNWw
11705     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:NNNNWw and __fp_basics_pack_weird_high:NNNNNNNNWw.)

33.2 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.

33.2.1 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.

```

11706 \cs_new_nopar:cpx { __fp_-_o:ww } \s__fp
11707 {
11708   \exp_not:c { __fp+_o:ww }
11709   \exp_not:n { \s__fp \__fp_neg_sign:N }
11710 }

```

(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.

```

11711 \cs_new:cpn { __fp+_o:ww }
11712   \s__fp #1 \__fp_chk:w #2 #3 ; \s__fp \__fp_chk:w #4 #5
11713 {
11714   \if_case:w
11715     \if_meaning:w #2 #4
11716       #2 \exp_stop_f:
11717     \else:
11718       \if_int_compare:w #2 > #4 \exp_stop_f:
11719         \c_three
11720       \else:
11721         \c_minus_one
11722       \fi:
11723     \fi:
11724     \exp_after:wN \__fp_add_zeros_o:Nww \__int_value:w
11725   \or:   \exp_after:wN \__fp_add_normal_o:Nww \__int_value:w
11726   \or:   \exp_after:wN \__fp_add_inf_o:Nww \__int_value:w
11727   \or:   \__fp_case_return_i_o:ww
11728   \else: \exp_after:wN \__fp_add_return_ii_o:Nww \__int_value:w
11729   \fi:
11730   #1 #5

```

```

11731     \s__fp \__fp_chk:w #2 #3 ;
11732     \s__fp \__fp_chk:w #4 #5
11733 }
(End definition for \__fp_+_o:ww.)

```

__fp_add_return_ii_o:Nww Ignore the first operand, and return the second, but using the sign #1 rather than #4. As usual, expand after the floating point.

```

11734 \cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
11735 { \__fp_exp_after_o:w \s__fp \__fp_chk:w #3 #1 }
(End definition for \__fp_add_return_ii_o:Nww.)

```

__fp_add_zeros_o:Nww Adding two zeros yields \c_zero_fp, except if both zeros were -0.

```

11736 \cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
11737 {
11738   \if_int_compare:w #2 #1 = 20 \exp_stop_f:
11739   \exp_after:wN \__fp_add_return_ii_o:Nww
11740   \else:
11741     \__fp_case_return_i_o:ww
11742   \fi:
11743   #1
11744   \s__fp \__fp_chk:w 0 #2
11745 }
(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.

```

11746 \cs_new:Npn \__fp_add_inf_o:Nww
11747   #1 \s__fp \__fp_chk:w 2 #2 #3; \s__fp \__fp_chk:w 2 #4
11748 {
11749   \if_meaning:w #1 #2
11750   \__fp_case_return_i_o:ww
11751   \else:
11752     \__fp_case_use:nw
11753     {
11754       \if_meaning:w #1 #4
11755       \exp_after:wN \__fp_invalid_operation_o:Nww
11756       \exp_after:wN +
11757       \else:
11758       \exp_after:wN \__fp_invalid_operation_o:Nww
11759       \exp_after:wN -
11760     \fi:
11761     }
11762   \fi:
11763   \s__fp \__fp_chk:w 2 #2 #3;
11764   \s__fp \__fp_chk:w 2 #4
11765 }
(End definition for \__fp_add_inf_o:Nww.)

```

```

\__fp_add_normal_o:Nww \__fp_add_normal_o:Nww <sign2> \s__fp \__fp_chk:w 1 <sign1> <exp1>
<body1> ; \s__fp \__fp_chk:w 1 <initial sign2> <exp2> <body2> ;

```

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```

11766 \cs_new:Npn \__fp_add_normal_o:Nww #1 \s__fp \__fp_chk:w 1 #2
11767 {
11768   \if_meaning:w #1#2
11769     \exp_after:wN \__fp_add_npos_o:NnwNnw
11770   \else:
11771     \exp_after:wN \__fp_sub_npos_o:NnwNnw
11772   \fi:
11773   #2
11774 }

```

(End definition for __fp_add_normal_o:Nww.)

33.2.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

```

\__fp_add_npos_o:NnwNnw \__fp_add_npos_o:NnwNnw <sign1> <exp1> <body1> ; \s__fp \__fp_chk:w 1
<initial sign2> <exp2> <body2> ;

```

Since we are doing an addition, the final sign is $\langle sign_1 \rangle$. Start an $__int_eval:w$, responsible for computing the exponent: the result, and the $\langle final\ sign \rangle$ are then given to $__fp_sanitize:Nw$ which checks for overflow. The exponent is computed as the largest exponent #2 or #5, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by $__fp_add_big_i:wNww$ or $__fp_add_big_ii:wNww$. We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```

11775 \cs_new:Npn \__fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp \__fp_chk:w 1 #4 #5
11776 {
11777   \exp_after:wN \__fp_sanitize:Nw
11778   \exp_after:wN #1
11779   \int_use:N \__int_eval:w
11780   \if_int_compare:w #2 > #5 \exp_stop_f:
11781     #2
11782     \exp_after:wN \__fp_add_big_i_o:wNww \__int_value:w -
11783   \else:
11784     #5
11785     \exp_after:wN \__fp_add_big_ii_o:wNww \__int_value:w
11786   \fi:
11787   \__int_eval:w #5 - #2 ; #1 #3;
11788 }

```

(End definition for __fp_add_npos_o:NnwNnw.)

```

\__fp_add_big_i_o:wNww \__fp_add_big_i_o:wNww <shift> ; <final sign> <body1> ; <body2> ;
\__fp_add_big_ii_o:wNww Shift the significand of the small number, then add with \__fp_add_significand_
o:NnnwnnnnnN.

```

```

11789 \cs_new:Npn \__fp_add_big_i_o:wNww #1; #2 #3; #4;

```

```

11790 {
11791   \_fp_decimate:nNnnnn {#1}
11792   \_fp_add_significand_o:NnnwnnnnN
11793   #4
11794   #3
11795   #2
11796 }
11797 \cs_new:Npn \_fp_add_big_ii_o:wNww #1; #2 #3; #4;
11798 {
11799   \_fp_decimate:nNnnnn {#1}
11800   \_fp_add_significand_o:NnnwnnnnN
11801   #3
11802   #4
11803   #2
11804 }

```

(End definition for _fp_add_big_i_o:wNww. This function is documented on page ??.)

```

\_fp_add_significand_o:NnnwnnnnN   \_fp_add_significand_o:NnnwnnnnN <rounding digit> {\langle Y'_1 \rangle} {\langle Y'_2 \rangle} <extra-digits>
\_fp_add_significand_pack:NNNNNNN ; {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} <final sign>
\_fp_add_significand_test_o:N

```

To round properly, we must know at which digit the rounding should occur. This requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as $0.99 \dots 95 \rightarrow 1.00 \dots 0$, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```

11805 \cs_new:Npn \_fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
11806 {
11807   \exp_after:wN \_fp_add_significand_test_o:N
11808   \int_use:N \_int_eval:w 1#5#6 + #2
11809   \exp_after:wN \_fp_add_significand_pack:NNNNNNN
11810   \int_use:N \_int_eval:w 1#7#8 + #3 ; #1
11811 }
11812 \cs_new:Npn \_fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
11813 {
11814   \if_meaning:w 2 #1
11815   + \c_one
11816   \fi:
11817   ; #2 #3 #4 #5 #6 #7 ;
11818 }
11819 \cs_new:Npn \_fp_add_significand_test_o:N #1
11820 {
11821   \if_meaning:w 2 #1
11822   \exp_after:wN \_fp_add_significand_carry_o:wwwNN
11823   \else:
11824   \exp_after:wN \_fp_add_significand_no_carry_o:wwwNN
11825   \fi:
11826 }

```

(End definition for _fp_add_significand_o:NnnwnnnnN. This function is documented on page ??.)

```

\__fp_add_significand_no_carry_o:wwwNN \__fp_add_significand_no_carry_o:wwwNN <8d> ; <6d> ; <2d> ; <rounding
digit> <sign>

```

If there's no carry, grab all the digits again and round. The packing function `__fp_basics_pack_high:NNNNw` takes care of the case where rounding brings a carry.

```

11827 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
11828   #1; #2; #3#4 ; #5#6
11829   {
11830     \exp_after:wN \__fp_basics_pack_high:NNNNw
11831     \int_use:N \__int_eval:w 1 #1
11832     \exp_after:wN \__fp_basics_pack_low:NNNNw
11833     \int_use:N \__int_eval:w 1 #2 #3#4
11834     + \__fp_round:NNN #6 #4 #5
11835     \exp_after:wN ;
11836   }
(End definition for \__fp_add_significand_no_carry_o:wwwNN.)

```

```

\__fp_add_significand_carry_o:wwwNN \__fp_add_significand_carry_o:wwwNN <8d> ; <6d> ; <2d> ; <rounding
digit> <sign>

```

The case where there is a carry is very similar. Rounding can even raise the first digit from 1 to 2, but we don't care.

```

11837 \cs_new:Npn \__fp_add_significand_carry_o:wwwNN
11838   #1; #2; #3#4; #5#6
11839   {
11840     + \c_one
11841     \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
11842     \int_use:N \__int_eval:w 1 1 #1
11843     \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
11844     \int_use:N \__int_eval:w 1 #2#3 +
11845     \exp_after:wN \__fp_round:NNN
11846     \exp_after:wN #6
11847     \exp_after:wN #3
11848     \__int_value:w \__fp_round_digit:Nw #4 #5 ;
11849     \exp_after:wN ;
11850   }
(End definition for \__fp_add_significand_carry_o:wwwNN.)

```

33.2.3 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:Nwnnw <initial sign2> <exp2> <body2> ;
\__fp_sub_npos_ii_o:Nwnnw

```

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:Nwnnw` with the opposite of $\langle sign_1 \rangle$.

```

11851 \cs_new:Npn \__fp_sub_npos_o:NnwNnw #1#2#3; \s__fp \__fp_chk:w 1 #4#5#6;
11852   {
11853     \if_case:w \__fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
11854     \exp_after:wN \__fp_sub_eq_o:Nwnnw

```



```

11855 \or:
11856 \exp_after:wN \_fp_sub_npos_i_o:Nnwnw
11857 \else:
11858 \exp_after:wN \_fp_sub_npos_ii_o:Nnwnw
11859 \fi:
11860 #1 {#2} #3; {#5} #6;
11861 }
11862 \cs_new:Npn \_fp_sub_eq_o:Nnwnw #1#2; #3; { \exp_after:wN \c_zero_fp }
11863 \cs_new:Npn \_fp_sub_npos_ii_o:Nnwnw #1 #2; #3;
11864 {
11865 \exp_after:wN \_fp_sub_npos_i_o:Nnwnw
11866 \int_use:N \_int_eval:w \c_two - #1 \_int_eval_end:
11867 #3; #2;
11868 }

```

(End definition for _fp_sub_npos_o:Nnwnw. This function is documented on page ??.)

_fp_sub_npos_i_o:Nnwnw

After the computation is done, _fp_sanitize:Nw checks for overflow/underflow. It expects the $\langle final\ sign \rangle$ and the $\langle exponent \rangle$ (delimited by ;). Start an integer expression for the exponent, which starts with the exponent of the largest number, and may be decreased if the two numbers are very close. If the two numbers have the same exponent, call the **near** auxiliary. Otherwise, decimate y , then call the **far** auxiliary to evaluate the difference between the two significands. Note that we decimate by 1 less than one could expect.

```

11869 \cs_new:Npn \_fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
11870 {
11871 \exp_after:wN \_fp_sanitize:Nw
11872 \exp_after:wN #1
11873 \int_use:N \_int_eval:w
11874 #2
11875 \if_int_compare:w #2 = #4 \exp_stop_f:
11876 \exp_after:wN \_fp_sub_back_near_o:nnnnnnnnN
11877 \else:
11878 \exp_after:wN \_fp_decimate:nNnnnn \exp_after:wN
11879 { \int_use:N \_int_eval:w #2 - #4 - \c_one \exp_after:wN }
11880 \exp_after:wN \_fp_sub_back_far_o:NnnwnnnnnN
11881 \fi:
11882 #5
11883 #3
11884 #1
11885 }

```

(End definition for _fp_sub_npos_i_o:Nnwnw.)

_fp_sub_back_near_o:nnnnnnnnN
_fp_sub_back_near_pack:NNNNNNw
_fp_sub_back_near_after:wNNNNw

_fp_sub_back_near_o:nnnnnnnnN $\{\langle Y_1 \rangle\} \{\langle Y_2 \rangle\} \{\langle Y_3 \rangle\} \{\langle Y_4 \rangle\} \{\langle X_1 \rangle\}$
 $\{\langle X_2 \rangle\} \{\langle X_3 \rangle\} \{\langle X_4 \rangle\} \langle final\ sign \rangle$

In this case, the subtraction is exact, so we discard the $\langle final\ sign \rangle$ #9. The very large shifts of 10^9 and $1.1 \cdot 10^9$ are unnecessary here, but allow the auxiliaries to be reused later. Each integer expression produces a 10 digit result. If the resulting 16 digits start with a 0, then we need to shift the group, padding with trailing zeros.

```

11886 \cs_new:Npn \__fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
11887 {
11888   \exp_after:wN \__fp_sub_back_near_after:wNNNNw
11889   \int_use:N \__int_eval:w 10#5#6 - #1#2 - \c_eleven
11890   \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
11891   \int_use:N \__int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
11892 }
11893 \cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
11894 { + #1#2 ; {#3#4#5#6} {#7} ; }
11895 \cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
11896 {
11897   \if_meaning:w 0 #1
11898   \exp_after:wN \__fp_sub_back_shift:wnnnn
11899   \fi:
11900   ; {#1#2#3#4} {#5}
11901 }

```

(End definition for __fp_sub_back_near_o:nnnnnnnnN. This function is documented on page ??.)

```

\__fp_sub_back_shift:wnnnn
\__fp_sub_back_shift_ii:ww
  \__fp_sub_back_shift_iii:NNNNNNNNw
    \__fp_sub_back_shift_iv:nnnnw

```

__fp_sub_back_shift:wnnnn ; { $\langle Z_1 \rangle$ } { $\langle Z_2 \rangle$ } { $\langle Z_3 \rangle$ } { $\langle Z_4 \rangle$ } ;
This function is called with $\langle Z_1 \rangle \leq 999$. Act with \number to trim leading zeros from $\langle Z_1 \rangle$ $\langle Z_2 \rangle$ (we don't do all four blocks at once, since non-zero blocks would then overflow 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.

```

11902 \cs_new:Npn \__fp_sub_back_shift:wnnnn ; #1#2
11903 {
11904   \exp_after:wN \__fp_sub_back_shift_ii:ww
11905   \__int_value:w #1 #2 0 ;
11906 }
11907 \cs_new:Npn \__fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
11908 {
11909   \if_meaning:w @ #1 @
11910   - \c_seven
11911   - \exp_after:wN \use_i:nnn
11912   \exp_after:wN \__fp_sub_back_shift_iii:NNNNNNNNw
11913   \__int_value:w #2#3 0 ~ 123456789;
11914   \else:
11915   - \__fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
11916   \fi:
11917   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNNN
11918   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNNN
11919   \exp_after:wN \__fp_sub_back_shift_iv:nnnnw
11920   \exp_after:wN ;
11921   \__int_value:w
11922   #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
11923 }

```

```

11924 \cs_new:Npn \__fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
11925 \cs_new:Npn \__fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }
(End definition for \__fp_sub_back_shift:wnnnn. This function is documented on page ??.)

```

```

\__fp_sub_back_far_o:NnnwnnnnN \__fp_sub_back_far_o:NnnwnnnnN <rounding> {<Y'1>} {<Y'2>} <extra-digits>
; {<X1>} {<X2>} {<X3>} {<X4>} <final sign>

```

If the difference is greater than $10^{\langle expo_x \rangle}$, call the `very_far` auxiliary. If the result is less than $10^{\langle expo_x \rangle}$, call the `not_far` auxiliary. If it is too close a call to know yet, namely if $1\langle Y'_1 \rangle \langle Y'_2 \rangle = \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle 0$, then call the `quite_far` auxiliary. We use the odd combination of space and semi-colon delimiters to allow the `not_far` auxiliary to grab each piece individually, the `very_far` auxiliary to use `__fp_pack_eight:wNNNNNNNN`, and the `quite_far` to ignore the significands easily (using the `;` delimiter).

```

11926 \cs_new:Npn \__fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
11927 {
11928   \if_case:w
11929     \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
11930     \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
11931     \c_zero
11932   \else:
11933     \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: \c_one
11934   \fi:
11935   \else:
11936     \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: \c_one
11937   \fi:
11938     \exp_after:wN \__fp_sub_back_quite_far_o:wwNN
11939   \or:   \exp_after:wN \__fp_sub_back_very_far_o:wwwNNN
11940   \else: \exp_after:wN \__fp_sub_back_not_far_o:wwwNNN
11941   \fi:
11942   #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
11943 }

```

(End definition for `__fp_sub_back_far_o:NnnwnnnnN`.)

```

\__fp_sub_back_quite_far_o:wwNN
\__fp_sub_back_quite_far_ii:NN

```

The easiest case is when $x - y$ is extremely close to a power of 10, namely the first digit of x is 1, and all others vanish when subtracting y . Then the `<rounding>` #3 and the `<final sign>` #4 control whether we get 1 or 0.9999999999999999. In the usual round-to-nearest mode, we will get 1 whenever the `<rounding>` digit is less than or equal to 5 (remember that the `<rounding>` digit is only equal to 5 if there was no further non-zero digit).

```

11944 \cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
11945 {
11946   \exp_after:wN \__fp_sub_back_quite_far_ii:NN
11947   \exp_after:wN #3
11948   \exp_after:wN #4
11949 }
11950 \cs_new:Npn \__fp_sub_back_quite_far_ii:NN #1#2
11951 {
11952   \if_case:w \__fp_round_neg:NNN #2 0 #1
11953   \exp_after:wN \use_i:nn
11954   \else:

```

```

11955     \exp_after:wN \use_ii:nn
11956     \fi:
11957     { ; {1000} {0000} {0000} {0000} ; }
11958     { - \c_one ; {9999} {9999} {9999} {9999} ; }
11959 }

```

(End definition for `__fp_sub_back_quite_far_o:wwwNN`. This function is documented on page ??.)

`__fp_sub_back_not_far_o:wwwNN`

In the present case, x and y have different exponents, but y is large enough that $x - y$ has a smaller exponent than x . Decrement the exponent (with `- \c_one`). Then proceed in a way similar to the `near` auxiliaries seen earlier, but multiplying x by 10 (`#30` and `#40` below), and with the added quirk that the *rounding* digit has to be taken into account. Namely, we may have to decrease the result by one unit if `__fp_round_neg:NNN` returns 1. This function expects the *final sign* `#6`, the last digit of `1100000000+#40-#2`, and the *rounding* digit. Instead of redoing the computation for the second argument, we note that `__fp_round_neg:NNN` only cares about its parity, which is identical to that of the last digit of `#2`.

```

11960 \cs_new:Npn \__fp_sub_back_not_far_o:wwwNN #1 ~ #2; #3 ~ #4; #5#6
11961 {
11962   - \c_one
11963   \exp_after:wN \__fp_sub_back_near_after:wNNNNw
11964   \int_use:N \__int_eval:w 1#30 - #1 - \c_eleven
11965   \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
11966   \int_use:N \__int_eval:w 11 0000 0000 + #40 - #2
11967   - \exp_after:wN \__fp_round_neg:NNN
11968   \exp_after:wN #6
11969   \use_none:nnnnnnn #2 #5
11970   \exp_after:wN ;
11971 }

```

(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.

```

11972 \cs_new:Npn \__fp_sub_back_very_far_o:wwwNN #1#2#3#4#5#6#7
11973 {
11974   \__fp_pack_eight:wNNNNNNNN
11975   \__fp_sub_back_very_far_ii_o:nnNwwNN
11976   { 0 #1#2#3 #4#5#6#7 }
11977   ;
11978 }
11979 \cs_new:Npn \__fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
11980 {
11981   \exp_after:wN \__fp_basics_pack_high:NNNNNw
11982   \int_use:N \__int_eval:w 1#4 - #1 - \c_one

```

```

11983 \exp_after:wN \__fp_basics_pack_low:NNNNw
11984 \int_use:N \__int_eval:w 2#5 - #2
11985 - \exp_after:wN \__fp_round_neg:NNN
11986 \exp_after:wN #7
11987 \__int_value:w
11988 \if_int_odd:w \__int_eval:w #5 - #2 \__int_eval_end:
11989 1 \else: 2 \fi:
11990 \__int_value:w \__fp_round_digit:Nw #3 #6 ;
11991 \exp_after:wN ;
11992 }

```

(End definition for __fp_sub_back_very_far_o:wwwNN. This function is documented on page ??.)

33.3 Multiplication

33.3.1 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.

```

11993 \cs_new_nopar:cpn { __fp*_o:ww }
11994 {
11995   \__fp_mul_cases_o:NnNnw
11996   *
11997   { - \c_two + }
11998   \__fp_mul_npos_o:Nww
11999   { }
12000 }

```

(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.

```

12001 \cs_new:Npn \__fp_mul_cases_o:NnNnw
12002 #1#2#3#4 \s__fp \__fp_chk:w #5#6#7; \s__fp \__fp_chk:w #8#9
12003 {
12004   \if_case:w \__int_eval:w
12005     \if_int_compare:w #5 #8 = \c_eleven
12006     \c_one

```

```

12007         \else:
12008             \if_meaning:w 3 #8
12009             \c_three
12010         \else:
12011             \if_meaning:w 3 #5
12012             \c_two
12013         \else:
12014             \if_int_compare:w #5 #8 = \c_ten
12015             \c_nine #2 - \c_two
12016         \else:
12017             (#5 #2 #8) / \c_two * \c_two + \c_seven
12018         \fi:
12019     \fi:
12020 \fi:
12021 \fi:
12022     \if_meaning:w #6 #9 - \c_one \fi:
12023     \__int_eval_end:
12024     \__fp_case_use:nw { #3 0 }
12025 \or: \__fp_case_use:nw { #3 2 }
12026 \or: \__fp_case_return_i_o:ww
12027 \or: \__fp_case_return_ii_o:ww
12028 \or: \__fp_case_return_o:Nww \c_zero_fp
12029 \or: \__fp_case_return_o:Nww \c_minus_zero_fp
12030 \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
12031 \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
12032 \or: \__fp_case_return_o:Nww \c_inf_fp
12033 \or: \__fp_case_return_o:Nww \c_minus_inf_fp
12034 #4
12035 \fi:
12036 \s__fp \__fp_chk:w #5 #6 #7;
12037 \s__fp \__fp_chk:w #8 #9
12038 }
(End definition for \__fp_mul_cases_o:nNnnww.)

```

33.3.2 Absolute multiplication

In this subsection, we perform the multiplication of two positive normal numbers.

```

\__fp_mul_npos_o:Nww \__fp_mul_npos_o:Nww  $\langle final\ sign \rangle$  \s__fp \__fp_chk:w 1  $\langle sign_1 \rangle$  { $\langle exp_1 \rangle$ }
 $\langle body_1 \rangle$  ; \s__fp \__fp_chk:w 1  $\langle sign_2 \rangle$  { $\langle exp_2 \rangle$ }  $\langle body_2 \rangle$  ;

```

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 $\langle final\ sign \rangle$ is needed to do the rounding properly in the significand computation. We setup the post-expansion here, triggered by `__fp_mul_significand_o:nnnnNnnnn`.

```

12039 \cs_new:Npn \__fp_mul_npos_o:Nww
12040 #1 \s__fp \__fp_chk:w #2 #3 #4 #5 ; \s__fp \__fp_chk:w #6 #7 #8 #9 ;
12041 {
12042     \exp_after:wN \__fp_sanitize:Nw

```

```

12043 \exp_after:wN #1
12044 \int_use:N \__int_eval:w
12045 #4 + #8
12046 \__fp_mul_significand_o:nnnnNnnnn #5 #1 #9
12047 }
(End definition for \__fp_mul_npos_o:Nww.)

```

```

\__fp_mul_significand_o:nnnnNnnnn \__fp_mul_significand_o:nnnnNnnnn  $\{ \langle X_1 \rangle \} \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \} \langle sign \rangle$ 
\__fp_mul_significand_drop:NNNNw  $\{ \langle Y_1 \rangle \} \{ \langle Y_2 \rangle \} \{ \langle Y_3 \rangle \} \{ \langle Y_4 \rangle \}$ 
\__fp_mul_significand_keep:NNNNw

```

Note the three semicolons at the end of the definition. One is for the last `__fp_mul_significand_drop:NNNNw`; 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:NNNNw`.

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`.

```

12048 \cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
12049 {
12050   \exp_after:wN \__fp_mul_significand_test_f:NNN
12051   \exp_after:wN #5
12052   \int_use:N \__int_eval:w 99990000 + #1*#6 +
12053   \exp_after:wN \__fp_mul_significand_keep:NNNNw
12054   \int_use:N \__int_eval:w 99990000 + #1*#7 + #2*#6 +
12055   \exp_after:wN \__fp_mul_significand_keep:NNNNw
12056   \int_use:N \__int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
12057   \exp_after:wN \__fp_mul_significand_drop:NNNNw
12058   \int_use:N \__int_eval:w 99990000 + #1*#9 + #2*#8 + #3*#7 + #4*#6 +
12059   \exp_after:wN \__fp_mul_significand_drop:NNNNw
12060   \int_use:N \__int_eval:w 99990000 + #2*#9 + #3*#8 + #4*#7 +
12061   \exp_after:wN \__fp_mul_significand_drop:NNNNw
12062   \int_use:N \__int_eval:w 99990000 + #3*#9 + #4*#8 +
12063   \exp_after:wN \__fp_mul_significand_drop:NNNNw
12064   \int_use:N \__int_eval:w 100000000 + #4*#9 ;
12065   ; \exp_after:wN ;
12066 }
12067 \cs_new:Npn \__fp_mul_significand_drop:NNNNw #1#2#3#4#5 #6;
12068 { #1#2#3#4#5 ; + #6 }
12069 \cs_new:Npn \__fp_mul_significand_keep:NNNNw #1#2#3#4#5 #6;
12070 { #1#2#3#4#5 ; #6 ; }

```

(End definition for `__fp_mul_significand_o:nnnnNnnnn`. This function is documented on page ??.)

```

\__fp_mul_significand_test_f:NNN \__fp_mul_significand_test_f:NNN  $\langle sign \rangle$  1  $\langle digits\ 1-8 \rangle$  ;  $\langle digits\ 9-12 \rangle$  ;
 $\langle digits\ 13-16 \rangle$  ; +  $\langle digits\ 17-20 \rangle$  +  $\langle digits\ 21-24 \rangle$  +  $\langle digits\ 25-28 \rangle$  +  $\langle digits\ 29-32 \rangle$  ; \exp_after:wN ;

```

If the $\langle digit\ 1 \rangle$ is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if $\langle digit\ 1 \rangle$ is zero, we care about digits 17 and 18, and whether further digits are zero.

```

12071 \cs_new:Npn \__fp_mul_significand_test_f:NNN #1 #2 #3
12072 {
12073   \if_meaning:w 0 #3
12074     \exp_after:wN \__fp_mul_significand_small_f:NNwwwN
12075   \else:
12076     \exp_after:wN \__fp_mul_significand_large_f:NwwNNNN
12077   \fi:
12078   #1 #3
12079 }
(End definition for \__fp_mul_significand_test_f:NNN.)

```

$\backslash_fp_mul_significand_large_f:NwwNNNN$ In this branch, $\langle digit\ 1 \rangle$ is non-zero. The result is thus $\langle digits\ 1-16 \rangle$, plus some rounding which depends on the digits 16, 17, and whether all subsequent digits are zero or not. Here, $\backslash_fp_round_digit:Nw$ takes digits 17 and further (as an integer expression), and replaces it by a $\langle rounding\ digit \rangle$, suitable for $\backslash_fp_round:NNN$.

```

12080 \cs_new:Npn \__fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
12081 {
12082   \exp_after:wN \__fp_basics_pack_high:NNNNNw
12083   \int_use:N \__int_eval:w 1#2
12084   \exp_after:wN \__fp_basics_pack_low:NNNNNw
12085   \int_use:N \__int_eval:w 1#3#4#5#6#7
12086   + \exp_after:wN \__fp_round:NNN
12087     \exp_after:wN #1
12088     \exp_after:wN #7
12089     \__int_value:w \__fp_round_digit:Nw
12090 }
(End definition for \__fp_mul_significand_large_f:NwwNNNN.)

```

$\backslash_fp_mul_significand_small_f:NNwwwN$ In this branch, $\langle digit\ 1 \rangle$ is zero. Our result will thus be $\langle digits\ 2-17 \rangle$, plus some rounding which depends on the digits 17, 18, and whether all subsequent digits are zero or not. The 8 digits 1#3 are followed, after expansion of the `small_pack` auxiliary, by the next digit, to form a 9 digit number.

```

12091 \cs_new:Npn \__fp_mul_significand_small_f:NNwwwN #1 #2#3; #4#5; #6; + #7
12092 {
12093   - \c_one
12094   \exp_after:wN \__fp_basics_pack_high:NNNNNw
12095   \int_use:N \__int_eval:w 1#3#4
12096   \exp_after:wN \__fp_basics_pack_low:NNNNNw
12097   \int_use:N \__int_eval:w 1#5#6#7
12098   + \exp_after:wN \__fp_round:NNN
12099     \exp_after:wN #1
12100     \exp_after:wN #7
12101     \__int_value:w \__fp_round_digit:Nw
12102 }
(End definition for \__fp_mul_significand_small_f:NNwwwN.)

```


33.4 Division

33.4.1 Signs, and special numbers

Time is now ripe to tackle the hardest of the four elementary operations: division.

`__fp/_o:ww` Filtering special floating point is very similar to what we did for multiplications, with a few variations. Invalid operation exceptions display / rather than *. In the formula for dispatch, we replace `\c_two +` by `-`. The case of normal numbers is treated using `__fp_div_npos_o:Nww` rather than `__fp_mul_npos_o:Nww`. There are two additional cases: if the first operand is normal and the second is a zero, then the division by zero exception is raised: cases 10 and 11 of the `\if_case:w` construction in `__fp_mul_cases_o:NnNnw` are provided as the fourth argument here.

```

12103 \cs_new_nopar:cpn { __fp/_o:ww }
12104 {
12105     \__fp_mul_cases_o:NnNnw
12106     /
12107     { - }
12108     \__fp_div_npos_o:Nww
12109     {
12110         \or:
12111         \__fp_case_use:nw
12112         { \__fp_division_by_zero_o:NNnw \c_inf_fp / }
12113         \or:
12114         \__fp_case_use:nw
12115         { \__fp_division_by_zero_o:NNnw \c_minus_inf_fp / }
12116     }
12117 }

```

(End definition for `__fp/_o:ww`.)

```

\__fp_div_npos_o:Nww \__fp_div_npos_o:Nww <final sign> \s__fp \__fp_chk:w 1 <sign_A> {<exp A>}
{<A1>} {<A2>} {<A3>} {<A4>} ; \s__fp \__fp_chk:w 1 <sign_Z> {<exp Z>}
{<Z1>} {<Z2>} {<Z3>} {<Z4>} ;

```

We want to compute A/Z . As for multiplication, `__fp_sanitizew` checks for overflow or underflow; we provide it with the *<final sign>*, and an integer expression in which we compute the exponent. We set up the arguments of `__fp_div_significand_i_o:wnnw`, namely an integer $\langle y \rangle$ obtained by adding 1 to the first 5 digits of Z (explanation given soon below), then the four $\{A_i\}$, then the four $\{Z_i\}$, a semi-colon, and the *<final sign>*, used for rounding at the end.

```

12118 \cs_new:Npn \__fp_div_npos_o:Nww
12119     #1 \s__fp \__fp_chk:w 1 #2 #3 #4 ; \s__fp \__fp_chk:w 1 #5 #6 #7#8#9;
12120 {
12121     \exp_after:wN \__fp_sanitizew
12122     \exp_after:wN #1
12123     \int_use:N \__int_eval:w
12124     #3 - #6
12125     \exp_after:wN \__fp_div_significand_i_o:wnnw
12126     \int_use:N \__int_eval:w #7 \use_i:nnnn #8 + \c_one ;
12127     #4

```

```

12128      {\#7}{\#8}\#9 ;
12129      #1
12130    }
(End definition for \_fp_div_npos_o:Nww.)

```

33.4.2 Work plan

In this subsection, we explain how to avoid overflowing \TeX 's integers when performing the division of two positive normal numbers.

We are given two numbers, $A = 0.A_1A_2A_3A_4$ and $Z = 0.Z_1Z_2Z_3Z_4$, in blocks of 4 digits, and we know that the first digits of A_1 and of Z_1 are non-zero. To compute A/Z , we proceed as follows.

- Find an integer $Q_A \simeq 10^4 A/Z$.
- Replace A by $B = 10^4 A - Q_A Z$.
- Find an integer $Q_B \simeq 10^4 B/Z$.
- Replace B by $C = 10^4 B - Q_B Z$.
- Find an integer $Q_C \simeq 10^4 C/Z$.
- Replace C by $D = 10^4 C - Q_C Z$.
- Find an integer $Q_D \simeq 10^4 D/Z$.
- Consider $E = 10^4 D - Q_D Z$, and ensure correct rounding.

The result is then $Q = 10^{-4}Q_A + 10^{-8}Q_B + 10^{-12}Q_C + 10^{-16}Q_D + \text{rounding}$. Since the Q_i are integers, B , C , D , and E are all exact multiples of 10^{-16} , in other words, computing with 16 digits after the decimal separator yields exact results. The problem will be overflow: in general B , C , D , and E may be greater than 1.

Unfortunately, things are not as easy as they seem. In particular, we want all intermediate steps to be positive, since negative results would require extra calculations at the end. This requires that $Q_A \leq 10^4 A/Z$ etc. A reasonable attempt would be to define Q_A as

$$\backslash\text{int_eval:n}\left\{\frac{A_1A_2}{Z_1+1}-1\right\}\leq 10^4\frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that $\varepsilon\text{-TeX}$'s $\backslash\text{int_eval:w}$ rounds divisions instead of truncating (really, $1/2$ would be sufficient, but we work with integers). We add 1 to Z_1 because $Z_1 \leq 10^4 Z < Z_1 + 1$ and we need Q_A to be an underestimate. However, we are now underestimating Q_A too much: it can be wrong by up to 100, for instance when $Z = 0.1$ and $A \simeq 1$. Then B could take values up to 10 (maybe more), and a few steps down the line, we would run into arithmetic overflow, since \TeX can only handle integers less than roughly $2 \cdot 10^9$.

A better formula is to take

$$Q_A = \backslash\text{int_eval:n}\left\{\frac{10 \cdot A_1A_2}{\lfloor 10^{-3} \cdot Z_1Z_2 \rfloor + 1} - 1\right\}.$$

This is always less than $10^9 A / (10^5 Z)$, as we wanted. In words, we take the 5 first digits of Z into account, and the 8 first digits of A , using 0 as a 9-th digit rather than the true digit for efficiency reasons. We shall prove that using this formula to define all the Q_i avoids any overflow. For convenience, let us denote

$$y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1,$$

so that, taking into account the fact that ε -TeX rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1 A_2 0}{y} - \frac{3}{2}. \end{aligned}$$

Note that $10^4 < y \leq 10^5$, and $999 \leq Q_A \leq 99989$. Also note that this formula does not cause an overflow as long as $A < (2^{31} - 1)/10^9 \simeq 2.147 \dots$, since the numerator involves an integer slightly smaller than $10^9 A$.

Let us bound B :

$$\begin{aligned} 10^5 B &= A_1 A_2 0 + 10 \cdot 0.A_3 A_4 - 10 \cdot Z_1.Z_2 Z_3 Z_4 \cdot Q_A \\ &< A_1 A_2 0 \cdot \left(1 - 10 \cdot \frac{Z_1.Z_2 Z_3 Z_4}{y} \right) + \frac{3}{2} \cdot 10 \cdot Z_1.Z_2 Z_3 Z_4 + 10 \\ &\leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1.Z_2 Z_3 Z_4)}{y} + \frac{3}{2} y + 10 \\ &\leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2} y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y. \end{aligned}$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$\begin{aligned} 10^5 B &< 10^9 A / y + 1.6y, \\ 10^5 C &< 10^9 B / y + 1.6y, \\ 10^5 D &< 10^9 C / y + 1.6y, \\ 10^5 E &< 10^9 D / y + 1.6y. \end{aligned}$$

The goal is now to prove that none of B , C , D , and E can go beyond $(2^{31} - 1)/10^9 = 2.147 \dots$.

Combining the various inequalities together with $A < 1$, we get

$$\begin{aligned} 10^5 B &< 10^9 / y + 1.6y, \\ 10^5 C &< 10^{13} / y^2 + 1.6(y + 10^4), \\ 10^5 D &< 10^{17} / y^3 + 1.6(y + 10^4 + 10^8 / y), \\ 10^5 E &< 10^{21} / y^4 + 1.6(y + 10^4 + 10^8 / y + 10^{12} / y^2). \end{aligned}$$

All of those bounds are convex functions of y (since every power of y involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range $10^4 < y \leq 10^5$. Thus,

$$\begin{aligned} 10^5 B &< \max(1.16 \cdot 10^5, 1.7 \cdot 10^5), \\ 10^5 C &< \max(1.32 \cdot 10^5, 1.77 \cdot 10^5), \\ 10^5 D &< \max(1.48 \cdot 10^5, 1.777 \cdot 10^5), \\ 10^5 E &< \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5). \end{aligned}$$

All of those bounds are less than $2.147 \cdot 10^5$, and we are thus within \TeX 's bounds in all cases!

We will later need to have a bound on the Q_i . Their definitions imply that $Q_A < 10^9 A/y - 1/2 < 10^5 A$ and similarly for the other Q_i . Thus, all of them are less than 177770.

The last step is to ensure correct rounding. We have

$$A/Z = \sum_{i=1}^4 (10^{-4i} Q_i) + 10^{-16} E/Z$$

exactly. Furthermore, we know that the result will be in $[0.1, 10)$, hence will be rounded to a multiple of 10^{-16} or of 10^{-15} , so we only need to know the integer part of E/Z , and a “rounding” digit encoding the rest. Equivalently, we need to find the integer part of $2E/Z$, and determine whether it was an exact integer or not (this serves to detect ties). Since

$$\frac{2E}{Z} = 2 \frac{10^5 E}{10^5 Z} \leq 2 \frac{10^5 E}{10^4} < 36,$$

this integer part is between 0 and 35 inclusive. We let $\varepsilon\text{-}\text{\TeX}$ round

$$P = \backslash\text{int_eval:n} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

which differs from $2E/Z$ by at most

$$\frac{1}{2} + 2 \left| \frac{E}{Z} - \frac{E}{10^{-8} Z_1 Z_2} \right| + 2 \left| \frac{10^8 E - E_1 E_2}{Z_1 Z_2} \right| < 1,$$

($1/2$ comes from $\varepsilon\text{-}\text{\TeX}$'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.

33.4.3 Implementing the significand division

`_fp_div_significand_i_o:wnnw` $_fp_div_significand_i_o:wnnw \langle y \rangle ; \{ \langle A_1 \rangle \} \{ \langle A_2 \rangle \} \{ \langle A_3 \rangle \} \{ \langle A_4 \rangle \}$
 $\{ \langle Z_1 \rangle \} \{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \} \{ \langle Z_4 \rangle \} ; \langle sign \rangle$

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace $\langle A_1 \rangle$ and $\langle A_2 \rangle$, and prepare the $\langle continuation \rangle$ arguments for 4 consecutive calls to `_fp_div_significand_calc:wnnnnnnnn`. Each of these calls will need $\langle y \rangle$ ($\#1$), and it turns out that we need post-expansion there, hence the `_int_value:w`. Here, $\#4$ is six brace groups, which give the six first n-type arguments of the `calc` function.

```

12131 \cs_new:Npn \_fp_div_significand_i_o:wnnw #1 ; #2#3 #4 ;
12132 {
12133   \exp_after:wN \_fp_div_significand_test_o:w
12134   \int_use:N \_int_eval:w
12135   \exp_after:wN \_fp_div_significand_calc:wnnnnnnnn
12136   \int_use:N \_int_eval:w 999999 + #2 #3 0 / #1 ;
12137   #2 #3 ;
12138   #4
12139   { \exp_after:wN \_fp_div_significand_ii:wnn \_int_value:w #1 }
12140   { \exp_after:wN \_fp_div_significand_ii:wnn \_int_value:w #1 }
12141   { \exp_after:wN \_fp_div_significand_ii:wnn \_int_value:w #1 }
12142   { \exp_after:wN \_fp_div_significand_iii:wnnnnnn \_int_value:w #1 }
12143 }

```

(End definition for `_fp_div_significand_i_o:wnnw`.)

`_fp_div_significand_calc:wnnnnnnnn` $_fp_div_significand_calc:wnnnnnnnn \langle 10^6 + Q_A \rangle ; \langle A_1 \rangle \langle A_2 \rangle ; \{ \langle A_3 \rangle \}$
`_fp_div_significand_calc_i:wnnnnnnnn` $\{ \langle A_4 \rangle \} \{ \langle Z_1 \rangle \} \{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \} \{ \langle Z_4 \rangle \} \{ \langle continuation \rangle \}$
`_fp_div_significand_calc_ii:wnnnnnnnn` expands to

$\langle 10^6 + Q_A \rangle \langle continuation \rangle ; \langle B_1 \rangle \langle B_2 \rangle ; \{ \langle B_3 \rangle \} \{ \langle B_4 \rangle \} \{ \langle Z_1 \rangle \} \{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \}$
 $\{ \langle Z_4 \rangle \}$

where $B = 10^4 A - Q_A \cdot Z$. This function is also used to compute C , D , E (with the input shifted accordingly), and is used in `l3fp-expo`.

We know that $0 < Q_A < 1.8 \cdot 10^5$, so the product of Q_A with each Z_i is within $\text{T}_{\text{E}}\text{X}$'s bounds. However, it is a little bit too large for our purposes: we would not be able to use the usual trick of adding a large power of 10 to ensure that the number of digits is fixed.

The bound on Q_A , implies that $10^6 + Q_A$ starts with the digit 1, followed by 0 or 1. We test, and call different auxiliaries for the two cases. An earlier implementation did the tests within the computation, but since we added a $\langle continuation \rangle$, this is not possible because the macro has 9 parameters.

The result we want is then (the overall power of 10 is arbitrary):

$$10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i \rangle \cdot \#5\#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i \rangle \cdot \#7) \\ + 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i \rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8),$$

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 \TeX 's limits once more.

```

12144 \cs_new:Npn \__fp_div_significand_calc:wwnnnnnnn #1
12145 {
12146   \if_meaning:w 1 #1
12147   \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnn
12148   \else:
12149   \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnn
12150   \fi:
12151 }
12152 \cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
12153 {
12154   1 1 #1
12155   #9 \exp_after:wN ;
12156   \int_use:N \__int_eval:w \c__fp_Bigg_leading_shift_int
12157   + #2 - #1 * #5 - #5#60
12158   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
12159   \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
12160   + #3 - #1 * #6 - #70
12161   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
12162   \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
12163   + #4 - #1 * #7 - #80
12164   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
12165   \int_use:N \__int_eval:w \c__fp_Bigg_trailing_shift_int
12166   - #1 * #8 ;
12167   {#5}{#6}{#7}{#8}
12168 }
12169 \cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
12170 {
12171   1 0 #1
12172   #9 \exp_after:wN ;
12173   \int_use:N \__int_eval:w \c__fp_Bigg_leading_shift_int
12174   + #2 - #1 * #5
12175   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
12176   \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
12177   + #3 - #1 * #6
12178   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
12179   \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
12180   + #4 - #1 * #7
12181   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
12182   \int_use:N \__int_eval:w \c__fp_Bigg_trailing_shift_int
12183   - #1 * #8 ;

```

```

12184     {#5}{#6}{#7}{#8}
12185   }

```

(End definition for `_fp_div_significand_calc:wwnnnnnnn`. This function is documented on page ??.)

```

\_fp_div_significand_ii:wwn    \_fp_div_significand_ii:wwn <y> ; <B1> ; {<B2>} {<B3>} {<B4>} {<Z1>}
                               {<Z2>} {<Z3>} {<Z4>} <continuations> <sign>

```

Compute Q_B by evaluating $\langle B_1 \rangle \langle B_2 \rangle 0 / y - 1$. The result will be output to the left, in an `_int_eval:w` which we start now. Once that is evaluated (and the other Q_i also, since later expansions are triggered by this one), a packing auxiliary takes care of placing the digits of Q_B in an appropriate way for the final addition to obtain Q . This auxiliary is also used to compute Q_C and Q_D with the inputs C and D instead of B .

```

12186 \cs_new:Npn \_fp_div_significand_ii:wwn #1; #2; #3
12187 {
12188   \exp_after:wN \_fp_div_significand_pack:NNN
12189   \int_use:N \_int_eval:w
12190   \exp_after:wN \_fp_div_significand_calc:wwnnnnnnn
12191   \int_use:N \_int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
12192 }

```

(End definition for `_fp_div_significand_ii:wwn`.)

```

\_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 .

```

12193 \cs_new:Npn \_fp_div_significand_iii:wwnnnnn #1; #2; #3; #4; #5 #6; #7
12194 {
12195   0
12196   \exp_after:wN \_fp_div_significand_iv:wwnnnnnnn
12197   \int_use:N \_int_eval:w (\c_two * #2 #3) / #6 #7 ; % <- P
12198   #2 ; {#3} {#4} {#5}
12199   {#6} {#7}
12200 }

```

(End definition for `_fp_div_significand_iii:wwnnnnn`.)

```

\_fp_div_significand_iv:wwnnnnnnn <P> ; <E1> ; {<E2>} {<E3>} {<E4>}
\_fp_div_significand_v:NNw        {<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>
\_fp_div_significand_vi:Nw

```

This adds to the current expression $(10^7 + 10 \cdot Q_D)$ a contribution of $5 \cdot P + \text{sign}(T)$ with $T = 2E - PZ$. This amounts to adding $P/2$ to Q_D , with an extra *rounding* digit. This *rounding* digit is 0 or 5 if T does not contribute, *i.e.*, if $0 = T = 2E - PZ$, in other words if $10^{16}A/Z$ is an integer or half-integer. Otherwise it is in the appropriate range, $[1, 4]$ or $[6, 9]$. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute T exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation `#1 · #6#7` below does not cause an overflow: naively, P can be up to 35, and `#6#7` up to 10^8 , but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For $P < 10$, the product obeys $P \cdot \#6\#7 < 10^8 \cdot P < 10^9$.
- For large $P \geq 3$, the rounding error on P , which is at most 1, is less than a factor of 2, hence $P \leq 4E/Z$. Also, $\#6\#7 \leq 10^8 \cdot Z$, hence $P \cdot \#6\#7 \leq 4E \cdot 10^8 < 10^9$.

Both inequalities could be made tighter if needed.

Note however that $P \cdot \#8\#9$ may overflow, since the two factors are now independent, and the result may reach $3.5 \cdot 10^9$. Thus we compute the two lower levels separately. The rest is standard, except that we use $+$ as a separator (ending integer expressions explicitly). T is negative if the first character is $-$, it is positive if the first character is neither 0 nor $-$. It is also positive if the first character is 0 and second argument of `__fp_div_significand_vi:Nw`, a sum of several terms, is also zero. Otherwise, there was an exact agreement: $T = 0$.

```

12201 \cs_new:Npn \__fp_div_significand_iv:wnnnnnnnn #1; #2;#3#4#5 #6#7#8#9
12202 {
12203   + \c_five * #1
12204   \exp_after:wN \__fp_div_significand_vi:Nw
12205   \int_use:N \__int_eval:w -20 + 2*#2#3 - #1*#6#7 +
12206   \exp_after:wN \__fp_div_significand_v:NN
12207   \int_use:N \__int_eval:w 199980 + 2*#4 - #1*#8 +
12208   \exp_after:wN \__fp_div_significand_v:NN
12209   \int_use:N \__int_eval:w 200000 + 2*#5 - #1*#9 ;
12210 }
12211 \cs_new:Npn \__fp_div_significand_v:NN #1#2 { #1#2 \__int_eval_end: + }
12212 \cs_new:Npn \__fp_div_significand_vi:Nw #1#2;
12213 {
12214   \if_meaning:w 0 #1
12215     \if_int_compare:w \__int_eval:w #2 > \c_zero + \c_one \fi:
12216   \else:
12217     \if_meaning:w - #1 - \else: + \fi: \c_one
12218   \fi:
12219   ;
12220 }

```

(End definition for `__fp_div_significand_iv:wnnnnnnnn`, `__fp_div_significand_v:NNw`, and `__fp_div_significand_vi:`

`__fp_div_significand_pack:NNN`

At this stage, we are in the following situation: \TeX is in the process of expanding several integer expressions, thus functions at the bottom expand before those above.

```

\__fp_div_significand_test_o:w 106 + QA \__fp_div_significand_-
pack:NNN 106 + QB \__fp_div_significand_pack:NNN 106 + QC \__fp_-
div_significand_pack:NNN 107 + 10 · QD + 5 · P + ε ; <sign>

```

Here, $\varepsilon = \text{sign}(T)$ is 0 in case $2E = PZ$, 1 in case $2E > PZ$, which means that P was the correct value, but not with an exact quotient, and -1 if $2E < PZ$, *i.e.*, P was an overestimate. The packing function we define now does nothing special: it removes the 10^6 and carries two digits (for the 10^5 's and the 10^4 's).

```

12221 \cs_new:Npn \__fp_div_significand_pack:NNN 1 #1 #2 { + #1 #2 ; }

```

(End definition for `__fp_div_significand_pack:NNN`.)


```
\_fp_div_significand_test_o:w 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.

```

12222 \cs_new:Npn \__fp_div_significand_test_o:w 10 #1
12223 {
12224   \if_meaning:w 0 #1
12225     \exp_after:wN \__fp_div_significand_small_o:wwwNNNNwN
12226   \else:
12227     \exp_after:wN \__fp_div_significand_large_o:wwwNNNNwN
12228   \fi:
12229   #1
12230 }

```

(End definition for _fp_div_significand_test_o:w.)

$$\backslash_fp_div_significand_small_o:wwwNNNNwN \ 0 \langle 4d \rangle ; \quad \langle 4d \rangle ; \langle 4d \rangle ; \langle 5d \rangle$$

Standard use of `__fp_basics_pack_low:NNNNNw` and `__fp_basics_pack_high:NNNNNw`. We finally get to use the `\final sign` which has been sitting there for a while.

```

12231 \cs_new:Npn \__fp_div_significand_small_o:wwwNNNNwN
12232   0 #1; #2; #3; #4#5#6#7#8; #9
12233   {
12234     \exp_after:wN \__fp_basics_pack_high:NNNNNw
12235     \int_use:N \__int_eval:w 1 #1#2
12236     \exp_after:wN \__fp_basics_pack_low:NNNNNw
12237     \int_use:N \__int_eval:w 1 #3#4#5#6#7
12238     + \__fp_round:NNN #9 #7 #8
12239     \exp_after:wN ;
12240   }

```

(End definition for _fp_div_significand_small_o:wwwNNNNwN.)

$\backslash_fp_div_significand_large_o:wwwNNNNwN$	$\backslash_fp_div_significand_large_o:wwwNNNNwN$	$\langle 5d \rangle$	$\langle 4d \rangle$	$\langle 4d \rangle$	$\langle 5d \rangle$
	$\langle sign \rangle$				

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 $\langle \textit{rounding digit} \rangle$ from the last two of our 18 digits.

```

12241 \cs_new:Npn \__fp_div_significand_large_o:wwwNNNNwN
12242   #1; #2; #3; #4#5#6#7#8; #9
12243   {
12244     + \c_one
12245     \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNwN
12246     \int_use:N \__int_eval:w 1 #1 #2
12247     \exp_after:wN \__fp_basics_pack_weird_low:NNNNwN
12248     \int_use:N \__int_eval:w 1 #3 #4 #5 #6 +
12249     \exp_after:wN \__fp_round:NNN
12250     \exp_after:wN #9
12251     \exp_after:wN #6

```

```

12252         \__int_value:w \__fp_round_digit:Nw #7 #8 ;
12253         \exp_after:wN ;
12254     }
(End definition for \__fp_div_significand_large_o:wwwNNNNwN.)

```

33.5 Unary operations

__fp_-_o:w This function flips the sign of the *floating point* and expands after it in the input stream, just like **__fp+_o:ww** etc. We add a hook used by **l3fp-expo**: anything before **\s__fp** is ignored.

```

12255 \cs_new:cpn { __fp_-_o:w } #1 \s__fp \__fp_chk:w #2 #3
12256 {
12257     \exp_after:wN \__fp_exp_after_o:w
12258     \exp_after:wN \s__fp
12259     \exp_after:wN \__fp_chk:w
12260     \exp_after:wN #2
12261     \int_use:N \__int_eval:w \c_two - #3 \__int_eval_end:
12262 }
(End definition for \__fp_-_o:w.)

```

__fp_abs_o:w This function sets the sign of the *floating point* to be positive, and expands after itself in the input stream, just like **__fp_-_o:w**. We must leave the sign of **nan** invariant.

```

12263 \cs_new:Npn \__fp_abs_o:w \s__fp \__fp_chk:w #1 #2
12264 {
12265     \exp_after:wN \__fp_exp_after_o:w
12266     \exp_after:wN \s__fp
12267     \exp_after:wN \__fp_chk:w
12268     \exp_after:wN #1
12269     \__int_value:w \if_meaning:w 1 #2 1 \else: 0 \fi: \exp_stop_f:
12270 }
(End definition for \__fp_abs_o:w.)

```

```

12271 </initex | package>

```

34 l3fp-extended implementation

```

12272 <*initex | package>

```

```

12273 <@@=fp>

```

34.1 Description of extended fixed points

In this module, we work on (almost) fixed-point numbers with extended (24 digits) precision. This is 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 trailing 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:Nw`. 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.

34.2 Helpers for extended fixed points

`\c__fp_one_fixed_tl` The extended fixed-point number 1, used in `l3fp-expo`.

```
12274 \tl_const:Nn \c__fp_one_fixed_tl
12275 { {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 $2^{31}-1$).

```
12276 \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 $a_1 \leq 2^{31} - 10001$.

```
12277 \cs_new:Npn \__fp_fixed_add_one:wN #1#2; #3
12278 {
12279   \exp_after:wN #3 \exp_after:wN
12280   { \int_use:N \__int_eval:w \c_ten_thousand + #1 } #2 ;
12281 }
(End definition for \__fp_fixed_add_one:wN.)
```

`__fp_fixed_mul_after:wn` The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the $\langle continuation \rangle$ #2 in front. The $\langle continuation \rangle$ was brought up through the expansions by the packing functions.

```
12282 \cs_new:Npn \__fp_fixed_mul_after:wn #1; #2 { #2 {#1} }
(End definition for \__fp_fixed_mul_after:wn.)
```

34.3 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.

```

12283 \cs_new:Npn \__fp_fixed_div_int:wnN #1#2#3#4#5#6 ; #7 ; #8
12284 {
12285   \exp_after:wN \__fp_fixed_div_int_after:Nw
12286   \exp_after:wN #8
12287   \int_use:N \__int_eval:w \c_minus_one
12288   \__fp_fixed_div_int:wnN
12289   #1; {\#7} \__fp_fixed_div_int_auxi:wnn
12290   #2; {\#7} \__fp_fixed_div_int_auxi:wnn
12291   #3; {\#7} \__fp_fixed_div_int_auxi:wnn
12292   #4; {\#7} \__fp_fixed_div_int_auxi:wnn
12293   #5; {\#7} \__fp_fixed_div_int_auxi:wnn
12294   #6; {\#7} \__fp_fixed_div_int_auxii:wnn ;
12295 }
12296 \cs_new:Npn \__fp_fixed_div_int:wnN #1; #2 #3
12297 {
12298   \exp_after:wN #3
12299   \int_use:N \__int_eval:w #1 / #2 - \c_one ;
12300   {\#2}
12301   {\#1}
12302 }

```

```

12303 \cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1; #2 #3
12304 {
12305   + #1
12306   \exp_after:wN \__fp_fixed_div_int_pack:Nw
12307   \int_use:N \__int_eval:w 9999
12308   \exp_after:wN \__fp_fixed_div_int:wN
12309   \int_use:N \__int_eval:w #3 - #1*#2 \__int_eval_end:
12310 }
12311 \cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + \c_two ; }
12312 \cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
12313 \cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }
(End definition for \__fp_fixed_div_int:wN. This function is documented on page ??.)

```

34.4 Adding and subtracting fixed points

`__fp_fixed_add:wnn` Computes $a + b$ (resp. $a - b$) and feeds the result to the *continuation*. This function requires $0 \leq a_1, b_1 < 50000$, and requires the result to be positive (this happens automatically for addition). The two functions only differ 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 two signs, 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.

```

12314 \cs_new_nopar:Npn \__fp_fixed_add:wnn { \__fp_fixed_add:Nnnnnwnn + }
12315 \cs_new_nopar:Npn \__fp_fixed_sub:wnn { \__fp_fixed_add:Nnnnnwnn - }
12316 \cs_new:Npn \__fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
12317 {
12318   \exp_after:wN \__fp_fixed_add_after:NNNNNwn
12319   \int_use:N \__int_eval:w 9 9999 9998 + #2#3 #1 #7#8
12320   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
12321   \int_use:N \__int_eval:w 1 9999 9998 + #4#5
12322   \__fp_fixed_add:nnNnnwn #6 #1
12323 }
12324 \cs_new:Npn \__fp_fixed_add:nnNnnwn #1#2 #3 #4#5 #6#7 ; #8
12325 {
12326   #3 #4#5
12327   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
12328   \int_use:N \__int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
12329 }
12330 \cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
12331 { + #1 ; {#7} {#2#3#4#5} {#6} }
12332 \cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
12333 { #7 {#1#2#3#4#5} {#6} }

```

(End definition for `__fp_fixed_add:wnn` and `__fp_fixed_sub:wnn`. These functions are documented on page ??.)

34.5 Multiplying fixed points

`_fp_fixed_mul:wnn` 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

$$\begin{aligned}
 a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\
 & + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
 & + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\
 & + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
 & + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} + a_1 \cdot b_5 + a_5 \cdot b_1 \right) \cdot 10^{-24}
 \end{aligned}$$

where the $O(10^{-24})$ stands for terms which are at most $5 \cdot 10^{-24}$; ignoring those leads to an error of at most 5 ulp. Note how the first 15 terms only depend on a_1, \dots, a_4 and b_1, \dots, b_4 , while the last 6 terms only depend on a_1, a_2, a_5, a_6 , and the corresponding parts of b . Hence, the first function grabs a_1, \dots, a_4 , the rest of a , and b_1, \dots, b_4 , and writes the 15 first terms of the expression, including a left parenthesis for the fraction. The `i` auxiliary receives $a_5, a_6, b_1, b_2, a_1, a_2, b_5, b_6$ and finally the $\langle continuation \rangle$ as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The packing auxiliaries bring the $\langle continuation \rangle$ up through the expansion chain, as `#7`, and it is finally placed in front of the 6 brace groups by `_fp_fixed_mul_after:wn`.

```

12334 \cs_new:Npn \_fp_fixed_mul:wnn #1#2#3#4 #5; #6#7#8#9
12335 {
12336   \exp_after:wN \_fp_fixed_mul_after:wn
12337   \int_use:N \_int_eval:w \c__fp_leading_shift_int
12338   \exp_after:wN \_fp_pack:NNNNNwn
12339   \int_use:N \_int_eval:w \c__fp_middle_shift_int
12340   + #1*#6
12341   \exp_after:wN \_fp_pack:NNNNNwn
12342   \int_use:N \_int_eval:w \c__fp_middle_shift_int
12343   + #1*#7 + #2*#6
12344   \exp_after:wN \_fp_pack:NNNNNwn
12345   \int_use:N \_int_eval:w \c__fp_middle_shift_int
12346   + #1*#8 + #2*#7 + #3*#6
12347   \exp_after:wN \_fp_pack:NNNNNwn
12348   \int_use:N \_int_eval:w \c__fp_middle_shift_int
12349   + #1*#9 + #2*#8 + #3*#7 + #4*#6
12350   \exp_after:wN \_fp_pack:NNNNNwn
12351   \int_use:N \_int_eval:w \c__fp_trailing_shift_int
12352   + #2*#9 + #3*#8 + #4*#7
12353   + ( #3*#9 + #4*#8
12354     + \_fp_fixed_mul:nnnnnnwn #5 {#6}{#7} {#1}{#2}
12355   )
12356 \cs_new:Npn \_fp_fixed_mul:nnnnnnwn #1#2 #3#4 #5#6 #7#8 ; #9

```

```

12357 {
12358     #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c_ten_thousand
12359     + #1*#3 + #5*#7 ;
12360     {#9} ;
12361 }

```

(End definition for `_fp_fixed_mul:wnn`. This function is documented on page ??.)

34.6 Combining product and sum of fixed points

`_fp_fixed_mul_add:wnn` Compute $a \times b + c$, $c - a \times b$, and $1 - a \times b$ and feed the result to the $\langle continuation \rangle$.
`_fp_fixed_mul_sub_back:wnn` Those functions require $0 \leq a_1, b_1, c_1 \leq 10000$. Since those functions are at the heart of
`_fp_fixed_mul_one_minus_mul:wnn` the computation of Taylor expansions, we over-optimize them a bit, and in particular we
do not factor out the common parts of the three functions.

For definiteness, consider the task of computing $a \times b + c$. We will perform carries in

$$\begin{aligned}
 a \times b + c = & (a_1 \cdot b_1 + c_1 c_2) \cdot 10^{-8} \\
 & + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
 & + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4) \cdot 10^{-16} \\
 & + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
 & + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} + a_1 \cdot b_5 + a_5 \right)
 \end{aligned}$$

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:wnn`, will be taken in the integer expression for the 10^{-24} level. The $\langle continuation \rangle$ is placed correctly to be taken upstream by packing auxiliaries.

```

12362 \cs_new:Npn \_fp_fixed_mul_add:wnn #1; #2; #3#4#5#6#7#8; #9
12363 {
12364     \exp_after:wN \_fp_fixed_mul_after:wn
12365     \int_use:N \_int_eval:w \c\_fp_big_leading_shift_int
12366     \exp_after:wN \_fp_pack_big:NNNNNNwn
12367     \int_use:N \_int_eval:w \c\_fp_big_middle_shift_int + #3 #4
12368     \_fp_fixed_mul_add:Nwnnnwnnn +
12369     + #5 #6 ; #2 ; #1 ; #2 ; +
12370     + #7 #8 ; {#9} ;
12371 }
12372 \cs_new:Npn \_fp_fixed_mul_sub_back:wnn #1; #2; #3#4#5#6#7#8; #9
12373 {
12374     \exp_after:wN \_fp_fixed_mul_after:wn
12375     \int_use:N \_int_eval:w \c\_fp_big_leading_shift_int
12376     \exp_after:wN \_fp_pack_big:NNNNNNwn
12377     \int_use:N \_int_eval:w \c\_fp_big_middle_shift_int + #3 #4
12378     \_fp_fixed_mul_add:Nwnnnwnnn -

```

```

12379         + #5 #6 ; #2 ; #1 ; #2 ; -
12380         + #7 #8 ; {#9} ;
12381     }
12382 \cs_new:Npn \__fp_fixed_one_minus_mul:wwn #1; #2; #3
12383 {
12384     \exp_after:wN \__fp_fixed_mul_after:wn
12385     \int_use:N \__int_eval:w \c__fp_big_leading_shift_int
12386     \exp_after:wN \__fp_pack_big:NNNNNNwn
12387     \int_use:N \__int_eval:w \c__fp_big_middle_shift_int + 1 0000 0000
12388     \__fp_fixed_mul_add:Nwnnnwnnn -
12389     ; #2 ; #1 ; #2 ; -
12390     ; {#3} ;
12391 }
(End definition for \__fp_fixed_mul_add:www, \__fp_fixed_mul_sub_back:www, and \__fp_fixed_mul_one_minus_mul:wwn)

```

__fp_fixed_mul_add:Nwnnnwnnn Here, $\langle op \rangle$ is either + or -. Arguments #3, #4, #5 are $\langle b_1 \rangle$, $\langle b_2 \rangle$, $\langle b_3 \rangle$; arguments #7, #8, #9 are $\langle a_1 \rangle$, $\langle a_2 \rangle$, $\langle a_3 \rangle$. We can build three levels: $a_1 \cdot b_1$ for 10^{-8} , $(a_1 \cdot b_2 + a_2 \cdot b_1)$ for 10^{-12} , and $(a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4)$ for 10^{-16} . The a - b products use the sign #1. Note that #2 is empty for __fp_fixed_one_minus_mul:wwn. We call the ii auxiliary for levels 10^{-20} and 10^{-24} , keeping the pieces of $\langle a \rangle$ we've read, but not $\langle b \rangle$, since there is another copy later in the input stream.

```

12392 \cs_new:Npn \__fp_fixed_mul_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
12393 {
12394     #1 #7*#3
12395     \exp_after:wN \__fp_pack_big:NNNNNNwn
12396     \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
12397     #1 #7*#4 #1 #8*#3
12398     \exp_after:wN \__fp_pack_big:NNNNNNwn
12399     \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
12400     #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
12401     \exp_after:wN \__fp_pack_big:NNNNNNwn
12402     \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
12403     #1 \__fp_fixed_mul_add:nnnnwnnnn {#7}{#8}{#9}
12404 }
(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 $\langle b \rangle$.

```

12405 \cs_new:Npn \__fp_fixed_mul_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
12406 {

```



```

12407      ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
12408      \exp_after:wN \_fp_pack_big:NNNNNNwn
12409      \int_use:N \_int_eval:w \c\_fp_big_trailing_shift_int
12410      \_fp_fixed_mul_add:nnnnwnnwN
12411      { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
12412      { #7 + #4*#8 + #3*#9 + #2 }
12413      {#1} #5;
12414      {#6}
12415    }
(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.

```

12416 \cs_new:Npn \_fp_fixed_mul_add:nnnnwnnwN #1#2 #3#4#5; #6#7#8; #9
12417 {
12418   #9 (#4* #1 *#7)
12419   #9 (#5*#6+#4* #2 *#7+#3*#8) / \c_ten_thousand
12420 }
(End definition for \_fp_fixed_mul_add:nnnnwnnwN.)

```

34.7 Converting from fixed point to floating point

`_fp_fixed_to_float:wN` yields
`_fp_fixed_to_float:Nw`

$\langle exponent' \rangle$; $\{ \langle a'_1 \rangle \} \{ \langle a'_2 \rangle \} \{ \langle a'_3 \rangle \} \{ \langle a'_4 \rangle \}$;

And the `to_fixed` version gives six brace groups instead of 4, ensuring that $1000 \leq \langle a'_1 \rangle \leq 9999$. At this stage, we know that $\langle a'_1 \rangle$ is positive (otherwise, it is sign of an error before), and we assume that it is less than 10^8 .¹⁷

```

12421 \cs_new:Npn \_fp_fixed_to_float:Nw #1#2; { \_fp_fixed_to_float:wN #2; #1 }
12422 \cs_new:Npn \_fp_fixed_to_float:wN #1#2#3#4#5#6; #7
12423 {
12424   + \c_four % for the 8-digit-at-the-start thing.
12425   \exp_after:wN \exp_after:wN
12426   \exp_after:wN \_fp_fixed_to_loop:N
12427   \exp_after:wN \use_none:n
12428   \int_use:N \_int_eval:w
12429   1 0000 0000 + #1 \exp_after:wN \_fp_use_none_stop_f:n
12430   \_int_value:w 1#2 \exp_after:wN \_fp_use_none_stop_f:n
12431   \_int_value:w 1#3#4 \exp_after:wN \_fp_use_none_stop_f:n
12432   \_int_value:w 1#5#6
12433   \exp_after:wN ;
12434   \exp_after:wN ;

```

¹⁷Bruno: I must double check this assumption.

```

12435 }
12436 \cs_new:Npn \__fp_fixed_to_loop:N #1
12437 {
12438   \if_meaning:w 0 #1
12439     - \c_one
12440     \exp_after:wN \__fp_fixed_to_loop:N
12441   \else:
12442     \exp_after:wN \__fp_fixed_to_loop_end:w
12443     \exp_after:wN #1
12444   \fi:
12445 }
12446 \cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
12447 {
12448   \if_meaning:w ; #1
12449     \exp_after:wN \__fp_fixed_to_float_zero:w
12450   \else:
12451     \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
12452     \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
12453     \exp_after:wN \__fp_fixed_to_float_pack:ww
12454     \exp_after:wN ;
12455   \fi:
12456   #1 #2 0000 0000 0000 0000 ;
12457 }
12458 \cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
12459 {
12460   - \c_two * \c__fp_max_exponent_int ;
12461   {0000} {0000} {0000} {0000} ;
12462 }
12463 \cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
12464 {
12465   \if_int_compare:w #2 > \c_four
12466     \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
12467   \fi:
12468   ; #1 ;
12469 }
12470 \cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
12471 {
12472   \exp_after:wN \__fp_basics_pack_high:NNNNNw
12473   \int_use:N \__int_eval:w 1 #1#2
12474   \exp_after:wN \__fp_basics_pack_low:NNNNNw
12475   \int_use:N \__int_eval:w 1 #3#4 + \c_one ;
12476 }

```

(End definition for __fp_fixed_to_float:wN and __fp_fixed_to_float:Nw.)

__fp_fixed_inv_to_float:wN Starting from fixed_dtf A ; B ; we want to compute A/B , and express it as a floating point number. Normalize both numbers by removing leading brace groups of zeros and leaving the appropriate exponent shift in the input stream.

```

12477 \cs_new:Npn \__fp_fixed_inv_to_float:wN #1#2; #3
12478 {

```

```

12479 + \__int_eval:w % ^^A todo: remove the +?
12480 \if_int_compare:w #1 < \c_one_thousand
12481 \__fp_fixed_dtf_zeros:wNnnnnnn
12482 \fi:
12483 \__fp_fixed_dtf_no_zero:Nwn + {#1} #2 \s__fp
12484 \__fp_fixed_dtf_approx:n
12485 {10000} {0000} {0000} {0000} {0000} {0000} ;
12486 }
12487 \cs_new:Npn \__fp_fixed_div_to_float:ww #1#2; #3#4;
12488 {
12489 \if_int_compare:w #1 < \c_one_thousand
12490 \__fp_fixed_dtf_zeros:wNnnnnnn
12491 \fi:
12492 \__fp_fixed_dtf_no_zero:Nwn - {#1} #2 \s__fp
12493 {
12494 \if_int_compare:w #3 < \c_one_thousand
12495 \__fp_fixed_dtf_zeros:wNnnnnnn
12496 \fi:
12497 \__fp_fixed_dtf_no_zero:Nwn + {#3} #4 \s__fp
12498 \__fp_fixed_dtf_approx:n
12499 }
12500 }
12501 \cs_new:Npn \__fp_fixed_dtf_no_zero:Nwn #1#2 \s__fp #3 { #3 #2; }
12502 \cs_new:Npn \__fp_fixed_dtf_zeros:wNnnnnnn
12503 \fi: \__fp_fixed_dtf_no_zero:Nwn #1#2#3#4#5#6#7
12504 {
12505 \fi:
12506 #1 \c_minus_one
12507 \exp_after:wN \use_i_ii:nnn
12508 \exp_after:wN \__fp_fixed_dtf_zeros:NN
12509 \exp_after:wN #1
12510 \int_use:N \__int_eval:w 10 0000 + #2 \__int_eval_end: #3#4#5#6#7
12511 ; 1 ;
12512 }
12513 \cs_new:Npn \__fp_fixed_dtf_zeros:NN #1#2
12514 {
12515 \if_meaning:w 0 #2
12516 #1 \c_one
12517 \else:
12518 \__fp_fixed_dtf_zeros_end:wNww #2
12519 \fi:
12520 \__fp_fixed_dtf_zeros:NN #1
12521 }
12522 \cs_new:Npn \__fp_fixed_dtf_zeros_end:wNww
12523 #1 \fi: \__fp_fixed_dtf_zeros:NN #2 #3; #4 \s__fp
12524 {
12525 \fi:
12526 \if_meaning:w ; #1
12527 #2 \c_two * \c__fp_max_exponent_int
12528 \use_i_ii:nnn

```

```

12529      \fi:
12530      \__fp_fixed_dtf_zeros_auxi:ww
12531      #1#3 0000 0000 0000 0000 0000 0000 ;
12532    }
12533    \cs_new:Npn \__fp_fixed_dtf_zeros_auxi:ww
12534    {
12535      \__fp_pack_twice_four:wNNNNNNNN
12536      \__fp_pack_twice_four:wNNNNNNNN
12537      \__fp_pack_twice_four:wNNNNNNNN
12538      \__fp_fixed_dtf_zeros_auxii:ww
12539    } ;
12540  }
12541  \cs_new:Npn \__fp_fixed_dtf_zeros_auxii:ww #1; #2; #3 { #3 #1; }

```

We get

`__fp_fixed_dtf_approx:n` $\langle B' \rangle$; $\langle A' \rangle$;

where $\langle B' \rangle$ and $\langle A' \rangle$ are each 6 brace groups, representing fixed point numbers in the range $[0.1, 1)$. Denote by $x \in [1000, 9999]$ and $y \in [0, 9999]$ the first two groups of $\langle B' \rangle$. We first find an estimate a for the inverse of B' by computing

$$\begin{aligned}
\alpha &= \left[\frac{10^9}{x+1} \right] \\
\beta &= \left[\frac{10^9}{x} \right] \\
a &= 10^3 \alpha + (\beta - \alpha) \cdot \left(10^3 - \left[\frac{y}{10} \right] \right) - 1750,
\end{aligned}$$

where $\left[\frac{\cdot}{\cdot} \right]$ denotes ε -TeX's rounding division. The idea is to interpolate between α and β with a parameter $y/10^4$. The shift by 1750 helps to ensure that a is an underestimate of the correct value. We will prove that

$$1 - 2.255 \cdot 10^{-5} < \frac{B'a}{10^8} < 1.$$

We can then compute the inverse $B'a/10^8$ using $1/(1-\epsilon) \simeq (1+\epsilon)(1+\epsilon^2)$, which is correct up to a relative error of $\epsilon^4 < 2.6 \cdot 10^{-19}$. Since we target a 16-digit value, this is small enough.

Let us prove the upper bound first.

$$10^7 B'a < \left(10^3 x + \left[\frac{y}{10} \right] + \frac{3}{2} \right) \left(\left(10^3 - \left[\frac{y}{10} \right] \right) \beta + \left[\frac{y}{10} \right] \alpha - 1750 \right) \quad (1)$$

$$< \left(10^3 x + \left[\frac{y}{10} \right] + \frac{3}{2} \right) \left(\left(10^3 - \left[\frac{y}{10} \right] \right) \left(\frac{10^9}{x} + \frac{1}{2} \right) + \left[\frac{y}{10} \right] \left(\frac{10^9}{x+1} + \frac{1}{2} \right) - 1750 \right) \quad (2)$$

$$< \left(10^3 x + \left[\frac{y}{10} \right] + \frac{3}{2} \right) \left(\frac{10^{12}}{x} - \left[\frac{y}{10} \right] \frac{10^9}{x(x+1)} - 1250 \right) \quad (3)$$

We recognize a quadratic polynomial in $[y/10]$ with a negative leading coefficient, $([y/10] + a)(b - c[y/10]) \leq (b + ca)^2/(4c)$. Hence,

$$10^7 B' a < \frac{10^{15}}{x(x+1)} \left(x + \frac{1}{2} + \frac{3}{4} 10^{-3} - 6.25 \cdot 10^{-10} x(x+1) \right)^2$$

We want to prove that the squared expression is less than $x(x+1)$, which we do by simplifying the difference, and checking its sign,

$$x(x+1) - \left(x + \frac{1}{2} + \frac{3}{4} 10^{-3} - 6.25 \cdot 10^{-10} x(x+1) \right)^2 > -\frac{1}{4} (1 + 1.5 \cdot 10^{-3})^2 - 10^{-3} x + 1.25 \cdot 10^{-9} x(x+1) (x+0.5)$$

Now, the lower bound. The same computation as (1) imply

$$10^7 B' a > \left(10^3 x + \left\lfloor \frac{y}{10} \right\rfloor - \frac{1}{2} \right) \left(\frac{10^{12}}{x} - \left\lfloor \frac{y}{10} \right\rfloor \frac{10^9}{x(x+1)} - 2250 \right)$$

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 $y = 0$ or $y = 9999$, 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 $a < 10^8/B \leq 10^9$, hence we can compute a safely as a TeX integer, and even add 10^9 to it to ease grabbing of all the digits.

```

12542 \cs_new:Npn \__fp_fixed_dtf_approx:n #1
12543 {
12544   \exp_after:wN \__fp_fixed_dtf_approx:wnn
12545   \int_use:N \__int_eval:w 10 0000 0000 / ( #1 + \c_one ) ;
12546   {#1}
12547 }
12548 \cs_new:Npn \__fp_fixed_dtf_approx:wnn #1; #2#3
12549 {
12550   <assert> \assert:n { \tl_count:n {#1} = 6 }
12551   \exp_after:wN \__fp_fixed_dtf_approx:NNNNNw
12552   \int_use:N \__int_eval:w 10 0000 0000 - 1750
12553   + #1000 + (10 0000 0000/#2-#1) * (1000-#3/10) ;
12554   {#2}{#3}
12555 }
12556 \cs_new:Npn \__fp_fixed_dtf_approx:NNNNNw 1#1#2#3#4#5#6; #7; #8;
12557 {
12558   + \c_four % because of the line below "dtf_epsilon" here.
12559   \__fp_fixed_mul:wnn {000#1}{#2#3#4#5}{#6}{0000}{0000}{0000} ; #7;
12560   \__fp_fixed_dtf_epsilon:wN
12561   \__fp_fixed_mul:wnn {000#1}{#2#3#4#5}{#6}{0000}{0000}{0000} ;
12562   \__fp_fixed_mul:wnn #8;
12563   \__fp_fixed_to_float:wN ?
12564 }
12565 \cs_new:Npn \__fp_fixed_dtf_epsilon:wN #1#2#3#4#5#6;
12566 {
12567   <assert> \assert:n { #1 = 0000 }
```

```

12568 \assert:n { #2 = 9999 }
12569 \exp_after:wN \__fp_fixed_dtf_epsilon:NNNNNww
12570 \int_use:N \__int_eval:w 1 9999 9998 - #3#4 +
12571 \exp_after:wN \__fp_fixed_dtf_epsilon_pack:NNNNNw
12572 \int_use:N \__int_eval:w 2 0000 0000 - #5#6 ; {0000} ;
12573 }
12574 \cs_new:Npn \__fp_fixed_dtf_epsilon_pack:NNNNNw #1#2#3#4#5#6;
12575 { #1 ; {#2#3#4#5} {#6} }
12576 \cs_new:Npn \__fp_fixed_dtf_epsilon:NNNNNww #1#2#3#4#5#6; #7;
12577 {
12578 \__fp_fixed_mul:wwn %^A todo: optimize to use \__fp_mul_significand.
12579 {0000} {#2#3#4#5} {#6} #7 ;
12580 {0000} {#2#3#4#5} {#6} #7 ;
12581 \__fp_fixed_add_one:wN
12582 \__fp_fixed_mul:wwn {10000} {#2#3#4#5} {#6} #7 ;
12583 }
(End definition for \__fp_fixed_inv_to_float:wN and \__fp_fixed_div_to_float:ww.)
12584 \</initex | package>

```

35 l3fp-expo implementation

```

12585 \*initex | package>
12586 \<@@=fp>

```

35.1 Logarithm

35.1.1 Work plan

As for many other functions, we filter out special cases in `__fp_ln_o:w`. Then `__fp_ln_npos_o:w` receives a positive normal number, which we write in the form $a \cdot 10^b$ with $a \in [0.1, 1)$.

The rest of this section is actually not in sync with the code. Or is the code not in sync with the section?

We are given a positive normal number, of the form $a \cdot 10^b$ with $a \in [0.1, 1)$. To compute its logarithm, we find a small integer $5 \leq c < 50$ such that $0.91 \leq ac/5 < 1.1$, and use the relation

$$\ln(a \cdot 10^b) = b \cdot \ln(10) - \ln(c/5) + \ln(ac/5).$$

The logarithms $\ln(10)$ and $\ln(c/5)$ are looked up in a table. The last term is computed using the following Taylor series of \ln near 1:

$$\ln\left(\frac{ac}{5}\right) = \ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + t^2 \left(\frac{1}{3} + t^2 \left(\frac{1}{5} + t^2 \left(\frac{1}{7} + t^2 \left(\frac{1}{9} + \dots\right)\right)\right)\right)\right)$$

where $t = 1 - 10/(ac + 5)$. We can now see one reason for the choice of $ac \sim 5$: then $ac + 5 = 10(1 - \epsilon)$ with $-0.05 < \epsilon \leq 0.045$, hence

$$t = \frac{\epsilon}{1 - \epsilon} = \epsilon(1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4) \dots,$$

is not too difficult to compute.

35.1.2 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 12587 \tl_const:Nn \c__fp_ln_i_fixed_t1 { {0000}{0000}{0000}{0000}{0000}{0000} }
\c__fp_ln_ii_fixed_t1 12588 \tl_const:Nn \c__fp_ln_ii_fixed_t1 { {6931}{4718}{0559}{9453}{0941}{7232} }
\c__fp_ln_iii_fixed_t1 12589 \tl_const:Nn \c__fp_ln_iii_fixed_t1 { {10986}{1228}{8668}{1096}{9139}{5245} }
\c__fp_ln_iv_fixed_t1 12590 \tl_const:Nn \c__fp_ln_iv_fixed_t1 { {13862}{9436}{1119}{8906}{1883}{4464} }
\c__fp_ln_vii_fixed_t1 12591 \tl_const:Nn \c__fp_ln_vii_fixed_t1 { {19459}{1014}{9055}{3133}{0510}{5353} }
\c__fp_ln_viii_fixed_t1 12592 \tl_const:Nn \c__fp_ln_viii_fixed_t1 { {20794}{4154}{1679}{8359}{2825}{1696} }
\c__fp_ln_ix_fixed_t1 12593 \tl_const:Nn \c__fp_ln_ix_fixed_t1 { {21972}{2457}{7336}{2193}{8279}{0490} }
\c__fp_ln_x_fixed_t1 12594 \tl_const:Nn \c__fp_ln_x_fixed_t1 { {23025}{8509}{2994}{0456}{8401}{7991} }
12595 \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.)

35.1.3 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`.

```

12596 \cs_new:Npn \__fp_ln_o:w \s__fp \__fp_chk:w #1 #2
12597 {
12598   \if_meaning:w 2 #2
12599     \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
12600   \fi:
12601   \if_case:w #1 \exp_stop_f:
12602     \__fp_case_use:nw
12603     { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
12604   \or:
12605   \else:
12606     \__fp_case_return_same_o:w
12607   \fi:
12608   \__fp_ln_npos_o:w \s__fp \__fp_chk:w #1#2
12609 }

```

(End definition for `__fp_ln_o:w`.)

35.1.4 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.

