The LaTeX3 Sources

The LaTeX3 Project*

March 30, 2010

Abstract

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

The TeX and ε-TeX 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 LaTeX3 programming language.

At present, the expl3 modules are designed to be loaded on top of LaTeX 2_ε. In time, a LaTeX3 format will be produced based on this code. This allows the code to be used in LaTeX 2_ε packages *now* while a stand-alone LaTeX3 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.

*Frank Mittelbach, Denys Duchier, Chris Rowley, Rainer Schöpf, Johannes Braams, Michael Downes, David Carlisle, Alan Jeffrey, Morten Høgholm, Thomas Lotze, Javier Bezos, Will Robertson, Joseph Wright

Contents

I Introduction to expl3 and this document **1**

1 Naming functions and variables **1**

1.0.1 Terminological inexactitude . . . 3

2 Documentation conventions **3**

II The l3names package: A systematic naming scheme for TeX **5**

3 Setting up the LaTeX3 programming language **5**

4 Using the modules **5**

III The l3basics package: Basic Definitions **6**

5 Predicates and conditionals **7**

5.1 Primitive conditionals . . . 8

5.2 Non-primitive conditionals . . . 10

5.3 Applications . . . 11

6 Control sequences **11**

7 Selecting and discarding tokens from the input stream **12**

7.1 Extending the interface . . . 14

7.2 Selecting tokens from delimited arguments . . . 14

8 That which belongs in other modules but needs to be defined earlier **15**

9 Defining functions **16**

9.1 Defining new functions using primitive parameter text . . . 17

9.2 Defining new functions using the signature . . . 18

9.3 Defining functions using primitive parameter text . . . 20

9.4 Defining functions using the signature (no checks) 21
9.5 Undefining functions 23
9.6 Copying function definitions 23
9.7 Internal functions 24

10 The innards of a function **24**

11 Grouping and scanning **25**

12 Checking the engine **26**

IV The l3expan package: Controlling Expansion of Function Arguments **26**

13 Brief overview **26**

14 Defining new variants **27**
14.1 Methods for defining variants 27

15 Introducing the variants **28**

16 Manipulating the first argument **29**

17 Manipulating two arguments **30**

18 Manipulating three arguments **31**

19 Preventing expansion **32**

20 Unbraced expansion **33**

V The l3prg package: Program control structures **33**

21 Conditionals and logical operations **33**

22 Defining a set of conditional functions **34**

23 The boolean data type **35**

24 Boolean expressions **37**

25 Case switches **38**

26 Generic loops **39**

27 Choosing modes **39**

28 Alignment safe grouping and scanning **40**

29 Producing n copies **40**

30 Sorting **41**

30.1 Variable type and scope 42

VI The l3quark package: "Quarks" **42**

31 Functions **43**

32 Recursion **44**

33 Constants **44**

VII The l3token package: A token of my appreciation... **45**

34 Character tokens **45**

35 Generic tokens **48**

35.1 Useless code: because we can! 52

36 Peeking ahead at the next token **52**

VIII The l3int package: Integers/counters **54**

37 Functions **54**

38 Formatting a counter value **56**

38.1 Internal functions 56

39 Variable and constants **57**

40 Conversion **59**

IX The l3intexpr package: Integer expressions **59**

41 Calculating and comparing integers **59**

42 Primitive (internal) functions **61**

X The l3skip package: Dimension and skip registers **62**

43 Skip registers **63**

43.1 Functions 63

43.2 Formatting a skip register value 65

43.3 Variable and constants 65

44 Dim registers **65**

44.1 Functions 65

44.2 Variable and constants 68

45 Muskips **68**

XI The l3tl package: Token Lists **69**

46 Functions **69**

47 Predicates and conditionals **73**

48 Working with the contents of token lists **75**

49 Variables and constants **76**

50 Searching for and replacing tokens 77

51 Heads or tails? 78

XII The l3toks package: Token Registers 79

52 Allocation and use 80

53 Adding to the contents of token registers 82

54 Predicates and conditionals 83

55 Variable and constants 84

XIII The l3seq package: Sequences 84

56 Functions for creating/initialising sequences 85

57 Adding data to sequences 86

58 Working with sequences 87

59 Predicates and conditionals 88

60 Internal functions 89

61 Functions for 'Sequence Stacks' 89

XIV The l3clist package: Comma separated lists 90

62 Functions for creating/initialising comma-lists 90

63 Putting data in 91

64 Getting data out 92

65 Mapping functions 93

66 Predicates and conditionals 94

67 Higher level functions 95

68 Functions for ‘comma-list stacks’ 96

69 Internal functions 96

XV The l3prop package: Property Lists 97

70 Functions 97

71 Predicates and conditionals 100

72 Internal functions 100

XVI The l3io package: Low-level file i/o 101

73 Opening and closing streams 102
73.1 Writing to files 103
73.2 Reading from files 104

74 Internal functions 105

75 Variables and constants 105

XVII The l3msg package: Communicating with the user 106

76 Creating new messages 106

77 Message classes 107

78 Redirecting messages 108

79 Support functions for output 109

80 Low-level functions 110

81 Kernel-specific functions **110**

82 Variables and constants **112**

XVIII The l3box package: Boxes **112**

83 Generic functions **113**

84 Horizontal mode **116**

85 Vertical mode **117**

XIX The l3xref package: Cross references **118**

XX The l3keyval package: Key-value parsing **119**

86 Features of l3keyval **119**

87 Functions for keyval processing **120**

88 Internal functions **121**

89 Variables and constants **122**

XXI The l3keys package: Key–value support **122**

90 Creating keys **124**

91 Sub-dividing keys **127**

91.1 Multiple choices 127

92 Setting keys **128**

92.1 Examining keys: internal representation 129

93 Internal functions **129**

94 Variables and constants **132**

XXII The l3calc package: Infix notation arithmetic in LaTeX3 132

95 User functions **133**

XXIII The l3file package: File Loading 135

96 Loading files **135**

XXIV Implementation 136

97 l3names implementation **136**

97.1 Internal functions . . . 136
97.2 Package loading . . . 137
97.3 Catcode assignments . . . 137
97.4 Setting up primitive names . . . 138
97.5 Reassignment of primitives . . . 139
97.6 expl3 code switches . . . 149
97.7 Package loading . . . 151
97.8 Finishing up . . . 154
97.9 Showing memory usage . . . 156

98 l3basics implementation **157**

98.1 Renaming some TeX primitives (again) . . . 157
98.2 Defining functions . . . 159
98.3 Selecting tokens . . . 160
98.4 Gobbling tokens from input . . . 161
98.5 Expansion control from l3expan . . . 161
98.6 Conditional processing and definitions . . . 161
98.7 Dissecting a control sequence . . . 166
98.8 Exist or free . . . 168

98.9 Defining and checking (new) functions 170
98.10 More new definitions 173
98.11 Copying definitions 175
98.12 Undefining functions 176
98.13 Engine specific definitions 177
98.14 Scratch functions 177
98.15 Defining functions from a given number of arguments 177
98.16 Using the signature to define functions 179

99 l3expan implementation **182**
99.1 Internal functions and variables 182
99.2 Module code 183
99.3 General expansion 183
99.4 Hand-tuned definitions 187
99.5 Definitions with the 'general' technique 188
99.6 Preventing expansion 189
99.7 Defining function variants 190
99.8 Last-unbraced versions 192

100 l3prg implementation **193**
100.1 Variables 193
100.2 Module code 194
100.3 Choosing modes 194
100.4 Producing n copies 196
100.5 Booleans 199
100.6 Parsing boolean expressions 200
100.7 Case switch 205
100.8 Sorting 207
100.9 Variable type and scope 210

101 l3quark implementation **211**

102 l3token implementation **214**

102.1 Documentation of internal functions . . . 214
102.2 Module code . . . 215
102.3 Character tokens . . . 215
102.4 Generic tokens . . . 217
102.5 Peeking ahead at the next token . . . 225

103 l3int implementation **231**

103.1 Internal functions and variables . . . 231
103.2 Module loading and primitives definitions . . . 232
103.3 Allocation and setting . . . 232
103.4 Defining constants . . . 239
103.5 Scanning and conversion . . . 240

104 l3intexpr implementation **243**

105 l3skip implementation **248**

105.1 Skip registers . . . 249
105.2 Dimen registers . . . 252
105.3 Muskips . . . 255

106 l3tl implementation **255**

106.1 Functions . . . 256
106.2 Variables and constants . . . 261
106.3 Predicates and conditionals . . . 261
106.4 Working with the contents of token lists . . . 267
106.5 Checking for and replacing tokens . . . 272
106.6 Heads or tails? . . . 274

107 l3toks implementation **276**

107.1 Allocation and use . . . 276
107.2 Adding to token registers' contents . . . 279
107.3 Predicates and conditionals . . . 280
107.4 Variables and constants . . . 281

108 l3seq implementation **282**

108.1 Allocating and initialisation . . . 282
108.2 Predicates and conditionals . . . 283
108.3 Getting data out . . . 284
108.4 Putting data in . . . 285
108.5 Mapping . . . 286
108.6 Manipulation . . . 287
108.7 Sequence stacks . . . 287

109 l3clist implementation **288**

109.1 Allocation and initialisation . . . 288
109.2 Predicates and conditionals . . . 289
109.3 Retrieving data . . . 290
109.4 Storing data . . . 291
109.5 Mapping . . . 292
109.6 Higher level functions . . . 294
109.7 Stack operations . . . 296

110 l3prop implementation **296**

110.1 Functions . . . 297
110.2 Predicates and conditionals . . . 300
110.3 Mapping functions . . . 301

111 l3io implementation **303**

111.1 Variables and constants . . . 303
111.2 Stream management . . . 304
111.3 Immediate writing . . . 309
111.4 Deferred writing . . . 310

112 Special characters for writing **310**

112.1 Reading input . . . 310

113 l3msg implementation **311**

113.1 Variables and constants . . . 311

113.2 Output helper functions . . . 313

113.3 Generic functions . . . 313

113.4 General functions . . . 315

113.5 Redirection functions . . . 319

113.6 Kernel-specific functions . . . 319

114 l3box implementation **322**

114.1 Generic boxes . . . 322

114.2 Vertical boxes . . . 325

114.3 Horizontal boxes . . . 326

115 l3xref implementation **327**

115.1 Internal functions and variables . . . 327

115.2 Module code . . . 327

116 l3xref test file **330**

117 l3keyval implementation **332**

117.1 Module code . . . 332

117.1.1 Variables and constants . . . 340

117.1.2 Internal functions . . . 341

117.1.3 Properties . . . 348

117.1.4 Messages . . . 351

118 l3calc implementation **353**

118.1 Variables . . . 353

118.2 Internal functions . . . 354

118.3 Module code . . . 354

118.4 Higher level commands . . . 364

119 l3file implementation **367**

Index **371**

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 though 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 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*: the plain TeX `\edef`.

`f` The `f` specifier stands for *full expansion*, and in contrast to *x* stops at the first non-expandable item without trying to execute it.

`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 module[1] name and then a descriptive part. Variables end with a short identifier to show the variable type:

`bool` Either true or false.

`box` Box register.

`clist` Comma separated list.

`dim` 'Rigid' lengths.

`int` Integer-valued count register.

[1]The 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.

num A 'fake' integer type using only macros. Useful for setting up allocation routines.

prop Property list.

skip 'Rubber' lengths.

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

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

tl Token list variables: placeholder for a token list.

toks Token register.

1.0.1 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 mayn't 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 retreived with specific functions.

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

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

2 Documentation conventions

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

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

\ExplSyntaxOn
\ExplSyntaxOff

`\ExplSyntaxOn ... \ExplSyntaxOff`

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

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

\seq_new:N
\seq_new:c

`\seq_new:N` ⟨*sequence*⟩

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

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

\cs_to_str:N ⋆

`\cs_to_str:N` ⟨*cs*⟩

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

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:

\xetex_if_engine:*TF* ⋆

`\xetex_if_engine:TF` ⟨*true code*⟩ ⟨*false 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 ⟨*true code*⟩ and ⟨*false code*⟩ will be shown. The two variant forms `T` and `F` take only ⟨*true code*⟩ and ⟨*false code*⟩, respectively. Here, the star also shows that this function is expandable. With some minor exceptions, *all* conditional functions in the expl3 modules should be defined in this way.

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

`\l_tmpa_tl` A short piece of text will describe the variable: there is no syntax illustration in this case.

In some cases, the function is similar to one in LaTeX 2ε or plain TeX. In these cases, the text will include an extra '**TeXhackers note**' section:

`\token_to_str:N ⋆` `\token_to_str:N` ⟨*token*⟩

The normal description text.

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

Part II

The l3names package
A systematic naming scheme for TeX

3 Setting up the LaTeX3 programming language

This module is at the core of the LaTeX3 programming language. It performs the following tasks:

- defines new names for all TeX primitives;
- defines catcode regimes for programming;
- provides settings for when the code is used in a format;
- provides tools for when the code is used as a package within a LaTeX 2ε context.

4 Using the modules

The modules documented in `source3` are designed to be used on top of LaTeX 2ε and are loaded all as one with the usual `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions. These modules will also form the basis of the LaTeX3 format, but work in this area is incomplete and not included in this documentation.

As the modules use a coding syntax different from standard LaTeX it provides a few functions for setting it up.

```
\ExplSyntaxOn
\ExplSyntaxOff
```

\ExplSyntaxOn ⟨*code*⟩ \ExplSyntaxOff

Issues a catcode regime where spaces are ignored and colon and underscore are letters. A space character may by input with ~ instead.

```
\ExplSyntaxNamesOn
\ExplSyntaxNamesOff
```

\ExplSyntaxNamesOn ⟨*code*⟩ \ExplSyntaxNamesOff

Issues a catcode regime where colon and underscore are letters, but spaces remain the same.