```

12610 \cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3;
12611 { %^A todo: ln(1) should be "exact zero", not "underflow"
12612   \exp_after:wN \__fp_sanitize:Nw
12613   \__int_value:w % for the overall sign
12614   \if_int_compare:w #1 < \c_one
12615     2
12616   \else:
12617     0
12618   \fi:

```

```

12619 \exp_after:wN \exp_stop_f:
12620 \int_use:N \__int_eval:w % for the exponent
12621 \__fp_ln_significand:NNNNnnnN #2#3
12622 \__fp_ln_exponent:wn {#1}
12623 }
(End definition for \__fp_ln_npos_o:w.)

```

$\backslash_fp_ln_significand:NNNNnnnN \langle X_1 \rangle \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \} \langle continuation \rangle$
 This function expands to
 $\langle continuation \rangle \{ \langle Y_1 \rangle \} \{ \langle Y_2 \rangle \} \{ \langle Y_3 \rangle \} \{ \langle Y_4 \rangle \} \{ \langle Y_5 \rangle \} \{ \langle Y_6 \rangle \} ;$

where $Y = -\ln(X)$ as an extended fixed point.

```

12624 \cs_new:Npn \__fp_ln_significand:NNNNnnnN #1#2#3#4
12625 {
12626 \exp_after:wN \__fp_ln_x_ii:wnnnn
12627 \__int_value:w
12628 \if_case:w #1 \exp_stop_f:
12629 \or:
12630 \if_int_compare:w #2 < \c_four
12631 \__int_eval:w \c_ten - #2
12632 \else:
12633 6
12634 \fi:
12635 \or: 4
12636 \or: 3
12637 \or: 2
12638 \or: 2
12639 \or: 2
12640 \else: 1
12641 \fi:
12642 ; { #1 #2 #3 #4 }
12643 }
(End definition for \__fp_ln_significand:NNNNnnnN.)

```

$\backslash_fp_ln_x_ii:wnnnn$ We have thus found c . It is chosen such that $0.7 \leq ac < 1.4$ in all cases. Compute $1 + x = 1 + ac \in [1.7, 2.4)$.

```

12644 \cs_new:Npn \__fp_ln_x_ii:wnnnn #1; #2#3#4#5
12645 {
12646 \exp_after:wN \__fp_ln_div_after:Nw
12647 \cs:w c__fp_ln \tex_romannumeral:D #1 _fixed_tl \exp_after:wN \cs_end:
12648 \__int_value:w
12649 \exp_after:wN \__fp_ln_x_iv:wnnnnnnnn
12650 \int_use:N \__int_eval:w
12651 \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
12652 \int_use:N \__int_eval:w 9999 9999 + #1*#2#3 +
12653 \exp_after:wN \__fp_ln_x_iii:NNNNNw
12654 \int_use:N \__int_eval:w 1 0000 0000 + #1*#4#5 ;
12655 {20000} {0000} {0000} {0000}

```



```

12656 } %^A todo: reoptimize (a generalization attempt failed).
12657 \cs_new:Npn \__fp_ln_x_iii:NNNNNw #1 #2#3#4#5 #6; { #1; {#2#3#4#5} {#6} }
12658 \cs_new:Npn \__fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
12659 {
12660   #1#2#3#4#5 + \c_one ;
12661   {#1#2#3#4#5} {#6}
12662 }

```

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).¹⁸

$\backslash_fp_ln_x_iv:wnnnnnnnn \langle 1 \text{ or } 2 \rangle \langle 8d \rangle ; \{ \langle 4d \rangle \} \{ \langle 4d \rangle \} \langle fixed_tl \rangle$

The number is x . Compute y by adding 1 to the five first digits.

```

12663 \cs_new:Npn \__fp_ln_x_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
12664 {
12665   \exp_after:wN \__fp_div_significand_pack:NNN
12666   \int_use:N \__int_eval:w
12667   \__fp_ln_div_i:w #1 ;
12668   #6 #7 ; {#8} {#9}
12669   {#2} {#3} {#4} {#5}
12670   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
12671   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
12672   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
12673   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
12674   { \exp_after:wN \__fp_ln_div_vi:wnn \__int_value:w #1 }
12675 }
12676 \cs_new:Npn \__fp_ln_div_i:w #1;
12677 {
12678   \exp_after:wN \__fp_div_significand_calc:wnnnnnnnn
12679   \int_use:N \__int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
12680 }
12681 \cs_new:Npn \__fp_ln_div_ii:wnn #1; #2;#3 % y; B1;B2 <- for k=1
12682 {
12683   \exp_after:wN \__fp_div_significand_pack:NNN
12684   \int_use:N \__int_eval:w
12685   \exp_after:wN \__fp_div_significand_calc:wnnnnnnnn
12686   \int_use:N \__int_eval:w 999999 + #2 #3 / #1 ; % Q2
12687   #2 #3 ;
12688 }
12689 \cs_new:Npn \__fp_ln_div_vi:wnn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
12690 {
12691   \exp_after:wN \__fp_div_significand_pack:NNN
12692   \int_use:N \__int_eval:w 1000000 + #2 #3 / #1 ; % Q6
12693 }

```

We now have essentially¹⁹

$\backslash_fp_ln_div_after:Nw \langle fixed\ tl \rangle \backslash_fp_div_significand_pack:NNN 10^6 +$
 $Q_1 \backslash_fp_div_significand_pack:NNN 10^6 + Q_2 \backslash_fp_div_significand_$
 $pack:NNN 10^6 + Q_3 \backslash_fp_div_significand_pack:NNN 10^6 + Q_4 \backslash_fp_$
 $div_significand_pack:NNN 10^6 + Q_5 \backslash_fp_div_significand_pack:NNN$
 $10^6 + Q_6 ; \langle exponent \rangle ; \langle continuation \rangle$

¹⁸Bruno: to be completed.

¹⁹Bruno: 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.

where $\langle fixed\ tl \rangle$ holds the logarithm of a number in $[1, 10]$, and $\langle exponent \rangle$ is the exponent. Also, the expansion is done backwards. Then `_fp_div_significand_pack:NNN` puts things in the correct order to add the Q_i together and put semicolons between each piece. Once those have been expanded, we get

```
\_fp\_ln\_div\_after:Nw  $\langle fixed\ tl \rangle \langle 1d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ;$ 
 $\langle 4d \rangle ; \langle exponent \rangle ;$ 
```

Just as with division, we know that the first two digits are 1 and 0 because of bounds on the final result of the division $2/(x+1)$, which is between roughly 0.8 and 1.2. We then compute $1 - 2/(x+1)$, after testing whether $2/(x+1)$ is greater than or smaller than 1.

```
12694 \cs_new:Npn \_fp\_ln\_div\_after:Nw #1#2;
12695 {
12696   \if_meaning:w 0 #2
12697     \exp_after:wN \_fp\_ln\_t\_small:Nw
12698   \else:
12699     \exp_after:wN \_fp\_ln\_t\_large:NNw
12700     \exp_after:wN -
12701   \fi:
12702   #1
12703 }
12704 \cs_new:Npn \_fp\_ln\_t\_small:Nw #1 #2; #3; #4; #5; #6; #7;
12705 {
12706   \exp_after:wN \_fp\_ln\_t\_large:NNw
12707   \exp_after:wN + % <sign>
12708   \exp_after:wN #1
12709   \int_use:N \_int\_eval:w 9999 - #2 \exp_after:wN ;
12710   \int_use:N \_int\_eval:w 9999 - #3 \exp_after:wN ;
12711   \int_use:N \_int\_eval:w 9999 - #4 \exp_after:wN ;
12712   \int_use:N \_int\_eval:w 9999 - #5 \exp_after:wN ;
12713   \int_use:N \_int\_eval:w 9999 - #6 \exp_after:wN ;
12714   \int_use:N \_int\_eval:w 1 0000 - #7 ;
12715 }
12716 \_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 `_fp_ln_t_small:w`, they can have less than 4 digits.

```
12716 \cs_new:Npn \_fp\_ln\_t\_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
12717 {
12718   \exp_after:wN \_fp\_ln\_square\_t\_after:w
12719   \int_use:N \_int\_eval:w 9999 0000 + #3*#3
12720   \exp_after:wN \_fp\_ln\_square\_t\_pack:NNNNw
12721   \int_use:N \_int\_eval:w 9999 0000 + 2*#3*#4
12722   \exp_after:wN \_fp\_ln\_square\_t\_pack:NNNNw
12723   \int_use:N \_int\_eval:w 9999 0000 + 2*#3*#5 + #4*#4
12724   \exp_after:wN \_fp\_ln\_square\_t\_pack:NNNNw
```

```

12725      \int_use:N \__int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
12726      \exp_after:wN \__fp_ln_square_t_pack:NNNNw
12727      \int_use:N \__int_eval:w 1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
12728      + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
12729      % ; ; ;
12730      \exp_after:wN \__fp_ln_twice_t_after:w
12731      \int_use:N \__int_eval:w -1 + 2*#3
12732      \exp_after:wN \__fp_ln_twice_t_pack:Nw
12733      \int_use:N \__int_eval:w 9999 + 2*#4
12734      \exp_after:wN \__fp_ln_twice_t_pack:Nw
12735      \int_use:N \__int_eval:w 9999 + 2*#5
12736      \exp_after:wN \__fp_ln_twice_t_pack:Nw
12737      \int_use:N \__int_eval:w 9999 + 2*#6
12738      \exp_after:wN \__fp_ln_twice_t_pack:Nw
12739      \int_use:N \__int_eval:w 9999 + 2*#7
12740      \exp_after:wN \__fp_ln_twice_t_pack:Nw
12741      \int_use:N \__int_eval:w 10000 + 2*#8 ; ;
12742      { \__fp_ln_c:NwNw #1 }
12743      #2
12744      }
12745      \cs_new:Npn \__fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
12746      \cs_new:Npn \__fp_ln_twice_t_after:w #1; { ; ; ; {#1} }
12747      \cs_new:Npn \__fp_ln_square_t_pack:NNNNw #1 #2#3#4#5 #6;
12748      { + #1#2#3#4#5 ; {#6} }
12749      \cs_new:Npn \__fp_ln_square_t_after:w 1 0 #1#2#3 #4;
12750      { \__fp_ln_Taylor:wwNw {0#1#2#3} {#4} }
(End definition for \__fp_ln_x_ii:wnnnn.)

```

__fp_ln_Taylor:wwNw Denoting $T = t^2$, we get

```

\__fp_ln_Taylor:wwNw {<T1>} {<T2>} {<T3>} {<T4>} {<T5>} {<T6>} ; ;
{<(2t)1>} {<(2t)2>} {<(2t)3>} {<(2t)4>} {<(2t)5>} {<(2t)6>} ; { \__fp_ln_
c:NwNn <sign> } <fixed tl> <exponent> ; <continuation>

```

And we want to compute

$$\ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + T \left(\frac{1}{3} + T \left(\frac{1}{5} + T \left(\frac{1}{7} + T \left(\frac{1}{9} + \cdots\right)\right)\right)\right)\right)$$

The process looks as follows

```

\loop 5; A;
\div_int 5; 1.0; \add A; \mul T; {\loop \eval 5-2;}
\add 0.2; A; \mul T; {\loop \eval 5-2;}
\mul B; T; {\loop 3;}
\loop 3; C;

```

20

²⁰Bruno: add explanations.

This uses the routine for dividing a number by a small integer ($< 10^4$).

```

12751 \cs_new:Npn \__fp_ln_Taylor:wwNw
12752 { \__fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000}{0000} ; }
12753 \cs_new:Npn \__fp_ln_Taylor_loop:www #1; #2; #3;
12754 {
12755   \if_int_compare:w #1 = \c_one
12756     \__fp_ln_Taylor_break:w
12757   \fi:
12758   \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl ; #1;
12759   \__fp_fixed_add:wwn #2;
12760   \__fp_fixed_mul:wwn #3;
12761   {
12762     \exp_after:wN \__fp_ln_Taylor_loop:www
12763     \int_use:N \__int_eval:w #1 - \c_two ;
12764   }
12765   #3;
12766 }
12767 \cs_new:Npn \__fp_ln_Taylor_break:w \fi: #1 \__fp_fixed_add:wwn #2#3; #4 ;;
12768 {
12769   \fi:
12770   \exp_after:wN \__fp_fixed_mul:wwn
12771   \exp_after:wN { \int_use:N \__int_eval:w 10000 + #2 } #3;
12772 }

```

(End definition for __fp_ln_Taylor:wwNw. This function is documented on page ??.)

__fp_ln_c:NwNw __fp_ln_c:NwNw $\langle sign \rangle \{ \langle r_1 \rangle \} \{ \langle r_2 \rangle \} \{ \langle r_3 \rangle \} \{ \langle r_4 \rangle \} \{ \langle r_5 \rangle \} \{ \langle r_6 \rangle \}$; $\langle fixed\ tl \rangle$
 $\langle exponent \rangle$; $\langle continuation \rangle$

We are now reduced to finding $\ln(c)$ and $\langle exponent \rangle \ln(10)$ in a table, and adding it to the mixture. The first step is to get $\ln(c) - \ln(x) = -\ln(a)$, then we get $\ln(10)$ and add or subtract.

For now, $\ln(x)$ is given as $\cdot 10^0$. Unless both the exponent is 1 and $c = 1$, we shift to working in units of $\cdot 10^4$, since the final result will be at least $\ln(10/7) \simeq 0.35$.²¹

```

12773 \cs_new:Npn \__fp_ln_c:NwNw #1 #2; #3
12774 {
12775   \if_meaning:w + #1
12776     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_sub:wwn
12777   \else:
12778     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_add:wwn
12779   \fi:
12780   #3 ; #2 ;
12781 }

```

²²

(End definition for __fp_ln_c:NwNw. This function is documented on page ??.)

__fp_ln_exponent:wn __fp_ln_exponent:wn $\{ \langle s_1 \rangle \} \{ \langle s_2 \rangle \} \{ \langle s_3 \rangle \} \{ \langle s_4 \rangle \} \{ \langle s_5 \rangle \} \{ \langle s_6 \rangle \}$; $\{ \langle exponent \rangle \}$

²¹Bruno: that was wrong at some point, I must check.

²²Bruno: this *must* be updated with correct values!

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.

```

12782 \cs_new:Npn \__fp_ln_exponent:wn #1; #2
12783 {
12784   \if_case:w #2 \exp_stop_f:
12785     \c_zero \__fp_case_return:nw { \__fp_fixed_to_float:Nw 2 }
12786   \or:
12787     \exp_after:wN \__fp_ln_exponent_one:ww \__int_value:w
12788   \else:
12789     \if_int_compare:w #2 > \c_zero
12790       \exp_after:wN \__fp_ln_exponent_small:NNww
12791       \exp_after:wN 0
12792       \exp_after:wN \__fp_fixed_sub:wwn \__int_value:w
12793     \else:
12794       \exp_after:wN \__fp_ln_exponent_small:NNww
12795       \exp_after:wN 2
12796       \exp_after:wN \__fp_fixed_add:wwn \__int_value:w -
12797     \fi:
12798   \fi:
12799   #2; #1;
12800 }

```

Now we painfully write all the cases.²³ No overflow nor underflow can happen, except when computing $\ln(1)$.

```

12801 \cs_new:Npn \__fp_ln_exponent_one:ww 1; #1;
12802 {
12803   \c_zero
12804   \exp_after:wN \__fp_fixed_sub:wwn \c__fp_ln_x_fixed_t1 ; #1;
12805   \__fp_fixed_to_float:wN 0
12806 }

```

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.

```

12807 \cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
12808 {
12809   \c_four
12810   \exp_after:wN \__fp_fixed_mul:wwn
12811     \c__fp_ln_x_fixed_t1 ;
12812     {#3}{0000}{0000}{0000}{0000}{0000} ;
12813   #2
12814     {0000}{#4}{#5}{#6}{#7}{#8};
12815   \__fp_fixed_to_float:wN #1

```

²³Bruno: do rounding.

12816 }

(End definition for _fp_ln_exponent:wn. This function is documented on page ??.)

35.2 Exponential

35.2.1 Sign, exponent, and special numbers

_fp_exp_o:w

```
12817 \cs_new:Npn \_fp_exp_o:w \s__fp \_fp_chk:w #1#2
12818 {
12819   \if_case:w #1 \exp_stop_f:
12820     \_fp_case_return_o:Nw \c_one_fp
12821   \or:
12822     \exp_after:wN \_fp_exp_normal:w
12823   \or:
12824     \if_meaning:w 0 #2
12825       \exp_after:wN \_fp_case_return_o:Nw
12826       \exp_after:wN \c_inf_fp
12827     \else:
12828       \exp_after:wN \_fp_case_return_o:Nw
12829       \exp_after:wN \c_zero_fp
12830     \fi:
12831   \or:
12832     \_fp_case_return_same_o:w
12833   \fi:
12834   \s__fp \_fp_chk:w #1#2
12835 }
```

(End definition for _fp_exp_o:w.)

_fp_exp_normal:w

_fp_exp_pos:Nnwnw

```
12836 \cs_new:Npn \_fp_exp_normal:w \s__fp \_fp_chk:w 1#1
12837 {
12838   \if_meaning:w 0 #1
12839     \_fp_exp_pos:Nnwnw + \_fp_fixed_to_float:wN
12840   \else:
12841     \_fp_exp_pos:Nnwnw - \_fp_fixed_inv_to_float:wN
12842   \fi:
12843 }
12844 \cs_new:Npn \_fp_exp_pos:Nnwnw #1#2#3 \fi: #4#5;
12845 {
12846   \fi:
12847   \exp_after:wN \_fp_sanitize:Nw
12848   \exp_after:wN 0
12849   \_int_value:w #1 \_int_eval:w
12850   \if_int_compare:w #4 < - \c_eight
12851     \c_one
12852     \exp_after:wN \_fp_add_big_i_o:wNww
12853     \int_use:N \_int_eval:w \c_one - #4 ;
12854     0 {1000}{0000}{0000}{0000} ; #5;
```

```

12855         \tex_romannumeral:D
12856     \else:
12857         \if_int_compare:w #4 > \c_five % cf \c__fp_max_exponent_int
12858         \exp_after:wN \__fp_exp_overflow:
12859         \tex_romannumeral:D
12860     \else:
12861         \if_int_compare:w #4 < \c_zero
12862         \exp_after:wN \use_i:nn
12863     \else:
12864         \exp_after:wN \use_ii:nn
12865     \fi:
12866     {
12867         \c_zero
12868         \__fp_decimate:nNnnnn { - #4 }
12869         \__fp_exp_Taylor:Nnnwn
12870     }
12871     {
12872         \__fp_decimate:nNnnnn { \c_sixteen - #4 }
12873         \__fp_exp_pos_large:NnnNwn
12874     }
12875     #5
12876     {#4}
12877     #1 #2 0
12878     \tex_romannumeral:D
12879     \fi:
12880     \fi:
12881     \exp_after:wN \c_zero
12882 }
12883 \cs_new:Npn \__fp_exp_overflow:
12884 { + \c_two * \c__fp_max_exponent_int ; {1000} {0000} {0000} {0000} ; }
(End definition for \__fp_exp_normal:w and \__fp_exp_pos:Nnnwnw.)

```

`__fp_exp_Taylor:Nnnwn`
`__fp_exp_Taylor_loop:www`
`__fp_exp_Taylor_break:Nww`

This function is called for numbers in the range $[10^{-9}, 10^{-1})$. Our only task is to compute the Taylor series. The first argument is irrelevant (rounding digit used by some other functions). The next three arguments, at least 16 digits, delimited by a semicolon, form a fixed point number, so we pack it in blocks of 4 digits.

```

12885 \cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
12886 {
12887     #6
12888     \__fp_pack_twice_four:wNNNNNNNN
12889     \__fp_pack_twice_four:wNNNNNNNN
12890     \__fp_pack_twice_four:wNNNNNNNN
12891     \__fp_exp_Taylor_ii:ww
12892     ; #2#3#4 0000 0000 ;
12893 }
12894 \cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;
12895 { \__fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s_stop }
12896 \cs_new:Npn \__fp_exp_Taylor_loop:www #1; #2; #3;
12897 {

```



```

12898 \if_int_compare:w #1 = \c_one
12899 \exp_after:wN \__fp_exp_Taylor_break:Nww
12900 \fi:
12901 \__fp_fixed_div_int:wwN #3 ; #1 ;
12902 \__fp_fixed_add_one:wN
12903 \__fp_fixed_mul:wwN #2 ;
12904 {
12905 \exp_after:wN \__fp_exp_Taylor_loop:www
12906 \int_use:N \__int_eval:w #1 - 1 ;
12907 #2 ;
12908 }
12909 }
12910 \cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2; #3 \s__stop
12911 { \__fp_fixed_add_one:wN #2 ; }

```

(End definition for __fp_exp_Taylor:Nnnwn. This function is documented on page ??.)

```

\__fp_exp_pos_large:NnnNwn
\__fp_exp_large_after:wwN
\__fp_exp_large:w
\__fp_exp_large_v:wN
\__fp_exp_large_iv:wN
\__fp_exp_large_iii:wN
\__fp_exp_large_ii:wN
\__fp_exp_large_i:wN
\__fp_exp_large_:wN

```

The first two arguments are irrelevant (a rounding digit, and a brace group with 8 zeros). The third argument is the integer part of our number, then we have the decimal part delimited by a semicolon, and finally the exponent, in the range $[0, 5]$. Remove leading zeros from the integer part: putting #4 in there too ensures that an integer part of 0 is also removed. Then read digits one by one, looking up $\exp(\langle digit \rangle \cdot 10^{\langle exponent \rangle})$ in a table, and multiplying that to the current total. The loop is done by having the auxiliary for one exponent call the auxiliary for the next exponent. The current total is expressed by leaving the exponent behind in the input stream (we are currently within an `__int_eval:w`), and keeping track of a fixed point number, #1 for the numbered auxiliaries. Our usage of `\if_case:w` is somewhat dirty for optimization: \TeX jumps to the appropriate case, but we then close the `\if_case:w` “by hand”, using `\or:` and `\fi:` as delimiters.

```

12912 \cs_new:Npn \__fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
12913 {
12914 \exp_after:wN \exp_after:wN
12915 \cs:w \__fp_exp_large\tex_romannumeral:D #6:wN \exp_after:wN \cs_end:
12916 \exp_after:wN \c__fp_one_fixed_tl
12917 \exp_after:wN ;
12918 \__int_value:w #3 #4 \exp_stop_f:
12919 #5 00000 ;
12920 }
12921 \cs_new:Npn \__fp_exp_large:w #1 \or: #2 \fi:
12922 { \fi: \__fp_fixed_mul:wwN #1; }
12923 \cs_new:Npn \__fp_exp_large_v:wN #1; #2
12924 {
12925 \if_case:w #2 ~ \exp_after:wN \__fp_fixed_continue:wN \or:
12926 + 4343 \__fp_exp_large:w {8806}{8182}{2566}{2921}{5872}{6150} \or:
12927 + 8686 \__fp_exp_large:w {7756}{0047}{2598}{6861}{0458}{3204} \or:
12928 + 13029 \__fp_exp_large:w {6830}{5723}{7791}{4884}{1932}{7351} \or:
12929 + 17372 \__fp_exp_large:w {6015}{5609}{3095}{3052}{3494}{7574} \or:
12930 + 21715 \__fp_exp_large:w {5297}{7951}{6443}{0315}{3251}{3576} \or:
12931 + 26058 \__fp_exp_large:w {4665}{6719}{0099}{3379}{5527}{2929} \or:
12932 + 30401 \__fp_exp_large:w {4108}{9724}{3326}{3186}{5271}{5665} \or:
12933 + 34744 \__fp_exp_large:w {3618}{6973}{3140}{0875}{3856}{4102} \or:

```

```

12934         + 39087 \__fp_exp_large:w {3186}{9209}{6113}{3900}{6705}{9685} \or:
12935     \fi:
12936     #1;
12937     \__fp_exp_large_iv:wN
12938 }
12939 \cs_new:Npn \__fp_exp_large_iv:wN #1; #2
12940 {
12941     \if_case:w #2 ~           \exp_after:wN \__fp_fixed_continue:wn \or:
12942     + 435 \__fp_exp_large:w {1970}{0711}{1401}{7046}{9938}{8888} \or:
12943     + 869 \__fp_exp_large:w {3881}{1801}{9428}{4368}{5764}{8232} \or:
12944     + 1303 \__fp_exp_large:w {7646}{2009}{8905}{4704}{8893}{1073} \or:
12945     + 1738 \__fp_exp_large:w {1506}{3559}{7005}{0524}{9009}{7592} \or:
12946     + 2172 \__fp_exp_large:w {2967}{6283}{8402}{3667}{0689}{6630} \or:
12947     + 2606 \__fp_exp_large:w {5846}{4389}{5650}{2114}{7278}{5046} \or:
12948     + 3041 \__fp_exp_large:w {1151}{7900}{5080}{6878}{2914}{4154} \or:
12949     + 3475 \__fp_exp_large:w {2269}{1083}{0850}{6857}{8724}{4002} \or:
12950     + 3909 \__fp_exp_large:w {4470}{3047}{3316}{5442}{6408}{6591} \or:
12951     \fi:
12952     #1;
12953     \__fp_exp_large_iii:wN
12954 }
12955 \cs_new:Npn \__fp_exp_large_iii:wN #1; #2
12956 {
12957     \if_case:w #2 ~           \exp_after:wN \__fp_fixed_continue:wn \or:
12958     + 44 \__fp_exp_large:w {2688}{1171}{4181}{6135}{4484}{1263} \or:
12959     + 87 \__fp_exp_large:w {7225}{9737}{6812}{5749}{2581}{7748} \or:
12960     + 131 \__fp_exp_large:w {1942}{4263}{9524}{1255}{9365}{8421} \or:
12961     + 174 \__fp_exp_large:w {5221}{4696}{8976}{4143}{9505}{8876} \or:
12962     + 218 \__fp_exp_large:w {1403}{5922}{1785}{2837}{4107}{3977} \or:
12963     + 261 \__fp_exp_large:w {3773}{0203}{0092}{9939}{8234}{0143} \or:
12964     + 305 \__fp_exp_large:w {1014}{2320}{5473}{5004}{5094}{5533} \or:
12965     + 348 \__fp_exp_large:w {2726}{3745}{7211}{2566}{5673}{6478} \or:
12966     + 391 \__fp_exp_large:w {7328}{8142}{2230}{7421}{7051}{8866} \or:
12967     \fi:
12968     #1;
12969     \__fp_exp_large_ii:wN
12970 }
12971 \cs_new:Npn \__fp_exp_large_ii:wN #1; #2
12972 {
12973     \if_case:w #2 ~           \exp_after:wN \__fp_fixed_continue:wn \or:
12974     + 5 \__fp_exp_large:w {2202}{6465}{7948}{0671}{6516}{9579} \or:
12975     + 9 \__fp_exp_large:w {4851}{6519}{5409}{7902}{7796}{9107} \or:
12976     + 14 \__fp_exp_large:w {1068}{6474}{5815}{2446}{2146}{9905} \or:
12977     + 18 \__fp_exp_large:w {2353}{8526}{6837}{0199}{8540}{7900} \or:
12978     + 22 \__fp_exp_large:w {5184}{7055}{2858}{7072}{4640}{8745} \or:
12979     + 27 \__fp_exp_large:w {1142}{0073}{8981}{5684}{2836}{6296} \or:
12980     + 31 \__fp_exp_large:w {2515}{4386}{7091}{9167}{0062}{6578} \or:
12981     + 35 \__fp_exp_large:w {5540}{6223}{8439}{3510}{0525}{7117} \or:
12982     + 40 \__fp_exp_large:w {1220}{4032}{9431}{7840}{8020}{0271} \or:
12983     \fi:

```

```

12984     #1;
12985     \__fp_exp_large_i:wN
12986 }
12987 \cs_new:Npn \__fp_exp_large_i:wN #1; #2
12988 {
12989     \if_case:w #2 ~      \exp_after:wN \__fp_fixed_continue:wn \or:
12990     + 1 \__fp_exp_large:w {2718}{2818}{2845}{9045}{2353}{6029} \or:
12991     + 1 \__fp_exp_large:w {7389}{0560}{9893}{0650}{2272}{3043} \or:
12992     + 2 \__fp_exp_large:w {2008}{5536}{9231}{8766}{7740}{9285} \or:
12993     + 2 \__fp_exp_large:w {5459}{8150}{0331}{4423}{9078}{1103} \or:
12994     + 3 \__fp_exp_large:w {1484}{1315}{9102}{5766}{0342}{1116} \or:
12995     + 3 \__fp_exp_large:w {4034}{2879}{3492}{7351}{2260}{8387} \or:
12996     + 4 \__fp_exp_large:w {1096}{6331}{5842}{8458}{5992}{6372} \or:
12997     + 4 \__fp_exp_large:w {2980}{9579}{8704}{1728}{2747}{4359} \or:
12998     + 4 \__fp_exp_large:w {8103}{0839}{2757}{5384}{0077}{1000} \or:
12999     \fi:
13000     #1;
13001     \__fp_exp_large_:wN
13002 }
13003 \cs_new:Npn \__fp_exp_large_:wN #1; #2
13004 {
13005     \if_case:w #2 ~      \exp_after:wN \__fp_fixed_continue:wn \or:
13006     + 1 \__fp_exp_large:w {1105}{1709}{1807}{5647}{6248}{1171} \or:
13007     + 1 \__fp_exp_large:w {1221}{4027}{5816}{0169}{8339}{2107} \or:
13008     + 1 \__fp_exp_large:w {1349}{8588}{0757}{6003}{1039}{8374} \or:
13009     + 1 \__fp_exp_large:w {1491}{8246}{9764}{1270}{3178}{2485} \or:
13010     + 1 \__fp_exp_large:w {1648}{7212}{7070}{0128}{1468}{4865} \or:
13011     + 1 \__fp_exp_large:w {1822}{1188}{0039}{0508}{9748}{7537} \or:
13012     + 1 \__fp_exp_large:w {2013}{7527}{0747}{0476}{5216}{2455} \or:
13013     + 1 \__fp_exp_large:w {2225}{5409}{2849}{2467}{6045}{7954} \or:
13014     + 1 \__fp_exp_large:w {2459}{6031}{1115}{6949}{6638}{0013} \or:
13015     \fi:
13016     #1;
13017     \__fp_exp_large_after:wwn
13018 }
13019 \cs_new:Npn \__fp_exp_large_after:wwn #1; #2; #3
13020 {
13021     \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
13022     \__fp_fixed_mul:wwn #1;
13023 }

```

(End definition for __fp_exp_pos_large:NnnNwn and others.)

35.3 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.

```

13024 \cs_new:cpn { __fp_ \iow_char:N \^_o:ww }
13025   \s__fp \__fp_chk:w #1#2#3; \s__fp \__fp_chk:w #4#5#6;
13026   {
13027     \if_meaning:w 0 #4
13028       \__fp_case_return_o:Nw \c_one_fp
13029     \fi:
13030     \if_case:w #2 \exp_stop_f:
13031       \exp_after:wN \use_i:nn
13032     \or:
13033       \__fp_case_return_o:Nw \c_nan_fp
13034     \else:
13035       \exp_after:wN \__fp_pow_neg:www
13036       \tex_romannumeral:D -'0 \exp_after:wN \use:nn
13037     \fi:
13038     {
13039       \if_meaning:w 1 #1
13040         \exp_after:wN \__fp_pow_normal:ww
13041       \else:

```

```

13042         \exp_after:wN \__fp_pow_zero_or_inf:ww
13043         \fi:
13044         \s__fp \__fp_chk:w #1#2#3;
13045     }
13046     { \s__fp \__fp_chk:w #4#5#6; \s__fp \__fp_chk:w #1#2#3; }
13047     \s__fp \__fp_chk:w #4#5#6;
13048 }

```

(End definition for __fp_~o:ww.)

__fp_pow_zero_or_inf:ww Raising -0 or $-\infty$ to **nan** yields **nan**. For other powers, the result is $+0$ if 0 is raised to a positive power or ∞ to a negative power, and $+\infty$ otherwise. Thus, if the type of a and the sign of b coincide, the result is 0 , since those conveniently take the same possible values, 0 and 2 . Otherwise, either $a = \pm 0$ with $b < 0$ and we have a division by zero, or $a = \pm \infty$ and $b > 0$ and the result is also $+\infty$, but without any exception.

```

13049 \cs_new:Npn \__fp_pow_zero_or_inf:ww \s__fp \__fp_chk:w #1#2; \s__fp \__fp_chk:w #3#4
13050 {
13051     \if_meaning:w 1 #4
13052         \__fp_case_return_same_o:w
13053     \fi:
13054     \if_meaning:w #1 #4
13055         \__fp_case_return_o:Nw \c_zero_fp
13056     \fi:
13057     \if_meaning:w 0 #1
13058         \__fp_case_use:nw
13059         {
13060             \__fp_division_by_zero_o:NNww \c_inf_fp ^
13061             \s__fp \__fp_chk:w #1 #2 ;
13062         }
13063     \else:
13064         \__fp_case_return_o:Nw \c_inf_fp
13065     \fi:
13066     \s__fp \__fp_chk:w #3#4
13067 }

```

(End definition for __fp_pow_zero_or_inf:ww.)

__fp_pow_normal:ww We have in front of us a , and $b \neq 0$, we know that a is a normal number, and we wish to compute $|a|^b$. If $|a| = 1$, we return 1 , unless $a = -1$ and b is **nan**. Indeed, returning 1 at this point would wrongly raise “invalid” when the sign is considered. If $|a| \neq 1$, test the type of b :

0 Impossible, we already filtered $b = \pm 0$.

1 Call __fp_pow_npos:ww.

2 Return $+\infty$ or $+0$ depending on the sign of b and whether the exponent of a is positive or not.

3 Return b .

```

13068 \cs_new:Npn \__fp_pow_normal:ww \s__fp \__fp_chk:w 1 #1#2#3; \s__fp \__fp_chk:w #4#5
13069 {
13070   \if_int_compare:w \pdfTeX_strcmp:D { #2 #3 }
13071     { 1 {1000} {0000} {0000} {0000} } = \c_zero
13072     \if_int_compare:w #4 #1 = 32 \exp_stop_f:
13073     \exp_after:wN \__fp_case_return_ii_o:ww
13074   \fi:
13075   \__fp_case_return_o:Nww \c_one_fp
13076 \fi:
13077 \if_case:w #4 \exp_stop_f:
13078 \or:
13079   \exp_after:wN \__fp_pow_npos:Nww
13080   \exp_after:wN #5
13081 \or:
13082   \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
13083   \if_int_compare:w #2 > \c_zero
13084     \exp_after:wN \__fp_case_return_o:Nww
13085     \exp_after:wN \c_inf_fp
13086   \else:
13087     \exp_after:wN \__fp_case_return_o:Nww
13088     \exp_after:wN \c_zero_fp
13089   \fi:
13090 \or:
13091   \__fp_case_return_ii_o:ww
13092 \fi:
13093 \s__fp \__fp_chk:w 1 #1 {#2} #3 ;
13094 \s__fp \__fp_chk:w #4 #5
13095 }

```

(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.

```

13096 \cs_new:Npn \__fp_pow_npos:Nww #1 \s__fp \__fp_chk:w 1#2#3
13097 {
13098   \exp_after:wN \__fp_sanitize:Nw
13099   \exp_after:wN 0
13100   \__int_value:w
13101   \if:w #1 \if_int_compare:w #3 > \c_zero 0 \else: 2 \fi:
13102     \exp_after:wN \__fp_pow_npos_aux:NNww
13103     \exp_after:wN +
13104     \exp_after:wN \__fp_fixed_to_float:wN
13105   \else:
13106     \exp_after:wN \__fp_pow_npos_aux:NNww
13107     \exp_after:wN -

```

```

13108         \exp_after:wN \__fp_fixed_inv_to_float:wN
13109         \fi:
13110         {#3}
13111     }

```

(End definition for __fp_pow_npos:Nww.)

__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 $-\ln(x)$.

```

13112 \cs_new:Npn \__fp_pow_npos_aux:NNnw #1#2#3#4#5; \s_fp \__fp_chk:w 1#6#7#8;
13113 {
13114     #1
13115     \__int_eval:w
13116     \__fp_ln_significand:NNNNnnnN #4#5
13117     \__fp_pow_exponent:wnN {#3}
13118     \__fp_fixed_mul:wnn #8 {0000}{0000} ;
13119     \__fp_pow_B:wnN #7;
13120     #1 #2 0 % fixed_to_float:wN
13121 }
13122 \cs_new:Npn \__fp_pow_exponent:wnN #1; #2
13123 {
13124     \if_int_compare:w #2 > \c_zero
13125     \exp_after:wN \__fp_pow_exponent:Nwnnnnnwn % n\ln(10) - (-\ln(x))
13126     \exp_after:wN +
13127     \else:
13128     \exp_after:wN \__fp_pow_exponent:Nwnnnnnwn % -( |n|\ln(10) + (-\ln(x)) )
13129     \exp_after:wN -
13130     \fi:
13131     #2; #1;
13132 }
13133 \cs_new:Npn \__fp_pow_exponent:Nwnnnnnwn #1#2; #3#4#5#6#7#8; #9
13134 { %^A todo: use that in ln.
13135     \exp_after:wN \__fp_fixed_mul_after:wN
13136     \int_use:N \__int_eval:w \c__fp_leading_shift_int
13137     \exp_after:wN \__fp_pack:NNNNNwn
13138     \int_use:N \__int_eval:w \c__fp_middle_shift_int
13139     #1#2*23025 - #1 #3
13140     \exp_after:wN \__fp_pack:NNNNNwn
13141     \int_use:N \__int_eval:w \c__fp_middle_shift_int
13142     #1 #2*8509 - #1 #4
13143     \exp_after:wN \__fp_pack:NNNNNwn
13144     \int_use:N \__int_eval:w \c__fp_middle_shift_int
13145     #1 #2*2994 - #1 #5
13146     \exp_after:wN \__fp_pack:NNNNNwn
13147     \int_use:N \__int_eval:w \c__fp_middle_shift_int
13148     #1 #2*0456 - #1 #6
13149     \exp_after:wN \__fp_pack:NNNNNwn
13150     \int_use:N \__int_eval:w \c__fp_trailing_shift_int
13151     #1 #2*8401 - #1 #7
13152     #1 ( #2*7991 - #8 ) / 1 0000 ; {#9} ;

```

```

13153 }
13154 \cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
13155 {
13156   \if_int_compare:w #7 < \c_zero
13157     \exp_after:wN \__fp_pow_C_neg:w \__int_value:w -
13158   \else:
13159     \if_int_compare:w #7 < 22 \exp_stop_f:
13160       \exp_after:wN \__fp_pow_C_pos:w \__int_value:w
13161     \else:
13162       \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
13163     \fi:
13164   \fi:
13165   #7 \exp_after:wN ;
13166   \int_use:N \__int_eval:w 10 0000 + #1 \__int_eval_end:
13167   #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^A todo: how many 0?
13168 }
13169 \cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
13170 {
13171   + \c_two * \c__fp_max_exponent_int
13172   \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_t1 ;
13173 }
13174 \cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
13175 {
13176   \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
13177   \prg_replicate:nn {#1} {0}
13178 }
13179 \cs_new:Npn \__fp_pow_C_pos:w #1; 1
13180 { \__fp_pow_C_pos_loop:wN #1; }
13181 \cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
13182 {
13183   \if_meaning:w 0 #1
13184     \exp_after:wN \__fp_pow_C_pack:w
13185     \exp_after:wN #2
13186   \else:
13187     \if_meaning:w 0 #2
13188       \exp_after:wN \__fp_pow_C_pos_loop:wN \__int_value:w
13189     \else:
13190       \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
13191     \fi:
13192     \__int_eval:w #1 - \c_one \exp_after:wN ;
13193   \fi:
13194 }
13195 \cs_new:Npn \__fp_pow_C_pack:w
13196 { \exp_after:wN \__fp_exp_large_v:wN \c__fp_one_fixed_t1 ; }
(End definition for \__fp_pow_npos_aux:NNnww.)

```

`__fp_pow_neg:www` 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_-_o:w`. 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.

```

13197 \cs_new:Npn \_fp_pow_neg:www \s_fp \_fp_chk:w #1#2; #3; #4;
13198 {
13199   \if_case:w \_fp_pow_neg_case:w #4 ;
13200   \cs:w \_fp_-_o:w \exp_after:wN \cs_end:
13201   \or:
13202     \if_int_compare:w \_int_eval:w #1 / \c_two = \c_one
13203     \_fp_invalid_operation_o:Nww ^ #3; #4;
13204     \tex_romannumeral:D -'0
13205     \exp_after:wN \exp_after:wN
13206     \exp_after:wN \_fp_use_none_until_s:w
13207   \fi:
13208   \fi:
13209   \_fp_exp_after_o:w
13210   \s_fp \_fp_chk:w #1#2;
13211 }

```

(End definition for `_fp_pow_neg:www`.)

```

\_fp_pow_neg_case:w
\_fp_pow_neg_case_aux:nnnnn
\_fp_pow_neg_case_aux:NNNNNNNNw

```

This function expects a floating point number, and “returns” -1 if it is an even integer, 0 if it is an odd integer, and 1 if it is not an integer. Zeros are even, $\pm\infty$ and `nan` are non-integers. The sign of normal numbers is irrelevant to parity. If the exponent is greater than sixteen, then the number is even. If the exponent is non-positive, the number cannot be an integer. We also separate the ranges of exponent $[1, 8]$ and $[9, 16]$. In the former case, check that the last 8 digits are zero (otherwise we don’t have an integer). In both cases, consider the appropriate 8 digits, either `#4#5` or `#2#3`, remove the first few: we are then left with $\langle digit \rangle \langle digits \rangle$; which would be the digits surrounding the decimal period. If the $\langle digits \rangle$ are non-zero, the number is not an integer. Otherwise, check the parity of the $\langle digit \rangle$ and return `\c_zero` or `\c_minus_one`.

```

13212 \cs_new:Npn \_fp_pow_neg_case:w \s_fp \_fp_chk:w #1#2#3;
13213 {
13214   \if_case:w #1 \exp_stop_f:
13215     \c_minus_one
13216   \or: \_fp_pow_neg_case_aux:nnnnn #3
13217   \else: \c_one
13218   \fi:
13219 }
13220 \cs_new:Npn \_fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
13221 {
13222   \if_int_compare:w #1 > \c_eight
13223   \if_int_compare:w #1 > \c_sixteen
13224     \c_minus_one
13225   \else:
13226     \exp_after:wN \exp_after:wN
13227     \exp_after:wN \_fp_pow_neg_case_aux:NNNNNNNNw
13228     \prg_replicate:nn { \c_sixteen - #1 } { 0 } #4#5 ;
13229   \fi:

```

```

13230 \else:
13231 \if_int_compare:w #1 > \c_zero
13232 \if_int_compare:w #4#5 = \c_zero
13233 \exp_after:wN \exp_after:wN
13234 \exp_after:wN \_fp_pow_neg_case_aux:NNNNNNNw
13235 \prg_replicate:nn { \c_eight - #1 } { 0 } #2#3 ;
13236 \else:
13237 \c_one
13238 \fi:
13239 \else:
13240 \c_one
13241 \fi:
13242 \fi:
13243 }
13244 \cs_new:Npn \_fp_pow_neg_case_aux:NNNNNNNw #1#2#3#4#5#6#7#8#9;
13245 {
13246 \if_int_compare:w 0 #9 = \c_zero
13247 \if_int_odd:w #8 \exp_stop_f:
13248 \c_zero
13249 \else:
13250 \c_minus_one
13251 \fi:
13252 \else:
13253 \c_one
13254 \fi:
13255 }

```

(End definition for `_fp_pow_neg_case:w`, `_fp_pow_neg_case_aux:nnnnn`, and `_fp_pow_neg_case_aux:NNNNNNNw`.)

```

13256 </initex | package>

```

36 Implementation

```
13257 <*initex | package>
```

```
13258 <@@=fp>
```

36.1 Direct trigonometric functions

The approach for all trigonometric functions (sine, cosine, tangent, cotangent, cosecant, and secant) is the same.

- Filter out special cases (± 0 , $\pm \inf$ and NaN).
- Keep the sign for later, and work with the absolute value x of the argument.
- For numbers less than 1, shift the significand to convert them to fixed point numbers. Very small numbers take a slightly different route.
- For numbers ≥ 1 , subtract a multiple of $\pi/2$ to bring them to the range to $[0, \pi/2]$. (This is called argument reduction.)

- Reduce further to $[0, \pi/4]$ using $\sin x = \cos(\pi/2 - x)$.
- Use the appropriate power series depending on the octant $\lfloor \frac{x}{\pi/4} \rfloor \bmod 8$, the sign, and the function to compute.

36.1.1 Sign and special numbers

`__fp_sin_o:w` The sine of ± 0 or NaN is the same floating point number. The sine of $\pm\infty$ raises an invalid operation exception. Otherwise, `__fp_trig_exponent:NNNNNwn` checks the exponent: if the number is tiny, use `__fp_trig_epsilon_o:w` which returns $\sin \epsilon = \epsilon$. For larger inputs, use the series `__fp_sin_series:NNwww` after argument reduction. In this second case, we will use a sign #2, an initial octant of 0, and convert the result of the series to a floating point directly, since $\sin(x) = \#2 \sin|x|$.

```

13259 \cs_new:Npn \__fp_sin_o:w \s__fp \__fp_chk:w #1#2
13260 {
13261   \if_case:w #1 \exp_stop_f:
13262     \__fp_case_return_same_o:w
13263   \or:
13264     \__fp_case_use:nw
13265     {
13266       \__fp_trig_exponent:NNNNNwn \__fp_trig_epsilon_o:w
13267       \__fp_sin_series:NNwww \__fp_fixed_to_float:wN #2 \c_zero
13268     }
13269   \or: \__fp_case_use:nw { \__fp_invalid_operation_o:nw { sin } }
13270   \else: \__fp_case_return_same_o:w
13271   \fi:
13272   \s__fp \__fp_chk:w #1#2
13273 }

```

(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, `__fp_trig_exponent:NNNNNwn` checks the exponent: if the number is tiny, use `__fp_trig_epsilon_one_o:w` which returns $\cos \epsilon = 1$. For larger inputs, use the same series as for sine, but using a positive sign 0 and with an initial octant of 2, because $\cos(x) = +\sin(\pi/2 + |x|)$.

```

13274 \cs_new:Npn \__fp_cos_o:w \s__fp \__fp_chk:w #1#2
13275 {
13276   \if_case:w #1 \exp_stop_f:
13277     \__fp_case_return_o:Nw \c_one_fp
13278   \or:
13279     \__fp_case_use:nw
13280     {
13281       \__fp_trig_exponent:NNNNNwn \__fp_trig_epsilon_one_o:w
13282       \__fp_sin_series:NNwww \__fp_fixed_to_float:wN 0 \c_two
13283     }
13284   \or: \__fp_case_use:nw { \__fp_invalid_operation_o:nw { cos } }
13285   \else: \__fp_case_return_same_o:w
13286   \fi:

```

```

13287     \s__fp \__fp_chk:w #1#2
13288 }

```

(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:Nnw defined below). The cosecant of $\pm\infty$ raises an invalid operation exception. The cosecant of NaN is itself. Otherwise, __fp_trig_exponent:NNNNNwn checks the exponent: if the number is tiny, use __fp_trig_epsilon_inv_o:w which returns $\csc \epsilon = 1/\epsilon$. For larger inputs, use the same series as for sine, using the sign #2, a starting octant of 0, and inverting during the conversion from the fixed point sine to the floating point result, because $\csc(x) = \#2(\sin|x|)^{-1}$.

```

13289 \cs_new:Npn \__fp_csc_o:w \s__fp \__fp_chk:w #1#2
13290 {
13291     \if_case:w #1 \exp_stop_f:
13292         \__fp_cot_zero_o:Nnw #2 { csc }
13293     \or:
13294         \__fp_case_use:nw
13295         {
13296             \__fp_trig_exponent:NNNNNwn \__fp_trig_epsilon_inv_o:w
13297             \__fp_sin_series:NNwww \__fp_fixed_inv_to_float:wN #2 \c_zero
13298         }
13299     \or: \__fp_case_use:nw { \__fp_invalid_operation_o:nw { csc } }
13300     \else: \__fp_case_return_same_o:w
13301     \fi:
13302     \s__fp \__fp_chk:w #1#2
13303 }

```

(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, __fp_trig_exponent:NNNNNwn checks the exponent: if the number is tiny, use __fp_trig_epsilon_one_o:w which returns $\sec \epsilon = 1$. For larger inputs, use the same series as for sine, using a positive sign 0, a starting octant of 2, and inverting upon conversion, because $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```

13304 \cs_new:Npn \__fp_sec_o:w \s__fp \__fp_chk:w #1#2
13305 {
13306     \if_case:w #1 \exp_stop_f:
13307         \__fp_case_return_o:Nw \c_one_fp
13308     \or:
13309         \__fp_case_use:nw
13310         {
13311             \__fp_trig_exponent:NNNNNwn \__fp_trig_epsilon_one_o:w
13312             \__fp_sin_series:NNwww \__fp_fixed_inv_to_float:wN 0 \c_two
13313         }
13314     \or: \__fp_case_use:nw { \__fp_invalid_operation_o:nw { sec } }
13315     \else: \__fp_case_return_same_o:w
13316     \fi:
13317     \s__fp \__fp_chk:w #1#2
13318 }

```

(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. Otherwise, `_fp_trig_exponent:NNNNNwn` checks the exponent: if the number is tiny, use `_fp_trig_epsilon_o:w` which returns $\tan \epsilon = \epsilon$. For larger inputs, use `_fp_tan_series_o:NNwww` for the calculation after argument reduction, with a sign #2 and an initial octant of 1 (this shift is somewhat arbitrary). See `_fp_cot_o:w` for an explanation of the 0 argument.

```

13319 \cs_new:Npn \_fp_tan_o:w \s_fp \_fp_chk:w #1#2
13320 {
13321   \if_case:w #1 \exp_stop_f:
13322     \_fp_case_return_same_o:w
13323   \or:
13324     \_fp_case_use:nw
13325     {
13326       \_fp_trig_exponent:NNNNNwn \_fp_trig_epsilon_o:w
13327       \_fp_tan_series_o:NNwww 0 #2 \c_one
13328     }
13329   \or: \_fp_case_use:nw { \_fp_invalid_operation_o:nw { tan } }
13330   \else: \_fp_case_return_same_o:w
13331   \fi:
13332   \s_fp \_fp_chk:w #1#2
13333 }

```

(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:Nnw`). 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.

```

13334 \cs_new:Npn \_fp_cot_o:w \s_fp \_fp_chk:w #1#2
13335 {
13336   \if_case:w #1 \exp_stop_f:
13337     \_fp_cot_zero_o:Nnw #2 { cot }
13338   \or:
13339     \_fp_case_use:nw
13340     {
13341       \_fp_trig_exponent:NNNNNwn \_fp_trig_epsilon_inv_o:w
13342       \_fp_tan_series_o:NNwww 2 #2 \c_three
13343     }
13344   \or: \_fp_case_use:nw { \_fp_invalid_operation_o:nw { cot } }
13345   \else: \_fp_case_return_same_o:w
13346   \fi:
13347   \s_fp \_fp_chk:w #1#2
13348 }
13349 \cs_new:Npn \_fp_cot_zero_o:Nnw #1 #2 #3 \fi:

```

```

13350 {
13351   \fi:
13352   \if_meaning:w 0 #1
13353     \exp_after:wN \__fp_division_by_zero_o:Nnw \exp_after:wN \c_inf_fp
13354   \else:
13355     \exp_after:wN \__fp_division_by_zero_o:Nnw \exp_after:wN \c_minus_inf_fp
13356   \fi:
13357   {#2}
13358 }

```

(End definition for __fp_cot_o:w. This function is documented on page ??.)

36.1.2 Small and tiny arguments

__fp_trig_exponent:NNNNNwn The first five arguments control what trigonometric function we compute, then follows a normal floating point number. If the floating point is smaller than 10^{-8} , then call the `_epsilon` auxiliary #1. Otherwise, call the function #2, with arguments #3; #4; 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. Argument reduction leaves a shift into the integer expression for the octant. Numbers less than 1 are converted using `__fp_trig_small:w` which simply shifts the significand, while large numbers need argument reduction.

```

13359 \cs_new:Npn \__fp_trig_exponent:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7
13360 {
13361   \if_int_compare:w #7 > - \c_eight
13362     \exp_after:wN #2
13363     \exp_after:wN #3
13364     \exp_after:wN #4
13365     \int_use:N \__int_eval:w #5
13366     \if_int_compare:w #7 > \c_zero
13367       \exp_after:wN \__fp_trig_large:ww \__int_value:w
13368     \else:
13369       \exp_after:wN \__fp_trig_small:ww \__int_value:w
13370     \fi:
13371   \else:
13372     \exp_after:wN #1
13373     \exp_after:wN #6
13374   \fi:
13375   #7 ;
13376 }

```

(End definition for __fp_trig_exponent:NNNNNwn.)

__fp_trig_epsilon_o:w Sine and tangent of tiny numbers give the number itself: the relative error is less than $5 \cdot 10^{-17}$, which is appropriate. Cosine and secant simply give 1. Cotangent and cosecant compute $1/\epsilon$. This is actually slightly wrong because further terms in the power series could affect the rounding for cotangent.

```

13377 \cs_new:Npn \__fp_trig_epsilon_o:w #1 #2 ;
13378 { \__fp_exp_after_o:w \s__fp \__fp_chk:w 1 #1 {#2} }
13379 \cs_new:Npn \__fp_trig_epsilon_one_o:w #1 ; #2 ;

```

```

13380 { \exp_after:wN \c_one_fp }
13381 \group_begin:
13382 \char_set_catcode_letter:N /
13383 \cs_new:Npn \__fp_trig_epsilon_inv_o:w #1 #2 ;
13384 {
13385   \exp_after:wN \__fp/_o:ww
13386   \c_one_fp
13387   \s__fp \__fp_chk:w 1 #1 {#2}
13388 }
13389 \group_end:
(End definition for \__fp_trig_epsilon_o:w, \__fp_trig_epsilon_one_o:w, and \__fp_trig_epsilon_inv_o:w.)

```

`__fp_trig_small:ww` Floating point numbers less than 1 are converted to fixed point numbers by prepending a number of zeroes to the significand. Since we have already filtered out numbers less than 10^{-8} , we add at most 7 zeroes, hence no digit is lost in converting to a fixed point number.

```

13390 \cs_new:Npn \__fp_trig_small:ww #1; #2#3#4#5;
13391 {
13392   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
13393   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
13394   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
13395   \exp_after:wN .
13396   \exp_after:wN ;
13397   \tex_romannumeral:D -'0
13398   \prg_replicate:nn { - #1 } { 0 } #2#3#4#5 0000 0000 ;
13399 }
(End definition for \__fp_trig_small:ww.)

```

36.1.3 Reduction of large arguments

In the case of a floating point argument greater or equal to 1, we need to perform argument reduction.

`__fp_trig_large:ww` We shift the significand by one digit at a time, subtracting a multiple of 2π at each step.
`__fp_trig_large:www` We use a value of 2π rounded up, consistent with the choice of `\c_pi_fp`. This is not quite correct from an accuracy perspective, but has the nice property that $\sin(180\deg) = 0$ exactly. The arguments of `__fp_trig_large:www` are a leading block of up to 5 digits, three brace groups of 4 digits each, and the exponent, decremented at each step. The multiple of 2π to subtract is estimated as $\lceil \#1/6283 \rceil$ (the formula chosen always gives a non-negative integer). The subtraction has a form similar to our usual multiplications (see `l3fp-basics` or `l3fp-extended`). Once the exponent reaches 0, we are done subtracting 2π , and we call `__fp_trig_octant_loop:nnnnnw` to do the reduction by $\pi/2$.
`__fp_trig_large_o:wnnnn`
`__fp_trig_large_break:w`

```

13400 \cs_new:Npn \__fp_trig_large:ww #1; #2#3;
13401 { \__fp_trig_large:www #2; #3 ; #1; }
13402 \cs_new:Npn \__fp_trig_large:www #1; #2; #3;
13403 {
13404   \if_meaning:w 0 #3 \__fp_trig_large_break:w \fi:
13405   \exp_after:wN \__fp_trig_large_o:wnnnn

```

```

13406 \int_use:N \__int_eval:w ( #1 - 3141 ) / 6283 ;
13407 {#1} #2
13408 \exp_after:wN ;
13409 \int_use:N \__int_eval:w \c_minus_one + #3;
13410 }
13411 \cs_new:Npn \__fp_trig_large_o:w nnnn #1; #2#3#4#5
13412 {
13413 \exp_after:wN \__fp_trig_large:www
13414 \int_use:N \__int_eval:w \c__fp_leading_shift_int + #20 - #1*62831
13415 \exp_after:wN \__fp_pack:NNNNw
13416 \int_use:N \__int_eval:w \c__fp_middle_shift_int + #30 - #1*8530
13417 \exp_after:wN \__fp_pack:NNNNw
13418 \int_use:N \__int_eval:w \c__fp_middle_shift_int + #40 - #1*7179
13419 \exp_after:wN \__fp_pack:NNNNw
13420 \int_use:N \__int_eval:w \c__fp_trailing_shift_int + #50 - #1*5880
13421 \exp_after:wN ;
13422 }
13423 \cs_new:Npn \__fp_trig_large_break:w \fi: #1; #2;
13424 { \fi: \__fp_trig_octant_loop:nnnnw #2 {0000} {0000} ; }
(End definition for \__fp_trig_large:ww and others.)

```

$\backslash_fp_trig_octant_loop:nnnnw$
 $\backslash_fp_trig_octant_break:w$

We receive a fixed point number as argument. As long as it is greater than half of $\backslash c_pi_fp$, namely 1.5707963267948970, subtract that fixed-point approximation of $\pi/2$, and leave $+ \backslash c_two$ in the integer expression for the octant. Once the argument becomes smaller, break the initial loop. If the number is greater than 0.7854 (overestimate of $\pi/4$), then compute $\pi/2 - x$ and increment the octant. The result is in all cases in the range $[0, 0.7854]$, appropriate for the series expansions.

```

13425 \cs_new:Npn \__fp_trig_octant_loop:nnnnw #1#2#3#4#5#6;
13426 {
13427 \if_int_compare:w #1#2 < 157079633 \exp_stop_f:
13428 \if_int_compare:w #1#2 = 157079632 \exp_stop_f:
13429 \if_int_compare:w #3#4 > 67948969 \exp_stop_f:
13430 \use_i_i:nnn
13431 \fi:
13432 \fi:
13433 \__fp_trig_octant_break:w
13434 \fi:
13435 + \c_two
13436 \__fp_fixed_sub:wwn
13437 {#1} {#2} {#3} {#4} {0000} {0000} ;
13438 {15707} {9632} {6794} {8970} {0000} {0000} ;
13439 \__fp_trig_octant_loop:nnnnw
13440 }
13441 \cs_new:Npn \__fp_trig_octant_break:w #1 \fi: + #2#3 #4#5; #6; #7;
13442 {
13443 \fi:
13444 \if_int_compare:w #4 < 7854 \exp_stop_f:
13445 \exp_after:wN \__fp_use_i_until_s:nw
13446 \exp_after:wN .

```



```

13447 \fi:
13448 + \c_one
13449 \__fp_fixed_sub:wn #6 ; {#4} #5 ; . ;
13450 }

```

(End definition for `__fp_trig_octant_loop:nnnnnw` and `__fp_trig_octant_break:w`.)

36.2 Computing the power series

```

\__fp_sin_series:NNwww
\__fp_sin_series_aux:NNnww

```

Here we receive a conversion function `__fp_fixed_to_float:wN` or `__fp_fixed_inv_to_float:wN`, a $\langle sign \rangle$ (0 or 2), a (non-negative) $\langle octant \rangle$ delimited by a dot, a $\langle fixed\ point \rangle$ number, and junk delimited by a semicolon. The auxiliary receives:

- The final sign, which depends on the octant #3 and the original sign #2,
- The octant #3, which will control the series we use.
- The square #4 * #4 of the argument, computed with `__fp_fixed_mul:wn`.
- The number itself.

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 fixed point number is converted to a floating point number with the given sign, and `__fp_sanitize:Nw` checks for overflow and underflow.

```

13451 \cs_new:Npn \__fp_sin_series:NNwww #1#2#3 . #4; #5;
13452 {
13453   \__fp_fixed_mul:wn #4; #4;
13454   {
13455     \exp_after:wN \__fp_sin_series_aux:NNnww
13456     \exp_after:wN #1
13457     \__int_value:w
13458     \if_int_odd:w \__int_eval:w ( #3 + \c_two ) / \c_four \__int_eval_end:
13459       #2
13460     \else:
13461       \if_meaning:w #2 0 2 \else: 0 \fi:
13462       \fi:
13463     {#3}
13464   }
13465   #4 ;
13466 }
13467 \cs_new:Npn \__fp_sin_series_aux:NNnww #1#2#3 #4; #5;
13468 {

```

```

13469 \if_int_odd:w \__int_eval:w #3 / \c_two \__int_eval_end:
13470 \exp_after:wN \use_i:nn
13471 \else:
13472 \exp_after:wN \use_ii:nn
13473 \fi:
13474 { % 1/18!
13475 \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0001}{5619}{2070};
13476 #4; {0000}{0000}{0000}{0477}{9477}{3324};
13477 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0000}{0011}{4707}{4559}{7730};
13478 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0000}{2087}{6756}{9878}{6810};
13479 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0027}{5573}{1922}{3985}{8907};
13480 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{2480}{1587}{3015}{8730}{1587};
13481 \__fp_fixed_mul_sub_back:wwwn #4; {0013}{8888}{8888}{8888}{8888}{8889};
13482 \__fp_fixed_mul_sub_back:wwwn #4; {0416}{6666}{6666}{6666}{6666}{6667};
13483 \__fp_fixed_mul_sub_back:wwwn #4; {5000}{0000}{0000}{0000}{0000}{0000};
13484 \__fp_fixed_mul_sub_back:wwwn #4; {10000}{0000}{0000}{0000}{0000}{0000};
13485 }
13486 { % 1/17!
13487 \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0028}{1145}{7254};
13488 #4; {0000}{0000}{0000}{7647}{1637}{3182};
13489 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0000}{0160}{5904}{3836}{8216};
13490 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0002}{5052}{1083}{8544}{1719};
13491 \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0275}{5731}{9223}{9858}{9065};
13492 \__fp_fixed_mul_sub_back:wwwn #4; {0001}{9841}{2698}{4126}{9841}{2698};
13493 \__fp_fixed_mul_sub_back:wwwn #4; {0083}{3333}{3333}{3333}{3333}{3333};
13494 \__fp_fixed_mul_sub_back:wwwn #4; {1666}{6666}{6666}{6666}{6666}{6667};
13495 \__fp_fixed_mul_sub_back:wwwn #4; {10000}{0000}{0000}{0000}{0000}{0000};
13496 \__fp_fixed_mul:wwn #5;
13497 }
13498 {
13499 \exp_after:wN \__fp_sanitize:Nw
13500 \exp_after:wN #2
13501 \int_use:N \__int_eval:w #1
13502 }
13503 #2
13504 }

```

(End definition for __fp_sin_series:NNwww and __fp_sin_series_aux:NNnww.)

__fp_tan_series_o:NNwww
__fp_tan_series_aux_o:Nnww

Contrarily to __fp_sin_series:NNwww which received the 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 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 itself is computed by `__fp_fixed_div_to_float:ww`, which converts it directly to a floating point number to avoid rounding issues. For octants #2 (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.

```

13505 \cs_new:Npn \__fp_tan_series_o:NNwww #1#2#3. #4; #5;
13506 {
13507   \__fp_fixed_mul:wwn #4; #4;
13508   {
13509     \exp_after:wN \__fp_tan_series_aux_o:Nnww
13510     \__int_value:w
13511     \if_int_odd:w \__int_eval:w #3 / \c_two \__int_eval_end:
13512     \exp_after:wN \reverse_if:N
13513     \fi:
13514     \if_meaning:w #1#2 2 \else: 0 \fi:
13515     {#3}
13516   }
13517   #4 ;
13518 }
13519 \cs_new:Npn \__fp_tan_series_aux_o:Nnww #1 #2 #3; #4;
13520 {
13521   \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{1527}{3493}{0856}{7059};
13522   #3; {0000}{0159}{6080}{0274}{5257}{6472};
13523   \__fp_fixed_mul_sub_back:wwwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
13524   \__fp_fixed_mul_sub_back:wwwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
13525   \__fp_fixed_mul_sub_back:wwwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
13526   \__fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
13527   \__fp_fixed_mul:wwn #4;
13528   {
13529     \__fp_fixed_mul_sub_back:wwwn {0000}{0007}{0258}{0681}{9408}{4706};
13530     #3; {0000}{2343}{7175}{1399}{6151}{7670};
13531     \__fp_fixed_mul_sub_back:wwwn #3; {0019}{2638}{4588}{9232}{8861}{3691};
13532     \__fp_fixed_mul_sub_back:wwwn #3; {0536}{6357}{0691}{4344}{6852}{4252};
13533     \__fp_fixed_mul_sub_back:wwwn #3; {5263}{1578}{9473}{6842}{1052}{6315};
13534     \__fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
13535     {
13536       \exp_after:wN \__fp_sanitize:Nw
13537       \exp_after:wN #1
13538       \int_use:N \__int_eval:w
13539       \reverse_if:N \if_int_odd:w
13540       \__int_eval:w (#2 - \c_one) / \c_two \__int_eval_end:
13541       \exp_after:wN \__fp_reverse_args:Nww
13542       \fi:
13543       \__fp_fixed_div_to_float:ww

```

```

13544         }
13545     }
13546 }
(End definition for \__fp_tan_series_o:NNwww and \__fp_tan_series_aux_o:Nnww.)
13547 </initex | package>

```

37 13fp-convert implementation

```

13548 <*initex | package>
13549 <@@=fp>

```

37.1 Trimming trailing zeros

`__fp_trim_zeros:w` If #1 ends with a 0, the loop auxiliary takes that zero as an end-delimiter for its first argument, and the second argument is the same loop auxiliary. Once the last trailing zero is reached, the second argument will be the dot auxiliary, which removes a trailing dot if any. We then clean-up with the `end` auxiliary, keeping only the number.

```

13550 \cs_new:Npn \__fp_trim_zeros:w #1 ;
13551 {
13552     \__fp_trim_zeros_loop:w #1
13553     ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s__stop
13554 }
13555 \cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
13556 \cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
13557 \cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__stop { #1 }

```

(End definition for `__fp_trim_zeros:w`. This function is documented on page ??.)

37.2 Scientific notation

`\fp_to_scientific:N` The three public functions evaluate their argument, then pass it to `__fp_to_scientific_dispatch:w`.

```

\fp_to_scientific:c
\fp_to_scientific:n
13558 \cs_new:Npn \fp_to_scientific:N #1
13559 { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
13560 \cs_generate_variant:Nn \fp_to_scientific:N { c }
13561 \cs_new_nopar:Npn \fp_to_scientific:n
13562 {
13563     \exp_after:wN \__fp_to_scientific_dispatch:w
13564     \tex_romannumeral:D -'0 \__fp_parse:n
13565 }

```

(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` 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”.

```

13566 \group_begin:
13567 \char_set_catcode_other:N E
13568 \tl_to_lowercase:n
13569 {
13570   \group_end:
13571   \cs_new:Npn \__fp_to_scientific_dispatch:w \s__fp \__fp_chk:w #1#2
13572   {
13573     \if_meaning:w 2 #2 \exp_after:wN - \tex_romannumeral:D -'0 \fi:
13574     \if_case:w #1 \exp_stop_f:
13575       \__fp_case_return:nw { 0 }
13576     \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
13577     \or:
13578       \__fp_case_use:nw
13579       {
13580         \__fp_invalid_operation:nnw
13581         {
13582           \exp_after:wN 1
13583           \exp_after:wN E
13584           \int_use:N \c__fp_max_exponent_int
13585         }
13586         { fp_to_scientific }
13587       }
13588     \or:
13589       \__fp_case_use:nw
13590       {
13591         \__fp_invalid_operation:nnw
13592         { 0 }
13593         { fp_to_scientific }
13594       }
13595     \fi:
13596     \s__fp \__fp_chk:w #1 #2
13597   }
13598   \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
13599   \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
13600   {
13601     \if_int_compare:w #2 = \c_one
13602       \exp_after:wN \__fp_to_scientific_normal:wNw
13603     \else:
13604       \exp_after:wN \__fp_to_scientific_normal:wNw
13605       \exp_after:wN E
13606       \int_use:N \__int_eval:w #2 - \c_one
13607     \fi:
13608     ; #3 #4 #5 #6 ;
13609   }

```

```

13610 }
13611 \cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
13612 { \__fp_trim_zeros:w #2.#3 ; #1 }
(End definition for \__fp_to_scientific_dispatch:w, \__fp_to_scientific_normal:wnnnnn, and \__fp_to_scientific_normal:wNw)

```

37.3 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`

```

13613 \cs_new:Npn \fp_to_decimal:N #1
13614 { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
13615 \cs_generate_variant:Nn \fp_to_decimal:N { c }
13616 \cs_new_nopar:Npn \fp_to_decimal:n
13617 {
13618   \exp_after:wN \__fp_to_decimal_dispatch:w
13619   \tex_romannumeral:D -'0 \__fp_parse:n
13620 }

```

(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:wnnnnn

```

The structure is similar to `__fp_to_scientific_dispatch:w`. Insert - for negative numbers. Zero gives 0, $\pm\infty$ and NaN yield an “invalid operation” exception; note that $\pm\infty$ produces a very large output, which we don’t expand now since it most likely won’t be needed. Normal numbers with an exponent in the range [1,15] have that number of digits before the decimal separator: “decimate” them, and remove leading zeros with `__int_value:w`, then trim trailing zeros and dot. Normal numbers with an exponent 16 or larger have no decimal separator, we only need to add trailing zeros. When the exponent is non-positive, the result should be 0. $\langle zeros \rangle \langle digits \rangle$, trimmed.

```

13621 \cs_new:Npn \__fp_to_decimal_dispatch:w \s__fp \__fp_chk:w #1#2
13622 {
13623   \if_meaning:w 2 #2 \exp_after:wN - \tex_romannumeral:D -'0 \fi:
13624   \if_case:w #1 \exp_stop_f:
13625     \__fp_case_return:nw { 0 }
13626   \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
13627   \or:
13628     \__fp_case_use:nw
13629     {
13630       \__fp_invalid_operation:nnw
13631       {
13632         \exp_after:wN \exp_after:wN \exp_after:wN 1
13633         \prg_replicate:nn \c__fp_max_exponent_int 0
13634       }
13635       { fp_to_decimal }
13636     }
13637   \or:
13638     \__fp_case_use:nw
13639     {
13640       \__fp_invalid_operation:nnw

```

```

13641         { 0 }
13642         { fp_to_decimal }
13643     }
13644     \fi:
13645     \s__fp \__fp_chk:w #1 #2
13646 }
13647 \cs_new:Npn \__fp_to_decimal_normal:wnnnnn
13648 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
13649 {
13650     \int_compare:nNnTF {#2} > \c_zero
13651     {
13652         \int_compare:nNnTF {#2} < \c_sixteen
13653         {
13654             \__fp_decimate:nNnnnn { \c_sixteen - #2 }
13655             \__fp_to_decimal_large:Nnnw
13656         }
13657         {
13658             \exp_after:wN \exp_after:wN
13659             \exp_after:wN \__fp_to_decimal_huge:wnnnn
13660             \prg_replicate:nn { #2 - \c_sixteen } { 0 } ;
13661         }
13662         {#3} {#4} {#5} {#6}
13663     }
13664     {
13665         \exp_after:wN \__fp_trim_zeros:w
13666         \exp_after:wN 0
13667         \exp_after:wN .
13668         \tex_romannumeral:D -'0 \prg_replicate:nn { - #2 } { 0 }
13669         #3#4#5#6 ;
13670     }
13671 }
13672 \cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
13673 {
13674     \exp_after:wN \__fp_trim_zeros:w \__int_value:w
13675     \if_int_compare:w #2 > \c_zero
13676     #2
13677     \fi:
13678     \exp_stop_f:
13679     #3.#4 ;
13680 }
13681 \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.)

37.4 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

```

13682 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
13683 \cs_generate_variant:Nn \fp_to_tl:N { c }

```

```

13684 \cs_new_nopar:Npn \fp_to_tl:n
13685 {
13686   \exp_after:wN \__fp_to_tl_dispatch:w
13687   \tex_romannumeral:D -'0 \__fp_parse:n
13688 }

```

(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
\__fp_to_tl_normal:nnnnn

```

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.

```

13689 \cs_new:Npn \__fp_to_tl_dispatch:w \s__fp \__fp_chk:w #1#2
13690 {
13691   \if_meaning:w 2 #2 \exp_after:wN - \tex_romannumeral:D -'0 \fi:
13692   \if_case:w #1 \exp_stop_f:
13693     \__fp_case_return:nw { 0 }
13694   \or: \exp_after:wN \__fp_to_tl_normal:nnnnn
13695   \or: \__fp_case_return:nw { \tl_to_str:n {inf} }
13696   \else: \__fp_case_return:nw { \tl_to_str:n {nan} }
13697   \fi:
13698 }
13699 \cs_new:Npn \__fp_to_tl_normal:nnnnn #1
13700 {
13701   \if_int_compare:w #1 > \c_sixteen
13702     \exp_after:wN \__fp_to_scientific_normal:wnnnnn
13703   \else:
13704     \if_int_compare:w #1 < - \c_two
13705       \exp_after:wN \exp_after:wN
13706       \exp_after:wN \__fp_to_scientific_normal:wnnnnn
13707     \else:
13708       \exp_after:wN \exp_after:wN
13709       \exp_after:wN \__fp_to_decimal_normal:wnnnnn
13710     \fi:
13711   \fi:
13712   \s__fp \__fp_chk:w 1 0 {#1}
13713 }

```

(End definition for `__fp_to_tl_dispatch:w` and `__fp_to_tl_normal:nnnnn`.)

37.5 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.

37.6 Convert to dimension or integer

```

\fp_to_dim:N
\fp_to_dim:c
\fp_to_dim:n

```

These three public functions rely on `\fp_to_decimal:n` internally. We make sure to produce pt with category other.


```

13714 \cs_new:Npx \fp_to_dim:N #1
13715 { \exp_not:N \fp_to_decimal:N #1 \tl_to_str:n {pt} }
13716 \cs_generate_variant:Nn \fp_to_dim:N { c }
13717 \cs_new:Npx \fp_to_dim:n #1
13718 { \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
13719 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
13720 \cs_generate_variant:Nn \fp_to_int:N { c }
13721 \cs_new_nopar:Npn \fp_to_int:n
13722 {
13723   \exp_after:wN \__fp_to_int_dispatch:w
13724   \tex_romannumeral:D -'0 \__fp_parse:n
13725 }
(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.

```

13726 \cs_new:Npn \__fp_to_int_dispatch:w #1;
13727 {
13728   \exp_after:wN \__fp_to_decimal_dispatch:w \tex_romannumeral:D -'0
13729   \__fp_round:Nwn \__fp_round_to_nearest:NNN #1; { 0 }
13730 }
(End definition for \__fp_to_int_dispatch:w.)

```

37.7 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.

```

13731 \cs_new:Npn \dim_to_fp:n #1
13732 {
13733   \exp_after:wN \__fp_from_dim_test:N
13734   \__int_value:w \etex_glueexpr:D #1 ;
13735 }
13736 \cs_new:Npn \__fp_from_dim_test:N #1
13737 {
13738   \if_meaning:w 0 #1
13739     \__fp_case_return:nw \c_zero_fp
13740   \else:
13741     \if_meaning:w - #1

```

```

13742         \exp_after:wN \__fp_from_dim:Nw
13743         \exp_after:wN 2
13744         \__int_value:w
13745     \else:
13746         \exp_after:wN \__fp_from_dim:Nw
13747         \exp_after:wN 0
13748         \__int_value:w #1
13749     \fi:
13750 \fi:
13751 }
13752 \cs_new:Npn \__fp_from_dim:Nw #1 #2;
13753 {
13754     \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
13755     #2 000 0000 00 {10}987654321; #1
13756 }
13757 \cs_new:Npn \__fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
13758 { \__fp_from_dim:wnnnnwN #1 {#2#300} {0000} ; }
13759 \cs_new:Npn \__fp_from_dim:wnnnnwN #1; #2#3#4#5#6; #7
13760 {
13761     \__fp_mul_npos_o:Nww #7
13762     \s__fp \__fp_chk:w 1 #7 {#5} #1 ;
13763     \s__fp \__fp_chk:w 1 0 {-4} {1525} {8789} {0625} {0000} ;
13764 }

```

(End definition for `\dim_to_fp:n`. This function is documented on page 181.)

37.8 Use and eval

\fp_use:N Those public functions are simple copies of the decimal conversions.

\fp_use:c 13765 \cs_new_eq:NN \fp_use:N \fp_to_decimal:N
\fp_eval:n 13766 \cs_generate_variant:Nn \fp_use:N { c }
13767 \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 170.)

\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.

```

13768 \cs_new:Npn \fp_abs:n #1
13769 { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }

```

(End definition for `\fp_abs:n`. This function is documented on page 181.)

\fp_max:nn Similar to `\fp_abs:n`, for consistency with `\int_max:nn`, etc.

\fp_min:nn 13770 \cs_new:Npn \fp_max:nn #1#2
13771 { \fp_to_decimal:n { max (__fp_parse:n {#1} , __fp_parse:n {#2}) } }
13772 \cs_new:Npn \fp_min:nn #1#2
13773 { \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 181.)

37.9 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.

```
13774 \cs_new:Npn \__fp_array_to_clist:n #1
13775 {
13776   \tl_if_empty:nF {#1}
13777   {
13778     \__fp_expand:n
13779     {
13780       { \use_ii:nn }
13781       \__fp_array_to_clist_loop:Nw #1 { ? \__prg_break: } ;
13782       \__prg_break_point:
13783     }
13784   }
13785 }
13786 \cs_new:Npx \__fp_array_to_clist_loop:Nw #1#2;
13787 {
13788   \exp_not:N \use_none:n #1
13789   \exp_not:N \exp_after:wN
13790   {
13791     \exp_not:N \exp_after:wN ,
13792     \exp_not:N \exp_after:wN \c_space_tl
13793     \exp_not:N \tex_romannumeral:D -'0
13794     \exp_not:N \__fp_to_tl_dispatch:w #1 #2 ;
13795   }
13796   \exp_not:N \__fp_array_to_clist_loop:Nw
13797 }
```

(End definition for `__fp_array_to_clist:n`. This function is documented on page ??.)

```
13798 </initex | package>
```

38 l3fp-assign implementation

```
13799 <*initex | package>
```

```
13800 <@@=fp>
```

38.1 Assigning values

`\fp_new:N` Floating point variables are initialized to be +0.

```

13801 \cs_new_protected:Npn \fp_new:N #1
13802 { \cs_new_eq:NN #1 \c_zero_fp }
13803 \cs_generate_variant:Nn \fp_new:N {c}

```

(End definition for `\fp_new:N`. This function is documented on page 169.)

`\fp_set:Nn` Simply use `__fp_parse:n` within various x-expanding assignments.

```

\fp_set:cn 13804 \cs_new_protected:Npn \fp_set:Nn #1#2
\fp_gset:Nn 13805 { \tl_set:Nx #1 { \__fp_parse:n {#2} } }
\fp_gset:cn 13806 \cs_new_protected:Npn \fp_gset:Nn #1#2
\fp_const:Nn 13807 { \tl_gset:Nx #1 { \__fp_parse:n {#2} } }
\fp_const:cn 13808 \cs_new_protected:Npn \fp_const:Nn #1#2
13809 { \tl_const:Nx #1 { \__fp_parse:n {#2} } }
13810 \cs_generate_variant:Nn \fp_set:Nn {c}
13811 \cs_generate_variant:Nn \fp_gset:Nn {c}
13812 \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 13813 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
\fp_set_eq:Nc 13814 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\fp_set_eq:cc 13815 \cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }
\fp_gset_eq:NN 13816 \cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }

```

(End definition for `\fp_set_eq:NN` and others. These functions are documented on page ??.)

`\fp_zero:N` Setting a floating point to zero: copy `\c_zero_fp`.

```

\fp_zero:c 13817 \cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\fp_gzero:N 13818 \cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\fp_gzero:c 13819 \cs_generate_variant:Nn \fp_zero:N { c }
13820 \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 13821 \cs_new_protected:Npn \fp_zero_new:N #1
\fp_gzero_new:N 13822 { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\fp_gzero_new:c 13823 \cs_new_protected:Npn \fp_gzero_new:N #1
13824 { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
13825 \cs_generate_variant:Nn \fp_zero_new:N { c }
13826 \cs_generate_variant:Nn \fp_gzero_new:N { c }

```

(End definition for `\fp_zero_new:N` and others. These functions are documented on page ??.)

38.2 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`

```

13827 \cs_new_protected_nopar:Npn \fp_add:Nn { \__fp_add:NNNn \fp_set:Nn + }
13828 \cs_new_protected_nopar:Npn \fp_gadd:Nn { \__fp_add:NNNn \fp_gset:Nn + }
13829 \cs_new_protected_nopar:Npn \fp_sub:Nn { \__fp_add:NNNn \fp_set:Nn - }
13830 \cs_new_protected_nopar:Npn \fp_gsub:Nn { \__fp_add:NNNn \fp_gset:Nn - }
13831 \cs_new_protected:Npn \__fp_add:NNNn #1#2#3#4
13832 { #1 #3 { #3 #2 \__fp_parse:n {#4} } }
13833 \cs_generate_variant:Nn \fp_add:Nn { c }
13834 \cs_generate_variant:Nn \fp_gadd:Nn { c }
13835 \cs_generate_variant:Nn \fp_sub:Nn { c }
13836 \cs_generate_variant:Nn \fp_gsub:Nn { c }

```

(End definition for `\fp_add:Nn` and others. These functions are documented on page ??.)

38.3 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`

```

13837 \cs_new_protected:Npn \fp_show:N #1
13838 {
13839   \fp_if_exist:NTF #1
13840   { \__msg_show_variable:n { > ~ \fp_to_tl:N #1 } }
13841   {
13842     \__msg_kernel_error:nx { kernel } { variable-not-defined }
13843     { \token_to_str:N #1 }
13844   }
13845 }
13846 \cs_new_protected:Npn \fp_show:n #1
13847 { \__msg_show_variable:n { > ~ \fp_to_tl:n {#1} } }
13848 \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 ??.)

38.4 Some useful constants and scratch variables

`\c_one_fp` Some constants.
`\c_e_fp`

```

13849 \fp_const:Nn \c_e_fp { 2.718 2818 2845 9045 }
13850 \fp_const:Nn \c_one_fp { 1 }

```

(End definition for `\c_one_fp` and `\c_e_fp`. These variables are documented on page 174.)

`\c_pi_fp` We do not round π to the closest multiple of 10^{-15} , which would underestimate it by
`\c_one_degree_fp` roughly $2.4 \cdot 10^{-16}$, but instead round it up to the next nearest multiple, which is an
overestimate by roughly $7.7 \cdot 10^{-16}$. This particular choice of rounding has very nice

properties: it is exactly divisible by 4 and by 180 as a 16-digit precision floating point number, hence ensuring that $\sin(180\text{deg}) = \sin(\pi) = 0$ exactly, with no rounding artifact.

```
13851 \fp_const:Nn \c_pi_fp { 3.141 5926 5358 9794 }
13852 \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 174.)

`\l_tmpa_fp` Scratch variables are simply initialized there.

```
\l_tmpb_fp 13853 \fp_new:N \l_tmpa_fp
\g_tmpa_fp 13854 \fp_new:N \l_tmpb_fp
\g_tmpb_fp 13855 \fp_new:N \g_tmpa_fp
13856 \fp_new:N \g_tmpb_fp
```

(End definition for `\l_tmpa_fp` and others. These variables are documented on page 174.)