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

\RequirePackage{expl3}
\ProvidesExplPackage {⟨*package*⟩}
{⟨*date*⟩} {⟨*version*⟩} {⟨*description*⟩}

The package l3names (this module) provides \ProvidesExplPackage which is a wrapper for \ProvidesPackage and sets up the LaTeX3 catcode settings for programming automatically. Similar for the relationship between \ProvidesExplClass and \ProvidesClass. Spaces are not ignored in the arguments of these commands.

```
\GetIdInfo
\filename
\filenameext
\filedate
\fileversion
\filetimestamp
\fileauthor
\filedescription
```

\RequirePackage{l3names}
\GetIdInfo $Id: ⟨*cvs or svn info field*⟩ $ {⟨*description*⟩}

Extracts all information from a CVS or SVN field. Spaces are not ignored in these fields. The information pieces are stored in separate control sequences with \filename for the part of the file name leading up to the period, \filenameext for the extension, \filedate for date, \fileversion for version, \filetimestamp for the time and \fileauthor for the author.

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 LaTeX catcodes and the LaTeX3 catcode 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{\filename}{\filedate}{\fileversion}{\filedescription}
```

Part III

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.

5 Predicates and conditionals

LaTeX3 has three concepts for conditional flow processing:

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

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

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

These type of functions are known as 'conditionals'; whenever a `TF` function is defined it will usually be accompanied by `T` and `F` functions as well. These are provided for convenience when the branch only needs to go a single way. Package writers are free to choose which types to define but the kernel definitions will always provide all three versions.

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

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

Predicates 'Predicates' are functions that return a special type of boolean value which can be tested by the function `\if_predicate:w` or in 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

```
\if_predicate:w \cs_if_free_p:N \l_tmpz_tl ⟨true code⟩ \else:
⟨false code⟩ \fi:
```

or in expressions utilizing the boolean logic parser:

```
\bool_if:nTF {
  \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl
} {⟨true code⟩} {⟨false code⟩}
```

Like their branching cousins, predicate functions ensure that all underlying primitive `\else:` or `\fi:` have been removed before returning the boolean true or false values.[2]

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

Primitive conditionals There is a third variety of conditional, which is the original concept used in plain TeX and LaTeX. 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.

5.1 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 `\intexpr_compare_p:nNn` which is a wrapper for `\if_num: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_`.

`\if_true:` ⋆
`\if_false:` ⋆
`\or:` ⋆
`\else:` ⋆
`\fi:` ⋆
`\reverse_if:N` ⋆

```
\if_true: ⟨true code⟩ \else: ⟨false code⟩ \fi:
\if_false: ⟨true code⟩ \else: ⟨false code⟩ \fi:
\reverse_if:N ⟨primitive conditional⟩
```

`\if_true:` always executes ⟨*true code*⟩, while `\if_false:` always executes ⟨*false code*⟩.

[2] If defined using the interface provided.

\reverse_if:N reverses any two-way primitive conditional. \else: and \fi: delimit the branches of the conditional. \or: is used in case switches, see l3intexpr for more.

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

\if_meaning:w ⋆

\if_meaning:w ⟨*arg*$_1$⟩ ⟨*arg*$_2$⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

\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.

TeXhackers note: This is TeX's \ifx.

\if:w ⋆
\if_charcode:w ⋆
\if_catcode:w ⋆

\if:w ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:
\if_catcode:w ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

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

\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.

\if_cs_exist:N ⋆
\if_cs_exist:w ⋆

\if_cs_exist:N ⟨*cs*⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:
\if_cs_exist:w ⟨*tokens*⟩ \cs_end: ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

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.

```
\if_mode_horizontal: ⋆
\if_mode_vertical:   ⋆
\if_mode_math:       ⋆
\if_mode_inner:      ⋆
```
`\if_mode_horizontal:` ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`

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

5.2 Non-primitive conditionals

```
\cs_if_eq_name_p:NN
```
`\cs_if_eq_name_p:NN` ⟨cs_1⟩ ⟨cs_2⟩

Returns 'true' if ⟨cs_1⟩ and ⟨cs_2⟩ are textually the same, i.e. have the same name, otherwise it returns 'false'.

```
\cs_if_eq_p:NN ⋆
\cs_if_eq_p:cN ⋆
\cs_if_eq_p:Nc ⋆
\cs_if_eq_p:cc ⋆
\cs_if_eq:NNTF ⋆
\cs_if_eq:cNTF ⋆
\cs_if_eq:NcTF ⋆
\cs_if_eq:ccTF ⋆
```
`\cs_if_eq_p:NNTF` ⟨cs_1⟩ ⟨cs_2⟩
`\cs_if_eq:NNTF` ⟨cs_1⟩ ⟨cs_2⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

These functions check if ⟨cs_1⟩ and ⟨cs_2⟩ have same meaning.

```
\cs_if_free_p:N ⋆
\cs_if_free_p:c ⋆
\cs_if_free:NTF ⋆
\cs_if_free:cTF ⋆
```
`\cs_if_free_p:N` ⟨*cs*⟩
`\cs_if_free:NTF` ⟨*cs*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Returns 'true' if ⟨*cs*⟩ is either undefined or equal to `\tex_relax:D` (the function that is assigned to newly created control sequences by TeX when `\cs:w ... \cs_end:` is used). In addition to this, 'true' is only returned if ⟨*cs*⟩ does not have a signature equal to `D`, i.e., 'do not use' functions are not free to be redefined.

```
\cs_if_exist_p:N ⋆
\cs_if_exist_p:c ⋆
\cs_if_exist:NTF ⋆
\cs_if_exist:cTF ⋆
```
`\cs_if_exist_p:N` ⟨*cs*⟩
`\cs_if_exist:NTF` ⟨*cs*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

These functions check if ⟨*cs*⟩ exists, i.e., if ⟨*cs*⟩ is present in the hash table and is not the primitive `\tex_relax:D`.

```
\cs_if_do_not_use_p:N ⋆
```
`\cs_if_do_not_use_p:N` ⟨*cs*⟩

These functions check if ⟨*cs*⟩ has the arg spec `D` for '*do not use*'. There are no `TF`-type conditionals for this function as it is only used internally and not expected to be widely used. (For now, anyway.)

`\chk_if_free_cs:N`
`\chk_if_free_cs:c`

`\chk_if_free_cs:N` ⟨*cs*⟩

This function checks that ⟨*cs*⟩ is ⟨*free*⟩ according to the criteria for `\cs_if_free_p:N` above. If not, an error is generated.

`\chk_if_exist_cs:N`
`\chk_if_exist_cs:c`

`\chk_if_exist_cs:N` ⟨*cs*⟩

This function checks that ⟨*cs*⟩ is defined. If it is not an error is generated.

`\c_true_bool`
`\c_false_bool`

Constants that represent 'true' or 'false', respectively. Used to implement predicates.

5.3 Applications

`\str_if_eq_p:nn` ⋆

`\str_if_eq_p:nn {`⟨*string*$_1$⟩`} {`⟨*string*$_2$⟩`}`

Expands to 'true' if ⟨*string*$_1$⟩ is the same as ⟨*string*$_2$⟩, otherwise 'false'. Ignores spaces within the strings.

`\str_if_eq_var_p:nf` ⋆

`\str_if_eq_var_p:nf {`⟨*string*$_1$⟩`} {`⟨*string*$_2$⟩`}`

A variant of `\str_if_eq_p:nn` which has the advantage of obeying spaces in at least the second argument.

6 Control sequences

`\cs:w` ⋆
`\cs_end:` ⋆

`\cs:w` ⟨*tokens*⟩ `\cs_end:`

This is the TeX internal way of generating a control sequence from some token list. ⟨*tokens*⟩ get expanded and must ultimately result in a sequence of characters.

TeXhackers note: These functions are the primitives `\csname` and `\endcsname`. `\cs:w` is considered weird because it expands tokens until it reaches `\cs_end:`.

\cs_show:N
\cs_show:c

\cs_show:N ⟨*cs*⟩
\cs_show:c {⟨*arg*⟩}

This function shows in the console output the *meaning* of the control sequence ⟨*cs*⟩ or that created by ⟨*arg*⟩.

TeXhackers note: This is TeX's \show and associated csname version of it.

\cs_meaning:N ⋆
\cs_meaning:c ⋆

\cs_meaning:N ⟨*cs*⟩
\cs_meaning:c {⟨*arg*⟩}

This function expands to the *meaning* of the control sequence ⟨*cs*⟩ or that created by ⟨*arg*⟩.

TeXhackers note: This is TeX's \meaning and associated csname version of it.

7 Selecting and discarding tokens from the input stream

The conditional processing cannot be implemented without being able to gobble and select which tokens to use from the input stream.

\use:n ⋆
\use:nn ⋆
\use:nnn ⋆
\use:nnnn ⋆

\use:n {⟨*arg*⟩}

Functions that returns all of their arguments to the input stream after removing the surrounding braces around each argument.

TeXhackers note: \use:n is LaTeX 2_ε's \@firstofone/\@iden.

\use:c ⋆

\use:c {⟨*cs*⟩}

Function that returns to the input stream the control sequence created from its argument. Requires two expansions before a control sequence is returned.

TeXhackers note: \use:c is LaTeX 2_ε's \@nameuse.

\use:x

\use:x {⟨*expandable tokens*⟩}

Function that fully expands its argument before passing it to the input stream. Contents of the argument must be fully expandable.

TEXhackers note: LuaTEX provides `\expanded` which performs this operation in an expandable manner, but we cannot assume this behaviour on all platforms yet.

`\use_none:n` ⋆
`\use_none:nn` ⋆
`\use_none:nnn` ⋆
`\use_none:nnnn` ⋆
`\use_none:nnnnn` ⋆
`\use_none:nnnnnn` ⋆
`\use_none:nnnnnnn` ⋆
`\use_none:nnnnnnnn` ⋆
`\use_none:nnnnnnnnn` ⋆

`\use_none:n {`⟨*arg*$_1$⟩`}`
`\use_none:nn {`⟨*arg*$_1$⟩`} {`⟨*arg*$_2$⟩`}`

These functions gobble the tokens or brace groups from the input stream.

TEXhackers note: `\use_none:n`, `\use_none:nn`, `\use_none:nnnn` are LATEX 2ε's `\@gobble`, `\@gobbletwo`, and `\@gobblefour`.

`\use_i:nn` ⋆
`\use_ii:nn` ⋆

`\use_i:nn {`⟨*code*$_1$⟩`} {`⟨*code*$_2$⟩`}`

Functions that execute the first or second argument respectively, after removing the surrounding braces. Primarily used to implement conditionals.

TEXhackers note: These are LATEX 2ε's `\@firstoftwo` and `\@secondoftwo`, respectively.

`\use_i:nnn` ⋆
`\use_ii:nnn` ⋆
`\use_iii:nnn` ⋆

`\use_i:nnn {`⟨*arg*$_1$⟩`} {`⟨*arg*$_2$⟩`} {`⟨*arg*$_3$⟩`}`

Functions that pick up one of three arguments and execute them after removing the surrounding braces.

TEXhackers note: LATEX 2ε has only `\@thirdofthree`.

`\use_i:nnnn` ⋆
`\use_ii:nnnn` ⋆
`\use_iii:nnnn` ⋆
`\use_iv:nnnn` ⋆

`\use_i:nnnn {`⟨*arg*$_1$⟩`} {`⟨*arg*$_2$⟩`} {`⟨*arg*$_3$⟩`} {`⟨*arg*$_4$⟩`}`

Functions that pick up one of four arguments and execute them after removing the surrounding braces.

7.1 Extending the interface

`\use_i_ii:nnn` ⋆ `\use_i_ii:nnn {`⟨*arg*$_1$⟩`} {`⟨*arg*$_2$⟩`} {`⟨*arg*$_3$⟩`}`

This function used in the expansion module reads three arguments and returns (without braces) the first and second argument while discarding the third argument.

If you wish to select multiple arguments while discarding others, use a syntax like this. Its definition is