```
13857 </initex | package>
```

39 l3fp-old implementation

```
13858 <*initex | package>
```

```
13859 <@@=fp>
```

39.1 Compatibility

`\c_undefined_fp` The old floating point number `\c_undefined_fp` is now implemented as a `nan`.

```
13860 \fp_const:Nn \c_undefined_fp { nan }
```

(End definition for `\c_undefined_fp`. This variable is documented on page ??.)

`\fp_if_undefined_p:N` An old floating point is undefined if it is `inf` or `nan`, *i.e.*, if its type is 2 or 3.

```
\fp_if_undefined:NTF 13861 \prg_new_conditional:Npnn \fp_if_undefined:N #1 { p , T , F , TF }
13862 { \exp_after:wN \_fp_if_undefined:w #1 }
13863 \cs_new:Npn \_fp_if_undefined:w \s_fp \_fp_chk:w #1#2;
13864 {
13865   \if_int_compare:w #1 > \c_one
13866     \prg_return_true: \else: \prg_return_false: \fi:
13867 }
```

(End definition for `\fp_if_undefined:N`. These functions are documented on page ??.)

`\fp_if_zero_p:N` An old floating point is zero if it is ± 0 , *i.e.*, its type is 0.

```
\fp_if_zero:NTF 13868 \prg_new_conditional:Npnn \fp_if_zero:N #1 { p , T , F , TF }
13869 { \exp_after:wN \_fp_if_zero:w #1 }
13870 \cs_new:Npn \_fp_if_zero:w \s_fp \_fp_chk:w #1#2;
13871 { \if_meaning:w #1 0 \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\fp_if_zero:N`. These functions are documented on page ??.)

`\fp_abs:N` Simply expand the floating point variable to feed it to `__fp_abs_o:w` or `__fp_-_o:w`,
`\fp_abs:c` expanded within an expanding token list assignment. The `\prg_do_nothing:` is not
`\fp_gabs:N` necessary, but it reminds us more clearly that `__fp_abs_o:w` and `__fp_-_o:w` expand
`\fp_gabs:c` after their result.
`\fp_neg:N` 13872 `\cs_new_protected_nopar:Npn \fp_abs:N { __fp_abs:NNN \tl_set:Nx __fp_abs_o:w }`
`\fp_neg:c` 13873 `\cs_new_protected_nopar:Npn \fp_gabs:N { __fp_abs:NNN \tl_gset:Nx __fp_abs_o:w }`
`\fp_gneg:N` 13874 `\cs_new_protected_nopar:Npx \fp_neg:N`
`\fp_gneg:c` 13875 `{`
`__fp_abs:NNN` 13876 `\exp_not:N __fp_abs:NNN`
13877 `\exp_not:N \tl_set:Nx`
13878 `\exp_not:c { __fp_-_o:w }`
13879 `}`
13880 `\cs_new_protected_nopar:Npx \fp_gneg:N`
13881 `{`
13882 `\exp_not:N __fp_abs:NNN`
13883 `\exp_not:N \tl_gset:Nx`
13884 `\exp_not:c { __fp_-_o:w }`
13885 `}`
13886 `\cs_new_protected:Npn __fp_abs:NNN #1#2#3`
13887 `{ #1 #3 { \exp_after:wN #2 #3 \prg_do_nothing: } }`
13888 `\cs_generate_variant:Nn \fp_abs:N { c }`
13889 `\cs_generate_variant:Nn \fp_gabs:N { c }`
13890 `\cs_generate_variant:Nn \fp_neg:N { c }`
13891 `\cs_generate_variant:Nn \fp_gneg:N { c }`
(End definition for `\fp_abs:N` and others. These functions are documented on page ??.)

`\fp_mul:Nn` See `\fp_add:Nn` for details.
`\fp_mul:cn` 13892 `\cs_new_protected_nopar:Npn \fp_mul:Nn { __fp_mul:NNNn \fp_set:Nn * }`
`\fp_gmul:Nn` 13893 `\cs_new_protected_nopar:Npn \fp_gmul:Nn { __fp_mul:NNNn \fp_gset:Nn * }`
`\fp_gmul:cn` 13894 `\cs_new_protected_nopar:Npn \fp_div:Nn { __fp_mul:NNNn \fp_set:Nn / }`
`\fp_div:Nn` 13895 `\cs_new_protected_nopar:Npn \fp_gdiv:Nn { __fp_mul:NNNn \fp_gset:Nn / }`
`\fp_div:cn` 13896 `\cs_new_protected_nopar:Npn \fp_pow:Nn { __fp_mul:NNNn \fp_set:Nn ^ }`
`\fp_gdiv:Nn` 13897 `\cs_new_protected_nopar:Npn \fp_gpow:Nn { __fp_mul:NNNn \fp_gset:Nn ^ }`
`\fp_gdiv:cn` 13898 `\cs_new_protected:Npn __fp_mul:NNNn #1#2#3#4`
13899 `{ #1 #3 { #3 #2 __fp_parse:n {#4} } }`
`\fp_pow:Nn` 13900 `\cs_generate_variant:Nn \fp_mul:Nn { c }`
`\fp_pow:cn` 13901 `\cs_generate_variant:Nn \fp_gmul:Nn { c }`
`\fp_gpow:Nn` 13902 `\cs_generate_variant:Nn \fp_div:Nn { c }`
`\fp_gpow:cn` 13903 `\cs_generate_variant:Nn \fp_gdiv:Nn { c }`
`__fp_mul:NNNn` 13904 `\cs_generate_variant:Nn \fp_pow:Nn { c }`
13905 `\cs_generate_variant:Nn \fp_gpow:Nn { c }`
(End definition for `\fp_mul:Nn` and others. These functions are documented on page ??.)

`\fp_exp:Nn` Here, an added twist is that each value computed by these expensive unary operations is
`\fp_exp:cn` stored as a constant floating point number.

`\fp_gexp:Nn` 13906 `\cs_set_protected:Npn __fp_tmp:w #1#2#3#4#5`
`\fp_gexp:cn` 13907 `{`
`\fp_ln:Nn` 13908 `\cs_new_protected_nopar:Npn #1 { #5 {#4} \tl_set_eq:NN #3 }`
`\fp_ln:cn` 13909 `\cs_new_protected_nopar:Npn #2 { #5 {#4} \tl_gset_eq:NN #3 }`

`\fp_gln:Nn`
`\fp_gln:cn`
`\fp_sin:Nn`
`\fp_sin:cn`
`\fp_gsin:Nn`
`\fp_gsin:cn`
`\fp_cos:Nn`
`\fp_cos:cn`
`\fp_gcos:Nn`
`\fp_gcos:cn`

```

13910 \cs_generate_variant:Nn #1 { c }
13911 \cs_generate_variant:Nn #2 { c }
13912 }
13913 \__fp_tmp:w \fp_exp:Nn \fp_gexp:Nn \__fp_exp_o:w {exp} \__fp_assign_to:nNNNn
13914 \__fp_tmp:w \fp_ln:Nn \fp_gln:Nn \__fp_ln_o:w {ln} \__fp_assign_to:nNNNn
13915 \__fp_tmp:w \fp_sin:Nn \fp_gsin:Nn \__fp_sin_o:w {sin} \__fp_assign_to:nNNNn
13916 \__fp_tmp:w \fp_cos:Nn \fp_gcos:Nn \__fp_cos_o:w {cos} \__fp_assign_to:nNNNn
13917 \__fp_tmp:w \fp_tan:Nn \fp_gtan:Nn \__fp_tan_o:w {tan} \__fp_assign_to:nNNNn
13918 \cs_new_protected:Npn \__fp_assign_to:nNNNn #1#2#3#4#5
13919 {
13920 \exp_after:wN \__fp_assign_to_i:wNNNn
13921 \tex_romannumeral:D -'0 \__fp_parse:n {#5} {#1} #2#3#4
13922 }
13923 \cs_new_protected:Npn \__fp_assign_to_i:wNNNn \s__fp \__fp_chk:w #1#2#3; #4
13924 {
13925 \exp_args:Nc \__fp_assign_to_ii:NnNNN
13926 { c__fp_ #4 [ #1 # 2 \if_meaning:w 1 #1 #3 \fi: ] _fp }
13927 { #1#2#3 }
13928 }
13929 \cs_new_protected:Npn \__fp_assign_to_ii:NnNNN #1#2#3#4#5
13930 {
13931 \cs_if_exist:NF #1
13932 { \tl_const:Nx #1 { #4 \s__fp \__fp_chk:w #2; } }
13933 #3 #5 #1
13934 }

```

(End definition for `\fp_exp:Nn` and others. These functions are documented on page ??.)

`\fp_compare:NNNTF` Comparisons used to be easier between floating points stored in variables. No more.

```

13935 \cs_new_protected_nopar:Npn \fp_compare:NNNTF { \fp_compare:nNNTF }
13936 \cs_new_protected_nopar:Npn \fp_compare:NNNT { \fp_compare:nNnT }
13937 \cs_new_protected_nopar:Npn \fp_compare:NNNF { \fp_compare:nNnF }

```

(End definition for `\fp_compare:NNNTF`. This function is documented on page ??.)

`\fp_round_places:Nn` Rounding to a given number of places is easy, since it is provided by the `l3fp-round`
`\fp_ground_places:Nn` module.
`__fp_round_places:NNn`

```

13938 \cs_new_protected_nopar:Npn \fp_round_places:Nn
13939 { \__fp_round_places:NNn \tl_set:Nx }
13940 \cs_new_protected_nopar:Npn \fp_ground_places:Nn
13941 { \__fp_round_places:NNn \tl_gset:Nx }
13942 \cs_new_protected:Npn \__fp_round_places:NNn #1#2#3
13943 {
13944 #1 #2
13945 {
13946 \exp_after:wN \exp_after:wN
13947 \exp_after:wN \__fp_round:Nwn
13948 \exp_after:wN \exp_after:wN
13949 \exp_after:wN \__fp_round_to_nearest:NNN
13950 \exp_after:wN #2
13951 \exp_after:wN { \int_use:N \__int_eval:w #3 }

```



```

13952     }
13953   }
13954   \cs_generate_variant:Nn \fp_round_places:Nn { c }
13955   \cs_generate_variant:Nn \fp_ground_places:Nn { c }
(End definition for \fp_round_places:Nn and \fp_ground_places:Nn. These functions are documented
on page ??.)

```

`\fp_round_figures:Nn` Rounding to a given number of figures is the same as rounding to a number of places, after shifting by the exponent of the argument.

```

13956   \cs_new_protected:Npn \fp_round_figures:Nn #1#2
13957   {
13958     \__fp_round_places:NNn \tl_set:Nx #1
13959     { #2 - \exp_after:wN \__fp_exponent:w #1 }
13960   }
13961   \cs_new_protected:Npn \fp_ground_figures:Nn #1#2
13962   {
13963     \__fp_round_places:NNn \tl_gset:Nx #1
13964     { #2 - \exp_after:wN \__fp_exponent:w #1 }
13965   }
13966   \cs_generate_variant:Nn \fp_round_figures:Nn { c }
13967   \cs_generate_variant:Nn \fp_ground_figures:Nn { c }
(End definition for \fp_round_figures:Nn and \fp_ground_figures:Nn. These functions are docu-
mented on page ??.)
13968   </initex | package>

```

40 l3luatex implementation

```

13969   <*initex | package>

```

Announce and ensure that the required packages are loaded.

```

13970   <*package>
13971   \ProvidesExplPackage
13972     {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
13973   \__expl_package_check:
13974   </package>

```

`\lua_now_x:n` When LuaTeX is in use, this is all a question of primitives with new names. On the other hand, for pdfTeX and XeTeX the argument should be removed from the input stream before issuing an error. This is expandable, using `__msg_kernel_expandable_error:nnn` as done for V-type expansion in l3expan.

```

\lua_now_x:n
\lua_now_x:x
\lua_now:n
\lua_now:x
\lua_shipout_x:n
\lua_shipout_x:x
\lua_shipout:n
\lua_shipout:x
13975   \luatex_if_engine:TF
13976   {
13977     \cs_new_eq:NN \lua_now_x:n \luatex_directlua:D
13978     \cs_new_eq:NN \lua_shipout_x:n \luatex_latelua:D
13979   }
13980   {
13981     \cs_new:Npn \lua_now_x:n #1
13982     {
13983       \__msg_kernel_expandable_error:nnn

```

```

13984         { kernel } { bad-engine } { \lua_now_x:n }
13985     }
13986     \cs_new_protected:Npn \lua_shipout_x:n #1
13987     {
13988         \__msg_kernel_expandable_error:nnn
13989         { kernel } { bad-engine } { \lua_shipout_x:n }
13990     }
13991 }
13992 \cs_generate_variant:Nn \lua_now_x:n { x }
13993 \cs_new:Npn \lua_now:n #1
13994 { \lua_now_x:n { \exp_not:n {#1} } }
13995 \cs_generate_variant:Nn \lua_now:n { x }
13996 \cs_generate_variant:Nn \lua_shipout_x:n { x }
13997 \cs_new_protected:Npn \lua_shipout:n #1
13998 { \lua_shipout_x:n { \exp_not:n {#1} } }
13999 \cs_generate_variant:Nn \lua_shipout:n { x }

```

(End definition for \lua_now_x:n and \lua_now_x:x. These functions are documented on page ??.)

40.1 Category code tables

14000 <@@=cctab>

\g__cctab_allocate_int To allocate category code tables, both the read-only and stack tables need to be followed.
 \g__cctab_stack_int There is also a sequence stack for the dynamic tables themselves.
 \g__cctab_stack_seq

```

14001 \int_new:N \g__cctab_allocate_int
14002 \int_set:Nn \g__cctab_allocate_int { \c_minus_one }
14003 \int_new:N \g__cctab_stack_int
14004 \seq_new:N \g__cctab_stack_seq

```

(End definition for \g__cctab_allocate_int. This function is documented on page ??.)

\cctab_new:N Creating a new category code table is done slightly differently from other registers. Low-numbered tables are more efficiently-stored than high-numbered ones. There is also a need to have a stack of flexible tables as well as the set of read-only ones. To satisfy both of these requirements, odd numbered tables are used for read-only tables, and even ones for the stack. Here, therefore, the odd numbers are allocated.

```

14005 \cs_new_protected:Npn \cctab_new:N #1
14006 {
14007     \__chk_if_free_cs:N #1
14008     \int_gadd:Nn \g__cctab_allocate_int { \c_two }
14009     \int_compare:nNnTF
14010     \g__cctab_allocate_int < { \c_max_register_int + \c_one }
14011     {
14012         \tex_global:D \tex_chardef:D #1 \g__cctab_allocate_int
14013         \luatex_initcatcodetable:D #1
14014     }
14015     { \__msg_kernel_fatal:nnx { kernel } { out-of-registers } { cctab } }
14016 }
14017 \luatex_if_engine:F
14018 {

```

```

14019 \cs_set_protected:Npn \cctab_new:N #1
14020 {
14021   \__msg_kernel_error:nxx { kernel } { bad-engine }
14022   { \exp_not:N \cctab_new:N }
14023 }
14024 }
14025 <*package>
14026 \luatex_if_engine:T
14027 {
14028   \cs_set_protected:Npn \cctab_new:N #1
14029   {
14030     \__chk_if_free_cs:N #1
14031     \newcatcodetable #1
14032     \luatex_initcatcodetable:D #1
14033   }
14034 }
14035 </package>

```

(End definition for \cctab_new:N. This function is documented on page 186.)

\cctab_begin:N The aim here is to ensure that the saved tables are read-only. This is done by using a
\cctab_end: stack of tables which are not read only, and actually having them as “in use” copies.

\l__cctab_internal_tl

```

14036 \cs_new_protected:Npn \cctab_begin:N #1
14037 {
14038   \seq_gpush:Nx \g__cctab_stack_seq { \tex_the:D \luatex_catcodetable:D }
14039   \luatex_catcodetable:D #1
14040   \int_gadd:Nn \g__cctab_stack_int { \c_two }
14041   \int_compare:nNnT \g__cctab_stack_int > \c_max_register_int
14042   { \__msg_kernel_fatal:nn { kernel } { cctab-stack-full } }
14043   \luatex_savecatcodetable:D \g__cctab_stack_int
14044   \luatex_catcodetable:D \g__cctab_stack_int
14045 }
14046 \cs_new_protected_nopar:Npn \cctab_end:
14047 {
14048   \int_gsub:Nn \g__cctab_stack_int { \c_two }
14049   \seq_if_empty:NTF \g__cctab_stack_seq
14050   { \tl_set:Nn \l__cctab_internal_tl { 0 } }
14051   { \seq_gpop:NN \g__cctab_stack_seq \l__cctab_internal_tl }
14052   \luatex_catcodetable:D \l__cctab_internal_tl \scan_stop:
14053 }
14054 \luatex_if_engine:F
14055 {
14056   \cs_set_protected:Npn \cctab_begin:N #1
14057   {
14058     \__msg_kernel_error:nxxx { kernel } { bad-engine }
14059     { \exp_not:N \cctab_begin:N } {#1}
14060   }
14061   \cs_set_protected_nopar:Npn \cctab_end:
14062   {
14063     \__msg_kernel_error:nxx { kernel } { bad-engine }

```

```

14064         { \exp_not:N \cctab_end: }
14065     }
14066 }
14067 <*package>
14068 \luatex_if_engine:T
14069 {
14070     \cs_set_protected:Npn \cctab_begin:N #1 { \BeginCatcodeRegime #1 }
14071     \cs_set_protected_nopar:Npn \cctab_end: { \EndCatcodeRegime }
14072 }
14073 </package>
14074 \tl_new:N \l__cctab_internal_tl

```

(End definition for \cctab_begin:N. This function is documented on page ??.)

\cctab_gset:Nn Category code tables are always global, so only one version is needed. The set up here is simple, and means that at the point of use there is no need to worry about escaping category codes.

```

14075 \cs_new_protected:Npn \cctab_gset:Nn #1#2
14076 {
14077     \group_begin:
14078     #2
14079     \luatex_savecatcodetable:D #1
14080     \group_end:
14081 }
14082 \luatex_if_engine:F
14083 {
14084     \cs_set_protected:Npn \cctab_gset:Nn #1#2
14085     {
14086         \_msg_kernel_error:nxxx { kernel } { bad-engine }
14087         { \exp_not:N \cctab_gset:Nn } { #1 {#2} }
14088     }
14089 }

```

(End definition for \cctab_gset:Nn. This function is documented on page 186.)

\c_code_cctab Creating category code tables is easy using the function above. The **other** and **string**
\c_document_cctab ones are done by completely ignoring the existing codes as this makes life a lot less
\c_initex_cctab complex. The table for expl3 category codes is always needed, whereas when in package
\c_other_cctab mode the rest can be copied from the existing L^AT_EX 2_ε package luatex.
\c_str_cctab

```

14090 \luatex_if_engine:T
14091 {
14092     \cctab_new:N \c_code_cctab
14093     \cctab_gset:Nn \c_code_cctab { }
14094 }
14095 <*package>
14096 \luatex_if_engine:T
14097 {
14098     \cs_new_eq:NN \c_document_cctab \CatcodeTableLaTeX
14099     \cs_new_eq:NN \c_initex_cctab \CatcodeTableIniTeX
14100     \cs_new_eq:NN \c_other_cctab \CatcodeTableOther
14101     \cs_new_eq:NN \c_str_cctab \CatcodeTableString

```

```

14102 }
14103 </package>
14104 <*initex>
14105 \luatex_if_engine:T
14106 {
14107   \cctab_new:N \c_document_cctab
14108   \cctab_new:N \c_other_cctab
14109   \cctab_new:N \c_str_cctab
14110   \cctab_gset:Nn \c_document_cctab
14111   {
14112     \char_set_catcode_space:n { 9 }
14113     \char_set_catcode_space:n { 32 }
14114     \char_set_catcode_other:n { 58 }
14115     \char_set_catcode_math_subscript:n { 95 }
14116     \char_set_catcode_active:n { 126 }
14117   }
14118   \cctab_gset:Nn \c_other_cctab
14119   {
14120     \int_step_inline:nnnn { 0 } { 1 } { 127 }
14121     { \char_set_catcode_other:n {#1} }
14122   }
14123   \cctab_gset:Nn \c_str_cctab
14124   {
14125     \int_step_inline:nnnn { 0 } { 1 } { 127 }
14126     { \char_set_catcode_other:n {#1} }
14127     \char_set_catcode_space:n { 32 }
14128   }
14129 }
14130 </initex>

```

(End definition for `\c_code_cctab`. This function is documented on page 187.)

40.2 Messages

```

14131 \__msg_kernel_new:nnnn { kernel } { bad-engine }
14132 { LuaTeX-engine-not-in-use!~Ignoring-#1. }
14133 {
14134   The~feature~you~are~using~is~only~available~
14135   with~the~LuaTeX~engine.~LaTeX3-ignored~‘#1#2’.
14136 }
14137 \__msg_kernel_new:nnnn { kernel } { cctab-stack-full }
14138 { The~category~code~table~stack~is~exhausted. }
14139 {
14140   LaTeX~has~been~asked~to~switch~to~a~new~category~code~table,~
14141   but~there~is~no~more~space~to~do~this!
14142 }

```

40.3 Deprecated functions

Deprecated 2011-12-21, for removal by 2012-03-31.

`\c_string_cctab`

```
14143 <*deprecated>
14144 \cs_new_eq:NN \c_string_cctab \c_str_cctab
14145 </deprecated>
(End definition for \c_string_cctab. This variable is documented on page ??.)
14146 </initex | package>
```

41 l3candidates Implementation

```
14147 <*initex | package>
14148 <*package>
14149 \ProvidesExplPackage
14150   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
14151   \__expl_package_check:
14152 </package>
```

41.1 Additions to l3box

```
14153 <@@=box>
```

41.2 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.

```
14154 \fp_new:N \l__box_angle_fp
(End definition for \l__box_angle_fp. This variable is documented on page 190.)
```

`\l__box_cos_fp` These are used to hold the calculated sine and cosine values while carrying out a rotation.

`\l__box_sin_fp`

```
14155 \fp_new:N \l__box_cos_fp
14156 \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 190.)
```

`\l__box_top_dim` These are the positions of the four edges of a box before manipulation.

`\l__box_bottom_dim`

`\l__box_left_dim`

`\l__box_right_dim`

```
14157 \dim_new:N \l__box_top_dim
14158 \dim_new:N \l__box_bottom_dim
14159 \dim_new:N \l__box_left_dim
14160 \dim_new:N \l__box_right_dim
(End definition for \l__box_top_dim and others. These variables are documented on page ??.)
```

`\l__box_top_new_dim` These are the positions of the four edges of a box after manipulation.

`\l__box_bottom_new_dim`

`\l__box_left_new_dim`

`\l__box_right_new_dim`

```
14161 \dim_new:N \l__box_top_new_dim
14162 \dim_new:N \l__box_bottom_new_dim
14163 \dim_new:N \l__box_left_new_dim
14164 \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.

```
14165 \box_new:N \l__box_internal_box
```

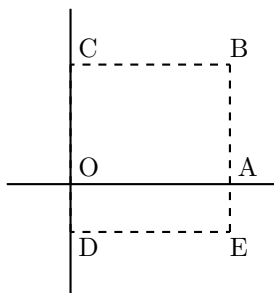


Figure 1: Co-ordinates of a box prior to rotation.

(End definition for `\l__box_internal_box`. This variable is documented on page 190.)

```

\box_rotate:Nn Rotation of a box starts with working out the relevant sine and cosine. The actual
  \__box_rotate:N rotation is in an auxiliary to keep the flow slightly clearer
  \__box_rotate_x:nnN 14166 \cs_new_protected:Npn \box_rotate:Nn #1#2
  \__box_rotate_y:nnN 14167 {
    \__box_rotate_quadrant_one: 14168 \hbox_set:Nn #1
    \__box_rotate_quadrant_two: 14169 {
      \__box_rotate_quadrant_three: 14170 \group_begin:
      \__box_rotate_quadrant_four: 14171 \fp_set:Nn \l__box_angle_fp {#2}
      14172 \fp_set:Nn \l__box_sin_fp { sin ( \l__box_angle_fp * deg ) }
      14173 \fp_set:Nn \l__box_cos_fp { cos ( \l__box_angle_fp * deg ) }
      14174 \__box_rotate:N #1
      14175 \group_end:
      14176 }
      14177 }

```

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.

```

14178 \cs_new_protected:Npn \__box_rotate:N #1
14179 {
14180 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
14181 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
14182 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
14183 \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.)

```

14184     \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
14185     {
14186         \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
14187         { \__box_rotate_quadrant_one: }
14188         { \__box_rotate_quadrant_two: }
14189     }
14190     {
14191         \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
14192         { \__box_rotate_quadrant_three: }
14193         { \__box_rotate_quadrant_four: }
14194     }

```

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.

```

14195     \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
14196     \hbox_set:Nn \l__box_internal_box
14197     {
14198         \tex_kern:D -\l__box_left_new_dim
14199         \hbox:n
14200         {
14201             \__driver_box_rotate_begin:
14202             \box_use:N \l__box_internal_box
14203             \__driver_box_rotate_end:
14204         }
14205     }

```

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.

```

14206     \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
14207     \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
14208     \box_set_wd:Nn \l__box_internal_box
14209     { \l__box_right_new_dim - \l__box_left_new_dim }
14210     \box_use:N \l__box_internal_box
14211 }

```

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.

```

14212 \cs_new_protected:Npn \__box_rotate_x:nnN #1#2#3
14213 {
14214   \dim_set:Nn #3
14215   {
14216     \fp_to_dim:n
14217     {
14218       \l__box_cos_fp * \dim_to_fp:n {#1}
14219       - ( \l__box_sin_fp * \dim_to_fp:n {#2} )
14220     }
14221   }
14222 }
14223 \cs_new_protected:Npn \__box_rotate_y:nnN #1#2#3
14224 {
14225   \dim_set:Nn #3
14226   {
14227     \fp_to_dim:n
14228     {
14229       \l__box_sin_fp * \dim_to_fp:n {#1}
14230       + \l__box_cos_fp * \dim_to_fp:n {#2}
14231     }
14232   }
14233 }

```

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.

```

14234 \cs_new_protected:Npn \__box_rotate_quadrant_one:
14235 {
14236   \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
14237   \l__box_top_new_dim
14238   \__box_rotate_y:nnN \l__box_left_dim \l__box_bottom_dim
14239   \l__box_bottom_new_dim
14240   \__box_rotate_x:nnN \l__box_left_dim \l__box_top_dim
14241   \l__box_left_new_dim
14242   \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
14243   \l__box_right_new_dim
14244 }
14245 \cs_new_protected:Npn \__box_rotate_quadrant_two:
14246 {
14247   \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
14248   \l__box_top_new_dim
14249   \__box_rotate_y:nnN \l__box_left_dim \l__box_top_dim
14250   \l__box_bottom_new_dim

```

```

14251 \__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
14252 \l__box_left_new_dim
14253 \__box_rotate_x:nnN \l__box_left_dim \l__box_bottom_dim
14254 \l__box_right_new_dim
14255 }
14256 \cs_new_protected:Npn \__box_rotate_quadrant_three:
14257 {
14258 \__box_rotate_y:nnN \l__box_left_dim \l__box_bottom_dim
14259 \l__box_top_new_dim
14260 \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
14261 \l__box_bottom_new_dim
14262 \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
14263 \l__box_left_new_dim
14264 \__box_rotate_x:nnN \l__box_left_dim \l__box_top_dim
14265 \l__box_right_new_dim
14266 }
14267 \cs_new_protected:Npn \__box_rotate_quadrant_four:
14268 {
14269 \__box_rotate_y:nnN \l__box_left_dim \l__box_top_dim
14270 \l__box_top_new_dim
14271 \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
14272 \l__box_bottom_new_dim
14273 \__box_rotate_x:nnN \l__box_left_dim \l__box_bottom_dim
14274 \l__box_left_new_dim
14275 \__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
14276 \l__box_right_new_dim
14277 }

```

(End definition for `\box_rotate:Nn`. This function is documented on page 189.)

`\l__box_scale_x_fp`
`\l__box_scale_y_fp`

Scaling is potentially-different in the two axes.

```

14278 \fp_new:N \l__box_scale_x_fp
14279 \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 190.)

`\box_resize:Nnn`
`\box_resize:cnn`
`__box_resize:Nnn`

Resizing a box starts by working out the various dimensions of the existing box.

```

14280 \cs_new_protected:Npn \box_resize:Nnn #1#2#3
14281 {
14282 \hbox_set:Nn #1
14283 {
14284 \group_begin:
14285 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
14286 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
14287 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
14288 \dim_zero:N \l__box_left_dim

```

The x -scaling and resulting box size is easy enough to work out: the dimension is that given as #2, and the scale is simply the new width divided by the old one.

```

14289 \fp_set:Nn \l__box_scale_x_fp
14290 { \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.

```

14291         \fp_set:Nn \l__box_scale_y_fp
14292         {
14293             \dim_to_fp:n {#3} /
14294             ( \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim } )
14295         }

```

Hand off to the auxiliary which does the work.

```

14296         \__box_resize:Nnn #1 {#2} {#3}
14297         \group_end:
14298     }
14299 }
14300 \cs_generate_variant:Nn \box_resize:Nnn { c }

```

With at least one real scaling to do, the next phase is to find the new edge co-ordinates. In the x direction this is relatively easy: just scale the right edge. This is done using the absolute value of the scale so that the new edge is in the correct place. In the y direction, both dimensions have to be scaled, and this again needs the absolute scale value. Once that is all done, the common resize/rescale code can be employed.

```

14301 \cs_new_protected:Npn \__box_resize:Nnn #1#2#3
14302 {
14303     \dim_set:Nn \l__box_right_new_dim { \dim_abs:n {#2} }
14304     \dim_set:Nn \l__box_bottom_new_dim
14305     { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
14306     \dim_set:Nn \l__box_top_new_dim
14307     { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
14308     \__box_resize_common:N #1
14309 }

```

(End definition for \box_resize:Nnn and \box_resize:cnn. These functions are documented on page ??.)

```

\box_resize_to_ht_plus_dp:Nn
\box_resize_to_ht_plus_dp:cn
\box_resize_to_wd:Nn
\box_resize_to_wd:cn

```

Scaling to a total height or to a width is a simplified version of the main resizing operation, with the scale simply copied between the two parts. The internal auxiliary is called using the scaling value twice, as the sign for both parts is needed (as this allows the same internal code to be used as for the general case).

```

14310 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2
14311 {
14312     \hbox_set:Nn #1
14313     {
14314         \group_begin:
14315         \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
14316         \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
14317         \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
14318         \dim_zero:N \l__box_left_dim
14319         \fp_set:Nn \l__box_scale_y_fp
14320         {
14321             \dim_to_fp:n {#2} /
14322             ( \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim } )
14323         }

```

```

14324         \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
14325         \__box_resize:Nnn #1 {#2} {#2}
14326     \group_end:
14327 }
14328 }
14329 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
14330 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
14331 {
14332     \hbox_set:Nn #1
14333     {
14334         \group_begin:
14335         \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
14336         \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
14337         \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
14338         \dim_zero:N \l__box_left_dim
14339         \fp_set:Nn \l__box_scale_x_fp
14340         { \dim_to_fp:n {#2} / ( \dim_to_fp:n \l__box_right_dim ) }
14341         \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
14342         \__box_resize:Nnn #1 {#2} {#2}
14343     \group_end:
14344 }
14345 }
14346 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }

```

(End definition for `\box_resize_to_ht_plus_dp:Nn` and `\box_resize_to_ht_plus_dp:cn`. These functions are documented on page ??.)

`\box_scale:Nnn`
`\box_scale:cn`

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.

```

14347 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
14348 {
14349     \hbox_set:Nn #1
14350     {
14351         \group_begin:
14352         \fp_set:Nn \l__box_scale_x_fp {#2}
14353         \fp_set:Nn \l__box_scale_y_fp {#3}
14354         \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
14355         \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
14356         \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
14357         \dim_zero:N \l__box_left_dim
14358         \dim_set:Nn \l__box_top_new_dim
14359         { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
14360         \dim_set:Nn \l__box_bottom_new_dim
14361         { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
14362         \dim_set:Nn \l__box_right_new_dim
14363         { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
14364         \__box_resize_common:N #1

```

```
14365         \group_end:
```

```
14366     }
```

```
14367 }
```

```
14368 \cs_generate_variant:Nn \box_scale:Nnn { c }
```

(End definition for `\box_scale:Nnn` and `\box_scale:cnm`. 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.

```
14369 \cs_new_protected:Npn \__box_resize_common:N #1
```

```
14370 {
```

```
14371     \hbox_set:Nn \l__box_internal_box
```

```
14372     {
```

```
14373         \__driver_box_scale_begin:
```

```
14374         \hbox_overlap_right:n { \box_use:N #1 }
```

```
14375         \__driver_box_scale_end:
```

```
14376     }
```

The new height and depth can be applied directly.

```
14377     \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
```

```
14378     \box_set_dp:Nn \l__box_internal_box { \l__box_bottom_new_dim }
```

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling factors resizing the box is all that is needed. However, for case of a negative scaling the material must be shifted such that the reference point ends up in the right place.

```
14379     \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
```

```
14380     {
```

```
14381         \hbox_to_wd:nn { \l__box_right_new_dim }
```

```
14382         {
```

```
14383             \tex_kern:D \l__box_right_new_dim
```

```
14384             \box_use:N \l__box_internal_box
```

```
14385             \tex_hss:D
```

```
14386         }
```

```
14387     }
```

```
14388     {
```

```
14389         \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
```

```
14390         \hbox:n
```

```
14391         {
```

```
14392             \tex_kern:D \c_zero_dim
```

```
14393             \box_use:N \l__box_internal_box
```

```
14394             \tex_hss:D
```

```
14395         }
```

```
14396     }
```

```
14397 }
```

(End definition for `__box_resize_common:N`.)

41.3 Viewing part of a box

\box_clip:N A wrapper around the driver-dependent code.

```
\box_clip:c 14398 \cs_new_protected:Npn \box_clip:N #1
14399 { \hbox_set:Nn #1 { \_driver_box_use_clip:N #1 } }
14400 \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 14401 \cs_new_protected:Npn \box_trim:Nnnnn #1#2#3#4#5
14402 {
14403   \hbox_set:Nn \l__box_internal_box
14404   {
14405     \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
14406     \box_use:N #1
14407     \tex_kern:D -\__dim_eval:w #4 \__dim_eval_end:
14408   }
```

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.

```
14409   \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
14410   {
14411     \hbox_set:Nn \l__box_internal_box
14412     {
14413       \box_move_down:nn \c_zero_dim
14414       { \box_use:N \l__box_internal_box }
14415     }
14416     \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
14417   }
14418   {
14419     \hbox_set:Nn \l__box_internal_box
14420     {
14421       \box_move_down:nn { #3 - \box_dp:N #1 }
14422       { \box_use:N \l__box_internal_box }
14423     }
14424     \box_set_dp:Nn \l__box_internal_box \c_zero_dim
14425   }
```

Same thing, this time from the top of the box.

```
14426   \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
14427   {
14428     \hbox_set:Nn \l__box_internal_box
14429     { \box_move_up:nn \c_zero_dim { \box_use:N \l__box_internal_box } }
14430     \box_set_ht:Nn \l__box_internal_box
14431     { \box_ht:N \l__box_internal_box - (#5) }
```

```

14432     }
14433     {
14434         \hbox_set:Nn \l__box_internal_box
14435         {
14436             \box_move_up:nn { #5 - \box_ht:N \l__box_internal_box }
14437             { \box_use:N \l__box_internal_box }
14438         }
14439         \box_set_ht:Nn \l__box_internal_box \c_zero_dim
14440     }
14441     \box_set_eq:NN #1 \l__box_internal_box
14442 }
14443 \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`
`\box_viewport:cnnnn`

The same general logic as for the trim operation, but with absolute dimensions. As a result, there are some things to watch out for in the vertical direction.

```

14444 \cs_new_protected:Npn \box_viewport:Nnnnn #1#2#3#4#5
14445 {
14446     \hbox_set:Nn \l__box_internal_box
14447     {
14448         \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
14449         \box_use:N #1
14450         \tex_kern:D \__dim_eval:w #4 - \box_wd:N #1 \__dim_eval_end:
14451     }
14452     \dim_compare:nNnTF {#3} < \c_zero_dim
14453     {
14454         \hbox_set:Nn \l__box_internal_box
14455         {
14456             \box_move_down:nn \c_zero_dim
14457             { \box_use:N \l__box_internal_box }
14458         }
14459         \box_set_dp:Nn \l__box_internal_box { -\dim_eval:n {#3} }
14460     }
14461     {
14462         \hbox_set:Nn \l__box_internal_box
14463         { \box_move_down:nn {#3} { \box_use:N \l__box_internal_box } }
14464         \box_set_dp:Nn \l__box_internal_box \c_zero_dim
14465     }
14466     \dim_compare:nNnTF {#5} > \c_zero_dim
14467     {
14468         \hbox_set:Nn \l__box_internal_box
14469         { \box_move_up:nn \c_zero_dim { \box_use:N \l__box_internal_box } }
14470         \box_set_ht:Nn \l__box_internal_box
14471         {
14472             #5
14473             \dim_compare:nNnT {#3} > \c_zero_dim
14474             { - (#3) }
14475         }
14476     }

```

```

14477 {
14478   \hbox_set:Nn \l__box_internal_box
14479   {
14480     \box_move_up:nn { -\dim_eval:n {#5} }
14481     { \box_use:N \l__box_internal_box }
14482   }
14483   \box_set_ht:Nn \l__box_internal_box \c_zero_dim
14484 }
14485 \box_set_eq:NN #1 \l__box_internal_box
14486 }
14487 \cs_generate_variant:Nn \box_viewport:Nnnnn { c }

```

(End definition for `\box_viewport:Nnnnn` and `\box_viewport:cnnnn`. These functions are documented on page ??.)

41.4 Additions to l3clist

14488 `<@@=clist>`

`\clist_item:Nn` To avoid needing to test the end of the list at each step, we first compute the $\langle length \rangle$ of the list. If the item number is 0, less than $-\langle length \rangle$, or more than $\langle length \rangle$, the result is empty. If it is negative, but not less than $-\langle length \rangle$, add $\langle length \rangle + 1$ to the item number before performing the loop. The loop itself is very simple, return the item if the counter reached 1, otherwise, decrease the counter and repeat.

```

14489 \cs_new:Npn \clist_item:Nn #1#2
14490 {
14491   \exp_args:Nfo \__clist_item:nnNn
14492   { \clist_count:N #1 }
14493   #1
14494   \__clist_item_N_loop:nw
14495   {#2}
14496 }
14497 \cs_new:Npn \__clist_item:nnNn #1#2#3#4
14498 {
14499   \int_compare:nNnTF {#4} < \c_zero
14500   {
14501     \int_compare:nNnTF {#4} < { - #1 }
14502     { \use_none_delimit_by_q_stop:w }
14503     { \exp_args:Nf #3 { \int_eval:n { #4 + \c_one + #1 } } }
14504   }
14505   {
14506     \int_compare:nNnTF {#4} > {#1}
14507     { \use_none_delimit_by_q_stop:w }
14508     { #3 {#4} }
14509   }
14510   { } , #2 , \q_stop
14511 }
14512 \cs_new:Npn \__clist_item_N_loop:nw #1 #2,
14513 {
14514   \int_compare:nNnTF {#1} = \c_zero

```



```

14515     { \use_i_delimit_by_q_stop:nw { \exp_not:n {#2} } }
14516     { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
14517   }
14518   \cs_generate_variant:Nn \clist_item:Nn { c }

```

(End definition for \clist_item:Nn and \clist_item:cn. These functions are documented on page ??.)

```

\clist_item:nn
  \__clist_item_n:nw
  \__clist_item_n_loop:nw
  \__clist_item_n_end:n
  \__clist_item_n_strip:w

```

This starts in the same way as \clist_item:Nn by counting the items of the comma list. The final item should be space-trimmed before being brace-stripped, hence we insert a couple of odd-looking \prg_do_nothing: to avoid losing braces. Blank items are ignored.

```

14519   \cs_new:Npn \clist_item:nn #1#2
14520   {
14521     \exp_args:Nf \__clist_item:nnNn
14522     { \clist_count:n {#1} }
14523     {#1}
14524     \__clist_item_n:nw
14525     {#2}
14526   }
14527   \cs_new:Npn \__clist_item_n:nw #1
14528   { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
14529   \cs_new:Npn \__clist_item_n_loop:nw #1 #2,
14530   {
14531     \exp_args:No \tl_if_blank:nTF {#2}
14532     { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
14533     {
14534       \int_compare:nNnTF {#1} = \c_zero
14535       { \exp_args:No \__clist_item_n_end:n {#2} }
14536       {
14537         \exp_args:Nf \__clist_item_n_loop:nw
14538         { \int_eval:n { #1 - 1 } }
14539         \prg_do_nothing:
14540       }
14541     }
14542   }
14543   \cs_new:Npn \__clist_item_n_end:n #1 #2 \q_stop
14544   {
14545     \__tl_trim_spaces:nn { \q_mark #1 }
14546     { \exp_last_unbraced:No \__clist_item_n_strip:w } ,
14547   }
14548   \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 ??.)

```

\clist_set_from_seq:NN
\clist_set_from_seq:cN
\clist_set_from_seq:Nc
\clist_set_from_seq:cc

```

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_gset_from_seq:NN
\clist_gset_from_seq:cN
\clist_gset_from_seq:Nc
\clist_gset_from_seq:cc

```

```

14549   \cs_new_protected:Npn \clist_set_from_seq:NN
14550   { \__clist_set_from_seq:NnNNN \clist_clear:N \tl_set:Nx }
14551   \cs_new_protected:Npn \clist_gset_from_seq:NN
14552   { \__clist_set_from_seq:NnNNN \clist_gclear:N \tl_gset:Nx }

```

```

\__clist_set_from_seq:NnNNN
  \__clist_wrap_item:n
  \__clist_set_from_seq:w

```

```

14553 \cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4
14554 {
14555   \seq_if_empty:NTF #4
14556   { #1 #3 }
14557   {
14558     #2 #3
14559     {
14560       \exp_last_unbraced:Nf \use_none:n
14561       { \seq_map_function:NN #4 \__clist_wrap_item:n }
14562     }
14563   }
14564 }
14565 \cs_new:Npn \__clist_wrap_item:n #1
14566 {
14567   ,
14568   \tl_if_empty:oTF { \__clist_set_from_seq:w #1 ~ , #1 ~ }
14569   { \exp_not:n {#1} }
14570   { \exp_not:n { {#1} } }
14571 }
14572 \cs_new:Npn \__clist_set_from_seq:w #1 , #2 ~ { }
14573 \cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }
14574 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
14575 \cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }
14576 \cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }

```

(End definition for \clist_set_from_seq:NN and others. These functions are documented on page ??.)

\clist_const:Nn Creating and initializing a constant comma list is done in a way similar to \clist_set:Nn and \clist_const:cn, being careful to strip spaces.

```

\clist_const:Nx 14577 \cs_new_protected:Npn \clist_const:Nn #1#2
\clist_const:cx 14578 { \tl_const:Nx #1 { \__clist_trim_spaces:n {#2} } }
14579 \cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }

```

(End definition for \clist_const:Nn and others. These functions are documented on page ??.)

\clist_if_empty_p:n As usual, we insert a token (here ?) before grabbing any argument: this avoids losing braces. The argument of \tl_if_empty:oTF is empty if #1 is ? followed by blank spaces (besides, this particular variant of the emptiness test is optimized). If the item of the comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second auxiliary will grab \prg_return_false: as #2, unless every item in the comma list was blank and the loop actually got broken by the trailing \q_mark \prg_return_false: item.

```

\clist_if_empty_n:TF
\__clist_if_empty_n:w
\__clist_if_empty_n:wNw

```

```

14580 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
14581 {
14582   \__clist_if_empty_n:w ? #1
14583   , \q_mark \prg_return_false:
14584   , \q_mark \prg_return_true:
14585   \q_stop
14586 }
14587 \cs_new:Npn \__clist_if_empty_n:w #1 ,
14588 {

```

```

14589 \tl_if_empty:oTF { \use_none:nn #1 ? }
14590 { \__clist_if_empty_n:w ? }
14591 { \__clist_if_empty_n:wNw }
14592 }
14593 \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 191.)

```

41.5 Additions to l3coffins

```

14594 <@@=coffin>

```

41.6 Rotating coffins

`\l__coffin_sin_fp` Used for rotations to get the sine and cosine values.

```

\l__coffin_cos_fp 14595 \fp_new:N \l__coffin_sin_fp
14596 \fp_new:N \l__coffin_cos_fp

```

(End definition for `\l__coffin_sin_fp`. This function is documented on page ??.)

`\l__coffin_bounding_prop` A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```

14597 \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.

```

14598 \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 14599 \dim_new:N \l__coffin_left_corner_dim
\l__coffin_bottom_corner_dim 14600 \dim_new:N \l__coffin_right_corner_dim
\l__coffin_top_corner_dim 14601 \dim_new:N \l__coffin_bottom_corner_dim
14602 \dim_new:N \l__coffin_top_corner_dim

```

(End definition for `\l__coffin_left_corner_dim`. This function is documented on page ??.)

`\coffin_rotate:Nn` Rotating a coffin requires several steps which can be conveniently run together. The first step is to convert the angle given in degrees to one in radians. 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`

```

14603 \cs_new_protected:Npn \coffin_rotate:Nn #1#2
14604 {
14605   \fp_set:Nn \l__coffin_sin_fp { sin ( ( #2 ) * deg ) }
14606   \fp_set:Nn \l__coffin_cos_fp { cos ( ( #2 ) * deg ) }

```

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

```

14607 \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
14608 { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
14609 \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
14610 { \__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.

```

14611 \__coffin_set_bounding:N #1
14612 \prop_map_inline:Nn \l__coffin_bounding_prop
14613 { \__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.

```

14614 \__coffin_find_corner_maxima:N #1
14615 \__coffin_find_bounding_shift:
14616 \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.

```

14617 \hbox_set:Nn \l__coffin_internal_box
14618 {
14619   \tex_kern:D
14620   \__dim_eval:w
14621   \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim
14622   \__dim_eval_end:
14623   \box_move_down:nn { \l__coffin_bottom_corner_dim }
14624   { \box_use:N #1 }
14625 }

```

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.

```

14626 \box_set_ht:Nn \l__coffin_internal_box
14627 { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
14628 \box_set_dp:Nn \l__coffin_internal_box { 0 pt }
14629 \box_set_wd:Nn \l__coffin_internal_box
14630 { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
14631 \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.

```

14632 \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
14633 { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
14634 \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
14635 { \__coffin_shift_pole:Nnnnn #1 {##1} ##2 }
14636 }
14637 \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.

```

14638 \cs_new_protected:Npn \__coffin_set_bounding:N #1
14639 {
14640   \prop_put:Nnx \l__coffin_bounding_prop { tl }
14641   { { 0 pt } { \dim_use:N \box_ht:N #1 } }
14642   \prop_put:Nnx \l__coffin_bounding_prop { tr }
14643   { { \dim_use:N \box_wd:N #1 } { \dim_use:N \box_ht:N #1 } }
14644   \dim_set:Nn \l__coffin_internal_dim { - \box_dp:N #1 }
14645   \prop_put:Nnx \l__coffin_bounding_prop { bl }
14646   { { 0 pt } { \dim_use:N \l__coffin_internal_dim } }
14647   \prop_put:Nnx \l__coffin_bounding_prop { br }
14648   { { \dim_use:N \box_wd:N #1 } { \dim_use:N \l__coffin_internal_dim } }
14649 }

```

(End definition for __coffin_set_bounding:N. This function is documented on page ??.)

_coffin_rotate_bounding:nnn Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

```

14650 \cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
14651 {
14652   \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
14653   \prop_put:Nnx \l__coffin_bounding_prop {#1}
14654   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
14655 }
14656 \cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
14657 {
14658   \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
14659   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
14660   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
14661 }

```

(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.

```

14662 \cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
14663 {
14664   \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
14665   \__coffin_rotate_vector:nnNN {#5} {#6}
14666   \l__coffin_x_prime_dim \l__coffin_y_prime_dim
14667   \__coffin_set_pole:Nnx #1 {#2}
14668   {
14669     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
14670     { \dim_use:N \l__coffin_x_prime_dim }
14671     { \dim_use:N \l__coffin_y_prime_dim }
14672   }
14673 }

```

(End definition for _coffin_rotate_pole:nnnnn. 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.

```

14674 \cs_new_protected:Npn \_coffin_rotate_vector:nnNN #1#2#3#4
14675 {
14676   \dim_set:Nn #3
14677   {
14678     \fp_to_dim:n
14679     {
14680       \dim_to_fp:n {#1} * \l\_coffin_cos_fp
14681       - ( \dim_to_fp:n {#2} * \l\_coffin_sin_fp )
14682     }
14683   }
14684   \dim_set:Nn #4
14685   {
14686     \fp_to_dim:n
14687     {
14688       \dim_to_fp:n {#1} * \l\_coffin_sin_fp
14689       + ( \dim_to_fp:n {#2} * \l\_coffin_cos_fp )
14690     }
14691   }
14692 }

```

(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.

```

14693 \cs_new_protected:Npn \_coffin_find_corner_maxima:N #1
14694 {
14695   \dim_set:Nn \l\_coffin_top_corner_dim { -\c_max_dim }
14696   \dim_set:Nn \l\_coffin_right_corner_dim { -\c_max_dim }
14697   \dim_set:Nn \l\_coffin_bottom_corner_dim { \c_max_dim }
14698   \dim_set:Nn \l\_coffin_left_corner_dim { \c_max_dim }
14699   \prop_map_inline:cn { l\_coffin_corners_ \_int_value:w #1 _prop }
14700   { \_coffin_find_corner_maxima_aux:nn ##2 }
14701 }
14702 \cs_new_protected:Npn \_coffin_find_corner_maxima_aux:nn #1#2
14703 {
14704   \dim_set:Nn \l\_coffin_left_corner_dim
14705   { \dim_min:nn { \l\_coffin_left_corner_dim } {#1} }
14706   \dim_set:Nn \l\_coffin_right_corner_dim
14707   { \dim_max:nn { \l\_coffin_right_corner_dim } {#1} }
14708   \dim_set:Nn \l\_coffin_bottom_corner_dim
14709   { \dim_min:nn { \l\_coffin_bottom_corner_dim } {#2} }
14710   \dim_set:Nn \l\_coffin_top_corner_dim

```

```

14711     { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
14712   }

```

(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.

```

14713 \cs_new_protected_nopar:Npn \__coffin_find_bounding_shift:
14714 {
14715   \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
14716   \prop_map_inline:Nn \l__coffin_bounding_prop
14717     { \__coffin_find_bounding_shift_aux:nn ##2 }
14718 }
14719 \cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
14720 {
14721   \dim_set:Nn \l__coffin_bounding_shift_dim
14722     { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
14723 }

```

(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.

```

14724 \cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
14725 {
14726   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _ prop } {#2}
14727   {
14728     { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
14729     { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
14730   }
14731 }
14732 \cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
14733 {
14734   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _ prop } {#2}
14735   {
14736     { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
14737     { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
14738     {#5} {#6}
14739   }
14740 }

```

(End definition for `__coffin_shift_corner:Nnnn`. This function is documented on page ??.)

41.7 Resizing coffins

```

\l__coffin_scale_x_fp
\l__coffin_scale_y_fp

```

Storage for the scaling factors in x and y , respectively.

```

14741 \fp_new:N \l__coffin_scale_x_fp
14742 \fp_new:N \l__coffin_scale_y_fp

```

(End definition for `\l__coffin_scale_x_fp`. This function is documented on page ??.)

`\l_coffin_scaled_total_height_dim` When scaling, the values given have to be turned into absolute values.

`\l_coffin_scaled_width_dim`

```

14743 \dim_new:N \l__coffin_scaled_total_height_dim
14744 \dim_new:N \l__coffin_scaled_width_dim

```

(End definition for \l_coffin_scaled_total_height_dim. This function is documented on page ??.)

`\coffin_resize:Nnn` Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.

`\coffin_resize:cnn`

```

14745 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
14746 {
14747   \fp_set:Nn \l__coffin_scale_x_fp
14748     { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
14749   \fp_set:Nn \l__coffin_scale_y_fp
14750     {
14751       \dim_to_fp:n {#3} / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
14752     }
14753   \box_resize:Nnn #1 {#2} {#3}
14754   \__coffin_resize_common:Nnn #1 {#2} {#3}
14755 }
14756 \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.

```

14757 \cs_new_protected:Npn \__coffin_resize_common:Nnn #1#2#3
14758 {
14759   \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
14760     { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
14761   \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
14762     { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }

```

Negative x -scaling values will place the poles in the wrong location: this is corrected here.

```

14763   \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
14764     {
14765       \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
14766         { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
14767       \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
14768         { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
14769     }
14770 }

```

(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.


```

14771 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
14772 {
14773   \fp_set:Nn \l__coffin_scale_x_fp {#2}
14774   \fp_set:Nn \l__coffin_scale_y_fp {#3}
14775   \box_scale:Nnn #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
14776   \dim_set:Nn \l__coffin_internal_dim
14777     { \coffin_ht:N #1 + \coffin_dp:N #1 }
14778   \dim_set:Nn \l__coffin_scaled_total_height_dim
14779     { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
14780   \dim_set:Nn \l__coffin_scaled_width_dim
14781     { -\fp_abs:n { \l__coffin_scale_x_fp } \coffin_wd:N #1 }
14782   \__coffin_resize_common:Nnn #1
14783     { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
14784 }
14785 \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.

```

14786 \cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
14787 {
14788   \dim_set:Nn #3
14789     { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
14790   \dim_set:Nn #4
14791     { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
14792 }

```

(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
14793 \cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
14794 {
14795   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
14796   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
14797     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
14798 }
14799 \cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
14800 {
14801   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
14802   \__coffin_set_pole:Nnx #1 {#2}
14803   {
14804     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
14805     {#5} {#6}
14806   }
14807 }

```

(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`

```

14808 \cs_new_protected:Npn \__coffin_x_shift_corner:Nnnn #1#2#3#4
14809 {
14810   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
14811   {
14812     { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
14813   }
14814 }
14815 \cs_new_protected:Npn \__coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
14816 {
14817   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } {#2}
14818   {
14819     { \dim_eval:n #3 + \box_wd:N #1 } {#4}
14820     {#5} {#6}
14821   }
14822 }

```

(End definition for `_coffin_x_shift_corner:Nnnn`. This function is documented on page ??.)

41.8 Additions to l3file

```

14823 <@@=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`

```

14824 \cs_new_nopar:Npn \ior_map_break:
14825 { \__prg_map_break:Nn \ior_map_break: { } }
14826 \cs_new_nopar:Npn \ior_map_break:n
14827 { \__prg_map_break:Nn \ior_map_break: }

```

(End definition for `\ior_map_break:` and `\ior_map_break:n`. These functions are documented on page 193.)

`\ior_map_inline:Nn` Mapping to an input stream can be done on either a token or a string basis, hence the
`\ior_str_map_inline:Nn` set up. Within that, there is a check to avoid reading past the end of a file, hence the
`__ior_map_inline:NNn` two applications of `\ior_if_eof:N`. This mapping cannot be nested as the stream has
`__ior_map_inline:NNNn` only one “current line”.
`__ior_map_inline_loop:NNN`

`\l__ior_internal_tl`

```

14828 \cs_new_protected_nopar:Npn \ior_map_inline:Nn
14829 { \__ior_map_inline:NNn \ior_get:NN }
14830 \cs_new_protected_nopar:Npn \ior_str_map_inline:Nn
14831 { \__ior_map_inline:NNn \ior_get_str:NN }
14832 \cs_new_protected_nopar:Npn \__ior_map_inline:NNn
14833 {
14834   \int_gincr:N \g__prg_map_int
14835   \exp_args:Nc \__ior_map_inline:NNNn
14836   { __prg_map_ \int_use:N \g__prg_map_int :n }
14837 }
14838 \cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
14839 {
14840   \cs_set:Npn #1 ##1 {#4}

```

```

14841 \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
14842 \__prg_break_point:Nn \ior_map_break:
14843 { \int_gdecr:N \g__prg_map_int }
14844 }
14845 \cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
14846 {
14847   #2 #3 \l__ior_internal_tl
14848   \ior_if_eof:NF #3
14849   {
14850     \exp_args:No #1 \l__ior_internal_tl
14851     \__ior_map_inline_loop:NNN #1#2#3
14852   }
14853 }
14854 \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 ??.)

41.9 Additions to l3fp

14855 <@@=fp>

\fp_set_from_dim:Nn
\fp_set_from_dim:cn
\fp_gset_from_dim:Nn
\fp_gset_from_dim:cn

Use the appropriate function from l3fp-convert.

```

14856 \cs_new_protected:Npn \fp_set_from_dim:Nn #1#2
14857 { \tl_set:Nx #1 { \dim_to_fp:n {#2} } }
14858 \cs_new_protected:Npn \fp_gset_from_dim:Nn #1#2
14859 { \tl_gset:Nx #1 { \dim_to_fp:n {#2} } }
14860 \cs_generate_variant:Nn \fp_set_from_dim:Nn { c }
14861 \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 ??.)

41.10 Additions to l3prop

14862 <@@=prop>

\prop_map_tokens:Nn
\prop_map_tokens:cn
__prop_map_tokens:nwn

The mapping 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.

```

14863 \cs_new:Npn \prop_map_tokens:Nn #1#2
14864 {
14865   \exp_last_unbraced:Nno \__prop_map_tokens:nwn {#2} #1
14866   \s__prop \q_recursion_tail \s__prop { }
14867   \__prg_break_point:Nn \prop_map_break: { }
14868 }
14869 \cs_new:Npn \__prop_map_tokens:nwn #1 \s__prop #2 \s__prop #3
14870 {
14871   \if_meaning:w \q_recursion_tail #2
14872   \exp_after:wN \prop_map_break:
14873   \fi:
14874   \use:n {#1} {#2} {#3}
14875   \__prop_map_tokens:nwn {#1}

```

```

14876 }
14877 \cs_generate_variant:Nn \prop_map_tokens:Nn { c }
(End definition for \prop_map_tokens:Nn and \prop_map_tokens:cn. These functions are documented
on page ??.)

```

\prop_get:Nn Getting the value corresponding to a key in a property list in an expandable fashion is similar to mapping some tokens. Go through the property list one $\langle key \rangle$ – $\langle value \rangle$ pair at a time: the arguments of `__prop_get_Nn:nwn` are the $\langle key \rangle$ we are looking for, a $\langle key \rangle$ of the property list, and its associated value. The $\langle keys \rangle$ are compared (as strings). If they match, the $\langle value \rangle$ is returned, within `\exp_not:n`. The loop terminates even if the $\langle key \rangle$ is missing, and yields an empty value, because we have appended the appropriate $\langle key \rangle$ – $\langle empty\ value \rangle$ pair to the property list.

```

14878 \cs_new:Npn \prop_get:Nn #1#2
14879 {
14880   \exp_last_unbraced:Noo \__prop_get_Nn:nwn { \tl_to_str:n {#2} } #1
14881   \s__prop \tl_to_str:n {#2} \s__prop { }
14882   \__prg_break_point:
14883 }
14884 \cs_new:Npn \__prop_get_Nn:nwn #1 \s__prop #2 \s__prop #3
14885 {
14886   \str_if_eq_x:nnTF {#1} {#2}
14887   { \__prg_break:n { \exp_not:n {#3} } }
14888   { \__prop_get_Nn:nwn {#1} }
14889 }
14890 \cs_generate_variant:Nn \prop_get:Nn { c }

```

(End definition for `\prop_get:Nn` and `\prop_get:cn`. These functions are documented on page ??.)

41.11 Additions to l3seq

```

14891 \@@=seq

```

\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 `__seq_item:nnn { ? __prg_break: } { }` will be used by the auxiliary, terminating the loop and returning nothing at all.

```

14892 \cs_new:Npn \seq_item:Nn #1#2
14893 {
14894   \exp_last_unbraced:Nfo \__seq_item:nnn
14895   {
14896     \int_eval:n
14897     {
14898       \int_compare:nNnT {#2} < \c_zero
14899       { \seq_count:N #1 + \c_one + }
14900       #2
14901     }
14902   }
14903   #1
14904   { ? \__prg_break: }
14905   { }

```

```

14906     \__prg_break_point:
14907   }
14908   \cs_new:Npn \__seq_item:nnn #1#2#3
14909   {
14910     \use_none:n #2
14911     \int_compare:nNnTF {#1} = \c_one
14912     { \__prg_break:n { \exp_not:n {#3} } }
14913     { \exp_args:Nf \__seq_item:nnn { \int_eval:n { #1 - 1 } } }
14914   }
14915   \cs_generate_variant:Nn \seq_item:Nn { c }

```

(End definition for \seq_item:Nn and \seq_item:cn. These functions are documented on page ??.)

```

\seq_mapthread_function:NNN
\seq_mapthread_function:NcN
\seq_mapthread_function:cNN
\seq_mapthread_function:ccN
\__seq_mapthread_function:NN
  \__seq_mapthread_function:Nnnwnn

```

The idea here is to first expand both of the sequences, adding the usual { ? __prg_break: } { } to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first token of #2 and #5, which for items in the sequences will both be __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.

```

14916   \cs_new:Npn \seq_mapthread_function:NNN #1#2#3
14917   {
14918     \exp_after:wN \__seq_mapthread_function:NN
14919     \exp_after:wN #3
14920     \exp_after:wN #1
14921     #2
14922     { ? \__prg_break: } { }
14923     \__prg_break_point:
14924   }
14925   \cs_new:Npn \__seq_mapthread_function:NN #1#2
14926   {
14927     \exp_after:wN \__seq_mapthread_function:Nnnwnn
14928     \exp_after:wN #1
14929     #2
14930     { ? \__prg_break: } { }
14931     \q_stop
14932   }
14933   \cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \q_stop #5#6
14934   {
14935     \use_none:n #2
14936     \use_none:n #5
14937     #1 {#3} {#6}
14938     \__seq_mapthread_function:Nnnwnn #1 #4 \q_stop
14939   }
14940   \cs_generate_variant:Nn \seq_mapthread_function:NNN {      Nc }
14941   \cs_generate_variant:Nn \seq_mapthread_function:NNN { c , cc }

```

(End definition for \seq_mapthread_function:NNN and others. These functions are documented on page ??.)

```

\seq_set_from_clist:NN
\seq_set_from_clist:cN
\seq_set_from_clist:Nc
\seq_set_from_clist:cc
\seq_set_from_clist:Nn
\seq_set_from_clist:cn
\seq_gset_from_clist:NN
\seq_gset_from_clist:cN
\seq_gset_from_clist:Nc
\seq_gset_from_clist:cc
\seq_gset_from_clist:Nn

```

Setting a sequence from a comma-separated list is done using a simple mapping.

```

14942 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
14943 {
14944   \tl_set:Nx #1
14945   { \clist_map_function:NN #2 \__seq_wrap_item:n }
14946 }
14947 \cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
14948 {
14949   \tl_set:Nx #1
14950   { \clist_map_function:nN {#2} \__seq_wrap_item:n }
14951 }
14952 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
14953 {
14954   \tl_gset:Nx #1
14955   { \clist_map_function:NN #2 \__seq_wrap_item:n }
14956 }
14957 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
14958 {
14959   \tl_gset:Nx #1
14960   { \clist_map_function:nN {#2} \__seq_wrap_item:n }
14961 }
14962 \cs_generate_variant:Nn \seq_set_from_clist:NN { Nc }
14963 \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
14964 \cs_generate_variant:Nn \seq_set_from_clist:Nn { c }
14965 \cs_generate_variant:Nn \seq_gset_from_clist:NN { Nc }
14966 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
14967 \cs_generate_variant:Nn \seq_gset_from_clist:Nn { c }

```

(End definition for `\seq_set_from_clist:NN` and others. These functions are documented on page ??.)

```

\seq_reverse:N Previously, \seq_reverse:N was coded by collecting the items in reverse order after an
\seq_reverse:c \exp_stop_f: marker.
\seq_greverse:N
\seq_greverse:c \cs_new_protected:Npn \seq_reverse:N #1
\__seq_tmp:w {
\__seq_reverse:NN \cs_set_eq:NN \@@_item:n \@@_reverse_item:nw
\__seq_reverse_item:nwn \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.

```

14968 \cs_new_protected_nopar:Npn \__seq_tmp:w { }
14969 \cs_new_protected_nopar:Npn \seq_reverse:N
14970 { \__seq_reverse:NN \tl_set:Nx }
14971 \cs_new_protected_nopar:Npn \seq_greverse:N
14972 { \__seq_reverse:NN \tl_gset:Nx }
14973 \cs_new_protected:Npn \__seq_reverse:NN #1 #2
14974 {
14975   \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
14976   \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
14977   #1 #2 { #2 \exp_not:n { } }
14978   \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
14979 }
14980 \cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
14981 {
14982   #2
14983   \exp_not:n { \__seq_item:n {#1} #3 }
14984 }
14985 \cs_generate_variant:Nn \seq_reverse:N { c }
14986 \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End definition for \seq_reverse:N and others. These functions are documented on page ??.)

$\text{\seq_set_filter:NNn}$
 $\text{\seq_gset_filter:NNn}$
 $\text{__seq_set_filter:NNNn}$

Similar to $\text{\seq_map_inline:Nn}$, without a $\text{__prg_break_point:}$ 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.

```

14987 \cs_new_protected_nopar:Npn \seq_set_filter:NNn
14988 { \__seq_set_filter:NNNn \tl_set:Nx }
14989 \cs_new_protected_nopar:Npn \seq_gset_filter:NNn
14990 { \__seq_set_filter:NNNn \tl_gset:Nx }
14991 \cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
14992 {
14993   \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
14994   #1 #2 { #3 }
14995   \__seq_pop_item_def:
14996 }

```

(End definition for $\text{\seq_set_filter:NNn}$ and $\text{\seq_gset_filter:NNn}$. These functions are documented on page 195.)

\seq_set_map:NNn
 \seq_gset_map:NNn
 $\text{__seq_set_map:NNNn}$

Very similar to $\text{\seq_set_filter:NNn}$. We could actually merge the two within a single function, but it would have weird semantics.

```

14997 \cs_new_protected_nopar:Npn \seq_set_map:NNn
14998 { \__seq_set_map:NNNn \tl_set:Nx }
14999 \cs_new_protected_nopar:Npn \seq_gset_map:NNn
15000 { \__seq_set_map:NNNn \tl_gset:Nx }

```

```

15001 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
15002 {
15003   \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
15004   #1 #2 { #3 }
15005   \__seq_pop_item_def:
15006 }

```

(End definition for `\seq_set_map:NNn` and `\seq_gset_map:NNn`. These functions are documented on page 195.)

41.12 Additions to l3skip

```

15007 <@@=dim>

```

`\dim_to_pt:n` A copy of the internal function `__dim_strip_pt:n`, which should perhaps be eliminated in favor of `\dim_to_pt:n`.

```

15008 \cs_new_eq:NN \dim_to_pt:n \__dim_strip_pt:n

```

(End definition for `\dim_to_pt:n`. This function is documented on page 195.)

`\dim_to_unit:nn` An analog of `\dim_ratio:nn` that produces a decimal number as its result, rather than a rational fraction for use within dimension expressions. The naive implementation as

```

\cs_new:Npn \dim_to_unit:nn #1#2
{ \dim_to_pt:n { 1pt * \dim_ratio:nn {#1} {#2} } }

```

would not ignore trailing tokens (see documentation), so we need a bit more work.

```

15009 \cs_new:Npn \dim_to_unit:nn #1#2
15010 {
15011   \dim_to_pt:n
15012   {
15013     1pt * \__dim_to_unit:n { \dim_to_pt:n {#1} pt }
15014     / \__dim_to_unit:n { \dim_to_pt:n {#2} pt }
15015   }
15016 }
15017 \cs_new:Npn \__dim_to_unit:n #1
15018 { \__int_value:w \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_to_unit:nn`. This function is documented on page 195.)

```

15019 <@@=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.

```

15020 \cs_new:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
15021 {
15022   \skip_if_finite:nTF {#1}
15023   {
15024     #3 = \etex_gluestretch:D #1 \scan_stop:
15025     #4 = \etex_glueshrink:D #1 \scan_stop:
15026   }

```



```

15027     {
15028         #3 = \c_zero_skip
15029         #4 = \c_zero_skip
15030         #2
15031     }
15032 }

```

(End definition for `\skip_split_finite_else_action:nnNN`. This function is documented on page 196.)

41.13 Additions to `l3tl`

```

15033 <@@=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.

`\tl_if_single_token:nTF`

```

15034 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
15035 {
15036     \tl_if_head_is_N_type:nTF {#1}
15037     { \__str_if_eq_x_return:nn { \exp_not:o { \use_none:n #1 } } { } }
15038     { \__str_if_eq_x_return:nn { \exp_not:n {#1} } { ~ } }
15039 }

```

(End definition for `\tl_if_single_token:n`. These functions are documented on page 196.)

`\tl_reverse_tokens:n`
`__tl_reverse_group:nn`

The same as `\tl_reverse:n` but with recursion within brace groups.

```

15040 \cs_new:Npn \tl_reverse_tokens:n #1
15041 {
15042     \etex_unexpanded:D \exp_after:wN
15043     {
15044         \tex_romannumeral:D
15045         \__tl_act:NNNnn
15046         \__tl_reverse_normal:nN
15047         \__tl_reverse_group:nn
15048         \__tl_reverse_space:n
15049         { }
15050         {#1}
15051     }
15052 }
15053 \cs_new:Npn \__tl_reverse_group:nn #1
15054 {
15055     \__tl_act_group_recurse:Nnn
15056     \__tl_act_reverse_output:n
15057     { \tl_reverse_tokens:n }
15058 }

```

`__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.

```

15059 \cs_new:Npn \__tl_act_group_recurse:Nnn #1#2#3

```

```

15060 {
15061   \exp_args:Nf #1
15062   { \exp_after:wN \exp_after:wN \exp_after:wN { #2 {#3} } }
15063 }

```

(End definition for `\tl_reverse_tokens:n`. This function is documented on page 196.)

```

\tl_count_tokens:n
\__tl_act_count_normal:nN
\__tl_act_count_group:nn
\__tl_act_count_space:n

```

The token count is computed through an `\int_eval:n` construction. Each 1+ is output to the *left*, into the integer expression, and the sum is ended by the `\c_zero` inserted by `__tl_act_end:wn`. Somewhat a hack.

```

15064 \cs_new:Npn \tl_count_tokens:n #1
15065 {
15066   \int_eval:n
15067   {
15068     \__tl_act:NNNnn
15069     \__tl_act_count_normal:nN
15070     \__tl_act_count_group:nn
15071     \__tl_act_count_space:n
15072     { }
15073     {#1}
15074   }
15075 }
15076 \cs_new:Npn \__tl_act_count_normal:nN #1 #2 { 1 + }
15077 \cs_new:Npn \__tl_act_count_space:n #1 { 1 + }
15078 \cs_new:Npn \__tl_act_count_group:nn #1 #2
15079 { 2 + \tl_count_tokens:n {#2} + }

```

(End definition for `\tl_count_tokens:n`. This function is documented on page 196.)

```

\c__tl_act_uppercase_tl
\c__tl_act_lowercase_tl

```

These constants contain the correspondence between lowercase and uppercase letters, in the form `aAbBcC...` and `AaBbCc...` respectively.

```

15080 \tl_const:Nn \c__tl_act_uppercase_tl
15081 {
15082   aA bB cC dD eE fF gG hH iI jJ kK lL mM
15083   nN oO pP qQ rR sS tT uU vV wW xX yY zZ
15084 }
15085 \tl_const:Nn \c__tl_act_lowercase_tl
15086 {
15087   Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm
15088   Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz
15089 }

```

(End definition for `\c__tl_act_uppercase_tl` and `\c__tl_act_lowercase_tl`. These variables are documented on page ??.)

```

\tl_expandable_uppercase:n
\tl_expandable_lowercase:n
\__tl_act_case_normal:nN
\__tl_act_case_group:nn
\__tl_act_case_space:n

```

The only difference between uppercasing and lowercasing is the table of correspondence that is used. As for other token list actions, we feed `__tl_act:NNNnn` three functions, and this time, we use the *parameters* argument to carry which case-changing we are applying. A space is simply output. A normal token is compared to each letter in the alphabet using `\str_if_eq:nn` tests, and converted if necessary to upper/lowercase, before being output. For a group, we must perform the conversion within the group

(the `\exp_after:wN` trigger `\romannumeral`, which expands fully to give the converted group), then output.

```

15090 \cs_new:Npn \tl_expandable_uppercase:n #1
15091 {
15092   \etex_unexpanded:D \exp_after:wN
15093   {
15094     \tex_romannumeral:D
15095     \__tl_act_case_aux:nn { \c__tl_act_uppercase_tl } {#1}
15096   }
15097 }
15098 \cs_new:Npn \tl_expandable_lowercase:n #1
15099 {
15100   \etex_unexpanded:D \exp_after:wN
15101   {
15102     \tex_romannumeral:D
15103     \__tl_act_case_aux:nn { \c__tl_act_lowercase_tl } {#1}
15104   }
15105 }
15106 \cs_new:Npn \__tl_act_case_aux:nn
15107 {
15108   \__tl_act:NNNnn
15109   \__tl_act_case_normal:nN
15110   \__tl_act_case_group:nn
15111   \__tl_act_case_space:n
15112 }
15113 \cs_new:Npn \__tl_act_case_space:n #1 { \__tl_act_output:n {~} }
15114 \cs_new:Npn \__tl_act_case_normal:nN #1 #2
15115 {
15116   \exp_args:Nf \__tl_act_output:n
15117   {
15118     \exp_args:NNo \str_case:nnn #2 {#1}
15119     { \exp_stop_f: #2 }
15120   }
15121 }
15122 \cs_new:Npn \__tl_act_case_group:nn #1 #2
15123 {
15124   \exp_after:wN \__tl_act_output:n \exp_after:wN
15125   { \exp_after:wN { \tex_romannumeral:D \__tl_act_case_aux:nn {#1} {#2} } }
15126 }

```

(End definition for `\tl_expandable_uppercase:n` and `\tl_expandable_lowercase:n`. These functions are documented on page 196.)

`\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`

```

15127 \cs_new:Npn \tl_item:nn #1#2
15128 {
15129   \exp_args:Nf \__tl_item:nn
15130   {

```

```

15131         \int_eval:n
15132         {
15133             \int_compare:nNnT {#2} < \c_zero
15134             { \tl_count:n {#1} + \c_one + }
15135             #2
15136         }
15137     }
15138     #1
15139     \q_recursion_tail
15140     \__prg_break_point:
15141 }
15142 \cs_new:Npn \__tl_item:nn #1#2
15143 {
15144     \__quark_if_recursion_tail_break:nN {#2} \__prg_break:
15145     \int_compare:nNnTF {#1} = \c_one
15146     { \__prg_break:n { \exp_not:n {#2} } }
15147     { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
15148 }
15149 \cs_new_nopar:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
15150 \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 ??.)

41.14 Additions to l3tokens

```

15151 <@@=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`

```

15152 \group_begin:
15153 \char_set_catcode_active:N \^^@
15154 \cs_set:Npn \char_tmp:NN #1#2
15155 {
15156     \cs_new:Npn #1 ##1
15157     {
15158         \char_set_catcode_active:n { '##1 }
15159         \group_begin:
15160         \char_set_lccode:nn { '\^^@ } { '##1 }
15161         \tl_to_lowercase:n { \group_end: #2 ^^@ }
15162     }
15163 }
15164 \char_tmp:NN \char_set_active:Npn \cs_set:Npn
15165 \char_tmp:NN \char_set_active:Npx \cs_set:Npx
15166 \char_tmp:NN \char_gset_active:Npn \cs_gset:Npn
15167 \char_tmp:NN \char_gset_active:Npx \cs_gset:Npx
15168 \char_tmp:NN \char_set_active_eq:NN \cs_set_eq:NN
15169 \char_tmp:NN \char_gset_active_eq:NN \cs_gset_eq:NN
15170 \group_end:

```

(End definition for `\char_set_active:Npn` and `\char_set_active:Npx`. These functions are documented on page 197.)

```

15171 <@@=peek>

```

`\peek_N_type:TF` All tokens are N-type tokens, except in four cases: begin-group tokens, end-group tokens, space tokens with character code 32, and outer tokens. Since `\l_peek_token` might be
`__peek_execute_branches_N_type:` outer, we cannot use the convenient `\bool_if:nTF` function, and must resort to the old
`__peek_N_type:w` trick of using `\ifodd` to expand a set of tests. The `false` branch of this test is taken if the
`__peek_N_type_aux:nnw` 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.