```
\cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}
```

7.2 Selecting tokens from delimited arguments

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

`\use_none_delimit_by_q_nil:w` ⋆
`\use_none_delimit_by_q_stop:w` ⋆
`\use_none_delimit_by_q_recursion_stop:w` ⋆ `\use_none_delimit_by_q_nil:w` ⟨*balanced text*⟩ `\q_nil`

Gobbles ⟨*balanced text*⟩. Useful in gobbling the remainder in a list structure or terminating recursion.

`\use_i_delimit_by_q_nil:nw` ⋆
`\use_i_delimit_by_q_stop:nw` ⋆
`\use_i_delimit_by_q_recursion_stop:nw` ⋆ `\use_i_delimit_by_q_nil:nw {`⟨*arg*⟩`}` ⟨*balanced text*⟩ `\q_`

Gobbles ⟨*balanced text*⟩ and executes ⟨*arg*⟩ afterwards. This can also be used to get the first item in a token list.

`\use_i_after_fi:nw` ⋆
`\use_i_after_else:nw` ⋆
`\use_i_after_or:nw` ⋆
`\use_i_after_orelse:nw` ⋆

`\use_i_after_fi:nw {`⟨*arg*⟩`} \fi:`
`\use_i_after_else:nw {`⟨*arg*⟩`} \else:` ⟨*balanced text*⟩ `\fi:`
`\use_i_after_or:nw {`⟨*arg*⟩`} \or:` ⟨*balanced text*⟩ `\fi:`
`\use_i_after_orelse:nw {`⟨*arg*⟩`} \or:/\else:` ⟨*balanced text*⟩ `\fi:`

Executes ⟨*arg*⟩ after executing closing out `\fi:`. `\use_i_after_orelse:nw` can be used anywhere where `\use_i_after_else:nw` or `\use_i_after_or:nw` are used.

8 That which belongs in other modules but needs to be defined earlier

`\exp_after:wN` ⋆ `\exp_after:wN` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩

Expands ⟨*token*$_2$⟩ once and then continues processing from ⟨*token*$_1$⟩.

TEXhackers note: This is TEX's `\expandafter`.

`\exp_not:N` ⋆
`\exp_not:n` ⋆

`\exp_not:N` ⟨*token*⟩
`\exp_not:n {`⟨*tokens*⟩`}`

In an expanding context, this function prevents ⟨*token*⟩ or ⟨*tokens*⟩ from expanding.

TEXhackers note: These are TEX's `\noexpand` and ε-TEX's `\unexpanded`, respectively.

`\prg_do_nothing:` ⋆ This is as close as we get to a null operation or no-op.

TEXhackers note: Definition as in LATEX's `\empty` but not used for the same thing.

`\iow_log:x`
`\iow_term:x`
`\iow_shipout_x:Nn`

`\iow_log:x {`⟨*message*⟩`}`
`\iow_shipout_x:Nn` ⟨*write_stream*⟩ `{`⟨*message*⟩`}`

Writes ⟨*message*⟩ to either to log or the terminal.

`\msg_kernel_bug:x` `\msg_kernel_bug:x {`⟨*message*⟩`}`

Internal function for calling errors in our code.

`\cs_record_meaning:N` Placeholder for a function to be defined by `l3chk`.

`\c_minus_one`
`\c_zero`
`\c_sixteen`

Numeric constants.

9 Defining functions

There are two types of function definitions in LaTeX3: versions that check if the function name is still unused, and versions that simply make the definition. The latter are used for internal scratch functions that get new meanings all over the place.

For each type there is an additional choice to be made: Does the function to be defined contain delimited arguments? The answer in 99% of the cases is no. For this type the programmer will know the number of arguments and in most cases use the argument signature to signal this, e.g., `\foo_bar:nnn` presumably takes three arguments. We therefore also provide functions that automatically detect how many arguments are required and construct the parameter text on the fly.

A definition of a new function can be done locally and globally. Currently nearly all function definitions are done locally on top level, in other words they are global but don't show it. Therefore I think it may be better to remove the local variants in the future and declare all checked function definitions global.

TeXhackers note: While TeX makes all definition functions directly available to the user LaTeX3 hides them very carefully to avoid the problems with definitions that are overwritten accidentally. Many functions that are in TeX a combination of prefixes and definition functions are provided as individual functions.

A slew of functions are defined in the following sections for defining new functions. Here's a quick summary to get an idea of what's available:
`\cs_(g)(new/set)(_protected)(_nopar):(N/c)(p)(n/x)`

That stands for, respectively, the following variations:

`g` Global or local;

`new/set` Define a new function or re-define an existing one;

`protected` Prevent expansion of the function in `x` arguments;

`nopar` Restrict the argument(s) from containing `\par`;

`N/c` Either a control sequence or a 'csname';

`p` Either the a primitive TeX argument or the number of arguments is detected from the argument signature, i.e., `\foo:nnn` is assumed to have three arguments `#1#2#3`;

`n/x` Either an unexpanded or an expanded definition.

That adds up to 128 variations (!). However, the system is very logical and only a handful will usually be required often.

9.1 Defining new functions using primitive parameter text

\cs_new:Npn
\cs_new:Npx
\cs_new:cpn
\cs_new:cpx

\cs_new:Npn ⟨*cs*⟩ ⟨*parms*⟩ {⟨*code*⟩}

Defines a function that may contain \par tokens in the argument(s) when called. This is not allowed for normal functions.

\cs_gnew:Npn
\cs_gnew:Npx
\cs_gnew:cpn
\cs_gnew:cpx

\cs_gnew:Npn ⟨*cs*⟩ ⟨*parms*⟩ {⟨*code*⟩}

Global versions of the above functions.

\cs_new_nopar:Npn
\cs_new_nopar:Npx
\cs_new_nopar:cpn
\cs_new_nopar:cpx

\cs_new_nopar:Npn ⟨*cs*⟩ ⟨*parms*⟩ {⟨*code*⟩}

Defines a new function, making sure that ⟨*cs*⟩ is unused so far. ⟨*parms*⟩ may consist of arbitrary parameter specification in TeX syntax. It is under the responsibility of the programmer to name the new function according to the rules laid out in the previous section. ⟨*code*⟩ is either passed literally or may be subject to expansion (under the x variants).

\cs_gnew_nopar:Npn
\cs_gnew_nopar:Npx
\cs_gnew_nopar:cpn
\cs_gnew_nopar:cpx

\cs_gnew_nopar:Npn ⟨*cs*⟩ ⟨*parms*⟩ {⟨*code*⟩}

Like \cs_new_nopar:Npn but defines the new function globally. See comments above.

\cs_new_protected:Npn
\cs_new_protected:Npx
\cs_new_protected:cpn
\cs_new_protected:cpx

\cs_new_protected:Npn ⟨*cs*⟩ ⟨*parms*⟩ {⟨*code*⟩}

Defines a function that is both robust and may contain \par tokens in the argument(s) when called.

\cs_gnew_protected:Npn
\cs_gnew_protected:Npx
\cs_gnew_protected:cpn
\cs_gnew_protected:cpx

\cs_gnew_protected:Npn ⟨*cs*⟩ ⟨*parms*⟩ {⟨*code*⟩}

Global versions of the above functions.

```
\cs_new_protected_nopar:Npn
\cs_new_protected_nopar:Npx
\cs_new_protected_nopar:cpn
\cs_new_protected_nopar:cpx
```

\cs_new_protected_nopar:Npn ⟨*cs*⟩ ⟨*parms*⟩ {⟨*code*⟩}

Defines a function that does not expand when inside an x type expansion.

```
\cs_gnew_protected_nopar:Npn
\cs_gnew_protected_nopar:Npx
\cs_gnew_protected_nopar:cpn
\cs_gnew_protected_nopar:cpx
```

\cs_gnew_protected_nopar:Npn ⟨*cs*⟩ ⟨*parms*⟩ {⟨*code*⟩}

Global versions of the above functions.

9.2 Defining new functions using the signature

```
\cs_new:Nn
\cs_new:Nx
\cs_new:cn
\cs_new:cx
```

\cs_new:Nn ⟨*cs*⟩ {⟨*code*⟩}

Defines a new function, making sure that ⟨*cs*⟩ is unused so far. The parameter text is automatically detected from the length of the function signature. If ⟨*cs*⟩ is missing a colon in its name, an error is raised. It is under the responsibility of the programmer to name the new function according to the rules laid out in the previous section. ⟨*code*⟩ is either passed literally or may be subject to expansion (under the x variants).

TEXhackers note: Internally, these use TEX's \long. These forms are recommended for low-level definitions as experience has shown that \par tokens often turn up in programming situations that wouldn't have been expected.

```
\cs_gnew:Nn
\cs_gnew:Nx
\cs_gnew:cn
\cs_gnew:cx
```

\cs_gnew:Nn ⟨*cs*⟩ {⟨*code*⟩}

Global versions of the above functions.

\cs_new_nopar:Nn
\cs_new_nopar:Nx
\cs_new_nopar:cn
\cs_new_nopar:cx

\cs_new_nopar:Nn ⟨*cs*⟩ {⟨*code*⟩}

Version of the above in which \par is not allowed to appear within the argument(s) of the defined functions.

\cs_gnew_nopar:Nn
\cs_gnew_nopar:Nx
\cs_gnew_nopar:cn
\cs_gnew_nopar:cx

\cs_gnew_nopar:Nn ⟨*cs*⟩ {⟨*code*⟩}

Global versions of the above functions.

\cs_new_protected:Nn
\cs_new_protected:Nx
\cs_new_protected:cn
\cs_new_protected:cx

\cs_new_protected:Nn ⟨*cs*⟩ {⟨*code*⟩}

Defines a function that is both robust and may contain \par tokens in the argument(s) when called.

\cs_gnew_protected:Nn
\cs_gnew_protected:Nx
\cs_gnew_protected:cn
\cs_gnew_protected:cx

\cs_gnew_protected:Nn ⟨*cs*⟩ {⟨*code*⟩}

Global versions of the above functions.

\cs_new_protected_nopar:Nn
\cs_new_protected_nopar:Nx
\cs_new_protected_nopar:cn
\cs_new_protected_nopar:cx

\cs_new_protected_nopar:Nn ⟨*cs*⟩ {⟨*code*⟩}

Defines a function that does not expand when inside an x type expansion. \par is not allowed in the argument(s) of the defined function.

\cs_gnew_protected_nopar:Nn
\cs_gnew_protected_nopar:Nx
\cs_gnew_protected_nopar:cn
\cs_gnew_protected_nopar:cx

\cs_gnew_protected_nopar:Nn ⟨*cs*⟩ {⟨*code*⟩}

Global versions of the above functions.

9.3 Defining functions using primitive parameter text

Besides the function definitions that check whether or not their argument is an unused function we need function definitions that overwrite currently used definitions. The following functions are provided for this purpose.

```
\cs_set:Npn
\cs_set:Npx
\cs_set:cpn
\cs_set:cpx
```

`\cs_set:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{`⟨*code*⟩`}`

Like `\cs_set_nopar:Npn` but allows `\par` tokens in the arguments of the function being defined.

TEXhackers note: These are equivalent to TEX's `\long\def` and so on. These forms are recommended for low-level definitions as experience has shown that `\par` tokens often turn up in programming situations that wouldn't have been expected.

```
\cs_gset:Npn
\cs_gset:Npx
\cs_gset:cpn
\cs_gset:cpx
```

`\cs_gset:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{`⟨*code*⟩`}`

Global variant of `\cs_set:Npn`.

```
\cs_set_nopar:Npn
\cs_set_nopar:Npx
\cs_set_nopar:cpn
\cs_set_nopar:cpx
```

`\cs_set_nopar:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{`⟨*code*⟩`}`

Like `\cs_new_nopar:Npn` etc. but does not check the ⟨*cs*⟩ name.

TEXhackers note: `\cs_set_nopar:Npn` is the LATEX3 name for TEX's `\def` and `\cs_set_nopar:Npx` corresponds to the primitive `\edef`. The `\cs_set_nopar:cpn` function was known in LATEX2 as `\@namedef`. `\cs_set_nopar:cpx` has no equivalent.

```
\cs_gset_nopar:Npn
\cs_gset_nopar:Npx
\cs_gset_nopar:cpn
\cs_gset_nopar:cpx
```

`\cs_gset_nopar:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{`⟨*code*⟩`}`

Like `\cs_set_nopar:Npn` but defines the ⟨*cs*⟩ globally.

TEXhackers note: `\cs_gset_nopar:Npn` and `\cs_gset_nopar:Npx` are TEX's `\gdef` and `\xdef`.

```
\cs_set_protected:Npn
\cs_set_protected:Npx
\cs_set_protected:cpn
\cs_set_protected:cpx
```
\cs_set_protected:Npn ⟨*cs*⟩ ⟨*parms*⟩ {⟨*code*⟩}

Naturally robust macro that won't expand in an x type argument. These varieties allow \par tokens in the arguments of the function being defined.

```
\cs_gset_protected:Npn
\cs_gset_protected:Npx
\cs_gset_protected:cpn
\cs_gset_protected:cpx
```
\cs_gset_protected:Npn ⟨*cs*⟩ ⟨*parms*⟩ {⟨*code*⟩}

Global versions of the above functions.

```
\cs_set_protected_nopar:Npn
\cs_set_protected_nopar:Npx
\cs_set_protected_nopar:cpn
\cs_set_protected_nopar:cpx
```
\cs_set_protected_nopar:Npn ⟨*cs*⟩ ⟨*parms*⟩ {⟨*code*⟩}

Naturally robust macro that won't expand in an x type argument. If you want for some reason to expand it inside an x type expansion, prefix it with \exp_after:wN \prg_do_nothing:.

```
\cs_gset_protected_nopar:Npn
\cs_gset_protected_nopar:Npx
\cs_gset_protected_nopar:cpn
\cs_gset_protected_nopar:cpx
```
\cs_gset_protected_nopar:Npn ⟨*cs*⟩ ⟨*parms*⟩ {⟨*code*⟩}

Global versions of the above functions.

9.4 Defining functions using the signature (no checks)

As above but now detecting the parameter text from inspecting the signature.

```
\cs_set:Nn
\cs_set:Nx
\cs_set:cn
\cs_set:cx
```
\cs_set:Nn ⟨*cs*⟩ {⟨*code*⟩}

Like \cs_set_nopar:Nn but allows \par tokens in the arguments of the function being defined.

```
\cs_gset:Nn
\cs_gset:Nx
\cs_gset:cn
\cs_gset:cx
```
`\cs_gset:Nn ⟨cs⟩ {⟨code⟩}`

Global variant of `\cs_set:Nn`.

```
\cs_set_nopar:Nn
\cs_set_nopar:Nx
\cs_set_nopar:cn
\cs_set_nopar:cx
```
`\cs_set_nopar:Nn ⟨cs⟩ {⟨code⟩}`

Like `\cs_new_nopar:Nn` etc. but does not check the ⟨*cs*⟩ name.

```
\cs_gset_nopar:Nn
\cs_gset_nopar:Nx
\cs_gset_nopar:cn
\cs_gset_nopar:cx
```
`\cs_gset_nopar:Nn ⟨cs⟩ {⟨code⟩}`

Like `\cs_set_nopar:Nn` but defines the ⟨*cs*⟩ globally.

```
\cs_set_protected:Nn
\cs_set_protected:cn
\cs_set_protected:Nx
\cs_set_protected:cx
```
`\cs_set_protected:Nn ⟨cs⟩ {⟨code⟩}`

Naturally robust macro that won't expand in an `x` type argument. These varieties also allow `\par` tokens in the arguments of the function being defined.

```
\cs_gset_protected:Nn
\cs_gset_protected:cn
\cs_gset_protected:Nx
\cs_gset_protected:cx
```
`\cs_gset_protected:Nn ⟨cs⟩ {⟨code⟩}`

Global versions of the above functions.

```
\cs_set_protected_nopar:Nn
\cs_set_protected_nopar:cn
\cs_set_protected_nopar:Nx
\cs_set_protected_nopar:cx
```
`\cs_set_protected_nopar:Nn ⟨cs⟩ {⟨code⟩}`

Naturally robust macro that won't expand in an `x` type argument. This also comes as a `long` version. If you for some reason want to expand it inside an `x` type expansion, prefix it with `\exp_after:wN \prg_do_nothing:`.

```
\cs_gset_protected_nopar:Nn
\cs_gset_protected_nopar:cn
\cs_gset_protected_nopar:Nx
\cs_gset_protected_nopar:cx
```

`\cs_gset_protected_nopar:Nn` ⟨*cs*⟩ {⟨*code*⟩}

Global versions of the above functions.

9.5 Undefining functions

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

`\cs_gundefine:N` ⟨*cs*⟩

Undefines the control sequence locally or globally. In a global context, this is useful for reclaiming a small amount of memory but shouldn't often be needed for this purpose. In a local context, this can be useful if you need to clear a definition before applying a short-term modification to something.

9.6 Copying function definitions

```
\cs_new_eq:NN
\cs_new_eq:cN
\cs_new_eq:Nc
\cs_new_eq:cc
\cs_gnew_eq:NN
\cs_gnew_eq:cN
\cs_gnew_eq:Nc
\cs_gnew_eq:cc
```

`\cs_new_eq:NN` ⟨cs_1⟩ ⟨cs_2⟩

Gives the function ⟨cs_1⟩ locally or globally the current meaning of ⟨cs_2⟩. If ⟨cs_1⟩ already exists then an error is called.

```
\cs_set_eq:NN
\cs_set_eq:cN
\cs_set_eq:Nc
\cs_set_eq:cc
\cs_gset_eq:NN
\cs_gset_eq:cN
\cs_gset_eq:Nc
\cs_gset_eq:cc
```

`\cs_set_eq:cN` ⟨cs_1⟩ ⟨cs_2⟩

Gives the function ⟨cs_1⟩ the current meaning of ⟨cs_2⟩. Again, we may always do this globally.

| \cs_set_eq:NwN |

\cs_set_eq:NwN ⟨*cs*$_1$⟩ ⟨*cs*$_2$⟩
\cs_set_eq:NwN ⟨*cs*$_1$⟩ = ⟨*cs*$_2$⟩

These functions assign the meaning of ⟨*cs*$_2$⟩ locally or globally to the function ⟨*cs*$_1$⟩. Because the TEX primitive operation is being used which may have an equal sign and (a certain number of) spaces between ⟨*cs*$_1$⟩ and ⟨*cs*$_2$⟩ the name contains a `w`. (Not happy about this convention!).

TEXhackers note: `\cs_set_eq:NwN` is the LATEX3 name for TEX's `\let`.

9.7 Internal functions

| \pref_global:D |
| \pref_long:D |
| \pref_protected:D |

`\pref_global:D \cs_set_nopar:Npn`

Prefix functions that can be used in front of some definition functions (namely ...). The result of prefixing a function definition with `\pref_global:D` makes the definition global, `\pref_long:D` change the argument scanning mechanism so that it allows `\par` tokens in the argument of the prefixed function, and `\pref_protected:D` makes the definition robust in `\write`s etc.

None of these internal functions should be used by a programmer since the necessary combinations are all available as separate function, e.g., `\cs_set:Npn` is internally implemented as `\pref_long:D \cs_set_nopar:Npn`.

TEXhackers note: These prefixes are the primitives `\global`, `\long`, and `\protected`. The `\outer` prefix isn't used at all within LATEX3 because ... (it causes more hassle than it's worth? It's nevery proved useful in any meaningful way?)

10 The innards of a function

| \cs_to_str:N ⋆ |

`\cs_to_str:N` ⟨*cs*⟩

This function returns the name of ⟨*cs*⟩ as a sequence of letters with the escape character removed.

| \token_to_str:N ⋆ |
| \token_to_str:c ⋆ |

`\token_to_str:N` ⟨*arg*⟩

This function return the name of ⟨*arg*⟩ as a sequence of letters including the escape character.

TEXhackers note: This is TEX's `\string`.

`\token_to_meaning:N ⋆` \token_to_meaning:N ⟨*arg*⟩

This function returns the type and definition of ⟨*arg*⟩ as a sequence of letters.

TEXhackers note: This is TEX's \meaning.

`\cs_get_function_name:N ⋆`
`\cs_get_function_signature:N ⋆` \cs_get_function_name:N \⟨*fn*⟩:⟨*args*⟩

The name variant strips off the leading escape character and the trailing argument specification (including the colon) to return ⟨*fn*⟩. The signature variants does the same but returns the signature ⟨*args*⟩ instead.

`\cs_split_function:NN ⋆` \cs_split_function:NN \⟨*fn*⟩:⟨*args*⟩ ⟨*post process*⟩

Strips off the leading escape character, splits off the signature without the colon, informs whether or not a colon was present and then prefixes these results with ⟨*post process*⟩, i.e., ⟨*post process*⟩{⟨*name*⟩}{⟨*signature*⟩}⟨*true*⟩/⟨*false*⟩. For example, \cs_get_function_name:N is nothing more than \cs_split_function:NN \⟨*fn*⟩:⟨*args*⟩ \use_i:nnn.

`\cs_get_arg_count_from_signature:N ⋆` \cs_get_arg_count_from_signature:N \⟨*fn*⟩:⟨*args*⟩

Returns the number of chars in ⟨*args*⟩, signifying the number of arguments that the function uses.

Other functions regarding arbitrary tokens can be found in the l3token module.

11 Grouping and scanning

`\scan_stop:` \scan_stop:

This function stops TEX's scanning ahead when ending a number.

TEXhackers note: This is the TEX primitive \relax renamed.

`\group_begin:`
`\group_end:` \group_begin: ⟨...⟩ \group_end:

Encloses ⟨...⟩ inside a group.

TeXhackers note: These are the TeX primitives `\begingroup` and `\endgroup` renamed.

`\group_execute_after:N` `\group_execute_after:N` ⟨*token*⟩

Adds ⟨*token*⟩ to the list of tokens to be inserted after the current group ends (through an explicit or implicit `\group_end:`).

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

12 Checking the engine

`\xetex_if_engine:TF` ⋆ `\xetex_if_engine:TF {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

This function detects if we're running a XeTeX-based format.

`\luatex_if_engine:TF` ⋆ `\luatex_if_engine:TF {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

This function detects if we're running a LuaTeX-based format.

`\c_xetex_is_engine_bool`
`\c_luatex_is_engine_bool` Boolean variables used for the above functions.

Part IV

The l3expan package Controlling Expansion of Function Arguments

13 Brief overview

The functions in this module all have prefix `exp`.

Not all possible variations are implemented for every base function. Instead only those that are used within the LaTeX3 kernel or otherwise seem to be of general interest are implemented. Consult the module description to find out which functions are actually defined. The next section explains how to define missing variants.

14 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 the third argument gets expanded once. If `\seq_gpush:No` wouldn't be defined the example above 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.

14.1 Methods for defining variants

`\cs_generate_variant:Nn`

`\cs_generate_variant:Nn` ⟨*parent control sequence*⟩
  `{`⟨*variant argument specifier*⟩`}`

The ⟨*parent control sequence*⟩ is first separated into the ⟨*base name*⟩ and ⟨*original*⟩ argument specifier. The ⟨*variant*⟩ is then used to modify this by replacing the beginning of the ⟨*original*⟩ with the ⟨*variant*⟩. Thus the ⟨*variant*⟩ must be no longer than the ⟨*original*⟩ argument specifier. This new specifier is used to create a modified function which will expand its arguments as required. 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 }
\cs_generate_variant:Nn \foo:Nn { cV }
```

would generate the functions `\foo:NV` and `\foo:cV` in the same way. `\cs_generate_-variant:Nn` can only be applied if the ⟨*parent control sequence*⟩ is already defined. If the ⟨*parent control sequence*⟩ is protected then the new sequence will also be protected.

Internal functions

`\cs_generate_internal_variant:n` `\cs_generate_internal_variant:n {`⟨*args*⟩`}`

Defines the appropriate `\exp_args:N`⟨*args*⟩ function, if necessary, to perform the expansion control specified by ⟨*args*⟩.

15 Introducing the variants

The available internal functions for argument expansion come in two flavours, some of them are faster then 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 very fast.
- In general `n`, `x`, and `o` (if not in the last position) will need special processing which is not fast and not expandable, i.e., functions of this type may not work correctly in arguments that are itself subject to `x` expansion. Therefore it is best to use the "expandable" functions (i.e., those that contain only `c`, `N`, `o` or `f` in the last position) whenever possible.

The `V` type returns the value of a register, which can be one of `tl`, `num`, `int`, `skip`, `dim`, `toks`, or built-in TeX registers. The `v` type is the same except it first creates a control sequence out of its argument before returning the value. This recent addition to the argument specifiers may shake things up a bit as most places where `o` is used will be replaced by `V`. The documentation you are currently reading will therefore require a fair bit of re-writing.

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

The `f` type is so special that it deserves an example. Let's pretend we want to set `\aaa` equal to the control sequence stemming from turning `b \l_tmpa_tl b` into a control sequence. Furthermore we want to store the execution of it in a ⟨*toks*⟩ register. In this example we assume `\l_tmpa_tl` contains the text string `lur`. The straight forward approach is

```
\toks_set:No \l_tmpa_toks {\cs_set_eq:Nc \aaa {b \l_tmpa_tl b}}
```

Unfortunately this only puts `\exp_args:NNc \cs_set_eq:NN \aaa {b \l_tmpa_tl b}` into `\l_tmpa_toks` and not `\cs_set_eq:NwN \aaa = \blurb` as we probably wanted. Using `\toks_set:Nx` is not an option as that will die horribly. Instead we can do a

```
\toks_set:Nf \l_tmpa_toks {\cs_set_eq:Nc \aaa {b \l_tmpa_tl b}}
```

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

```
\cs_set_nopar:Npn \toks_set:Nf {\exp_args:NNf \toks_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!

16 Manipulating the first argument

`\exp_args:No` ⋆ `\exp_args:No` ⟨*funct*⟩ ⟨arg_1⟩ ⟨arg_2⟩ ...

The first argument of ⟨*funct*⟩ (i.e., ⟨arg_1⟩) is expanded once, the result is surrounded by braces and passed to ⟨*funct*⟩. ⟨*funct*⟩ may have more than one argument—all others are passed unchanged.

\exp_args:Nc ⋆
\exp_args:cc ⋆

\exp_args:Nc ⟨*funct*⟩ ⟨*arg*$_1$⟩ ⟨*arg*$_2$⟩ ...

The first argument of ⟨*funct*⟩ (i.e., ⟨*arg*$_1$⟩) is expanded until only characters remain. (An internal error occurs if something else is the result of this expansion.) Then the result is turned into a control sequence and passed to ⟨*funct*⟩ as the first argument. ⟨*funct*⟩ may have more than one argument—all others are passed unchanged.

In the `:cc` variant, the ⟨*funct*⟩ control sequence itself is constructed (with the same process as described above) before ⟨*arg*$_1$⟩ is turned into a control sequence and passed as its argument.

\exp_args:NV ⋆

\exp_args:NV ⟨*funct*⟩ ⟨*register*⟩

The first argument of ⟨*funct*⟩ (i.e., ⟨*register*⟩) is expanded to its value. By value we mean a number stored in an `int` or `num` register, the length value of a `dim`, `skip` or `muskip` register, the contents of a `toks` register or the unexpanded contents of a `tl var.` register. The value is passed onto ⟨*funct*⟩ in braces.

\exp_args:Nv ⋆

\exp_args:Nv ⟨*funct*⟩ {⟨*register*⟩}

Like the `V` type except the register is given by a list of characters from which a control sequence name is generated.

\exp_args:Nx

\exp_args:Nx ⟨*funct*⟩ ⟨*arg*$_1$⟩ ⟨*arg*$_2$⟩ ...

The first argument of ⟨*funct*⟩ (i.e., ⟨*arg*$_1$⟩) is fully expanded until only unexpandable tokens remain, the result is surrounded by braces and passed to ⟨*funct*⟩. ⟨*funct*⟩ may have more than one argument—all others are passed unchanged. As mentioned before, this type of function is relatively slow.

\exp_args:Nf ⋆

\exp_args:Nf ⟨*funct*⟩ ⟨*arg*$_1$⟩ ⟨*arg*$_2$⟩ ...

The first argument of ⟨*funct*⟩ (i.e., ⟨*arg*$_1$⟩) undergoes full expansion until the first unexpandable token is encountered, the result is surrounded by braces and passed to ⟨*funct*⟩. ⟨*funct*⟩ may have more than one argument—all others are passed unchanged. Beware of its special behavior as explained above.

17 Manipulating two arguments

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

\exp_args:Nnx ⟨*funct*⟩ ⟨*arg*$_1$⟩ ⟨*arg*$_2$⟩ ...

The above functions all manipulate the first two arguments of ⟨*funct*⟩. They are all slow and non-expandable.

`\exp_args:NNo ⋆`
`\exp_args:NNc ⋆`
`\exp_args:NNv ⋆`
`\exp_args:NNV ⋆`
`\exp_args:NNf ⋆`
`\exp_args:Nno ⋆`
`\exp_args:NnV ⋆`
`\exp_args:Nnf ⋆`
`\exp_args:Noo ⋆`
`\exp_args:Noc ⋆`
`\exp_args:Nco ⋆`
`\exp_args:Ncf ⋆`
`\exp_args:Ncc ⋆`
`\exp_args:Nff ⋆`
`\exp_args:Nfo ⋆`
`\exp_args:NVV ⋆`

`\exp_args:NNo` ⟨*funct*⟩ ⟨*arg* $_1$⟩ ⟨*arg* $_2$⟩ ...

These are the fast and expandable functions for the first two arguments.

18 Manipulating three arguments

So far not all possible functions are provided and even the selection below may be reduced in the future as far as the non-expandable functions are concerned.

`\exp_args:NNnx`
`\exp_args:NNox`
`\exp_args:Nnnx`
`\exp_args:Nnox`
`\exp_args:Noox`
`\exp_args:Ncnx`
`\exp_args:Nccx`

`\exp_args:Nnnx` ⟨*funct*⟩ ⟨*arg* $_1$⟩ ⟨*arg* $_2$⟩ ⟨*arg* $_3$⟩ ...

All the above functions are non-expandable.

\exp_args:NNNo ⋆
\exp_args:NNNV ⋆
\exp_args:NNoo ⋆
\exp_args:NNno ⋆
\exp_args:Nnno ⋆
\exp_args:Nnnc ⋆
\exp_args:Nooo ⋆
\exp_args:Nccc ⋆
\exp_args:NcNc ⋆
\exp_args:NcNo ⋆
\exp_args:Ncco ⋆

`\exp_args:NNoo` ⟨*funct*⟩ ⟨arg_1⟩ ⟨arg_2⟩ ⟨arg_3⟩ ...

These are the fast and expandable functions for the first three arguments.

19 Preventing expansion

\exp_not:N
\exp_not:c
\exp_not:n

`\exp_not:N` ⟨*token*⟩
`\exp_not:n {`⟨*token list*⟩`}`

This function will prohibit the expansion of ⟨*token*⟩ in situation where ⟨*token*⟩ would otherwise be replaced by it definition, e.g., inside an argument that is handled by the `x` convention.

TEXhackers note: `\exp_not:N` is the primitive `\noexpand` renamed and `\exp_not:n` is the ε-TEX primitive `\unexpanded`.

\exp_not:o
\exp_not:f

`\exp_not:o {`⟨*token list*⟩`}`

Same as `\exp_not:n` except ⟨*token list*⟩ is expanded once for the `o` type and for the `f` type the token list is expanded until an unexpandable token is found, and the result of these expansions is then prohibited from being expanded further.

\exp_not:V
\exp_not:v

`\exp_not:V` ⟨*register*⟩
`\exp_not:v {`⟨*token list*⟩`}`

The value of ⟨*register*⟩ is retrieved and then passed on to `\exp_not:n` which will prohibit further expansion. The `v` type first creates a control sequence from ⟨*token list*⟩ but is otherwise identical to `V`.

\exp_stop_f:

⟨*f expansion*⟩ ... `\exp_stop_f:`

This function stops an `f` type expansion. An example use is one such as