```

15172 \group_begin:
15173   \char_set_catcode_other:N \O
15174   \char_set_catcode_other:N \U
15175   \char_set_catcode_other:N \T
15176   \char_set_catcode_other:N \E
15177   \char_set_catcode_other:N \R
15178   \tl_to_lowercase:n
15179   {
15180     \cs_new_protected_nopar:Npn \__peek_execute_branches_N_type:
15181     {
15182       \if_int_odd:w
15183         \if_catcode:w \exp_not:N \l_peek_token { \c_two \fi:
15184         \if_catcode:w \exp_not:N \l_peek_token } \c_two \fi:
15185         \if_meaning:w \l_peek_token \c_space_token \c_two \fi:
15186         \c_one
15187         \exp_after:wN \__peek_N_type:w
15188         \token_to_meaning:N \l_peek_token
15189         \q_mark \__peek_N_type_aux:nnw
15190         OUTER \q_mark \use_none_delimit_by_q_stop:w
15191         \q_stop
15192         \exp_after:wN \__peek_true:w
15193       \else:
15194         \exp_after:wN \__peek_false:w
15195       \fi:
15196     }
15197     \cs_new_protected:Npn \__peek_N_type:w #1 OUTER #2 \q_mark #3
15198     { #3 {#1} {#2} }
15199   }
15200 \group_end:
15201 \cs_new_protected:Npn \__peek_N_type_aux:nnw #1 #2 #3 \fi:
15202 {
15203   \fi:
15204   \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }

```

```

15205         { \_peek_true:w }
15206         { \tl_if_empty:nTF {#2} { \_peek_true:w } { \_peek_false:w } }
15207     }
15208 \cs_new_protected_nopar:Npn \peek_N_type:TF
15209     { \_peek_token_generic:NNTF \_peek_execute_branches_N_type: \scan_stop: }
15210 \cs_new_protected_nopar:Npn \peek_N_type:T
15211     { \_peek_token_generic:NNT \_peek_execute_branches_N_type: \scan_stop: }
15212 \cs_new_protected_nopar:Npn \peek_N_type:F
15213     { \_peek_token_generic:NNF \_peek_execute_branches_N_type: \scan_stop: }
(End definition for \peek_N_type:. This function is documented on page 198.)
15214 </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	
\!	7763
\"	2729, 2744
\#	6, 2744, 9527
\\$	2730, 2744
\%	2744, 9529
.	178
\	2731, 2744, 7763, 7764, 11174, 11200
\)	11119, 11355
**	179
*	2709, 2711, 2715, 2722, 9494, 9495, 10929
\,	8453, 8455, 11373
\-	319, 10346
\.bool_gset:N	8762
\.bool_gset:c	8762
\.bool_gset_inverse:N	8770
\.bool_gset_inverse:c	8770
\.bool_set:N	8762
\.bool_set:c	8762
\.bool_set_inverse:N	8770
\.bool_set_inverse:c	8770
\.choice:	8778
\.choice_code:n	9078
\.choice_code:x	9078
\.choices:Vn	8780
\.choices:nn	8780
\.choices:on	8780
\.choices:xn	8780
\.clist_gset:N	8790
\.clist_gset:c	8790
\.clist_set:N	8790
\.clist_set:c	8790
\.code:n	8788
\.default:V	8798
\.default:n	8798
\.default:o	8798
\.default:x	8798
\.dim_gset:N	8806
\.dim_gset:c	8806
\.dim_set:N	8806
\.dim_set:c	8806
\.fp_gset:N	8814
\.fp_gset:c	8814
\.fp_set:N	8814
\.fp_set:c	8814
\.generate_choices:n	9115
\.initial:V	8822
\.initial:n	8822
\.initial:o	8822
\.initial:x	8822
\.int_gset:N	8830
\.int_gset:c	8830
\.int_set:N	8830
\.int_set:c	8830
\.meta:n	8838
\.meta:nn	8840
\.multichoice:	8842
\.multichoices:Vn	8842
\.multichoices:nn	8842
\.multichoices:on	8842
\.multichoices:xn	8842
\.skip_gset:N	8852
\.skip_gset:c	8852
\.skip_set:N	8852
\.skip_set:c	8852
\.tl_gset:N	8860
\.tl_gset:c	8860
\.tl_gset_x:N	8860
\.tl_gset_x:c	8860
\.tl_set:N	8860
\.tl_set:c	8860
\.tl_set_x:N	8860
\.tl_set_x:c	8860
\.value_forbidden:	8876
\.value_required:	8876
\/	318
?:	178
\:	1077, 2821, 3020
\:::	34, 1630, 1631, 1632, 1632, 1633, 1634, 1635, 1636, 1638, 1640, 1647, 1652, 1658, 1777, 1778, 1779, 1780, 1781, 1782, 1783, 1784, 1785, 1786, 1787, 1788, 1789, 1790, 1791, 1792, 1793, 1794, 1795, 1796, 1797, 1798, 1799, 1800, 1801, 1802, 1803, 1804, 1806, 1811, 1813, 1818, 1823, 1860, 1861, 1862, 1863, 1864

\:N	34, 1634, 1634,	9669, 9676, 9683, 10132, 10135,
	1787, 1793, 1794, 1798, 1799, 1863	10136, 10137, 10144, 10147, 10148
\:V	34, 1652, 1652, 1783	_fp_decimate_auxi:Nnnnn 9883
\:V_unbraced	1805, 1813	_fp_decimate_auxii:Nnnnn 9883
\:c	34, 1636, 1636,	_fp_decimate_auxiii:Nnnnn 9883
	1778, 1786, 1788, 1795, 1802, 1803	_fp_decimate_auxiv:Nnnnn 9883
\:f	34, 1640,	_fp_decimate_auxix:Nnnnn 9883
	1640, 1779, 1780, 1781, 1785, 1862	_fp_decimate_auxv:Nnnnn 9883
\:f_unbraced	1805, 1806	_fp_decimate_auxvi:Nnnnn 9883
\:n	34, 1633, 1633, 1778, 1781,	_fp_decimate_auxvii:Nnnnn 9883
	1782, 1783, 1789, 1793, 1795, 1796,	_fp_decimate_auxviii:Nnnnn ... 9883
	1798, 1800, 1801, 1803, 1860, 1863	_fp_decimate_auxx:Nnnnn 9883
\:o ..	34, 1638, 1638, 1779, 1782, 1784,	_fp_decimate_auxxi:Nnnnn 9883
	1785, 1786, 1790, 1791, 1793, 1794,	_fp_decimate_auxxii:Nnnnn 9883
	1796, 1797, 1799, 1801, 1804, 1861	_fp_decimate_auxxiii:Nnnnn ... 9883
\:o_unbraced		_fp_decimate_auxxiv:Nnnnn 9883
	1805, 1811, 1860, 1861, 1862, 1863	_fp_decimate_auxxv:Nnnnn 9883
\:p	34, 1635, 1635	_fp_decimate_auxxvi:Nnnnn 9883
\:v	34, 1652, 1658	{ ... 4, 2744, 7761, 8359, 8363, 8364, 9526
\:v_unbraced	1805, 1818	} ... 5, 2744, 7762, 8359, 8363, 8364, 9528
\:x	34, 1647, 1647, 1777, 1787,	^ 7, 10, 90, 2732,
	1788, 1789, 1790, 1791, 1792, 1798,	2744, 8270, 9501, 13024, 15153, 15160
	1799, 1800, 1801, 1802, 1803, 1804	\ 2733, 2744, 10346
\:x_unbraced	1805, 1823, 1864	_bool__0:w 2216
\:N	11372	_bool__1:w 2214
\;	2821, 3019, 3020	_bool_(Nw 2194
\=	8452, 8454	_bool_)0:w 2210
\?	11322	_bool_)1:w 2210
\@	1077, 1078, 4729, 4730	_bool_:Nw 2192
\@end	742	_bool_S0:w 2210
\@hyph	745	_bool_S1:w 2210
\@input	746	_bool_choose:NNN 2196, 2200, 2201, 2201
\@italiccorr	747	_bool_eval_skip_to_end_auxi:Nw ...
\@underline	748 2216, 2216, 2217, 2219, 2233
\@addtofilelist	9249	_bool_eval_skip_to_end_auxii:Nw ..
\@currname	9142 2216, 2221, 2225
\@filelist	9248, 9275, 9277, 9292	_bool_eval_skip_to_end_auxiii:Nw .
\@ifpackageloaded	142, 223 2216, 2229, 2231
\@namedef	205	_bool_get_next:NN 2180,
\@nil	200, 208	2182, 2182, 2193, 2197, 2214, 2215
\@popfilename	170, 188, 190	_bool_if_left_parentheses:wwn ...
\@pushfilename	170, 171, 186 2162, 2165, 2172, 2173
\@tempa	55, 57, 65	_bool_if_or:wwn 2162, 2168, 2176, 2177
\\	1474, 2744, 7595, 7596, 7597, 7598,	_bool_if_parse:NNNww . 2169, 2178, 2178
	7698, 7716, 7718, 7723, 7724, 7738,	_bool_if_right_parentheses:wwn ..
	7739, 7742, 7743, 7814, 7903, 7911, 2162, 2167, 2174, 2175
	8120, 8127, 8139, 8140, 8155, 8156,	_bool_p:Nw 2199
	8359, 8363, 8368, 8404, 9017, 9030,	_box_resize:Nnn
	9031, 9037, 9044, 9057, 9063, 9532,	.. 14280, 14296, 14301, 14325, 14342

- _box_resize_common:N 14308, 14364, 14369, 14369
- _box_rotate:N 14166, 14174, 14178
- _box_rotate_quadrant_four: 14166, 14193, 14267
- _box_rotate_quadrant_one: 14166, 14187, 14234
- _box_rotate_quadrant_three: 14166, 14192, 14256
- _box_rotate_quadrant_two: 14166, 14188, 14245
- _box_rotate_x:nnN 14166, 14212, 14240, 14242, 14251, 14253, 14262, 14264, 14273, 14275
- _box_rotate_y:nnN 14166, 14223, 14236, 14238, 14247, 14249, 14258, 14260, 14269, 14271
- _box_show:NNnn . 6657, 6667, 6674, 6674
- _chk_if_exist_cs:N 24, 1216, 1216, 1225, 1886
- _chk_if_exist_cs:c 1047, 1052, 1057, 1062, 1216, 1224
- _chk_if_free_cs:N 25, 1193, 1193, 1203, 1215, 1230, 1278, 1620, 3461, 3476, 4187, 4411, 4501, 4597, 4603, 4608, 6563, 14007, 14030
- _chk_if_free_cs:c 1193, 1214
- _chk_if_free_msg:nn 7649, 7649, 7659, 7672
- _clist_concat:NNNN 5857, 5858, 5860, 5861
- _clist_count:n 6169, 6174, 6187
- _clist_count:w . 6169, 6183, 6188, 6192
- _clist_get:wN .. 5933, 5938, 5941, 5975
- _clist_if_empty_n:w 14580, 14582, 14587, 14590
- _clist_if_empty_n:wNw 14580, 14591, 14593
- _clist_if_in_return:nn 6067, 6069, 6074, 6076
- _clist_item:nnNn 14489, 14491, 14497, 14521
- _clist_item_N_loop:nw 14489, 14494, 14512, 14516
- _clist_item_n:nw .. 14519, 14524, 14527
- _clist_item_n_end:n 14519, 14535, 14543
- _clist_item_n_loop:nw 14519, 14528, 14529, 14532, 14537
- _clist_item_n_strip:w 14519, 14546, 14548
- _clist_map_function:Nw 6092, 6096, 6101, 6105, 6128
- _clist_map_function_n:Nn 6108, 6110, 6114, 6118
- _clist_map_unbrace:Nw 6108, 6117, 6121
- _clist_map_variable:Nnw 6141, 6146, 6157, 6162
- _clist_pop:NNN . 5944, 5945, 5947, 5948
- _clist_pop:wN 5944, 5961, 5967
- _clist_pop:wwNNN 5944, 5953, 5956, 5991
- _clist_pop_TF:NNN 5970, 5983, 5985, 5986
- _clist_put_left:NNNn 5907, 5908, 5910, 5911
- _clist_put_right:NNNn 5920, 5921, 5923, 5924
- _clist_remove_all: 6034, 6044, 6048, 6060
- _clist_remove_all:NNn 6034, 6035, 6037, 6038
- _clist_remove_all:w .. 6034, 6061, 6062
- _clist_remove_duplicates:NN 6018, 6019, 6021, 6022
- _clist_set_from_seq:NNNN 14549, 14550, 14552, 14553
- _clist_set_from_seq:w 14549, 14568, 14572
- _clist_tmp:w 5838, 5838, 6040, 6061, 6078, 6080
- _clist_trim_spaces:n 5880, 5880, 5902, 5904, 14578
- _clist_trim_spaces:nn 5880, 5883, 5887, 5893, 5898
- _clist_trim_spaces_generic:nn 5874, 5877, 5879
- _clist_trim_spaces_generic:nw 5874, 5874, 5882, 5892, 5897, 6110, 6118
- _clist_use:nwn 6194, 6208, 6219
- _clist_use:nwwwnwn 6194, 6205, 6207, 6216
- _clist_use:wnn . 6194, 6201, 6202, 6215
- _clist_wrap_item:n 14549, 14561, 14565
- _coffin_align:NnnNnnnnN 7186, 7223, 7241, 7249, 7249, 7340
- _coffin_calculate_intersection:Nnn 7078, 7078, 7251, 7254, 7540
- _coffin_calculate_intersection:nnnnnnnn 7078, 7084, 7093, 7486
- _coffin_calculate_intersection_aux:nnnnnnN 7105, 7120, 7129, 7136, 7162, 7171

_coffin_display_attach:Nnnnn
 7452, 7491, 7513, 7532, 7538
 _coffin_display_handles_aux:nnnn
 7452, 7519, 7524, 7530
 _coffin_display_handles_aux:nnnnnn
 7452, 7477, 7481
 _coffin_find_bounding_shift:
 14615, 14713, 14713
 _coffin_find_bounding_shift_aux:nn
 14713, 14717, 14719
 _coffin_find_corner_maxima:N
 14614, 14693, 14693
 _coffin_find_corner_maxima_aux:nn
 14693, 14700, 14702
 _coffin_get_pole:NnN
 6975, 6975, 7080, 7081, 7304,
 7305, 7308, 7309, 7466, 7467, 7470
 _coffin_gset_eq_structure:NN
 6992, 6999
 _coffin_if_exist:NT
 6827, 6827, 6838, 6858, 6875,
 6905, 6922, 6956, 7008, 7019, 7564
 _coffin_mark_handle_aux:nnnnNnn ..
 7397, 7435, 7440, 7444
 _coffin_offset_corner:Nnnnn
 7287, 7290, 7292
 _coffin_offset_corners:Nnn
 7206, 7207, 7213, 7214, 7287, 7287
 _coffin_offset_pole:Nnnnnnn
 7268, 7271, 7273
 _coffin_offset_poles:Nnn 7204, 7205,
 7210, 7211, 7233, 7234, 7268, 7268
 _coffin_reset_structure:N
 6841, 6867, 6888,
 6912, 6935, 6985, 6985, 7198, 7228
 _coffin_resize_common:Nnn
 14754, 14757, 14757, 14782
 _coffin_rotate_bounding:nnn
 14613, 14650, 14650
 _coffin_rotate_corner:Nnnn
 14608, 14650, 14656
 _coffin_rotate_pole:Nnnnn
 14610, 14662, 14662
 _coffin_rotate_vector:nnNN 14652,
 14658, 14664, 14665, 14674, 14674
 _coffin_scale_corner:Nnnn
 14760, 14793, 14793
 _coffin_scale_pole:Nnnnn
 14762, 14793, 14799
 _coffin_scale_vector:nnNN
 14786, 14786, 14795, 14801
 _coffin_set_bounding:N
 14611, 14638, 14638
 _coffin_set_eq_structure:NN
 6959, 6992, 6992
 _coffin_set_pole:Nnn . 7006, 7028, 7032
 _coffin_set_pole:Nnx
 6892, 6939, 7006, 7010, 7021, 7280,
 7317, 7321, 7329, 7333, 14667, 14802
 _coffin_shift_corner:Nnnn
 14633, 14724, 14724
 _coffin_shift_pole:Nnnnn
 14635, 14724, 14732
 _coffin_update_B:nnnnnnnnN
 7302, 7310, 7325
 _coffin_update_T:nnnnnnnnN
 7302, 7306, 7313
 _coffin_update_corners:N
 6869, 6890, 6914, 6937, 7033, 7033
 _coffin_update_poles:N . 6868, 6889,
 6913, 6936, 7044, 7044, 7201, 7232
 _coffin_update_vertical_poles:NNN
 7217, 7236, 7302, 7302
 _coffin_x_shift_corner:Nnnn
 14766, 14808, 14808
 _coffin_x_shift_pole:Nnnnn
 14768, 14808, 14815
 _cs_count_signature:N
 25, 1325, 1325, 1336
 _cs_count_signature:c 1325, 1335
 _cs_count_signature:nnN 1325, 1326, 1327
 _cs_generate_from_signature:NNn ..
 1354, 1358
 _cs_generate_from_signature:nnNNn
 1360, 1363
 _cs_generate_internal_variant:n ..
 2029, 2042, 2048
 _cs_generate_internal_variant:wwnNwnn
 2050, 2061
 _cs_generate_internal_variant:wwnw
 2042
 _cs_generate_internal_variant_loop:n
 2042, 2059, 2068, 2071
 _cs_generate_variant:N 1887, 1894, 1902
 _cs_generate_variant:Nnnw
 1930, 1932, 1932, 1950
 _cs_generate_variant:nnNN
 1890, 1923, 1923
 _cs_generate_variant:ww 1894, 1907, 1915

_cs_generate_variant:wwNN	_dim_case_aux:nnn	4307 , 4310 , 4312
..... 1939 , 2022 , 2022	_dim_case_aux:nw	4307 , 4313 , 4314 , 4318
_cs_generate_variant:wwNw	_dim_case_end:nw	4307 , 4317 , 4320
..... 1894 , 1916 , 1917	_dim_compare:w	4271 , 4273 , 4276
_cs_generate_variant_loop:nNwN ...	_dim_compare:wNN	4271 , 4279 , 4282 , 4294
..... 1940 , 1952 , 1952 , 1966	_dim_compare:w	4271
_cs_generate_variant_loop_end:nwwwNNnn	_dim_compare<:w	4271
..... 1942 , 1952 , 1973	_dim_compare>:w	4271
_cs_generate_variant_loop_invalid:NNwNNnn	_dim_compare_end:w	4279 , 4305
..... 1952 , 1959 , 1995	_dim_eval:w	89 , 4181 , 4182 , 4211 , 4222 , 4227 , 4234 , 4240 , 4241 , 4242 , 4248 , 4249 , 4250 , 4265 , 4268 , 4274 , 4295 , 4300 , 4378 , 4380 , 4384 , 4401 , 4568 , 6601 , 6603 , 6605 , 6614 , 6616 , 6618 , 6620 , 6694 , 6714 , 6727 , 6742 , 6770 , 11035 , 14405 , 14407 , 14448 , 14450 , 14620 , 15018
_cs_generate_variant_loop_long:wNNnn	_dim_eval_end:	89 , 4181 , 4183 , 4211 , 4222 , 4227 , 4234 , 4244 , 4252 , 4265 , 4268 , 4378 , 4380 , 4384 , 4569 , 6601 , 6603 , 6605 , 6614 , 6616 , 6618 , 6620 , 6694 , 6714 , 6727 , 6742 , 6770 , 14405 , 14407 , 14448 , 14450 , 14622 , 15018
..... 1945 , 1952 , 1982	_dim_maxmin:wwN	4231 , 4240 , 4248 , 4254
_cs_generate_variant_loop_same:w	_dim_ratio:n	4262 , 4263 , 4264
..... 1952 , 1955 , 1968	_dim_set_max:NNnn	4571 , 4573 , 4575 , 4577 , 4579 , 4580
_cs_generate_variant_same:N	_dim_strip_bp:n	89 , 4379 , 4379
..... 1971 , 2010 , 2010	_dim_strip_pt:n	89 , 4380 , 4381 , 4381 , 15008
_cs_get_function_name:N	_dim_strip_pt:w	4381 , 4384 , 4388
25 , 1095 , 1095	_dim_to_unit:n	15009 , 15013 , 15014 , 15017
_cs_get_function_signature:N	_driver_box_rotate_begin:	14201
..... 25 , 1095 , 1097	_driver_box_rotate_end:	14203
_cs_parm_from_arg_count:nnF	_driver_box_scale_begin:	14373
..... 925 , 1295 , 1295 , 1339	_driver_box_scale_end:	14375
_cs_parm_from_arg_count_test:nnF	_driver_box_use_clip:N	14399
..... 1295 , 1297 , 1316	_driver_color_ensure_current:	7616 , 7622 , 7625
_cs_show:www	_exp_arg_last_unbraced:nn	1805 , 1805 , 1808 , 1812 , 1815 , 1820
_cs_split_function:NN	_exp_arg_next:Nnn	1630 , 1631 , 1637
.. 25 , 907 , 920 , 1007 , 1008 , 1076 , 1082 , 1096 , 1098 , 1326 , 1360 , 1888	_exp_arg_next:nnn	1630 , 1630 , 1639 , 1642 , 1650 , 1654 , 1660
_cs_split_function_auxi:w	_exp_eval_error_msg:w	1664 , 1668 , 1677
..... 1076 , 1085 , 1090	_exp_eval_register:N	1655 , 1664 , 1664 , 1676 , 1710 , 1728 , 1746 , 1747 , 1754 , 1816 , 1829 , 1841 , 1847 , 1856 , 1876
_cs_split_function_auxii:w		
..... 1076 , 1091 , 1092		
_cs_tmp:w		
25 , 1226 , 1234 , 1235 , 1236 , 1237 , 1238 , 1239 , 1240 , 1241 , 1242 , 1244 , 1245 , 1246 , 1247 , 1248 , 1249 , 1250 , 1251 , 1252 , 1253 , 1254 , 1255 , 1256 , 1257 , 1258 , 1259 , 1260 , 1261 , 1262 , 1263 , 1264 , 1265 , 1266 , 1267 , 1350 , 1375 , 1376 , 1377 , 1378 , 1379 , 1380 , 1381 , 1382 , 1383 , 1384 , 1385 , 1386 , 1387 , 1388 , 1389 , 1390 , 1391 , 1392 , 1393 , 1394 , 1395 , 1396 , 1397 , 1398 , 1399 , 1407 , 1408 , 1409 , 1410 , 1411 , 1412 , 1413 , 1414 , 1415 , 1416 , 1417 , 1418 , 1419 , 1420 , 1421 , 1422 , 1423 , 1424 , 1425 , 1426 , 1427 , 1428 , 1429 , 1430 , 1905 , 1920 , 2030 , 2052 , 4465 , 4475		
_cs_to_str:N		
... 1067 , 1071 , 1073 , 1074		
_cs_to_str:w		
..... 1067 , 1070 , 1074		
_dim_abs:N		
..... 4231 , 4233 , 4236		

```

\__expl_eval_register:c .. 1661, 1664,
    1675, 1705, 1722, 1821, 1831, 1881
\__expl_last_two_unbraced:noN .....
    ..... 1865, 1866, 1867
\__expl_package_check: .....
    ..... 7, 219, 764, 1628,
    2087, 2485, 2609, 3381, 4179, 4593,
    5393, 5834, 6278, 6558, 6780, 7606,
    7639, 8443, 9131, 9717, 13973, 14151
\__expl_primitive:NN 310, 310, 317, 318,
    319, 320, 321, 322, 323, 324, 325,
    326, 327, 328, 329, 330, 331, 332,
    333, 334, 335, 336, 337, 338, 339,
    340, 341, 342, 343, 344, 345, 346,
    347, 348, 349, 350, 351, 352, 353,
    354, 355, 356, 357, 358, 359, 360,
    361, 362, 363, 364, 365, 366, 367,
    368, 369, 370, 371, 372, 373, 374,
    375, 376, 377, 378, 379, 380, 381,
    382, 383, 384, 385, 386, 387, 388,
    389, 390, 391, 392, 393, 394, 395,
    396, 397, 398, 399, 400, 401, 402,
    403, 404, 405, 406, 407, 408, 409,
    410, 411, 412, 413, 414, 415, 416,
    417, 418, 419, 420, 421, 422, 423,
    424, 425, 426, 427, 428, 429, 430,
    431, 432, 433, 434, 435, 436, 437,
    438, 439, 440, 441, 442, 443, 444,
    445, 446, 447, 448, 449, 450, 451,
    452, 453, 454, 455, 456, 457, 458,
    459, 460, 461, 462, 463, 464, 465,
    466, 467, 468, 469, 470, 471, 472,
    473, 474, 475, 476, 477, 478, 479,
    480, 481, 482, 483, 484, 485, 486,
    487, 488, 489, 490, 491, 492, 493,
    494, 495, 496, 497, 498, 499, 500,
    501, 502, 503, 504, 505, 506, 507,
    508, 509, 510, 511, 512, 513, 514,
    515, 516, 517, 518, 519, 520, 521,
    522, 523, 524, 525, 526, 527, 528,
    529, 530, 531, 532, 533, 534, 535,
    536, 537, 538, 539, 540, 541, 542,
    543, 544, 545, 546, 547, 548, 549,
    550, 551, 552, 553, 554, 555, 556,
    557, 558, 559, 560, 561, 562, 563,
    564, 565, 566, 567, 568, 569, 570,
    571, 572, 573, 574, 575, 576, 577,
    578, 579, 580, 581, 582, 583, 584,
    585, 586, 587, 588, 589, 590, 591,
    592, 593, 594, 595, 596, 597, 598,
    599, 600, 601, 602, 603, 604, 605,
    606, 607, 608, 609, 610, 611, 612,
    613, 614, 615, 616, 617, 618, 619,
    620, 621, 622, 623, 624, 625, 626,
    627, 628, 629, 630, 631, 632, 633,
    634, 635, 636, 637, 638, 639, 640,
    641, 642, 643, 644, 645, 646, 647,
    648, 649, 650, 651, 652, 653, 654,
    655, 656, 657, 658, 659, 660, 661,
    662, 663, 664, 665, 666, 667, 668,
    669, 670, 671, 672, 673, 674, 675,
    676, 677, 678, 679, 680, 681, 682,
    683, 684, 685, 686, 687, 688, 689,
    690, 691, 692, 693, 694, 695, 696,
    697, 698, 699, 700, 701, 702, 703,
    704, 705, 706, 707, 708, 709, 710,
    711, 712, 713, 714, 715, 716, 717,
    718, 719, 720, 721, 722, 723, 724,
    725, 726, 727, 728, 729, 730, 731,
    732, 733, 734, 735, 736, 737, 738, 739
\__expl_status_pop:w ..... 207
\__file_add_path:nN .... 9182, 9183, 9184
\__file_add_path_search:nN .....
    ..... 9182, 9188, 9192
\__file_input:V ..... 9233
\__file_input:n ..... 9235, 9241
\__file_input:n\__file_input:V .. 9225
\__file_input_aux:n 9225, 9238, 9242, 9258
\__file_input_aux:o ..... 9225, 9239
\__file_name_sanitize:nn .....
    ..... 167, 9162, 9162, 9183,
    9230, 9260, 9268, 9311, 9321, 9426
\__file_path_include:n . 9259, 9260, 9261
\__fp_ ..... 11613, 11620, 13385
\__fp__o:ww ..... 11610
\__fp_*o:ww ..... 11993
\__fp_+o:ww ..... 11711
\__fp_-o:w ..... 12255
\__fp_-o:ww ..... 11706
\__fp_/o:ww ..... 12103
\__fp^o:ww ..... 13024
\__fp_o:w ..... 11602
\__fp_abs:NNN ..... 13872,
    13872, 13873, 13876, 13882, 13886
\__fp_abs_o:w . 12263, 12263, 13872, 13873
\__fp_add:NNNn ..... 13827,
    13827, 13828, 13829, 13830, 13831
\__fp_add_big_i_o:wNww .....
    ..... 11782, 11789, 11789, 12852

```

```

\__fp_add_big_ii_o:wNww ..... 11785, 11789, 11797
\__fp_add_inf_o:Nww . 11726, 11746, 11746
\__fp_add_normal_o:Nww 11725, 11766, 11766
\__fp_add_npos_o:NwwNww ..... 11769, 11775, 11775
\__fp_add_return_ii_o:Nww ..... 11728, 11734, 11734, 11739
\__fp_add_significand_carry_o:wwwNN ..... 11822, 11837, 11837
\__fp_add_significand_no_carry_o:wwwNN ..... 11824, 11827, 11827
\__fp_add_significand_o:NnnwnnnN .. 11792, 11800, 11805, 11805
\__fp_add_significand_pack:NNNNNN .. 11805, 11809, 11812
\__fp_add_significand_test_o:N .... 11805, 11807, 11819
\__fp_add_zeros_o:Nww 11724, 11736, 11736
\__fp_and_return:wNw 11610, 11616, 11622
\__fp_array_count:n ... 9959, 9959, 11089
\__fp_array_count_loop:Nw ..... 9959, 9962, 9966, 9967
\__fp_array_to_clist:n 10261, 13774, 13774
\__fp_array_to_clist_loop:Nw ..... 13774, 13781, 13786, 13796
\__fp_assign_to:nNNNn . 13906, 13913, 13914, 13915, 13916, 13917, 13918
\__fp_assign_to_i:wNNNn ..... 13906, 13920, 13923
\__fp_assign_to_ii:NnNNN ..... 13906, 13925, 13929
\__fp_basics_pack_high:NNNNNw ..... 11680, 11687, 11830, 11981, 12082, 12094, 12234, 12472
\__fp_basics_pack_high_carry:w .... 11680, 11690, 11694
\__fp_basics_pack_low:NNNNNw ..... 11680, 11680, 11832, 11983, 12084, 12096, 12236, 12474
\__fp_basics_pack_weird_high:NNNNNNNNw ..... 11696, 11704, 11841, 12245
\__fp_basics_pack_weird_low:NNNNw .. 11696, 11696, 11843, 12247
\__fp_case_return:nw .... 9914, 9914, 9928, 9930, 10321, 12785, 13575, 13625, 13693, 13695, 13696, 13739
\__fp_case_return_i_o:ww ..... 9921, 9921, 11727, 11741, 11750, 12026
\__fp_case_return_ii_o:ww ..... 9921, 9923, 12027, 13073, 13091
\__fp_case_return_o:Nw ..... 9915, 9915, 12820, 12825, 12828, 13028, 13033, 13055, 13064, 13277, 13307
\__fp_case_return_o:Nww ..... 9919, 9919, 12028, 12029, 12032, 12033, 13075, 13084, 13087
\__fp_case_return_same_o:w ..... 9917, 9917, 12606, 12832, 13052, 13262, 13270, 13285, 13300, 13315, 13322, 13330, 13345
\__fp_case_use:nw . 9913, 9913, 11752, 12024, 12025, 12030, 12031, 12111, 12114, 12599, 12602, 13058, 13264, 13269, 13279, 13284, 13294, 13299, 13309, 13314, 13324, 13329, 13339, 13344, 13578, 13589, 13628, 13638
\__fp_cfs_round_loop:N ..... 10795, 10795, 10800, 10833, 10857, 10885
\__fp_cfs_round_up:N 10803, 10811, 10815
\__fp_chk:w ..... 9728, 9729, 9732, 9741, 9742, 9743, 9744, 9745, 9747, 9748, 9751, 9757, 9761, 9781, 9784, 9785, 9795, 9805, 9818, 9837, 9925, 10101, 10106, 10264, 10273, 10275, 11017, 11428, 11453, 11454, 11562, 11570, 11573, 11584, 11585, 11593, 11594, 11602, 11613, 11616, 11628, 11653, 11712, 11731, 11732, 11734, 11735, 11736, 11744, 11747, 11763, 11764, 11766, 11775, 11851, 12002, 12036, 12037, 12040, 12119, 12255, 12259, 12263, 12267, 12596, 12608, 12610, 12817, 12834, 12836, 13025, 13044, 13046, 13047, 13049, 13061, 13066, 13068, 13093, 13094, 13096, 13112, 13197, 13210, 13212, 13259, 13272, 13274, 13287, 13289, 13302, 13304, 13317, 13319, 13332, 13334, 13347, 13359, 13378, 13387, 13571, 13596, 13599, 13621, 13645, 13648, 13689, 13712, 13762, 13763, 13863, 13870, 13923, 13932
\__fp_compare:wNNNNw ..... 11226
\__fp_compare_aux:wn 11436, 11439, 11447
\__fp_compare_back:ww ..... 11313, 11449, 11452, 11452, 11583
\__fp_compare_nan:w ..... 11452, 11457, 11458, 11479

```

_fp_compare_npos:nwnw 11463, 11480, 11480, 11853
 _fp_compare_return:w 11423, 11425, 11428
 _fp_compare_significand:nnnnnnnn 11480, 11483, 11488
 _fp_cos_o:w 13274, 13274, 13916
 _fp_cot_o:w 13334, 13334
 _fp_cot_zero_o:Nnw 13292, 13334, 13337, 13349
 _fp_csc_o:w 13289, 13289
 _fp_decimate:nNnnnn 9867, 9867, 9941, 10277, 11791, 11799, 11878, 12868, 12872, 13654
 _fp_decimate_:Nnnnn 9879, 9879
 _fp_decimate_pack:nnnnnnnnnw 9890, 9909, 9909
 _fp_decimate_pack:nnnnnnnw 9910, 9911
 _fp_decimate_tiny:Nnnnn 9879, 9881
 _fp_div_npos_o:Nww 12108, 12118, 12118
 _fp_div_significand_calc:wwnnnnnn 12135, 12144, 12144, 12190, 12678, 12685
 _fp_div_significand_calc_i:wwnnnnnn 12144, 12147, 12152
 _fp_div_significand_calc_ii:wwnnnnnn 12144, 12149, 12169
 _fp_div_significand_i_o:wnnw 12125, 12131, 12131
 _fp_div_significand_ii:wwn 12139, 12140, 12141, 12186, 12186
 _fp_div_significand_iii:wwnnnnnn 12142, 12193, 12193
 _fp_div_significand_iv:wwnnnnnn 12196, 12201, 12201
 _fp_div_significand_large_o:wwwNNNNwN 12227, 12241, 12241
 _fp_div_significand_pack:NNN 12188, 12221, 12221, 12665, 12683, 12691
 _fp_div_significand_small_o:wwwNNNNwN 12225, 12231, 12231
 _fp_div_significand_test_o:w 12133, 12222, 12222
 _fp_div_significand_v:NN 12206, 12208, 12211
 _fp_div_significand_v:NNw 12201
 _fp_div_significand_vi:Nw 12201, 12204, 12212
 _fp_division_by_zero_o:NNww 10073, 10113, 10117, 12112, 12115, 13060
 _fp_division_by_zero_o:Nnw 10065, 10113, 10116, 12603, 13353, 13355
 _fp_error:nffn 10042, 10076, 10104, 10126
 _fp_error:nnfn 10034, 10051, 10068, 10126
 _fp_error:nnnn 10126, 10126, 10128
 _fp_exp_Taylor:Nnnwn 12869, 12885, 12885, 13021
 _fp_exp_Taylor_break:Nww 12885, 12899, 12910
 _fp_exp_Taylor_ii:ww 12891, 12894
 _fp_exp_Taylor_loop:www 12885, 12895, 12896, 12905
 _fp_exp_after_array_f:w 9838, 9838, 9841, 11136, 11632, 11643, 11666, 11674
 _fp_exp_after_f:nw 9785, 9805, 10429, 10925, 11016
 _fp_exp_after_normal:Nwwwww 9827, 9835
 _fp_exp_after_normal:nNNw 9788, 9798, 9808, 9825, 9825
 _fp_exp_after_o:nw 9785, 9795
 _fp_exp_after_o:w 9785, 9785, 9918, 9922, 9924, 10271, 10315, 10333, 11601, 11618, 11735, 12257, 12265, 13209, 13378
 _fp_exp_after_special:nNNw 9790, 9800, 9810, 9815, 9815
 _fp_exp_after_stop_f:nw 9838, 9844
 _fp_exp_large:w 12912, 12921, 12926, 12927, 12928, 12929, 12930, 12931, 12932, 12933, 12934, 12942, 12943, 12944, 12945, 12946, 12947, 12948, 12949, 12950, 12958, 12959, 12960, 12961, 12962, 12963, 12964, 12965, 12966, 12974, 12975, 12976, 12977, 12978, 12979, 12980, 12981, 12982, 12990, 12991, 12992, 12993, 12994, 12995, 12996, 12997, 12998, 13006, 13007, 13008, 13009, 13010, 13011, 13012, 13013, 13014
 _fp_exp_large:wN 12912, 13001, 13003
 _fp_exp_large_after:wwn 12912, 13017, 13019
 _fp_exp_large_i:wN 12912, 12985, 12987
 _fp_exp_large_ii:wN 12912, 12969, 12971
 _fp_exp_large_iii:wN 12912, 12953, 12955
 _fp_exp_large_iv:wN 12912, 12937, 12939
 _fp_exp_large_v:wN 12912, 12923, 13196
 _fp_exp_normal:w 12822, 12836, 12836
 _fp_exp_o:w 12817, 12817, 13913

_fp_exp_overflow: 12858, 12883
 _fp_exp_pos:NNwnw . 12839, 12841, 12844
 _fp_exp_pos:Nnnwn 12836
 _fp_exp_pos_large:NnnNwn
 12873, 12912, 12912
 _fp_expand:n 9968, 9968, 13778
 _fp_expand_loop:nwnN
 9968, 9970, 9972, 9975
 _fp_exponent:w 9761, 9761, 13959, 13964
 _fp_fixed_add:Nnnnnwnn
 12314, 12314, 12315, 12316
 _fp_fixed_add:nnNnnwn
 12314, 12322, 12324
 _fp_fixed_add:wnn 12314,
 12314, 12759, 12767, 12778, 12796
 _fp_fixed_add_after:NNNNwn
 12314, 12318, 12332
 _fp_fixed_add_one:wN
 .. 12277, 12277, 12581, 12902, 12911
 _fp_fixed_add_pack:NNNNwn
 12314, 12320, 12327, 12330
 _fp_fixed_continue:wn
 12276, 12276, 12925, 12941,
 12957, 12973, 12989, 13005, 13172
 _fp_fixed_div_int:wnN
 12283, 12288, 12296, 12308
 _fp_fixed_div_int:wwN
 12283, 12283, 12758, 12901
 _fp_fixed_div_int_after:Nw
 12283, 12285, 12313
 _fp_fixed_div_int_auxi:wnn
 12283, 12289,
 12290, 12291, 12292, 12293, 12303
 _fp_fixed_div_int_auxii:wnn
 12283, 12294, 12311
 _fp_fixed_div_int_pack:Nw
 12283, 12306, 12312
 _fp_fixed_div_to_float:ww
 12477, 12487, 13543
 _fp_fixed_dtf_approx:NNNNw
 12551, 12556
 _fp_fixed_dtf_approx:n
 12484, 12498, 12542
 _fp_fixed_dtf_approx:wnn 12544, 12548
 _fp_fixed_dtf_epsilon:NNNNw
 12569, 12576
 _fp_fixed_dtf_epsilon:wN 12560, 12565
 _fp_fixed_dtf_epsilon_pack:NNNNw
 12571, 12574
 _fp_fixed_dtf_no_zero:Nwn
 .. 12483, 12492, 12497, 12501, 12503
 _fp_fixed_dtf_zeros:NN
 12508, 12513, 12520, 12523
 _fp_fixed_dtf_zeros:wNnnnnn
 12481, 12490, 12495, 12502
 _fp_fixed_dtf_zeros_auxi:ww
 12530, 12533
 _fp_fixed_dtf_zeros_auxii:ww
 12538, 12541
 _fp_fixed_dtf_zeros_end:wNw
 12518, 12522
 _fp_fixed_inv_to_float:wN . 12477,
 12477, 12841, 13108, 13297, 13312
 _fp_fixed_mul:nnnnnnwn
 12334, 12354, 12356
 _fp_fixed_mul:wnn
 12334, 12334, 12559,
 12561, 12562, 12578, 12582, 12760,
 12770, 12810, 12903, 12922, 13022,
 13118, 13453, 13496, 13507, 13527
 _fp_fixed_mul_add:Nwnnnwnnn
 .. 12368, 12378, 12388, 12392, 12392
 _fp_fixed_mul_add:nnnnwnnn
 12403, 12405, 12405
 _fp_fixed_mul_add:nnnnwnnwN
 12410, 12416, 12416
 _fp_fixed_mul_add:wwwn .. 12362, 12362
 _fp_fixed_mul_after:wn 12282, 12282,
 12336, 12364, 12374, 12384, 13135
 _fp_fixed_mul_one_minus_mul:wwn 12362
 _fp_fixed_mul_sub_back:wwwn
 12362, 12372, 13475, 13477,
 13478, 13479, 13480, 13481, 13482,
 13483, 13484, 13487, 13489, 13490,
 13491, 13492, 13493, 13494, 13495,
 13521, 13523, 13524, 13525, 13526,
 13529, 13531, 13532, 13533, 13534
 _fp_fixed_one_minus_mul:wwn ... 12382
 _fp_fixed_sub:wnn ... 12314, 12315,
 12776, 12792, 12804, 13436, 13449
 _fp_fixed_to_float:Nw
 12421, 12421, 12785
 _fp_fixed_to_float:wN
 12421, 12421, 12422, 12563, 12805,
 12815, 12839, 13104, 13267, 13282
 _fp_fixed_to_float_pack:ww 12453, 12463
 _fp_fixed_to_float_round_up:wnnnnw
 12466, 12470
 _fp_fixed_to_float_zero:w 12449, 12458

_fp_fixed_to_loop:N 12426, 12436, 12440
 _fp_fixed_to_loop_end:w . 12442, 12446
 _fp_from_dim:Nw
 13731, 13742, 13746, 13752
 _fp_from_dim:wNNnnnnnn
 13731, 13754, 13757
 _fp_from_dim:wnnnnwN 13731, 13758, 13759
 _fp_from_dim_test:N 13731, 13733, 13736
 _fp_if_undefined:w 13862, 13863
 _fp_if_zero:w 13869, 13870
 _fp_inf_fp:N 9747, 9748, 10089
 _fp_infix_compare:N . 11226, 11228,
 11233, 11238, 11248, 11256, 11264
 _fp_invalid_operation:nnw
 10031, 10113, 10113,
 10125, 13580, 13591, 13630, 13640
 _fp_invalid_operation_o:Nww
 10039, 10113, 10114,
 11755, 11758, 12030, 12031, 13203
 _fp_invalid_operation_o:nw
 10124, 10124, 12599, 13269,
 13284, 13299, 13314, 13329, 13344
 _fp_invalid_operation_tl_o:nf
 10048, 10113, 10115, 10260
 _fp_ln_Taylor:wwNw 12750, 12751, 12751
 _fp_ln_Taylor_break:w .. 12756, 12767
 _fp_ln_Taylor_loop:www
 12752, 12753, 12762
 _fp_ln_c:NwNw 12742, 12773, 12773
 _fp_ln_div_after:Nw 12646, 12694
 _fp_ln_div_i:w 12667, 12676
 _fp_ln_div_ii:wwn
 12670, 12671, 12672, 12673, 12681
 _fp_ln_div_vi:wwn 12674, 12689
 _fp_ln_exponent:wn 12622, 12782, 12782
 _fp_ln_exponent_one:ww .. 12787, 12801
 _fp_ln_exponent_small:NNww
 12790, 12794, 12807
 _fp_ln_npos_o:w .. 12608, 12610, 12610
 _fp_ln_o:w 12596, 12596, 13914
 _fp_ln_significand:NNNNnnN
 12621, 12624, 12624, 13116
 _fp_ln_square_t_after:w . 12718, 12749
 _fp_ln_square_t_pack:NNNNw
 12720, 12722, 12724, 12726, 12747
 _fp_ln_t_large:NNw 12699, 12706, 12716
 _fp_ln_t_small:Nw 12697, 12704
 _fp_ln_twice_t_after:w .. 12730, 12746
 _fp_ln_twice_t_pack:Nw 12732,
 12734, 12736, 12738, 12740, 12745
 _fp_ln_x_ii:wnnnn . 12626, 12644, 12644
 _fp_ln_x_iii:NNNNw 12653, 12657
 _fp_ln_x_iii_var:NNNNw . 12651, 12658
 _fp_ln_x_iv:wnnnnnnnn .. 12649, 12663
 _fp_max_fp:N 9749, 9755
 _fp_max_o:w 11063, 11556, 11556
 _fp_min_fp:N 9749, 9749
 _fp_min_o:w 11064, 11556, 11564
 _fp_minmax_auxi:ww
 11577, 11589, 11596, 11596
 _fp_minmax_auxii:ww
 11579, 11587, 11596, 11598
 _fp_minmax_break_o:w
 11562, 11570, 11600, 11600
 _fp_minmax_loop:Nww
 11558, 11566, 11572, 11572, 11592
 _fp_mul:NNNn .. 13892, 13892, 13893,
 13894, 13895, 13896, 13897, 13898
 _fp_mul_cases_o:NnNww
 11995, 12001, 12105
 _fp_mul_cases_o:nNnnw 12001
 _fp_mul_npos_o:Nww
 11998, 12039, 12039, 13761
 _fp_mul_significand 12578
 _fp_mul_significand_drop:NNNNw ..
 12048,
 12057, 12059, 12061, 12063, 12067
 _fp_mul_significand_keep:NNNNw ..
 12048, 12053, 12055, 12069
 _fp_mul_significand_large_f:NwwNNN
 12076, 12080, 12080
 _fp_mul_significand_o:nnnnNnnn ..
 12046, 12048, 12048
 _fp_mul_significand_small_f:NNwwwN
 12074, 12091, 12091
 _fp_mul_significand_test_f:NNN ...
 12050, 12071, 12071
 _fp_neg_sign:N 9770, 9770, 11709
 _fp_overflow:w 9777, 10113, 10118
 _fp_pack:NNNNw
 9845, 9848, 13415, 13417, 13419
 _fp_pack:NNNNwn 9845, 9849,
 12338, 12341, 12344, 12347, 12350,
 13137, 13140, 13143, 13146, 13149
 _fp_pack_Bigg:NNNNNw
 9858, 9861, 12158,
 12161, 12164, 12175, 12178, 12181
 _fp_pack_big:NNNNNw 9851, 9854

_fp_pack_big:NNNNNNwn 10619, 10639, 10669, 10697, 10699,
 [9851](#), 9856, 12366, 12376,
 12386, 12395, 12398, 12401, 12408
 _fp_pack_eight:wNNNNNNNN 10712, 10728, 10748, 10759, 10809,
 [9865](#), 9865, 11974 10820, 10849, 10858, 10890, 10903,
 _fp_pack_twice_four:wNNNNNNNN 10969, 11049, 11060, 11082, 11085,
 [9863](#), 9863, 10308, 11086, 11113, 11130, 11139, 11165,
 10309, 11917, 11918, 12451, 12452, 11206, 11218, 11244, 11291, 11303,
 12535, 12536, 12537, 12888, 12889, 11330, 11346, 11362, 11389, 11636
 12890, 13392, 13393, 13394, 13754
 _fp_parse:n _fp_parse_exponent:N
 [10339](#), 10418, [10897](#), 10897, 11426, [10415](#), 10580, 10702,
 11440, 11450, 13564, 13619, 13687, [10704](#), 10704, 10863, 10872, 10895
 13724, 13769, 13771, 13773, 13805, _fp_parse_exponent:Nw
 13807, 13809, 13832, 13899, 13921 [10604](#), 10617, 10666,
 _fp_parse_after:ww [10897](#), 10900, 10906 10694, [10699](#), 10699, 10847, 10888
 _fp_parse_apply_binary:NwNwN _fp_parse_exponent_aux:N
 [10973](#), 10973, 11161 [10704](#), 10707, 10714
 _fp_parse_apply_compare:NwNNNNwN . _fp_parse_exponent_body:N
 11297, 11305 [10730](#), [10734](#), 10734
 _fp_parse_apply_round:NNwN [11078](#), [11087](#) _fp_parse_exponent_digits:N
 _fp_parse_apply_unary:NNwN 10738, [10750](#), 10750, 10754
 [10988](#), 10993, 11044, 11104 _fp_parse_exponent_keep:N 10761
 _fp_parse_apply_unary_array:NNwN . _fp_parse_exponent_keep:NTF
 [10988](#), 10988, 11056 [10741](#), [10761](#)
 _fp_parse_compare:NNNNNNw _fp_parse_exponent_sign:N
 [11226](#), 11259, 11267, 11284 [10720](#), [10724](#), 10724, 10727
 _fp_parse_compare_end:NNNN _fp_parse_infix:NN
 [11226](#), 11279, 11293 [10424](#), [10493](#), 10514, 10925, 10930,
 _fp_parse_compare_expand:NNNNNw .. 11003, 11016, 11036, 11137, 11360
 [11226](#), _fp_parse_infix_
 11275, 11276, 11277, 11278, 11282 [10963](#), 11134, 11185, 11189, 11201,
 _fp_parse_digits:N 10375, 10376 11204, 11213, 11216, 11323, 11334,
 _fp_parse_digits_i:N ... [10357](#), 10374 11356, 11366, [11372](#), 11374, 11379
 _fp_parse_digits_ii:N .. [10357](#), 10373 _fp_parse_infix:Nw [11198](#)
 _fp_parse_digits_iii:N .. [10357](#), 10372 _fp_parse_infix_):N [11354](#)
 _fp_parse_digits_iv:N .. [10357](#), 10371 _fp_parse_infix_*:N [11183](#)
 _fp_parse_digits_v:N ... [10357](#), 10370 _fp_parse_infix_+:N [11173](#)
 _fp_parse_digits_vi:N _fp_parse_infix_-:N [11173](#)
 [10357](#), 10369, 10601, 10649 _fp_parse_infix_/N [11173](#)
 _fp_parse_digits_vii:N _fp_parse_infix_:N
 [10357](#), 10588, 10638 [11321](#), 11337, 11350, 11625
 _fp_parse_exp_after_?_f:nw [10420](#) _fp_parse_infix_:N [11226](#)
 _fp_parse_exp_after_f:nw [10420](#), 10429 _fp_parse_infix_<:N [11226](#)
 _fp_parse_exp_after_mark_f:nw _fp_parse_infix_>:N [11226](#)
 [10420](#), 10430 _fp_parse_infix_?:N [11321](#)
 _fp_parse_expand:w _fp_parse_infix_\meta{operation}:N
 [10339](#), 10339, 10341, [10339](#)
 10366, 10425, 10460, 10478, 10543, _fp_parse_infix_^:N [11183](#)
 10545, 10565, 10567, 10589, 10606, _fp_parse_infix_after_operand:NwN
 [10406](#), 10524, [10923](#), 10923, 11149
 _fp_parse_infix_and:N .. [11173](#), 11220
 _fp_parse_infix_check:NNN 10949, 10959

_fp_parse_infix_comma:w .	11382, 11392	_fp_parse_pack_trailing:NNNNNNww .	10594,
_fp_parse_infix_comma_gobble:w	11385, 11394	10621, 10621, 10663, 10674, 10681	
_fp_parse_infix_end:N	10908, 10937, 11370, 11371	_fp_parse_prefix(:Nw	11118
_fp_parse_infix_excl_aux:NN	11226, 11243, 11246	_fp_parse_prefix+:Nw	11099
_fp_parse_infix_excl_error:	11226, 11251, 11260	_fp_parse_prefix -:Nw	11100
_fp_parse_infix_juxtapose:N	10940, 10947, 11175	_fp_parse_prefix.:Nw	11147
_fp_parse_infix_mul:N	11173, 11192	_fp_parse_prefix:Nw	11100
_fp_parse_infix_or:N	11173, 11208	_fp_parse_return_semicolon:w	10340, 10340, 10364,
_fp_parse_large:N	10550, 10634, 10634	10710, 10742, 10757, 10807, 10818	
_fp_parse_large_leading:wwNN	10636, 10641, 10641	_fp_parse_round:Nw	11068, 11071, 11074, 11084
_fp_parse_large_round:NN	10677, 10822, 10822	_fp_parse_small:N	10571, 10582, 10582
_fp_parse_large_round_after:wNN	10831, 10851	_fp_parse_small_leading:wwNN	10586, 10591, 10591, 10653
_fp_parse_large_round_after_aux:wN	10854, 10867	_fp_parse_small_round:NN	10613, 10843, 10874, 10874
_fp_parse_large_round_dot_test:NNw	10835, 10840	_fp_parse_small_round_after:wN	10883, 10892
_fp_parse_large_trailing:wwNN	10647, 10671, 10671	_fp_parse_small_trailing:wwNN	10599, 10608, 10608, 10685
_fp_parse_letters:NN 10450, 10463, 10475		_fp_parse_stop_until:N	10922,
_fp_parse_lparen_after:NwN 11122, 11132		11143, 11168, 11263, 11333, 11349,	
_fp_parse_operand:Nw	10339,	11365, 11371, 11378, 11393, 11397	
10377, 10377, 10519, 10917, 11099		_fp_parse_strim_end:w	10556, 10563, 10567
_fp_parse_operand_digit:NN	10390, 10522, 10522	_fp_parse_strim_zeros:N	10537, 10556, 10556, 10560, 11153
_fp_parse_operand_other:NN	10393, 10443, 10443	_fp_parse_trim_end:w 10530, 10540, 10545	
_fp_parse_operand_other_prefix_aux:NNN	10454, 10498	_fp_parse_trim_zeros:N	10528, 10530, 10530, 10533
_fp_parse_operand_other_prefix_unknown:NNN	10501, 10506	_fp_parse_unary_type:N	10998, 11046, 11106
_fp_parse_operand_other_word_aux:Nw	10447, 10485	_fp_parse_until:Nw	10339, 10902,
_fp_parse_operand_register:NN	10385, 10398, 10404	10911, 10911, 11048, 11060, 11082,	
_fp_parse_operand_register_aux:www	10398, 10409, 10417	11109, 11111, 11126, 11128, 11164,	
_fp_parse_operand_relax:NN	10382, 10420, 10420	11303, 11329, 11345, 11388, 11635	
_fp_parse_pack_carry:w	10621, 10629, 10632	_fp_parse_until_test:NwN	10911, 10914, 10921, 10923,
_fp_parse_pack_leading:NNNNNww	10584, 10621, 10626, 10644	10975, 11307, 11640, 11663, 11671	
		_fp_parse_word_abs:N	11039
		_fp_parse_word_bp:N	11000
		_fp_parse_word_cc:N	11000
		_fp_parse_word_cm:N	11000
		_fp_parse_word_cos:N	11039
		_fp_parse_word_cot:N	11039
		_fp_parse_word_csc:N	11039
		_fp_parse_word_dd:N	11000
		_fp_parse_word_deg:N	11000

_fp_parse_word_em:N	11000	_fp_round:Nwn	10256 ,
_fp_parse_word_ex:N	11000		10258 , 10264 , 11090 , 13729 , 13947
_fp_parse_word_exp:N	11039	_fp_round:Nww	10256 , 10256 , 11091
_fp_parse_word_false:N	11000	_fp_round_digit:Nw	
_fp_parse_word_in:N	11000		9889 , 9909 , 10218 , 10218 ,
_fp_parse_word_inf:N	11000		11848 , 11990 , 12089 , 12101 , 12252
_fp_parse_word_ln:N	11039	_fp_round_neg:NNN	
_fp_parse_word_max:N ...	11052 , 11063		10229 , 10255 , 11952 , 11967 , 11985
_fp_parse_word_min:N ...	11052 , 11064	_fp_round_normal:NNwNnn	
_fp_parse_word_mm:N	11000		10256 , 10282 , 10302
_fp_parse_word_nan:N	11000	_fp_round_normal:NnnwNNnn	
_fp_parse_word_nc:N	11000		10256 , 10278 , 10280
_fp_parse_word_nd:N	11000	_fp_round_normal:NwNNnw	
_fp_parse_word_pc:N	11000		10256 , 10267 , 10275
_fp_parse_word_pi:N	11000	_fp_round_normal_end:wwNnn	
_fp_parse_word_pt:N	11000		10256 , 10310 , 10313
_fp_parse_word_round:N ..	11065 , 11065	_fp_round_pack:Nw .	10256 , 10286 , 10300
_fp_parse_word_sec:N	11039	_fp_round_places:NNn	13938 ,
_fp_parse_word_sin:N	11039		13939 , 13941 , 13942 , 13958 , 13963
_fp_parse_word_sp:N	11000	_fp_round_return_one:	
_fp_parse_word_tan:N	11039		10167 , 10173 , 10183 ,
_fp_parse_word_true:N	11000		10191 , 10195 , 10233 , 10241 , 10249
_fp_parse_zero:		_fp_round_s:NNnw	
	10552 , 10573 , 10577 , 10577		10202 , 10202 , 10826 , 10878
_fp_pow_B:wwN	13119 , 13154	_fp_round_special:NwwNnn	
_fp_pow_C_neg:w	13157 , 13174		10256 , 10305 , 10318
_fp_pow_C_overflow:w	13162 , 13169 , 13190	_fp_round_special_aux:Nw	
_fp_pow_C_pack:w ..	13176 , 13184 , 13195		10256 , 10324 , 10331
_fp_pow_C_pos:w	13160 , 13179	_fp_round_to_nearest:NNN	
_fp_pow_C_pos_loop:wN			10167 , 10188 , 10201 ,
	13180 , 13181 , 13188		10254 , 11080 , 11085 , 13729 , 13949
_fp_pow_exponent:Nwnnnnnwn		_fp_round_to_nearest_neg:NNN	
	13125 , 13128 , 13133		10229 , 10254 , 10255
_fp_pow_exponent:wnN ...	13117 , 13122	_fp_round_to_ninf:NNN	
_fp_pow_neg:www ..	13035 , 13197 , 13197		10167 , 10169 , 11074
_fp_pow_neg_case:w	13199 , 13212 , 13212	_fp_round_to_ninf_neg:NNN	10229 , 10229
_fp_pow_neg_case_aux:NNNNNNnw ...		_fp_round_to_pinf:NNN	
	13212 , 13227 , 13234 , 13244		10167 , 10179 , 11068
_fp_pow_neg_case_aux:nnnnn		_fp_round_to_pinf_neg:NNN	10229 , 10245
	13212 , 13216 , 13220	_fp_round_to_zero:NNN	
_fp_pow_normal:ww .	13040 , 13068 , 13068		10167 , 10178 , 11071
_fp_pow_npos:Nww ..	13079 , 13096 , 13096	_fp_round_to_zero_neg:NNN	10229 , 10238
_fp_pow_npos_aux:NNnw		_fp_sanitize:Nw	
	13102 , 13106 , 13112 , 13112		9772 , 9772 , 9783 , 10316 ,
_fp_pow_zero_or_inf:ww			10334 , 11777 , 11871 , 12042 , 12121 ,
	13042 , 13049 , 13049		12612 , 12847 , 13098 , 13499 , 13536
_fp_reverse_args:Nww	9727 , 9727 , 13541	_fp_sanitize:wN	9772 , 9783 , 10527 , 11152
_fp_round:NNN		_fp_sanitize_zero:w ..	9772 , 9779 , 9784
	10167 , 10201 , 10204 , 11834 ,	_fp_sec_o:w	13304 , 13304
	11845 , 12086 , 12098 , 12238 , 12249	_fp_sin_o:w	13259 , 13259 , 13915

_fp_sin_series:NNwww 13267,
 13282, 13297, 13312, 13451, 13451
 _fp_sin_series_aux:NNnw
 13451, 13455, 13467
 _fp_small_int:wTF ... 9925, 9925, 10258
 _fp_small_int_normal:NnwTF
 9925, 9929, 9934
 _fp_small_int_test:NnnwNTF
 9925, 9942, 9950
 _fp_small_int_true:wTF
 9925, 9928, 9933, 9953
 _fp_sub_back_far_o:NnnwnnnN
 11880, 11926, 11926
 _fp_sub_back_near_after:wNNNNw ...
 11886, 11888, 11895, 11963
 _fp_sub_back_near_o:nnnnnnnnN ...
 11876, 11886, 11886
 _fp_sub_back_near_pack:NNNNNNw ...
 11886, 11890, 11893, 11965
 _fp_sub_back_not_far_o:wwwNN
 11940, 11960, 11960
 _fp_sub_back_quite_far_ii:NN
 11944, 11946, 11950
 _fp_sub_back_quite_far_o:wwNN
 11938, 11944, 11944
 _fp_sub_back_shift:wnnnn
 11898, 11902, 11902
 _fp_sub_back_shift_ii:ww
 11902, 11904, 11907
 _fp_sub_back_shift_iii:NNNNNNNNw .
 11902, 11912, 11915, 11924
 _fp_sub_back_shift_iv:nnnnw
 11902, 11919, 11925
 _fp_sub_back_very_far_ii_o:nnNwNN
 11972, 11975, 11979
 _fp_sub_back_very_far_o:wwwNN ...
 11939, 11972, 11972
 _fp_sub_eq_o:Nwnnw 11851, 11854, 11862
 _fp_sub_npos_i_o:Nwnnw
 11856, 11865, 11869, 11869
 _fp_sub_npos_ii_o:Nwnnw
 11851, 11858, 11863
 _fp_sub_npos_o:NnwNnw
 11771, 11851, 11851
 _fp_tan_o:w 13319, 13319, 13917
 _fp_tan_series_aux_o:Nnw
 13505, 13509, 13519
 _fp_tan_series_o:NNwww
 13327, 13342, 13505, 13505
 _fp_ternary:NwN .. 11327, 11623, 11623
 _fp_ternary_auxi:NwN
 11623, 11629, 11661
 _fp_ternary_auxii:NwN
 11343, 11623, 11651, 11669
 _fp_ternary_break_point:n
 11623, 11629, 11648, 11660
 _fp_ternary_loop:Nw
 11623, 11626, 11653, 11658
 _fp_ternary_loop_break:w
 11623, 11628, 11648
 _fp_ternary_map_break:
 11623, 11656, 11660
 _fp_tmp:w 9883, 9893, 9894,
 9895, 9896, 9897, 9898, 9899, 9900,
 9901, 9902, 9903, 9904, 9905, 9906,
 9907, 9908, 10357, 10369, 10370,
 10371, 10372, 10373, 10374, 10375,
 11000, 11005, 11006, 11007, 11008,
 11009, 11010, 11011, 11012, 11020,
 11021, 11022, 11023, 11024, 11025,
 11026, 11027, 11028, 11029, 11052,
 11063, 11064, 11100, 11116, 11117,
 11155, 11175, 11176, 11177, 11178,
 11179, 11180, 11181, 11185, 13906,
 13913, 13914, 13915, 13916, 13917
 _fp_to_decimal_dispatch:w
 13614, 13618, 13621, 13621, 13728
 _fp_to_decimal_huge:wnnnn
 13621, 13659, 13681
 _fp_to_decimal_large:Nnw
 13621, 13655, 13672
 _fp_to_decimal_normal:wnnnnn
 13621, 13626, 13647, 13709
 _fp_to_int_dispatch:w
 13719, 13723, 13726, 13726
 _fp_to_scientific_dispatch:w
 13559, 13563, 13566, 13571
 _fp_to_scientific_normal:wNw
 13566, 13602, 13604, 13611
 _fp_to_scientific_normal:wnnnnn ..
 13566, 13576, 13598, 13702, 13706
 _fp_to_tl_dispatch:w
 13682, 13686, 13689, 13689, 13794
 _fp_to_tl_normal:nnnnn
 13689, 13694, 13699
 _fp_trap_division_by_zero_set:N ..
 10056, 10057, 10059, 10061, 10062
 _fp_trap_division_by_zero_set_error:
 10056, 10056

_fp_trap_division_by_zero_set_flag:	_fp_trig_small:ww . 13369, 13390, 13390
..... 10056, 10058	_fp_trim_zeros:w 13550, 13550, 13612, 13665, 13674
_fp_trap_division_by_zero_set_none: 10056, 10060
..... 10022, 10023, 10025, 10027, 10028	_fp_trim_zeros_dot:w 13550, 13553, 13556
_fp_trap_invalid_operation_set:N .	_fp_trim_zeros_end:w 13550, 13556, 13557
..... 10022, 10022	_fp_trim_zeros_loop:w 13550, 13552, 13553, 13555
_fp_trap_invalid_operation_set_error:	_fp_type_from_scan:N . 9840, 10342,
..... 10022, 10022	10350, 10422, 10980, 10982, 10999
_fp_trap_invalid_operation_set_flag:	_fp_type_from_scan:w 10342, 10352, 10355
..... 10022, 10024	_fp_underflow:w ... 9778, 10113, 10119
_fp_trap_invalid_operation_set_none:	_fp_use_i_until_s:nw 9724, 9725, 9766, 9973, 13445
..... 10022, 10026	_fp_use_ii_until_s:nnw 9724, 9726, 9764
_fp_trap_overflow_set:N 10082, 10083, 10085, 10087, 10088	_fp_use_none_stop_f:n 9721, 9721, 12429, 12430, 12431
_fp_trap_overflow_set:NnNn 10082, 10089, 10097, 10098	_fp_use_none_until_s:w 9724, 9724, 13206
_fp_trap_overflow_set_error: 10082, 10082	_fp_use_s:n 9722, 9722
_fp_trap_overflow_set_flag: 10082, 10084	_fp_use_s:nn 9722, 9723
_fp_trap_overflow_set_none: 10082, 10086	_fp_zero_fp:N . 9747, 9747, 10097, 10322
_fp_trap_underflow_set:N 10082, 10091, 10093, 10095, 10096	_int_abs:N 3394, 3396, 3400
_fp_trap_underflow_set_error: 10082, 10090	_int_case:nnn 3616, 3619, 3621
_fp_trap_underflow_set_flag: 10082, 10092	_int_case:nw ... 3616, 3622, 3623, 3627
_fp_trap_underflow_set_none: 10082, 10094	_int_case_end:nw 3616, 3626, 3629
_fp_trig_epsilon_inv_o:w 13296, 13341, 13377, 13383	_int_compare:NNw 3560, 3572, 3576
_fp_trig_epsilon_o:w 13266, 13326, 13377, 13377	_int_compare:Nw 3560, 3568, 3570, 3592
_fp_trig_epsilon_one_o:w 13281, 13311, 13377, 13379	_int_compare:nnN . 3560, 3587, 3595,
_fp_trig_exponent:NNNNwn 13266, 13281, 13296,	3597, 3599, 3601, 3603, 3605, 3607
13311, 13326, 13341, 13359, 13359	_int_compare:w 3560, 3562, 3565
_fp_trig_large:ww . 13367, 13400, 13400	_int_compare_<:NNw 3560
_fp_trig_large:www 13400, 13401, 13402, 13413	_int_compare_=:NNw 3560
_fp_trig_large_break:w 13400, 13404, 13423	_int_compare_>:NNw 3560
_fp_trig_large_o:wnnnn 13400, 13405, 13411	_int_constdef:Nw 3466, 3477, 3489, 3493
_fp_trig_octant_break:w 13425, 13433, 13441	_int_div_truncate:NwNw 3426, 3429, 3434, 3457
_fp_trig_octant_loop:nnnnnw 13424, 13425, 13425, 13439	_int_eval:w ... 75, 1300, 2285, 2612,
	2614, 2616, 2682, 2684, 2686, 2688,
	2690, 2692, 2694, 2696, 2698, 2700,
	2702, 2704, 3383, 3384, 3389, 3392,
	3397, 3405, 3406, 3413, 3414, 3428,
	3430, 3431, 3448, 3451, 3452, 3453,
	3478, 3515, 3517, 3539, 3563, 3592,
	3610, 3632, 3640, 3866, 3893, 4048,
	4092, 4167, 9646, 9771, 9871, 9874,
	9961, 10207, 10211, 10223, 10224,
	10283, 10287, 10326, 10446, 10472,
	10528, 10585, 10596, 10645, 10676,
	10682, 10683, 10830, 10832, 10855,
	10856, 10862, 10871, 10882, 10884,
	10944, 11089, 11153, 11273, 11441,

11779, 11787, 11808, 11810, 11831,
 11833, 11842, 11844, 11866, 11873,
 11879, 11889, 11891, 11964, 11966,
 11982, 11984, 11988, 12004, 12044,
 12052, 12054, 12056, 12058, 12060,
 12062, 12064, 12083, 12085, 12095,
 12097, 12123, 12126, 12134, 12136,
 12156, 12159, 12162, 12165, 12173,
 12176, 12179, 12182, 12189, 12191,
 12197, 12205, 12207, 12209, 12215,
 12235, 12237, 12246, 12248, 12261,
 12280, 12287, 12299, 12307, 12309,
 12319, 12321, 12328, 12337, 12339,
 12342, 12345, 12348, 12351, 12365,
 12367, 12375, 12377, 12385, 12387,
 12396, 12399, 12402, 12409, 12428,
 12473, 12475, 12479, 12510, 12545,
 12552, 12570, 12572, 12620, 12631,
 12650, 12652, 12654, 12666, 12679,
 12684, 12686, 12692, 12709, 12710,
 12711, 12712, 12713, 12714, 12719,
 12721, 12723, 12725, 12727, 12731,
 12733, 12735, 12737, 12739, 12741,
 12763, 12771, 12849, 12853, 12906,
 13115, 13136, 13138, 13141, 13144,
 13147, 13150, 13166, 13192, 13202,
 13365, 13406, 13409, 13414, 13416,
 13418, 13420, 13458, 13469, 13501,
 13511, 13538, 13540, 13606, 13951
 _int_eval_end: 75,
 1300, 2285, 2612, 2614, 2616, 2682,
 2684, 2686, 2688, 2690, 2692, 2694,
 2696, 2698, 2700, 2702, 2704, 3383,
 3385, 3389, 3392, 3397, 3432, 3448,
 3454, 3478, 3515, 3517, 3539, 3610,
 3632, 3640, 3866, 3893, 4168, 9649,
 9771, 9964, 10297, 10301, 11089,
 11273, 11441, 11701, 11866, 11988,
 12023, 12211, 12261, 12309, 12510,
 13166, 13458, 13469, 13511, 13540
 _int_from_alph:N 3972, 3988, 3991
 _int_from_alph:n 3972, 3977, 3980
 _int_from_alph:nN 3972, 3981, 3982, 3987
 _int_from_base:N 3993, 4010, 4014
 _int_from_base:nn 3993, 3998, 4002
 _int_from_base:nnN 3993, 4003, 4004, 4009
 _int_from_roman:NN
 4043, 4049, 4052, 4077, 4081
 _int_from_roman_clean_up:w
 4043, 4060, 4067, 4069, 4088
 _int_from_roman_end:w 4043, 4047, 4086
 _int_get_digits:n 3938, 3943, 3977, 3999
 _int_get_sign:n 3938, 3938, 3976, 3997
 _int_get_sign_and_digits:nNNN
 3938, 3940, 3945, 3948, 3971
 _int_get_sign_and_digits:oNNN
 3938, 3954, 3958, 3964
 _int_maxmin:wwN 3394, 3404, 3412, 3418
 _int_mod:ww 3426, 3451, 3456
 _int_step:Nnnnnn 3725, 3728, 3735, 3744
 _int_step:NnnnN
 3702, 3705, 3712, 3716, 3721
 _int_to_Roman_Q:w 3902, 3937
 _int_to_Roman_aux:N .. 3914, 3917, 3920
 _int_to_Roman_c:w 3902, 3934
 _int_to_Roman_d:w 3902, 3935
 _int_to_Roman_i:w 3902, 3930
 _int_to_Roman_l:w 3902, 3933
 _int_to_Roman_m:w 3902, 3936
 _int_to_Roman_v:w 3902, 3931
 _int_to_Roman_x:w 3902, 3932
 _int_to_base:nn 3835, 3836, 3837
 _int_to_base:nnN
 3835, 3840, 3841, 3843, 3857
 _int_to_base:nnnN 3835, 3848, 3855
 _int_to_letter:n 3835, 3846, 3849, 3863
 _int_to_roman:N
 3902, 3902, 3904, 3907, 3910
 _int_to_roman:w . 74, 793, 794, 894,
 896, 1069, 2283, 3383, 3573, 3905, 3915
 _int_to_roman_Q:w 3902, 3929
 _int_to_roman_c:w 3902, 3926
 _int_to_roman_d:w 3902, 3927
 _int_to_roman_i:w 3902, 3922
 _int_to_roman_l:w 3902, 3925
 _int_to_roman_m:w 3902, 3928
 _int_to_roman_v:w 3902, 3923
 _int_to_roman_x:w 3902, 3924
 _int_to_symbols:nnnn . 3751, 3755, 3765
 _int_value:w 75, 1075,
 2197, 2200, 2285, 3383, 3383, 3389,
 3392, 3396, 3404, 3412, 3448, 3451,
 3452, 3453, 3567, 3592, 3893, 4048,
 4171, 4265, 6817, 6848, 6849, 6850,
 6852, 6978, 6987, 6989, 6994, 6995,
 6996, 6997, 7001, 7002, 7003, 7004,
 7029, 7035, 7037, 7039, 7041, 7046,
 7051, 7056, 7063, 7070, 7200, 7230,
 7231, 7270, 7289, 7295, 7465, 7569,
 9646, 9829, 9830, 9831, 9832, 9833,

9888, 9954, 10269, 10285, 10290,
 10415, 10580, 10587, 10600, 10637,
 10648, 10654, 10665, 10686, 10702,
 10778, 10780, 10895, 11456, 11724,
 11725, 11726, 11728, 11782, 11785,
 11848, 11905, 11913, 11921, 11987,
 11990, 12089, 12101, 12139, 12140,
 12141, 12142, 12252, 12269, 12430,
 12431, 12432, 12613, 12627, 12648,
 12670, 12671, 12672, 12673, 12674,
 12787, 12792, 12796, 12849, 12918,
 13100, 13157, 13160, 13162, 13188,
 13190, 13367, 13369, 13457, 13510,
 13674, 13734, 13744, 13748, 14607,
 14609, 14632, 14634, 14659, 14699,
 14726, 14734, 14759, 14761, 14765,
 14767, 14796, 14810, 14817, 15018
 __ior_list_streams:Nn
 9370, 9371, 9372, 9465
 __ior_map_inline:NNNn 14828, 14835, 14838
 __ior_map_inline:NNn
 14828, 14829, 14831, 14832
 __ior_map_inline_loop:NNN
 14828, 14841, 14845, 14851
 __ior_open:Nn
 ... 167, 9186, 9206, 9335, 9335, 9351
 __ior_open:No 9318, 9331, 9335
 __ior_open_aux:Nn 9310, 9311, 9313
 __ior_open_aux:NnTF ... 9320, 9321, 9325
 __ior_open_stream:Nn
 9335, 9339, 9347, 9352
 __iow_indent:n 9515, 9516, 9534
 __iow_list_streams:Nn . 9463, 9464, 9465
 __iow_open:Nn ... 9425, 9426, 9428, 9444
 __iow_open_stream:Nn
 9425, 9432, 9440, 9445
 __iow_wrap_end: 9636
 __iow_wrap_end:w 9609
 __iow_wrap_indent: 9624
 __iow_wrap_indent:w 9609
 __iow_wrap_loop:w
 9555, 9564, 9564, 9579, 9614
 __iow_wrap_newline: 9616
 __iow_wrap_newline:w 9609
 __iow_wrap_special:w
 9568, 9609, 9609, 9613
 __iow_wrap_unindent: 9630
 __iow_wrap_unindent:w 9609
 __iow_wrap_word: 9569, 9571, 9571
 __iow_wrap_word_fits: . 9571, 9577, 9581
 __iow_wrap_word_newline: 9571, 9578, 9597
 __kernel_register_show:N .. 25, 1448,
 1448, 1458, 4089, 4398, 4488, 4550
 __kernel_register_show:c 1448, 1457, 4090
 __keys_bool_set:Nn
 8638, 8638, 8653, 8763, 8767
 __keys_bool_set:cn 8638, 8765, 8769
 __keys_bool_set_inverse:Nn
 8654, 8654, 8669, 8771, 8775
 __keys_bool_set_inverse:cn
 8654, 8773, 8777
 __keys_choice_code_store:n
 9066, 9066, 9077, 9079
 __keys_choice_code_store:x . 9066, 9081
 __keys_choice_find:n
 8673, 8722, 8992, 8992
 __keys_choice_make: 8641,
 8657, 8670, 8670, 8682, 8779, 9087
 __keys_choices_generate:n
 9082, 9082, 9116
 __keys_choices_generate_aux:n
 9082, 9089, 9096
 __keys_choices_make:nn
 .. 8680, 8680, 8781, 8783, 8785, 8787
 __keys_cmd_set:Vn 8696, 8720
 __keys_cmd_set:Vo 8696, 8716
 __keys_cmd_set:n 8696, 8698, 8702
 __keys_cmd_set:nn
 8646, 8662, 8672, 8674,
 8696, 8696, 8701, 8725, 8727, 8789
 __keys_cmd_set:nx .. 8642, 8644, 8658,
 8660, 8687, 8696, 8740, 8758, 9099
 __keys_default_set:n ... 8651, 8667,
 8708, 8708, 8799, 8801, 8803, 8805
 __keys_define:nnn 8569, 8571, 8577
 __keys_define:onn 8569, 8570
 __keys_define_elt:n ... 8574, 8578, 8578
 __keys_define_elt:nn .. 8574, 8578, 8583
 __keys_define_elt_aux:nn
 8578, 8581, 8586, 8588
 __keys_define_key:n ... 8592, 8621, 8621
 __keys_define_key:w ... 8621, 8625, 8636
 __keys_execute: 8938, 8963, 8963
 __keys_execute:nn
 .. 8963, 8964, 8967, 8983, 8994, 8995
 __keys_execute_unknown:
 .. 8896, 8898, 8963, 8964, 8965, 8973
 __keys_execute_unknown_alt:
 8896, 8963, 8974

_keys_execute_unknown_std:	_msg_class_chk_exist:nT
. 8898, 8963, 8973	. . . 7923, 7923, 7937, 8003, 8013, 8018
_keys_if_value:n 8956	_msg_class_new:nn . 7818, 7819, 7856,
_keys_if_value_p:n . . . 8921, 8931, 8956	7867, 7878, 7899, 7907, 7915, 7921
_keys_initialise:n	_msg_error:cnnnnn 7878, 7880, 7892
. . 8710, 8710, 8823, 8825, 8827, 8829	_msg_error_code:nnnnnn 8104
_keys_initialise:wn . . 8710, 8711, 8712	_msg_expandable_error:n
_keys_meta_make:n 8714, 8714, 8839 149, 8269, 8277, 8294
_keys_meta_make:nn . . . 8714, 8719, 8841	_msg_expandable_error:w 8269, 8283, 8289
_keys_multichoice_find:n	_msg_fatal_code:nnnnnn 8100
. 8721, 8721, 8726	_msg_interrupt_more_text:n
_keys_multichoice_make: 7746, 7748, 7751
. 8721, 8723, 8735, 8843	_msg_interrupt_text:n 7749, 7760, 7768
_keys_multichoices_make:nn	_msg_interrupt_wrap:nn
. . 8721, 8733, 8845, 8847, 8849, 8851 7738, 7742, 7746, 7746
_keys_property_find:n 8590, 8601, 8601	_msg_kernel_bug:x 8390, 8391
_keys_property_find:w	_msg_kernel_class_new:nN
. 8601, 8605, 8611, 8617	. . . 8062, 8063, 8100, 8104, 8105, 8106
_keys_set:nnn 8880, 8882, 8889	_msg_kernel_class_new_aux:nN
_keys_set:onn 8880, 8881 8062, 8064, 8065
_keys_set_elt:n 8885, 8897, 8904, 8904	_msg_kernel_error:nn
_keys_set_elt:nn 8885, 8897, 8904, 8909	. . 1175, 1189, 7088, 8100, 8103, 8498
_keys_set_elt_aux:nn	_msg_kernel_error:nnn 8100
. 8904, 8907, 8912, 8914	_msg_kernel_error:nnnn 8100
_keys_set_known:nnnN . 8890, 8892, 8903	_msg_kernel_error:nnnnn 8100
_keys_set_known:onnN 8890, 8891	_msg_kernel_error:nnnnnn . . . 148, 8100
_keys_value_or_default:n	_msg_kernel_error:nnx 938, 979, 1017,
. 8918, 8941, 8941	1022, 1175, 1187, 1220, 1371, 1453,
_keys_value_requirement:n	1926, 2137, 2589, 4792, 5302, 6683,
. 8749, 8749, 8877, 8879	6832, 7926, 8100, 8102, 8342, 8607,
_keys_variable_set:NnnN	8648, 8664, 8925, 9092, 9172, 9231,
. . . 8755, 8755, 8761, 8791, 8795,	9317, 9731, 10019, 13842, 14021, 14063
8807, 8811, 8815, 8819, 8831, 8835,	_msg_kernel_error:nnxx . . . 929, 958,
8853, 8857, 8861, 8865, 8869, 8873	1036, 1175, 1175, 1188, 1190, 1197,
_keys_variable_set:cnnN	1207, 1341, 1987, 6980, 7653, 7663,
. 8755, 8793, 8797,	7951, 8100, 8101, 8596, 8629, 8676,
8809, 8813, 8817, 8821, 8833, 8837,	8729, 8935, 8969, 10016, 14058, 14086
8855, 8859, 8863, 8867, 8871, 8875	_msg_kernel_error:nnxxx 8100
_keyval_parse:n 8451, 8459, 8545	_msg_kernel_error:nnxxxx . . 2002, 8100
_keyval_parse_elt:w	_msg_kernel_expandable_error:nn . .
. 8467, 8473, 8473, 8476, 8480	. 2313, 5397, 8292, 8316, 10432, 11396
_keyval_split_key:n . . 8485, 8503, 8503	_msg_kernel_expandable_error:nnn .
_keyval_split_key:w . . 8503, 8507, 8509	1681, 3556, 3709, 4996, 5753, 6212,
_keyval_split_key_value:w	8292, 8311, 10438, 10489, 10510,
. 8479, 8483, 8483	10517, 10767, 10772, 10783, 10790,
_keyval_split_key_value:wTF	10962, 11142, 11359, 13983, 13988
. 8483, 8496, 8501	_msg_kernel_expandable_error:nnnn
_keyval_split_value:w 8497, 8511, 8511 8292, 8306, 11253, 11340, 11638
_keyval_split_value_aux:w . 8529, 8532	_msg_kernel_expandable_error:nnnnn
 8292, 8301, 10127, 11093

_msg_kernel_expandable_error:nnnnnn	_msg_redirect_loop_chk:onn 8048
148, 8292, 8292, 8303, 8308, 8313, 8318	_msg_redirect_loop_list:n
_msg_kernel_fatal:nn	8007, 8044, 8053
8100, 9341, 9434, 14042	_msg_show_item:n
_msg_kernel_fatal:nnn 8100	149, 5805, 6227, 8357, 8357
_msg_kernel_fatal:nnnn 8100	_msg_show_item:nn 6508, 8357, 8361
_msg_kernel_fatal:nnnnn 8100	_msg_show_item_unbraced:nn
_msg_kernel_fatal:nnnnnn 148, 8100	149, 7570, 8357, 8366, 9378
_msg_kernel_fatal:nnx 8100, 14015	_msg_show_variable:Nnn
_msg_kernel_fatal:nnxx 8100	149, 5804, 6226, 6507, 7566, 8334, 8334
_msg_kernel_fatal:nnxxx 8100	_msg_show_variable:n
_msg_kernel_fatal:nnxxxx 8100	149, 1467, 2144, 2145, 5308,
_msg_kernel_info:nn 8105	8334, 8339, 8346, 9377, 13840, 13847
_msg_kernel_info:nnn 8105	_msg_show_variable_aux:n
_msg_kernel_info:nnnn 8105	8334, 8347, 8348
_msg_kernel_info:nnnnn 8105	_msg_show_variable_aux:w
_msg_kernel_info:nnnnnn 148, 8105	8334, 8352, 8356
_msg_kernel_info:nnx 8105	_msg_term:nn 8321, 8332
_msg_kernel_info:nnxx 8105	_msg_term:nnn 8321, 8330, 8338, 9374
_msg_kernel_info:nnxxx 8105	_msg_term:nnnnn 8321, 8328
_msg_kernel_info:nnxxxx 8105	_msg_term:nnnnnV 8321
_msg_kernel_new:nnn 7593,	_msg_term:nnnnnn
8054, 8056, 8233, 8235, 8237, 8239,	149, 8321, 8321, 8327, 8329, 8331, 8333
8241, 8243, 8251, 8258, 8265, 8267,	_msg_use:nnnnnnn 7827, 7933, 7933, 8427
10152, 10154, 10156, 10158, 10160,	_msg_use_code:
10162, 11400, 11402, 11404, 11406,	7933, 7940, 7953, 7957, 7982, 7993
11408, 11410, 11412, 11414, 11416	_msg_use_hierarchy:nwnN
_msg_kernel_new:nnnn	7933, 7960, 7961, 7967
147, 7575, 7583, 7586, 8054, 8054,	_msg_use_redirect_module:n
8108, 8116, 8124, 8131, 8142, 8150,	7933, 7964, 7972, 7985
8159, 8166, 8173, 8180, 8189, 8196,	_msg_use_redirect_name:n
8203, 8210, 8217, 8225, 8548, 9011,	7933, 7948, 7954
9014, 9020, 9025, 9034, 9040, 9047,	_my_map_dbl:nn 4, 6, 11
9054, 9060, 9108, 9660, 9666, 9673,	_my_map_dbl_fn:nn 3, 10
9680, 9981, 10129, 10141, 14131, 14137	_peek_N_type:w 15172, 15187, 15197
_msg_kernel_set:nnn 8054, 8060	_peek_N_type_aux:nnw 15172, 15189, 15201
_msg_kernel_set:nnnn 147, 8054, 8058	_peek_def:nnnn 3185, 3186,
_msg_kernel_warning:nn 8105	3202, 3206, 3210, 3214, 3218, 3222,
_msg_kernel_warning:nnn 8105	3226, 3230, 3234, 3238, 3242, 3246
_msg_kernel_warning:nnnn 8105	_peek_def:nnnnn
_msg_kernel_warning:nnnnn 8105	3185, 3188, 3189, 3190, 3192
_msg_kernel_warning:nnnnnn 148, 8105	_peek_execute_branches: 3182, 3197
_msg_kernel_warning:nnx 8105	_peek_execute_branches_N_type:
_msg_kernel_warning:nnxx 8105	15172, 15180, 15209, 15211, 15213
_msg_kernel_warning:nnxxx 8105	_peek_execute_branches_catcode:
_msg_kernel_warning:nnxxxx 8038, 8105	3142, 3142, 3205, 3207, 3213, 3215
_msg_no_more_text:nnnn 7878, 7894, 7898	_peek_execute_branches_catcode_aux:
_msg_redirect:nnn 8007, 8008, 8010, 8011	3142, 3143, 3145, 3146
_msg_redirect_loop_chk:nnn	_peek_execute_branches_catcode_auxii:N
8007, 8023, 8028, 8052	3142, 3150, 3156

__peek_execute_branches_catcode_auxiii:	__prg_generate_conditional:nnnnnnw
..... 3142, 3153, 3166 935, 944, 950, 965
__peek_execute_branches_charcode: .	__prg_generate_conditional_count:nnNnn
.. 3142, 3144, 3221, 3223, 3229, 3231 910, 911, 913, 915, 917, 918
__peek_execute_branches_meaning: .	__prg_generate_conditional_count:nnNnnnn
.. 3134, 3134, 3237, 3239, 3245, 3247 910, 920, 923
__peek_false:w 3081, 3083, 3104,	__prg_generate_conditional_parm:nnNpnn
3122, 3139, 3162, 3172, 15194, 15206 897, 898, 900, 902, 904, 905
__peek_get_prefix_arg_replacement:wN	__prg_generate_p_form:wnnnnnn 967, 967
..... 3251, 3252, 3259, 3268, 3277	__prg_map_break:Nn .. 43, 1556, 1557,
__peek_ignore_spaces_execute_branches:	1563, 2373, 4987, 4989, 5668, 5670,
..... 3175, 3175, 3179,	6166, 6168, 6502, 6504, 14825, 14827
3209, 3217, 3225, 3233, 3241, 3249	__prg_replicate:N 2281, 2288, 2289, 2291
__peek_tmp:w 3081, 3084, 3093	__prg_replicate 2281, 2292
__peek_token_generic:NNF .. 3114, 15213	__prg_replicate_0:n 2281
__peek_token_generic:NNT .. 3112, 15211	__prg_replicate_1:n 2281
__peek_token_generic:NNTF ..	__prg_replicate_2:n 2281
..... 3095, 3095, 3113, 3115, 15209	__prg_replicate_3:n 2281
__peek_token_remove_generic:NNF . 3132	__prg_replicate_4:n 2281
__peek_token_remove_generic:NNT . 3130	__prg_replicate_5:n 2281
__peek_token_remove_generic:NNTF ..	__prg_replicate_6:n 2281
..... 3116, 3116, 3131, 3133	__prg_replicate_7:n 2281
__peek_true:w 3081, 3081, 3099, 3120,	__prg_replicate_8:n 2281
3137, 3160, 3170, 15192, 15205, 15206	__prg_replicate_9:n 2281
__peek_true_aux:w 3081, 3082, 3092, 3121	__prg_replicate_first:N 2281, 2284, 2290
__peek_true_remove:w .. 3089, 3089, 3120	__prg_replicate_first -:n 2281
__prg_break: ..	__prg_replicate_first_0:n 2281
. 43, 1565, 1566, 2373, 6449, 9962,	__prg_replicate_first_1:n 2281
13781, 14904, 14922, 14930, 15144	__prg_replicate_first_2:n 2281
__prg_break:n 1565,	__prg_replicate_first_3:n 2281
1567, 2373, 5543, 14887, 14912, 15146	__prg_replicate_first_4:n 2281
__prg_break_point: .. 43, 1565, 1565,	__prg_replicate_first_5:n 2281
1566, 1567, 2373, 5540, 6434, 9963,	__prg_replicate_first_6:n 2281
13782, 14882, 14906, 14923, 15140	__prg_replicate_first_7:n 2281
__prg_break_point:Nn 43, 1556,	__prg_replicate_first_8:n 2281
1556, 1557, 2373, 3748, 4948, 4965,	__prg_replicate_first_9:n 2281
4974, 5675, 5710, 5721, 6098, 6112,	__prg_set_eq_conditional:NNNn
6131, 6149, 6478, 6498, 14842, 14867 998, 999, 1001, 1002
__prg_case_end:nw 25, 1532, 1554, 1555, 3629, 4320, 4943	__prg_set_eq_conditional:nnNnnNNw .
__prg_compare_error: 75, 3545, 1006, 1014, 1014
3545, 3549, 3563, 3565, 4274, 4276	__prg_set_eq_conditional_F_form:nnn
__prg_compare_error:NNw 3545 1014
__prg_compare_error:Nw 3551, 3580	__prg_set_eq_conditional_F_form:wNnnnnn
__prg_generate_F_form:wnnnnnn 967, 988 1060
__prg_generate_TF_form:wnnnnnn 967, 993	__prg_set_eq_conditional_TF_form:nnn
__prg_generate_T_form:wnnnnnn 967, 983 1014
__prg_generate_conditional:nnNnnnnn	__prg_set_eq_conditional_TF_form:wNnnnn
..... 907, 926, 935, 935 1050
	__prg_set_eq_conditional_T_form:nnn
 1014