```
\tl_set:Nf \l_tmpa_tl {
  \if_case:w \l_tmpa_int
    \or:   \use_i_after_orelse:nw {\exp_stop_f: \textbullet}
    \or:   \use_i_after_orelse:nw {\exp_stop_f: \textendash}
    \else: \use_i_after_fi:nw     {\exp_stop_f: else-item}
  \fi:
}
```

This ensures the expansion in stopped right after finishing the conditional but without expanding `\textbullet` etc.

TEXhackers note: This function is a space token but it is better to distinguish this expansion stopping token from a desired space token when writing code.

20 Unbraced expansion

`\exp_last_unbraced:Nf`
`\exp_last_unbraced:NV`
`\exp_last_unbraced:Nv`
`\exp_last_unbraced:NcV`
`\exp_last_unbraced:NNo`
`\exp_last_unbraced:NNV`
`\exp_last_unbraced:NNNo`

`\exp_last_unbraced:NV` ⟨*token*⟩ ⟨*variable name*⟩

There are a small number of occasions where the last argument in an expansion run must be expanded unbraced. These functions should only be used inside functions, *not* for creating variants.

Part V

The l3prg package Program control structures

21 Conditionals and logical operations

Conditional processing in LATEX3 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.

LaTeX3 has two primary forms of conditional flow processing based on these states. One type 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 other type 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

22 Defining a set of conditional functions

`\prg_return_true:`
`\prg_return_false:`

These functions exit conditional processing when used in conjunction with the generating functions listed below.

`\prg_set_conditional:Nnn`
`\prg_set_conditional:Npnn`
`\prg_new_conditional:Nnn`
`\prg_new_conditional:Npnn`
`\prg_set_protected_conditional:Nnn`
`\prg_set_protected_conditional:Npnn`
`\prg_new_protected_conditional:Nnn`
`\prg_new_protected_conditional:Npnn`
`\prg_set_eq_conditional:NNn`
`\prg_new_eq_conditional:NNn`

`\prg_set_conditional:Nnn` ⟨*test*⟩ ⟨*conds*⟩ ⟨*code*⟩
`\prg_set_conditional:Npnn` ⟨*test*⟩ ⟨*param*⟩ ⟨*conds*⟩ ⟨*code*⟩

This defines a conditional ⟨*base function*⟩ which upon evaluation using `\prg_return_true:` and `\prg_return_false:` to finish branches, returns a state. Currently the states are either ⟨*true*⟩ or ⟨*false*⟩ although this can change as more states may be introduced, say an ⟨*error*⟩ state. ⟨*conds*⟩ is a comma separated list possibly consisting of `p` for denoting a predicate function returning the boolean ⟨*true*⟩ or ⟨*false*⟩ values and `TF`, `T` and `F` for the functions that act on the tokens following in the input stream. The `:Nnn` form implicitly determines the number of arguments from the function being defined whereas the `:Npnn` form expects a primitive parameter text.

An example can easily clarify matters here:

```
\prg_set_conditional:Nnn \foo_if_bar:NN {p,TF,T} {
```

```
  \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, \foo_if_bar:NNT but not \foo_if_bar:NNF (because F is missing from the ⟨*conds*⟩ 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 an error if that branch is executed.

23 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 conditonal \bool_ functions are expandable and expect the input to also be fully expandable (which will generally mean being constructed from predicate functions, possibly nested).

\bool_new:N \bool_new:c

\bool_new:N ⟨*bool*⟩

Define a new boolean variable. The initial value is ⟨*false*⟩. A boolean is actually just either \c_true_bool or \c_false_bool.

```
\bool_set_true:N
\bool_set_true:c
\bool_set_false:N
\bool_set_false:c
\bool_gset_true:N
\bool_gset_true:c
\bool_gset_false:N
\bool_gset_false:c
```

`\bool_gset_false:N` ⟨*bool*⟩

Set ⟨*bool*⟩ either ⟨*true*⟩ or ⟨*false*⟩. We can also do this globally.

```
\bool_set_eq:NN
\bool_set_eq:Nc
\bool_set_eq:cN
\bool_set_eq:cc
\bool_gset_eq:NN
\bool_gset_eq:Nc
\bool_gset_eq:cN
\bool_gset_eq:cc
```

`\bool_set_eq:NN` ⟨*bool*$_1$⟩ ⟨*bool*$_2$⟩

Set ⟨*bool*$_1$⟩ equal to the value of ⟨*bool*$_2$⟩.

```
\bool_if_p:N ⋆
\bool_if:NTF ⋆
\bool_if_p:c ⋆
\bool_if:cTF ⋆
```

`\bool_if:NTF` ⟨*bool*⟩ `{`⟨*true*⟩`}` `{`⟨*false*⟩`}`
`\bool_if_p:N` ⟨*bool*⟩

Test the truth value of ⟨*bool*⟩ and execute the ⟨*true*⟩ or ⟨*false*⟩ code. `\bool_if_p:N` is a predicate function for use in `\if_predicate:w` tests or `\bool_if:nTF`-type functions described below.

```
\bool_while_do:Nn
\bool_while_do:cn
\bool_until_do:Nn
\bool_until_do:cn
\bool_do_while:Nn
\bool_do_while:cn
\bool_do_until:Nn
\bool_do_until:cn
```

`\bool_while_do:Nn` ⟨*bool*⟩ `{`⟨*code*⟩`}`
`\bool_until_do:Nn` ⟨*bool*⟩ `{`⟨*code*⟩`}`

The 'while' versions execute ⟨*code*⟩ as long as the boolean is true and the 'until' versions execute ⟨*code*⟩ as long as the boolean is false. The `while_do` functions execute the body after testing the boolean and the `do_while` functions executes the body first and then tests the boolean.

24 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean ⟨*true*⟩ or ⟨*false*⟩ values, it seems only fitting that we also provide a parser for ⟨*boolean expressions*⟩.

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

```
\intexpr_compare_p:n {1=1} &&
(
  \intexpr_compare_p:n {2=3} ||
  \intexpr_compare_p:n {4=4} ||
  \intexpr_compare_p:n {1=\error} % is skipped
) &&
!(\intexpr_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 anymore, the remaining tests within the current group are skipped.

\bool_if_p:n ⋆ \bool_if:n*TF* ⋆	`\bool_if:nTF` {⟨*boolean expression*⟩} {⟨*true*⟩} {⟨*false*⟩}

The functions evaluate the truth value of ⟨*boolean expression*⟩ where each predicate is separated by `&&` or `||` denoting logical 'And' and 'Or' functions. `(` and `)` denote grouping of sub-expressions while `!` is used to as a prefix to either negate a single expression or a group. Hence

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

from above returns ⟨*true*⟩.

Logical operators take higher precedence the later in the predicate they appear. "⟨*x*⟩ `||` ⟨*y*⟩ `&&` ⟨*z*⟩" is interpreted as the equivalent of "⟨*x*⟩ OR [⟨*y*⟩ AND ⟨*z*⟩]" (but now we have grouping you shouldn't write this sort of thing, anyway).

`\bool_not_p:n ⋆` `\bool_not_p:n` {⟨*boolean expression*⟩}

Longhand for writing `!(`⟨*boolean expression*⟩`)` within a boolean expression. Might not stick around.

`\bool_xor_p:nn ⋆` `\bool_xor_p:nn` {⟨*boolean expression*⟩} {⟨*boolean expression*⟩}

Implements an 'exclusive or' operation between two boolean expressions. There is no infix operation for this.

`\bool_set:Nn`
`\bool_set:cn`
`\bool_gset:Nn`
`\bool_gset:cn`

`\bool_set:Nn` ⟨*bool*⟩ {⟨*boolean expression*⟩}

Sets ⟨*bool*⟩ to the logical outcome of evaluating ⟨*boolean expression*⟩.

25 Case switches

`\prg_case_int:nnn ⋆`

```
\prg_case_int:nnn {⟨integer expr⟩} {
  {⟨integer expr_1⟩} {⟨code_1⟩}
  {⟨integer expr_2⟩} {⟨code_2⟩}
  ...
  {⟨integer expr_n⟩} {⟨code_n⟩}
} {⟨else case⟩}
```

This function evaluates the first ⟨*integer expr*⟩ and then compares it to the values found in the list. Thus the expression

```
\prg_case_int:nnn{2*5}{
  {5}{Small}  {4+6}{Medium}  {-2*10}{Negative}
}{Other}
```

evaluates first the term to look for and then tries to find this value in the list of values. If the value is found, the code on its right is executed after removing the remainder of the list. If the value is not found, the ⟨*else case*⟩ is executed. The example above will return "Medium".

The function is expandable and is written in such a way that `f` style expansion can take place cleanly, i.e., no tokens from within the function are left over.

`\prg_case_dim:nnn ⋆`

```
\prg_case_int:nnn {⟨dim expr⟩} {
  {⟨dim expr_1⟩} {⟨code_1⟩}
  {⟨dim expr_2⟩} {⟨code_2⟩}
  ...
  {⟨dim expr_n⟩} {⟨code_n⟩}
} {⟨else case⟩}
```

This function works just like `\prg_case_int:nnn` except it works for ⟨*dim*⟩ registers.

```
\prg_case_str:nnn ⋆
```

\prg_case_str:nnn {⟨*string*⟩} {
  {⟨*string*$_1$⟩} {⟨*code*$_1$⟩}
  {⟨*string*$_2$⟩} {⟨*code*$_2$⟩}
  ...
  {⟨*string*$_n$⟩} {⟨*code*,$_n$⟩}
} {⟨*else case*⟩}

This function works just like `\prg_case_int:nnn` except it compares strings. Each string is evaluated fully using `x` style expansion.

The function is expandable[3] and is written in such a way that `f` style expansion can take place cleanly, i.e., no tokens from within the function are left over.

```
\prg_case_tl:Nnn ⋆
```

\prg_case_tl:Nnn ⟨*tl var.*⟩ {
  ⟨*tl var.* $_1$⟩ {⟨*code*$_1$⟩} ⟨*tl var.* $_2$⟩ {⟨*code*$_2$⟩} ... ⟨*tl var.* $_n$⟩
{⟨*code*$_n$⟩}
} {⟨*else case*⟩}

This function works just like `\prg_case_int:nnn` except it compares token list variables.

The function is expandable[4] and is written in such a way that `f` style expansion can take place cleanly, i.e., no tokens from within the function are left over.

26 Generic loops

```
\bool_while_do:nn
\bool_until_do:nn
\bool_do_while:nn
\bool_do_until:nn
```

\bool_while_do:nn {⟨*boolean expression*⟩} {⟨*code*⟩}
\bool_until_do:nn {⟨*boolean expression*⟩} {⟨*code*⟩}

The 'while' versions execute the code as long as ⟨*boolean expression*⟩ is true and the 'until' versions execute ⟨*code*⟩ as long as ⟨*boolean expression*⟩ is false. The `while_do` functions execute the body after testing the boolean and the `do_while` functions executes the body first and then tests the boolean.

27 Choosing modes