__prg_set_eq_conditional_T_form:wNnnnn
 1055
 __prg_set_eq_conditional_loop:nnnnNw
 1014, 1026, 1028, 1043
 __prg_set_eq_conditional_p_form:nnn
 1014
 __prg_set_eq_conditional_p_form:wNnnnn
 1045
 __prg_variable_get_scope:N
 42, 2339, 2345
 __prg_variable_get_scope:w
 2339, 2348, 2351
 __prg_variable_get_type:N 42, 2339, 2360
 __prg_variable_get_type:w
 2339, 2362, 2365, 2369
 __prop_get_Nn:nwn
 14878, 14880, 14884, 14888
 __prop_if_in:N 6428, 6439, 6442
 __prop_if_in:nwn 6428, 6430, 6436, 6440
 __prop_map_function:Nwn
 6474, 6476, 6480, 6486, 6494
 __prop_map_tokens:nwn
 14863, 14865, 14869, 14875
 __prop_put:NNNnn 6388, 6389, 6391, 6392
 __prop_put_if_new:NNnn
 6408, 6409, 6411, 6412
 __prop_split:NnTF 127,
 6305, 6305, 6318, 6324, 6334, 6342,
 6351, 6364, 6374, 6396, 6416, 6461
 __prop_split_aux:NnTF . 6305, 6306, 6307
 __prop_split_aux:w 6305, 6309, 6312, 6315
 __quark_if_recursion_tail_break:NN
 48, 2526, 2526, 2600, 4981
 __quark_if_recursion_tail_break:nN
 2526,
 2532, 2602, 4954, 6103, 6116, 15144
 __scan_new:N 48,
 2585, 2585, 2597, 6280, 9728, 9734,
 9735, 9736, 9737, 9738, 9739, 9740
 __seq_count:n 5725, 5730, 5733
 __seq_get_left:Nnw 5572, 5576
 __seq_get_left:NnwN 5568
 __seq_get_right_loop:nn
 5592, 5594, 5603, 5606
 __seq_if_in: 5525, 5534, 5542
 __seq_item:n 112,
 5395, 5395, 5462, 5464, 5470, 5472,
 5477, 5530, 5573, 5576, 5585, 5615,
 5616, 5628, 5687, 5692, 5698, 5702,
 5742, 5747, 5756, 5759, 5762, 5765,
 14975, 14976, 14978, 14983, 15003
 __seq_item:nnn 14892, 14894, 14908, 14913
 __seq_map_function:NNn
 5671, 5673, 5677, 5681
 __seq_mapthread_function:NN
 14916, 14918, 14925
 __seq_mapthread_function:Nnnwnn ...
 14916, 14927, 14933, 14938
 __seq_pop:NNNN
 .. 5550, 5550, 5580, 5582, 5610, 5612
 __seq_pop_TF:NNNN 5550, 5558,
 5638, 5640, 5648, 5650, 5652, 5654
 __seq_pop_item_def: 112, 5517,
 5684, 5700, 5710, 5721, 14995, 15005
 __seq_pop_left:NNN
 .. 5579, 5580, 5582, 5583, 5648, 5650
 __seq_pop_left:NnwNNN . 5579, 5584, 5585
 __seq_pop_right_aux:NNN
 .. 5609, 5610, 5612, 5613, 5652, 5654
 __seq_pop_right_loop:nn
 5609, 5620, 5630, 5633
 __seq_push_item_def:
 5684, 5686, 5691, 5694
 __seq_push_item_def:n 112,
 5501, 5684, 5684, 5708, 14993, 15003
 __seq_push_item_def:x . 5684, 5689, 5715
 __seq_remove_all_aux:NNn
 5495, 5496, 5498, 5499
 __seq_remove_duplicates:NN
 5479, 5480, 5482, 5483
 __seq_reverse:NN
 14968, 14970, 14972, 14973
 __seq_reverse_item:nwn
 14968, 14976, 14980
 __seq_set_filter:NNNn
 14987, 14988, 14990, 14991
 __seq_set_map:NNNn
 14997, 14998, 15000, 15001
 __seq_set_split:NNnn
 5421, 5422, 5424, 5425
 __seq_set_split_auxi:w
 5421, 5432, 5439, 5445
 __seq_set_split_auxii:w 5421, 5447, 5451
 __seq_set_split_end:
 .. 5421, 5434, 5438, 5445, 5449, 5451
 __seq_tmp:w .. 14968, 14968, 14975, 14978
 __seq_use:NnNnn . 5735, 5742, 5743, 5756
 __seq_use:nwnn 5735, 5749, 5765
 __seq_use:nwwwnwn 5735, 5746, 5748, 5758

_seq_wrap_item:n	_tl_case_end:nw
... 5428, 5452, 5477, 5477, 5513,	4931, 4939, 4943
14945, 14950, 14955, 14960, 14993	_tl_count:n 4999, 5002, 5007, 5009
_skip_if_finite:wwNw . 4465, 4469, 4473	_tl_head_auxi:nw 5146, 5149, 5151, 5162
_str_case:nw ... 1532, 1535, 1537, 1541	_tl_head_auxii:nw 5146, 5152, 5153
_str_case_end:nw 1532, 1540, 1551, 1555	_tl_if_blank_p:NNw 4838
_str_case_x:nw . 1532, 1546, 1548, 1552	_tl_if_empty_return:o
_str_count_ignore_spaces:N 4839, 4872, 4872, 4882
..... 9574, 9642, 9642	_tl_if_head_eq_meaning_normal:nN .
_str_count_ignore_spaces:n 5232, 5236
..... 9642, 9643, 9644	_tl_if_head_eq_meaning_special:nN
_str_count_loop:NNNNNNNN 5233, 5245
..... 9642, 9647, 9651, 9657	_tl_if_head_is_space:w 5285, 5288, 5290
_str_head:w 5176, 5178, 5182	_tl_item:nn . 15127, 15129, 15142, 15147
_str_if_eq_x_return:nn	_tl_map_function:Nn
. 26, 1524, 1524, 2871, 2880, 2896, 4944, 4946, 4952, 4955, 4962
2918, 2938, 2956, 2974, 2987, 2998,	_tl_map_variable:Nnn
3008, 4930, 5266, 5374, 15037, 15038 4970, 4972, 4978, 4983
_str_tail:w 5176, 5186, 5190	_tl_replace:NNNnn
_tl_act:NNNnn 4776, 4777, 4779, 4781, 4783, 4788
5074, 5074, 5125, 15045, 15068, 15108	_tl_replace:w .. 4776, 4813, 4816, 4821
_tl_act_case_aux:nn	_tl_replace_all:
..... 15095, 15103, 15106, 15125 4776, 4781, 4783, 4814, 4817
_tl_act_case_group:nn	_tl_replace_once: 4776, 4777, 4779, 4819
..... 15090, 15110, 15122	_tl_replace_once_end:w 4776, 4822, 4824
_tl_act_case_normal:nN	_tl_rescan:w 4738, 4756, 4764
..... 15090, 15109, 15114	_tl_reverse_group:nn 15040, 15047, 15053
_tl_act_case_space:n 15090, 15111, 15113	_tl_reverse_group_preserve:nn
_tl_act_count_group:nn 5120, 5127, 5136
..... 15064, 15070, 15078	_tl_reverse_items:nwNwn
_tl_act_count_normal:nN 5012, 5014, 5015, 5019, 5022
..... 15064, 15069, 15076	_tl_reverse_items:wn
_tl_act_count_space:n 5012, 5016, 5023, 5026
..... 15064, 15071, 15077	_tl_reverse_normal:nN
_tl_act_end:w 5074 5120, 5126, 5134, 15046
_tl_act_end:wn 5095, 5101	_tl_reverse_space:n
_tl_act_group:nwnNNN . 5074, 5087, 5103 5120, 5128, 5138, 15048
_tl_act_group_recurse:Nnn	_tl_set_rescan:NNnn
..... 15055, 15059, 15059 4738, 4739, 4741, 4743, 4744
_tl_act_loop:w	_tl_tmp:w 4797,
.. 5074, 5077, 5081, 5098, 5106, 5113	4817, 4822, 4918, 4919, 5036, 5073
_tl_act_normal:NwnNNN 5074, 5084, 5092	_tl_trim_spaces:nn
_tl_act_output:n 103, 5029, 5036, 5038, 5876, 14545
.... 5074, 5116, 15113, 15116, 15124	_tl_trim_spaces_auxi:w
_tl_act_result:n 5036, 5040, 5050, 5053, 5059
.. 5079, 5101, 5116, 5117, 5118, 5119	_tl_trim_spaces_auxii:w ... 5044, 5058
_tl_act_reverse_output:n	_tl_trim_spaces_auxii:w_tl_trim_spaces_auxiii:w
. 5074, 5118, 5135, 5137, 5139, 15056 5036
_tl_act_space:wnNNN . 5074, 5088, 5110	_tl_trim_spaces_auxiii:w
_tl_case:Nw 4931, 4934, 4936, 4940 5045, 5061, 5064, 5068
	_tl_trim_spaces_auxiv:w 5036, 5047, 5070

_token_if_chardef:w		\advance	333
..... <u>2858</u> , <u>2873</u> , <u>2882</u> , <u>2887</u>		\afterassignment	343
_token_if_dim_register:w		\aftergroup	344
..... <u>2858</u> , <u>2898</u> , <u>2905</u>		false	181
_token_if_int_register:w		nan	180
..... <u>2858</u> , <u>2920</u> , <u>2929</u>		tan	180
_token_if_long_macro:w		\assert:n	<u>12550</u> , <u>12567</u> , <u>12568</u>
..... <u>2858</u> , <u>3000</u> , <u>3010</u> , <u>3015</u>		\assert_str_eq:nn	<u>10908</u>
_token_if_macro_p:w	<u>2818</u> , <u>2829</u> , <u>2832</u>	\AtBeginDocument	<u>7620</u> , <u>9299</u>
_token_if_muskip_register:w		\atop	440
..... <u>2858</u> , <u>2940</u> , <u>2947</u>		\atopwithdelims	441
_token_if_primitive:Nnw	<u>3017</u> , <u>3030</u> , <u>3034</u>	max	179
_token_if_primitive:Nw	<u>3017</u> , <u>3053</u> , <u>3059</u>		
_token_if_primitive_loop:N		B	
..... <u>3017</u> , <u>3037</u> , <u>3050</u> , <u>3056</u>		\B	<u>4730</u> , <u>4732</u>
_token_if_primitive_nullfont:N		\badness	588
..... <u>3017</u> , <u>3038</u> , <u>3042</u>		\baselineskip	516
_token_if_primitive_space:w		\batchmode	409
..... <u>3017</u> , <u>3036</u> , <u>3041</u>		\BeginCatcodeRegime	<u>14070</u>
_token_if_primitive_undefined:N		\begingroup	<u>13</u> , <u>62</u> , <u>68</u> , <u>72</u> , <u>309</u> , <u>347</u>
..... <u>3017</u> , <u>3062</u> , <u>3068</u>		\beginL	701
_token_if_protected_macro:w		\beginR	703
..... <u>2858</u> , <u>2989</u> , <u>2994</u>		\belowdisplayshortskip	453
_token_if_skip_register:w		\belowdisplayskip	454
..... <u>2858</u> , <u>2958</u> , <u>2965</u>		\binoppenalty	477
_token_if_toks_register:w		\bodydir	735
..... <u>2858</u> , <u>2976</u> , <u>2983</u>		\bool_do_until:cn	<u>2248</u>
_use_none_delimit_by_s_stop:w		\bool_do_until:Nn	<u>40</u> , <u>2248</u> , <u>2250</u> , <u>2251</u> , <u>2253</u>
..... <u>48</u> , <u>2598</u> , <u>2598</u>		\bool_do_until:nx	<u>40</u> , <u>2254</u> , <u>2275</u> , <u>2278</u>
	4138, 11199	\bool_do_while:cn	<u>2248</u>
~	2734, 2744, 9530	\bool_do_while:Nn	<u>40</u> , <u>2248</u> , <u>2248</u> , <u>2249</u> , <u>2252</u>
		\bool_do_while:nx	<u>40</u> , <u>2254</u> , <u>2262</u> , <u>2265</u>
Numbers		.bool_gset:c	152
\8	8454	\bool_gset:cn	<u>2114</u>
\9	8455	.bool_gset:N	152
		\bool_gset:Nn	<u>38</u> , <u>2114</u> , <u>2116</u> , <u>2119</u>
_	<u>317</u> , <u>1070</u> , <u>2744</u> , <u>2901</u> , <u>2923</u> , <u>2943</u> ,	\bool_gset_eq:cc	<u>2106</u> , <u>2113</u>
	<u>2961</u> , <u>2979</u> , <u>2992</u> , <u>3003</u> , <u>3013</u> , <u>7761</u> ,	\bool_gset_eq:cN	<u>2106</u> , <u>2112</u>
	<u>7762</u> , <u>8112</u> , <u>8157</u> , <u>8176</u> , <u>8271</u> , <u>8359</u> ,	\bool_gset_eq:Nc	<u>2106</u> , <u>2111</u>
	<u>8363</u> , <u>8364</u> , <u>8368</u> , <u>8369</u> , <u>9495</u> , <u>9533</u>	\bool_gset_eq>NN	<u>38</u> , <u>2106</u> , <u>2110</u>
A		\bool_gset_false:c	<u>2094</u>
\A	<u>1896</u> , <u>2820</u> , <u>4729</u> , <u>4731</u>	\bool_gset_false:N	<u>37</u> , <u>2094</u> , <u>2100</u> , <u>2105</u>
\above	438	.bool_gset_inverse:c	152
\abovedisplayshortskip	451	.bool_gset_inverse:N	152
\abovedisplayskip	452	\bool_gset_true:c	<u>2094</u>
\abovewithdelims	439	\bool_gset_true:N	<u>38</u> , <u>2094</u> , <u>2098</u> , <u>2104</u>
\accent	489	\bool_if:cTF	<u>2120</u>
\adjdemerits	526	\bool_if:N	<u>2120</u>
		\bool_if:n	<u>2154</u>

- \bool_if:NF 249, 2130, 2245, 2251, 3966, 7487, 8979
- \bool_if:nF 2269, 2278
- \bool_if:NT 2129, 2243, 2249, 3966, 3967, 7086, 8944
- \bool_if:nT 2256, 2265, 14993
- \bool_if:NTF 38, 1365, 2120, 2131, 3952, 8623, 9583
- \bool_if:nTF ... 39, 2143, 2154, 8919, 8929
- \bool_if_exist:c 2153
- \bool_if_exist:cTF 2152
- \bool_if_exist:N 2152
- \bool_if_exist:NF 8640, 8656
- \bool_if_exist:NTF 38, 2134, 2152
- \bool_if_exist_p:c 2152
- \bool_if_exist_p:N 38, 2152
- \bool_if_p:c 2120
- \bool_if_p:N 38, 2120, 2128
- \bool_if_p:n 39, 2115, 2117, 2154, 2156, 2162, 2162, 2235, 2238
- \bool_new:c 2092
- \bool_new:N 37, 2092, 2092, 2093, 2148, 2149, 2150, 2151, 6804, 8564, 8640, 8656, 9492
- \bool_not_p:n 40, 2235, 2235
- .bool_set:c 152
- \bool_set:cn 2114
- .bool_set:N 152
- \bool_set:Nn 38, 2114, 2114, 2118
- \bool_set_eq:cc 2106, 2109
- \bool_set_eq:cN 2106, 2108
- \bool_set_eq:Nc 2106, 2107
- \bool_set_eq:NN 38, 2106, 2106
- \bool_set_false:c 2094
- \bool_set_false:N 37, 264, 2094, 2096, 2103, 7082, 7485, 8585, 8911, 9585
- .bool_set_inverse:c 152
- .bool_set_inverse:N 152
- \bool_set_true:c 2094
- \bool_set_true:N 38, 278, 2094, 2094, 2102, 7100, 7115, 7146, 8580, 8906, 9551, 9622
- \bool_show:c 2132
- \bool_show:N 38, 2132, 2132, 2147
- \bool_show:n 38, 2132, 2135, 2141
- \bool_until_do:cn 2242
- \bool_until_do:Nn 40, 2242, 2244, 2245, 2247
- \bool_until_do:nn .. 41, 2254, 2267, 2272
- \bool_while_do:cn 2242
- \bool_while_do:Nn 40, 2242, 2242, 2243, 2246
- \bool_while_do:nn .. 41, 2254, 2254, 2259
- \bool_xor_p:nn 40, 2236, 2236
- \botmark 424
- \botmarks 650
- \box 632
- \box_clear:c 6568
- \box_clear:N 128, 6568, 6568, 6572, 6575, 6840, 6899, 6948
- \box_clear_new:c 6574
- \box_clear_new:N .. 128, 6574, 6574, 6578
- \box_clip:c 14398
- \box_clip:N 189, 14398, 14398, 14400
- \box_dp:c 6594, 6970
- \box_dp:N 129, 6594, 6595, 6598, 6601, 6969, 7040, 7042, 7059, 7073, 7226, 7244, 7557, 7597, 14181, 14286, 14316, 14336, 14355, 14409, 14416, 14421, 14644
- \box_gclear:c 6568
- \box_gclear:N . 128, 6568, 6570, 6573, 6577
- \box_gclear_new:c 6574
- \box_gclear_new:N . 128, 6574, 6576, 6579
- \box_gset_eq:cc 6580
- \box_gset_eq:cN 6580
- \box_gset_eq:Nc 6580
- \box_gset_eq:NN 128, 6571, 6580, 6582, 6585
- \box_gset_eq_clear:cc 6586
- \box_gset_eq_clear:cN 6586
- \box_gset_eq_clear:Nc 6586
- \box_gset_eq_clear:NN 128, 6586, 6588, 6591
- \box_gset_to_last:c 6642
- \box_gset_to_last:N 131, 6642, 6644, 6647
- \box_ht:c 6594, 6972
- \box_ht:N 130, 6594, 6594, 6597, 6603, 6895, 6943, 6971, 7036, 7038, 7059, 7066, 7225, 7243, 7555, 7596, 14180, 14285, 14315, 14335, 14354, 14426, 14431, 14436, 14641, 14643
- \box_if_empty:cTF 6636
- \box_if_empty:N 6636
- \box_if_empty:NF 6640
- \box_if_empty:NT 6639
- \box_if_empty:NTF 130, 6636, 6641
- \box_if_empty_p:c 6636
- \box_if_empty_p:N 130, 6636, 6638
- \box_if_exist:c 6593
- \box_if_exist:cTF 6592
- \box_if_exist:N 6592
- \box_if_exist:NTF 129, 6575, 6577, 6592, 6680

<code>\box_if_exist_p:c</code>	6592	<code>\box_set_eq:NN</code>	
<code>\box_if_exist_p:N</code>	129 , 6592	128 , 6569 , 6580 , 6580 , 6583 ,
<code>\box_if_horizontal:cTF</code>	6624		6584 , 6958 , 7246 , 7560 , 14441 , 14485
<code>\box_if_horizontal:N</code>	6624	<code>\box_set_eq_clear:cc</code>	6586
<code>\box_if_horizontal:NF</code>	6630	<code>\box_set_eq_clear:cN</code>	6586
<code>\box_if_horizontal:NT</code>	6629	<code>\box_set_eq_clear:Nc</code>	6586
<code>\box_if_horizontal:NTF</code> ..	130 , 6624 , 6631	<code>\box_set_eq_clear:NN</code>	
<code>\box_if_horizontal_p:c</code>	6624	128 , 6586 , 6586 , 6589 , 6590
<code>\box_if_horizontal_p:N</code> ..	130 , 6624 , 6628	<code>\box_set_ht:cn</code>	6600
<code>\box_if_vertical:cTF</code>	6624	<code>\box_set_ht:Nn</code> ..	130 , 6600 , 6602 , 6606 ,
<code>\box_if_vertical:N</code>	6626		7225 , 7243 , 7554 , 14206 , 14377 ,
<code>\box_if_vertical:NF</code>	6634		14430 , 14439 , 14470 , 14483 , 14626
<code>\box_if_vertical:NT</code>	6633	<code>\box_set_to_last:c</code>	6642
<code>\box_if_vertical:NTF</code> ...	130 , 6624 , 6635	<code>\box_set_to_last:N</code>	
<code>\box_if_vertical_p:c</code>	6624	131 , 6642 , 6642 , 6645 , 6646
<code>\box_if_vertical_p:N</code> ...	130 , 6624 , 6632	<code>\box_set_wd:cn</code>	6600
<code>\box_log:c</code>	6659	<code>\box_set_wd:Nn</code> ..	130 , 6600 , 6604 , 6608 ,
<code>\box_log:cnn</code>	6659		7227 , 7245 , 7558 , 14208 , 14389 , 14629
<code>\box_log:N</code>	131 , 6659 , 6659 , 6661	<code>\box_show:c</code>	6653
<code>\box_log:Nnn</code> ..	132 , 6659 , 6660 , 6662 , 6673	<code>\box_show:cnn</code>	6653
<code>\box_move_down:nn</code> ..	129 , 6613 , 6619 ,	<code>\box_show:N</code>	131 , 6653 , 6653 , 6655
	14413 , 14421 , 14456 , 14463 , 14623	<code>\box_show:Nnn</code> ..	131 , 6653 , 6654 , 6656 , 6658
<code>\box_move_left:nn</code>	129 , 6613 , 6613	<code>\box_trim:cnnnn</code>	14401
<code>\box_move_right:nn</code>	129 , 6613 , 6615	<code>\box_trim:Nnnnn</code> ..	190 , 14401 , 14401 , 14443
<code>\box_move_up:nn</code>	129 , 6613 , 6617 ,	<code>\box_use:c</code>	6609
	7264 , 7552 , 14429 , 14436 , 14469 , 14480	<code>\box_use:N</code>	129 , 6609 ,
<code>\box_new:c</code>	6560		6610 , 6612 , 7261 , 7264 , 7342 , 7479 ,
<code>\box_new:N</code>	128 , 6560 ,		7549 , 7552 , 14195 , 14202 , 14210 ,
	6561 , 6567 , 6575 , 6577 , 6648 , 6649 ,		14374 , 14384 , 14393 , 14406 , 14414 ,
	6650 , 6651 , 6652 , 6782 , 6847 , 14165		14422 , 14429 , 14437 , 14449 , 14457 ,
<code>\box_resize:cnn</code>	14280		14463 , 14469 , 14481 , 14624 , 14631
<code>\box_resize:Nnn</code>		<code>\box_use_clear:c</code>	6609
 188 , 14280 , 14280 , 14300 , 14753	<code>\box_use_clear:N</code> ..	129 , 6609 , 6609 , 6611
<code>\box_resize_to_ht_plus_dp:cn</code>	14310	<code>\box_viewport:cnnnn</code>	14444
<code>\box_resize_to_ht_plus_dp:Nn</code>		<code>\box_viewport:Nnnnn</code>	
 189 , 14310 , 14310 , 14329	190 , 14444 , 14444 , 14487
<code>\box_resize_to_wd:cn</code>	14310	<code>\box_wd:c</code>	6594 , 6974
<code>\box_resize_to_wd:Nn</code>		<code>\box_wd:N</code> ..	130 , 6594 , 6596 , 6599 , 6605 ,
 189 , 14310 , 14330 , 14346		6973 , 7038 , 7042 , 7048 , 7053 , 7194 ,
<code>\box_rotate:Nn</code> ..	189 , 14166 , 14166 , 14616		7227 , 7245 , 7262 , 7550 , 7559 , 7598 ,
<code>\box_scale:cnn</code>	14347		14182 , 14287 , 14317 , 14337 , 14356 ,
<code>\box_scale:Nnn</code>			14450 , 14643 , 14648 , 14812 , 14819
 189 , 14347 , 14347 , 14368 , 14775	<code>\boxmaxdepth</code>	594
<code>\box_set_dp:cn</code>	6600	<code>\brokenpenalty</code>	551
<code>\box_set_dp:Nn</code> ..	130 , 6600 , 6600 , 6607 ,	<code>abs</code>	179
	7226 , 7244 , 7556 , 14207 , 14378 ,		
	14416 , 14424 , 14459 , 14464 , 14628		
<code>\box_set_eq:cc</code>	6580		
<code>\box_set_eq:cN</code>	6580		
<code>\box_set_eq:Nc</code>	6580		

C

<code>cc</code>	181
<code>nc</code>	181
<code>pc</code>	181

\c__coffin_corners_prop . 6785, 6785, 6786, 6787, 6788, 6789, 6851, 6988	\c__int_from_roman_m_int 4029
\c__coffin_poles_prop 6790, 6790, 6792, 6793, 6794, 6796, 6797, 6798, 6799, 6800, 6801, 6853, 6990	\c__int_from_roman_V_int 4029
\c__fp_big_leading_shift_int 9851, 9851, 12365, 12375, 12385	\c__int_from_roman_v_int 4029
\c__fp_big_middle_shift_int 9851, 9852, 12367, 12377, 12387, 12396, 12399, 12402	\c__int_from_roman_X_int 4029
\c__fp_big_trailing_shift_int 9851, 9853, 12409	\c__int_from_roman_x_int 4029
\c__fp_Bigg_leading_shift_int 9858, 9858, 12156, 12173	\c__iow_wrap_end_marker_tl .. 9498, 9557
\c__fp_Bigg_middle_shift_int 9858, 9859, 12159, 12162, 12176, 12179	\c__iow_wrap_indent_marker_tl 9498, 9518
\c__fp_Bigg_trailing_shift_int 9858, 9860, 12165, 12182	\c__iow_wrap_marker_tl 9498, 9500, 9509, 9567, 9612
\c__fp_leading_shift_int 9845, 9845, 12337, 13136, 13414	\c__iow_wrap_newline_marker_tl 9498, 9532
\c__fp_ln_i_fixed_tl 12587, 12587	\c__iow_wrap_unindent_marker_tl 9498, 9520
\c__fp_ln_ii_fixed_tl 12587, 12588	\c__keys_code_root_tl 8555, 8555, 8699, 8985, 8987, 8999, 9005, 9010, 9123
\c__fp_ln_iii_fixed_tl ... 12587, 12589	\c__keys_props_root_tl 8557, 8557, 8591, 8627, 8634, 8762, 8764, 8766, 8768, 8770, 8772, 8774, 8776, 8778, 8780, 8782, 8784, 8786, 8788, 8790, 8792, 8794, 8796, 8798, 8800, 8802, 8804, 8806, 8808, 8810, 8812, 8814, 8816, 8818, 8820, 8822, 8824, 8826, 8828, 8830, 8832, 8834, 8836, 8838, 8840, 8842, 8844, 8846, 8848, 8850, 8852, 8854, 8856, 8858, 8860, 8862, 8864, 8866, 8868, 8870, 8872, 8874, 8876, 8878, 9078, 9080, 9115
\c__fp_ln_iv_fixed_tl 12587, 12590	\c__keys_value_forbidden_tl . 8558, 8558
\c__fp_ln_ix_fixed_tl 12587, 12594	\c__keys_value_required_tl .. 8558, 8559
\c__fp_ln_vi_fixed_tl 12587, 12591	\c__keys_vars_root_tl 8555, 8556, 8704, 8705, 8706, 8709, 8752, 8946, 8948, 8951, 8959, 9069, 9072, 9074, 9085, 9105
\c__fp_ln_vii_fixed_tl ... 12587, 12592	\c__max_constdef_int 3466, 3470, 3490, 3494
\c__fp_ln_viii_fixed_tl .. 12587, 12593	\c__msg_kernel_bug_more_text_tl 8390, 8398, 8402
\c__fp_ln_x_fixed_tl 12587, 12595, 12804, 12811	\c__msg_kernel_bug_text_tl 8390, 8393, 8400
\c__fp_max_exponent_int 9746, 9746, 9752, 9758, 9774, 9775, 12460, 12527, 12857, 12884, 13171, 13584, 13633	\c__msg_more_text_prefix_tl 7642, 7643, 7681, 7690, 7881
\c__fp_middle_shift_int . 9845, 9846, 12339, 12342, 12345, 12348, 13138, 13141, 13144, 13147, 13416, 13418	\c__msg_text_prefix_tl 7642, 7642, 7646, 7679, 7688, 7861, 7872, 7887, 7904, 7912, 7918, 8297, 8324
\c__fp_one_fixed_tl 12274, 12274, 12758, 12916, 13172, 13196	\c__tl_act_lowercase_tl 15080, 15085, 15103
\c__fp_trailing_shift_int 9845, 9847, 12351, 13150, 13420	\c__tl_act_uppercase_tl 15080, 15080, 15095
\c__int_from_roman_C_int 4029	\c__tl_rescan_marker_tl 4728, 4736, 4747, 4765
\c__int_from_roman_c_int 4029	\c_active_char_token 3316, 3317
\c__int_from_roman_D_int 4029	
\c__int_from_roman_d_int 4029	
\c__int_from_roman_I_int 4029	
\c__int_from_roman_i_int 4029	
\c__int_from_roman_L_int 4029	
\c__int_from_roman_l_int 4029	
\c__int_from_roman_M_int 4029	

- \c_alignment_tab_token 3310, 3311
- \c_alignment_token 53, 2706, 2712, 2762, 3311
- \c_catcode_active_tl 53, 2721, 2723, 2800, 3317
- \c_catcode_letter_token 53, 2706, 2718, 2790, 3313
- \c_catcode_other_space_tl 166, 9493, 9496, 9508, 9510, 9512, 9533
- \c_catcode_other_token 53, 2706, 2719, 2795, 3314
- \c_code_cctab . . . 186, 14090, 14092, 14093
- \c_document_cctab 187, 14090, 14098, 14107, 14110
- \c_e_fp 174, 13849, 13849
- \c_eight 73, 2634, 2666, 4028, 4093, 4098, 9937, 10643, 10675, 12850, 13222, 13235, 13361
- \c_eleven 73, 2640, 2672, 4093, 4101, 11889, 11964, 12005
- \c_empty_box 131, 6569, 6571, 6648, 6648, 7339
- \c_empty_clist 121, 5836, 5836, 5935, 5950, 5972, 5988
- \c_empty_coffin 139, 6963, 6963, 6964, 7340
- \c_empty_prop 127, 6282, 6282
- \c_empty_seq . . 112, 5402, 5402, 5552, 5560
- \c_empty_tl 102, 3841, 4598, 4614, 4616, 4641, 4641, 4850, 5402, 5836, 6282
- \c_false_bool 21, 937, 976, 1016, 1021, 1065, 1066, 1087, 1311, 1318, 1487, 1489, 1498, 1510, 1925, 2092, 2097, 2101, 2210, 2212, 2216, 2239, 3941, 3946, 3955
- \c_fifteen 73, 2648, 2680, 4093, 4104, 11048, 11185
- \c_five 73, 2628, 2660, 4093, 4097, 10190, 11180, 12203, 12857
- \c_four 73, 2626, 2658, 4093, 4096, 9626, 9632, 11181, 12424, 12465, 12558, 12630, 12809, 13458
- \c_fourteen 73, 2646, 2678, 4093, 4103, 11185
- \c_group_begin_token 53, 2706, 2706, 2747, 5221, 5256, 6700, 6748
- \c_group_end_token 53, 2706, 2707, 2752, 6705, 6706, 6756
- \c_inf_fp 174, 9741, 9743, 11005, 11568, 12032, 12112, 12826, 13060, 13064, 13085, 13353
- \c_initex_cctab 187, 14090, 14099
- \c_job_name_tl 102, 4642, 4655, 4659
- \c_keys_code_root_tl 9123
- \c_letter_token 3310, 3313
- \c_log_iow . . . 166, 9410, 9410, 9475, 9476
- \c_luatex_is_engine_bool 1599, 1600
- \c_math_shift_token 3310, 3312
- \c_math_subscript_token 53, 2706, 2716, 2780
- \c_math_superscript_token 53, 2706, 2714, 2775
- \c_math_toggle_token 53, 2706, 2710, 2757, 3312
- \c_max_dim 82, 4402, 4403, 4493, 14695, 14696, 14697, 14698, 14715
- \c_max_int 73, 4111, 4111, 6654, 6660
- \c_max_muskip 88, 4554, 4555
- \c_max_register_int 73, 824, 825, 827, 8176, 14010, 14041
- \c_max_skip 85, 4492, 4493
- \c_minus_inf_fp 174, 9741, 9744, 11560, 12033, 12115, 12603, 13355
- \c_minus_one 73, 812, 813, 816, 817, 1172, 1332, 3468, 3529, 4093, 4748, 4749, 7805, 9360, 9403, 9410, 9453, 9499, 9525, 10902, 11271, 11559, 11721, 12287, 12506, 13215, 13224, 13250, 13409, 14002
- \c_minus_zero_fp . . 173, 9741, 9742, 12029
- \c_msg_coding_error_text_tl 7578, 7589, 7695, 7695, 8111, 8119, 8145, 8153, 8162, 8169, 8183, 8192, 8199, 8206, 8213, 8220, 8228, 9028, 9043, 9050, 9111
- \c_msg_continue_text_tl 7695, 7700, 7739
- \c_msg_critical_text_tl 7695, 7702, 7875
- \c_msg_fatal_text_tl . . 7695, 7704, 7864
- \c_msg_help_text_tl . . . 7695, 7706, 7743
- \c_msg_no_info_text_tl . 7695, 7708, 7738
- \c_msg_on_line_text_tl . 7695, 7713, 7730
- \c_msg_return_text_tl 7695, 7711, 7714, 8114, 8122, 8129, 8406
- \c_msg_trouble_text_tl 7695, 7721
- \c_nan_fp 9741, 9745, 10045, 10053, 10125, 10433, 10440, 10491, 10512, 11006, 11095, 13033
- \c_nine 73, 2636, 2668, 4093, 4099, 10361, 10388, 10549, 10570, 10597, 10611, 10646, 10673, 10736, 10752, 10797, 10813, 10824, 10876, 11178, 11179, 12015

- \c_one 73, 2620, 2652, 3442, 3527, 4093,
4093, 5009, 5733, 6187, 6191, 6657,
8039, 8045, 9546, 9774, 9967, 10168,
10220, 10221, 10224, 10301, 10326,
10559, 10623, 10633, 10798, 10818,
10832, 11089, 11126, 11376, 11381,
11388, 11567, 11683, 11695, 11699,
11815, 11840, 11879, 11933, 11936,
11958, 11962, 11982, 12006, 12022,
12093, 12126, 12215, 12217, 12244,
12299, 12439, 12475, 12516, 12545,
12614, 12660, 12755, 12851, 12853,
12898, 13192, 13202, 13217, 13237,
13240, 13253, 13327, 13448, 13540,
13601, 13606, 13865, 14010, 14503,
14899, 14911, 15134, 15145, 15186
- \c_one_degree_fp 174, 11008, 13851, 13852
- \c_one_fp 173, 11009,
11011, 11228, 11233, 11238, 11249,
11605, 12820, 13028, 13075, 13277,
13307, 13380, 13386, 13849, 13850
- \c_one_hundred 73, 4108, 4108
- \c_one_thousand
. 73, 4108, 4109, 12480, 12489, 12494
- \c_other_cctab
.... 187, 14090, 14100, 14108, 14118
- \c_other_char_token 3310, 3314
- \c_parameter_token
..... 53, 2706, 2713, 2766, 2769
- \c_pdftex_is_engine_bool 1599, 1601
- \c_pi_fp 174, 11007, 13851, 13851
- \c_seven 73, 812, 822, 2632, 2664, 4093,
10682, 11258, 11303, 11910, 12017
- \c_six 73, 812, 821, 2630, 2662, 4093
- \c_sixteen 73, 812,
819, 1174, 4026, 4093, 9297, 9360,
9385, 9411, 9453, 9871, 9941, 10277,
11060, 11082, 11125, 12872, 13223,
13228, 13652, 13654, 13660, 13701
- \c_space_tl
102, 4661, 4661, 5110, 6183, 6192,
7731, 9254, 9544, 9558, 9628, 13792
- \c_space_token 53, 2706,
2717, 2785, 3177, 5222, 5257, 15185
- \c_str_cctab
187, 14090, 14101, 14109, 14123, 14144
- \c_string_cctab 14143, 14144
- \c_ten 73, 2638, 2670, 3866,
4093, 4100, 11176, 11177, 12014, 12631
- \c_ten_thousand
. 73, 4108, 4110, 12280, 12358, 12419
- \c_term_ior 166, 9297, 9297, 9308, 9366, 9459
- \c_term_iow 166, 9410, 9411, 9423, 9477, 9478
- \c_thirteen 73, 2644, 2676, 4093, 4102
- \c_thirty_two 73, 4105, 4105, 11175
- \c_three .. 73, 2624, 2656, 4093, 4095,
9776, 10446, 10474, 10945, 11325,
11329, 11339, 11719, 12009, 13342
- \c_token_A_int 3017, 3052
- \c_true_bool 21,
976, 1065, 1065, 1086, 1329, 1488,
1499, 1509, 2095, 2099, 2122, 2211,
2213, 2217, 2240, 3941, 3946, 3959
- \c_twelve ... 73, 812, 823, 2341, 2356,
2642, 2674, 2865, 4093, 11108, 11111
- \c_two 73, 2622, 2654, 3442, 4024,
4093, 4094, 8040, 9771, 9775, 11345,
11479, 11635, 11866, 11997, 12012,
12015, 12017, 12197, 12261, 12311,
12460, 12527, 12763, 12884, 13171,
13202, 13282, 13312, 13435, 13458,
13469, 13511, 13540, 13704, 14008,
14040, 14048, 15183, 15184, 15185
- \c_two_hundred_fifty_five 73, 4106, 4106
- \c_two_hundred_fifty_six . 73, 4106, 4107
- \c_undefined_fp 13860, 13860
- \c_xetex_is_engine_bool 1599, 1602
- \c_zero 73, 812,
820, 976, 986, 991, 996, 1073, 1075,
1516, 1521, 1526, 1554, 1673, 1682,
2312, 2315, 2316, 2317, 2318, 2319,
2320, 2321, 2322, 2323, 2324, 2334,
2336, 2511, 2518, 2535, 2562, 2571,
2618, 2650, 2834, 3437, 3496, 3497,
3547, 3555, 3704, 3707, 3839, 4093,
4392, 4459, 5102, 5293, 5294, 6666,
6667, 8285, 9705, 9707, 9776, 9936,
9961, 10172, 10176, 10178, 10182,
10186, 10199, 10211, 10222, 10223,
10232, 10236, 10240, 10243, 10248,
10252, 10284, 10289, 10465, 10470,
10480, 10528, 10595, 10664, 10717,
10765, 10779, 10807, 10856, 10884,
10909, 10935, 11128, 11153, 11358,
11471, 11650, 11655, 11931, 12215,
12785, 12789, 12803, 12861, 12867,
12881, 13071, 13083, 13101, 13124,
13156, 13231, 13232, 13246, 13248,

- 13267, 13297, 13366, 13650, 13675,
 14499, 14514, 14534, 14898, 15133
 \c_zero_dim 82, 4198, 4402,
 4402, 4492, 6715, 6729, 7096, 7099,
 7102, 7111, 7114, 7117, 7126, 7133,
 7190, 7195, 7202, 14392, 14413,
 14424, 14429, 14439, 14452, 14456,
 14464, 14466, 14469, 14473, 14483
 \c_zero_fp 173, 9741, 9741, 9784,
 11010, 11229, 11234, 11239, 11248,
 11607, 11862, 12028, 12829, 13055,
 13088, 13739, 13802, 13817, 13818,
 14184, 14186, 14191, 14379, 14763
 \c_zero_muskip 88, 4513, 4554, 4554
 \c_zero_skip
 .. 85, 4422, 4492, 4492, 15028, 15029
 \catcode 4, 5, 6, 7, 10, 112, 113, 114, 115,
 116, 117, 118, 119, 120, 126, 127,
 128, 129, 130, 131, 132, 133, 236,
 237, 238, 239, 240, 241, 242, 243, 636
 \catcodetable 729
 \CatcodeTableIniTeX 14099
 \CatcodeTableLaTeX 14098
 \CatcodeTableOther 14100
 \CatcodeTableString 14101
 \cctab_begin:N
 186, 14036, 14036, 14056, 14059, 14070
 \cctab_end:
 186, 14036, 14046, 14061, 14064, 14071
 \cctab_gset:Nn 186, 14075, 14075, 14084,
 14087, 14093, 14110, 14118, 14123
 \cctab_new:N
 . 186, 14005, 14005, 14019, 14022,
 14028, 14092, 14107, 14108, 14109
 \char 490, 2876
 \char_gset_active:Npn . 197, 15152, 15166
 \char_gset_active:Npx 15152, 15167
 \char_gset_active_eq:NN 197, 15152, 15169
 \char_make_active:N 3319, 3335
 \char_make_active:n 3319, 3353
 \char_make_alignment:N 3319
 \char_make_alignment:n 3319
 \char_make_alignment_tab:N 3324
 \char_make_alignment_tab:n 3342
 \char_make_begin_group:N 3321
 \char_make_begin_group:n 3339
 \char_make_comment:N 3319, 3336
 \char_make_comment:n 3319, 3354
 \char_make_end_group:N 3322
 \char_make_end_group:n 3340
 \char_make_end_line:N 3319, 3325
 \char_make_end_line:n 3319, 3343
 \char_make_escape:N 3319, 3320
 \char_make_escape:n 3319, 3338
 \char_make_group_begin:N 3319
 \char_make_group_begin:n 3319
 \char_make_group_end:N 3319
 \char_make_group_end:n 3319
 \char_make_ignore:N 3319, 3331
 \char_make_ignore:n 3319, 3349
 \char_make_invalid:N 3319, 3337
 \char_make_invalid:n 3319, 3355
 \char_make_letter:N 3319, 3333
 \char_make_letter:n 3319, 3351
 \char_make_math_shift:N 3323
 \char_make_math_shift:n 3341
 \char_make_math_subscript:N . 3319, 3329
 \char_make_math_subscript:n . 3319, 3347
 \char_make_math_superscript:N 3319, 3327
 \char_make_math_superscript:n 3319, 3345
 \char_make_math_toggle:N 3319
 \char_make_math_toggle:n 3319
 \char_make_other:N 3319, 3334
 \char_make_other:n 3319, 3352
 \char_make_parameter:N 3319, 3326
 \char_make_parameter:n 3319, 3344
 \char_make_space:N 3319, 3332
 \char_make_space:n 3319, 3350
 \char_set_active:Npn . 197, 15152, 15164
 \char_set_active:Npx 15152, 15165
 \char_set_active_eq:NN 197, 15152, 15168
 \char_set_catcode:nn 51, 253,
 254, 255, 256, 257, 258, 259, 260,
 261, 2611, 2611, 2618, 2620, 2622,
 2624, 2626, 2628, 2630, 2632, 2634,
 2636, 2638, 2640, 2642, 2644, 2646,
 2648, 2650, 2652, 2654, 2656, 2658,
 2660, 2662, 2664, 2666, 2668, 2670,
 2672, 2674, 2676, 2678, 2680, 2865
 \char_set_catcode:w 3282, 3283, 3290, 3292
 \char_set_catcode_active:N 50,
 2617, 2643, 2722, 2729, 2730, 2731,
 2732, 2733, 2734, 3335, 7764, 15153
 \char_set_catcode_active:n . 50, 2649,
 2675, 3353, 8452, 8453, 14116, 15158
 \char_set_catcode_alignment:N
 50, 2617, 2625, 2711, 3324
 \char_set_catcode_alignment:n
 50, 271, 2649, 2657, 3342

```

\char_set_catcode_comment:N ..... 10345, 10399, 10400, 11174, 13567,
..... 50, 2617, 2645, 3336 15173, 15174, 15175, 15176, 15177
\char_set_catcode_comment:n ..... \char_set_catcode_other:n 50, 270, 275,
..... 50, 2649, 2677, 3354 2649, 2673, 3352, 14114, 14121, 14126
\char_set_catcode_end_line:N ..... \char_set_catcode_parameter:N .....
..... 50, 2617, 2627, 3325 ..... 50, 2617, 2629, 3326
\char_set_catcode_end_line:n ..... \char_set_catcode_parameter:n .....
..... 50, 2649, 2659, 3343 ..... 50, 2649, 2661, 3344
\char_set_catcode_escape:N ..... \char_set_catcode_space:N .....
..... 50, 2617, 2617, 3320 ..... 50, 2617, 2637, 3332
\char_set_catcode_escape:n ..... \char_set_catcode_space:n .. 50, 276,
..... 50, 2649, 2649, 3338 2649, 2669, 3350, 14112, 14113, 14127
\char_set_catcode_group_begin:N .... \char_set_lccode:nn ..... 51, 2681,
..... 50, 2617, 2619, 3321 2687, 2821, 2822, 2823, 2859, 2860,
\char_set_catcode_group_begin:n .... 2861, 2862, 2863, 3020, 3021, 3022,
..... 50, 2649, 2651, 3339 7761, 7762, 7763, 8271, 8272, 8273,
\char_set_catcode_group_end:N ..... 8274, 8454, 8455, 9495, 10346, 15160
..... 50, 2617, 2621, 3322 \char_set_lccode:w 3282, 3285, 3296, 3298
\char_set_catcode_group_end:n ..... \char_set_mathcode:nn ... 52, 2681, 2681
..... 50, 2649, 2653, 3340 \char_set_mathcode:w 3282, 3284, 3293, 3295
\char_set_catcode_ignore:N ..... \char_set_sfcode:nn ..... 52, 2681, 2699
..... 50, 2617, 2635, 3331 \char_set_sfcode:w 3282, 3287, 3302, 3304
\char_set_catcode_ignore:n ..... \char_set_uccode:nn ..... 52, 2681, 2693
..... 50, 268, 269, 2649, 2667, 3349 \char_set_uccode:w 3282, 3286, 3299, 3301
\char_set_catcode_invalid:N ..... \char_show_value_catcode:n 51, 2611, 2615
..... 50, 2617, 2647, 3337 \char_show_value_catcode:w .. 3289, 3291
\char_set_catcode_invalid:n ..... \char_show_value_lccode:n 51, 2681, 2691
..... 50, 2649, 2679, 3355 \char_show_value_lccode:w ... 3289, 3297
\char_set_catcode_letter:N ..... \char_show_value_mathcode:n .....
..... 50, 2617, 2639, 3333, 10929, 52, 2681, 2685
11119, 11184, 11199, 11200, 11322, \char_show_value_mathcode:w . 3289, 3294
11355, 11373, 11611, 11612, 13382 \char_show_value_sfcode:n 53, 2681, 2703
\char_set_catcode_letter:n ..... \char_show_value_sfcode:w ... 3289, 3303
..... 50, 272, 274, 2649, 2671, 3351 \char_show_value_uccode:n 52, 2681, 2697
\char_set_catcode_math_subscript:N . \char_show_value_uccode:w ... 3289, 3300
..... 50, 2617, 2633, 2715, 3330 \char_tmp:NN ..... 15154, 15164,
..... 15165, 15166, 15167, 15168, 15169
\char_set_catcode_math_subscript:n . \char_value_catcode:n 51, 253, 254, 255,
..... 50, 2649, 2665, 3348, 14115 256, 257, 258, 259, 260, 261, 2611, 2613
\char_set_catcode_math_superscript:N \char_value_catcode:w ..... 3289, 3290
..... 50, 2617, 2631, 3328, 8270 \char_value_lccode:n .... 51, 2681, 2689
\char_set_catcode_math_superscript:n \char_value_lccode:w ..... 3289, 3296
..... 50, 273, 2649, 2663, 3346 \char_value_mathcode:n .. 52, 2681, 2683
\char_set_catcode_math_toggle:N .... \char_value_mathcode:w ..... 3289, 3293
..... 50, 2617, 2623, 2709, 3323 \char_value_sfcode:n .... 52, 2681, 2701
\char_set_catcode_math_toggle:n .... \char_value_sfcode:w ..... 3289, 3302
..... 50, 2649, 2655, 3341 \char_value_uccode:n .... 52, 2681, 2695
\char_set_catcode_other:N ..... \char_value_uccode:w ..... 3289, 3299
..... 50, 2617, 2641, 2819, 2820, \chardef ..... 122, 135, 138, 283, 325
3019, 3334, 9494, 10343, 10344, \chk_if_free_cs:N ..... 1619, 1620

```

.choice:	152	\clist_gpush:Nv	6001, 6010
.choices:nn	152	\clist_gpush:Nx	6001, 6012
.choices:on	152	\clist_gput_left:cn	5907, 6013
.choices:Vn	152	\clist_gput_left:co	5907, 6015
.choices:xn	152	\clist_gput_left:cV	5907, 6014
\cleaders	508	\clist_gput_left:cx	5907, 6016
\clist_clear:c	5841, 5842	\clist_gput_left:Nn	
\clist_clear:N 114, 5907, 5909, 5918, 5919, 6009	
... 113, 5841, 5841, 6024, 8895, 14550		\clist_gput_left:No	5907, 6011
\clist_clear_new:c	5845, 5846	\clist_gput_left:Nv	5907, 6010
\clist_clear_new:N	113, 5845, 5845	\clist_gput_left:Nx	5907, 6012
\clist_concat:ccc	5857	\clist_gput_right:cn	5920
\clist_concat:NNN		\clist_gput_right:co	5920
... 114, 5857, 5857, 5870, 5908, 5921		\clist_gput_right:cV	5920
\clist_const:cn	14577	\clist_gput_right:cx	5920
\clist_const:cx	14577	\clist_gput_right:Nn	
\clist_const:Nn	191, 14577, 14577, 14579	... 114, 5920, 5922, 5931, 5932	
\clist_const:Nx	14577	\clist_gput_right:No	5920
\clist_count:c	6169, 6265	\clist_gput_right:Nv	5920
\clist_count:N	118,	\clist_gput_right:Nx	5920
6169, 6169, 6177, 6198, 6264, 14492		\clist_gremove_all:cn	6034
\clist_count:n ... 6169, 6178, 6266, 14522		\clist_gremove_all:Nn	
\clist_display:c	6247, 6249	... 115, 6034, 6036, 6064, 6245	
\clist_display:N	6247, 6248	\clist_gremove_duplicates:c	6018
\clist_gclear:c	5841, 5844	\clist_gremove_duplicates:N	
\clist_gclear:N	113, 5841, 5843, 14552	... 115, 6018, 6020, 6033	
\clist_gclear_new:c	5845, 5848	\clist_gremove_element:Nn ... 6243, 6245	
\clist_gclear_new:N	113, 5845, 5847	.clist_gset:c	152
\clist_gconcat:ccc	5857	\clist_gset:cn	5901
\clist_gconcat:NNN		\clist_gset:co	5901
... 114, 5857, 5859, 5871, 5910, 5923		\clist_gset:cV	5901
\clist_get:cn	5933, 6241	\clist_gset:cx	5901
\clist_get:cNTF	5970	.clist_gset:N	152
\clist_get:NN		\clist_gset:Nn ... 114, 5901, 5903, 5906	
... 120, 5933, 5933, 5943, 5970, 6240		\clist_gset:No	5901, 6253
\clist_get:NNF	5980	\clist_gset:Nv	5901
\clist_get:NNT	5979	\clist_gset:Nx	5901
\clist_get:NNTF	120, 5970, 5981	\clist_gset_eq:cc	5849, 5856
\clist_gpop:cn	5944	\clist_gset_eq:cN	5849, 5855
\clist_gpop:cNTF	5970	\clist_gset_eq:Nc	5849, 5854
\clist_gpop:NN 120, 5944, 5946, 5969, 5984		\clist_gset_eq:NN . 114, 5849, 5853, 6021	
\clist_gpop:NNF	5999	\clist_gset_from_seq:cc	14549
\clist_gpop:NNT	5998	\clist_gset_from_seq:cN	14549
\clist_gpop:NNTF	120, 5970, 6000	\clist_gset_from_seq:Nc	14549
\clist_gpush:cn	6001, 6013	\clist_gset_from_seq:NN	
\clist_gpush:co	6001, 6015	... 191, 14549, 14551, 14575, 14576	
\clist_gpush:cV	6001, 6014	\clist_gtrim_spaces:c	6251
\clist_gpush:cx	6001, 6016	\clist_gtrim_spaces:N .. 6251, 6253, 6255	
\clist_gpush:Nn	121, 6001, 6009	\clist_if_empty:c	6066
\clist_gpush:No	6001, 6011	\clist_if_empty:cTF	6065

\clist_if_empty:N	6065	\clist_map_break:	
\clist_if_empty:n	14580	117, 6098, 6103, 6112, 6116,
\clist_if_empty:NF		6131, 6149, 6165, 6165, 6166, 6168	
.....	5866, 6051, 6094, 6124, 6143	\clist_map_break:n	118, 6165, 6167
\clist_if_empty:NTF	115, 6065, 8247	\clist_map_function:cN	6092
\clist_if_empty:nTF	191, 14580	\clist_map_function:NN	116, 6092,
\clist_if_empty_p:c	6065	6092, 6107, 6174, 6227, 14945, 14955	
\clist_if_empty_p:N	115, 6065	\clist_map_function:nN	
\clist_if_empty_p:n	191, 14580	. 6108, 6108, 8722, 9089, 14950, 14960	
\clist_if_eq:cc	6261	\clist_map_inline:cN	6122
\clist_if_eq:ccTF	6257	\clist_map_inline:Nn	116, 6025,
\clist_if_eq:cN	6260	6122, 6122, 6138, 6140, 9277, 9292	
\clist_if_eq:cNTF	6257	\clist_map_inline:nn	6122, 6135, 8684, 8737
\clist_if_eq:Nc	6259	\clist_map_variable:cNn	6141
\clist_if_eq:NcTF	6257	\clist_map_variable:NNn	
\clist_if_eq:NN	6258	117, 6141, 6141, 6155, 6164
\clist_if_eq:NNTF	6257	\clist_map_variable:nNn	6141, 6152
\clist_if_eq_p:cc	6257	\clist_new:c	5839, 5840
\clist_if_eq_p:cN	6257	\clist_new:N	113, 5839,
\clist_if_eq_p:Nc	6257	5839, 6017, 6235, 6236, 6237, 6238	
\clist_if_eq_p:NN	6257	\clist_pop:cN	5944
\clist_if_exist:c	5873	\clist_pop:cNTF	5970
\clist_if_exist:cTF	5872	\clist_pop:NN	120, 5944, 5944, 5968, 5982
\clist_if_exist:N	5872	\clist_pop:NNF	5996
\clist_if_exist:NT	9275	\clist_pop:NNT	5995
\clist_if_exist:NTF	114, 5872, 6196, 9248	\clist_pop:NNTF	120, 5970, 5997
\clist_if_exist_p:c	5872	\clist_push:cN	6001, 6005
\clist_if_exist_p:N	114, 5872	\clist_push:co	6001, 6007
\clist_if_in:cnTF	6067	\clist_push:cV	6001, 6006
\clist_if_in:coTF	6067	\clist_push:cx	6001, 6008
\clist_if_in:cVTF	6067	\clist_push:Nn	121, 6001, 6001
\clist_if_in:Nn	6067	\clist_push:No	6001, 6003
\clist_if_in:nn	6071	\clist_push:NV	6001, 6002
\clist_if_in:NnF	6027, 6085, 6086	\clist_push:Nx	6001, 6004
\clist_if_in:nnF	6090	\clist_put_left:cN	5907, 6005
\clist_if_in:NnT	6083, 6084	\clist_put_left:co	5907, 6007
\clist_if_in:nnT	6089	\clist_put_left:cV	5907, 6006
\clist_if_in:NnTF	116, 6067, 6087, 6088	\clist_put_left:cx	5907, 6008
\clist_if_in:nnTF	6067, 6091, 10012	\clist_put_left:Nn	
\clist_if_in:NoTF	6067	... 114, 5907, 5907, 5916, 5917, 6001	
\clist_if_in:noTF	6067	\clist_put_left:No	5907, 6003
\clist_if_in:NVTF	6067	\clist_put_left:NV	5907, 6002
\clist_if_in:nVTF	6067	\clist_put_left:Nx	5907, 6004
\clist_item:cN	14489	\clist_put_right:cN	5920
\clist_item:Nn	191, 14489, 14489, 14518	\clist_put_right:co	5920
\clist_item:nn	14519, 14519	\clist_put_right:cV	5920
\clist_length:c	6263, 6266	\clist_put_right:cx	5920
\clist_length:N	6263, 6264	\clist_put_right:Nn	
\clist_length:n	6263, 6265	... 114, 5920, 5920, 5929, 5930, 6028	
		\clist_put_right:No	5920

\clist_put_right:NV	5920	\coffin_attach:cnnNnnnn	7221
\clist_put_right:Nx	5920 , 8976	\coffin_attach:Nnncnnnn	7221
\clist_remove_all:cn	6034	\coffin_attach:NnnNnnnn 138 , 7221 , 7221 , 7248
\clist_remove_all:Nn 115 , 6034 , 6034 , 6063 , 6244	\coffin_attach_mark:NnnNnnnn 7221 , 7239 , 7409 , 7430 , 7446
\clist_remove_duplicates:c	6018	\coffin_clear:c	6836
\clist_remove_duplicates:N 115 , 6018 , 6018 , 6032	\coffin_clear:N ...	136 , 6836 , 6836 , 6844
\clist_remove_element:Nn	6243 , 6244	\coffin_display_handles:cn	7452
.clist_set:c	152	\coffin_display_handles:Nn 139 , 7452 , 7452 , 7537
\clist_set:cn	5901	\coffin_dp:c	6969 , 6970
\clist_set:co	5901	\coffin_dp:N	138 , 6969 , 6969 , 14751 , 14777
\clist_set:cV	5901	\coffin_ht:c	6969 , 6972
\clist_set:cx	5901	\coffin_ht:N	139 , 6969 , 6971 , 14751 , 14777
.clist_set:N	152	\coffin_if_exist:cTF	6813
\clist_set:Nn 114 , 5901 , 5901 , 5905 , 5908 , 5910 , 5921 , 5923 , 6073 , 6137 , 6154 , 6231	\coffin_if_exist:N	6813
\clist_set:No	5901 , 6252	\coffin_if_exist:NF	6825
\clist_set:NV	5901	\coffin_if_exist:NT	6824
\clist_set:Nx	5901	\coffin_if_exist:NTF	136 , 6813 , 6826 , 6829
\clist_set_eq:cc	5849 , 5852	\coffin_if_exist_p:c	6813
\clist_set_eq:cN	5849 , 5851	\coffin_if_exist_p:N ...	136 , 6813 , 6823
\clist_set_eq:Nc	5849 , 5850	\coffin_join:cnnccnnnn	7184
\clist_set_eq:NN ..	114 , 5849 , 5849 , 6019	\coffin_join:cnnNnnnn	7184
\clist_set_from_seq:cc	14549	\coffin_join:Nnncnnnn	7184
\clist_set_from_seq:cN	14549	\coffin_join:NnnNnnnn	138 , 7184 , 7184 , 7220
\clist_set_from_seq:Nc	14549	\coffin_mark_handle:cnnn	7397
\clist_set_from_seq:NN 191 , 14549 , 14549 , 14573 , 14574	\coffin_mark_handle:Nnnn 139 , 7397 , 7397 , 7451
\clist_show:c	6224 , 6249	\coffin_new:c	6845
\clist_show:N 121 , 6224 , 6224 , 6232 , 6234 , 6248	\coffin_new:N 136 , 6845 , 6845 , 6855 , 6963 , 6965 , 6966 , 6967 , 6968 , 7345 , 7346 , 7347
\clist_show:n	121 , 6224 , 6229	\coffin_resize:cnn	14745
\clist_top:cN	6239 , 6241	\coffin_resize:Nnn	191 , 14745 , 14745 , 14756
\clist_top:NN	6239 , 6240	\coffin_rotate:cn	14603
\clist_trim_spaces:c	6251	\coffin_rotate:Nn	191 , 14603 , 14603 , 14637
\clist_trim_spaces:N ...	6251 , 6252 , 6254	\coffin_scale:cnn	14771
\clist_use:c	6268 , 6270	\coffin_scale:Nnn	192 , 14771 , 14771 , 14785
\clist_use:cn	6194	\coffin_set_eq:cc	6954
\clist_use:cnnn	6194	\coffin_set_eq:cN	6954
\clist_use:N	6268 , 6269	\coffin_set_eq:Nc	6954
\clist_use:Nn	119 , 6194 , 6221 , 6223	\coffin_set_eq:NN	136 , 6954 , 6954 , 6962 , 7218 , 7237 , 7266 , 7473
\clist_use:Nnnn	119 , 6194 , 6194 , 6214 , 6222	\coffin_set_horizontal_pole:cnn ..	7006
\closein	384	\coffin_set_horizontal_pole:Nnn 137 , 7006 , 7006 , 7030
\closeout	379	\coffin_set_vertical_pole:cnn ...	7006
\clubpenalties	692	\coffin_set_vertical_pole:Nnn 137 , 7006 , 7017 , 7031
\clubpenalty	519		
.code:n	152		
\coffin_attach:cnnccnnnn	7221		

<code>\coffin_show_structure:c</code>	7562	2558, 2577, 2578, 2579, 2580, 3465,
<code>\coffin_show_structure:N</code>		3486, 3498, 3499, 3504, 3505, 3507,
.	139 , 7562 , 7562, 7574	3508, 3510, 3511, 3522, 3523, 3524,
<code>\coffin_typeset:cnnnn</code>	7337	3525, 3534, 3535, 3536, 3537, 3541,
<code>\coffin_typeset:Nnnnn</code> 138 , 7337 , 7337 , 7344		3542, 3971, 4191, 4197, 4200, 4201,
<code>\coffin_wd:c</code>	6969 , 6974	4206, 4207, 4213, 4214, 4216, 4217,
<code>\coffin_wd:N</code> 139 , 6969 , 6973, 14748, 14781		4219, 4220, 4224, 4225, 4229, 4230,
<code>\color</code>	7405 , 7417 , 7460 , 7500	4397, 4399, 4415, 4421, 4424, 4425,
<code>\color_ensure_current:</code> 140 , 6863, 6907,		4430, 4431, 4437, 4438, 4440, 4441,
6930 , 7614 , 7615 , 7619 , 7624 , 7629		4443, 4444, 4448, 4449, 4453, 4454,
<code>\color_group_begin:</code>		4479, 4486, 4487, 4489, 4505, 4511,
140 , 6862 , 6884 , 6907, 6930, 7608 , 7608		4515, 4516, 4521, 4522, 4528, 4529,
<code>\color_group_end:</code>		4531, 4532, 4534, 4535, 4539, 4540,
140 , 6865 , 6886 , 6910, 6933, 7608 , 7609		4544, 4545, 4549, 4551, 4582, 4583,
<code>\columnwidth</code>	6882 , 6928	4584, 4585, 4600, 4611, 4612, 4617,
<code>\copy</code>	576	4618, 4623, 4624, 4637, 4638, 4674,
<code>\count</code>	627	4675, 4676, 4677, 4678, 4679, 4696,
<code>\countdef</code>	326	4697, 4698, 4699, 4700, 4701, 4702,
<code>\cr</code>	351	4703, 4720, 4721, 4722, 4723, 4724,
<code>\crrcr</code>	352	4725, 4726, 4727, 4768, 4769, 4770,
<code>\cs:w</code>	17 , 780 , 782, 797, 799,	4771, 4784, 4785, 4786, 4787, 4830,
858, 1119, 1147, 1355, 1404, 1637,		4831, 4836, 4837, 4840, 4841, 4842,
1676, 1690, 1692, 1696, 1697, 1698,		4843, 4844, 4845, 4846, 4847, 4856,
1733, 1739, 1759, 1761, 1766, 1773,		4857, 4858, 4859, 4868, 4869, 4870,
1774, 1836, 1840, 1870, 2070, 2289,		4871, 4891, 4892, 4893, 4894, 4913,
2291, 3544, 5326, 9840, 9869, 10422,		4914, 4915, 4922, 4923, 4924, 4942,
10457, 10655, 10687, 10950, 10978,		4957, 4969, 4985, 4992, 4998, 5010,
11046, 11106, 12647, 12915, 13200		5011, 5034, 5035, 5133, 5144, 5145,
<code>\cs_end:</code> 17 , 780 , 783, 797, 799, 803, 858,		5164, 5174, 5206, 5207, 5208, 5209,
1113, 1119, 1141, 1147, 1288, 1355,		5306, 5319, 5320, 5359, 5453, 5454,
1404, 1637, 1676, 1690, 1692, 1696,		5465, 5466, 5467, 5468, 5473, 5474,
1697, 1698, 1733, 1739, 1759, 1761,		5475, 5476, 5493, 5494, 5519, 5520,
1766, 1773, 1774, 1836, 1840, 1870,		5544, 5545, 5546, 5547, 5548, 5549,
2070, 2286, 2292, 2293, 2294, 2295,		5578, 5590, 5591, 5608, 5635, 5636,
2297, 2299, 2301, 2303, 2305, 2307,		5641, 5642, 5643, 5644, 5645, 5646,
2309, 3544, 5326, 9840, 9877, 9998,		5655, 5656, 5657, 5658, 5659, 5660,
10422, 10457, 10659, 10691, 10953,		5661, 5662, 5663, 5664, 5665, 5666,
10984, 10999, 12647, 12915, 13200		5683, 5712, 5723, 5724, 5734, 5755,
<code>\cs_generate_from_arg_count:cNnn</code> . . .		5769, 5807, 5826, 5870, 5871, 5905,
.	1337 , 1346	5906, 5916, 5917, 5918, 5919, 5929,
<code>\cs_generate_from_arg_count:Ncnn</code> . . .		5930, 5931, 5932, 5943, 5968, 5969,
.	1337 , 1348	5979, 5980, 5981, 5995, 5996, 5997,
<code>\cs_generate_from_arg_count:NNnn</code> . . .		5998, 5999, 6000, 6032, 6033, 6063,
.	15 , 1337 , 1337, 1347, 1349, 1367	6064, 6083, 6084, 6085, 6086, 6087,
<code>\cs_generate_variant:Nn</code> 28 , 1884 , 1884,		6088, 6089, 6090, 6091, 6107, 6140,
2073, 2074, 2075, 2076, 2077, 2078,		6164, 6177, 6214, 6223, 6234, 6254,
2079, 2080, 2081, 2093, 2102, 2103,		6255, 6328, 6329, 6330, 6331, 6338,
2104, 2105, 2118, 2119, 2128, 2129,		6339, 6358, 6359, 6360, 6361, 6382,
2130, 2131, 2147, 2246, 2247, 2252,		6383, 6384, 6385, 6386, 6387, 6400,
2253, 2524, 2525, 2555, 2556, 2557,		6402, 6404, 6406, 6420, 6421, 6451,

6452, 6453, 6454, 6455, 6456, 6457,	
6458, 6468, 6469, 6470, 6471, 6472,	
6473, 6488, 6489, 6500, 6510, 6522,	
6523, 6529, 6530, 6531, 6534, 6567,	
6572, 6573, 6578, 6579, 6584, 6585,	
6590, 6591, 6597, 6598, 6599, 6606,	
6607, 6608, 6611, 6612, 6628, 6629,	
6630, 6631, 6632, 6633, 6634, 6635,	
6638, 6639, 6640, 6641, 6646, 6647,	
6655, 6658, 6661, 6673, 6691, 6692,	
6697, 6698, 6703, 6704, 6722, 6723,	
6733, 6734, 6739, 6740, 6745, 6746,	
6751, 6752, 6767, 6768, 6823, 6824,	
6825, 6826, 6844, 6855, 6872, 6902,	
6919, 6953, 6962, 7030, 7031, 7032,	
7220, 7248, 7344, 7451, 7537, 7574,	
8052, 8327, 8414, 8415, 8416, 8577,	
8653, 8669, 8701, 8761, 8888, 8889,	
8902, 8903, 9077, 9241, 9258, 9309,	
9312, 9322, 9323, 9324, 9351, 9369,	
9424, 9427, 9444, 9462, 9468, 9471,	
9474, 10128, 13560, 13615, 13683,	
13716, 13720, 13766, 13803, 13810,	
13811, 13812, 13815, 13816, 13819,	
13820, 13825, 13826, 13833, 13834,	
13835, 13836, 13848, 13888, 13889,	
13890, 13891, 13900, 13901, 13902,	
13903, 13904, 13905, 13910, 13911,	
13954, 13955, 13966, 13967, 13992,	
13995, 13996, 13999, 14300, 14329,	
14346, 14368, 14400, 14443, 14487,	
14518, 14573, 14574, 14575, 14576,	
14579, 14637, 14756, 14785, 14860,	
14861, 14877, 14890, 14915, 14940,	
14941, 14962, 14963, 14964, 14965,	
14966, 14967, 14985, 14986, 15150	
\cs_gnew:cpn	1578
\cs_gnew:cpx	1582
\cs_gnew:Npn	1570
\cs_gnew:Npx	1574
\cs_gnew_eq:cc	1590
\cs_gnew_eq:cN	1588
\cs_gnew_eq:Nc	1589
\cs_gnew_eq:NN	1587
\cs_gnew_nopar:cpn	1577
\cs_gnew_nopar:cpx	1581
\cs_gnew_nopar:Npn	1569
\cs_gnew_nopar:Npx	1573
\cs_gnew_protected:cpn	1580
\cs_gnew_protected:cpx	1584
\cs_gnew_protected:Npn	1572
\cs_gnew_protected:Npx	1576
\cs_gnew_protected_nopar:cpn	1579
\cs_gnew_protected_nopar:cpx	1583
\cs_gnew_protected_nopar:Npn	1571
\cs_gnew_protected_nopar:Npx	1575
\cs_gset:cn	1399
\cs_gset:cpn	1250,
1252, 4961, 6127, 6493, 7688, 7690	
\cs_gset:cpx	1250, 1253
\cs_gset:cx	1399
\cs_gset:Nn	14, 1350
\cs_gset:Npn	
12, 843, 845, 1236, 1252, 5687, 15166	
\cs_gset:Npx	
... 843, 847, 1237, 1253, 5692, 15167	
\cs_gset:Nx	1350
\cs_gset_eq:cc ...	1268, 1275, 2113, 4632
\cs_gset_eq:cN	1268, 1274,
1293, 2112, 4630, 5697, 8541, 8543	
\cs_gset_eq:Nc	1268, 1273, 2111, 4631, 5702
\cs_gset_eq:NN	16,
1268, 1272, 1273, 1274, 1275, 1285,	
1492, 1493, 1494, 1495, 1496, 1497,	
1498, 1499, 1503, 1504, 1505, 1506,	
1507, 1508, 1509, 1510, 2099, 2101,	
2110, 4598, 4629, 9366, 9459, 15169	
\cs_gset_nopar:cn	1399
\cs_gset_nopar:cpn	1242, 1246
\cs_gset_nopar:cpx	1242, 1247
\cs_gset_nopar:cx	1399
\cs_gset_nopar:Nn	14, 1350
\cs_gset_nopar:Npn	12, 843, 843,
846, 850, 854, 1234, 1246, 3729, 7728	
\cs_gset_nopar:Npx	843, 844, 848,
852, 856, 1235, 1247, 3736, 4604,	
4609, 4669, 4671, 4673, 4689, 4691,	
4693, 4695, 4713, 4715, 4717, 4719	
\cs_gset_nopar:Nx	1350
\cs_gset_protected:cn	1399
\cs_gset_protected:cpn	1262, 1264
\cs_gset_protected:cpx	1262, 1265
\cs_gset_protected:cx	1399
\cs_gset_protected:Nn	15, 1350
\cs_gset_protected:Npn	
... 12, 843, 853, 1240, 1264, 7659	
\cs_gset_protected:Npx	843, 855, 1241, 1265
\cs_gset_protected:Nx	1350
\cs_gset_protected_nopar:cn	1399
\cs_gset_protected_nopar:cpx	1256, 1258

\cs_gset_protected_nopar:cpx	1256 , 1259	\cs_if_exist_use:NF	1157
\cs_gset_protected_nopar:cx	1399	\cs_if_exist_use:NT	1159
\cs_gset_protected_nopar:Nn	15 , 1350	\cs_if_exist_use:NTF	17 , 17 , 188 , 1155 , 1155
\cs_gset_protected_nopar:Npn	13 , 843 , 849 , 1238 , 1258	\cs_if_free:c	1139
\cs_gset_protected_nopar:Npx	843 , 851 , 1239 , 1259	\cs_if_free:cT	2065
\cs_gset_protected_nopar:Nx	1350	\cs_if_free:cTF	1127 , 7925
\cs_gundefine:c	1594	\cs_if_free:N	1127
\cs_gundefine:N	1593	\cs_if_free:NF	1195 , 1205
\cs_if_eq:ccF	1447	\cs_if_free:NTF	22 , 1127 , 2026 , 9699 , 9701
\cs_if_eq:ccT	1446	\cs_if_free_p:c	1127
\cs_if_eq:ccTF	1431 , 1445	\cs_if_free_p:N	22 , 1127
\cs_if_eq:cNF	1439	\cs_meaning:c	800 , 801 , 811
\cs_if_eq:cNT	1438	\cs_meaning:N	16 , 787 , 789 , 808 , 1470
\cs_if_eq:cNTF	1431 , 1437 , 7894	\cs_new:cn	1399
\cs_if_eq:NcF	1443	\cs_new:cpn	1250 , 1254 , 1578 , 2192 , 2194 , 2199 , 2293 , 2294 , 2295 , 2296 , 2298 , 2300 , 2302 , 2304 , 2306 , 2308 , 2310 , 2315 , 2316 , 2317 , 2318 , 2319 , 2320 , 2321 , 2322 , 2323 , 2324 , 3582 , 3594 , 3596 , 3598 , 3600 , 3602 , 3604 , 3606 , 4297 , 4299 , 4301 , 4303 , 9885 , 10359 , 10436 , 11042 , 11102 , 11120 , 11147 , 11186 , 11226 , 11231 , 11236 , 11241 , 11602 , 11711 , 12255 , 13024
\cs_if_eq:NcT	1442	\cs_new:cpx	1250 , 1255 , 1582
\cs_if_eq:NcTF	1431 , 1441	\cs_new:cx	1399
\cs_if_eq:NN	1431	\cs_new:Nn	6 , 13 , 1350
\cs_if_eq:NNF	1439 , 1443 , 1447	\cs_new:Npn	1 , 11 , 1226 , 1236 , 1254 , 1325 , 1327 , 1474 , 1524 , 1532 , 1537 , 1543 , 1548 , 1554 , 1557 , 1566 , 1567 , 1570 , 1630 , 1631 , 1632 , 1633 , 1634 , 1635 , 1636 , 1638 , 1640 , 1652 , 1658 , 1664 , 1675 , 1677 , 1684 , 1685 , 1687 , 1689 , 1691 , 1693 , 1700 , 1702 , 1707 , 1712 , 1718 , 1724 , 1730 , 1736 , 1742 , 1749 , 1756 , 1763 , 1770 , 1805 , 1806 , 1811 , 1813 , 1818 , 1828 , 1830 , 1832 , 1833 , 1835 , 1837 , 1843 , 1849 , 1851 , 1858 , 1865 , 1867 , 1869 , 1870 , 1871 , 1873 , 1878 , 1952 , 1968 , 1973 , 1982 , 1995 , 2010 , 2068 , 2162 , 2172 , 2174 , 2176 , 2178 , 2182 , 2201 , 2219 , 2225 , 2231 , 2235 , 2236 , 2242 , 2244 , 2248 , 2250 , 2254 , 2262 , 2267 , 2275 , 2281 , 2288 , 2290 , 2292 , 2345 , 2351 , 2360 , 2365 , 2494 , 2500 , 2508 , 2515 , 2526 , 2532 , 2598 , 2599 , 2601 , 2613 , 2683 , 2689 , 2695 , 2701 , 2832 , 2887 , 2905 , 2929 , 2947 , 2965 , 2983 , 2994
\cs_if_eq:NNT	1438 , 1442 , 1446		
\cs_if_eq:NNTF	21 , 1431 , 1437 , 1441 , 1445		
\cs_if_eq_p:cc	1431 , 1444		
\cs_if_eq_p:cN	1431 , 1436		
\cs_if_eq_p:Nc	1431 , 1440		
\cs_if_eq_p:NN	21 , 1431 , 1436 , 1440 , 1444		
\cs_if_exist:c	1111 , 2153 , 3513 , 4209 , 4433 , 4524 , 4640 , 5460 , 5873 , 6423 , 6593 , 11422		
\cs_if_exist:cF	9068		
\cs_if_exist:cT	8948		
\cs_if_exist:cTF	1099 , 1164 , 1166 , 1168 , 1170 , 6817 , 7646 , 8421 , 8591 , 8985 , 8999 , 9005 , 9084 , 10508		
\cs_if_exist:N	1099 , 2152 , 3512 , 4208 , 4432 , 4523 , 4639 , 5459 , 5872 , 6422 , 6592 , 11421		
\cs_if_exist:NF	1218 , 13931		
\cs_if_exist:NT	1490 , 1501 , 7628 , 9196 , 9214		
\cs_if_exist:NTF	21 , 1099 , 1156 , 1158 , 1160 , 1162 , 1450 , 6815 , 7622 , 8336 , 9383		
\cs_if_exist_p:c	1099		
\cs_if_exist_p:N	21 , 1099		
\cs_if_exist_use:c	1155 , 1169		
\cs_if_exist_use:cF	1165 , 10010 , 10487		
\cs_if_exist_use:cT	1167		
\cs_if_exist_use:cTF	1155 , 1163		
\cs_if_exist_use:N	1155 , 1161		

3015, 3034, 3041, 3042, 3050, 3059,
3068, 3084, 3156, 3252, 3255, 3264,
3273, 3392, 3394, 3400, 3418, 3426,
3434, 3447, 3449, 3456, 3544, 3551,
3565, 3570, 3576, 3587, 3616, 3621,
3623, 3646, 3654, 3662, 3668, 3674,
3682, 3690, 3696, 3702, 3716, 3750,
3751, 3765, 3771, 3803, 3835, 3837,
3843, 3855, 3863, 3896, 3898, 3900,
3902, 3907, 3912, 3917, 3937, 3938,
3943, 3948, 3972, 3980, 3982, 3991,
3993, 4002, 4004, 4014, 4023, 4025,
4027, 4043, 4052, 4086, 4088, 4129,
4144, 4231, 4236, 4254, 4262, 4264,
4276, 4282, 4305, 4307, 4312, 4314,
4377, 4379, 4381, 4388, 4473, 4476,
4481, 4484, 4546, 4764, 4813, 4824,
4872, 4925, 4926, 4927, 4928, 4931,
4936, 4944, 4952, 4991, 4993, 4999,
5004, 5009, 5012, 5019, 5026, 5028,
5038, 5050, 5058, 5064, 5070, 5074,
5081, 5092, 5101, 5103, 5110, 5116,
5118, 5120, 5134, 5136, 5138, 5146,
5151, 5153, 5165, 5167, 5176, 5182,
5184, 5190, 5236, 5245, 5290, 5358,
5360, 5385, 5395, 5445, 5451, 5477,
5576, 5630, 5671, 5677, 5725, 5733,
5735, 5756, 5758, 5765, 5767, 5825,
5874, 5879, 5880, 5887, 5967, 6060,
6062, 6092, 6101, 6108, 6114, 6121,
6169, 6187, 6194, 6215, 6216, 6219,
6221, 6315, 6436, 6442, 6474, 6480,
7807, 7808, 7809, 7810, 7811, 7812,
7898, 7923, 8053, 8289, 8292, 8301,
8306, 8311, 8316, 8356, 8357, 8361,
8366, 8501, 8721, 8983, 8992, 9644,
9651, 9721, 9722, 9723, 9724, 9725,
9726, 9727, 9747, 9748, 9749, 9755,
9761, 9770, 9772, 9783, 9784, 9785,
9795, 9805, 9815, 9825, 9835, 9838,
9848, 9849, 9854, 9856, 9861, 9863,
9865, 9867, 9879, 9881, 9883, 9909,
9911, 9913, 9914, 9915, 9917, 9919,
9921, 9923, 9925, 9933, 9934, 9950,
9959, 9966, 9968, 9975, 9994, 10113,
10114, 10115, 10116, 10117, 10118,
10119, 10126, 10167, 10169, 10178,
10179, 10188, 10202, 10218, 10229,
10238, 10245, 10256, 10264, 10275,
10280, 10300, 10302, 10313, 10318,
10331, 10339, 10340, 10350, 10355,
10377, 10404, 10417, 10420, 10430,
10443, 10463, 10485, 10498, 10506,
10522, 10530, 10545, 10556, 10567,
10577, 10582, 10591, 10608, 10621,
10626, 10632, 10634, 10641, 10671,
10699, 10704, 10714, 10724, 10734,
10750, 10795, 10811, 10822, 10840,
10851, 10867, 10874, 10892, 10897,
10906, 10911, 10921, 10923, 10930,
10959, 10973, 10988, 10993, 10998,
11054, 11065, 11084, 11087, 11132,
11157, 11201, 11213, 11246, 11251,
11256, 11267, 11282, 11293, 11305,
11323, 11337, 11356, 11370, 11374,
11392, 11394, 11428, 11447, 11452,
11479, 11480, 11488, 11500, 11506,
11512, 11520, 11528, 11534, 11540,
11548, 11556, 11564, 11572, 11596,
11598, 11600, 11613, 11622, 11623,
11648, 11653, 11660, 11661, 11669,
11680, 11687, 11694, 11696, 11704,
11734, 11736, 11746, 11766, 11775,
11789, 11797, 11805, 11812, 11819,
11827, 11837, 11851, 11862, 11863,
11869, 11886, 11893, 11895, 11902,
11907, 11924, 11925, 11926, 11944,
11950, 11960, 11972, 11979, 12001,
12039, 12048, 12067, 12069, 12071,
12080, 12091, 12118, 12131, 12144,
12152, 12169, 12186, 12193, 12201,
12211, 12212, 12221, 12222, 12231,
12241, 12263, 12276, 12277, 12282,
12283, 12296, 12303, 12311, 12312,
12313, 12316, 12324, 12330, 12332,
12334, 12356, 12362, 12372, 12382,
12392, 12405, 12416, 12421, 12422,
12436, 12446, 12458, 12463, 12470,
12477, 12487, 12501, 12502, 12513,
12522, 12533, 12541, 12542, 12548,
12556, 12565, 12574, 12576, 12596,
12610, 12624, 12644, 12657, 12658,
12663, 12676, 12681, 12689, 12694,
12704, 12716, 12745, 12746, 12747,
12749, 12751, 12753, 12767, 12773,
12782, 12801, 12807, 12817, 12836,
12844, 12883, 12885, 12894, 12896,
12910, 12912, 12921, 12923, 12939,
12955, 12971, 12987, 13003, 13019,
13049, 13068, 13096, 13112, 13122,

13133, 13154, 13169, 13174, 13179,
 13181, 13195, 13197, 13212, 13220,
 13244, 13259, 13274, 13289, 13304,
 13319, 13334, 13349, 13359, 13377,
 13379, 13383, 13390, 13400, 13402,
 13411, 13423, 13425, 13441, 13451,
 13467, 13505, 13519, 13550, 13555,
 13556, 13557, 13558, 13571, 13598,
 13611, 13613, 13621, 13647, 13672,
 13681, 13682, 13689, 13699, 13719,
 13726, 13731, 13736, 13752, 13757,
 13759, 13768, 13770, 13772, 13774,
 13863, 13870, 13981, 13993, 14489,
 14497, 14512, 14519, 14527, 14529,
 14543, 14548, 14565, 14572, 14587,
 14593, 14863, 14869, 14878, 14884,
 14892, 14908, 14916, 14925, 14933,
 14980, 15009, 15017, 15020, 15040,
 15053, 15059, 15064, 15076, 15077,
 15078, 15090, 15098, 15106, 15113,
 15114, 15122, 15127, 15142, 15156
 \cs_new:Npx 1226, 1237, 1255, 1574, 6178,
 6188, 8277, 9516, 13714, 13717, 13786
 \cs_new:Nx 1350
 \cs_new_eq:cc 1001, 1268, 1283, 1590
 \cs_new_eq:cN 1268, 1281, 1588, 11099
 \cs_new_eq:Nc 1268, 1282, 1589
 \cs_new_eq:NN 15, 1268, 1276,
 1281, 1282, 1283, 1478, 1479, 1480,
 1481, 1482, 1483, 1484, 1485, 1486,
 1487, 1488, 1489, 1555, 1556, 1565,
 1569, 1570, 1571, 1572, 1573, 1574,
 1575, 1576, 1577, 1578, 1579, 1580,
 1581, 1582, 1583, 1584, 1587, 1588,
 1589, 1590, 1593, 1594, 1597, 1600,
 1601, 1602, 1614, 1615, 1616, 1617,
 1620, 1646, 2092, 2106, 2107, 2108,
 2109, 2110, 2111, 2112, 2113, 2468,
 2469, 2470, 2471, 2472, 2473, 2476,
 2477, 2478, 2594, 2705, 2706, 2707,
 3077, 3078, 3079, 3283, 3284, 3285,
 3286, 3287, 3307, 3308, 3311, 3312,
 3313, 3314, 3317, 3320, 3321, 3322,
 3323, 3324, 3325, 3326, 3327, 3329,
 3331, 3332, 3333, 3334, 3335, 3336,
 3337, 3338, 3339, 3340, 3341, 3342,
 3343, 3344, 3345, 3347, 3349, 3350,
 3351, 3352, 3353, 3354, 3355, 3358,
 3359, 3360, 3361, 3362, 3363, 3364,
 3365, 3366, 3367, 3368, 3369, 3370,
 3371, 3372, 3373, 3383, 3384, 3385,
 3386, 3387, 3489, 3493, 3543, 3629,
 4089, 4090, 4117, 4118, 4119, 4161,
 4167, 4168, 4171, 4181, 4182, 4183,
 4320, 4396, 4398, 4478, 4480, 4483,
 4488, 4548, 4550, 4565, 4568, 4569,
 4625, 4626, 4627, 4628, 4629, 4630,
 4631, 4632, 4943, 4990, 5329, 5330,
 5331, 5332, 5333, 5334, 5335, 5336,
 5339, 5340, 5341, 5342, 5343, 5344,
 5345, 5346, 5349, 5350, 5351, 5352,
 5353, 5356, 5357, 5363, 5366, 5367,
 5368, 5369, 5370, 5402, 5403, 5404,
 5405, 5406, 5407, 5408, 5409, 5410,
 5411, 5412, 5413, 5414, 5415, 5416,
 5417, 5418, 5419, 5420, 5455, 5456,
 5457, 5458, 5770, 5771, 5772, 5773,
 5774, 5775, 5776, 5777, 5778, 5779,
 5780, 5781, 5782, 5783, 5784, 5785,
 5786, 5787, 5788, 5789, 5790, 5791,
 5792, 5793, 5794, 5795, 5813, 5814,
 5817, 5818, 5821, 5822, 5836, 5839,
 5840, 5841, 5842, 5843, 5844, 5845,
 5846, 5847, 5848, 5849, 5850, 5851,
 5852, 5853, 5854, 5855, 5856, 6001,
 6002, 6003, 6004, 6005, 6006, 6007,
 6008, 6009, 6010, 6011, 6012, 6013,
 6014, 6015, 6016, 6240, 6241, 6244,
 6245, 6248, 6249, 6264, 6265, 6266,
 6269, 6270, 6282, 6283, 6284, 6285,
 6286, 6287, 6288, 6289, 6290, 6291,
 6292, 6293, 6294, 6295, 6296, 6297,
 6298, 6299, 6300, 6512, 6513, 6526,
 6543, 6544, 6545, 6546, 6547, 6548,
 6549, 6550, 6594, 6595, 6596, 6609,
 6610, 6621, 6622, 6623, 6705, 6706,
 6707, 6708, 6709, 6710, 6711, 6712,
 6720, 6721, 6758, 6759, 6760, 6761,
 6762, 6763, 6764, 6765, 6766, 6772,
 6969, 6970, 6971, 6972, 6973, 6974,
 7608, 8372, 8375, 8376, 8377, 8378,
 8379, 8973, 9118, 9119, 9120, 9123,
 9297, 9308, 9380, 9410, 9411, 9423,
 9465, 9480, 9691, 9693, 9695, 9710,
 9711, 9844, 10004, 10005, 10006,
 10007, 10201, 10254, 10255, 10429,
 10497, 10922, 13765, 13767, 13802,
 13813, 13814, 13977, 13978, 14098,
 14099, 14100, 14101, 14144, 15008
 \cs_new_nopar:cn 1399

- \cs_new_nopar:cpn
 ... [1242](#), [1248](#), [1577](#), [2210](#), [2211](#),
 [2212](#), [2213](#), [2214](#), [2215](#), [2216](#), [2217](#),
 [11002](#), [11014](#), [11032](#), [11993](#), [12103](#)
 \cs_new_nopar:cpx
 ... [1242](#), [1249](#), [1581](#), [2056](#), [11706](#)
 \cs_new_nopar:cx [1399](#)
 \cs_new_nopar:Nn [13](#), [1350](#)
 \cs_new_nopar:Npn
 ... [11](#), [1226](#), [1234](#), [1248](#), [1335](#),
 [1436](#), [1437](#), [1438](#), [1439](#), [1440](#), [1441](#),
 [1442](#), [1443](#), [1444](#), [1445](#), [1446](#), [1447](#),
 [1512](#), [1569](#), [1778](#), [1779](#), [1780](#), [1781](#),
 [1782](#), [1783](#), [1784](#), [1785](#), [1786](#), [1793](#),
 [1794](#), [1795](#), [1796](#), [1797](#), [1860](#), [1861](#),
 [1862](#), [1863](#), [2333](#), [2335](#), [3081](#), [3082](#),
 [3083](#), [3134](#), [3142](#), [3144](#), [3146](#), [3166](#),
 [3290](#), [3291](#), [3293](#), [3294](#), [3296](#), [3297](#),
 [3299](#), [3300](#), [3302](#), [3303](#), [3922](#), [3923](#),
 [3924](#), [3925](#), [3926](#), [3927](#), [3928](#), [3929](#),
 [3930](#), [3931](#), [3932](#), [3933](#), [3934](#), [3935](#),
 [3936](#), [4122](#), [4814](#), [4819](#), [4950](#), [4986](#),
 [4988](#), [5166](#), [5175](#), [5542](#), [5667](#), [5669](#),
 [6165](#), [6167](#), [6501](#), [6503](#), [7727](#), [8410](#),
 [8411](#), [9479](#), [9642](#), [10124](#), [10376](#),
 [11620](#), [12314](#), [12315](#), [13561](#), [13616](#),
 [13684](#), [13721](#), [14824](#), [14826](#), [15149](#)
 \cs_new_nopar:Npx
 ... [1226](#), [1235](#), [1249](#), [1573](#), [1911](#)
 \cs_new_nopar:Nx [1350](#)
 \cs_new_protected:cn [1399](#)
 \cs_new_protected:cpn
 ... [1262](#), [1266](#), [1580](#), [7822](#), [7823](#),
 [8067](#), [8423](#), [8762](#), [8764](#), [8766](#), [8768](#),
 [8770](#), [8772](#), [8774](#), [8776](#), [8780](#), [8782](#),
 [8784](#), [8786](#), [8788](#), [8790](#), [8792](#), [8794](#),
 [8796](#), [8798](#), [8800](#), [8802](#), [8804](#), [8806](#),
 [8808](#), [8810](#), [8812](#), [8814](#), [8816](#), [8818](#),
 [8820](#), [8822](#), [8824](#), [8826](#), [8828](#), [8830](#),
 [8832](#), [8834](#), [8836](#), [8838](#), [8840](#), [8844](#),
 [8846](#), [8848](#), [8850](#), [8852](#), [8854](#), [8856](#),
 [8858](#), [8860](#), [8862](#), [8864](#), [8866](#), [8868](#),
 [8870](#), [8872](#), [8874](#), [9078](#), [9080](#), [9115](#)
 \cs_new_protected:cpx [1262](#),
 [1267](#), [1584](#), [7832](#), [7834](#), [7836](#), [7838](#),
 [7840](#), [7849](#), [7851](#), [7853](#), [8076](#), [8078](#),
 [8080](#), [8082](#), [8084](#), [8093](#), [8095](#), [8097](#)
 \cs_new_protected:cx [1399](#)
 \cs_new_protected:Nn [13](#), [1350](#)
- \cs_new_protected:Npn
 ... [11](#), [1226](#), [1240](#), [1266](#),
 [1268](#), [1276](#), [1284](#), [1286](#), [1337](#), [1358](#),
 [1363](#), [1448](#), [1465](#), [1572](#), [1647](#), [1823](#),
 [1884](#), [1902](#), [1915](#), [1917](#), [1923](#), [1932](#),
 [2022](#), [2048](#), [2061](#), [2092](#), [2094](#), [2096](#),
 [2098](#), [2100](#), [2114](#), [2116](#), [2132](#), [2141](#),
 [2374](#), [2464](#), [2465](#), [2487](#), [2585](#), [2611](#),
 [2615](#), [2617](#), [2619](#), [2621](#), [2623](#), [2625](#),
 [2627](#), [2629](#), [2631](#), [2633](#), [2635](#), [2637](#),
 [2639](#), [2641](#), [2643](#), [2645](#), [2647](#), [2649](#),
 [2651](#), [2653](#), [2655](#), [2657](#), [2659](#), [2661](#),
 [2663](#), [2665](#), [2667](#), [2669](#), [2671](#), [2673](#),
 [2675](#), [2677](#), [2679](#), [2681](#), [2685](#), [2687](#),
 [2691](#), [2693](#), [2697](#), [2699](#), [2703](#), [2705](#),
 [3089](#), [3095](#), [3112](#), [3114](#), [3116](#), [3130](#),
 [3132](#), [3459](#), [3466](#), [3496](#), [3497](#), [3500](#),
 [3502](#), [3506](#), [3509](#), [3514](#), [3516](#), [3526](#),
 [3528](#), [3538](#), [3732](#), [3744](#), [4091](#), [4185](#),
 [4192](#), [4198](#), [4199](#), [4202](#), [4204](#), [4210](#),
 [4212](#), [4215](#), [4218](#), [4221](#), [4223](#), [4226](#),
 [4228](#), [4400](#), [4409](#), [4416](#), [4422](#), [4423](#),
 [4426](#), [4428](#), [4434](#), [4436](#), [4439](#), [4442](#),
 [4445](#), [4447](#), [4450](#), [4452](#), [4490](#), [4499](#),
 [4506](#), [4512](#), [4514](#), [4517](#), [4519](#), [4525](#),
 [4527](#), [4530](#), [4533](#), [4536](#), [4538](#), [4541](#),
 [4543](#), [4552](#), [4580](#), [4595](#), [4601](#), [4606](#),
 [4613](#), [4615](#), [4619](#), [4621](#), [4633](#), [4635](#),
 [4662](#), [4664](#), [4666](#), [4668](#), [4670](#), [4672](#),
 [4680](#), [4682](#), [4684](#), [4686](#), [4688](#), [4690](#),
 [4692](#), [4694](#), [4704](#), [4706](#), [4708](#), [4710](#),
 [4712](#), [4714](#), [4716](#), [4718](#), [4744](#), [4772](#),
 [4774](#), [4788](#), [4826](#), [4828](#), [4832](#), [4834](#),
 [4958](#), [4967](#), [4970](#), [4978](#), [5030](#), [5032](#),
 [5140](#), [5142](#), [5297](#), [5307](#), [5314](#), [5325](#),
 [5425](#), [5461](#), [5463](#), [5469](#), [5471](#), [5479](#),
 [5481](#), [5483](#), [5495](#), [5497](#), [5499](#), [5550](#),
 [5558](#), [5568](#), [5583](#), [5585](#), [5592](#), [5603](#),
 [5613](#), [5684](#), [5689](#), [5694](#), [5706](#), [5713](#),
 [5802](#), [5838](#), [5861](#), [5901](#), [5903](#), [5911](#),
 [5924](#), [5933](#), [5941](#), [5948](#), [5956](#), [5986](#),
 [6018](#), [6020](#), [6022](#), [6034](#), [6036](#), [6038](#),
 [6076](#), [6122](#), [6135](#), [6141](#), [6152](#), [6157](#),
 [6224](#), [6229](#), [6252](#), [6253](#), [6305](#), [6307](#),
 [6316](#), [6322](#), [6332](#), [6340](#), [6349](#), [6392](#),
 [6412](#), [6490](#), [6505](#), [6517](#), [6561](#), [6568](#),
 [6570](#), [6574](#), [6576](#), [6580](#), [6582](#), [6586](#),
 [6588](#), [6600](#), [6602](#), [6604](#), [6613](#), [6615](#),
 [6617](#), [6619](#), [6642](#), [6644](#), [6653](#), [6659](#),
 [6662](#), [6674](#), [6688](#), [6689](#), [6690](#), [6693](#),

6695, 6699, 6701, 6713, 6715, 6716,	\cs_new_protected_nopar:cn	<u>1399</u>
6718, 6724, 6725, 6726, 6728, 6730,	\cs_new_protected_nopar:cpn . .	<u>1256</u> ,
6732, 6735, 6737, 6741, 6743, 6747,	1260, 1579, 8778, 8842, 8876, 8878	
6749, 6753, 6769, 6827, 6836, 6845,	\cs_new_protected_nopar:cpx . .	<u>1256</u> ,
6856, 6873, 6903, 6920, 6954, 6975,	1261, 1352, 1401, 1583, 2053, 3194	
6985, 6992, 6999, 7006, 7017, 7028,	\cs_new_protected_nopar:cx	<u>1399</u>
7033, 7044, 7078, 7093, 7171, 7184,	\cs_new_protected_nopar:Nn	<u>13</u> , <u>1350</u>
7221, 7239, 7249, 7268, 7273, 7287,	\cs_new_protected_nopar:Npn <u>11</u> , <u>1226</u> ,	
7292, 7302, 7313, 7325, 7337, 7397,	1238, 1243, 1260, 1269, 1270, 1271,	
7444, 7452, 7481, 7530, 7538, 7562,	1272, 1273, 1274, 1275, 1281, 1282,	
7649, 7670, 7675, 7677, 7684, 7686,	1283, 1346, 1348, 1457, 1476, 1571,	
7693, 7734, 7746, 7751, 7768, 7792,	1777, 1787, 1788, 1789, 1790, 1791,	
7798, 7892, 7933, 7954, 7967, 7972,	1792, 1798, 1799, 1800, 1801, 1802,	
7998, 8009, 8011, 8028, 8054, 8056,	1803, 1804, 1864, 2337, 3085, 3087,	
8058, 8060, 8321, 8328, 8330, 8332,	3175, 3518, 3520, 3530, 3532, 3540,	
8334, 8346, 8348, 8382, 8383, 8384,	3545, 3725, 4572, 4574, 4576, 4578,	
8385, 8386, 8387, 8388, 8419, 8459,	4738, 4740, 4742, 4776, 4778, 4780,	
8473, 8483, 8503, 8509, 8511, 8532,	4782, 4910, 4911, 4912, 4976, 5323,	
8538, 8569, 8571, 8578, 8583, 8588,	5421, 5423, 5579, 5581, 5609, 5611,	
8601, 8611, 8621, 8636, 8638, 8654,	5700, 5857, 5859, 5907, 5909, 5920,	
8680, 8696, 8702, 8708, 8710, 8712,	5922, 5944, 5946, 6388, 6390, 6408,	
8714, 8719, 8733, 8749, 8755, 8880,	6410, 6656, 6918, 6952, 7609, 7615,	
8882, 8890, 8892, 8904, 8909, 8914,	7619, 7953, 8007, 8670, 8723, 8963,	
8941, 9009, 9066, 9082, 9096, 9162,	8965, 8974, 9271, 9370, 9463, 9476,	
9182, 9184, 9192, 9225, 9235, 9242,	9478, 9571, 9581, 9597, 9616, 9630,	
9259, 9261, 9266, 9308, 9310, 9313,	9636, 9694, 9696, 10022, 10024,	
9325, 9335, 9352, 9358, 9372, 9397,	10026, 10056, 10058, 10060, 10082,	
9399, 9423, 9425, 9428, 9445, 9451,	10084, 10086, 10090, 10092, 10094,	
9466, 9469, 9472, 9515, 9522, 9564,	13827, 13828, 13829, 13830, 13872,	
9609, 9687, 9698, 9700, 9704, 9706,	13873, 13892, 13893, 13894, 13895,	
9729, 9992, 10008, 10028, 10062,	13896, 13897, 13908, 13909, 13935,	
10088, 10096, 10098, 13801, 13804,	13936, 13937, 13938, 13940, 14046,	
13806, 13808, 13817, 13818, 13821,	14713, 14828, 14830, 14832, 14968,	
13823, 13831, 13837, 13846, 13886,	14969, 14971, 14987, 14989, 14997,	
13898, 13918, 13923, 13929, 13942,	14999, 15180, 15208, 15210, 15212	
13956, 13961, 13986, 13997, 14005,	\cs_new_protected_nopar:Npx	
14036, 14075, 14166, 14178, 14212, <u>1226</u> , 1239, 1261, 1575,	
14223, 14234, 14245, 14256, 14267,	1905, 1909, 2052, 9624, 13874, 13880	
14280, 14301, 14310, 14330, 14347,	\cs_new_protected_nopar:Nx	<u>1350</u>
14369, 14398, 14401, 14444, 14549,	\cs_set:cn	<u>1399</u>
14551, 14553, 14577, 14603, 14638,	\cs_set:cpn	
14650, 14656, 14662, 14674, 14693,	. <u>1250</u> , 1250, 7679, 7681, 8699, 10101	
14702, 14719, 14724, 14732, 14745,	\cs_set:cpx	<u>1250</u> ,
14757, 14771, 14786, 14793, 14799,	1251, 2375, 2379, 2383, 2387, 2393,	
14808, 14815, 14838, 14845, 14856,	2402, 2411, 2420, 2431, 2433, 2435,	
14858, 14942, 14947, 14952, 14957,	2437, 2439, 2441, 2443, 2445, 2449	
14973, 14991, 15001, 15197, 15201	\cs_set:cx	<u>1399</u>
\cs_new_protected:Npx	\cs_set:Nn	<u>14</u> , <u>1350</u>
. <u>1226</u> , 1241, 1267, 1576	\cs_set:Npn	<u>3</u> , <u>11</u> , <u>829</u> , 831, 858, 864,
\cs_new_protected:Nx	865, 866, 867, 868, 869, 870, 871,	

- 872, 873, 874, 875, 876, 877, 878,
 879, 880, 881, 882, 883, 884, 885,
 886, 887, 888, 889, 890, 891, 892,
 1045, 1050, 1055, 1060, 1073, 1074,
 1082, 1090, 1092, 1095, 1097, 1155,
 1157, 1159, 1161, 1163, 1165, 1167,
 1169, 1226, 1242, 1250, 1350, 1399,
 1605, 1606, 1607, 1608, 2460, 2461,
 3186, 3192, 3389, 3402, 3410, 4238,
 4246, 4321, 4329, 4337, 4343, 4349,
 4357, 4365, 4371, 4918, 5036, 6040,
 6078, 6309, 10031, 10039, 10048,
 10065, 10073, 14840, 15154, 15164
 \cs_set:Npx ... [829](#), [833](#), [1251](#), [4797](#), [15165](#)
 \cs_set:Nx [1350](#)
 \cs_set_eq:cc . [999](#), [1268](#), [1271](#), [2109](#), [4628](#)
 \cs_set_eq:cN [1268](#), [1269](#), [2108](#), [4626](#), [9993](#)
 \cs_set_eq:Nc [1268](#), [1270](#), [2107](#), [4627](#)
 \cs_set_eq:NN ... [16](#), [1268](#), [1268](#), [1269](#),
[1270](#), [1271](#), [1272](#), [1279](#), [1905](#), [1920](#),
[2052](#), [2095](#), [2097](#), [2106](#), [2766](#), [3093](#),
[3097](#), [3118](#), [3120](#), [3197](#), [4625](#), [5615](#),
[5616](#), [5628](#), [8896](#), [8898](#), [8900](#), [9532](#),
[9533](#), [9534](#), [14975](#), [14976](#), [14978](#), [15168](#)
 \cs_set_eq:NwN [1610](#), [1611](#)
 \cs_set_nopar:cn [1399](#)
 \cs_set_nopar:cpn [1242](#), [1244](#)
 \cs_set_nopar:cpx [1242](#), [1245](#)
 \cs_set_nopar:cx [1399](#)
 \cs_set_nopar:Nn [14](#), [1350](#)
 \cs_set_nopar:Npn [12](#), [829](#),
[829](#), [831](#), [832](#), [833](#), [835](#), [836](#), [837](#),
[840](#), [857](#), [893](#), [895](#), [1067](#), [1191](#), [1244](#)
 \cs_set_nopar:Npx
 [829](#), [830](#), [834](#), [838](#), [842](#), [861](#),
[1245](#), [1649](#), [1825](#), [3099](#), [3104](#), [3121](#),
[3122](#), [4663](#), [4665](#), [4667](#), [4681](#), [4683](#),
[4685](#), [4687](#), [4705](#), [4707](#), [4709](#), [4711](#),
[9166](#), [9526](#), [9527](#), [9528](#), [9529](#), [9530](#)
 \cs_set_nopar:Nx [1350](#)
 \cs_set_protected:cn [1399](#)
 \cs_set_protected:cpn .. [1262](#), [1262](#), [8422](#)
 \cs_set_protected:cpx
 .. [1262](#), [1263](#), [8428](#), [8430](#), [8432](#), [8434](#)
 \cs_set_protected:cx [1399](#)
 \cs_set_protected:Nn [14](#), [1350](#)
 \cs_set_protected:Npn
 [12](#), [829](#), [839](#), [859](#), [905](#),
[918](#), [923](#), [935](#), [950](#), [967](#), [983](#), [988](#),
[993](#), [1002](#), [1014](#), [1028](#), [1175](#), [1187](#),
[1189](#), [1193](#), [1203](#), [1216](#), [1228](#), [1262](#),
[1295](#), [1316](#), [4465](#), [5530](#), [6931](#), [7819](#),
[8063](#), [8065](#), [8391](#), [10357](#), [11000](#),
[11012](#), [11052](#), [11100](#), [11155](#), [13906](#),
[14019](#), [14028](#), [14056](#), [14070](#), [14084](#)
 \cs_set_protected:Npx ... [829](#), [841](#), [1263](#)
 \cs_set_protected:Nx [1350](#)
 \cs_set_protected_nopar:cn [1399](#)
 \cs_set_protected_nopar:cpn
 [1256](#), [1256](#), [8426](#)
 \cs_set_protected_nopar:cpx . [1256](#), [1257](#)
 \cs_set_protected_nopar:cx [1399](#)
 \cs_set_protected_nopar:Nn [14](#), [1350](#)
 \cs_set_protected_nopar:Npn [12](#), [265](#),
[829](#), [835](#), [839](#), [841](#), [845](#), [847](#), [849](#),
[851](#), [853](#), [855](#), [897](#), [899](#), [901](#), [903](#),
[910](#), [912](#), [914](#), [916](#), [998](#), [1000](#), [1171](#),
[1173](#), [1214](#), [1224](#), [1256](#), [6908](#), [7624](#),
[7629](#), [7778](#), [9475](#), [9477](#), [14061](#), [14071](#)
 \cs_set_protected_nopar:Npx
 [251](#), [829](#), [837](#), [1257](#), [7940](#)
 \cs_set_protected_nopar:Nx [1350](#)
 \cs_show:c [1459](#), [1476](#), [9010](#)
 \cs_show:N [16](#), [1459](#), [1465](#), [1477](#), [5300](#)
 \cs_to_str:N
 .. [4](#), [18](#), [1067](#), [1067](#), [1086](#), [2349](#), [9480](#)
 \cs_undefine:c [1284](#), [1286](#), [1594](#)
 \cs_undefine:N
[16](#), [1284](#), [1284](#), [1593](#), [8101](#), [8102](#), [8103](#)
 \csname [14](#), [33](#), [36](#), [63](#), [69](#), [73](#), [122](#), [135](#),
[138](#), [174](#), [177](#), [183](#), [191](#), [196](#), [198](#),
[208](#), [211](#), [221](#), [289](#), [291](#), [296](#), [298](#), [414](#)
 \currentgrouplevel [666](#)
 \currentgrouptype [667](#)
 \currentifbranch [663](#)
 \currentiflevel [662](#)
 \currentifttype [664](#)
- D**
- dd [181](#)
 nd [181](#)
 \dagger [4134](#), [4140](#)
 \day [622](#)
 \ddagger [4135](#), [4141](#)
 \deadcycles [556](#)
 \def [55](#), [57](#), [140](#), [152](#),
[159](#), [164](#), [210](#), [220](#), [247](#), [280](#), [310](#), [321](#)
 .default:n [153](#)
 .default:o [153](#)
 .default:V [153](#)

- `.default:x` 153
- `\defaultthyphenchar` 606
- `\defaultskewchar` 607
- `\delcode` 637
- `\delimiter` 431
- `\delimiterfactor` 480
- `\delimitershortfall` 479
- `\deprecated` 2464, 2465, 8382, 8383, 8384, 8385, 8386, 8387, 8388
- `\detokenize` 33, 36, 122, 135, 138, 174, 177, 183, 192, 197, 199, 205, 208, 211, 221, 289, 291, 296, 298, 654
- `\dim_abs:n` 77, 4231, 4231, 14303
- `\dim_add:cn` 4221
- `\dim_add:Nn` 77, 4221, 4221, 4223, 4224
- `\dim_case:nnn` 80, 4307, 4307, 4565
- `\dim_compare:n` 4271
- `\dim_compare:nF` 4331, 4346
- `\dim_compare:nNn` 4266
- `\dim_compare:nNnF` 4359, 4374
- `\dim_compare:nNnT` 4351, 4368, 4581, 7190, 7195, 14473
- `\dim_compare:nNnTF` 78, 4266, 4316, 7096, 7099, 7102, 7111, 7114, 7117, 7126, 7133, 7202, 7315, 7327, 14409, 14426, 14452, 14466
- `\dim_compare:nT` 4323, 4340
- `\dim_compare:nTF` 79, 4271
- `\dim_compare_p:n` 79, 4271
- `\dim_compare_p:nNn` 78, 4266
- `\dim_const:cn` 4192
- `\dim_const:Nn` 76, 4192, 4192, 4197, 4402, 4403
- `\dim_do_until:nn` ... 81, 4321, 4343, 4347
- `\dim_do_until:nNnn` .. 80, 4349, 4371, 4375
- `\dim_do_while:nn` ... 81, 4321, 4337, 4341
- `\dim_do_while:nNnn` .. 80, 4349, 4365, 4369
- `\dim_eval:n` 81, 4310, 4377, 4377, 6895, 6943, 7012, 7023, 7040, 7042, 7048, 7059, 7073, 7298, 7299, 14459, 14480, 14728, 14729, 14736, 14737, 14812, 14819
- `\dim_eval:w` 4567, 4568
- `\dim_eval_end:` 4567, 4569
- `\dim_gadd:cn` 4221
- `\dim_gadd:Nn` 77, 4221, 4223, 4225
- `.dim_gset:c` 153
- `\dim_gset:cn` 4210
- `.dim_gset:N` 153
- `\dim_gset:Nn` 77, 4195, 4210, 4212, 4214, 4575, 4579
- `\dim_gset_eq:cc` 4215
- `\dim_gset_eq:cN` 4215
- `\dim_gset_eq:Nc` 4215
- `\dim_gset_eq:NN` 77, 4215, 4218, 4219, 4220
- `\dim_gset_max:cn` 4571
- `\dim_gset_max:Nn` 4571, 4574, 4583
- `\dim_gset_min:cn` 4571
- `\dim_gset_min:Nn` 4571, 4578, 4585
- `\dim_gsub:cn` 4221
- `\dim_gsub:Nn` 77, 4221, 4228, 4230
- `\dim_gzero:c` 4198
- `\dim_gzero:N` ... 76, 4198, 4199, 4201, 4205
- `\dim_gzero_new:c` 4202
- `\dim_gzero_new:N` ... 76, 4202, 4204, 4207
- `\dim_if_exist:c` 4209
- `\dim_if_exist:cTF` 4208
- `\dim_if_exist:N` 4208
- `\dim_if_exist:NnTF` .. 76, 4203, 4205, 4208
- `\dim_if_exist_p:c` 4208
- `\dim_if_exist_p:N` 76, 4208
- `\dim_max:nn` ... 77, 4231, 4238, 14707, 14711
- `\dim_min:nn` 77, 4231, 4246, 14705, 14709, 14722
- `\dim_new:c` 4184
- `\dim_new:N` 76, 4184, 4185, 4191, 4194, 4203, 4205, 4404, 4405, 4406, 4407, 6783, 6805, 6806, 6809, 6810, 6811, 6812, 7385, 7387, 7388, 14157, 14158, 14159, 14160, 14161, 14162, 14163, 14164, 14598, 14599, 14600, 14601, 14602, 14743, 14744
- `\dim_ratio:nn` 78, 4262, 4262
- `.dim_set:c` 153
- `\dim_set:cn` 4210
- `.dim_set:N` 153
- `\dim_set:Nn` 77, 4210, 4210, 4212, 4213, 4573, 4577, 6879, 6925, 7098, 7103, 7113, 7118, 7128, 7135, 7148, 7173, 7193, 7252, 7253, 7255, 7257, 7275, 7276, 7386, 7489, 7490, 7541, 7542, 7543, 7545, 14180, 14181, 14182, 14214, 14225, 14285, 14286, 14287, 14303, 14304, 14306, 14315, 14316, 14317, 14335, 14336, 14337, 14354, 14355, 14356, 14358, 14360, 14362, 14644, 14676, 14684, 14695, 14696, 14697, 14698, 14704,

14706, 14708, 14710, 14715, 14721, 14776, 14778, 14780, 14788, 14790	\displaylimits 466
\dim_set_eq:cc 4215	\displaystyle 444
\dim_set_eq:cN 4215	\displaywidowpenalties 694
\dim_set_eq:Nc 4215	\displaywidowpenalty 455
\dim_set_eq:NN 77, 4215, 4215, 4216, 4217, 6881, 6882, 6927, 6928	\displaywidth 457
\dim_set_max:cn 4571	\divide 334
\dim_set_max:Nn 4571, 4572, 4582	\doublehyphendemerits 524
\dim_set_min:cn 4571	\dp 635
\dim_set_min:Nn 4571, 4576, 4584	\dump 618
\dim_show:c 4398	
\dim_show:N 82, 4398, 4398, 4399	E
\dim_show:n 82, 4400, 4400	\E 15176
\dim_sub:cn 4221	sec 180
\dim_sub:Nn 77, 4221, 4226, 4228, 4229	\edef 34, 110, 124, 171, 173, 188, 208, 286, 293, 322
\dim_to_fp:n 181, 7141, 7143, 7153, 7154, 7155, 7156, 7177, 7178, 7179, 7180, 11034, 13731, 13731, 14218, 14219, 14229, 14230, 14290, 14293, 14294, 14321, 14322, 14340, 14680, 14681, 14688, 14689, 14748, 14751, 14789, 14791, 14857, 14859	deg 180
\dim_to_pt:n 195, 15008, 15008, 15011, 15013, 15014	\else 15, 64, 179, 194, 214, 375
\dim_to_unit:nn 195, 15009, 15009	\else: 24, 766, 769, 805, 971, 1103, 1106, 1115, 1121, 1131, 1134, 1143, 1149, 1290, 1311, 1320, 1331, 1434, 1517, 1522, 1528, 1606, 1670, 1906, 1956, 1957, 1958, 2014, 2017, 2124, 2158, 2187, 2206, 2326, 2328, 2330, 2332, 2504, 2520, 2543, 2551, 2564, 2573, 2748, 2753, 2758, 2763, 2770, 2776, 2781, 2786, 2791, 2796, 2801, 2806, 2811, 2816, 2836, 2844, 2850, 2853, 2892, 2895, 2914, 2917, 2934, 2937, 2952, 2955, 2970, 2973, 3046, 3055, 3063, 3072, 3138, 3152, 3161, 3171, 3181, 3401, 3422, 3438, 3441, 3585, 3612, 3634, 3642, 3893, 4237, 4258, 4269, 4279, 4306, 4461, 4852, 4864, 4877, 4887, 4903, 5158, 5202, 5225, 5241, 5249, 5259, 5281, 5554, 5563, 5937, 5952, 5974, 5990, 6446, 6625, 6627, 6637, 9387, 9390, 9655, 9765, 9774, 9775, 9776, 9789, 9799, 9809, 9873, 9930, 9940, 9945, 9955, 10000, 10192, 10220, 10221, 10270, 10288, 10323, 10327, 10363, 10383, 10387, 10391, 10453, 10471, 10479, 10535, 10539, 10551, 10562, 10572, 10603, 10616, 10651, 10661, 10680, 10693, 10709, 10717, 10719, 10729, 10740, 10756, 10770, 10776, 10781, 10788, 10802, 10806, 10817, 10834, 10846, 10859, 10887, 10938, 10942, 10948, 10966, 11069, 11072, 11092, 11110, 11127, 11166,
\dim_untill_do:nn 81, 4321, 4329, 4334	
\dim_untill_do:nNnn 80, 4349, 4357, 4362	
\dim_use:c 4396	
\dim_use:N 82, 4234, 4240, 4241, 4242, 4248, 4249, 4250, 4274, 4295, 4378, 4384, 4396, 4396, 4397, 4401, 7036, 7038, 7042, 7053, 7066, 7283, 7596, 7597, 7598, 11035, 14641, 14643, 14646, 14648, 14654, 14660, 14669, 14670, 14671, 14797, 14804	
\dim_while_do:nn 81, 4321, 4321, 4326	
\dim_while_do:nNnn 81, 4349, 4349, 4354	
\dim_zero:c 4198	
\dim_zero:N 76, 4198, 4198, 4199, 4200, 4203, 7089, 7090, 14183, 14288, 14318, 14338, 14357	
\dim_zero_new:c 4202	
\dim_zero_new:N 76, 4202, 4202, 4206	
\dimen 628	
\dimendef 327	
\dimexpr 681	
\directlua 16, 730	
\discretionary 491	
\displayindent 456	

- 11191, 11207, 11219, 11261, 11272,
- 11279, 11317, 11331, 11347, 11363,
- 11380, 11384, 11432, 11443, 11464,
- 11467, 11470, 11473, 11484, 11493,
- 11496, 11578, 11581, 11588, 11606,
- 11620, 11637, 11717, 11720, 11728,
- 11740, 11751, 11757, 11770, 11783,
- 11823, 11857, 11877, 11914, 11932,
- 11935, 11940, 11954, 11989, 12007,
- 12010, 12013, 12016, 12075, 12148,
- 12216, 12217, 12226, 12269, 12441,
- 12450, 12517, 12605, 12616, 12632,
- 12640, 12698, 12777, 12788, 12793,
- 12827, 12840, 12856, 12860, 12863,
- 13034, 13041, 13063, 13086, 13101,
- 13105, 13127, 13158, 13161, 13186,
- 13189, 13217, 13225, 13230, 13236,
- 13239, 13249, 13252, 13270, 13285,
- 13300, 13315, 13330, 13345, 13354,
- 13368, 13371, 13460, 13461, 13471,
- 13514, 13603, 13696, 13703, 13707,
- 13740, 13745, 13866, 13871, 15193
- \emergencystretch 539
- \end 106, 413
- \EndCatcodeRegime 14071
- \endcsname 14, 33, 36, 63, 69, 73, 122, 135,
- 138, 174, 177, 183, 192, 197, 199,
- 208, 211, 221, 289, 291, 296, 298, 415
- \endgroup 13, 62, 68, 72, 348
- \endinput 87, 387
- \endL 702
- \endlinechar 121, 134, 244, 429
- \endR 704
- \eqno 449
- \errhelp 91, 395
- \errmessage 105, 389
- \ERROR 2460, 2461
- \errorcontextlines 396
- \errorstopmode 410
- \escapechar 428
- \etex_beginL:D 701
- \etex_beginR:D 703
- \etex_botmarks:D 650
- \etex_clubpenalties:D 692
- \etex_currentgrouplevel:D 666
- \etex_currentgroupstype:D 667
- \etex_currentifbranch:D 663
- \etex_currentiflevel:D 662
- \etex_currentifttype:D 664
- \etex_detokenize:D 654, 946, 1010, 1892, 4990, 4991
- \etex_dimexpr:D 681, 4182
- \etex_displaywidowpenalties:D 694
- \etex_endL:D 702
- \etex_endR:D 704
- \etex_eTeXrevision:D 646
- \etex_eTeXversion:D 645
- \etex_everyeof:D 706, 4747
- \etex_firstmarks:D 649
- \etex_fontcharhp:D 674
- \etex_fontcharht:D 673
- \etex_fontcharic:D 676
- \etex_fontcharwd:D 675
- \etex_glueexpr:D 682, 4435, 4446, 4451,
- 4470, 4477, 4482, 4485, 4491, 13734
- \etex_glueshrink:D 685, 15025
- \etex_glueshrinkorder:D 687
- \etex_gluestretch:D 684, 15024
- \etex_gluestretchorder:D 686
- \etex_gluetomu:D 688
- \etex_ifcsname:D 643, 781
- \etex_ifdefined:D 642, 780, 824
- \etex_iffontchar:D 672
- \etex_interactionmode:D 670, 6666, 6669, 6670
- \etex_interlinepenalties:D 691
- \etex_lastlinefit:D 690
- \etex_lastnodetype:D 671
- \etex_marks:D 647
- \etex_middle:D 695
- \etex_muexpr:D 683, 4526, 4537, 4542, 4547, 4553
- \etex_mutogluue:D 689
- \etex_numexpr:D 680, 3384
- \etex_pagediscards:D 698
- \etex_parshapedimen:D 679
- \etex_parshapeindent:D 677
- \etex_parshapelength:D 678
- \etex_predisplaydirection:D 705
- \etex_protected:D 707, 831, 833, 835, 836,
- 837, 838, 840, 842, 850, 852, 854, 856
- \etex_readline:D 657, 9404
- \etex_savinghyphcodes:D 696
- \etex_savingvdiscards:D 697
- \etex_scantokens:D 655, 4758
- \etex_showgroups:D 668
- \etex_showifs:D 669

<code>\etex_showtokens:D</code>	2193, 2196, 2200, 2284, 2347, 2348,
... 656, 4092, 4401, 4491, 4553, 8353	2362, 2497, 2503, 2505, 2512, 2519,
<code>\etex_splitbotmarks:D</code>	2521, 2529, 2536, 2829, 2848, 2873,
<code>\etex_splitdiscards:D</code>	2882, 2898, 2920, 2940, 2958, 2976,
<code>\etex_splitfirstmarks:D</code>	2989, 3000, 3010, 3030, 3053, 3054,
<code>\etex_TeXeTstate:D</code>	3056, 3062, 3065, 3137, 3139, 3149,
<code>\etex_topmarks:D</code>	3150, 3151, 3153, 3159, 3160, 3162,
<code>\etex_tracingassigns:D</code>	3169, 3170, 3172, 3178, 3179, 3182,
<code>\etex_tracinggroups:D</code>	3259, 3268, 3277, 3396, 3401, 3404,
<code>\etex_tracingifs:D</code>	3405, 3412, 3413, 3429, 3430, 3451,
<code>\etex_tracingnesting:D</code>	3452, 3562, 3567, 3572, 3590, 3592,
<code>\etex_tracingscantokens:D</code>	3865, 3893, 3904, 3914, 4047, 4092,
<code>\etex_unexpanded:D</code>	4233, 4237, 4240, 4241, 4248, 4249,
786, 1869, 1872, 1875, 1880, 3578,	4273, 4278, 4291, 4294, 4383, 4401,
5122, 5148, 5169, 15042, 15092, 15100	4469, 4491, 4553, 4645, 4652, 4756,
<code>\etex_unless:D</code>	4757, 4808, 4816, 4821, 4862, 4874,
<code>\etex_widowpenalties:D</code>	4875, 4991, 5095, 5122, 5152, 5155,
<code>\eTeXrevision</code>	5156, 5157, 5159, 5172, 5178, 5186,
<code>\eTeXversion</code>	5197, 5216, 5238, 5248, 5251, 5273,
<code>\everycr</code>	5274, 5275, 5293, 5294, 5295, 5534,
<code>\everydisplay</code>	5572, 5584, 5594, 5595, 5619, 5620,
<code>\everyeof</code>	5621, 5673, 5742, 5743, 5746, 5747,
<code>\everyhbox</code>	5938, 5953, 5975, 5991, 6048, 6056,
<code>\everyjob</code>	6061, 6201, 6202, 6205, 6206, 6312,
<code>\everymath</code>	6483, 7787, 8282, 8283, 8284, 8285,
<code>\everypar</code>	8296, 8353, 8354, 8467, 8469, 8625,
<code>\everyvbox</code>	8711, 8717, 9135, 9147, 9647, 9654,
<code>\exhyphenpenalty</code>	9657, 9764, 9766, 9767, 9777, 9778,
<code>\exp_after:wN</code> .. 32, 784, 784, 797, 799,	9779, 9788, 9790, 9798, 9800, 9808,
804, 806, 894, 896, 940, 953, 970,	9810, 9817, 9818, 9819, 9820, 9821,
972, 1019, 1024, 1031, 1071, 1075,	9822, 9827, 9828, 9829, 9830, 9831,
1084, 1085, 1114, 1116, 1119, 1142,	9832, 9833, 9887, 9889, 9916, 9920,
1144, 1147, 1289, 1291, 1299, 1319,	9929, 9938, 9939, 9946, 9953, 9956,
1321, 1355, 1404, 1470, 1561, 1630,	9977, 10045, 10053, 10070, 10079,
1637, 1639, 1642, 1643, 1650, 1654,	10125, 10168, 10204, 10205, 10206,
1655, 1660, 1661, 1666, 1671, 1673,	10267, 10268, 10271, 10282, 10286,
1676, 1684, 1686, 1688, 1690, 1692,	10293, 10294, 10305, 10306, 10315,
1695, 1696, 1697, 1701, 1704, 1709,	10322, 10324, 10325, 10333, 10352,
1714, 1715, 1716, 1720, 1721, 1722,	10362, 10381, 10382, 10384, 10385,
1726, 1727, 1728, 1732, 1733, 1734,	10389, 10390, 10392, 10393, 10406,
1738, 1739, 1740, 1744, 1745, 1746,	10407, 10409, 10411, 10412, 10413,
1747, 1751, 1752, 1753, 1754, 1758,	10414, 10424, 10425, 10433, 10440,
1759, 1760, 1765, 1766, 1767, 1768,	10447, 10448, 10450, 10451, 10454,
1772, 1773, 1774, 1775, 1808, 1809,	10455, 10456, 10457, 10465, 10466,
1812, 1815, 1816, 1820, 1821, 1829,	10475, 10476, 10478, 10480, 10481,
1831, 1832, 1834, 1836, 1839, 1840,	10482, 10491, 10501, 10502, 10512,
1845, 1846, 1850, 1853, 1854, 1855,	10524, 10525, 10527, 10533, 10537,
1859, 1866, 1868, 1869, 1870, 1872,	10550, 10552, 10560, 10571, 10573,
1875, 1880, 1888, 1889, 1890, 1891,	10579, 10584, 10586, 10588, 10594,
1904, 1907, 1928, 1935, 1955, 2070,	10595, 10599, 10601, 10613, 10614,

10636, 10638, 10644, 10647, 10649,
10653, 10658, 10663, 10664, 10674,
10675, 10677, 10678, 10681, 10685,
10690, 10701, 10707, 10718, 10720,
10727, 10728, 10730, 10731, 10738,
10744, 10754, 10800, 10803, 10815,
10826, 10827, 10828, 10829, 10831,
10833, 10835, 10836, 10837, 10843,
10844, 10854, 10857, 10858, 10861,
10863, 10864, 10870, 10878, 10879,
10880, 10881, 10883, 10885, 10894,
10900, 10914, 10915, 10917, 10918,
10936, 10937, 10939, 10940, 10946,
10947, 10949, 10952, 10963, 10964,
10965, 10967, 10968, 10969, 10975,
10976, 11003, 11034, 11035, 11044,
11045, 11046, 11056, 11057, 11058,
11078, 11079, 11080, 11095, 11104,
11105, 11106, 11122, 11123, 11137,
11138, 11149, 11150, 11152, 11160,
11161, 11162, 11165, 11167, 11168,
11169, 11189, 11190, 11192, 11193,
11194, 11204, 11205, 11206, 11208,
11209, 11210, 11216, 11217, 11218,
11220, 11221, 11222, 11243, 11244,
11259, 11260, 11262, 11263, 11264,
11284, 11285, 11286, 11287, 11288,
11289, 11290, 11291, 11296, 11297,
11298, 11299, 11300, 11301, 11307,
11308, 11310, 11311, 11312, 11326,
11327, 11330, 11332, 11333, 11334,
11342, 11343, 11346, 11348, 11349,
11350, 11360, 11361, 11362, 11364,
11365, 11366, 11377, 11378, 11379,
11382, 11385, 11389, 11425, 11439,
11449, 11457, 11458, 11558, 11559,
11566, 11567, 11605, 11607, 11629,
11630, 11633, 11640, 11641, 11644,
11645, 11651, 11656, 11663, 11664,
11671, 11672, 11724, 11725, 11726,
11728, 11739, 11755, 11756, 11758,
11759, 11769, 11771, 11777, 11778,
11782, 11785, 11807, 11809, 11822,
11824, 11830, 11832, 11835, 11841,
11843, 11845, 11846, 11847, 11849,
11854, 11856, 11858, 11862, 11865,
11871, 11872, 11876, 11878, 11879,
11880, 11888, 11890, 11891, 11898,
11904, 11911, 11912, 11917, 11918,
11919, 11920, 11938, 11939, 11940,
11946, 11947, 11948, 11953, 11955,
11963, 11965, 11967, 11968, 11970,
11981, 11983, 11985, 11986, 11991,
12042, 12043, 12050, 12051, 12053,
12055, 12057, 12059, 12061, 12063,
12065, 12074, 12076, 12082, 12084,
12086, 12087, 12088, 12094, 12096,
12098, 12099, 12100, 12121, 12122,
12125, 12133, 12135, 12139, 12140,
12141, 12142, 12147, 12149, 12155,
12158, 12161, 12164, 12172, 12175,
12178, 12181, 12188, 12190, 12196,
12204, 12206, 12208, 12225, 12227,
12234, 12236, 12239, 12245, 12247,
12249, 12250, 12251, 12253, 12257,
12258, 12259, 12260, 12265, 12266,
12267, 12268, 12279, 12285, 12286,
12298, 12306, 12308, 12318, 12320,
12327, 12336, 12338, 12341, 12344,
12347, 12350, 12364, 12366, 12374,
12376, 12384, 12386, 12395, 12398,
12401, 12408, 12425, 12426, 12427,
12429, 12430, 12431, 12433, 12434,
12440, 12442, 12443, 12449, 12451,
12452, 12453, 12454, 12466, 12472,
12474, 12507, 12508, 12509, 12544,
12551, 12569, 12571, 12612, 12619,
12626, 12646, 12647, 12649, 12651,
12653, 12665, 12670, 12671, 12672,
12673, 12674, 12678, 12683, 12685,
12691, 12697, 12699, 12700, 12706,
12707, 12708, 12709, 12710, 12711,
12712, 12713, 12718, 12720, 12722,
12724, 12726, 12730, 12732, 12734,
12736, 12738, 12740, 12758, 12762,
12770, 12771, 12776, 12778, 12787,
12790, 12791, 12792, 12794, 12795,
12796, 12804, 12810, 12822, 12825,
12826, 12828, 12829, 12847, 12848,
12852, 12858, 12862, 12864, 12881,
12899, 12905, 12914, 12915, 12916,
12917, 12925, 12941, 12957, 12973,
12989, 13005, 13031, 13035, 13036,
13040, 13042, 13073, 13079, 13080,
13082, 13084, 13085, 13087, 13088,
13098, 13099, 13102, 13103, 13104,
13106, 13107, 13108, 13125, 13126,
13128, 13129, 13135, 13137, 13140,
13143, 13146, 13149, 13157, 13160,
13162, 13165, 13172, 13176, 13184,

- 13185, 13188, 13190, 13192, 13196,
 13200, 13205, 13206, 13226, 13227,
 13233, 13234, 13353, 13355, 13362,
 13363, 13364, 13367, 13369, 13372,
 13373, 13380, 13385, 13392, 13393,
 13394, 13395, 13396, 13405, 13408,
 13413, 13415, 13417, 13419, 13421,
 13445, 13446, 13455, 13456, 13470,
 13472, 13499, 13500, 13509, 13512,
 13536, 13537, 13541, 13559, 13563,
 13573, 13576, 13582, 13583, 13602,
 13604, 13605, 13614, 13618, 13623,
 13626, 13632, 13658, 13659, 13665,
 13666, 13667, 13674, 13682, 13686,
 13691, 13694, 13702, 13705, 13706,
 13708, 13709, 13719, 13723, 13728,
 13733, 13742, 13743, 13746, 13747,
 13789, 13791, 13792, 13862, 13869,
 13887, 13920, 13946, 13947, 13948,
 13949, 13950, 13951, 13959, 13964,
 14872, 14918, 14919, 14920, 14927,
 14928, 15042, 15062, 15092, 15100,
 15124, 15125, 15187, 15192, 15194
 \exp_args:cc
 ... 796, 798, 975, 985, 990, 995, 1689
 \exp_args:Nc 29, 796, 796,
 800, 808, 1215, 1225, 1243, 1269,
 1274, 1281, 1336, 1347, 1403, 1436,
 1437, 1438, 1439, 1458, 1689, 4962,
 8987, 9995, 11176, 11177, 11178,
 11179, 11180, 11181, 13925, 14835
 \exp_args:Ncc 1271, 1275, 1283,
 1444, 1445, 1446, 1447, 1689, 1691
 \exp_args:Nccc 1689, 1693
 \exp_args:Ncco 1749, 1770
 \exp_args:Nccx 1793, 1802
 \exp_args:Ncf 1712, 1736
 \exp_args:NcNc 1749, 1756
 \exp_args:NcNo 1749, 1763
 \exp_args:Ncnx 1793, 1803
 \exp_args:Nco 1712, 1730
 \exp_args:Ncx 1778, 1788
 \exp_args:Nf 30, 1700, 1700, 3619,
 3767, 3836, 3848, 3857, 3950, 3963,
 3977, 3987, 3998, 4009, 4310, 6190,
 8294, 14503, 14516, 14521, 14537,
 14913, 15061, 15116, 15129, 15147
 \exp_args:Nff 1778, 1780
 \exp_args:Nfo 1778, 1779, 14491
 \exp_args:NNc 1270, 1273,
 1282, 1349, 1440, 1441, 1442, 1443,
 1477, 1689, 1689, 3728, 3735, 8296
 \exp_args:Nnc 1778, 1778
 \exp_args:NNf 1712, 1712, 3705, 3712, 3721
 \exp_args:Nnf 1778, 1781
 \exp_args:Nnnc 1793, 1795
 \exp_args:NNNo 31, 1684, 1687
 \exp_args:NNno 1793, 1793
 \exp_args:Nnno 1793, 1796
 \exp_args:NNNV 1749, 1749
 \exp_args:NNnx 31, 1793, 1798
 \exp_args:Nnnx 1793, 1800
 \exp_args:NNo 30,
 1684, 1685, 3755, 6306, 9561, 15118
 \exp_args:Nno 30,
 1778, 1782, 3251, 4281, 6145, 10030,
 10038, 10047, 10064, 10072, 10100
 \exp_args:NNoo 31, 1793, 1794
 \exp_args:NNox 1793, 1799
 \exp_args:Nnox 1793, 1801
 \exp_args:NNV 1712, 1724
 \exp_args:NNv 1712, 1718
 \exp_args:NnV 1778, 1783
 \exp_args:NNx 31, 1778, 1787
 \exp_args:Nnx 1778, 1789
 \exp_args:No 29, 1684,
 1684, 3755, 3840, 4475, 4747, 4910,
 4911, 4912, 4925, 4926, 4927, 4928,
 4951, 4968, 4977, 5031, 5033, 5141,
 5143, 5166, 5175, 5448, 5877, 6055,
 6069, 6074, 8785, 8803, 8827, 8849,
 8987, 9643, 14531, 14535, 14850, 15149
 \exp_args:Noc 1778, 1786
 \exp_args:Nof 1778, 1785
 \exp_args:Noo 1778, 1784
 \exp_args:Nooo 1793, 1797
 \exp_args:Noox 1793, 1804
 \exp_args:Nox 1778, 1790
 \exp_args:NV
 30, 1700, 1707, 8783, 8801, 8825, 8847
 \exp_args:Nv 30, 1700, 1702
 \exp_args:NVV 1712, 1742
 \exp_args:Nx 30, 1297, 1777,
 1777, 7753, 8787, 8805, 8829, 8851
 \exp_args:Nxo 1778, 1791
 \exp_args:Nxx 1778, 1792
 \exp_last_two_unbraced:Noo
 32, 1865, 1865, 7083, 7306, 7310

`\exp_last_unbraced:Nco` 13715, 13718, 13788, 13789, 13791,
 1828, 1835, 6128, 6494
`\exp_last_unbraced:NcV` 13792, 13793, 13794, 13796, 13876,
 13877, 13882, 13883, 14022, 14059,
`\exp_last_unbraced:Nf` 14064, 14087, 15003, 15183, 15184
 32, 1828, 1833, 3846, 14560
`\exp_last_unbraced:Nfo` 1828, 1862, 14894
`\exp_last_unbraced:NNNo` 1828, 1858
`\exp_last_unbraced:NnNo` 1828, 1863
`\exp_last_unbraced:NNNV` 1828, 1851
`\exp_last_unbraced:NNo`
 1828, 1849, 5109, 6096, 6476, 7280
`\exp_last_unbraced:Nno` 1828, 1860, 14865
`\exp_last_unbraced:NNV` 1828, 1843
`\exp_last_unbraced:No` 1828,
 1832, 7435, 7440, 7518, 7524, 14546
`\exp_last_unbraced:Noo`
 1828, 1861, 6430, 14880
`\exp_last_unbraced:NV` 1828, 1828
`\exp_last_unbraced:Nv` 1828, 1830
`\exp_last_unbraced:Nx` ... 32, 1828, 1864
`\exp_not:c` 33, 1869,
 1870, 1980, 2030, 2376, 2380, 2385,
 2389, 2394, 2396, 2397, 2398, 2403,
 2405, 2406, 2407, 2412, 2414, 2415,
 2416, 2421, 2423, 2424, 2425, 2432,
 2434, 2436, 2438, 2440, 2442, 2444,
 2446, 2450, 2451, 2452, 3198, 7833,
 7835, 7837, 7839, 7845, 7850, 7852,
 7854, 8077, 8079, 8081, 8083, 8089,
 8094, 8096, 8098, 8287, 8429, 8431,
 8433, 8435, 8490, 8515, 8643, 8645,
 8659, 8661, 8759, 11708, 13878, 13884
`\exp_not:f` 33, 1869, 1871
`\exp_not:N` 33, 784,
 785, 944, 1006, 1009, 1354, 1355,
 1403, 1404, 1630, 1666, 1870, 1904,
 1939, 1948, 1977, 1978, 1979, 2030,
 2070, 2381, 2384, 2388, 2394, 2403,
 2412, 2421, 2511, 2518, 2535, 2562,
 2571, 2723, 2747, 2752, 2757, 2762,
 2769, 2775, 2780, 2785, 2790, 2795,
 2800, 2810, 2815, 2843, 2848, 3101,
 3106, 3124, 3148, 3151, 3158, 3159,
 3168, 3169, 3740, 4388, 4389, 4392,
 4747, 4754, 4764, 4766, 4800, 4801,
 5194, 5197, 5213, 5216, 5247, 5254,
 5447, 5449, 5717, 6180, 6183, 6191,
 6192, 6667, 7844, 8088, 8287, 8643,
 8645, 8659, 8661, 8689, 8690, 8742,
 8743, 8759, 9101, 9102, 9559, 9627,
`\exp_not:n` 33, 784, 786,
 945, 947, 1011, 1299, 1515, 1630,
 1825, 1984, 1999, 2377, 2511, 2518,
 2535, 2562, 2571, 3102, 3107, 3121,
 3125, 3197, 3199, 3741, 4604, 4663,
 4669, 4681, 4689, 4705, 4713, 4802,
 5268, 5477, 5577, 5632, 5718, 5757,
 5766, 5896, 6062, 6184, 6190, 6215,
 6220, 6395, 6397, 6415, 7827, 7844,
 7942, 8071, 8088, 8279, 8692, 8745,
 8759, 9179, 9404, 9470, 9473, 9554,
 9705, 11709, 13994, 13998, 14515,
 14548, 14569, 14570, 14887, 14912,
 14977, 14980, 14983, 15038, 15146
`\exp_not:o` 33, 1869,
 1869, 4634, 4636, 4665, 4671, 4681,
 4683, 4685, 4687, 4689, 4691, 4693,
 4695, 4705, 4707, 4709, 4711, 4713,
 4715, 4717, 4719, 4766, 4813, 4825,
 4930, 5027, 5029, 5267, 5509, 5865,
 5867, 5962, 6055, 8492, 8506, 8517,
 8520, 8527, 8536, 8978, 8980, 15037
`\exp_not:V`
 33, 1869, 1873, 4683, 4691, 4707, 4715
`\exp_not:v` 33, 1869, 1878, 9104
`\exp_stop_f:` 33,
 1640, 1646, 2349, 3398, 3408, 3416,
 3584, 3589, 4289, 8296, 9721, 9927,
 10194, 10210, 10361, 10388, 10549,
 10570, 10597, 10611, 10646, 10673,
 10682, 10717, 10736, 10752, 10797,
 10813, 10824, 10876, 11313, 11477,
 11479, 11482, 11490, 11491, 11716,
 11718, 11738, 11780, 11853, 11875,
 11929, 11930, 12269, 12601, 12619,
 12628, 12784, 12819, 12918, 13030,
 13072, 13077, 13159, 13214, 13247,
 13261, 13276, 13291, 13306, 13321,
 13336, 13427, 13428, 13429, 13444,
 13574, 13624, 13678, 13692, 15119
`\expandafter` 13,
 14, 32, 36, 62, 63, 65, 68, 69, 72,
 73, 87, 106, 138, 173, 176, 182, 186,
 190, 191, 195, 196, 198, 208, 210,
 213, 215, 220, 288, 290, 295, 297, 345
`\ExplFileDate` 50, 305, 763, 1627,

- 2086, 2484, 2608, 3380, 4178, 4592,
 5392, 5833, 6277, 6557, 6779, 7605,
 7638, 8442, 9130, 9716, 13972, 14150
 \ExplFileDescription . 305, 763, 1627,
 2086, 2484, 2608, 3380, 4178, 4592,
 5392, 5833, 6277, 6557, 6779, 7605,
 7638, 8442, 9130, 9716, 13972, 14150
 \ExplFileName 305, 763, 1627,
 2086, 2484, 2608, 3380, 4178, 4592,
 5392, 5833, 6277, 6557, 6779, 7605,
 7638, 8442, 9130, 9716, 13972, 14150
 \ExplFileVersion . . 50, 305, 763, 1627,
 2086, 2484, 2608, 3380, 4178, 4592,
 5392, 5833, 6277, 6557, 6779, 7605,
 7638, 8442, 9130, 9716, 13972, 14150
 \ExplSyntaxNamesOff 285, 293
 \ExplSyntaxNamesOn 285, 286
 \ExplSyntaxOff 4, 6, 109,
 110, 185, 193, 215, 246, 251, 265, 280
 \ExplSyntaxOn 4, 6,
 109, 124, 155, 162, 167, 213, 246, 247
- F**
- \F 2823, 3022, 10344
 \fam 337
 \fi 45, 66, 71, 108, 181, 201, 216, 376
 \fi: 24,
 766, 770, 807, 941, 954, 973, 1020,
 1025, 1032, 1070, 1075, 1108, 1109,
 1117, 1123, 1136, 1137, 1145, 1151,
 1212, 1292, 1312, 1322, 1333, 1434,
 1517, 1522, 1530, 1562, 1605, 1606,
 1607, 1608, 1669, 1672, 1679, 1680,
 1913, 1929, 1936, 1945, 1960, 1961,
 1962, 1976, 1996, 1998, 2019, 2020,
 2126, 2160, 2187, 2206, 2326, 2328,
 2330, 2332, 2334, 2336, 2498, 2506,
 2513, 2522, 2530, 2537, 2545, 2553,
 2566, 2575, 2748, 2753, 2758, 2763,
 2770, 2776, 2781, 2786, 2791, 2796,
 2801, 2806, 2811, 2816, 2838, 2844,
 2855, 2856, 2902, 2903, 2924, 2925,
 2944, 2945, 2962, 2963, 2980, 2981,
 3048, 3057, 3066, 3074, 3140, 3154,
 3163, 3173, 3183, 3401, 3424, 3441,
 3442, 3444, 3547, 3555, 3579, 3585,
 3591, 3614, 3636, 3644, 3894, 4237,
 4260, 4269, 4288, 4292, 4306, 4463,
 4854, 4866, 4879, 4889, 4906, 5096,
 5149, 5152, 5160, 5162, 5190, 5204,
 5227, 5243, 5252, 5261, 5283, 5287,
 5295, 5505, 5508, 5535, 5556, 5566,
 5618, 5624, 5939, 5954, 5977, 5993,
 6448, 6484, 6625, 6627, 6637, 7668,
 9392, 9393, 9658, 9768, 9776, 9780,
 9791, 9801, 9811, 9875, 9913, 9914,
 9915, 9916, 9917, 9918, 9919, 9920,
 9921, 9922, 9923, 9924, 9931, 9944,
 9947, 9954, 9957, 10002, 10105,
 10174, 10175, 10184, 10185, 10196,
 10197, 10198, 10208, 10209, 10213,
 10214, 10222, 10225, 10226, 10234,
 10235, 10242, 10250, 10251, 10272,
 10291, 10292, 10301, 10307, 10327,
 10328, 10341, 10365, 10386, 10394,
 10395, 10459, 10473, 10483, 10503,
 10541, 10542, 10545, 10547, 10548,
 10553, 10564, 10567, 10569, 10574,
 10605, 10618, 10623, 10629, 10632,
 10633, 10667, 10668, 10695, 10696,
 10711, 10717, 10721, 10726, 10732,
 10747, 10758, 10775, 10785, 10787,
 10793, 10805, 10808, 10819, 10838,
 10848, 10865, 10889, 10941, 10954,
 10955, 10970, 11075, 11076, 11077,
 11096, 11112, 11129, 11170, 11195,
 11211, 11223, 11265, 11274, 11280,
 11293, 11295, 11318, 11335, 11351,
 11367, 11387, 11390, 11434, 11445,
 11457, 11458, 11459, 11466, 11468,
 11469, 11475, 11476, 11485, 11486,
 11494, 11495, 11497, 11498, 11580,
 11590, 11591, 11596, 11597, 11598,
 11599, 11600, 11601, 11608, 11617,
 11622, 11646, 11648, 11650, 11657,
 11684, 11691, 11694, 11695, 11700,
 11722, 11723, 11729, 11742, 11760,
 11762, 11772, 11786, 11816, 11825,
 11859, 11881, 11899, 11916, 11933,
 11934, 11936, 11937, 11941, 11956,
 11989, 12018, 12019, 12020, 12021,
 12022, 12035, 12077, 12150, 12215,
 12217, 12218, 12228, 12269, 12444,
 12455, 12467, 12482, 12491, 12496,
 12503, 12505, 12519, 12523, 12525,
 12529, 12600, 12607, 12618, 12634,
 12641, 12701, 12757, 12767, 12769,
 12779, 12797, 12798, 12830, 12833,
 12842, 12844, 12846, 12865, 12879,
 12880, 12900, 12921, 12922, 12935,

12951, 12967, 12983, 12999, 13015,
 13029, 13037, 13043, 13053, 13056,
 13065, 13074, 13076, 13082, 13089,
 13092, 13101, 13109, 13130, 13163,
 13164, 13191, 13193, 13207, 13208,
 13218, 13229, 13238, 13241, 13242,
 13251, 13254, 13271, 13286, 13301,
 13316, 13331, 13346, 13349, 13351,
 13356, 13370, 13374, 13404, 13423,
 13424, 13431, 13432, 13434, 13441,
 13443, 13447, 13461, 13462, 13473,
 13513, 13514, 13542, 13573, 13595,
 13607, 13623, 13644, 13677, 13691,
 13697, 13710, 13711, 13749, 13750,
 13866, 13871, 13926, 14873, 15183,
 15184, 15185, 15195, 15201, 15203
 \file_add_path:nN
 161, 9182, 9182, 9220, 9227, 9315, 9327
 \file_if_exist:n 9218
 \file_if_exist:nTF 161, 9218
 \file_input:n 161, 9225, 9225
 \file_list: 162, 9271, 9271
 \file_path_include:n ... 162, 9259, 9259
 \file_path_remove:n 162, 9259, 9266
 \finalhyphendemerits 525
 \firstmark 423
 \firstmarks 649
 \floatingpenalty 570
 \font 336
 \fontcharhp 674
 \fontcharht 673
 \fontcharic 676
 \fontcharwd 675
 \fontdimen 603
 \fontname 427
 \fp_abs:c 13872
 \fp_abs:N 13872, 13872, 13888
 \fp_abs:n
 181, 13768, 13768, 14305, 14307,
 14359, 14361, 14363, 14779, 14781
 \fp_add:cn 13827
 \fp_add:Nn 170, 13827, 13827, 13833
 \fp_compare:n 11423
 \fp_compare:nF 11503, 11514
 \fp_compare:nNn 11436
 \fp_compare:NNNF 13937
 \fp_compare:nNnF ... 11531, 11542, 13937
 \fp_compare:NNNT 13936
 \fp_compare:nNnT 11537, 11550, 13936, 14763
 \fp_compare:NNNTF 13935, 13935
 \fp_compare:nNnTF . 172, 7144, 11436,
 13935, 14184, 14186, 14191, 14379
 \fp_compare:nT 11509, 11522
 \fp_compare:nTF 11423
 \fp_compare_p:n 11423
 \fp_compare_p:nNn 172, 11436
 \fp_const:cn 13804
 \fp_const:Nn 169, 13804, 13808, 13812,
 13849, 13850, 13851, 13852, 13860
 \fp_cos:cn 13906
 \fp_cos:Nn 13906, 13916
 \fp_div:cn 13892
 \fp_div:Nn 13892, 13894, 13902
 \fp_do_until:nN 173, 11500, 11500, 11504
 \fp_do_until:nNnn 172, 11528, 11528, 11532
 \fp_do_while:nN 173, 11500, 11506, 11510
 \fp_do_while:nNnn 172, 11528, 11534, 11538
 \fp_eval:n 170, 13765, 13767
 \fp_exp:cn 13906
 \fp_exp:Nn 13906, 13913
 \fp_flag_off:n 175, 9992, 9992
 \fp_flag_on:n . 175, 9994, 9994, 10035,
 10044, 10052, 10069, 10078, 10109
 \fp_gabs:c 13872
 \fp_gabs:N 13872, 13873, 13889
 \fp_gadd:cn 13827
 \fp_gadd:Nn 170, 13827, 13828, 13834
 \fp_gcos:cn 13906
 \fp_gcos:Nn 13906, 13916
 \fp_gdiv:cn 13892
 \fp_gdiv:Nn 13892, 13895, 13903
 \fp_gexp:cn 13906
 \fp_gexp:Nn 13906, 13913
 \fp_gln:cn 13906
 \fp_gln:Nn 13906, 13914
 \fp_gmul:cn 13892
 \fp_gmul:Nn 13892, 13893, 13901
 \fp_gneg:c 13872
 \fp_gneg:N 13872, 13880, 13891
 \fp_gpow:cn 13892
 \fp_gpow:Nn 13892, 13897, 13905
 \fp_ground_figures:Nn 13956, 13961, 13967
 \fp_ground_places:Nn 13938, 13940, 13955
 .fp_gset:c 153
 \fp_gset:cn 13804
 .fp_gset:N 153
 \fp_gset:Nn . 169, 13804, 13806, 13811,
 13828, 13830, 13893, 13895, 13897
 \fp_gset_eq:cc 13813
 \fp_gset_eq:cN 13813

<code>\fp_gset_eq:Nc</code>	13813	<code>\fp_set:cn</code>	13804
<code>\fp_gset_eq:NN</code>		<code>.fp_set:N</code>	153
.....	170 , 13813 , 13814 , 13816 , 13818	<code>\fp_set:Nn</code>	169 ,
<code>\fp_gset_from_dim:cn</code>	14856		7140 , 7142 , 13804 , 13804 , 13810 ,
<code>\fp_gset_from_dim:Nn</code>			13827 , 13829 , 13892 , 13894 , 13896 ,
.....	193 , 14856 , 14858 , 14861		14171 , 14172 , 14173 , 14289 , 14291 ,
<code>\fp_gsin:cn</code>	13906		14319 , 14339 , 14352 , 14353 , 14605 ,
<code>\fp_gsin:Nn</code>	13906 , 13915		14606 , 14747 , 14749 , 14773 , 14774
<code>\fp_gsub:cn</code>	13827	<code>\fp_set_eq:cc</code>	13813
<code>\fp_gsub:Nn</code>	170 , 13827 , 13830 , 13836	<code>\fp_set_eq:cN</code>	13813
<code>\fp_gtan:cn</code>	13906	<code>\fp_set_eq:Nc</code>	13813
<code>\fp_gtan:Nn</code>	13906 , 13917	<code>\fp_set_eq:NN</code>	170 , 13813 ,
<code>\fp_gzero:c</code>	13817		13813 , 13815 , 13817 , 14324 , 14341
<code>\fp_gzero:N</code>	169 , 13817 , 13818 , 13820 , 13824	<code>\fp_set_from_dim:cn</code>	14856
<code>\fp_gzero_new:c</code>	13821	<code>\fp_set_from_dim:Nn</code>	
<code>\fp_gzero_new:N</code>	169 , 13821 , 13823 , 13826	193 , 14856 , 14856 , 14860
<code>\fp_if_exist:c</code>	11422	<code>\fp_show:c</code>	13837
<code>\fp_if_exist:cTF</code>	11421	<code>\fp_show:N</code>	175 , 13837 , 13837 , 13848
<code>\fp_if_exist:N</code>	11421	<code>\fp_show:n</code>	13837 , 13846
<code>\fp_if_exist:NTF</code>		<code>\fp_sin:cn</code>	13906
.....	171 , 11421 , 13822 , 13824 , 13839	<code>\fp_sin:Nn</code>	13906 , 13915
<code>\fp_if_exist_p:c</code>	11421	<code>\fp_sub:cn</code>	13827
<code>\fp_if_exist_p:N</code>	171 , 11421	<code>\fp_sub:Nn</code>	170 , 13827 , 13829 , 13835
<code>\fp_if_flag_on:n</code>	9996	<code>\fp_tan:cn</code>	13906
<code>\fp_if_flag_on:nTF</code>	175 , 9996	<code>\fp_tan:Nn</code>	13906 , 13917
<code>\fp_if_flag_on_p:n</code>	175 , 9996	<code>\fp_to_decimal:c</code>	13613
<code>\fp_if_undefined:N</code>	13861	<code>\fp_to_decimal:N</code>	170 , 9985 ,
<code>\fp_if_undefined:NTF</code>	13861		13613 , 13613 , 13615 , 13715 , 13765
<code>\fp_if_undefined_p:N</code>	13861	<code>\fp_to_decimal:n</code>	13613 , 13616 ,
<code>\fp_if_zero:N</code>	13868		13718 , 13767 , 13769 , 13771 , 13773
<code>\fp_if_zero:NTF</code>	13868	<code>\fp_to_dim:c</code>	13714
<code>\fp_if_zero_p:N</code>	13868	<code>\fp_to_dim:N</code>	170 , 13714 , 13714 , 13716
<code>\fp_ln:cn</code>	13906	<code>\fp_to_dim:n</code>	
<code>\fp_ln:Nn</code>	13906 , 13914		7150 , 7175 , 13714 , 13717 , 14216 ,
<code>\fp_max:nn</code>	181 , 13770 , 13770		14227 , 14678 , 14686 , 14789 , 14791
<code>\fp_min:nn</code>	181 , 13770 , 13772	<code>\fp_to_int:c</code>	13719
<code>\fp_mul:cn</code>	13892	<code>\fp_to_int:N</code>	171 , 13719 , 13719 , 13720
<code>\fp_mul:Nn</code>	13892 , 13892 , 13900	<code>\fp_to_int:n</code>	13719 , 13721
<code>\fp_neg:c</code>	13872	<code>\fp_to_scientific:c</code>	13558
<code>\fp_neg:N</code>	13872 , 13874 , 13890	<code>\fp_to_scientific:N</code>	
<code>\fp_new:N</code>	169 ,	171 , 9986 , 13558 , 13558 , 13560
	6802 , 6803 , 13801 , 13801 , 13803 ,	<code>\fp_to_scientific:n</code>	13558 , 13561
	13822 , 13824 , 13853 , 13854 , 13855 ,	<code>\fp_to_tl:c</code>	13682
	13856 , 14154 , 14155 , 14156 , 14278 ,	<code>\fp_to_tl:N</code>	171 , 13682 , 13682 , 13683 , 13840
	14279 , 14595 , 14596 , 14741 , 14742	<code>\fp_to_tl:n</code>	9732 , 10034 , 10043 , 10068 ,
<code>\fp_pow:cn</code>	13892		10077 , 10106 , 13682 , 13684 , 13847
<code>\fp_pow:Nn</code>	13892 , 13896 , 13904	<code>\fp_trap:nn</code>	175 , 10008 ,
<code>\fp_round_figures:Nn</code>	13956 , 13956 , 13966		10008 , 10120 , 10121 , 10122 , 10123
<code>\fp_round_places:Nn</code>	13938 , 13938 , 13954	<code>\fp_until_do:nn</code>	173 , 11500 , 11512 , 11517
<code>.fp_set:c</code>	153	<code>\fp_until_do:nNnn</code>	172 , 11528 , 11540 , 11545

- \fp_use:c [13765](#)
- \fp_use:N [171](#), [13765](#), [13765](#), [13766](#)
- \fp_while_do:nn . [173](#), [11500](#), [11520](#), [11525](#)
- \fp_while_do:nNnn [173](#), [11528](#), [11548](#), [11553](#)
- \fp_zero:c [13817](#)
- \fp_zero:N [169](#), [13817](#), [13817](#), [13819](#), [13822](#)
- \fp_zero_new:c [13821](#)
- \fp_zero_new:N .. [169](#), [13821](#), [13821](#), [13825](#)
- \frozen@everydisplay [743](#)
- \frozen@everymath [744](#)
- \futurelet [332](#)
- G**
- \g_cctab_allocate_int [14001](#),
[14001](#), [14002](#), [14008](#), [14010](#), [14012](#)
- \g_cctab_stack_int ... [14001](#), [14003](#),
[14040](#), [14041](#), [14043](#), [14044](#), [14048](#)
- \g_cctab_stack_seq
.. [14001](#), [14004](#), [14038](#), [14049](#), [14051](#)
- \g_file_internal_ior [9186](#),
[9187](#), [9190](#), [9206](#), [9207](#), [9408](#), [9408](#)
- \g_file_record_seq [9145](#),
[9145](#), [9150](#), [9245](#), [9250](#), [9273](#), [9293](#)
- \g_file_stack_seq [9144](#), [9144](#), [9252](#), [9255](#)
- \g_ior_streams_prop
.. [9304](#), [9304](#), [9306](#), [9355](#), [9363](#), [9371](#)
- \g_ior_streams_seq [9298](#),
[9298](#), [9300](#), [9338](#), [9364](#), [9365](#), [9414](#)
- \g_iow_streams_prop [9417](#), [9417](#),
[9419](#), [9420](#), [9421](#), [9448](#), [9456](#), [9464](#)
- \g_iow_streams_seq
.. [9412](#), [9412](#), [9414](#), [9431](#), [9457](#), [9458](#)
- \g_keyval_level_int [8446](#), [8446](#),
[8491](#), [8516](#), [8540](#), [8542](#), [8544](#), [8546](#)
- \g_prg_map_int [43](#), [2372](#), [2372](#),
[3727](#), [3730](#), [3734](#), [3737](#), [3748](#), [4960](#),
[4961](#), [4963](#), [4965](#), [5696](#), [5697](#), [5703](#),
[5704](#), [6126](#), [6127](#), [6129](#), [6132](#), [6492](#),
[6493](#), [6495](#), [6498](#), [14834](#), [14836](#), [14843](#)
- \g_scan_marks_tl [2584](#), [2584](#), [2587](#), [2593](#)
- \g_file_current_name_tl
..... [161](#), [9133](#), [9133](#),
[9138](#), [9142](#), [9150](#), [9252](#), [9253](#), [9256](#)
- \g_peek_token [58](#), [3077](#), [3078](#), [3088](#)
- \g_tmpa_bool [38](#), [2148](#), [2150](#)
- \g_tmpa_box [131](#), [6649](#), [6651](#)
- \g_tmpa_clist [121](#), [6235](#), [6237](#)
- \g_tmpa_dim [82](#), [4404](#), [4406](#)
- \g_tmpa_fp [174](#), [13853](#), [13855](#)
- \g_tmpa_int [73](#), [4112](#), [4114](#)
- \g_tmpa_muskip [88](#), [4556](#), [4558](#)
- \g_tmpa_prop [127](#), [6301](#), [6303](#)
- \g_tmpa_seq [112](#), [5808](#), [5810](#)
- \g_tmpa_skip [85](#), [4494](#), [4496](#)
- \g_tmpa_tl [102](#), [5309](#), [5309](#)
- \g_tmpb_bool [38](#), [2148](#), [2151](#)
- \g_tmpb_box [131](#), [6649](#), [6652](#)
- \g_tmpb_clist [121](#), [6235](#), [6238](#)
- \g_tmpb_dim [82](#), [4404](#), [4407](#)
- \g_tmpb_fp [174](#), [13853](#), [13856](#)
- \g_tmpb_int [73](#), [4112](#), [4115](#)
- \g_tmpb_muskip [88](#), [4556](#), [4559](#)
- \g_tmpb_prop [127](#), [6301](#), [6304](#)
- \g_tmpb_seq [112](#), [5808](#), [5811](#)
- \g_tmpb_skip [85](#), [4494](#), [4497](#)
- \g_tmpb_tl [102](#), [5309](#), [5310](#)
- \gdef [323](#)
- \GetIdInfo [6](#), [139](#)
- \global [307](#), [338](#)
- \globaldefs [342](#)
- \glueexpr [682](#)
- \glueshrink [685](#)
- \glueshrinkorder [687](#)
- \gluestretch [684](#)
- \gluestretchorder [686](#)
- \gluetomu [688](#)
- \group_align_safe_begin: ... [42](#), [2164](#),
[2333](#), [2333](#), [3109](#), [3127](#), [4796](#), [5076](#)
- \group_align_safe_end:
..... [42](#), [2212](#), [2213](#), [2333](#), [2335](#),
[3091](#), [3101](#), [3106](#), [3124](#), [4805](#), [5102](#)
- \group_begin: [9](#), [790](#), [791](#),
[1076](#), [1459](#), [1477](#), [1894](#), [2028](#), [2042](#),
[2339](#), [2354](#), [2708](#), [2721](#), [2728](#), [2765](#),
[2818](#), [2858](#), [3018](#), [3185](#), [4728](#), [4746](#),
[4897](#), [5528](#), [6676](#), [7608](#), [7760](#), [7777](#),
[7818](#), [8062](#), [8269](#), [8451](#), [8461](#), [9164](#),
[9493](#), [9498](#), [9524](#), [10342](#), [10398](#),
[10928](#), [11118](#), [11173](#), [11183](#), [11198](#),
[11321](#), [11354](#), [11372](#), [11610](#), [13381](#),
[13566](#), [14077](#), [14170](#), [14284](#), [14314](#),
[14334](#), [14351](#), [15152](#), [15159](#), [15172](#)
- \group_end: ... [9](#), [790](#), [792](#), [1081](#), [1464](#),
[1477](#), [1901](#), [2031](#), [2047](#), [2344](#), [2359](#),
[2720](#), [2724](#), [2737](#), [2772](#), [2826](#), [2868](#),
[3024](#), [3250](#), [4735](#), [4753](#), [4901](#), [4904](#),
[5538](#), [5543](#), [6686](#), [7612](#), [7767](#), [7788](#),
[7922](#), [8107](#), [8291](#), [8458](#), [8469](#), [9178](#),
[9497](#), [9502](#), [9561](#), [10349](#), [10403](#),
[10972](#), [11146](#), [11182](#), [11197](#), [11225](#),

11353, 11369, 11399, 11621, 13389,
13570, 14080, 14175, 14297, 14326,
14343, 14365, 15161, 15170, 15200
\group_execute_after:N 1597
\group_insert_after:N . 9, 795, 795, 1597

H

\halign 349
\hangafter 527
\hangindent 528
\hbadness 589
\hbox 584
\hbox:n 132,
6688, 6688, 7402, 7457, 14199, 14390
\hbox_gset:cn 6689
\hbox_gset:cw 6699, 6711
\hbox_gset:Nn 132, 6689, 6690, 6692
\hbox_gset:Nw . 133, 6699, 6701, 6704, 6710
\hbox_gset_end: ... 133, 6699, 6706, 6712
\hbox_gset_inline_begin:c ... 6707, 6711
\hbox_gset_inline_begin:N ... 6707, 6710
\hbox_gset_inline_end: 6707, 6712
\hbox_gset_to_wd:cnn 6693
\hbox_gset_to_wd:Nnn 132, 6693, 6695, 6698
\hbox_overlap_left:n ... 132, 6716, 6716
\hbox_overlap_right:n
132, 6716, 6718, 14374
\hbox_set:cn 6689
\hbox_set:cw 6699, 6708
\hbox_set:Nn 132, 6689, 6689, 6690, 6691,
6860, 6964, 7188, 7259, 7547, 14168,
14195, 14196, 14282, 14312, 14332,
14349, 14371, 14399, 14403, 14411,
14419, 14428, 14434, 14446, 14454,
14462, 14468, 14478, 14617, 14631
\hbox_set:Nw
133, 6699, 6699, 6702, 6703, 6707, 6907
\hbox_set_end: 133, 6699, 6705, 6709, 6911
\hbox_set_inline_begin:c 6707, 6708
\hbox_set_inline_begin:N 6707, 6707
\hbox_set_inline_end: 6707, 6709
\hbox_set_to_wd:cnn 6693
\hbox_set_to_wd:Nnn
132, 6693, 6693, 6696, 6697
\hbox_to_wd:nn ... 132, 6713, 6713, 14381
\hbox_to_zero:n 132, 6713, 6715, 6717, 6719
\hbox_unpack:c 6720
\hbox_unpack:N
133, 6720, 6720, 6722, 7192, 7339
\hbox_unpack_clear:c 6720