```
\mode_if_vertical_p: ⋆
\mode_if_vertical:TF ⋆
```

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

Determines if TeX is in vertical mode or not and executes either ⟨*true code*⟩ or ⟨*false code*⟩ accordingly.

[3] Provided you use pdfTeX v1.30 or later
[4] Provided you use pdfTeX v1.30 or later

\mode_if_horizontal_p: ⋆
\mode_if_horizontal:TF ⋆ `\mode_if_horizontal:TF {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Determines if TEX is in horizontal mode or not and executes either ⟨*true code*⟩ or ⟨*false code*⟩ accordingly.

\mode_if_inner_p: ⋆
\mode_if_inner:TF ⋆ `\mode_if_inner:TF {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Determines if TEX is in inner mode or not and executes either ⟨*true code*⟩ or ⟨*false code*⟩ accordingly.

\mode_if_math_p: ⋆
\mode_if_math:TF ⋆ `\mode_if_math:TF {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Determines if TEX is in math mode or not and executes either ⟨*true code*⟩ or ⟨*false code*⟩ accordingly.

TEXhackers note: This version will choose the right branch even at the beginning of an alignment cell.

28 Alignment safe grouping and scanning

\scan_align_safe_stop: `\scan_align_safe_stop:`

This function gets TEX on the right track inside an alignment cell but without destroying any kerning.

\group_align_safe_begin:
\group_align_safe_end: `\group_align_safe_begin:` ⟨...⟩ `\group_align_safe_end:`

Encloses ⟨...⟩ inside a group but is safe inside an alignment cell. See the implementation of `\peek_token_generic:NNTF` for an application.

29 Producing n copies

There are often several different requirements for producing multiple copies of something. Sometimes one might want to produce a number of identical copies of a sequence of tokens whereas at other times the goal is to simulate a for loop as known from most real programming languages.

\prg_replicate:nn ⋆ `\prg_replicate:nn {⟨number⟩} {⟨arg⟩}`

Creates ⟨*number*⟩ copies of ⟨*arg*⟩. Note that it is expandable.

\prg_stepwise_function:nnnN ⋆ `\prg_stepwise_function:nnnN {⟨start⟩} {⟨step⟩} {⟨end⟩} ⟨function⟩`

This function performs ⟨*action*⟩ once for each step starting at ⟨*start*⟩ and ending once ⟨*end*⟩ is passed. ⟨*function*⟩ is placed directly in front of a brace group holding the current number so it should usually be a function taking one argument.

\prg_stepwise_inline:nnnn `\prg_stepwise_inline:nnnn {⟨start⟩} {⟨step⟩} {⟨end⟩} {⟨action⟩}`

Same as `\prg_stepwise_function:nnnN` except here ⟨*action*⟩ is performed each time with `##1` as a placeholder for the number currently being tested. This function is not expandable and it is nestable.

\prg_stepwise_variable:nnnNn `\prg_stepwise_variable:nnnn {⟨start⟩} {⟨step⟩} {⟨end⟩} ⟨temp-var⟩ {⟨action⟩}`

Same as `\prg_stepwise_inline:nnnn` except here the current value is stored in ⟨*temp-var*⟩ and the programmer can use it in ⟨*action*⟩. This function is not expandable.

30 Sorting

\prg_quicksort:n `\prg_quicksort:n { {⟨item_1⟩} {⟨item_2⟩} ... {⟨item n⟩} }`

Performs a Quicksort on the token list. The comparisons are performed by the function `\prg_quicksort_compare:nnTF` which is up to the programmer to define. When the sorting process is over, all items are given as argument to the function `\prg_quicksort_function:n` which the programmer also controls.

\prg_quicksort_function:n
\prg_quicksort_compare:nnTF
`\prg_quicksort_function:n {⟨element⟩}`
`\prg_quicksort_compare:nnTF {⟨element_1⟩} {⟨element_2⟩}`

The two functions the programmer must define before calling `\prg_quicksort:n`. As an example we could define

```
\cs_set_nopar:Npn\prg_quicksort_function:n #1{{#1}}
\cs_set_nopar:Npn\prg_quicksort_compare:nnTF #1#2#3#4 {\intexpr_compare:nNnTF{#1}>{
```

Then the function call

```
\prg_quicksort:n {876234520}
```

would return `{0}{2}{2}{3}{4}{5}{6}{7}{8}`. An alternative example where one sorts a list of words, `\prg_quicksort_compare:nnTF` could be defined as

```
\cs_set_nopar:Npn\prg_quicksort_compare:nnTF #1#2 {
  \intexpr_compare:nNnTF{\tl_compare:nn{#1}{#2}}>\c_zero }
```

30.1 Variable type and scope

`\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.)

Part VI

The l3quark package "Quarks"

A special type of constants in LaTeX3 are 'quarks'. These 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 is weird functions (for example as the stop token (i.e., `\q_stop`). They also permit the following ingenious trick: when you pick up a token in a temporary, and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary to the quark using `\if_meaning:w`. 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.

By convention all constants of type quark start out with `\q_`.

The documentation needs some updating.

31 Functions

`\quark_new:N`

`\quark_new:N` ⟨*quark*⟩

Defines ⟨*quark*⟩ to be a new constant of type `quark`.

`\quark_if_no_value_p:n` ⋆
`\quark_if_no_value:nTF` ⋆
`\quark_if_no_value_p:N` ⋆
`\quark_if_no_value:NTF` ⋆

`\quark_if_no_value:nTF` {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
`\quark_if_no_value:NTF` ⟨*tl var.*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

This tests whether or not ⟨*token list*⟩ contains only the quark `\q_no_value`.

If ⟨*token list*⟩ to be tested is stored in a token list variable use `\quark_if_no_value:NTF`, or `\quark_if_no_value:NF` or check the value directly with `\if_meaning:w`. All those cases are faster then `\quark_if_no_value:nTF` so should be preferred.[5]

TeXhackers note: But be aware of the fact that `\if_meaning:w` can result in an overflow of TeX's parameter stack since it leaves the corresponding `\fi:` on the input until the whole replacement text is processed. It is therefore better in recursions to use `\quark_if_no_value:NTF` as it will remove the conditional prior to processing the `T` or `F` case and so allows tail-recursion.

`\quark_if_nil_p:N` ⋆
`\quark_if_nil:NTF` ⋆

`\quark_if_nil:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

This tests whether or not ⟨*token*⟩ is equal to the quark `\q_nil`.

This is a useful test for recursive loops which typically has `\q_nil` as an end marker.

`\quark_if_nil_p:n` ⋆
`\quark_if_nil_p:V` ⋆
`\quark_if_nil_p:o` ⋆
`\quark_if_nil:nTF` ⋆
`\quark_if_nil:VTF` ⋆
`\quark_if_nil:oTF` ⋆

`\quark_if_nil:nTF` {⟨*tokens*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

This tests whether or not ⟨*tokens*⟩ is equal to the quark `\q_nil`.

This is a useful test for recursive loops which typically has `\q_nil` as an end marker.

[5] Clarify semantic of the "n" case ... i think it is not implement according to what we originally intended /FMi

32 Recursion

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

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

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

`\quark_if_recursion_tail_stop:N` ⋆
`\quark_if_recursion_tail_stop:n` ⋆
`\quark_if_recursion_tail_stop:o` ⋆

`\quark_if_recursion_tail_stop:n {`⟨*list element*⟩`}`
`\quark_if_recursion_tail_stop:N` ⟨*list element*⟩

This tests whether or not ⟨*list element*⟩ is equal to `\q_recursion_tail` and then exits, i.e., it gobbles the remainder of the list up to and including `\q_recursion_stop` which *must* be present.

If ⟨*list element*⟩ is not under your complete control it is advisable to use the `n`. If you wish to use the `N` form you *must* ensure it is really a single token such as if you have

`\tl_set:Nn \l_tmpa_tl {` ⟨*list element*⟩ `}`

`\quark_if_recursion_tail_stop_do:Nn` ⋆
`\quark_if_recursion_tail_stop_do:nn` ⋆
`\quark_if_recursion_tail_stop_do:on` ⋆

`\quark_if_recursion_tail_stop_do:nn`
`{`⟨*list element*⟩`} {`⟨*post action*⟩`}`
`\quark_if_recursion_tail_stop_do:Nn`
⟨*list element*⟩ `{`⟨*post action*⟩`}`

Same as `\quark_if_recursion_tail_stop:N` except here the second argument is executed after the recursion has been terminated.

33 Constants

`\q_no_value` The canonical 'missing value quark' that is returned by certain functions to denote that a requested value is not found in the data structure.

`\q_stop` This constant is used as a a marker in parameter text. This allows a scanning function to find the end of some input string.

`\q_nil` This constant represent the nil pointer in pointer structures.

`\q_error` Delimits the end of the computation for purposes of error recovery.

`\q_mark` Used in parameter text when we need a scanning boundary that is distinct from `\q_stop`.

Part VII

The l3token package A token of my appreciation...

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 TEX, 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 by 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.

34 Character tokens

```
\char_set_catcode:nn
\char_set_catcode:w
\char_value_catcode:n
\char_value_catcode:w
\char_show_value_catcode:n
\char_show_value_catcode:w
```

`\char_set_catcode:nn {`⟨*char number*⟩`} {`⟨*number*⟩`}`
`\char_set_catcode:w` ⟨*char*⟩ `=` ⟨*number*⟩
`\char_value_catcode:n {`⟨*char number*⟩`}`
`\char_show_value_catcode:n {`⟨*char number*⟩`}`

`\char_set_catcode:nn` sets the category code of a character, `\char_value_catcode:n` returns its value for use in integer tests and `\char_show_value_catcode:n` pausing the typesetting and prints the value on the terminal and in the log file. The `:w` form should be avoided. (Will: should we then just not mention it?)

`\char_set_catcode` is more usefully abstracted below.

TeXhackers note: `\char_set_catcode:w` is the TeX primitive `\catcode` renamed.

```
\char_make_escape:n
\char_make_begin_group:n
\char_make_end_group:n
\char_make_math_shift:n
\char_make_alignment: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
\char_make_invalid:n
```

`\char_make_letter:n {`⟨*character number*⟩`}`
`\char_make_letter:n {64}`
`\char_make_letter:n {‘\@}`

Sets the catcode of the character referred to by its ⟨*character number*⟩.

```
\char_make_escape:N
\char_make_begin_group:N
\char_make_end_group:N
\char_make_math_shift:N
\char_make_alignment: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
\char_make_invalid:N
```

`\char_make_letter:N {`⟨*character*⟩`}`
`\char_make_letter:N @`
`\char_make_letter:N \%`

Sets the catcode of the ⟨*character*⟩, which may have to be escaped.

TEXhackers note: `\char_make_other:N` is LaTeX 2ε's `\@makeother`.

`\char_set_lccode:nn`
`\char_set_lccode:w`
`\char_value_lccode:n`
`\char_value_lccode:w`
`\char_show_value_lccode:n`
`\char_show_value_lccode:w`

`\char_set_lccode:nn {`⟨*char*⟩`} {`⟨*number*⟩`}`
`\char_set_lccode:w` ⟨*char*⟩ `=` ⟨*number*⟩
`\char_value_lccode:n {`⟨*char*⟩`}`
`\char_show_value_lccode:n {`⟨*char*⟩`}`

Set the lower caser representation of ⟨*char*⟩ for when ⟨*char*⟩ is being converted in `\tl_to_lowercase:n`. As above, the `:w` form is only for people who really, really know what they are doing.

TEXhackers note: `\char_set_lccode:w` is the TEX primitive `\lccode` renamed.

`\char_set_uccode:nn`
`\char_set_uccode:w`
`\char_value_uccode:n`
`\char_value_uccode:w`
`\char_show_value_uccode:n`
`\char_show_value_uccode:w`

`\char_set_uccode:nn {`⟨*char*⟩`} {`⟨*number*⟩`}`
`\char_set_uccode:w` ⟨*char*⟩ `=` ⟨*number*⟩
`\char_value_uccode:n {`⟨*char*⟩`}`
`\char_show_value_uccode:n {`⟨*char*⟩`}`

Set the uppercase representation of ⟨*char*⟩ for when ⟨*char*⟩ is being converted in `\tl_to_uppercase:n`. As above, the `:w` form is only for people who really, really know what they are doing.

TEXhackers note: `\char_set_uccode:w` is the TEX primitive `\uccode` renamed.

`\char_set_sfcode:nn`
`\char_set_sfcode:w`
`\char_value_sfcode:n`
`\char_value_sfcode:w`
`\char_show_value_sfcode:n`
`\char_show_value_sfcode:w`

`\char_set_sfcode:nn {`⟨*char*⟩`} {`⟨*number*⟩`}`
`\char_set_sfcode:w` ⟨*char*⟩ `=` ⟨*number*⟩
`\char_value_sfcode:n {`⟨*char*⟩`}`
`\char_show_value_sfcode:n {`⟨*char*⟩`}`

Set the space factor for ⟨*char*⟩.

TEXhackers note: `\char_set_sfcode:w` is the TEX primitive `\sfcode` renamed.

\char_set_mathcode:nn
\char_set_mathcode:w
\char_gset_mathcode:nn
\char_gset_mathcode:w
\char_value_mathcode:n
\char_value_mathcode:w
\char_show_value_mathcode:n
\char_show_value_mathcode:w

\char_set_mathcode:nn {⟨*char*⟩} {⟨*number*⟩}
\char_set_mathcode:w ⟨*char*⟩ = ⟨*number*⟩
\char_value_mathcode:n {⟨*char*⟩}
\char_show_value_mathcode:n {⟨*char*⟩}

Set the math code for ⟨*char*⟩.

TEXhackers note: \char_set_mathcode:w is the TEX primitive \mathcode renamed.

35 Generic tokens

\token_new:Nn

\token_new:Nn ⟨*token*$_1$⟩ {⟨*token*$_2$⟩}

Defines ⟨*token*$_1$⟩ to globally be a snapshot of ⟨*token*$_2$⟩. This will be an implicit representation of ⟨*token*$_2$⟩.

\c_group_begin_token
\c_group_end_token
\c_math_shift_token
\c_alignment_tab_token
\c_parameter_token
\c_math_superscript_token
\c_math_subscript_token
\c_space_token
\c_letter_token
\c_other_char_token
\c_active_char_token

Some useful constants. They have category codes 1, 2, 3, 4, 6, 7, 8, 10, 11, 12, and 13 respectively. They are all implicit tokens.

\token_if_group_begin_p:N ⋆
\token_if_group_begin:N*TF* ⋆

\token_if_group_begin:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a begin group token.

\token_if_group_end_p:N ⋆
\token_if_group_end:N*TF* ⋆

\token_if_group_end:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is an end group token.

`\token_if_math_shift_p:N` ⋆
`\token_if_math_shift:NTF` ⋆ `\token_if_math_shift:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a math shift token.

`\token_if_alignment_tab_p:N` ⋆
`\token_if_alignment_tab:NTF` ⋆ `\token_if_aligment_tab:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is an aligment tab token.

`\token_if_parameter_p:N` ⋆
`\token_if_parameter:NTF` ⋆ `\token_if_parameter:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a parameter token.

`\token_if_math_superscript_p:N` ⋆
`\token_if_math_superscript:NTF` ⋆ `\token_if_math_superscript:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a math superscript token.

`\token_if_math_subscript_p:N` ⋆
`\token_if_math_subscript:NTF` ⋆ `\token_if_math_subscript:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a math subscript token.

`\token_if_space_p:N` ⋆
`\token_if_space:NTF` ⋆ `\token_if_space:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a space token.

`\token_if_letter_p:N` ⋆
`\token_if_letter:NTF` ⋆ `\token_if_letter:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a letter token.

`\token_if_other_char_p:N` ⋆
`\token_if_other_char:NTF` ⋆ `\token_if_other_char:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is an other char token.

\token_if_active_char_p:N ⋆
\token_if_active_char:NTF ⋆ \token_if_active_char:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is an active char token.

\token_if_eq_meaning_p:NN ⋆
\token_if_eq_meaning:NNTF ⋆ \token_if_eq_meaning:NNTF ⟨*token*$_1$⟩ ⟨*token*$_2$⟩{⟨*true*⟩} {⟨*false*⟩}

Check if the meaning of two tokens are identical.

\token_if_eq_catcode_p:NN ⋆
\token_if_eq_catcode:NNTF ⋆ \token_if_eq_catcode:NNTF ⟨*token*$_1$⟩ ⟨*token*$_2$⟩{⟨*true*⟩} {⟨*false*⟩}

Check if the category codes of two tokens are equal. If both tokens are control sequences the test will be true.

\token_if_eq_charcode_p:NN ⋆
\token_if_eq_charcode:NNTF ⋆ \token_if_eq_catcode:NNTF ⟨*token*$_1$⟩ ⟨*token*$_2$⟩{⟨*true*⟩} {⟨*false*⟩}

Check if the character codes of two tokens are equal. If both tokens are control sequences the test will be true.

\token_if_macro_p:N ⋆
\token_if_macro:NTF ⋆ \token_if_macro:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a macro.

\token_if_cs_p:N ⋆
\token_if_cs:NTF ⋆ \token_if_cs:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a control sequence or not. This can be useful for situations where the next token in the input stream is being looked at and you want to determine what should be done to it.

\token_if_expandable_p:N ⋆
\token_if_expandable:NTF ⋆ \token_if_expandable:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is expandable or not. Note that ⟨*token*⟩ can very well be an active character.

The next set of functions here are for picking apart control sequences. Sometimes it is useful to know if a control sequence has arguments and if so, how many. Similarly its status with respect to \long or \protected is good to have. Finally it can be very useful to know if a control sequence is of a certain type: Is this ⟨*toks*⟩ register we're trying to to something with really a ⟨*toks*⟩ register at all?

\token_if_long_macro_p:N ⋆ \token_if_long_macro:N*TF* ⋆

\token_if_long_macro:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a "long" macro.

\token_if_protected_macro_p:N ⋆ \token_if_protected_macro:N*TF* ⋆

\token_if_long_macro:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a "protected" macro. This test does *not* return ⟨*true*⟩ if the macro is also "long", see below.

\token_if_protected_long_macro_p:N ⋆ \token_if_protected_long_macro:N*TF* ⋆

\token_if_protected_long_macro:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a "protected long" macro.

\token_if_chardef_p:N ⋆ \token_if_chardef:N*TF* ⋆

\token_if_chardef:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is defined to be a chardef.

\token_if_mathchardef_p:N ⋆ \token_if_mathchardef:N*TF* ⋆

\token_if_mathchardef:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is defined to be a mathchardef.

\token_if_int_register_p:N ⋆ \token_if_int_register:N*TF* ⋆

\token_if_int_register:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is defined to be an integer register.

\token_if_dim_register_p:N ⋆ \token_if_dim_register:N*TF* ⋆

\token_if_dim_register:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is defined to be a dimension register.

```
\token_if_skip_register_p:N ⋆
\token_if_skip_register:NTF ⋆
```

`\token_if_skip_register:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is defined to be a skip register.

```
\token_if_toks_register_p:N ⋆
\token_if_toks_register:NTF ⋆
```

`\token_if_toks_register:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is defined to be a toks register.

```
\token_get_prefix_spec:N      ⋆
\token_get_arg_spec:N         ⋆
\token_get_replacement_spec:N ⋆
```

`\token_get_arg_spec:N` ⟨*token*⟩

If token is a macro with definition `\cs_set:Npn\next #1#2{x'#1--#2'y}`, the `prefix` function will return the string `\long`, the `arg` function returns the string `#1#2` and the `replacement` function returns the string `x'#1--#2'y`. If ⟨*token*⟩ isn't a macro, these functions return the `\scan_stop:` token.

If the `arg_spec` contains the string `->`, then the `spec` function will produce incorrect results.

35.1 Useless code: because we can!

```
\token_if_primitive_p:N ⋆
\token_if_primitive:NTF ⋆
```

`\token_if_primitive:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a primitive. Probably not a very useful function.

36 Peeking ahead at the next token

```
\l_peek_token
\g_peek_token
\l_peek_search_token
```

Some useful variables. Initially they are set to `?`.

```
\peek_after:NN
\peek_gafter:NN
```

`\peek_after:NN` ⟨*function*⟩⟨*token*⟩

Assign ⟨*token*⟩ to `\l_peek_token` and then run ⟨*function*⟩ which should perform some sort of test on this token. Leaves ⟨*token*⟩ in the input stream. `\peek_gafter:NN` does this globally to the token `\g_peek_token`.

TEXhackers note: This is the primitive `\futurelet` turned into a function.

```
\peek_meaning:NTF
\peek_meaning_ignore_spaces:NTF
\peek_meaning_remove:NTF
\peek_meaning_remove_ignore_spaces:NTF
```

`\peek_meaning:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

`\peek_meaning:NTF` checks (by using `\if_meaning:w`) if ⟨*token*⟩ equals the next token in the input stream and executes either ⟨*true code*⟩ or ⟨*false code*⟩ accordingly. `\peek_meaning_remove:NTF` does the same but additionally removes the token if found. The `ignore_spaces` versions skips blank spaces before making the decision.

TEXhackers note: This is equivalent to LATEX 2ε's `\@ifnextchar`.

```
\peek_charcode:NTF
\peek_charcode_ignore_spaces:NTF
\peek_charcode_remove:NTF
\peek_charcode_remove_ignore_spaces:NTF
```

`\peek_charcode:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Same as for the `\peek_meaning:NTF` functions above but these use `\if_charcode:w` to compare the tokens.

```
\peek_catcode:NTF
\peek_catcode_ignore_spaces:NTF
\peek_catcode_remove:NTF
\peek_catcode_remove_ignore_spaces:NTF
```

`\peek_catcode:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Same as for the `\peek_meaning:NTF` functions above but these use `\if_catcode:w` to compare the tokens.

```
\peek_token_generic:NNTF
\peek_token_remove_generic:NNTF
```

`\peek_token_generic:NNTF` ⟨*token*⟩⟨*function*⟩ {⟨*true*⟩} {⟨*false*⟩}

`\peek_token_generic:NNTF` looks ahead and checks if the next token in the input stream is equal to ⟨*token*⟩. It uses ⟨*function*⟩ to make that decision. `\peek_token_remove_generic:NNTF` does the same thing but additionally removes ⟨*token*⟩ from the input stream if it is found. This also works if ⟨*token*⟩ is either `\c_group_begin_token` or `\c_group_end_token`.

\peek_execute_branches_meaning:
\peek_execute_branches_charcode:
\peek_execute_branches_catcode: \peek_execute_branches_meaning:

These functions compare the token we are searching for with the token found (after optional ignoring of specific tokens). They come in the usual three versions when TEX is comparing tokens: meaning, character code, and category code.

Part VIII

The l3int package Integers/counters

LATEX3 maintains two type of integer registers for internal use. One (associated with the name `num`) for low level uses in the allocation mechanism using macros only and `int`: the one described here.

The `int` type uses the built-in counter registers of TEX and is therefore relatively fast compared to the `num` type and should be preferred in all cases as there is little chance we should ever run out of registers when being based on at least ε-TEX.

37 Functions

\int_new:N
\int_new:c
\int_new_local:N
\int_new_local:c \int_new:N ⟨*int*⟩

Defines ⟨*int*⟩ to be a new variable of type `int`. With the `local` variant, the variable is only available within the current group level (or below).

TEXhackers note: `\int_new:N` is the equivalent to plain TEX's `\newcount`.

\int_incr:N
\int_incr:c
\int_gincr:N
\int_gincr:c \int_incr:N ⟨*int*⟩

Increments ⟨*int*⟩ by one. For global variables the global versions should be used.

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

\int_decr:N ⟨*int*⟩

Decrements ⟨*int*⟩ by one. For global variables the global versions should be used.

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

\int_set:Nn ⟨*int*⟩ {⟨*integer expr*⟩}

These functions will set the ⟨*int*⟩ register to the ⟨*integer expr*⟩ value. This value can contain simple calc-like expressions as provided by ε-TEX.

\int_zero:N
\int_zero:c
\int_gzero:N
\int_gzero:c

\int_zero:N ⟨*int*⟩

These functions sets the ⟨*int*⟩ register to zero either locally or globally.

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

\int_add:Nn ⟨*int*⟩ {⟨*integer expr*⟩}

These functions will add to the ⟨*int*⟩ register the value ⟨*integer expr*⟩. If the second argument is a ⟨*int*⟩ register too, the surrounding braces can be left out.

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

\int_gsub:Nn ⟨*int*⟩ {⟨*integer expr*⟩}

These functions will subtract from the ⟨*int*⟩ register the value ⟨*integer expr*⟩. If the second argument is a ⟨*int*⟩ register too, the surrounding braces can be left out.

\int_use:N
\int_use:c

\int_use:N ⟨*int*⟩

This function returns the integer value kept in ⟨*int*⟩ in a way suitable for further processing.

TEXhackers note: The function \int_use:N could be implemented directly as the TEX primitive \tex_the:D which is also responsible to produce the values for other internal quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explaining.

\int_show:N
\int_show:c

`\int_show:N` ⟨*int*⟩

This function pauses the compilation and displays the integer value kept in ⟨*int*⟩ in the console output and log file.

TeXhackers note: The function `\int_show:N` could be implemented directly as the TeX primitive `\tex_showthe:D` which is also responsible to produce the values for other internal quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explaining.

38 Formatting a counter value

\int_to_arabic:n ⋆
\int_to_alph:n ⋆
\int_to_Alph:n ⋆
\int_to_roman:n ⋆
\int_to_Roman:n ⋆
\int_to_symbol:n ⋆

`\int_to_alph:n {`⟨*integer*⟩`}`
`\int_to_alph:n` ⟨*int*⟩

If some ⟨*integer*⟩ or the the current value of a ⟨*int*⟩ should be displayed or typeset in a special ways (e.g., as uppercase roman numerals) these function can be used. We need braces if the argument is a simple ⟨*integer*⟩, they can be omitted in case of a ⟨*int*⟩. By default the letters produced by `\int_to_roman:n` and `\int_to_Roman:n` have catcode 11.

All functions are fully expandable and will therefore produce the correct output when used inside of deferred writes, etc. In case the number in an `alph` or `Alph` function is greater than the default base number (26) it follows a simple conversion rule so that 27 is turned into `aa`, 50 into `ax` and so on and so forth. These two functions can be modified quite easily to take a different base number and conversion rule so that other languages can be supported.

TeXhackers note: These are more or less the internal LaTeX2 functions `\@arabic`, `\@alph`, `\Alph`, `\@roman`, `\@Roman`, and `\@fnsymbol` except that `\int_to_symbol:n` is also allowed outside math mode.

38.1 Internal functions

\int_to_roman:w ⋆

`\int_to_roman:w` ⟨*integer*⟩ ⟨*space*⟩ `or` ⟨*non-expandable token*⟩

Converts ⟨*integer*⟩ to it lowercase roman representation. Note that it produces a string of letters with catcode 12.

TEXhackers note: This is the TEX primitive `\romannumeral` renamed.

`\int_to_number:w ⋆` `\int_to_number:w` ⟨*integer*⟩ ⟨*space*⟩

Converts ⟨*integer*⟩ to its numerical string. Note that it produces a string of letters with catcode 12.

TEXhackers note: This is the TEX primitive `\number` renamed.

`\int_roman_lcuc_mapping:Nnn`
`\int_to_roman_lcuc:NN`

`\int_roman_lcuc_mapping:Nnn` ⟨*roman_char*⟩ {⟨*licr*⟩} {⟨*LICR*⟩}
`\int_to_roman_lcuc:NN` ⟨*roman_char*⟩ ⟨*char*⟩

`\int_roman_lcuc_mapping:Nnn` specifies how the roman numeral ⟨*roman_ char*⟩ (i, v, x, l, c, d, or m) should be interpreted when converting the number. ⟨*licr*⟩ is the lower case and ⟨*LICR*⟩ is the uppercase mapping. `\int_to_roman_lcuc:NN` is a recursive function converting the roman numerals.

`\int_convert_number_with_rule:nnN`
`\int_alph_default_conversion_rule:n`
`\int_Alph_default_conversion_rule:n`
`\int_symbol_math_conversion_rule:n`
`\int_symbol_text_conversion_rule:n`

`\int_convert_number_with_rule:nnN` {⟨int_1⟩} {⟨int_2⟩} ⟨*function*⟩
`\int_alph_default_conversion_rule:n` {⟨*int*⟩}

`\int_convert_number_with_rule:nnN` converts ⟨int_1⟩ into letters, symbols, whatever as defined by ⟨*function*⟩. ⟨int_2⟩ denotes the base number for the conversion.

39 Variable and constants

`\int_const:Nn` `\int_const:Nn` `\c_`⟨*value*⟩ {⟨*value*⟩}

Defines an integer constant of a certain ⟨*value*⟩. If the constant is negative or very large it internally uses an ⟨*int*⟩ register.