\hbox_unpack_clear:N 133, 6720, 6721, 6723
\hcoffin_set:cn 6856
\hcoffin_set:cw 6903
\hcoffin_set:Nn 136, 6856,
6856, 6872, 7399, 7411, 7454, 7494
\hcoffin_set:Nw ... 137, 6903, 6903, 6919
\hcoffin_set_end: . 137, 6903, 6908, 6918
\hfil 492
\hfill 494
\hfilneg 493
\hfuzz 591
\hoffset 566
\holdinginserts 569
\hrule 505
\hsize 530
\hskip 495
\hss 496
\ht 634
\hyphenation 620
\hyphenchar 604
\hyphenpenalty 522

I

pi 180
\if 191, 358
\if:w 24, 766, 772,
1070, 1954, 1957, 1958, 2012, 2015,
3061, 10532, 10536, 10558, 10652,
10684, 10706, 10726, 10799, 10842,
10853, 11188, 11203, 11215, 13101
\if_bool:N 42, 2089, 2089
\if_box_empty:N ... 135, 6621, 6623, 6637
\if_case:w 74, 1300,
3383, 3387, 3866, 9774, 9927, 11089,
11269, 11313, 11714, 11853, 11928,
11952, 12004, 12601, 12628, 12784,
12819, 12925, 12941, 12957, 12973,
12989, 13005, 13030, 13077, 13199,
13214, 13261, 13276, 13291, 13306,
13321, 13336, 13574, 13624, 13692
\if_catcode:w 24, 766,
774, 2747, 2752, 2757, 2762, 2769,
2775, 2780, 2785, 2790, 2795, 2800,
2810, 2843, 3143, 3148, 5212, 5254,
5272, 9653, 10379, 10469, 10716,
10763, 10932, 11270, 15183, 15184
\if_charcode:w 24,
766, 773, 2815, 3145, 5187, 5193, 5247
\if_cs_exist:N
. 24, 780, 780, 1104, 1132, 2851, 3070

<code>\if_cs_exist:w</code>	10171, 10181, 10193, 10208, 10209, 780, 781, 803, 1113, 1141, 1288, 9998
<code>\if_dim:w</code>	10220, 10221, 10231, 10247, 10266, 88, 4181, 4181, 4256, 4268, 4293
<code>\if_eof:w</code>	10301, 10304, 10320, 10327, 10380, 166, 9380, 9380, 9388
<code>\if_false:</code>	10500, 10623, 10629, 10764, 10961, 24, 766, 767, 2334, 3567, 4278, 5149, 5152, 5162, 5287, 5295, 5505, 5508, 5618, 5624
<code>\if_hbox:N</code>	11067, 11070, 11073, 11430, 11457, 11458, 11459, 11460, 11461, 11462, 11575, 11576, 11604, 11615, 11625, 11682, 11689, 11698, 11715, 11749, 11754, 11768, 11814, 11821, 11897, 11909, 12008, 12011, 12022, 12073, 12146, 12214, 12217, 12224, 12269, 12438, 12448, 12515, 12526, 12598, 12696, 12775, 12824, 12838, 13027, 13039, 13051, 13054, 13057, 13082, 13183, 13187, 13352, 13404, 13461, 13514, 13573, 13623, 13691, 13738, 13741, 13871, 13926, 14871, 15185
<code>\if_int_compare:w</code>	<code>\if_mode_horizontal:</code> .. 24, 776, 777, 2328
793, 1515, 1521, 1526, 2334, 2336, 2510, 2517, 2534, 2561, 2570, 2834, 3052, 3383, 3420, 3547, 3595, 3597, 3599, 3601, 3603, 3605, 3607, 3610, 4161, 4457, 9385, 9774, 9775, 9871, 9936, 9937, 10172, 10182, 10190, 10211, 10223, 10232, 10240, 10248, 10284, 10289, 10361, 10388, 10445, 10468, 10549, 10570, 10597, 10611, 10646, 10673, 10716, 10736, 10752, 10765, 10777, 10797, 10813, 10824, 10876, 10933, 10943, 11108, 11125, 11159, 11258, 11325, 11339, 11358, 11376, 11381, 11438, 11468, 11471, 11482, 11485, 11490, 11491, 11494, 11497, 11582, 11655, 11718, 11738, 11780, 11875, 11929, 11930, 11933, 11936, 12005, 12014, 12215, 12465, 12480, 12489, 12494, 12614, 12630, 12755, 12789, 12850, 12857, 12861, 12898, 13070, 13072, 13083, 13101, 13124, 13156, 13159, 13202, 13222, 13223, 13231, 13232, 13246, 13361, 13366, 13427, 13428, 13429, 13444, 13601, 13675, 13701, 13704, 13865	<code>\if_mode_inner:</code>
<code>\if_int_odd:w</code> 74, 3383, 3386, 3632, 3640, 10194, 10208, 10220, 11988, 13247, 13458, 13469, 13511, 13539, 15182	<code>\if_mode_math:</code>
<code>\if_meaning:w</code> .. 24, 766, 775, 937, 952, 969, 1016, 1021, 1030, 1101, 1119, 1129, 1147, 1318, 1329, 1433, 1560, 1666, 1667, 1904, 1925, 1934, 2122, 2187, 2206, 2496, 2502, 2528, 2541, 2549, 2805, 2848, 2890, 2893, 2912, 2915, 2932, 2935, 2950, 2953, 2968, 2971, 3044, 3136, 3177, 3401, 3436, 3441, 3442, 3579, 4237, 4286, 4850, 4862, 4874, 4885, 4900, 5094, 5155, 5238, 5533, 5552, 5560, 5935, 5950, 5972, 5988, 6444, 6482, 9763, 9776, 9787, 9797, 9807, 9952, 9954, 10105,	<code>\if_mode_vertical:</code> 24, 776, 778, 2326
	<code>\if_num:w</code>
	<code>\if_predicate:w</code> 42, 2089, 2090, 2156
	<code>\if_true:</code>
	<code>\if_vbox:N</code>
	<code>\ifcase</code>
	<code>\ifcat</code>
	<code>\ifcsname</code>
	<code>\ifdefined</code>
	<code>\ifdim</code>
	<code>\ifeof</code>
	<code>\iffalse</code>
	<code>\iffontchar</code>
	<code>\ifhbox</code>
	<code>\ifhmode</code>
	<code>\ifinner</code>
	<code>\ifmmode</code>
	<code>\ifnum</code>
	<code>\ifodd</code>
	<code>\iftrue</code>
	<code>\ifvbox</code>
	<code>\ifvmode</code>
	<code>\ifvoid</code>
	<code>\ifx</code>
	<code>\ignorespaces</code>
	<code>\immediate</code>
	<code>min</code>
	<code>sin</code>
	<code>\indent</code>
	<code>\initcatcodetable</code>

.initial:n	153	\int_do_while:nNnn	67, 3674, 3690, 3694
.initial:o	153	\int_eval:n	62, 1326, 1342, 3388, 3389, 3392, 3619, 3714, 3722, 3750, 3836, 3905, 3915, 3974, 3988, 3992, 3995, 4010, 4019, 5001, 5006, 5727, 6171, 6180, 6668, 9346, 9439, 9752, 14503, 14516, 14538, 14896, 14913, 15066, 15131, 15147
.initial:V	153	\int_eval:w	4166, 4167
.initial:x	153	\int_eval_end:	4166, 4168
\input	386	\int_from_alph:n	71, 3972, 3972
\input@path	9196, 9199, 9214	\int_from_base:nn	72, 3993, 3993, 4024, 4026, 4028, 4119
\inputlineno	388	\int_from_binary:n	71, 4023, 4023
\insert	568	\int_from_hexadecimal:n	71, 4023, 4025
\insertpenalties	571	\int_from_octal:n	72, 4023, 4027
\int_abs:n	62, 3394, 3394	\int_from_roman:n	72, 4043, 4043
\int_add:cn	3514	\int_gadd:cn	3514
\int_add:Nn	64, 3514, 3514, 3519, 3522, 9575, 9588, 9626	\int_gadd:Nn	64, 3514, 3518, 3523, 14008, 14040
\int_case:nnn	67, 2468, 3616, 3616, 3757, 3763, 5739, 6198	\int_gdecr:c	3526
\int_compare:n	3560	\int_gdecr:N	64, 3526, 3532, 3537, 3748, 4965, 5704, 6132, 6498, 8546, 14843
\int_compare:nF	3656, 3671	\int_gincr:c	3526
\int_compare:nNn	3608	\int_gincr:N	64, 3526, 3530, 3536, 3727, 3734, 4960, 5696, 6126, 6492, 8540, 14834
\int_compare:nNnF	3684, 3699, 3718	.int_gset:c	153
\int_compare:nNnT	3676, 3693, 4392, 14041, 14898, 15133	\int_gset:cn	3538
\int_compare:nNnTF	65, 2238, 3468, 3470, 3608, 3625, 3704, 3707, 3753, 3839, 3845, 3992, 4016, 4020, 4070, 9576, 13650, 13652, 14009, 14499, 14501, 14506, 14514, 14534, 14911, 15145	.int_gset:N	153
\int_compare:nT	3648, 3665, 9360, 9453	\int_gset:Nn	64, 3473, 3483, 3538, 3540, 3542
\int_compare:nTF	66, 3560	\int_gset_eq:cc	3506
\int_compare_p:n	66, 3560	\int_gset_eq:cN	3506
\int_compare_p:nNn	65, 3608	\int_gset_eq:Nc	3506
\int_const:cn	3466, 4029, 4030, 4031, 4032, 4033, 4034, 4035, 4036, 4037, 4038, 4039, 4040, 4041, 4042	\int_gset_eq:NN	63, 3506, 3509, 3510, 3511, 7805
\int_const:Nn	63, 3466, 3466, 3486, 4093, 4094, 4095, 4096, 4097, 4098, 4099, 4100, 4101, 4102, 4103, 4104, 4105, 4106, 4107, 4108, 4109, 4110, 4111, 9746, 9845, 9846, 9847, 9851, 9852, 9853, 9858, 9859, 9860	\int_gsub:cn	3514
\int_convert_from_base_ten:nn	4116, 4117	\int_gsub:Nn	64, 3514, 3520, 3525, 14048
\int_convert_to_base_ten:nn	4116, 4119	\int_gzero:c	3496
\int_convert_to_symbols:nnn	4116, 4118	\int_gzero:N	63, 3496, 3497, 3499, 3503
\int_decr:c	3526	\int_gzero_new:c	3500
\int_decr:N	64, 3526, 3528, 3533, 3535	\int_gzero_new:N	63, 3500, 3502, 3505
\int_div_round:nn	62, 3426, 3447	\int_if_even:n	3638
\int_div_truncate:nn	63, 3426, 3426, 3768, 3858	\int_if_even:nTF	67, 3630
\int_do_until:nn	68, 3646, 3668, 3672	\int_if_even_p:n	67, 3630
\int_do_until:nNnn	67, 3674, 3696, 3700	\int_if_exist:c	3513
\int_do_while:nn	68, 3646, 3662, 3666	\int_if_exist:cF	4059, 4066, 4068
		\int_if_exist:cTF	3512
		\int_if_exist:N	3512
		\int_if_exist:NnTF	64, 3501, 3503, 3512

`\int_if_exist_p:c` [3512](#)
`\int_if_exist_p:N` [64](#), [3512](#)
`\int_if_odd:n` [3630](#)
`\int_if_odd:nTF` [67](#), [3630](#)
`\int_if_odd_p:n` [67](#), [3630](#)
`\int_incr:c` [3526](#)
`\int_incr:N` [64](#), [3526](#), [3526](#),
[3531](#), [3534](#), [8686](#), [8739](#), [9098](#), [9594](#)
`\int_max:nn` [63](#), [3394](#), [3402](#)
`\int_min:nn` [63](#), [3394](#), [3410](#)
`\int_mod:nn` [63](#), [3426](#), [3449](#), [3758](#), [3849](#)
`\int_new:c` [3458](#)
`\int_new:N` [63](#), [2372](#), [3458](#), [3459](#), [3465](#),
[3472](#), [3482](#), [3501](#), [3503](#), [4112](#), [4113](#),
[4114](#), [4115](#), [4164](#), [8446](#), [8560](#), [9481](#),
[9483](#), [9484](#), [9485](#), [9486](#), [14001](#), [14003](#)
`.int_set:c` [153](#)
`\int_set:cn` [3538](#)
`.int_set:N` [153](#)
`\int_set:Nn` [64](#), [3538](#), [3538](#), [3540](#), [3541](#),
[6677](#), [6678](#), [8690](#), [8743](#), [9102](#), [9405](#),
[9482](#), [9531](#), [9545](#), [9573](#), [9601](#), [14002](#)
`\int_set_eq:cc` [3506](#)
`\int_set_eq:cN` [3506](#)
`\int_set_eq:Nc` [3506](#)
`\int_set_eq:NN` [63](#), [3506](#), [3506](#),
[3507](#), [3508](#), [6679](#), [9403](#), [9499](#), [9525](#)
`\int_show:c` [4089](#), [4090](#)
`\int_show:N` [72](#), [4089](#), [4089](#)
`\int_show:n` [72](#), [4091](#), [4091](#)
`\int_step_function:nnnN`
..... [69](#), [2476](#), [3702](#), [3702](#), [3747](#)
`\int_step_inline:nnnn`
.. [69](#), [2477](#), [3725](#), [3725](#), [14120](#), [14125](#)
`\int_step_variable:nnnNn`
..... [69](#), [2478](#), [3725](#), [3732](#)
`\int_sub:cn` [3514](#)
`\int_sub:Nn` [64](#), [3514](#), [3516](#), [3521](#), [3524](#), [9632](#)
`\int_to_Alph:n` [70](#), [3771](#), [3803](#)
`\int_to_alph:n` [70](#), [3771](#), [3771](#)
`\int_to_arabic:n` [69](#), [3750](#), [3750](#)
`\int_to_base:nn`
[71](#), [3835](#), [3835](#), [3897](#), [3899](#), [3901](#), [4117](#)
`\int_to_binary:n` [70](#), [3896](#), [3896](#)
`\int_to_hexadecimal:n` ... [71](#), [3896](#), [3898](#)
`\int_to_octal:n` [71](#), [3896](#), [3900](#)
`\int_to_Roman:n` [71](#), [3902](#), [3912](#)
`\int_to_roman:n` [71](#), [3902](#), [3902](#)
`\int_to_symbol:n` [4121](#), [4122](#)
`\int_to_symbol_math:n` .. [4121](#), [4126](#), [4129](#)
`\int_to_symbol_text:n` .. [4121](#), [4127](#), [4144](#)
`\int_to_symbols:nnn` .. [70](#), [3751](#), [3751](#),
[3767](#), [3773](#), [3805](#), [4118](#), [4131](#), [4146](#)
`\int_until_do:nn` ... [68](#), [3646](#), [3654](#), [3659](#)
`\int_until_do:nNnn` .. [68](#), [3674](#), [3682](#), [3687](#)
`\int_use:c` [3543](#), [3544](#)
`\int_use:N` [65](#), [3397](#), [3405](#),
[3406](#), [3413](#), [3414](#), [3428](#), [3430](#), [3431](#),
[3543](#), [3543](#), [3544](#), [3563](#), [3730](#), [3737](#),
[4092](#), [4961](#), [4963](#), [5697](#), [5703](#), [6127](#),
[6129](#), [6493](#), [6495](#), [7727](#), [8176](#), [8491](#),
[8516](#), [8542](#), [8544](#), [8691](#), [8744](#), [9103](#),
[9405](#), [9758](#), [9961](#), [10207](#), [10283](#),
[10287](#), [10326](#), [10528](#), [10585](#), [10596](#),
[10645](#), [10676](#), [10682](#), [10683](#), [10830](#),
[10832](#), [10855](#), [10862](#), [10871](#), [10882](#),
[10884](#), [11153](#), [11779](#), [11808](#), [11810](#),
[11831](#), [11833](#), [11842](#), [11844](#), [11866](#),
[11873](#), [11879](#), [11889](#), [11891](#), [11964](#),
[11966](#), [11982](#), [11984](#), [12044](#), [12052](#),
[12054](#), [12056](#), [12058](#), [12060](#), [12062](#),
[12064](#), [12083](#), [12085](#), [12095](#), [12097](#),
[12123](#), [12126](#), [12134](#), [12136](#), [12156](#),
[12159](#), [12162](#), [12165](#), [12173](#), [12176](#),
[12179](#), [12182](#), [12189](#), [12191](#), [12197](#),
[12205](#), [12207](#), [12209](#), [12235](#), [12237](#),
[12246](#), [12248](#), [12261](#), [12280](#), [12287](#),
[12299](#), [12307](#), [12309](#), [12319](#), [12321](#),
[12328](#), [12337](#), [12339](#), [12342](#), [12345](#),
[12348](#), [12351](#), [12365](#), [12367](#), [12375](#),
[12377](#), [12385](#), [12387](#), [12396](#), [12399](#),
[12402](#), [12409](#), [12428](#), [12473](#), [12475](#),
[12510](#), [12545](#), [12552](#), [12570](#), [12572](#),
[12620](#), [12650](#), [12652](#), [12654](#), [12666](#),
[12679](#), [12684](#), [12686](#), [12692](#), [12709](#),
[12710](#), [12711](#), [12712](#), [12713](#), [12714](#),
[12719](#), [12721](#), [12723](#), [12725](#), [12727](#),
[12731](#), [12733](#), [12735](#), [12737](#), [12739](#),
[12741](#), [12763](#), [12771](#), [12853](#), [12906](#),
[13136](#), [13138](#), [13141](#), [13144](#), [13147](#),
[13150](#), [13166](#), [13365](#), [13406](#), [13409](#),
[13414](#), [13416](#), [13418](#), [13420](#), [13501](#),
[13538](#), [13584](#), [13606](#), [13951](#), [14836](#)
`\int_value:w` [4170](#), [4171](#)
`\int_while_do:nn` ... [68](#), [3646](#), [3646](#), [3651](#)
`\int_while_do:nNnn` .. [68](#), [3674](#), [3674](#), [3679](#)
`\int_zero:c` [3496](#)
`\int_zero:N` [63](#), [3496](#), [3496](#), [3498](#), [3501](#),
[8683](#), [8736](#), [9088](#), [9547](#), [9549](#), [9620](#)
`\int_zero_new:c` [3500](#)

- \int_zero_new:N [63](#), [3500](#), [3500](#), [3504](#)
 - \interactionmode [670](#)
 - \interlinepenalties [691](#)
 - \interlinepenalty [550](#)
 - \ior_close:c [9358](#)
 - \ior_close:N
 - [163](#), [9190](#), [9337](#), [9358](#), [9358](#), [9369](#)
 - \ior_get:NN [163](#), [9397](#), [9397](#), [9693](#), [14829](#)
 - \ior_get_str:NN [163](#), [9399](#), [9399](#), [9695](#), [14831](#)
 - \ior_gto:NN [9693](#), [9694](#)
 - \ior_if_eof:N [9381](#)
 - \ior_if_eof:Nf [9207](#), [14841](#), [14848](#)
 - \ior_if_eof:Ntf [164](#), [9187](#), [9381](#)
 - \ior_if_eof:p:N [164](#), [9381](#)
 - \ior_list_streams: [163](#), [9370](#), [9370](#), [9710](#)
 - \ior_map_break:
 - [192](#), [14824](#), [14824](#), [14825](#), [14827](#), [14842](#)
 - \ior_map_break:n [193](#), [14824](#), [14826](#)
 - \ior_map_inline:Nn [192](#), [14828](#), [14828](#)
 - \ior_new:c [9308](#)
 - \ior_new:N [162](#), [9308](#), [9308](#), [9309](#), [9408](#)
 - \ior_open:cn [9310](#)
 - \ior_open:cnTF [9320](#)
 - \ior_open:Nn [162](#), [9310](#), [9310](#), [9312](#), [9320](#)
 - \ior_open:NnF [9323](#)
 - \ior_open:NnT [9322](#)
 - \ior_open:NnTF [162](#), [9320](#), [9324](#)
 - \ior_open_streams: [9709](#), [9710](#)
 - \ior_str_gto:NN [9693](#), [9696](#)
 - \ior_str_map_inline:Nn [192](#), [14828](#), [14830](#)
 - \ior_str_to:NN [9693](#), [9695](#), [9696](#)
 - \ior_to:NN [9693](#), [9693](#), [9694](#)
 - \iow_char:N [164](#), [9480](#), [9480](#), [13024](#)
 - \iow_close:c [9451](#)
 - \iow_close:N [163](#), [9430](#), [9451](#), [9451](#), [9462](#)
 - \iow_indent:n [165](#), [8140](#),
 - [9031](#), [9515](#), [9515](#), [9534](#), [10133](#), [10145](#)
 - \iow_list_streams: [163](#), [9463](#), [9463](#), [9711](#)
 - \iow_log:n [164](#), [7794](#), [7795](#), [7796](#),
 - [7919](#), [9285](#), [9286](#), [9287](#), [9475](#), [9476](#)
 - \iow_log:x [1171](#),
 - [1171](#), [1210](#), [2034](#), [7666](#), [9475](#), [9475](#)
 - \iow_new:c [9423](#)
 - \iow_new:N [162](#), [9423](#), [9423](#), [9424](#)
 - \iow_newline:
 - [165](#), [7756](#), [7772](#), [7774](#), [9479](#), [9479](#), [9542](#)
 - \iow_now:Nn [164](#), [9472](#), [9472](#),
 - [9474](#), [9476](#), [9478](#), [9699](#), [9705](#), [9707](#)
 - \iow_now:Nx [9472](#), [9475](#), [9477](#), [9701](#)
 - \iow_now_buffer_safe:Nn [9703](#), [9704](#)
 - \iow_now_buffer_safe:Nx [9703](#), [9706](#)
 - \iow_now_when_avail:Nn [9697](#), [9698](#)
 - \iow_now_when_avail:Nx [9697](#), [9700](#)
 - \iow_open:cn [9425](#)
 - \iow_open:Nn [163](#), [9425](#), [9425](#), [9427](#)
 - \iow_open_streams: [9709](#), [9711](#)
 - \iow_shipout:Nn [164](#), [9469](#), [9469](#), [9471](#)
 - \iow_shipout:Nx [9469](#)
 - \iow_shipout_x:Nn [164](#), [9466](#), [9466](#), [9468](#)
 - \iow_shipout_x:Nx [9466](#)
 - \iow_term:n
 - [164](#), [7800](#), [7801](#), [7802](#), [8325](#), [9475](#), [9478](#)
 - \iow_term:x [1171](#), [1173](#), [7770](#), [9475](#), [9477](#)
 - \iow_wrap:nnnN
 - [165](#), [7748](#), [7749](#), [7795](#), [7801](#),
 - [7917](#), [8323](#), [8347](#), [9522](#), [9522](#), [9688](#)
 - \iow_wrap:xnnnN [9686](#), [9687](#), [9705](#), [9707](#)
- J**
- \jobname [625](#)
- K**
- \kern [503](#)
 - \keys_define:nn [151](#), [8569](#), [8569](#), [9029](#)
 - \keys_if_choice_exist:nnn [9003](#)
 - \keys_if_choice_exist:nnnTF [159](#), [9003](#)
 - \keys_if_choice_exist_p:nnn [159](#), [9003](#)
 - \keys_if_exist:nn [8997](#)
 - \keys_if_exist:nnTF [159](#), [8997](#)
 - \keys_if_exist_p:nn [159](#), [8997](#)
 - \keys_set:nn
 - [157](#), [8713](#), [8717](#), [8720](#), [8880](#), [8880](#), [8888](#)
 - \keys_set:no [8880](#)
 - \keys_set:nV [8880](#)
 - \keys_set:nv [8880](#)
 - \keys_set_known:nnN [158](#), [8890](#), [8890](#), [8902](#)
 - \keys_set_known:noN [8890](#)
 - \keys_set_known:nVN [8890](#)
 - \keys_set_known:nvN [8890](#)
 - \keys_show:nn [159](#), [9009](#), [9009](#)
 - \keyval_parse:NNn [160](#), [8538](#), [8538](#),
 - [8574](#), [8885](#), [8897](#), [9118](#), [9119](#), [9120](#)
 - \KV_process_no_space_removal_no_sanitization:NNn
 - [9117](#), [9120](#)
 - \KV_process_space_removal_no_sanitization:NNn
 - [9117](#), [9119](#)
 - \KV_process_space_removal_sanitization:NNn
 - [9117](#), [9118](#)
- L**
- \l@expl@log@functions@bool [1202](#), [7658](#)

- \l__box_angle_fp
190, 14154, 14154, 14171, 14172, 14173
- \l__box_bottom_dim
.... 14157, 14158, 14181, 14238,
14242, 14247, 14253, 14258, 14262,
14271, 14273, 14286, 14294, 14305,
14316, 14322, 14336, 14355, 14361
- \l__box_bottom_new_dim
14161, 14162, 14207, 14239, 14250,
14261, 14272, 14304, 14360, 14378
- \l__box_cos_fp 190, 14155, 14155,
14173, 14186, 14191, 14218, 14230
- \l__box_internal_box 190, 14165,
14165, 14195, 14196, 14202, 14206,
14207, 14208, 14210, 14371, 14377,
14378, 14384, 14389, 14393, 14403,
14411, 14414, 14416, 14419, 14422,
14424, 14426, 14428, 14429, 14430,
14431, 14434, 14436, 14437, 14439,
14441, 14446, 14454, 14457, 14459,
14462, 14463, 14464, 14468, 14469,
14470, 14478, 14481, 14483, 14485
- \l__box_left_dim
14157, 14159, 14183, 14238, 14240,
14249, 14253, 14258, 14264, 14269,
14273, 14288, 14318, 14338, 14357
- \l__box_left_new_dim
..... 14161, 14163, 14198,
14209, 14241, 14252, 14263, 14274
- \l__box_right_dim 14157, 14160, 14182,
14236, 14242, 14247, 14251, 14260,
14262, 14271, 14275, 14287, 14290,
14317, 14337, 14340, 14356, 14363
- \l__box_right_new_dim . 14161, 14164,
14209, 14243, 14254, 14265, 14276,
14303, 14362, 14381, 14383, 14389
- \l__box_scale_x_fp
. 190, 14278, 14278, 14289, 14324,
14339, 14341, 14352, 14363, 14379
- \l__box_scale_y_fp 190, 14278,
14279, 14291, 14305, 14307, 14319,
14324, 14341, 14353, 14359, 14361
- \l__box_sin_fp 190, 14155,
14156, 14172, 14184, 14219, 14229
- \l__box_top_dim
.... 14157, 14157, 14180, 14236,
14240, 14249, 14251, 14260, 14264,
14269, 14275, 14285, 14294, 14307,
14315, 14322, 14335, 14354, 14359
- \l__box_top_new_dim
14161, 14161, 14206, 14237, 14248,
14259, 14270, 14306, 14358, 14377
- \l__cctab_internal_tl
.. 14036, 14050, 14051, 14052, 14074
- \l__clist_internal_clist
..... 5837, 5837, 5913,
5914, 5926, 5927, 6073, 6074, 6137,
6138, 6154, 6155, 6231, 6232, 8246
- \l__clist_internal_remove_clist
.. 6017, 6017, 6024, 6027, 6028, 6030
- \l__coffin_aligned_coffin
..... 6963, 6965, 7187,
7188, 7192, 7198, 7200, 7201, 7217,
7218, 7224, 7225, 7226, 7227, 7228,
7230, 7232, 7236, 7237, 7242, 7243,
7244, 7245, 7246, 7280, 7295, 7341,
7342, 7547, 7554, 7556, 7558, 7560
- \l__coffin_aligned_internal_coffin .
..... 6963, 6966, 7259, 7266
- \l__coffin_bottom_corner_dim
.... 14599, 14601, 14623, 14627,
14697, 14708, 14709, 14729, 14737
- \l__coffin_bounding_prop
.... 14597, 14597, 14612, 14640,
14642, 14645, 14647, 14653, 14716
- \l__coffin_bounding_shift_dim 14598,
14598, 14621, 14715, 14721, 14722
- \l__coffin_cos_fp
.. 14595, 14596, 14606, 14680, 14689
- \l__coffin_display_coffin
..... 7345, 7345, 7473, 7479,
7549, 7550, 7555, 7557, 7559, 7560
- \l__coffin_display_coord_coffin
..... 7345, 7346,
7411, 7431, 7447, 7494, 7514, 7533
- \l__coffin_display_font_tl
.. 7390, 7390, 7392, 7395, 7419, 7502
- \l__coffin_display_handles_prop
7348, 7348, 7349, 7351, 7353, 7355,
7357, 7359, 7361, 7363, 7365, 7367,
7369, 7371, 7373, 7375, 7377, 7379,
7381, 7383, 7422, 7426, 7505, 7509
- \l__coffin_display_offset_dim 7385,
7385, 7386, 7448, 7449, 7534, 7535
- \l__coffin_display_pole_coffin
.. 7345, 7347, 7399, 7410, 7454, 7492
- \l__coffin_display_poles_prop
..... 7389, 7389,
7464, 7469, 7472, 7474, 7476, 7483

- \l__coffin_display_x_dim [7387](#), [7387](#), [7489](#), [7544](#)
- \l__coffin_display_y_dim [7387](#), [7388](#), [7490](#), [7546](#)
- \l__coffin_error_bool [6804](#), [6804](#), [7082](#), [7086](#), [7100](#), [7115](#), [7146](#), [7485](#), [7487](#)
- \l__coffin_internal_box . [6782](#), [6782](#), [6891](#), [6895](#), [6899](#), [6938](#), [6943](#), [6948](#), [14617](#), [14626](#), [14628](#), [14629](#), [14631](#)
- \l__coffin_internal_dim [6782](#), [6783](#), [7193](#), [7195](#), [7196](#), [14644](#), [14646](#), [14648](#), [14776](#), [14779](#)
- \l__coffin_internal_tl [6782](#), [6784](#), [6791](#), [6792](#), [6793](#), [6794](#), [6795](#), [6796](#), [6797](#), [6798](#), [6799](#), [6800](#), [6801](#), [7278](#), [7279](#), [7281](#), [7423](#), [7424](#), [7427](#), [7428](#), [7436](#), [7441](#), [7506](#), [7507](#), [7510](#), [7511](#), [7520](#), [7525](#)
- \l__coffin_left_corner_dim [14599](#), [14599](#), [14621](#), [14630](#), [14698](#), [14704](#), [14705](#), [14728](#), [14736](#)
- \l__coffin_offset_x_dim [6805](#), [6805](#), [7190](#), [7191](#), [7194](#), [7202](#), [7204](#), [7206](#), [7212](#), [7215](#), [7235](#), [7255](#), [7263](#), [7543](#), [7551](#)
- \l__coffin_offset_y_dim [6805](#), [6806](#), [7205](#), [7207](#), [7212](#), [7215](#), [7235](#), [7257](#), [7264](#), [7545](#), [7552](#)
- \l__coffin_pole_a_tl [6807](#), [6807](#), [7080](#), [7085](#), [7304](#), [7307](#), [7308](#), [7311](#), [7466](#), [7468](#), [7471](#)
- \l__coffin_pole_b_tl [6807](#), [6808](#), [7081](#), [7085](#), [7305](#), [7307](#), [7309](#), [7311](#), [7467](#), [7468](#), [7470](#), [7471](#)
- \l__coffin_right_corner_dim . [14599](#), [14600](#), [14630](#), [14696](#), [14706](#), [14707](#)
- \l__coffin_scale_x_fp [14741](#), [14741](#), [14747](#), [14763](#), [14773](#), [14775](#), [14781](#), [14789](#)
- \l__coffin_scale_y_fp . [14741](#), [14742](#), [14749](#), [14774](#), [14775](#), [14779](#), [14791](#)
- \l__coffin_scaled_total_height_dim . [14743](#), [14743](#), [14778](#), [14783](#)
- \l__coffin_scaled_width_dim [14743](#), [14744](#), [14780](#), [14783](#)
- \l__coffin_sin_fp [14595](#), [14595](#), [14605](#), [14681](#), [14688](#)
- \l__coffin_slope_x_fp [6802](#), [6802](#), [7140](#), [7145](#), [7153](#), [7159](#)
- \l__coffin_slope_y_fp [6802](#), [6803](#), [7142](#), [7145](#), [7154](#), [7159](#)
- \l__coffin_top_corner_dim ... [14599](#), [14602](#), [14627](#), [14695](#), [14710](#), [14711](#)
- \l__coffin_x_dim [6809](#), [6809](#), [7089](#), [7098](#), [7118](#), [7121](#), [7128](#), [7135](#), [7137](#), [7148](#), [7163](#), [7252](#), [7256](#), [7275](#), [7283](#), [7489](#), [7541](#), [14652](#), [14654](#), [14658](#), [14660](#), [14664](#), [14669](#), [14795](#), [14797](#), [14801](#), [14804](#)
- \l__coffin_x_prime_dim .. [6809](#), [6811](#), [7252](#), [7256](#), [7541](#), [7544](#), [14666](#), [14670](#)
- \l__coffin_y_dim [6809](#), [6810](#), [7090](#), [7103](#), [7106](#), [7113](#), [7130](#), [7164](#), [7253](#), [7258](#), [7276](#), [7283](#), [7490](#), [7542](#), [14652](#), [14654](#), [14658](#), [14660](#), [14664](#), [14669](#), [14795](#), [14797](#), [14801](#), [14804](#)
- \l__coffin_y_prime_dim .. [6809](#), [6812](#), [7253](#), [7258](#), [7542](#), [7546](#), [14666](#), [14671](#)
- \l__exp_internal_tl [34](#), [857](#), [857](#), [861](#), [862](#), [1630](#), [1649](#), [1650](#), [1825](#), [1826](#)
- \l__expl_status_stack_tl [204](#)
- \l__file_internal_name_ior [166](#)
- \l__file_internal_name_tl [166](#), [9154](#), [9154](#), [9167](#), [9168](#), [9169](#), [9170](#), [9173](#), [9174](#), [9179](#), [9220](#), [9221](#), [9227](#), [9228](#), [9233](#), [9315](#), [9316](#), [9318](#), [9327](#), [9328](#), [9331](#)
- \l__file_internal_seq ... [9159](#), [9160](#), [9199](#), [9201](#), [9273](#), [9279](#), [9284](#), [9286](#)
- \l__file_internal_tl [9153](#), [9153](#), [9255](#), [9256](#)
- \l__file_saved_search_path_seq [9156](#), [9157](#), [9198](#), [9215](#)
- \l__file_search_path_seq [9155](#), [9155](#), [9198](#), [9200](#), [9201](#), [9204](#), [9215](#), [9263](#), [9264](#), [9269](#)
- \l__fp_division_by_zero_flag_token . [10004](#), [10005](#)
- \l__fp_invalid_operation_flag_token [10004](#), [10004](#)
- \l__fp_overflow_flag_token [10004](#), [10006](#)
- \l__fp_underflow_flag_token [10004](#), [10007](#)
- \l__ior_internal_tl [14828](#), [14847](#), [14850](#), [14854](#)
- \l__ior_stream_tl [9303](#), [9303](#), [9338](#), [9346](#), [9354](#)
- \l__iow_current_indentation_int [9484](#), [9486](#), [9547](#), [9589](#), [9604](#), [9626](#), [9632](#), [9634](#)

\l__iow_current_indentation_tl	9487 , 9489 , 9548 , 9587 , 9607 , 9627 , 9633	\l__msg_current_class_tl 7928 , 7929 , 7939 , 7976 , 7981 , 7984 , 7992 , 8021 , 8035
\l__iow_current_line_int 9484 , 9484 , 9549 , 9575 , 9576 , 9588 , 9594 , 9601 , 9620	\l__msg_hierarchy_seq 7931 , 7931 , 7959 , 7969 , 7974
\l__iow_current_line_tl 9487 , 9487 , 9550 , 9586 , 9592 , 9600 , 9606 , 9619 , 9621 , 9639	\l__msg_internal_tl 7641 , 7641 , 8351 , 8352 , 8354
\l__iow_current_word_int 9484 , 9485 , 9573 , 9575 , 9603	\l__msg_redirect_prop 7930 , 7930 , 7956 , 8001 , 8004
\l__iow_current_word_tl	.. 9487 , 9488 , 9566 , 9567 , 9574 , 9587 , 9593 , 9607	\l__peek_search_tl 3080 , 3080 , 3098 , 3119 , 3159 , 3169
\l__iow_line_start_bool 9492 , 9492 , 9551 , 9583 , 9585 , 9622	\l__peek_search_token 3079 , 3079 , 3097 , 3118 , 3136
\l__iow_newline_tl 9491 , 9491 , 9542 , 9543 , 9544 , 9546 , 9600 , 9619	\l__prop_internal_tl 127 , 6281 , 6281 , 6394 , 6397 , 6398 , 6414 , 6418 , 6516 , 6519 , 6520
\l__iow_stream_tl 9416 , 9416 , 9431 , 9439 , 9447	\l__seq_internal_a_tl 5400 , 5400 , 5430 , 5436 , 5441 , 5442 , 5510 , 5515 , 5529 , 5533
\l__iow_target_count_int 9483 , 9483 , 9545 , 9576	\l__seq_internal_b_tl 5400 , 5401 , 5506 , 5510 , 5532 , 5533
\l__iow_wrap_tl	9490 , 9490 , 9537 , 9540 , 9554 , 9556 , 9562 , 9599 , 9618 , 9638	\l__seq_remove_seq 5478 , 5478 , 5485 , 5488 , 5489 , 5491
\l__kernel_expl_bool 7 , 138 , 249 , 264 , 278 , 282 , 283	\l__tl_internal_a_tl	4895 , 4898 , 4900 , 4908
\l__keys_module_tl 8563 , 8563 , 8570 , 8573 , 8575 , 8603 , 8717 , 8881 , 8884 , 8886 , 8891 , 8894 , 8899 , 8917 , 8967 , 8970	\l__tl_internal_b_tl	4895 , 4899 , 4900 , 4909
\l__keys_no_value_bool 8564 , 8564 , 8580 , 8585 , 8623 , 8906 , 8911 , 8922 , 8932 , 8944 , 8979	\l__char_active_seq 53 , 2725 , 2725 , 2738 , 9165
\l__keys_property_tl 8566 , 8566 , 8591 , 8594 , 8597 , 8608 , 8619 , 8626 , 8627 , 8630 , 8634	\l__char_special_seq	.. 53 , 2725 , 2742 , 2743
\l__keys_unknown_clist 8567 , 8567 , 8895 , 8900 , 8976	\l__iow_line_count_int 166 , 9481 , 9481 , 9482 , 9546 , 9691
\l__keyval_key_tl 8447 , 8447 , 8492 , 8506 , 8510 , 8517	\l__iow_line_length_int 9690 , 9691
\l__keyval_parse_tl	.. 8449 , 8450 , 8466 , 8470 , 8488 , 8513 , 8522 , 8526 , 8535	\l__keys_choice_int	.. 156 , 8560 , 8560 , 8683 , 8686 , 8690 , 8691 , 8736 , 8739 , 8743 , 8744 , 9088 , 9098 , 9102 , 9103
\l__keyval_sanitise_tl 8449 , 8449 , 8462 , 8463 , 8464 , 8465 , 8468	\l__keys_choice_tl 156 , 8560 , 8561 , 8689 , 8742 , 9101
\l__keyval_value_tl 8447 , 8448 , 8519 , 8521 , 8524 , 8534 , 8536	\l__keys_key_tl 158 , 8562 , 8562 , 8649 , 8665 , 8916 , 8917 , 8978
\l__msg_class_loop_seq	.. 7932 , 7932 , 8022 , 8030 , 8039 , 8040 , 8043 , 8045	\l__keys_path_tl 158 , 8565 , 8565 , 8597 , 8603 , 8613 , 8616 , 8631 , 8642 , 8644 , 8646 , 8658 , 8660 , 8662 , 8672 , 8674 , 8677 , 8687 , 8709 , 8711 , 8716 , 8720 , 8725 , 8727 , 8730 , 8740 , 8752 , 8758 , 8789 , 8917 , 8926 , 8936 , 8946 , 8948 , 8951 , 8959 , 8964 , 8970 , 8994 , 8995 , 9069 , 9072 , 9074 , 9085 , 9093 , 9099 , 9105
\l__msg_class_tl 7928 , 7928 , 7944 , 7956 , 7977 , 7981 , 7984 , 7992 , 8031 , 8033 , 8035 , 8048	\l__keys_value_tl 158 , 8568 , 8568 , 8936 , 8943 , 8950 , 8980 , 8988
		\l__last_box 6771 , 6772

<code>\l_peek_token</code>	58, 3077, 3077, 3086, 3136, 3148, 3168, 3177, 15183, 15184, 15185, 15188
<code>\l_tmpa_bool</code>	38, 2148, 2148
<code>\l_tmpa_box</code>	131, 6649, 6649
<code>\l_tmpa_clist</code>	121, 6235, 6235
<code>\l_tmpa_coffin</code>	139, 6967, 6967
<code>\l_tmpa_dim</code>	82, 4404, 4404
<code>\l_tmpa_fp</code>	174, 13853, 13853
<code>\l_tmpa_int</code>	73, 4112, 4112
<code>\l_tmpa_muskip</code>	88, 4556, 4556
<code>\l_tmpa_prop</code>	127, 6301, 6301
<code>\l_tmpa_seq</code>	112, 5808, 5808
<code>\l_tmpa_skip</code>	85, 4494, 4494
<code>\l_tmpa_tl</code>	5, 102, 5311, 5311
<code>\l_tmpb_bool</code>	38, 2148, 2149
<code>\l_tmpb_box</code>	131, 6649, 6650
<code>\l_tmpb_clist</code>	121, 6235, 6236
<code>\l_tmpb_coffin</code>	139, 6967, 6968
<code>\l_tmpb_dim</code>	82, 4404, 4405
<code>\l_tmpb_fp</code>	174, 13853, 13854
<code>\l_tmpb_int</code>	73, 4112, 4113
<code>\l_tmpb_muskip</code>	88, 4556, 4557
<code>\l_tmpb_prop</code>	127, 6301, 6302
<code>\l_tmpb_seq</code>	112, 5808, 5809
<code>\l_tmpb_skip</code>	85, 4494, 4495
<code>\l_tmpb_tl</code>	102, 5311, 5312
<code>\l_tmpc_int</code>	4163, 4164
<code>\language</code>	420
<code>\lastbox</code>	577
<code>\lastkern</code>	510
<code>\lastlinefit</code>	690
<code>\lastnodetype</code>	671
<code>\lastpenalty</code>	616
<code>\lastskip</code>	511
<code>\lualua</code>	732
<code>\lccode</code>	639
<code>\leaders</code>	507
<code>\left</code>	475
<code>\lefthyphenmin</code>	531
<code>\leftskip</code>	533
<code>\leqno</code>	450
<code>\let</code>	60, 70, 307, 308, 320
<code>\limits</code>	467
<code>\linepenalty</code>	523
<code>\lineskip</code>	517
<code>\lineskiplimit</code>	518
<code>\linewidth</code>	6881, 6927
<code>\ln</code>	13125, 13128
<code>\long</code>	34, 310, 339
<code>\looseness</code>	535
<code>\lower</code>	572
<code>\lowercase</code>	611
<code>\lua_now:n</code>	185, 13975, 13993, 13995
<code>\lua_now:x</code>	13975
<code>\lua_now_x:n</code>	185, 4648, 13975, 13977, 13981, 13984, 13992, 13994
<code>\lua_now_x:x</code>	13975
<code>\lua_shipout:n</code> ..	185, 13975, 13997, 13999
<code>\lua_shipout:x</code>	13975
<code>\lua_shipout_x:n</code>	186, 13975, 13978, 13986, 13989, 13996, 13998
<code>\lua_shipout_x:x</code>	13975
<code>\luaescapestring</code>	40, 41
<code>\luatex_bodydir:D</code>	735, 753
<code>\luatex_catcodetable:D</code>	729, 749, 14038, 14039, 14044, 14052
<code>\luatex_directlua:D</code>	730, 1501, 13977
<code>\luatex_if_engine:F</code>	1479, 1504, 14017, 14054, 14082
<code>\luatex_if_engine:T</code> .	1478, 1503, 4643, 14026, 14068, 14090, 14096, 14105
<code>\luatex_if_engine:TF</code>	23, 1478, 1480, 1505, 13975
<code>\luatex_if_engine_p:</code>	23, 1478, 1487, 1509, 1600
<code>\luatex_initcatcodetable:D</code>	731, 750, 14013, 14032
<code>\luatex_latelua:D</code>	732, 751, 13978
<code>\luatex luatexversion:D</code>	733, 824
<code>\luatex_mathdir:D</code>	736, 754
<code>\luatex_pagedir:D</code>	737, 755
<code>\luatex_pardir:D</code>	738, 756
<code>\luatex_savecatcodetable:D</code>	734, 752, 14043, 14079
<code>\luatex_textdir:D</code>	739, 757
<code>\luatexbodydir</code>	753
<code>\luatexcatcodetable</code>	749
<code>\luatexinitcatcodetable</code>	750
<code>\luatexlatelua</code>	751
<code>\luatexmathdir</code>	754
<code>\luatexpagedir</code>	755
<code>\luatexpardir</code>	756
<code>\luatexsavecatcodetable</code>	752
<code>\luatextextdir</code>	757
<code>\luatexversion</code>	733
M	
<code>\M</code>	1895, 2819
<code>cm</code>	181

<code>em</code>	181	<code>\msg_class_set:nn</code>	8372, 8418, 8419
<code>mm</code>	181	<code>\msg_critical:nn</code>	7867
<code>\m@ne</code>	813	<code>\msg_critical:nnn</code>	7867
<code>\mag</code>	419	<code>\msg_critical:nnnn</code>	7867
<code>\mark</code>	421	<code>\msg_critical:nnnnn</code>	7867
<code>\marks</code>	647	<code>\msg_critical:nnnnnn</code>	144, 7867
<code>\mathaccent</code>	432	<code>\msg_critical:nnx</code>	7867
<code>\mathbin</code>	462	<code>\msg_critical:nnxx</code>	7867
<code>\mathchar</code>	433, 2885	<code>\msg_critical:nnxxx</code>	7867
<code>\mathchardef</code>	330	<code>\msg_critical:nnxxxx</code>	7867
<code>\mathchoice</code>	430	<code>\msg_critical_text:n</code>	142, 7807, 7808, 7870
<code>\mathclose</code>	463	<code>\msg_direct_interrupt:xxxxx</code>	8381, 8386
<code>\mathcode</code>	641	<code>\msg_direct_log:xx</code>	8381, 8387
<code>\mathdir</code>	736	<code>\msg_direct_term:xx</code>	8381, 8388
<code>\mathinner</code>	464	<code>\msg_error:nn</code>	7878
<code>\mathop</code>	465	<code>\msg_error:nnn</code>	7878
<code>\mathopen</code>	469	<code>\msg_error:nnnn</code>	7878
<code>\mathord</code>	470	<code>\msg_error:nnnnn</code>	7878
<code>\mathparagraph</code>	4137	<code>\msg_error:nnnnnn</code>	144, 7878
<code>\mathpunct</code>	471	<code>\msg_error:nnx</code>	7878
<code>\mathrel</code>	472	<code>\msg_error:nnxx</code>	7878
<code>\mathsection</code>	4136	<code>\msg_error:nnxxx</code>	7878
<code>\mathsurround</code>	483	<code>\msg_error:nnxxxx</code>	7878
<code>\maxdeadcycles</code>	553	<code>\msg_error_text:n</code>	142, 7807, 7809, 7885
<code>\maxdepth</code>	554	<code>\msg_fatal:nn</code>	7856
<code>\meaning</code>	613	<code>\msg_fatal:nnn</code>	7856
<code>\medmuskip</code>	484	<code>\msg_fatal:nnnn</code>	7856
<code>\message</code>	390	<code>\msg_fatal:nnnnn</code>	7856
<code>\MessageBreak</code>	77, 78, 79, 80, 81, 82, 83, 84, 148, 229	<code>\msg_fatal:nnnnnn</code>	143, 7856
<code>.meta:n</code>	153	<code>\msg_fatal:nnx</code>	7856
<code>.meta:nn</code>	154	<code>\msg_fatal:nnxx</code>	7856
<code>\middle</code>	695	<code>\msg_fatal:nnxxx</code>	7856
<code>\mkern</code>	437	<code>\msg_fatal:nnxxxx</code>	7856
<code>\mode_if_horizontal:</code>	2327	<code>\msg_fatal_text:n</code>	142, 7807, 7807, 7859
<code>\mode_if_horizontal:TF</code>	41, 2327	<code>\msg_generic_new:nn</code>	8381, 8383
<code>\mode_if_horizontal_p:</code>	41, 2327	<code>\msg_generic_new:nnn</code>	8381, 8382
<code>\mode_if_inner:</code>	2329	<code>\msg_generic_set:nn</code>	8381, 8385
<code>\mode_if_inner:TF</code>	41, 2329	<code>\msg_generic_set:nnn</code>	8381, 8384
<code>\mode_if_inner_p:</code>	41, 2329	<code>\msg_gset:nnn</code>	7670, 7693
<code>\mode_if_math:</code>	2331	<code>\msg_gset:nnnn</code>	142, 7670, 7673, 7686, 7694
<code>\mode_if_math:TF</code>	41, 2331, 4125	<code>\msg_if_exist:nn</code>	7644
<code>\mode_if_math_p:</code>	41, 2331	<code>\msg_if_exist:nnT</code>	7651, 7661
<code>\mode_if_vertical:</code>	2325	<code>\msg_if_exist:nnTF</code>	142, 7644, 7935
<code>\mode_if_vertical:TF</code>	41, 2325	<code>\msg_if_exist_p:nn</code>	142, 7644
<code>\mode_if_vertical_p:</code>	41, 2325	<code>\msg_info:nn</code>	7907
<code>\month</code>	623	<code>\msg_info:nnn</code>	7907
<code>\moveleft</code>	573	<code>\msg_info:nnnn</code>	7907
<code>\moveright</code>	574	<code>\msg_info:nnnnn</code>	7907
<code>\msg_class_new:nn</code>	8371, 8372	<code>\msg_info:nnnnnn</code>	144, 7907
		<code>\msg_info:nnx</code>	7907

<code>\msg_info:nnxx</code>	7907	<code>\msg_warning:nnn</code>	7899
<code>\msg_info:nnxxx</code>	7907	<code>\msg_warning:nnnn</code>	7899
<code>\msg_info:nnxxxx</code>	7907 , 8106	<code>\msg_warning:nnnnn</code>	7899
<code>\msg_info_text:n</code> ..	143 , 7807 , 7811 , 7911	<code>\msg_warning:nnnnnn</code>	144 , 7899
<code>\msg_interrupt:nnn</code>	146 , 7734 , 7734 , 7858 , 7869 , 7884 , 8393 , 8416	<code>\msg_warning:nnx</code>	7899
<code>\msg_interrupt:xxx</code>	8413	<code>\msg_warning:nnxx</code>	7899
<code>\msg_line_context:</code>		<code>\msg_warning:nnxxx</code>	7899
<code>\msg_line_number:</code>	142 , 1191 , 1191 , 1210 , 7666 , 7727 , 7728	<code>\msg_warning:nnxxxx</code>	7899 , 8105
.....	142 , 7727 , 7727 , 7732 , 8549	<code>\msg_warning_text:n</code>	143 , 7807 , 7810 , 7903
<code>\msg_log:n</code> ...	147 , 7792 , 7792 , 7909 , 8414	<code>\mskip</code>	434
<code>\msg_log:nn</code>	7915 , 8379	<code>\muexpr</code>	683
<code>\msg_log:nnn</code>	7915	<code>.multichoice:</code>	154
<code>\msg_log:nnnn</code>	7915	<code>.multichoices:nn</code>	154
<code>\msg_log:nnnnn</code>	7915	<code>.multichoices:on</code>	154
<code>\msg_log:nnnnnn</code>	144 , 7915	<code>.multichoices:Vn</code>	154
<code>\msg_log:nnx</code>	7915 , 8378	<code>.multichoices:xn</code>	154
<code>\msg_log:nnxx</code>	7915 , 8377	<code>\multiply</code>	335
<code>\msg_log:nnxxx</code>	7915 , 8376	<code>\muskip</code>	631
<code>\msg_log:nnxxxx</code>	7915 , 8375	<code>\muskip_add:cn</code>	4536
<code>\msg_log:x</code>	8413	<code>\muskip_add:Nn</code> .	87 , 4536 , 4536 , 4538 , 4539
<code>\msg_new:nnn</code>	7670 , 7675 , 8057	<code>\muskip_const:cn</code>	4506
<code>\msg_new:nnnn</code> .	141 , 7670 , 7670 , 7676 , 8055	<code>\muskip_const:Nn</code> 86 , 4506 , 4506 , 4511 , 4554 , 4555
<code>\msg_newline:</code>	8409 , 8410	<code>\muskip_eval:n</code>	87 , 4546 , 4546
<code>\msg_none:nn</code>	7921	<code>\muskip_gadd:cn</code>	4536
<code>\msg_none:nnn</code>	7921	<code>\muskip_gadd:Nn</code>	87 , 4536 , 4538 , 4540
<code>\msg_none:nnnn</code>	7921	<code>\muskip_gset:cn</code>	4525
<code>\msg_none:nnnnn</code>	7921	<code>\muskip_gset:Nn</code> .	87 , 4509 , 4525 , 4527 , 4529
<code>\msg_none:nnnnnn</code>	145 , 7921	<code>\muskip_gset_eq:cc</code>	4530
<code>\msg_none:nnx</code>	7921	<code>\muskip_gset_eq:cN</code>	4530
<code>\msg_none:nnxx</code>	7921	<code>\muskip_gset_eq:Nc</code>	4530
<code>\msg_none:nnxxx</code>	7921	<code>\muskip_gset_eq:NN</code> 87 , 4530 , 4533 , 4534 , 4535
<code>\msg_none:nnxxxx</code>	7921	<code>\muskip_gsub:cn</code>	4536
<code>\msg_redirect_class:nn</code> .	145 , 8007 , 8007	<code>\muskip_gsub:Nn</code>	87 , 4536 , 4543 , 4545
<code>\msg_redirect_module:nnn</code>	146 , 8007 , 8009	<code>\muskip_gzero:c</code>	4512
<code>\msg_redirect_name:nnn</code> .	146 , 7998 , 7998	<code>\muskip_gzero:N</code> .	86 , 4512 , 4514 , 4516 , 4520
<code>\msg_see_documentation_text:n</code>	143 , 7812 , 7812 , 7862 , 7873 , 7888 , 8396	<code>\muskip_gzero_new:c</code>	4517
<code>\msg_set:nnn</code>	7670 , 7684 , 8061	<code>\muskip_gzero_new:N</code> .	86 , 4517 , 4519 , 4522
<code>\msg_set:nnnn</code> .	142 , 7670 , 7677 , 7685 , 8059	<code>\muskip_if_exist:c</code>	4524
<code>\msg_term:n</code> ...	147 , 7792 , 7798 , 7901 , 8415	<code>\muskip_if_exist:cTF</code>	4523
<code>\msg_term:x</code>	8413	<code>\muskip_if_exist:N</code>	4523
<code>\msg_trace:nn</code>	8374 , 8379	<code>\muskip_if_exist:NTF</code> .	86 , 4518 , 4520 , 4523
<code>\msg_trace:nnx</code>	8374 , 8378	<code>\muskip_if_exist_p:c</code>	4523
<code>\msg_trace:nnxx</code>	8374 , 8377	<code>\muskip_if_exist_p:N</code>	86 , 4523
<code>\msg_trace:nnxxx</code>	8374 , 8376	<code>\muskip_new:c</code>	4498
<code>\msg_trace:nnxxxx</code>	8374 , 8375	<code>\muskip_new:N</code> .	86 , 4498 , 4499 , 4505 , 4508 , 4518 , 4520 , 4556 , 4557 , 4558 , 4559
<code>\msg_two_newlines:</code>	8409 , 8411	<code>\muskip_set:cn</code>	4525
<code>\msg_warning:nn</code>	7899	<code>\muskip_set:Nn</code> .	87 , 4525 , 4525 , 4527 , 4528

\muskip_set_eq:cc	4530		
\muskip_set_eq:cN	4530		
\muskip_set_eq:Nc	4530		
\muskip_set_eq:NN	87, 4530, 4530, 4531, 4532		
\muskip_show:c	4550		
\muskip_show:N	88, 4550, 4550, 4551		
\muskip_show:n	88, 4552, 4552		
\muskip_sub:cn	4536		
\muskip_sub:Nn	87, 4536, 4541, 4543, 4544		
\muskip_use:c	4548		
\muskip_use:N	87, 4547, 4548, 4548, 4549		
\muskip_zero:c	4512		
\muskip_zero:N	86, 4512, 4512, 4514, 4515, 4518		
\muskip_zero_new:c	4517		
\muskip_zero_new:N	86, 4517, 4517, 4521		
\muskipdef	329		
\mutoglu	689		
\my_map_dbl:nn	1		
N			
\N	2044		
in	181		
ln	179		
\newbox	6564		
\newcatcodetable	14031		
\newcount	3462		
\newdimen	4188		
\newlinechar	90, 385		
\newmuskip	4502		
\newread	9345		
\newskip	4412		
\newwrite	9438		
inf	180		
\noalign	353		
\noboundary	488		
\noexpand	36, 40, 41, 173, 176, 179, 181, 182, 191, 194, 195, 196, 198, 200, 201, 210, 212, 213, 214, 215, 216, 288, 290, 295, 297, 346		
\noindent	514		
\nolimits	468		
\nonscript	448		
\nonstopmode	411		
\nulldelimiterspace	481		
\nullfont	599		
\number	608		
\numexpr	680		
O			
\O	15173		
\omit	354		
\openin	380		
\openout	381		
\or	377		
\or:	24, 74, 766, 768, 1302, 1303, 1304, 1305, 1306, 1307, 1308, 1309, 1310, 1607, 3868, 3869, 3870, 3871, 3872, 3873, 3874, 3875, 3876, 3877, 3878, 3879, 3880, 3881, 3882, 3883, 3884, 3885, 3886, 3887, 3888, 3889, 3890, 3891, 3892, 9777, 9778, 9779, 9929, 11091, 11276, 11277, 11278, 11315, 11316, 11725, 11726, 11727, 11855, 11939, 12025, 12026, 12027, 12028, 12029, 12030, 12031, 12032, 12033, 12110, 12113, 12604, 12629, 12635, 12636, 12637, 12638, 12639, 12786, 12821, 12823, 12831, 12921, 12925, 12926, 12927, 12928, 12929, 12930, 12931, 12932, 12933, 12934, 12941, 12942, 12943, 12944, 12945, 12946, 12947, 12948, 12949, 12950, 12957, 12958, 12959, 12960, 12961, 12962, 12963, 12964, 12965, 12966, 12973, 12974, 12975, 12976, 12977, 12978, 12979, 12980, 12981, 12982, 12989, 12990, 12991, 12992, 12993, 12994, 12995, 12996, 12997, 12998, 13005, 13006, 13007, 13008, 13009, 13010, 13011, 13012, 13013, 13014, 13032, 13078, 13081, 13090, 13201, 13216, 13263, 13269, 13278, 13284, 13293, 13299, 13308, 13314, 13323, 13329, 13338, 13344, 13576, 13577, 13588, 13626, 13627, 13637, 13694, 13695		
cos	180		
cot	180		
round	180		
round+	180		
round-	180		
round0	180		
\outer	340		
\output	555		
\outputpenalty	565		
\over	442		
\overfullrule	593		
\overline	473		
\overwithdelims	443		

P

\P	1897, 10345, 10399	\pdftex_if_engine_p:	23, 1478, 1488, 1498, 1510, 1601
bp	181	\pdftex_pdfcolorstack:D	709
sp	181	\pdftex_pdfcompresslevel:D	710
\PackageError	75, 145, 226	\pdftex_pdfcreationdate:D	708
\pagedepth	557	\pdftex_pdfdecimaldigits:D	711
\pagedir	737	\pdftex_pdfhorigin:D	712
\pagediscards	698	\pdftex_pdfinfo:D	713
\pagefilllstretch	561	\pdftex_pdflastxform:D	714
\pagefillstretch	560	\pdftex_pdfliteral:D	715
\pagefilstretch	559	\pdftex_pdfminorversion:D	716
\pagegoal	563	\pdftex_pdfobjcompresslevel:D	717
\pageshrink	562	\pdftex_pdfoutput:D	718
\pagestretch	558	\pdftex_pdfpkresolution:D	723
\pagetotal	564	\pdftex_pdfrefxform:D	719
\par	513, 6724, 6725, 6727, 6729, 6731, 6736, 6742, 6755	\pdftex_pdfrestore:D	720
\pardir	738	\pdftex_pdfsave:D	721
\parfillskip	544	\pdftex_pdfsetmatrix:D	722
\parindent	537	\pdftex_pdftextrevision:D	724
\parshape	529	\pdftex_pdfvorigin:D	725
\parshapedimen	679	\pdftex_pdfxform:D	726
\parshapeindent	677	\pdftex_strcmp:D	727, 1515, 1521, 1526, 2510, 2517, 2534, 2561, 2570, 2834, 4458, 10765, 10778, 10934, 13070
\parshapelength	678	\pdftexrevision	724
\parskip	536	\pdfvorigin	725
\patterns	619	\pdfxform	726
\pausing	406	\peek_after:NN	3306, 3307
\pdf@strcmp	60	\peek_after:Nw	58, 3085, 3085, 3110, 3128, 3178, 3307
\pdfcolorstack	709	\peek_catcode:NTF	58, 3202
\pdfcompresslevel	710	\peek_catcode_ignore_spaces:NTF	58, 3202
\pdfcreationdate	708	\peek_catcode_remove:NTF	59, 3202
\pdfdecimaldigits	711	\peek_catcode_remove_ignore_spaces:NTF	59, 3202
\pdfhorigin	712	\peek_charcode:NTF	59, 3218
\pdfinfo	713	\peek_charcode_ignore_spaces:NTF	59, 3218
\pdflastxform	714	\peek_charcode_remove:NTF	59, 3218
\pdfliteral	715	\peek_charcode_remove_ignore_spaces:NTF	60, 3218
\pdfminorversion	716	\peek_gafter:NN	3306, 3308
\pdfobjcompresslevel	717	\peek_gafter:Nw	58, 3085, 3087, 3308
\pdfoutput	718	\peek_meaning:NTF	60, 3234
\pdfpkresolution	723	\peek_meaning_ignore_spaces:NTF	60, 3234
\pdfrefxform	719	\peek_meaning_remove:NTF	60, 3234
\pdfrestore	720	\peek_meaning_remove_ignore_spaces:NTF	60, 3234
\pdfsave	721	\peek_N_type:F	15212
\pdfsetmatrix	722	\peek_N_type:T	15210
\pdfstrcmp	34, 60, 70, 77, 92, 727		
\pdftex_if_engine:F	1482, 1493, 1507		
\pdftex_if_engine:T	1481, 1492, 1506		
\pdftex_if_engine:TF	23, 1478, 1483, 1494, 1508, 3487		

- \peek_N_type:TF 198, 15172, 15208
- \penalty 614
- \postdisplaypenalty 461
- \predisplaydirection 705
- \predisdisplaypenalty 460
- \predisplaysize 459
- \pretolerance 540
- \prevdepth 587
- \prevgraf 546
- \prg_break: 2600, 2602
- \prg_case_dim:nnn 4564, 4565
- \prg_case_int:nnn 2467, 2468
- \prg_case_str:nnn 2467, 2469
- \prg_case_str:onn 2467, 2470
- \prg_case_str:xxn 2467, 2471
- \prg_case_tl:cn 2467, 2473
- \prg_case_tl:Nnn 2467, 2472
- \prg_define_quicksort:nnn 2373, 2374, 2457
- \prg_do_nothing:
 - . . . 9, 1007, 1008, 1471, 1512, 1512,
 - 1565, 1964, 1969, 4743, 4757, 4817,
 - 4822, 5432, 5439, 5638, 5640, 5961,
 - 9971, 10023, 10057, 10083, 10091,
 - 11285, 13887, 14528, 14532, 14539
- \prg_new_conditional:Nnn 910,
 - 912, 2091, 2539, 2547, 2559, 2568, 9381
- \prg_new_conditional:Npnn
 - 35, 897, 899, 1431,
 - 1513, 1519, 2091, 2120, 2154, 2325,
 - 2327, 2329, 2331, 2745, 2750, 2755,
 - 2760, 2767, 2773, 2778, 2783, 2788,
 - 2793, 2798, 2803, 2808, 2813, 2827,
 - 2841, 2846, 2869, 2878, 2888, 2906,
 - 2930, 2948, 2966, 2984, 2996, 3005,
 - 3025, 3560, 3608, 3630, 3638, 4266,
 - 4271, 4455, 4467, 4561, 4838, 4848,
 - 4860, 4881, 4883, 4929, 5191, 5210,
 - 5229, 5264, 5270, 5285, 5373, 6428,
 - 6624, 6626, 6636, 6813, 7644, 8956,
 - 8997, 9003, 9996, 10761, 11423,
 - 11436, 13861, 13868, 14580, 15034
- \prg_new_eq_conditional:NNn 37,
 - 998, 1000, 2091, 2152, 2153, 3512,
 - 3513, 4208, 4209, 4432, 4433, 4523,
 - 4524, 4639, 4640, 5377, 5379, 5381,
 - 5459, 5460, 5521, 5523, 5796, 5797,
 - 5798, 5799, 5800, 5801, 5872, 5873,
 - 6065, 6066, 6258, 6259, 6260, 6261,
 - 6422, 6423, 6424, 6426, 6537, 6538,
 - 6539, 6540, 6592, 6593, 11421, 11422
- \prg_new_map_functions:Nn 2463, 2464
- \prg_new_protected_conditional:Nnn .
 - 910, 916, 2091
- \prg_new_protected_conditional:Npnn
 - 35, 897, 903, 2091, 4895,
 - 4916, 5525, 5637, 5639, 5647, 5649,
 - 5651, 5653, 5970, 5982, 5984, 6067,
 - 6071, 6362, 6372, 6459, 9218, 9320
- \prg_quicksort:n 2456
- \prg_quicksort_compare:nnTF . . 2459, 2461
- \prg_quicksort_function:n 2459, 2460
- \prg_replicate:nn 41, 2281,
 - 2281, 8238, 9634, 13177, 13228,
 - 13235, 13398, 13633, 13660, 13668
- \prg_return_false:
 - 37, 893, 895, 1102, 1107,
 - 1120, 1125, 1133, 1150, 1434, 1517,
 - 1522, 1529, 2091, 2125, 2159, 2326,
 - 2328, 2330, 2332, 2544, 2552, 2565,
 - 2574, 2748, 2753, 2758, 2763, 2770,
 - 2776, 2781, 2786, 2791, 2796, 2801,
 - 2806, 2811, 2816, 2837, 2844, 2849,
 - 2854, 2891, 2894, 2913, 2916, 2933,
 - 2936, 2951, 2954, 2969, 2972, 3028,
 - 3047, 3064, 3073, 3558, 3585, 3590,
 - 3613, 3635, 3641, 4269, 4290, 4305,
 - 4306, 4462, 4470, 4562, 4853, 4865,
 - 4878, 4888, 4905, 4920, 5203, 5226,
 - 5242, 5250, 5260, 5280, 5294, 5539,
 - 5562, 5973, 5989, 6081, 6370, 6380,
 - 6447, 6466, 6625, 6627, 6637, 6819,
 - 6821, 7647, 8961, 9001, 9007, 9222,
 - 9329, 9391, 10001, 10774, 10786,
 - 11431, 11444, 13866, 13871, 14583
- \prg_return_true: 37, 893,
 - 893, 1105, 1122, 1130, 1135, 1148,
 - 1153, 1434, 1517, 1522, 1527, 2091,
 - 2123, 2157, 2326, 2328, 2330, 2332,
 - 2542, 2550, 2563, 2572, 2748, 2753,
 - 2758, 2763, 2770, 2776, 2781, 2786,
 - 2791, 2796, 2801, 2806, 2811, 2816,
 - 2835, 2844, 2852, 2908, 2910, 3045,
 - 3071, 3585, 3611, 3633, 3643, 4269,
 - 4306, 4460, 4471, 4562, 4851, 4863,
 - 4876, 4886, 4902, 4920, 5201, 5224,
 - 5240, 5258, 5282, 5293, 5543, 5565,
 - 5976, 5992, 6081, 6368, 6378, 6445,
 - 6464, 6625, 6627, 6637, 6818, 7647,
 - 8960, 9000, 9006, 9223, 9332, 9386,
 - 9389, 9395, 9999, 10769, 10792,

11433, 11442, 13866, 13871, 14584	\prop_get:NoNTF	6459
\prg_set_conditional:Nnn . 910, 910, 2091	\prop_get:NVN	6332
\prg_set_conditional:Npnn 35,	\prop_get:NVNTF	6459
897, 897, 1099, 1111, 1127, 1139, 2091	\prop_get_gdel:NnN	6525, 6526
\prg_set_eq_conditional:NnN	\prop_gget:cnN	6515
. 37, 998, 998, 2091	\prop_gget:cVN	6515
\prg_set_map_functions:Nn . . . 2463, 2465	\prop_gget:NnN . . . 6515, 6517, 6522, 6523	
\prg_set_protected_conditional:Nnn .	\prop_gget:NVN	6515
. 910, 914, 2091	\prop_gget_aux:Nnnn	6515
\prg_set_protected_conditional:Npnn	\prop_gpop:cnN	6340
. 35, 897, 901, 2091	\prop_gpop:cnNTF	6362
\prg_stepwise_function:nnnN . 2475, 2476	\prop_gpop:coN	6340
\prg_stepwise_inline:nnnn . . . 2475, 2477	\prop_gpop:NnN	
\prg_stepwise_variable:nnnNn 2475, 2478	123, 6340, 6349, 6360, 6361, 6372, 6526	
\prop_clear:c 6285, 6286, 7199	\prop_gpop:NnNF	6386
\prop_clear:N 122, 6285, 6285	\prop_gpop:NnNT	6385
\prop_clear_new:c	\prop_gpop:NnNTF	125, 6362, 6387
. 6289, 6290, 6848, 6849, 8425	\prop_gpop:NoN	6340
\prop_clear_new:N 122, 6289, 6289	\prop_gput:ccx	6533
\prop_del:cn 6542, 6545	\prop_gput:cnn	6388
\prop_del:cV 6542, 6546	\prop_gput:cno	6388
\prop_del:Nn 6542, 6543	\prop_gput:cnV	6388
\prop_del:NV 6542, 6544	\prop_gput:cnx	6388
\prop_display:c 6511, 6513	\prop_gput:con	6388
\prop_display:N 6511, 6512	\prop_gput:coo	6388
\prop_gclear:c 6285, 6288	\prop_gput:cVn	6388
\prop_gclear:N 122, 6285, 6287	\prop_gput:cVV	6388
\prop_gclear_new:c 6289, 6292	\prop_gput:Nnn	
\prop_gclear_new:N 122, 6289, 6291	123, 6388, 6390, 6404, 6406, 6534, 9306	
\prop_gdel:cn 6542, 6549	\prop_gput:Nno	6388
\prop_gdel:cV 6542, 6550	\prop_gput:NnV	6388
\prop_gdel:Nn 6542, 6547	\prop_gput:Nnx	6388
\prop_gdel:NV 6542, 6548	\prop_gput:Non	6388
\prop_get:cn 14878	\prop_gput:Noo	6388
\prop_get:cnN 6332	\prop_gput:NVn	6388, 9355, 9448
\prop_get:cnNF 6977	\prop_gput:NVV	6388
\prop_get:cnNT 8031	\prop_gput_if_new:cnn	6408
\prop_get:cnNTF 6459, 7976	\prop_gput_if_new:Nnn 123, 6408, 6410, 6421	
\prop_get:coN 6332	\prop_gremove:cn	6316, 6549
\prop_get:coNTF 6459	\prop_gremove:cV	6316, 6550
\prop_get:cVN 6332	\prop_gremove:Nn	
\prop_get:cVNTF 6459	. . . 124, 6316, 6322, 6330, 6331, 6547	
\prop_get:Nn 194, 14878, 14878, 14890	\prop_gremove:NV	6316, 6548, 9363, 9456
\prop_get:NnN	\prop_gset_eq:cc	6293, 6300, 7001, 7003
. . . . 123, 6332, 6332, 6338, 6339,	\prop_gset_eq:cN	6293, 6299, 6850, 6852
6459, 6519, 7422, 7426, 7505, 7509	\prop_gset_eq:Nc	6293, 6298
\prop_get:NnNF 6469, 6472	\prop_gset_eq:NN	122, 6293, 6297
\prop_get:NnNT 6468, 6471	\prop_if_empty:c	6426
\prop_get:NnNTF 125, 6459, 6470, 6473, 7956	\prop_if_empty:CTF	6424
\prop_get:NoN 6332	\prop_if_empty:N	6424

\prop_if_empty:NTF .	124 , 6424 , 8254 , 9375	\prop_map_tokens:cn	14863
\prop_if_empty_p:c	6424	\prop_map_tokens:Nn	
\prop_if_empty_p:N	124 , 6424		193 , 14863 , 14863 , 14877
\prop_if_eq:cc	6540	\prop_new:c	6283 , 6284 , 7821
\prop_if_eq:ccTF	6536	\prop_new:N	122 , 6283 , 6283 ,
\prop_if_eq:cN	6538		6301 , 6302 , 6303 , 6304 , 6785 , 6790 ,
\prop_if_eq:cNTF	6536		7348 , 7389 , 7930 , 9304 , 9417 , 14597
\prop_if_eq:Nc	6539	\prop_pop:cnN	6340
\prop_if_eq:NcTF	6536	\prop_pop:cnNTF	6362
\prop_if_eq:NN	6537	\prop_pop:coN	6340
\prop_if_eq:NNTF	6536	\prop_pop:NnN	
\prop_if_eq_p:cc	6536		123 , 6340 , 6340 , 6358 , 6359 , 6362
\prop_if_eq_p:cN	6536	\prop_pop:NnNF	6383
\prop_if_eq_p:Nc	6536	\prop_pop:NnNT	6382
\prop_if_eq_p:NN	6536	\prop_pop:NnNTF	125 , 6362 , 6384
\prop_if_exist:c	6423	\prop_pop:NoN	6340
\prop_if_exist:cTF	6422	\prop_put:cnn	6388 , 7029 , 8020 , 8037
\prop_if_exist:N	6422	\prop_put:cno	6388
\prop_if_exist:NTF	124 , 6422	\prop_put:cnV	6388
\prop_if_exist_p:c	6422	\prop_put:cnx	
\prop_if_exist_p:N	124 , 6422		6388 , 7035 , 7037 , 7039 , 7041 , 7046 ,
\prop_if_in:ccTF	6528		7051 , 7056 , 7063 , 7070 , 7294 , 14659 ,
\prop_if_in:cnTF	6428		14726 , 14734 , 14796 , 14810 , 14817
\prop_if_in:coTF	6428	\prop_put:con	6388
\prop_if_in:cVTF	6428	\prop_put:coo	6388
\prop_if_in:Nn	6428	\prop_put:cVn	6388
\prop_if_in:NnF	6455 , 6456 , 6530	\prop_put:cVV	6388
\prop_if_in:NnT	6453 , 6454 , 6529	\prop_put:Nnn	123 ,
\prop_if_in:NnTF	124 , 6428 , 6457 , 6458 , 6531		6388 , 6388 , 6400 , 6402 , 6786 , 6787 ,
\prop_if_in:NoTF	6428		6788 , 6789 , 7349 , 7351 , 7353 , 7355 ,
\prop_if_in:NVTF	6428		7357 , 7359 , 7361 , 7363 , 7365 , 7367 ,
\prop_if_in_p:cn	6428		7369 , 7371 , 7373 , 7375 , 7377 , 7379 ,
\prop_if_in_p:co	6428		7381 , 7383 , 8004 , 9419 , 9420 , 9421
\prop_if_in_p:cV	6428	\prop_put:Nno	6388 , 6792 , 6793 , 6794 ,
\prop_if_in_p:Nn	124 , 6428 , 6451 , 6452		6796 , 6797 , 6798 , 6799 , 6800 , 6801
\prop_if_in_p:No	6428	\prop_put:NnV	6388
\prop_if_in_p:NV	6428	\prop_put:Nnx	6388 ,
\prop_map_break:	126 , 6478 , 6483 , 6498 ,		14640 , 14642 , 14645 , 14647 , 14653
	6501 , 6501 , 6502 , 6504 , 14867 , 14872	\prop_put:Non	6388
\prop_map_break:n	126 , 6501 , 6503	\prop_put:Noo	6388
\prop_map_function:cc	6474	\prop_put:NVn	6388
\prop_map_function:cN	6474 , 7568	\prop_put:NVV	6388
\prop_map_function:Nc	6474	\prop_put_if_new:cnn	6408
\prop_map_function:NN		\prop_put_if_new:Nnn	123 , 6408 , 6408 , 6420
	125 , 6474 , 6474 , 6488 , 6489 , 6508 , 9378	\prop_remove:cn	6316 , 6545 , 8016
\prop_map_inline:cn	6490 , 7270 ,	\prop_remove:cV	6316 , 6546
	7289 , 14607 , 14609 , 14632 , 14634 ,	\prop_remove:Nn	124 , 6316 , 6316 , 6328 ,
	14699 , 14759 , 14761 , 14765 , 14767		6329 , 6543 , 7469 , 7472 , 7476 , 8001
\prop_map_inline:Nn	125 , 6490 ,	\prop_remove:NV	6316 , 6544
	6490 , 6500 , 7474 , 7483 , 14612 , 14716	\prop_set_eq:cc	6293 , 6296 , 6994 , 6996 , 7229

- \prop_set_eq:cN .. [6293](#), [6295](#), [6987](#), [6989](#)
- \prop_set_eq:Nc [6293](#), [6294](#), [7464](#)
- \prop_set_eq:NN [122](#), [6293](#), [6293](#)
- \prop_show:c [6505](#), [6513](#)
- \prop_show:N .. [126](#), [6505](#), [6505](#), [6510](#), [6512](#)
- \protected ... [110](#), [124](#), [140](#), [152](#), [159](#),
[164](#), [220](#), [247](#), [280](#), [286](#), [293](#), [707](#), [3013](#)
- \protected@edef [9540](#)
- \ProvidesClass [161](#)
- \ProvidesExplClass [6](#), [139](#), [159](#)
- \ProvidesExplFile [6](#), [139](#), [164](#)
- \ProvidesExplPackage [6](#),
[139](#), [140](#), [152](#), [157](#), [304](#), [762](#), [1626](#),
[2085](#), [2483](#), [2607](#), [3379](#), [4177](#), [4591](#),
[5391](#), [5832](#), [6276](#), [6556](#), [6778](#), [7604](#),
[7637](#), [8441](#), [9129](#), [9715](#), [13971](#), [14149](#)
- \ProvidesFile [166](#)
- \ProvidesPackage [48](#), [154](#)
- Q**
- \q__tl_act_mark [5074](#), [5077](#), [5094](#)
- \q__tl_act_stop [5074](#), [5077](#), [5081](#), [5090](#),
[5092](#), [5098](#), [5103](#), [5106](#), [5110](#), [5113](#)
- \q_tl_act_mark [2581](#), [2581](#)
- \q_tl_act_stop [2581](#), [2582](#)
- \q_mark [45](#),
[1086](#), [1087](#), [1090](#), [1091](#), [1092](#), [1908](#),
[1909](#), [1911](#), [1915](#), [1918](#), [1943](#), [1952](#),
[1966](#), [1974](#), [1977](#), [1986](#), [2001](#), [2023](#),
[2051](#), [2054](#), [2062](#), [2166](#), [2167](#), [2168](#),
[2169](#), [2172](#), [2173](#), [2174](#), [2175](#), [2176](#),
[2177](#), [2178](#), [2488](#), [2489](#), [3573](#), [3576](#),
[4800](#), [4809](#), [4813](#), [4824](#), [5015](#), [5016](#),
[5019](#), [5022](#), [5023](#), [5029](#), [5043](#), [5044](#),
[5050](#), [5054](#), [5056](#), [5059](#), [5748](#), [5749](#),
[5760](#), [5763](#), [5884](#), [5893](#), [5898](#), [5953](#),
[5963](#), [5967](#), [5991](#), [6043](#), [6049](#), [6062](#),
[6111](#), [6119](#), [6207](#), [6208](#), [6217](#), [6218](#),
[6310](#), [6312](#), [6313](#), [7961](#), [7962](#), [7967](#),
[7970](#), [10353](#), [10355](#), [14545](#), [14583](#),
[14584](#), [14593](#), [15189](#), [15190](#), [15197](#)
- \q_nil [878](#), [881](#), [2165](#),
[2167](#), [2168](#), [2172](#), [2173](#), [2174](#), [2175](#),
[2176](#), [2177](#), [2377](#), [2381](#), [2488](#), [2488](#),
[2541](#), [2562](#), [3981](#), [4003](#), [4862](#), [4874](#),
[4875](#), [5042](#), [5046](#), [5064](#), [5067](#), [5070](#),
[5155](#), [5156](#), [8468](#), [8476](#), [8480](#), [8497](#),
[8505](#), [8506](#), [8509](#), [8520](#), [8527](#), [8532](#)
- \q_no_value [45](#),
[2222](#), [2488](#), [2490](#), [2549](#), [2571](#), [5553](#),
[5561](#), [5573](#), [5595](#), [5936](#), [5951](#), [6336](#),
[6347](#), [6356](#), [8496](#), [8502](#), [8705](#), [9194](#)
- \q_recursion_stop
. [4](#), [46](#), [880](#), [883](#), [947](#), [1011](#), [1535](#),
[1546](#), [1554](#), [1892](#), [2492](#), [2493](#), [3622](#),
[4313](#), [4934](#), [5885](#), [6148](#), [6184](#), [8468](#)
- \q_recursion_tail
. [4](#), [46](#), [947](#), [952](#), [1011](#), [1030](#), [2492](#),
[2492](#), [2496](#), [2502](#), [2511](#), [2518](#), [2528](#),
[2535](#), [4947](#), [4964](#), [4973](#), [5885](#), [6097](#),
[6111](#), [6130](#), [6148](#), [6184](#), [6433](#), [6477](#),
[6482](#), [6497](#), [8468](#), [14866](#), [14871](#), [15139](#)
- \q_stop [45](#), [879](#), [882](#),
[963](#), [967](#), [983](#), [988](#), [993](#), [1041](#), [1045](#),
[1050](#), [1055](#), [1060](#), [1088](#), [1090](#), [1091](#),
[1092](#), [1912](#), [1918](#), [1947](#), [1974](#), [1978](#),
[1982](#), [1990](#), [1996](#), [2005](#), [2023](#), [2057](#),
[2062](#), [2170](#), [2178](#), [2222](#), [2225](#), [2349](#),
[2351](#), [2363](#), [2365](#), [2369](#), [2377](#), [2381](#),
[2449](#), [2488](#), [2491](#), [2830](#), [2832](#), [2874](#),
[2883](#), [2887](#), [2899](#), [2905](#), [2921](#), [2929](#),
[2941](#), [2947](#), [2959](#), [2965](#), [2977](#), [2983](#),
[2990](#), [2995](#), [3001](#), [3011](#), [3015](#), [3031](#),
[3034](#), [3037](#), [3059](#), [3253](#), [3260](#), [3269](#),
[3278](#), [3552](#), [3568](#), [3570](#), [3574](#), [3582](#),
[4049](#), [4086](#), [4279](#), [4305](#), [4384](#), [4389](#),
[4471](#), [4473](#), [4809](#), [4824](#), [5017](#), [5019](#),
[5024](#), [5026](#), [5048](#), [5070](#), [5149](#), [5151](#),
[5165](#), [5180](#), [5188](#), [5190](#), [5198](#), [5217](#),
[5239](#), [5358](#), [5360](#), [5385](#), [5573](#), [5576](#),
[5584](#), [5585](#), [5750](#), [5760](#), [5763](#), [5765](#),
[5938](#), [5941](#), [5953](#), [5956](#), [5964](#), [5967](#),
[5975](#), [5991](#), [6049](#), [6209](#), [6217](#), [6218](#),
[6219](#), [6310](#), [6313](#), [7963](#), [8479](#), [8483](#),
[8496](#), [8501](#), [8507](#), [8509](#), [8529](#), [8532](#),
[8605](#), [8611](#), [8617](#), [8626](#), [8636](#), [8711](#),
[8712](#), [9559](#), [9648](#), [10353](#), [10355](#),
[10414](#), [10417](#), [14510](#), [14543](#), [14585](#),
[14593](#), [14931](#), [14933](#), [14938](#), [15191](#)
- \quark_if_nil:N [2539](#)
- \quark_if_nil:n [2559](#)
- \quark_if_nil:nF [2580](#)
- \quark_if_nil:nT [2384](#), [2388](#), [2579](#)
- \quark_if_nil:NTF [45](#), [2539](#), [3984](#), [4006](#), [8524](#)
- \quark_if_nil:nTF
[45](#), [2394](#), [2403](#), [2412](#), [2421](#), [2559](#), [2578](#)
- \quark_if_nil:oTF [2559](#), [8505](#)
- \quark_if_nil:VTF [2559](#)
- \quark_if_nil_p:N [45](#), [2539](#)
- \quark_if_nil_p:n [45](#), [2559](#), [2577](#)

\quark_if_nil_p:o	2559	\reverse_if:N	24, 766, 771,
\quark_if_nil_p:V	2559		3595, 3597, 3599, 3601, 4293, 4298,
\quark_if_no_value:cF	8946		4302, 4304, 5187, 13082, 13512, 13539
\quark_if_no_value:cTF	2539	\right	476
\quark_if_no_value:N	2547	\righthypphenmin	532
\quark_if_no_value:n	2568	\rightskip	534
\quark_if_no_value:NF	2557	\romannumeral	609
\quark_if_no_value:NT	2556	true	181
\quark_if_no_value:NTF		\rule	7406, 7461
	45, 2227, 2539, 2558, 7424, 7428,		
	7507, 7511, 9221, 9228, 9316, 9328		
\quark_if_no_value:nTF	45, 2559	S	
\quark_if_no_value_p:c	2539	\S	10343
\quark_if_no_value_p:N	45, 2539, 2555	\s__fp	9728, 9728, 9732, 9741, 9742, 9743,
\quark_if_no_value_p:n	45, 2559		9744, 9745, 9747, 9748, 9751, 9757,
\quark_if_recursion_tail_break:N	2599, 2599		9761, 9781, 9784, 9785, 9795, 9805,
			9817, 9837, 9913, 9915, 9917, 9918,
\quark_if_recursion_tail_break:n	2599, 2601		9919, 9921, 9922, 9923, 9925, 10101,
			10106, 10264, 10273, 10275, 10765,
\quark_if_recursion_tail_stop:N	46, 2494, 2494, 6160		11017, 11428, 11453, 11454, 11562,
			11570, 11573, 11584, 11585, 11593,
\quark_if_recursion_tail_stop:n	8,		11594, 11596, 11597, 11598, 11600,
	9, 46, 2508, 2508, 2524, 5889, 6190		11601, 11602, 11613, 11616, 11628,
\quark_if_recursion_tail_stop:o	2508, 8478		11653, 11706, 11709, 11712, 11731,
			11732, 11734, 11735, 11736, 11744,
\quark_if_recursion_tail_stop_do:Nn	46, 2494, 2500		11747, 11763, 11764, 11766, 11775,
			11851, 12002, 12036, 12037, 12040,
\quark_if_recursion_tail_stop_do:nn	46, 2508, 2515, 2525		12119, 12255, 12258, 12263, 12266,
			12483, 12492, 12497, 12501, 12523,
\quark_if_recursion_tail_stop_do:on	2508		12596, 12608, 12610, 12817, 12834,
			12836, 13025, 13044, 13046, 13047,
\quark_new:N	45, 2487, 2487, 2488, 2489,		13049, 13061, 13066, 13068, 13093,
	2490, 2491, 2492, 2493, 2581, 2582		13094, 13096, 13112, 13197, 13210,
			13212, 13259, 13272, 13274, 13287,
			13289, 13302, 13304, 13317, 13319,
			13332, 13334, 13347, 13359, 13378,
			13387, 13571, 13596, 13599, 13621,
			13645, 13648, 13689, 13712, 13762,
			13763, 13863, 13870, 13923, 13932
R		\s__fp_division	9736, 9739
\R	1898, 15177	\s__fp_exact	9736,
\radical	435		9740, 9741, 9742, 9743, 9744, 9745
\raise	575	\s__fp_invalid	9736, 9736
\read	382	\s__fp_mark	9734, 9734, 9972,
\readline	657		9973, 9975, 9979, 10903, 10908, 10934
\relax	4, 5, 6, 7,	\s__fp_overflow	9736, 9738, 9748
	10, 14, 63, 69, 73, 90, 112, 113, 114,	\s__fp_stop	9734, 9735, 10904, 10906,
	115, 116, 117, 118, 119, 120, 121,		11136, 11632, 11643, 11666, 11674
	122, 126, 127, 128, 129, 130, 131,	\s__fp_underflow	9736, 9737, 9747
	132, 133, 134, 135, 138, 236, 237,	\s__fp_unknown	10497
	238, 239, 240, 241, 242, 243, 244, 417		
\relpenalty	478		
\RequirePackage	58, 59		

- \s__prop . . . [127](#), [6280](#), [6280](#), [6310](#), [6313](#),
[6395](#), [6415](#), [6432](#), [6436](#), [6444](#), [6477](#),
[6480](#), [6497](#), [14866](#), [14869](#), [14881](#), [14884](#)
- \s__stop [48](#), [2597](#),
[2597](#), [2598](#), [12895](#), [12910](#), [13553](#), [13557](#)
- \savecatcodetable [734](#)
- \savinghyphcodes [696](#)
- \savingvdiscards [697](#)
- csc [180](#)
- \scan_align_safe_stop: [42](#), [2337](#), [2337](#), [4124](#)
- \scan_stop: [9](#), [263](#), [277](#), [790](#),
[790](#), [969](#), [1077](#), [1101](#), [1119](#), [1129](#),
[1147](#), [1460](#), [1461](#), [1667](#), [1892](#), [1895](#),
[1896](#), [1897](#), [1898](#), [1934](#), [1976](#), [2043](#),
[2044](#), [2340](#), [2355](#), [2594](#), [2766](#), [2843](#),
[3148](#), [3262](#), [3271](#), [3280](#), [3748](#), [4435](#),
[4446](#), [4451](#), [4477](#), [4482](#), [4485](#), [4526](#),
[4537](#), [4542](#), [4547](#), [4729](#), [4730](#), [4731](#),
[4732](#), [5188](#), [5616](#), [6670](#), [6688](#), [7402](#),
[7457](#), [9354](#), [9356](#), [9447](#), [9449](#), [14052](#),
[15024](#), [15025](#), [15209](#), [15211](#), [15213](#)
- \scantokens [655](#)
- \scriptfont [601](#)
- \scriptscriptfont [602](#)
- \scriptscriptstyle [447](#)
- \scriptspace [487](#)
- \scriptstyle [446](#)
- \scrollmode [412](#)
- \seq_clear:c [5405](#), [5406](#)
- \seq_clear:N
. . . [104](#), [5405](#), [5405](#), [5485](#), [7959](#), [8022](#)
- \seq_clear_new:c [5409](#), [5410](#)
- \seq_clear_new:N [104](#), [5409](#), [5409](#)
- \seq_concat:ccc [5455](#), [5457](#)
- \seq_concat:NNN . . . [105](#), [5455](#), [5455](#), [9200](#)
- \seq_count:c [5725](#), [5822](#)
- \seq_count:N [109](#),
[5725](#), [5725](#), [5734](#), [5739](#), [5821](#), [14899](#)
- \seq_display:c [5816](#), [5818](#)
- \seq_display:N [5816](#), [5817](#)
- \seq_gclear:c [5405](#), [5408](#)
- \seq_gclear:N [104](#), [5405](#), [5407](#)
- \seq_gclear_new:c [5409](#), [5412](#)
- \seq_gclear_new:N [104](#), [5409](#), [5411](#)
- \seq_gconcat:ccc [5455](#), [5458](#)
- \seq_gconcat:NNN [105](#), [5455](#), [5456](#)
- \seq_get:cN [5790](#), [5791](#), [5797](#)
- \seq_get:cNTF [5796](#)
- \seq_get:NN [111](#), [5790](#), [5790](#), [5796](#)
- \seq_get:NNTF [111](#), [5796](#)
- \seq_get_left:cN . . . [5568](#), [5791](#), [5797](#), [5814](#)
- \seq_get_left:cNTF [5637](#)
- \seq_get_left:NN . . . [105](#), [5568](#), [5568](#),
[5578](#), [5637](#), [5638](#), [5790](#), [5796](#), [5813](#)
- \seq_get_left:NNF [5642](#)
- \seq_get_left:NNT [5641](#)
- \seq_get_left:NNTF [106](#), [5637](#), [5643](#)
- \seq_get_right:cN [5592](#)
- \seq_get_right:cNTF [5637](#)
- \seq_get_right:NN
. . . [105](#), [5592](#), [5592](#), [5608](#), [5639](#), [5640](#)
- \seq_get_right:NNF [5645](#)
- \seq_get_right:NNT [5644](#)
- \seq_get_right:NNTF [106](#), [5637](#), [5646](#)
- \seq_gpop:cN [5790](#), [5795](#), [5801](#)
- \seq_gpop:cNTF [5796](#)
- \seq_gpop:NN
. . . [111](#), [5790](#), [5794](#), [5800](#), [9255](#), [14051](#)
- \seq_gpop:NNTF [111](#), [5796](#), [9338](#), [9431](#)
- \seq_gpop_left:cN [5579](#), [5795](#), [5801](#)
- \seq_gpop_left:cNTF [5647](#)
- \seq_gpop_left:NN
. [106](#), [5579](#), [5581](#), [5591](#), [5649](#), [5794](#), [5800](#)
- \seq_gpop_left:NNF [5659](#)
- \seq_gpop_left:NNT [5658](#)
- \seq_gpop_left:NNTF [107](#), [5647](#), [5660](#)
- \seq_gpop_right:cN [5609](#)
- \seq_gpop_right:cNTF [5647](#)
- \seq_gpop_right:NN
. [106](#), [5609](#), [5611](#), [5636](#), [5653](#)
- \seq_gpop_right:NNF [5665](#)
- \seq_gpop_right:NNT [5664](#)
- \seq_gpop_right:NNTF . . . [107](#), [5647](#), [5666](#)
- \seq_gpush:cn [5770](#), [5785](#)
- \seq_gpush:co [5770](#), [5788](#)
- \seq_gpush:cV [5770](#), [5786](#)
- \seq_gpush:cv [5770](#), [5787](#)
- \seq_gpush:cx [5770](#), [5789](#)
- \seq_gpush:Nn [111](#), [5770](#), [5780](#)
- \seq_gpush:No [5770](#), [5783](#), [9252](#)
- \seq_gpush:NV [5770](#), [5781](#), [9365](#), [9458](#)
- \seq_gpush:Nv [5770](#), [5782](#)
- \seq_gpush:Nx [5770](#), [5784](#), [14038](#)
- \seq_gput_left:cn [5469](#), [5785](#)
- \seq_gput_left:co [5469](#), [5788](#)
- \seq_gput_left:cV [5469](#), [5786](#)
- \seq_gput_left:cv [5469](#), [5787](#)
- \seq_gput_left:cx [5469](#), [5789](#)
- \seq_gput_left:Nn
. . . [105](#), [5469](#), [5469](#), [5473](#), [5474](#), [5780](#)

\seq_gput_left:No	5469, 5783	\seq_if_exist_p:c	5459
\seq_gput_left:Nv	5469, 5781	\seq_if_exist_p:N	105, 5459
\seq_gput_left:Nv	5469, 5782	\seq_if_in:cnTF	5525
\seq_gput_left:Nx	5469, 5784	\seq_if_in:coTF	5525
\seq_gput_right:cn	5469	\seq_if_in:cVTF	5525
\seq_gput_right:co	5469	\seq_if_in:cvTF	5525
\seq_gput_right:cV	5469	\seq_if_in:cxTF	5525
\seq_gput_right:cv	5469	\seq_if_in:Nn	5525
\seq_gput_right:cx	5469	\seq_if_in:NnF	5488, 5546, 5547, 9263
\seq_gput_right:Nn	105, 5469, 5471, 5475, 5476, 9245, 9250	\seq_if_in:NnT	5544, 5545
\seq_gput_right:No	5469, 9293	\seq_if_in:NnTF	108, 5525, 5548, 5549
\seq_gput_right:Nv	5469, 9150	\seq_if_in:NoTF	5525
\seq_gput_right:Nv	5469	\seq_if_in:NvF	9364, 9457
\seq_gput_right:Nx	5469	\seq_if_in:NvTF	5525
\seq_gremove_all:cn	5495	\seq_if_in:NvTF	5525
\seq_gremove_all:Nn	108, 5495, 5497, 5520	\seq_if_in:NxTF	5525
\seq_gremove_duplicates:c	5479	\seq_item:cn	14892
\seq_gremove_duplicates:N	107, 5479, 5481, 5494	\seq_item:Nn	194, 8039, 8040, 8045, 14892, 14892, 14915
\seq_greverse:c	14968	\seq_length:c	5820, 5822
\seq_greverse:N	194, 14968, 14971, 14986	\seq_length:N	5820, 5821
\seq_gset_eq:cc	5413, 5420	\seq_map_break:	109, 5667, 5667, 5668, 5670, 5674, 5675, 5710, 5721, 9210
\seq_gset_eq:cN	5413, 5419	\seq_map_break:n	109, 5667, 5669, 7979, 7993
\seq_gset_eq:Nc	5413, 5418	\seq_map_function:cN	5671
\seq_gset_eq:NN	104, 5413, 5417, 5482, 9414	\seq_map_function:NN	4, 108, 5671, 5671, 5683, 5730, 5805, 8043, 14561
\seq_gset_filter:NNn	195, 14987, 14989	\seq_map_inline:cn	5706
\seq_gset_from_clist:cc	14942	\seq_map_inline:Nn	108, 5486, 5706, 5706, 5712, 7974, 9165, 9204, 9286
\seq_gset_from_clist:cN	14942	\seq_map_variable:ccn	5713
\seq_gset_from_clist:cn	14942	\seq_map_variable:cNn	5713
\seq_gset_from_clist:Nc	14942	\seq_map_variable:Ncn	5713
\seq_gset_from_clist:NN	194, 14942, 14952, 14965, 14966	\seq_map_variable:NNn	108, 5713, 5713, 5723, 5724
\seq_gset_from_clist:Nn	14942, 14957, 14967	\seq_mapthread_function:ccN	14916
\seq_gset_map:NNn	195, 14997, 14999	\seq_mapthread_function:cNn	14916
\seq_gset_split:Nnn	104, 5421, 5423, 5454, 9300	\seq_mapthread_function:NcN	14916
\seq_gset_split:NnV	5421	\seq_mapthread_function:NNN	194, 14916, 14916, 14940, 14941
\seq_if_empty:c	5523	\seq_new:c	5403, 5404
\seq_if_empty:cTF	5521	\seq_new:N	4, 104, 2725, 2742, 5403, 5403, 5478, 5808, 5809, 5810, 5811, 7931, 7932, 9144, 9145, 9155, 9157, 9160, 9298, 9412, 14004
\seq_if_empty:N	5521	\seq_pop:cN	5790, 5793, 5799
\seq_if_empty:NTF	108, 5521, 8261, 14049, 14555	\seq_pop:cNTF	5796
\seq_if_empty_p:c	5521	\seq_pop:NN	111, 5790, 5792, 5798
\seq_if_empty_p:N	108, 5521	\seq_pop:NNTF	111, 5796
\seq_if_exist:c	5460	\seq_pop_left:cN	5579, 5793, 5799
\seq_if_exist:cTF	5459		
\seq_if_exist:N	5459		
\seq_if_exist:NTF	105, 5459, 5737		

\seq_pop_left:cNTF	5647	\seq_remove_duplicates:N	
\seq_pop_left:NN	107 , 5479 , 5479 , 5493 , 9284
.....	106 , 5579 , 5579 , 5590 , 5647 , 5792 , 5798	\seq_reverse:c	14968
\seq_pop_left:NNF	5656	\seq_reverse:N ..	194 , 14968 , 14969 , 14985
\seq_pop_left:NNT	5655	\seq_set_eq:cc	5413 , 5416
\seq_pop_left:NNTF	107 , 5647 , 5657	\seq_set_eq:cN	5413 , 5415
\seq_pop_right:cN	5609	\seq_set_eq:Nc	5413 , 5414
\seq_pop_right:cNTF	5647	\seq_set_eq:NN	
\seq_pop_right:NN	104 , 5413 , 5413 , 5480 , 9198 , 9215 , 9273
.....	106 , 5609 , 5609 , 5635 , 5651	\seq_set_filter:NNn ...	195 , 14987 , 14987
\seq_pop_right:NNF	5662	\seq_set_from_clist:cc	14942
\seq_pop_right:NNT	5661	\seq_set_from_clist:cN	14942
\seq_pop_right:NNTF	107 , 5647 , 5663	\seq_set_from_clist:cn	14942
\seq_push:cn	5770 , 5775	\seq_set_from_clist:Nc	14942
\seq_push:co	5770 , 5778	\seq_set_from_clist:NN	
\seq_push:cV	5770 , 5770 , 5776	194 , 14942 , 14942 , 14962 , 14963
\seq_push:cv	5777	\seq_set_from_clist:Nn	14942 , 14947 , 14964
\seq_push:cx	5770 , 5779	\seq_set_map:NNn	195 , 14997 , 14997
\seq_push:Nn	111 , 5770 , 5770	\seq_set_split:Nnn	
\seq_push:No	5770 , 5773	...	104 , 2738 , 2743 , 5421 , 5421 , 5453
\seq_push:Nv	5770 , 5771	\seq_set_split:NnV	5421 , 9199
\seq_push:Nv	5770 , 5772	\seq_show:c	5802 , 5818
\seq_push:Nx	5770 , 5774	\seq_show:N ...	112 , 5802 , 5802 , 5807 , 5817
\seq_put_left:cn	5461 , 5775	\seq_tmp:w	5615 , 5628
\seq_put_left:co	5461 , 5778	\seq_top:cN	5812 , 5814
\seq_put_left:cV	5461 , 5776	\seq_top:NN	5812 , 5813
\seq_put_left:cv	5461 , 5777	\seq_use:c	5824
\seq_put_left:cx	5461 , 5779	\seq_use:cn	5735
\seq_put_left:Nn		\seq_use:cnnn	5735
.....	105 , 5461 , 5461 , 5465 , 5466 , 5770 , 7969	\seq_use:N	5824 , 5825 , 5826
\seq_put_left:No	5461 , 5773	\seq_use:Nn ...	110 , 5735 , 5767 , 5769 , 5825
\seq_put_left:Nv	5461 , 5771	\seq_use:Nnnn .	110 , 5735 , 5735 , 5755 , 5768
\seq_put_left:Nv	5461 , 5772	\set@color	7628 , 7629
\seq_put_left:Nx	5461 , 5774	\setbox	583
\seq_put_right:cn	5461	\setlanguage	341
\seq_put_right:co	5461	\sfcode	638
\seq_put_right:cV	5461	\sffamily	7395
\seq_put_right:cv	5461	\shipout	548
\seq_put_right:cx	5461	\show	391
\seq_put_right:Nn	105 , 5461 , 5463 , 5467 , 5468 , 5489 , 8030 , 9264	\showbox	393
\seq_put_right:No	5461 , 9279	\showboxbreadth	407
\seq_put_right:Nv	5461	\showboxdepth	408
\seq_put_right:Nv	5461	\showgroups	668
\seq_put_right:Nx	5461	\showifs	669
\seq_remove_all:cn	5495	\showlists	394
\seq_remove_all:Nn		\showthe	392
.....	108 , 5495 , 5495 , 5519 , 9269	\showtokens	656
\seq_remove_duplicates:c	5479	\skewchar	605
		\skip	629
		\skip_add:cn	4445

<code>\skip_add:Nn</code> . . .	83 , 4445 , 4445 , 4447 , 4448	<code>\skip_set_eq:NN</code>	84 , 4439 , 4439 , 4440 , 4441
<code>\skip_const:cn</code>	4416	<code>\skip_show:c</code>	4488
<code>\skip_const:Nn</code>		<code>\skip_show:N</code>	85 , 4488 , 4488 , 4489
.	83 , 4416 , 4416 , 4421 , 4492 , 4493	<code>\skip_show:n</code>	85 , 4490 , 4490
<code>\skip_eval:n</code>	84 , 4458 , 4476 , 4476	<code>\skip_split_finite_else_action:nnNN</code>	
<code>\skip_gadd:cn</code>	4445	196 , 15020 , 15020
<code>\skip_gadd:Nn</code>	83 , 4445 , 4447 , 4449	<code>\skip_sub:cn</code>	4445
<code>.skip_gset:c</code>	154	<code>\skip_sub:Nn</code> . . .	84 , 4445 , 4450 , 4452 , 4453
<code>\skip_gset:cn</code>	4434	<code>\skip_use:c</code>	4478
<code>.skip_gset:N</code>	154	<code>\skip_use:N</code> 85 , 4470 , 4477 , 4478 , 4478 , 4479	
<code>\skip_gset:Nn</code> . .	83 , 4419 , 4434 , 4436 , 4438	<code>\skip_vertical:c</code>	4480
<code>\skip_gset_eq:cc</code>	4439	<code>\skip_vertical:N</code> 86 , 4480 , 4483 , 4485 , 4487	
<code>\skip_gset_eq:cN</code>	4439	<code>\skip_vertical:n</code>	4480 , 4484
<code>\skip_gset_eq:Nc</code>	4439	<code>\skip_zero:c</code>	4422
<code>\skip_gset_eq:NN</code>	84 , 4439 , 4442 , 4443 , 4444	<code>\skip_zero:N</code> 83 , 4422 , 4422 , 4423 , 4424 , 4427	
<code>\skip_gsub:cn</code>	4445	<code>\skip_zero_new:c</code>	4426
<code>\skip_gsub:Nn</code>	84 , 4445 , 4452 , 4454	<code>\skip_zero_new:N</code> . . .	83 , 4426 , 4426 , 4430
<code>\skip_gzero:c</code>	4422	<code>\skipdef</code>	328
<code>\skip_gzero:N</code> . .	83 , 4422 , 4423 , 4425 , 4429	<code>\space</code>	50 , 212
<code>\skip_gzero_new:c</code>	4426	<code>\spacefactor</code>	547
<code>\skip_gzero_new:N</code> . .	83 , 4426 , 4428 , 4431	<code>\spaceskip</code>	542
<code>\skip_horizontal:c</code>	4480	<code>\span</code>	355
<code>\skip_horizontal:N</code>		<code>\special</code>	617
.	86 , 4480 , 4480 , 4482 , 4486	<code>\splitbotmark</code>	426
<code>\skip_horizontal:n</code>	4480 , 4481	<code>\splitbotmarks</code>	652
<code>\skip_if_eq:nn</code>	4455	<code>\splitdiscards</code>	699
<code>\skip_if_eq:nnTF</code>	84 , 4455	<code>\splitfirstmark</code>	425
<code>\skip_if_eq_p:nn</code>	84 , 4455	<code>\splitfirstmarks</code>	651
<code>\skip_if_exist:c</code>	4433	<code>\splitmaxdepth</code>	595
<code>\skip_if_exist:cTF</code>	4432	<code>\splittopskip</code>	596
<code>\skip_if_exist:N</code>	4432	<code>\str_case:nnn</code>	
<code>\skip_if_exist:NTF</code> . .	83 , 4427 , 4429 , 4432	. . .	22 , 1532 , 1532 , 2081 , 2469 , 15118
<code>\skip_if_exist_p:c</code>	4432	<code>\str_case:onnn</code>	2073 , 2470
<code>\skip_if_exist_p:N</code>	83 , 4432	<code>\str_case_x:nnn</code>	23 , 1532 , 1543 , 2471
<code>\skip_if_finite:n</code>	4467	<code>\str_head:n</code> . . .	100 , 5176 , 5176 , 5200 , 5247
<code>\skip_if_finite:nTF</code>	84 , 4465 , 4562 , 15022	<code>\str_if_eq:nn</code>	1513
<code>\skip_if_finite_p:n</code>	84 , 4465	<code>\str_if_eq:nnF</code>	2077 , 2078 , 8246
<code>\skip_if_infinite_glue:n</code>	4561	<code>\str_if_eq:nnT</code> . . .	2075 , 2076 , 5503 , 7990
<code>\skip_if_infinite_glue:nTF</code>	4560	<code>\str_if_eq:nnTF</code>	22 , 1513 , 1539 ,
<code>\skip_if_infinite_glue_p:n</code>	4560	2079 , 2080 , 4054 , 4057 , 7815 , 8486	
<code>\skip_new:c</code>	4408	<code>\str_if_eq:noTF</code>	2073
<code>\skip_new:N</code>	83 , 4408 , 4409 , 4415 , 4418 ,	<code>\str_if_eq:nVTF</code>	2073
4427 , 4429 , 4494 , 4495 , 4496 , 4497		<code>\str_if_eq:onTF</code>	2073
<code>.skip_set:c</code>	154	<code>\str_if_eq:VnTF</code>	2073
<code>\skip_set:cn</code>	4434	<code>\str_if_eq:VVTf</code>	2073
<code>.skip_set:N</code>	154	<code>\str_if_eq:xxF</code>	1616
<code>\skip_set:Nn</code> . . .	83 , 4434 , 4434 , 4436 , 4437	<code>\str_if_eq:xxT</code>	1615
<code>\skip_set_eq:cc</code>	4439	<code>\str_if_eq:xxTF</code>	1613 , 1617
<code>\skip_set_eq:cN</code>	4439	<code>\str_if_eq_p:nn</code>	22 , 1513 , 2073 , 2074
<code>\skip_set_eq:Nc</code>	4439	<code>\str_if_eq_p:no</code>	2073

<code>\str_if_eq_p:nV</code>	2073	<code>\tex_chardef:D</code>	325, 819, 820, 821, 822, 823, 825, 1065, 1066, 2115, 2117, 3017, 3493, 3494, 9354, 9447, 14012
<code>\str_if_eq_p:on</code>	2073	<code>\tex_cleaders:D</code>	508
<code>\str_if_eq_p:Vn</code>	2073	<code>\tex_closein:D</code>	384, 9362
<code>\str_if_eq_p:VV</code>	2073	<code>\tex_closeout:D</code>	379, 9455
<code>\str_if_eq_p:xx</code>	1613, 1614	<code>\tex_clubpenalty:D</code>	519
<code>\str_if_eq_x:nn</code>	1519	<code>\tex_copy:D</code>	576, 6581, 6610
<code>\str_if_eq_x:nnF</code>	1616, 8033, 8594	<code>\tex_count:D</code>	627, 2912
<code>\str_if_eq_x:nnT</code>	1615	<code>\tex_countdef:D</code>	326, 816, 2915
<code>\str_if_eq_x:nnTF</code>	22, 1513, 1550, 1617, 6438, 9612, 14886	<code>\tex_cr:D</code>	351
<code>\str_if_eq_x_p:nn</code>	22, 1513, 1614	<code>\tex_crcr:D</code>	352
<code>\str_tail:n</code>	100, 5176, 5184	<code>\tex_csname:D</code>	414, 782
<code>\strcmp</code>	70	<code>\tex_day:D</code>	622
<code>\string</code>	77, 149, 230, 610	<code>\tex_deadcycles:D</code>	556
T		<code>\tex_def:D</code> ...	321, 796, 798, 800, 801, 829
<code>\T</code>	2822, 3021, 10400, 15175	<code>\tex_defaultthyphenchar:D</code>	606
<code>pt</code>	181	<code>\tex_defaultskewchar:D</code>	607
<code>\tabskip</code>	356	<code>\tex_delcode:D</code>	637
<code>\tex_above:D</code>	438	<code>\tex_delimiter:D</code>	431
<code>\tex_abovedisplayshortskip:D</code>	451	<code>\tex_delimiterfactor:D</code>	480
<code>\tex_abovedisplayskip:D</code>	452	<code>\tex_delimitershortfall:D</code>	479
<code>\tex_abovewithdelims:D</code>	439	<code>\tex_dimen:D</code>	628, 2890
<code>\tex_accent:D</code>	489	<code>\tex_dimendef:D</code>	327, 2893
<code>\tex_adjdemerits:D</code>	526	<code>\tex_discretionary:D</code>	491
<code>\tex_advance:D</code>	333, 3515, 3517, 3527, 3529, 4222, 4227, 4446, 4451, 4537, 4542	<code>\tex_displayindent:D</code>	456
<code>\tex_afterassignment:D</code>	343, 3092, 5599, 5625	<code>\tex_displaylimits:D</code>	466
<code>\tex_aftergroup:D</code>	344, 795	<code>\tex_displaystyle:D</code>	444
<code>\tex_atop:D</code>	440	<code>\tex_displaywidowpenalty:D</code>	455
<code>\tex_atopwithdelims:D</code>	441	<code>\tex_displaywidth:D</code>	457
<code>\tex_badness:D</code>	588	<code>\tex_divide:D</code>	334
<code>\tex_baselineskip:D</code>	516	<code>\tex_doublehyphendemerits:D</code>	524
<code>\tex_batchmode:D</code>	409	<code>\tex_dp:D</code>	635, 6595
<code>\tex_begingroup:D</code>	347, 791	<code>\tex_dump:D</code>	618
<code>\tex_belowdisplayshortskip:D</code>	453	<code>\tex_edef:D</code>	322, 830
<code>\tex_belowdisplayskip:D</code>	454	<code>\tex_else:D</code>	375, 769, 826
<code>\tex_binoppenalty:D</code>	477	<code>\tex_emergencystretch:D</code>	539
<code>\tex_botmark:D</code>	424	<code>\tex_end:D</code>	413, 742, 1185, 7865
<code>\tex_box:D</code>	632, 6587, 6609	<code>\tex_endcsname:D</code>	415, 783
<code>\tex_boxmaxdepth:D</code>	594	<code>\tex_endgroup:D</code>	348, 740, 792
<code>\tex_brokenpenalty:D</code>	551	<code>\tex_endinput:D</code>	387, 7876
<code>\tex_catcode:D</code> .	636, 1078, 1461, 1895, 1896, 1897, 1898, 2043, 2341, 2356, 2612, 2614, 2616, 3283, 4731, 4732	<code>\tex_endlinechar:D</code>	262, 263, 277, 429, 4748, 9403, 9405
<code>\tex_char:D</code>	490	<code>\tex_eqno:D</code>	449
		<code>\tex_errhelp:D</code>	395, 7753
		<code>\tex_errmessage:D</code>	389, 1177, 7780
		<code>\tex_errorcontextlines:D</code>	396, 7805
		<code>\tex_errorstopmode:D</code>	410
		<code>\tex_escapechar:D</code> .	428, 9499, 9525, 9531
		<code>\tex_everycr:D</code>	357

- `\tex_everydisplay:D` 458, 743
- `\tex_everyhbox:D` 597
- `\tex_everyjob:D` 626, 4645, 4647,
4652, 4654, 9135, 9137, 9147, 9149
- `\tex_everymath:D` 482, 744
- `\tex_everypar:D` 545
- `\tex_everyvbox:D` 598
- `\tex_exhyphenpenalty:D` 521
- `\tex_expandafter:D` 345, 784
- `\tex_fam:D` 337
- `\tex_fi:D` 376, 770, 828
- `\tex_finalhyphendemerits:D` 525
- `\tex_firstmark:D` 423
- `\tex_floatingpenalty:D` 570
- `\tex_font:D` 336
- `\tex_fontdimen:D` 603
- `\tex_fontname:D` 427
- `\tex_futurelet:D` 332, 3086, 3088
- `\tex_gdef:D` 323, 843
- `\tex_global:D` 307, 312, 314, 338, 1272,
1279, 2117, 3088, 3477, 3497, 3509,
3519, 3521, 3531, 3533, 3540, 4199,
4212, 4218, 4223, 4228, 4423, 4436,
4442, 4447, 4452, 4514, 4527, 4533,
4538, 4543, 5324, 6583, 6589, 6645,
6690, 6696, 6702, 6732, 6738, 6744,
6750, 9354, 9447, 9694, 9696, 14012
- `\tex_globaldefs:D` 342
- `\tex_halign:D` 349
- `\tex_hangafter:D` 527
- `\tex_hangingindent:D` 528
- `\tex_hbadness:D` 589
- `\tex_hbox:D`
584, 6688, 6689, 6694, 6700, 6714, 6715
- `\tex_hfil:D` 492
- `\tex_hfill:D` 494
- `\tex_hfilneg:D` 493
- `\tex_hfuzz:D` 591
- `\tex_hoffset:D` 566
- `\tex_holdinginserts:D` 569
- `\tex_hruler:D` 505
- `\tex_hsize:D`
530, 6879, 6881, 6882, 6925, 6927, 6928
- `\tex_hskip:D` 495, 4480
- `\tex_hss:D` . 496, 6717, 6719, 14385, 14394
- `\tex_ht:D` 634, 6594
- `\tex_hyphen:D` 319, 745
- `\tex_hyphenation:D` 620
- `\tex_hyphenchar:D` 604
- `\tex_hyphenpenalty:D` 522
- `\tex_if:D` 358, 772, 773
- `\tex_ifcase:D` 359, 3387
- `\tex_ifcat:D` 360, 774
- `\tex_ifdim:D` 363, 4181
- `\tex_ifeof:D` 364, 9380
- `\tex_iffalse:D` 369, 767
- `\tex_ifhbox:D` 365, 6621
- `\tex_ifhmode:D` 371, 777
- `\tex_ifinner:D` 374, 779
- `\tex_ifmmode:D` 372, 776
- `\tex_ifnum:D` 361, 793
- `\tex_ifodd:D`
... 362, 1202, 2089, 2090, 3386, 7658
- `\tex_iftrue:D` 370, 766
- `\tex_ifvbox:D` 366, 6622
- `\tex_ifvmode:D` 373, 778
- `\tex_ifvoid:D` 367, 6623
- `\tex_ifx:D` 368, 775
- `\tex_ignorespaces:D` 416
- `\tex_immediate:D`
... 378, 1172, 1174, 9449, 9455, 9473
- `\tex_indent:D` 512
- `\tex_input:D` 386, 746, 9254
- `\tex_inputlineno:D` . 388, 1192, 2038, 7727
- `\tex_insert:D` 568
- `\tex_insertpenalties:D` 571
- `\tex_interlinepenalty:D` 550
- `\tex_italiccorrection:D` 318, 747
- `\tex_jobname:D` 625, 4655, 4659, 9138
- `\tex_kern:D` 503, 7191, 7196, 7262, 7263,
7550, 7551, 14198, 14383, 14392,
14405, 14407, 14448, 14450, 14619
- `\tex_language:D` 420
- `\tex_lastbox:D` 577, 6643, 6772
- `\tex_lastkern:D` 510
- `\tex_lastpenalty:D` 616
- `\tex_lastskip:D` 511
- `\tex_lccode:D`
639, 1077, 1460, 2044, 2340, 2355,
2688, 2690, 2692, 3285, 4729, 4730
- `\tex_leaders:D` 507
- `\tex_left:D` 475
- `\tex_lefthyphenmin:D` 531
- `\tex_leftskip:D` 533
- `\tex_leqno:D` 450
- `\tex_let:D` 308, 312, 314, 320,
742, 743, 744, 745, 746, 747, 748,
749, 750, 751, 752, 753, 754, 755,
756, 757, 766, 767, 768, 769, 770,
771, 772, 773, 774, 775, 776, 777,

778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 811, 813, 829, 830, 843, 844, 1268, 1611, 2089, 2090	\tex_nonstopmode:D 411
\tex_limits:D 467	\tex_nulldelimiterspace:D 481
\tex_linepenalty:D 523	\tex_nullfont:D 599, 3044
\tex_lineskip:D 517	\tex_number:D 608, 3383
\tex_lineskiplimit:D 518	\tex_omit:D 354
\tex_long:D 339, 796, 798, 801, 832, 834, 840, 842, 846, 848, 854, 856	\tex_openin:D 380, 9356
\tex_looseness:D 535	\tex_openout:D 381, 9449
\tex_lower:D 572, 6620	\tex_or:D 377, 768
\tex_lowercase:D 611, 1079, 1462, 1899, 2045, 4733, 4773	\tex_outer:D 340
\tex_mag:D 419	\tex_output:D 555
\tex_mark:D 421	\tex_outputpenalty:D 565
\tex_mathaccent:D 432	\tex_over:D 442
\tex_mathbin:D 462	\tex_overfullrule:D 593
\tex_mathchar:D 433	\tex_overline:D 473
\tex_mathchardef:D . . . 330, 827, 3489, 3490	\tex_overwithdelims:D 443
\tex_mathchoice:D 430	\tex_pagedepth:D 557
\tex_mathclose:D 463	\tex_pagefilllstretch:D 561
\tex_mathcode:D 641, 2682, 2684, 2686, 3284	\tex_pagefillstretch:D 560
\tex_mathinner:D 464	\tex_pagefilstretch:D 559
\tex_mathop:D 465	\tex_pagegoal:D 563
\tex_mathopen:D 469	\tex_pageshrink:D 562
\tex_mathord:D 470	\tex_pagestretch:D 558
\tex_mathpunct:D 471	\tex_pagetotal:D 564
\tex_mathrel:D 472	\tex_par:D 513, 7611
\tex_mathsurround:D 483	\tex_parfillskip:D 544
\tex_maxdeadcycles:D 553	\tex_parindent:D 537
\tex_maxdepth:D 554	\tex_parshape:D 529
\tex_meaning:D 613, 787, 789	\tex_parskip:D 536
\tex_medmuskip:D 484	\tex_patterns:D 619
\tex_message:D 390	\tex_pausing:D 406
\tex_mkern:D 437	\tex_penalty:D 614
\tex_month:D 623	\tex_postdisplaypenalty:D 461
\tex_moveleft:D 573, 6614	\tex_predisdisplaypenalty:D 460
\tex_moveright:D 574, 6616	\tex_predisplaysize:D 459
\tex_mskip:D 434	\tex_pretolerance:D 540
\tex_multiply:D 335	\tex_prevdepth:D 587
\tex_muskip:D 631, 2932	\tex_prevgraf:D 546
\tex_muskipdef:D 329, 2935	\tex_radical:D 435
\tex_newlinechar:D 385, 4749	\tex_raise:D 575, 6618
\tex_noalign:D 353	\tex_read:D 382, 9398
\tex_noboundary:D 488	\tex_relax:D 417, 790, 3385, 4183, 10379, 10380, 10469, 10497, 10500, 10716, 10763, 10764, 10932, 10961, 11270
\tex_noexpand:D 346, 785	\tex_relpenny:D 478
\tex_noindent:D 514	\tex_right:D 476
\tex_nolimits:D 468	\tex_righthypenmin:D 532
\tex_nonscript:D 448	\tex_rightskip:D 534
	\tex_romannumeral:D 609, 794, 1534, 1545, 1643, 1655, 1661, 1701, 1705, 1710,

1716, 1722, 1728, 1740, 1745, 1747, 1754, 1809, 1816, 1821, 1829, 1831, 1834, 1841, 1847, 1856, 1872, 1876, 1881, 3180, 3618, 4278, 4309, 4933, 5124, 5287, 8281, 9812, 9874, 9977, 10168, 10315, 10333, 10362, 10408, 10425, 10434, 10441, 10449, 10452, 10458, 10467, 10477, 10492, 10513, 10526, 10534, 10538, 10561, 10589, 10602, 10615, 10639, 10650, 10657, 10660, 10679, 10689, 10692, 10708, 10728, 10739, 10745, 10755, 10801, 10804, 10816, 10845, 10858, 10886, 10899, 10901, 10916, 10919, 10969, 10977, 10986, 10991, 10996, 11003, 11036, 11047, 11059, 11081, 11090, 11091, 11095, 11107, 11124, 11139, 11151, 11163, 11206, 11218, 11244, 11291, 11302, 11309, 11319, 11328, 11344, 11362, 11383, 11386, 11426, 11440, 11450, 11631, 11634, 11642, 11665, 11673, 12647, 12855, 12859, 12878, 12915, 13036, 13204, 13397, 13564, 13573, 13619, 13623, 13668, 13687, 13691, 13724, 13728, 13793, 13921, 15044, 15094, 15102, 15125	<code>\tex_spacefactor:D</code> 547 <code>\tex_spaceskip:D</code> 542 <code>\tex_span:D</code> 355 <code>\tex_special:D</code> 617 <code>\tex_splitbotmark:D</code> 426 <code>\tex_splitfirstmark:D</code> 425 <code>\tex_splitmaxdepth:D</code> 595 <code>\tex_splittopskip:D</code> 596 <code>\tex_string:D</code> 610, 788 <code>\tex_tabskip:D</code> 356 <code>\tex_textfont:D</code> 600 <code>\tex_textstyle:D</code> 445 <code>\tex_the:D</code> 263, 418, 1192, 1673, 1677, 2038, 2614, 2684, 2690, 2696, 2702, 3290, 3293, 3296, 3299, 3302, 3543, 4396, 4478, 4491, 4548, 4553, 4647, 4654, 6670, 9137, 9149, 10410, 10778, 14038 <code>\tex_thickmuskip:D</code> 486 <code>\tex_thinmuskip:D</code> 485 <code>\tex_time:D</code> 621 <code>\tex_toks:D</code> 630, 2968 <code>\tex_toksdef:D</code> 331, 2971 <code>\tex_tolerance:D</code> 541 <code>\tex_topmark:D</code> 422 <code>\tex_topskip:D</code> 552 <code>\tex_tracingcommands:D</code> 397 <code>\tex_tracinglostchars:D</code> 398 <code>\tex_tracingmacros:D</code> 399 <code>\tex_tracingonline:D</code> 400, 6679 <code>\tex_tracingoutput:D</code> 401 <code>\tex_tracingpages:D</code> 402 <code>\tex_tracingparagraphs:D</code> 403 <code>\tex_tracingrestores:D</code> 404 <code>\tex_tracingstats:D</code> 405 <code>\tex_uccode:D</code> 640, 2694, 2696, 2698, 3286, 10446, 10472, 10944 <code>\tex_uchyph:D</code> 538 <code>\tex_undefined:D</code> 307, 314, 1285, 1293, 9993, 10004, 10005, 10006, 10007 <code>\tex_underline:D</code> 474, 748 <code>\tex_unhbox:D</code> 579, 6721 <code>\tex_unhcopy:D</code> 580, 6720 <code>\tex_unkern:D</code> 504 <code>\tex_unpenalty:D</code> 615 <code>\tex_unskip:D</code> 502 <code>\tex_unvbox:D</code> 581, 6766 <code>\tex_unvcopy:D</code> 582, 6765 <code>\tex_uppercase:D</code> 612, 4775 <code>\tex_vadjust:D</code> 515
--	---

<code>\tex_valign:D</code>	350	<code>\tl_concat:ccc</code>	4633, 5457
<code>\tex_vbadness:D</code>	590	<code>\tl_concat:NNN</code> .	91, 4633, 4633, 4637, 5455
<code>\tex_vbox:D</code>	585, 6724, 6727, 6729, 6731, 6742, 6748	<code>\tl_const:cn</code>	4601
<code>\tex_vcenter:D</code>	436	<code>\tl_const:cx</code>	4601, 9506
<code>\tex_vfil:D</code>	497	<code>\tl_const:Nn</code> 91, 2487, 2723, 4601, 4601, 4611,
<code>\tex_vfill:D</code>	499		4641, 4661, 4736, 7642, 7643, 7695,
<code>\tex_vfilneg:D</code>	498		7700, 7702, 7704, 7706, 7708, 7713,
<code>\tex_vfuzz:D</code>	592		7714, 7721, 8400, 8402, 8555, 8556,
<code>\tex_voffset:D</code>	567		8557, 8558, 8559, 9496, 9741, 9742,
<code>\tex_vrule:D</code>	506, 7402, 7457		9743, 9744, 9745, 12274, 12587,
<code>\tex_vsize:D</code>	549		12588, 12589, 12590, 12591, 12592,
<code>\tex_vskip:D</code>	500, 4483		12593, 12594, 12595, 15080, 15085
<code>\tex_vsplit:D</code>	578, 6770	<code>\tl_const:Nx</code>	4601, 4606, 4612,
<code>\tex_vss:D</code>	501		4655, 4659, 9500, 13809, 13932, 14578
<code>\tex_vtop:D</code>	586, 6725, 6736	<code>\tl_count:c</code>	4999, 5353, 5367
<code>\tex_wd:D</code>	633, 6596	<code>\tl_count:N</code>	98, 4999, 5004, 5011, 5352, 5366, 9546
<code>\tex_widowpenalty:D</code>	520	<code>\tl_count:n</code> 98, 927, 931, 1330, 1368, 4999,
<code>\tex_write:D</code>	383, 1172, 1174, 9467, 9470, 9473		4999, 5010, 5349, 5368, 12550, 15134
<code>\tex_xdef:D</code>	324, 844	<code>\tl_count:o</code>	4999, 5351, 5370
<code>\tex_xleaders:D</code>	509	<code>\tl_count:V</code>	4999, 5350, 5369
<code>\tex_xspaceskip:D</code>	543	<code>\tl_count_tokens:n</code> 196, 5363, 15064, 15064, 15079
<code>\tex_year:D</code>	624	<code>\tl_elt_count:c</code>	5348, 5353
<code>\textasteriskcentered</code>	4148, 4154	<code>\tl_elt_count:N</code>	5348, 5352
<code>\textbardbl</code>	4153	<code>\tl_elt_count:n</code>	5348, 5349
<code>\textdagger</code>	4149, 4155	<code>\tl_elt_count:o</code>	5348, 5351
<code>\textdaggerdbl</code>	4150, 4156	<code>\tl_elt_count:V</code>	5348, 5350
<code>\textdir</code>	739	<code>\tl_expandable_lowercase:n</code> 196, 15090, 15098
<code>\textfont</code>	600	<code>\tl_expandable_uppercase:n</code> 196, 15090, 15090
<code>\textparagraph</code>	4152	<code>\tl_gclear:c</code>	4613, 5408, 5844, 6288
<code>\textsection</code>	4151	<code>\tl_gclear:N</code>	91, 4613,
<code>\textstyle</code>	445		4615, 4618, 4622, 5407, 5843, 6287
<code>\TeXeTstate</code>	700	<code>\tl_gclear_new:c</code> .	4619, 5412, 5848, 6292
<code>\the</code>	112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 418	<code>\tl_gclear_new:N</code>	91, 4619, 4621, 4624, 5411, 5847, 6291
<code>\thickmuskip</code>	486	<code>\tl_gconcat:ccc</code>	4633, 5458
<code>\thinmuskip</code>	485	<code>\tl_gconcat:NNN</code> .	91, 4633, 4635, 4638, 5456
<code>\time</code>	621	<code>\tl_gput_left:cn</code>	4680
<code>\tiny</code>	7395	<code>\tl_gput_left:co</code>	4680
<code>\tl_case:cn</code>	2473, 4931	<code>\tl_gput_left:cV</code>	4680
<code>\tl_case:Nnn</code> . . .	95, 2472, 4931, 4931, 4942	<code>\tl_gput_left:cx</code>	4680
<code>\tl_clear:c</code>	4613, 5406, 5842, 6286	<code>\tl_gput_left:Nn</code> .	92, 4680, 4688, 4700, 5470
<code>\tl_clear:N</code>	91, 4613, 4613, 4617, 4620, 5405, 5841, 6285, 8351, 8462, 8466, 9548, 9550, 9554, 9621	<code>\tl_gput_left:No</code>	4680, 4692, 4702
<code>\tl_clear_new:c</code> 4619, 5410, 5846, 6290, 8704, 8706	<code>\tl_gput_left:NV</code>	4680, 4690, 4701
<code>\tl_clear_new:N</code>	91, 4619, 4619, 4623, 5409, 5845, 6289	<code>\tl_gput_left:Nx</code>	4680, 4694, 4703

\tl_gput_right:cn	4704	5612, 5654, 5860, 5904, 5947, 5985,
\tl_gput_right:co	4704	6037, 6391, 9138, 13807, 13873,
\tl_gput_right:cV	4704	13883, 13941, 13963, 14552, 14859,
\tl_gput_right:cx	4704	14954, 14959, 14972, 14990, 15000
\tl_gput_right:Nn	92, 2593, 4704, 4712, 4724, 5472	\tl_gset_eq:cc 4625, 4632, 5420, 5856, 6300
\tl_gput_right:No	4704, 4716, 4726	\tl_gset_eq:cN 4625, 4630, 5419, 5855, 6299
\tl_gput_right:Nv	4704, 4714, 4725	\tl_gset_eq:Nc 4625, 4631, 5418, 5854, 6298
\tl_gput_right:Nx	4704, 4718, 4727, 6391, 6411	\tl_gset_eq:NN 91, 4616, 4625, 4629, 5417, 5853,
\tl_gremove_all:cn	4832, 5346	6297, 6520, 9142, 9256, 13814, 13909
\tl_gremove_all:Nn	93, 4832, 4834, 4837, 5345	\tl_gset_rescan:cn 4738
\tl_gremove_all_in:cn	5338, 5346	\tl_gset_rescan:cno 4738
\tl_gremove_all_in:Nn	5338, 5345	\tl_gset_rescan:cnx 4738
\tl_gremove_in:cn	5338, 5342	\tl_gset_rescan:Nnn 93, 4738, 4740, 4770, 4771
\tl_gremove_in:Nn	5338, 5341	\tl_gset_rescan:Nno 4738
\tl_gremove_once:cn	4826, 5342	\tl_gset_rescan:Nnx 4738
\tl_gremove_once:Nn	92, 4826, 4828, 4831, 5341	.tl_gset_x:c 154
\tl_greplace_all:cn 4776, 5336		.tl_gset_x:N 154
\tl_greplace_all:Nnn 92, 4776, 4782, 4787, 4835, 5335		\tl_gtrim_spaces:c 5028
\tl_greplace_all_in:cn 5328, 5336		\tl_gtrim_spaces:N 99, 5028, 5032, 5035
\tl_greplace_all_in:Nnn 5328, 5335		\tl_head:f 5146
\tl_greplace_in:cn 5328, 5332		\tl_head:N 99, 5146, 5166
\tl_greplace_in:Nnn 5328, 5331		\tl_head:n 5146, 5146, 5164, 5166, 5356
\tl_greplace_once:cn 4776, 5332		\tl_head:V 5146
\tl_greplace_once:Nnn 92, 4776, 4778, 4785, 4829, 5331		\tl_head:v 100, 5146,
\tl_greverse:c 5140		5165, 5183, 5198, 5217, 5239, 5357
\tl_greverse:N 98, 5140, 5142, 5145		\tl_head_i:n 5355, 5356
.tl_gset:c 154		\tl_head_i:w 5355, 5357
\tl_gset:cf 4662		\tl_head_iii:f 5355
\tl_gset:cn 4662		\tl_head_iii:n 5355, 5358, 5359
\tl_gset:co 4662		\tl_head_iii:w 5355, 5358, 5360
\tl_gset:cV 4662		\tl_if_blank:n 4838
\tl_gset:cv 4662		\tl_if_blank:nF 4045, 4842, 4846, 6191
\tl_gset:cx 4662		\tl_if_blank:nT 4841, 4845
.tl_gset:N 154		\tl_if_blank:nTF 94, 4838, 4843, 4847, 5170, 14531
\tl_gset:Nc 5322, 5323		\tl_if_blank:oTF 4838, 8475
\tl_gset:Nf 4662		\tl_if_blank:VTF 4838
\tl_gset:Nn 92, 4662, 4668, 4677, 4679, 4741, 5317, 5582, 5650, 6325, 6354, 6377, 9253		\tl_if_blank_p:n 94, 4838, 4840, 4844
\tl_gset:No 4662, 4670		\tl_if_blank_p:o 4838
\tl_gset:Nv 4662		\tl_if_blank_p:V 4838
\tl_gset:Nv 4662		\tl_if_empty:c 5523, 6066, 6426
\tl_gset:Nx 4636, 4662, 4672, 4678, 4779, 4783, 5033, 5143, 5424, 5498,		\tl_if_empty:cTF 4848
		\tl_if_empty:N 4848, 5521, 6065, 6424
		\tl_if_empty:n 4860
		\tl_if_empty:NF 4858
		\tl_if_empty:nF 956, 1034,
		3196, 4871, 6117, 8134, 8138, 13776

\tl_if_empty:NT	4857	\tl_if_head_eq_charcode:nNTF	101, 3950, 3963, 5191, 5207
\tl_if_empty:nT	4870	\tl_if_head_eq_charcode_p:fN	5191
\tl_if_empty:NTF	94, 4848, 4859, 8521	\tl_if_head_eq_charcode_p:nN	101, 5191, 5206
\tl_if_empty:nTF	94, 4087, 4790, 4860, 4869, 5427, 5890, 7736, 8000, 8015, 8350, 8637, 15206	\tl_if_head_eq_meaning:nN	5229
\tl_if_empty:o	4881	\tl_if_head_eq_meaning:nNTF	101, 5191, 8502
\tl_if_empty:oTF	3036, 4872, 4919, 5292, 6079, 14568, 14589	\tl_if_head_eq_meaning_p:nN	101, 5191
\tl_if_empty:VTF	4860	\tl_if_head_group:n	5377
\tl_if_empty:x	5373	\tl_if_head_group:nTF	5376
\tl_if_empty:xTF	5372	\tl_if_head_group_p:n	5376
\tl_if_empty_p:c	4848	\tl_if_head_is_group:n	5270, 5377
\tl_if_empty_p:N	94, 4848, 4856	\tl_if_head_is_group:nTF	101, 5086, 5220, 5255, 5270
\tl_if_empty_p:n	94, 4860, 4868	\tl_if_head_is_group_p:n	101, 5270
\tl_if_empty_p:o	4872	\tl_if_head_is_N_type:n	5264, 5379
\tl_if_empty_p:V	4860	\tl_if_head_is_N_type:nTF	101, 5083, 5195, 5214, 5231, 5264, 15036
\tl_if_empty_p:x	5372	\tl_if_head_is_N_type_p:n	101, 5264
\tl_if_eq:cc	6261, 6540	\tl_if_head_is_space:n	5285, 5381
\tl_if_eq:ccTF	4883, 8958	\tl_if_head_is_space:nTF	101, 5285
\tl_if_eq:cN	6260, 6538	\tl_if_head_is_space_p:n	101, 5285
\tl_if_eq:cNTF	4883	\tl_if_head_N_type:n	5379
\tl_if_eq:Nc	6259, 6539	\tl_if_head_N_type:nTF	5376
\tl_if_eq:NcTF	4883	\tl_if_head_N_type_p:n	5376
\tl_if_eq:NN	4883, 6258, 6537	\tl_if_head_space:n	5381
\tl_if_eq:nn	4895	\tl_if_head_space:nTF	5376
\tl_if_eq:NNF	4894	\tl_if_head_space_p:n	5376
\tl_if_eq:NNT	4893, 5510, 7468, 7471	\tl_if_in:cnTF	4910
\tl_if_eq:NNTF	94, 4883, 4892, 4938, 7981, 8035, 9567	\tl_if_in:nn	4916
\tl_if_eq:nnTF	94, 4895	\tl_if_in:NnF	4911, 4914
\tl_if_eq_p:cc	4883	\tl_if_in:nnF	4911, 4923
\tl_if_eq_p:cN	4883	\tl_if_in:NnT	4910, 4913, 9170
\tl_if_eq_p:Nc	4883	\tl_if_in:nnT	4910, 4922
\tl_if_eq_p:NN	94, 4883, 4891	\tl_if_in:NnTF	95, 2587, 4910, 4912, 4915
\tl_if_exist:c	4640	\tl_if_in:nnTF	95, 4912, 4916, 4924, 7277, 8604, 8614, 9237
\tl_if_exist:cTF	4639	\tl_if_in:noTF	4916, 15204
\tl_if_exist:N	4639	\tl_if_in:onTF	4916
\tl_if_exist:NTF	91, 4620, 4622, 4639, 4995, 5299	\tl_if_in:VnTF	4916
\tl_if_exist_p:c	4639	\tl_if_single:n	4929
\tl_if_exist_p:N	91, 4639	\tl_if_single:Nf	4927
\tl_if_head_eq_catcode:nN	5210	\tl_if_single:nF	4927
\tl_if_head_eq_catcode:nNTF	100, 5191	\tl_if_single:NT	4926
\tl_if_head_eq_catcode_p:nN	100, 5191	\tl_if_single:nT	4926
\tl_if_head_eq_charcode:fNTF	5191	\tl_if_single:NTF	95, 4925, 4928
\tl_if_head_eq_charcode:nN	5191	\tl_if_single:nTF	95, 4928, 4929
\tl_if_head_eq_charcode:nNF	5209	\tl_if_single_p:N	95, 4925, 4925
\tl_if_head_eq_charcode:nNT	5208	\tl_if_single_p:n	95, 4925, 4929

<code>\tl_if_single_token:n</code>	15034	<code>\tl_put_right:cV</code>	4704
<code>\tl_if_single_token:nTF</code>	196, 15034	<code>\tl_put_right:cx</code>	4704
<code>\tl_if_single_token_p:n</code>	196, 15034	<code>\tl_put_right:Nn</code>	
<code>\tl_item:cn</code>	15127 92, 4704, 4704, 4720, 5464, 8522	
<code>\tl_item:Nn</code>	15127, 15149, 15150	<code>\tl_put_right:No</code>	4704, 4708, 4722
<code>\tl_item:nn</code>	197, 15127, 15127, 15149	<code>\tl_put_right:NV</code>	4704, 4706, 4721
<code>\tl_length:c</code>	5365, 5367	<code>\tl_put_right:Nx</code> ...	4704, 4710, 4723,
<code>\tl_length:N</code>	5365, 5366	6389, 6409, 8488, 8513, 8526, 8535,	
<code>\tl_length:n</code>	5365, 5368	9586, 9592, 9599, 9618, 9627, 9638	
<code>\tl_length:o</code>	5365, 5370	<code>\tl_remove_all:cn</code>	4832, 5344
<code>\tl_length:V</code>	5365, 5369	<code>\tl_remove_all:Nn</code>	
<code>\tl_length_tokens:n</code>	5362, 5363 93, 4832, 4832, 4836, 5343, 9174	
<code>\tl_map_break:</code> ..	96, 4948, 4954, 4965,	<code>\tl_remove_all_in:cn</code>	5338, 5344
4974, 4981, 4986, 4986, 4987, 4989		<code>\tl_remove_all_in:Nn</code>	5338, 5343
<code>\tl_map_break:n</code>	97, 4986, 4988	<code>\tl_remove_in:cn</code>	5338, 5340
<code>\tl_map_function:cn</code>	4944	<code>\tl_remove_in:Nn</code>	5338, 5339
<code>\tl_map_function:NN</code>		<code>\tl_remove_once:cn</code>	4826, 5340
..... 95, 4944, 4950, 4957, 5007		<code>\tl_remove_once:Nn</code>	
<code>\tl_map_function:nN</code> 92, 4826, 4826, 4830, 5339	
..... 96, 4944, 4944, 4951, 5002, 5428		<code>\tl_replace_all:cn</code>	4776, 5334
<code>\tl_map_inline:cn</code>	4958	<code>\tl_replace_all:Nnn</code>	
<code>\tl_map_inline:Nn</code> ..	96, 4958, 4967, 4969 92, 4776, 4780, 4786,	
<code>\tl_map_inline:nn</code>	96, 2864,	4833, 5333, 5436, 8464, 8465, 9544	
4958, 4958, 4968, 9503, 11030, 11039		<code>\tl_replace_all_in:cn</code>	5328, 5334
<code>\tl_map_variable:cn</code>	4970	<code>\tl_replace_all_in:Nnn</code>	5328, 5333
<code>\tl_map_variable:NNn</code> 96, 4970, 4976, 4985		<code>\tl_replace_in:cn</code>	5328, 5330
<code>\tl_map_variable:nNn</code> 96, 4970, 4970, 4977		<code>\tl_replace_in:Nnn</code>	5328, 5329
<code>\tl_new:c</code> ...	4595, 5404, 5840, 6284, 9071	<code>\tl_replace_once:cn</code>	4776, 5330
<code>\tl_new:cn</code>	5313	<code>\tl_replace_once:Nnn</code>	
<code>\tl_new:N</code> .	91, 2584, 3080, 4595, 4595, 92, 4776, 4776, 4784, 4827, 5329	
4600, 4620, 4622, 4908, 4909, 5309,		<code>\tl_rescan:nn</code>	93, 4738, 4742
5310, 5311, 5312, 5316, 5400, 5401,		<code>\tl_reverse:c</code>	5140
5403, 5837, 5839, 6281, 6283, 6516,		<code>\tl_reverse:N</code>	98, 5140, 5140, 5144
6784, 6807, 6808, 7390, 7641, 7928,		<code>\tl_reverse:n</code>	
7929, 8447, 8448, 8449, 8450, 8561,	 98, 5120, 5120, 5133, 5141, 5143	
8562, 8563, 8565, 8566, 8567, 8568,		<code>\tl_reverse:o</code>	5120
9133, 9153, 9154, 9303, 9416, 9487,		<code>\tl_reverse:V</code>	5120
9488, 9489, 9490, 9491, 14074, 14854		<code>\tl_reverse_items:n</code>	98, 5012, 5012
<code>\tl_new:Nn</code>	5313, 5314, 5319, 5320	<code>\tl_reverse_tokens:n</code>	
<code>\tl_new:Nx</code>	5313 196, 15040, 15040, 15057	
<code>\tl_put_left:cn</code>	4680	<code>.tl_set:c</code>	154
<code>\tl_put_left:co</code>	4680	<code>\tl_set:cf</code>	4662
<code>\tl_put_left:cV</code>	4680	<code>\tl_set:cn</code>	4662, 8705, 8709, 9074
<code>\tl_put_left:cx</code>	4680	<code>\tl_set:co</code>	4662
<code>\tl_put_left:Nn</code> 92, 4680, 4680, 4696, 5462		<code>\tl_set:cV</code>	4662
<code>\tl_put_left:No</code>	4680, 4684, 4698	<code>\tl_set:cv</code>	4662
<code>\tl_put_left:NV</code>	4680, 4682, 4697	<code>\tl_set:cx</code>	4662
<code>\tl_put_left:Nx</code>	4680, 4686, 4699	<code>.tl_set:N</code>	154
<code>\tl_put_right:cn</code>	4704	<code>\tl_set:Nc</code>	5322, 5324, 5325
<code>\tl_put_right:co</code>	4704	<code>\tl_set:Nf</code>	4662, 8352

`\tl_set:Nn`
 . 92, 3098, 3119, 3740, 4662, 4662,
 4674, 4676, 4739, 4898, 4899, 4980,
 5430, 5506, 5515, 5529, 5532, 5553,
 5561, 5580, 5588, 5600, 5648, 5717,
 5936, 5942, 5951, 5958, 6159, 6319,
 6335, 6336, 6344, 6345, 6347, 6353,
 6356, 6366, 6367, 6376, 6394, 6414,
 6463, 6791, 6795, 6982, 7278, 7279,
 7392, 7395, 7939, 8021, 8463, 8575,
 8608, 8619, 8689, 8742, 8886, 8899,
 8943, 9101, 9189, 9194, 9566, 14050
`\tl_set:No` 4662, 4664, 5326
`\tl_set:Nv` 4662
`\tl_set:Nx` ... 4634, 4662, 4666, 4675,
 4777, 4781, 5031, 5141, 5422, 5441,
 5496, 5570, 5610, 5626, 5652, 5858,
 5902, 5945, 5983, 6035, 6389, 8506,
 8510, 8519, 8534, 8573, 8603, 8613,
 8616, 8884, 8894, 8916, 8917, 9167,
 9168, 9209, 9346, 9439, 9537, 9542,
 9543, 9606, 9633, 13805, 13872,
 13877, 13939, 13958, 14550, 14857,
 14944, 14949, 14970, 14988, 14998
`\tl_set_eq:cc`
 .. 4625, 4628, 5416, 5852, 6296, 8751
`\tl_set_eq:cN` 4625, 4626, 5415, 5851, 6295
`\tl_set_eq:Nc`
 .. 4625, 4627, 5414, 5850, 6294, 8950
`\tl_set_eq:NN` 91, 4614, 4625, 4625, 5413,
 5849, 6293, 7984, 7992, 13813, 13908
`\tl_set_rescan:cnm` 4738
`\tl_set_rescan:cno` 4738
`\tl_set_rescan:cnx` 4738
`\tl_set_rescan:Nmn`
 93, 4738, 4738, 4768, 4769
`\tl_set_rescan:Nno` 4738
`\tl_set_rescan:Nnx` 4738
`.tl_set_x:c` 154
`.tl_set_x:N` 154
`\tl_show:c` 5297
`\tl_show:N` 102, 5297, 5297, 5306
`\tl_show:n` 102, 5307, 5307
`\tl_tail:f` 5146
`\tl_tail:N` 100, 5146, 5175
`\tl_tail:n` 5146, 5167, 5174, 5175
`\tl_tail:V` 5146
`\tl_tail:v` 5146
`\tl_tail:w` 5384, 5385
`\tl_to_lowercase:n`
 . 93, 2342, 2357, 2824, 2866, 3023,
 4772, 4772, 7765, 8275, 8456, 9496,
 10347, 10401, 13568, 15161, 15178
`\tl_to_str:c` 4991
`\tl_to_str:N` 97, 4991,
 4991, 4992, 9169, 9543, 9556, 9557
`\tl_to_str:n` 97, 809, 3253, 4283,
 4389, 4475, 4793, 4862, 4875, 4990,
 4990, 5156, 5179, 5188, 5308, 6306,
 6395, 6415, 6431, 6432, 7420, 7503,
 7828, 7829, 8072, 8073, 8359, 8363,
 8364, 8368, 8369, 8573, 8613, 8884,
 8894, 8916, 8994, 9010, 9239, 9280,
 9293, 9501, 9647, 13695, 13696,
 13715, 13718, 14880, 14881, 15204
`\tl_to_uppercase:n` 94, 4772, 4774
`\tl_trim_spaces:c` 5028
`\tl_trim_spaces:N` .. 99, 5028, 5030, 5034
`\tl_trim_spaces:n` 99, 5028,
 5028, 5031, 5033, 5448, 8510, 8534
`\tl_use:c` 4993, 6270
`\tl_use:N` 97, 4993, 4993, 4998, 6269
`\token_get_arg_spec:N` ... 61, 3251, 3264
`\token_get_prefix_spec:N` . 61, 3251, 3255
`\token_get_replacement_spec:N`
 61, 3251, 3273
`\token_if_active:N` 2798
`\token_if_active:Nf` 3372
`\token_if_active:NT` 3371
`\token_if_active:NTF` 55, 2798, 3373
`\token_if_active_char:Nf` 3372
`\token_if_active_char:NT` 3371
`\token_if_active_char:NTF` ... 3357, 3373
`\token_if_active_char_p:N` ... 3357, 3370
`\token_if_active_p:N` 55, 2798, 3370
`\token_if_alignment:N` 2760
`\token_if_alignment:Nf` 3360
`\token_if_alignment:NT` 3359
`\token_if_alignment:NTF` . 54, 2760, 3361
`\token_if_alignment_p:N` . 54, 2760, 3358
`\token_if_alignment_tab:Nf` 3360
`\token_if_alignment_tab:NT` 3359
`\token_if_alignment_tab:NTF` . 3357, 3361
`\token_if_alignment_tab_p:N` . 3357, 3358
`\token_if_chardef:N` 2869
`\token_if_chardef:NTF` ... 56, 2858, 2908
`\token_if_chardef_p:N` 56, 2858
`\token_if_cs:N` 2841
`\token_if_cs:NTF` 56, 2841

<code>\token_if_cs_p:N</code>	56, 2841	<code>\token_if_math_toggle:NT</code>	3363
<code>\token_if_dim_register:N</code>	2888	<code>\token_if_math_toggle:NTF</code> 54, 2755, 3365	
<code>\token_if_dim_register:NTF</code>	57, 2858	<code>\token_if_math_toggle_p:N</code> 54, 2755, 3362	
<code>\token_if_dim_register_p:N</code>	57, 2858	<code>\token_if_mathchardef:N</code>	2878
<code>\token_if_eq_catcode:NN</code>	2808	<code>\token_if_mathchardef:NTF</code> 57, 2858, 2910	
<code>\token_if_eq_catcode:NNTF</code>	55, 2808	<code>\token_if_mathchardef_p:N</code>	57, 2858
<code>\token_if_eq_catcode_p:NN</code>	55, 2808	<code>\token_if_muskip_register:N</code>	2930
<code>\token_if_eq_charcode:NN</code>	2813	<code>\token_if_muskip_register:NTF</code> . 57, 2858	
<code>\token_if_eq_charcode:NNTF</code>	55, 2813	<code>\token_if_muskip_register_p:N</code> . 57, 2858	
<code>\token_if_eq_charcode_p:NN</code>	55, 2813	<code>\token_if_other:N</code>	2793
<code>\token_if_eq_meaning:NN</code>	2803	<code>\token_if_other:NF</code>	3368
<code>\token_if_eq_meaning:NNF</code>	10107	<code>\token_if_other:NT</code>	3367
<code>\token_if_eq_meaning:NNT</code>	2352	<code>\token_if_other:NTF</code>	55, 2793, 3369
<code>\token_if_eq_meaning:NNTF</code>		<code>\token_if_other_char:NF</code>	3368
. 56, 2367, 2803, 11134		<code>\token_if_other_char:NT</code>	3367
<code>\token_if_eq_meaning_p:NN</code>	56, 2803	<code>\token_if_other_char:NTF</code>	3357, 3369
<code>\token_if_expandable:N</code>	2846	<code>\token_if_other_char_p:N</code>	3357, 3366
<code>\token_if_expandable:NTF</code>	56, 2846	<code>\token_if_other_p:N</code>	55, 2793, 3366
<code>\token_if_expandable_p:N</code>	56, 2846	<code>\token_if_parameter:N</code>	2767
<code>\token_if_group_begin:N</code>	2745	<code>\token_if_parameter:NTF</code>	55, 2765
<code>\token_if_group_begin:NTF</code>	54, 2745	<code>\token_if_parameter_p:N</code>	55, 2765
<code>\token_if_group_begin_p:N</code>	54, 2745	<code>\token_if_primitive:N</code>	3025
<code>\token_if_group_end:N</code>	2750	<code>\token_if_primitive:NTF</code>	57, 3017
<code>\token_if_group_end:NTF</code>	54, 2750	<code>\token_if_primitive_p:N</code>	57, 3017
<code>\token_if_group_end_p:N</code>	54, 2750	<code>\token_if_protected_long_macro:N</code> .	3005
<code>\token_if_int_register:N</code>	2906	<code>\token_if_protected_long_macro:NTF</code> .	
<code>\token_if_int_register:NTF</code>	57, 2858 56, 2858	
<code>\token_if_int_register_p:N</code>	57, 2858	<code>\token_if_protected_long_macro_p:N</code> .	
<code>\token_if_letter:N</code>	2788 56, 2858	
<code>\token_if_letter:NTF</code>	55, 2788	<code>\token_if_protected_macro:N</code>	2984
<code>\token_if_letter_p:N</code>	55, 2788	<code>\token_if_protected_macro:NTF</code> . 56, 2858	
<code>\token_if_long_macro:N</code>	2996	<code>\token_if_protected_macro_p:N</code> . 56, 2858	
<code>\token_if_long_macro:NTF</code>	56, 2858	<code>\token_if_skip_register:N</code>	2948
<code>\token_if_long_macro_p:N</code>	56, 2858	<code>\token_if_skip_register:NTF</code> . . . 57, 2858	
<code>\token_if_macro:N</code>	2827	<code>\token_if_skip_register_p:N</code> . . . 57, 2858	
<code>\token_if_macro:NTF</code>		<code>\token_if_space:N</code>	2783
. 56, 2818, 3027, 3257, 3266, 3275		<code>\token_if_space:NTF</code>	55, 2783
<code>\token_if_macro_p:N</code>	56, 2818	<code>\token_if_space_p:N</code>	55, 2783
<code>\token_if_math_shift:NF</code>	3364	<code>\token_if_toks_register:N</code>	2966
<code>\token_if_math_shift:NT</code>	3363	<code>\token_if_toks_register:NTF</code> . . . 57, 2858	
<code>\token_if_math_shift:NTF</code>	3357, 3365	<code>\token_if_toks_register_p:N</code> . . . 57, 2858	
<code>\token_if_math_shift_p:N</code>	3357, 3362	<code>\token_new:Nn</code> 53, 2705, 2705, 2710, 2712,	
<code>\token_if_math_subscript:N</code>	2778	2713, 2714, 2716, 2717, 2718, 2719	
<code>\token_if_math_subscript:NTF</code> . . 55, 2778		<code>\token_to_meaning:c</code>	800, 811, 2705
<code>\token_if_math_subscript_p:N</code> . . 55, 2778		<code>\token_to_meaning:N</code>	
<code>\token_if_math_superscript:N</code>	2773 54, 787, 787, 1198, 1208,	
<code>\token_if_math_superscript:NTF</code> 55, 2773		1908, 2705, 2830, 2874, 2883, 2899,	
<code>\token_if_math_superscript_p:N</code> 55, 2773		2921, 2941, 2959, 2977, 2990, 3001,	
<code>\token_if_math_toggle:N</code>	2755	3011, 3031, 3260, 3269, 3278, 15188	
<code>\token_if_math_toggle:NF</code>	3364		

\token_to_str:c . . .	800, 800, 930, 939, 960, 980, 1018, 1023, 1038, 1927, 2705
\token_to_str:N	5, 54, 787, 788, 800, 1070, 1071, 1198, 1208, 1210, 1221, 1342, 1372, 1454, 1469, 1988, 2003, 2036, 2138, 2363, 2590, 2705, 2876, 2885, 2901, 2923, 2943, 2961, 2979, 2992, 3003, 3013, 5276, 5303, 6684, 6833, 6981, 7595, 7599, 8246, 8253, 8260, 8343, 9166, 9526, 9527, 9528, 9529, 9530, 9985, 9986, 10353, 10361, 10362, 10388, 10549, 10570, 10585, 10597, 10598, 10611, 10612, 10637, 10646, 10648, 10673, 10676, 10726, 10736, 10737, 10752, 10753, 10797, 10813, 10824, 10876, 13843
\toks	630
\toksdef	331
\tolerance	541
\topmark	422
\topmarks	648
\topskip	552
\tracingassigns	658
\tracingcommands	397
\tracinggroups	665
\tracingifs	661
\tracinglostchars	398
\tracingmacros	399
\tracingnesting	660
\tracingonline	400
\tracingoutput	401
\tracingpages	402
\tracingparagraphs	403
\tracingrestores	404
\tracingscantokens	659
\tracingstats	405
U	
\U	15174
\uccode	640
\uchyph	538
\underline	474
\unexpanded	186, 190, 653
\unhbox	579
\unhcopy	580
\unkern	504
\unless	644
\unpenalty	615
\unskip	502
\unvbox	581
\unvcopy	582
\uppercase	612
\use:c	17, 858, 858, 955, 1033, 1164, 1166, 1168, 1170, 2184, 2203, 3579, 3909, 3919, 4062, 4071, 4073, 4075, 4076, 4080, 4287, 7861, 7872, 7887, 7896, 7904, 7912, 7918, 7944, 8324, 8627, 8634, 8757, 9611
\use:n	18, 864, 864, 986, 1289, 1451, 1479, 1481, 1485, 1493, 1495, 1503, 1507, 2726, 4743, 4982, 5248, 5267, 6161, 6681, 8287, 10030, 10038, 10047, 10064, 10072, 10100, 14874
\use:nn	864, 865, 1646, 2735, 3251, 4281, 6145, 13036
\use:nnn	864, 866, 1339
\use:nnnn	864, 867
\use:x	20, 859, 859, 942, 1004, 1937, 4386, 4751, 4762, 6664, 7825, 7842, 8069, 8086, 9176, 9401, 9552
\use_i:nn 19, 804, 868, 868, 894, 970, 1114, 1142, 1321, 1483, 1497, 1505, 2059, 2180, 2193, 2197, 2214, 2215, 5157, 6312, 9956, 11953, 12862, 13031, 13470
\use_i:nnn	19, 870, 870, 1096, 3260, 11911
\use_i:nnnn	19, 870, 874, 11929, 11936, 12126
\use_i_after_else:nw	1604, 1606
\use_i_after_fi:nw	1604, 1605
\use_i_after_or:nw	1604, 1607
\use_i_after_orelse:nw	1604, 1608
\use_i_delimit_by_q_nil:nw	20, 881, 881
\use_i_delimit_by_q_recursion_stop:nw	20, 47, 881, 883, 2503, 2519
\use_i_delimit_by_q_stop:nw 20, 881, 882, 14515
\use_i_ii:nnn	19, 870, 873, 1671, 12507, 12528, 13430
\use_ii:nn	19, 806, 868, 869, 896, 972, 1116, 1144, 1319, 1480, 1486, 1494, 1508, 1556, 2193, 5159, 6313, 8478, 9930, 11955, 12864, 13472, 13780
\use_ii:nnn 19, 870, 871, 1098, 3269, 8506, 8527
\use_ii:nnnn	19, 870, 875
\use_iii:nnn 19, 870, 872, 1561, 3278, 9939, 9946
\use_iii:nnnn	19, 870, 876
\use_iv:nnnn	19, 870, 877

- `\use_none:n` 19, 884, 884, 986,
 1291, 1478, 1482, 1484, 1492, 1496,
 1504, 1506, 2505, 2521, 3065, 3840,
 3955, 3959, 3964, 4839, 5071, 5156,
 5172, 5251, 5273, 5292, 5295, 5398,
 5599, 5621, 5625, 5679, 5963, 6056,
 7783, 8134, 8138, 8475, 9699, 9701,
 9895, 9899, 9903, 9907, 9967, 9995,
 10922, 11930, 11933, 12427, 13788,
 14560, 14910, 14935, 14936, 15037
`\use_none:nn` . 884, 885, 932, 940, 1471,
 4930, 5027, 5198, 5217, 5511, 5605,
 9844, 9894, 9898, 9902, 9906, 14589
`\use_none:nnn` 884, 886, 5239,
 8505, 8520, 9893, 9897, 9901, 9905
`\use_none:nnnn` . 884, 887, 1989, 2004, 3710
`\use_none:nnnnn`
 . 884, 888, 10025, 10059, 10085, 10093
`\use_none:nnnnnn` 884, 889, 1040
`\use_none:nnnnnnn` 884, 890, 962, 10027,
 10061, 10087, 10095, 10295, 11969
`\use_none:nnnnnnnn` 884, 891
`\use_none:nnnnnnnnn` 884, 892
`\use_none_delimit_by_q_nil:w` 20, 878, 878
`\use_none_delimit_by_q_recursion_stop:w`
 20, 47, 878, 880, 953, 1019,
 1024, 1031, 1928, 1935, 2497, 2512
`\use_none_delimit_by_q_stop:w`
 20, 878, 879,
 2384, 2388, 3590, 4291, 4801, 6043,
 7962, 9640, 9654, 14502, 14507, 15190
`\usepackage` 149, 230
- V**
- `\vadjust` 515
`\valign` 350
`.value_forbidden:` 154
`.value_required:` 154
`\vbadness` 590
`\vbox` 585
`\vbox:n` 133, 6724, 6724
`\vbox_gset:cn` 6730
`\vbox_gset:cw` 6747, 6763
`\vbox_gset:Nn` 134, 6730, 6732, 6734
`\vbox_gset:Nw` . 134, 6747, 6749, 6752, 6762
`\vbox_gset_end:` . . . 134, 6747, 6758, 6764
`\vbox_gset_inline_begin:c` . . . 6759, 6763
`\vbox_gset_inline_begin:N` . . . 6759, 6762
`\vbox_gset_inline_end:` 6759, 6764
`\vbox_gset_to_ht:cnn` 6741
`\vbox_gset_to_ht:Nnn` 134, 6741, 6743, 6746
`\vbox_gset_top:cn` 6735
`\vbox_gset_top:Nn` . 134, 6735, 6737, 6740
`\vbox_set:cn` 6730
`\vbox_set:cw` 6747, 6760
`\vbox_set:Nn`
 . . . 134, 6730, 6730, 6732, 6733, 6877
`\vbox_set:Nw`
 134, 6747, 6747, 6750, 6751, 6759, 6924
`\vbox_set_end:`
 . . . 134, 6747, 6753, 6758, 6761, 6934
`\vbox_set_inline_begin:c` 6759, 6760
`\vbox_set_inline_begin:N` 6759, 6759
`\vbox_set_inline_end:` 6759, 6761
`\vbox_set_split_to_ht:NNn` 134, 6769, 6769
`\vbox_set_to_ht:cnn` 6741
`\vbox_set_to_ht:Nnn`
 134, 6741, 6741, 6744, 6745
`\vbox_set_top:cn` 6735
`\vbox_set_top:Nn`
 134, 6735, 6735, 6738, 6739, 6891, 6938
`\vbox_to_ht:nn` 134, 6726, 6726, 6726
`\vbox_to_zero:n` . . . 134, 6726, 6726, 6728
`\vbox_top:n` 133, 6724, 6725
`\vbox_unpack:c` 6765
`\vbox_unpack:N`
 . . . 135, 6765, 6765, 6767, 6891, 6938
`\vbox_unpack_clear:c` 6765
`\vbox_unpack_clear:N` 135, 6765, 6766, 6768
`\vcenter` 436
`\vcoffin_set:cnn` 6873
`\vcoffin_set:cnw` 6920
`\vcoffin_set:Nnn` . . 137, 6873, 6873, 6902
`\vcoffin_set:Nnw` . . 137, 6920, 6920, 6953
`\vcoffin_set_end:` . . 137, 6920, 6931, 6952
`\vfil` 497
`\vfill` 499
`\vfilneg` 498
`\vfuzz` 592
`\voffset` 567
`\vrule` 506
`\vsize` 549
`\vskip` 500
`\vsplit` 578
`\vss` 501
`\vtop` 586
- W**
- `\wd` 633
`\widowpenalties` 693

\widowpenalty	520	\xetex_if_engine_p:	
TWOBARS	178	23, 1478, 1489, 1499, 1602
\write	383	\xetex_XeTeXversion:D	728, 1490
X			
\X	2043	\XeTeXversion	728
ex	181	\xleaders	509
\xdef	324	exp	179
\xetex_if_engine:F	1485, 1496	\xspaceskip	543
\xetex_if_engine:T	1484, 1495	Y	
\xetex_if_engine:TF	5, 23, 1478, 1486, 1497	\year	624