```
\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_hundred_one
\c_twohundred_fifty_five
\c_twohundred_fifty_six
\c_thousand
\c_ten_thousand
\c_ten_thousand_one
\c_ten_thousand_two
\c_ten_thousand_three
\c_ten_thousand_four
\c_twenty_thousand
```

Set of constants denoting useful values.

TEXhackers note: Some of these constants have been available under LATEX2 under names like `\m@ne`, `\z@`, `\@ne`,`\tw@`, `\thr@@`, etc.

`\c_max_int` Constant that denote the maximum value which can be stored in an ⟨*int*⟩ register.

`\c_max_register_int` Maximum number of registers.

```
\l_tmpa_int
\l_tmpb_int
\l_tmpc_int
\g_tmpa_int
\g_tmpb_int
```

Scratch register for immediate use. They are not used by conditionals or predicate functions.

40 Conversion

`\int_convert_from_base_ten:nn` `\int_convert_from_base_ten:nn {⟨number⟩} {⟨base⟩}`

Converts the base 10 number ⟨*number*⟩ into its equivalent representation written in base ⟨*base*⟩. Expandable.

`\int_convert_to_base_ten:nn` `\int_convert_to_base_ten:nn {⟨number⟩} {⟨base⟩}`

Converts the base ⟨*base*⟩ number ⟨*number*⟩ into its equivalent representation written in base 10. ⟨*number*⟩ can consist of digits and ascii letters. Expandable.

Part IX

The l3intexpr package Integer expressions

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* ('`int expr`').

41 Calculating and comparing integers

`\intexpr_eval:n ⋆` `\intexpr_eval:n {⟨int expr⟩}`

Evaluates an ⟨*integer expression*⟩, expanding to a properly terminated ⟨*number*⟩ that can be used in any situation that demands one, or which can be typeset. For example,

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

evaluates to −6. Two expansions are necessary to convert the ⟨*expression*⟩ into the ⟨*number*⟩ it represents. Full expansion to the ⟨*number*⟩ can be carried out using an `f` expansion in an expandable context or a `x` expansion in other cases.

`\intexpr_compare_p:n ⋆`
`\intexpr_compare:nTF ⋆`

`\intexpr_compare_p:n {⟨⟨int expr₁⟩ ⟨rel⟩ ⟨int expr₂⟩⟩}`
`\intexpr_compare:nTF {⟨⟨int expr₁⟩ ⟨rel⟩ ⟨int expr₂⟩⟩}`
⟨*true code*⟩ ⟨*false code*⟩

Evaluates ⟨*integer expression*$_1$⟩ and ⟨*integer expression*$_2$⟩ as described for `\intexpr_eval:n`, and then carries out a comparison of the resulting integers using C-like operators:

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

Based on the result of the comparison either the ⟨*true code*⟩ or ⟨*false code*⟩ is executed. Both integer expressions are evaluated fully in the process. Note the syntax, which allows natural input in the style of

```
\intexpr_compare_p:n {5+3 != \l_tmpb_int}
```

`=` is available as comparator (in addition to those familiar to C users) as standard TeX practice is to compare values using a single `=`.

`\intexpr_compare_p:nNn` ⋆
`\intexpr_compare:nNnTF` ⋆

`\intexpr_compare_p:nNn {`⟨*int expr*$_1$⟩`}` ⟨*rel*⟩ `{`⟨*int expr*$_2$⟩`}`

Evaluates ⟨*integer expression*$_1$⟩ and ⟨*integer expression*$_2$⟩ as described for `\intexpr_eval:n`, then compares the two results using one of the relations `=`, `<` or `>`. These functions are faster than the `n` variants described above but do not support an extended set of relational operators.

`\intexpr_max:nn` ⋆
`\intexpr_min:nn` ⋆

`\intexpr_max:nn {`⟨*int expr*$_1$⟩`} {`⟨*int expr*$_2$⟩`}`

Evaluates ⟨*integer expression*$_1$⟩ and ⟨*integer expression*$_2$⟩ as described for `\intexpr_eval:n`, expanding to the larger or smaller of the two resulting ⟨*numbers*⟩ (for `max` and `min`, respectively).

`\intexpr_abs:n` ⋆

`\intexpr_abs:n {`⟨*int expr*⟩`}`

Evaluates ⟨*integer expression*⟩ as described for `\intexpr_eval:n` and expands to the absolute value of the resulting ⟨*number*⟩.

`\intexpr_if_odd:nTF` ⋆
`\intexpr_if_odd_p:n` ⋆
`\intexpr_if_even:nTF` ⋆
`\intexpr_if_even_p:n` ⋆

`\intexpr_if_odd:nTF {`⟨*int expr*⟩`} {`⟨*true*⟩`} {`⟨*false*⟩`}`

Evaluates ⟨*integer expression*⟩ as described for `\intexpr_eval:n` and execute ⟨*true code*⟩ or ⟨*false code*⟩ depending on whether the resulting ⟨*number*⟩ is odd or even.

`\intexpr_div_truncate:nn` ⋆
`\intexpr_div_round:nn` ⋆
`\intexpr_mod:nn` ⋆

`\intexpr_div_truncate:nn` {⟨*int expr*$_1$⟩} {⟨*int expr*$_2$⟩}
`\intexpr_mod:nn` {⟨*int expr*$_1$⟩} {⟨*int expr*$_2$⟩}

Evaluates ⟨*integer expression*$_1$⟩ and ⟨*integer expression*$_2$⟩ as described for `\intexpr_eval:n`, expanding to the appropriate result of division of the resulting ⟨*numbers*⟩. The `truncate` function expands to the integer part of the division with the decimal simply discarded, whereas `round` will use the decimal part to round the integer up if appropriate. The `mod` function expands to the integer remainder of the division.

42 Primitive (internal) functions

`\if_num:w` ⋆
`\if_inexpr_compare:w` ⋆

`\if_num:w` ⟨*number*$_1$⟩ ⟨*rel*⟩ ⟨*number*$_2$⟩ ⟨*true*⟩ `\else:` ⟨*false*⟩ `\fi:`

Compare two integers using ⟨*rel*⟩, which must be one of `=`, `<` or `>` with category code 12. The `\else:` branch is optional.

TEXhackers note: These are both names for the TEX primitive `\ifnum`.

`\if_intexpr_case:w` ⋆
`\if_case:w` ⋆
`\or:` ⋆

`\if_case:w` ⟨*number*⟩ ⟨*case*$_0$⟩ `\or:` ⟨*case*$_1$⟩ `\or: ... \else:` ⟨*default*⟩ `\fi:`

Selects a case to execute based on the value of ⟨*number*⟩. The first case (⟨*case*$_0$⟩) is executed if ⟨*number*⟩ is 0, the second (⟨*case*$_1$⟩) if the ⟨*number*⟩ is 1, *etc.* The ⟨*number*⟩ may be a literal, a constant or an integer expression (*e.g.* using `\intexpr_eval:n`).

TEXhackers note: These are the TEX primitives `\ifcase` (with two different names depending on context) and `\or`.

`\intexpr_value:w` ⋆

`\intexpr_value:w` ⟨*integer*⟩
`\intexpr_value:w` ⟨*tokens*⟩ ⟨*optional space*⟩

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

TEXhackers note: This is the TEX primitive `\number`.

`\intexpr_eval:w` ⋆
`\intexpr_eval_end:`

`\intexpr_eval:w` ⟨*int expr*⟩ `\intexpr_eval_end:`

Evaluates ⟨*integer expression*⟩ as described for `\intexpr_eval:n`. The evalution stops when an unexpandable token with category code other than 12 is read or when `\intexpr_eval_end:` is reached. The latter is gobbled by the scanner mechanism: `\intexpr_eval_end:` itself is unexpandable but used correctly the entire construct is expandable.

TEXhackers note: This is the ε-TEX primitive `\numexpr`.

`\if_intexpr_odd:w ⋆`

`\if_intexpr_odd:w` ⟨*tokens*⟩ ⟨*true*⟩ `\else:` ⟨*false*⟩ `\fi:`
`\if_intexpr_odd:w` ⟨*number*⟩ ⟨*true*⟩ `\else:` ⟨*false*⟩ `\fi:`

Expands ⟨*tokens*⟩ until a non-numeric tokens is found, and tests whether the resulting ⟨*number*⟩ is odd. If so, ⟨*true code*⟩ is executed. The `\else:` branch is optional.

TEXhackers note: This is the TEX primitive `\ifodd`.

`\intexpr_while_do:nn ⋆`
`\intexpr_until_do:nn ⋆`
`\intexpr_do_while:nn ⋆`
`\intexpr_do_until:nn ⋆`

`\intexpr_while_do:nn {`⟨⟨*int expr*$_1$⟩ ⟨*rel*⟩ ⟨*int expr*$_2$⟩⟩`} {`⟨*code*⟩`}`

In the case of the `while_do` version, the integer comparison is evaluated as described for `\intexpr_compare_p:n`, and if `true` execute the ⟨*code*⟩. The test and code then alternate until the result is ⟨*false*⟩. The `do_while` alternative first executes the ⟨*code*⟩ and then evaluates the integer comparison. In the `until` cases, the ⟨*code*⟩ is executed if the test is `false`: the loop is ended when the relation is `true`.

`\intexpr_while_do:nNnn ⋆`
`\intexpr_until_do:nNnn ⋆`
`\intexpr_do_while:nNnn ⋆`
`\intexpr_do_until:nNnn ⋆`

`\intexpr_while_do:nNnn` ⟨*int expr*⟩ ⟨*rel*⟩ ⟨*int expr*⟩ `{`⟨*code*⟩`}`

These behave in the same manner as the preceding loops but use the relation logic described for `\intexpr_compare_p:nNn`.

Part X

The l3skip package

Dimension and skip registers

LaTeX3 knows about two types of length registers for internal use: rubber lengths (`skip`s) and rigid lengths (`dim`s).

43 Skip registers

43.1 Functions

\skip_new:N
\skip_new:c
\skip_new_local:N
\skip_new_local:c

\skip_new:N ⟨*skip*⟩

Defines ⟨*skip*⟩ to be a new variable of type `skip`. With the `local` variant, the variable is only available within the current group level (or below).

TeXhackers note: `\skip_new:N` is the equivalent to plain TeX's `\newskip`.

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

\skip_zero:N ⟨*skip*⟩

Locally or globally reset ⟨*skip*⟩ to zero. For global variables the global versions should be used.

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

\skip_set:Nn ⟨*skip*⟩ {⟨*skip value*⟩}

These functions will set the ⟨*skip*⟩ register to the ⟨*length*⟩ value.

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

\skip_add:Nn ⟨*skip*⟩ {⟨*length*⟩}

These functions will add to the ⟨*skip*⟩ register the value ⟨*length*⟩. If the second argument is a ⟨*skip*⟩ register too, the surrounding braces can be left out.

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

`\skip_gsub:Nn` ⟨*skip*⟩ `{`⟨*length*⟩`}`

These functions will subtract from the ⟨*skip*⟩ register the value ⟨*length*⟩. If the second argument is a ⟨*skip*⟩ register too, the surrounding braces can be left out.

`\skip_use:N`
`\skip_use:c`

`\skip_use:N` ⟨*skip*⟩

This function returns the length value kept in ⟨*skip*⟩ in a way suitable for further processing.

TEXhackers note: The function `\skip_use:N` could be implemented directly as the TEX primitive `\tex_the:D` which is also responsible to produce the values for other internal quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explanatory.

`\skip_show:N`
`\skip_show:c`

`\skip_show:N` ⟨*skip*⟩

This function pauses the compilation and displays the length value kept in ⟨*skip*⟩ in the console output and log file.

TEXhackers note: The function `\skip_show:N` could be implemented directly as the TEX primitive `\tex_showthe:D` which is also responsible to produce the values for other internal quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explaining.

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

`\skip_horizontal:N` ⟨*skip*⟩
`\skip_horizontal:n` `{`⟨*length*⟩`}`

The `hor` functions insert ⟨*skip*⟩ or ⟨*length*⟩ with the TEX primitive `\hskip`. The `vertical` variants do the same with `\vskip`. The `n` versions evaluate ⟨*length*⟩ with `\skip_eval:n`.

`\skip_if_infinite_glue_p:n`
`\skip_if_infinite_glue:nTF`

`\skip_if_infinite_glue:nTF` `{`⟨*skip*⟩`}` `{`⟨*true*⟩`}` `{`⟨*false*⟩`}`

Checks if ⟨*skip*⟩ contains infinite stretch or shrink components and executes either ⟨*true*⟩ or ⟨*false*⟩. Also works on input like `3pt plus .5in`.

`\skip_split_finite_else_action:nnNN` `\skip_split_finite_else_action:nnNN` {⟨*skip*⟩} {⟨*action*⟩} ⟨$dimen_1$⟩ ⟨$dimen_2$⟩

Checks if ⟨*skip*⟩ contains finite glue. If it does then it assigns ⟨$dimen_1$⟩ the stretch component and ⟨$dimen_2$⟩ the shrink component. If it contains infinite glue set ⟨$dimen_1$⟩ and ⟨$dimen_2$⟩ to zero and execute `#2` which is usually an error or warning message of some sort.

`\skip_eval:n ⋆` `\skip_eval:n` {⟨*skip expr*⟩}

Evaluates the value of ⟨*skip expr*⟩ so that `\skip_eval:n {5pt plus 3fil + 3pt minus 1fil}` puts `8.0pt plus 3.0fil minus 1.0fil` back into the input stream. Expandable.

TEXhackers note: This is the ε-TEX primitive `\glueexpr` turned into a function taking an argument.

43.2 Formatting a skip register value

43.3 Variable and constants

`\c_max_skip` Constant that denotes the maximum value which can be stored in a ⟨*skip*⟩ register.

`\c_zero_skip` Set of constants denoting useful values.

`\l_tmpa_skip`
`\l_tmpb_skip`
`\l_tmpc_skip`
`\g_tmpa_skip`
`\g_tmpb_skip` Scratch register for immediate use.

44 Dim registers

44.1 Functions

`\dim_new:N`
`\dim_new:c`
`\dim_new_local:N`
`\dim_new_local:c` `\dim_new:N` ⟨*dim*⟩

Defines ⟨*dim*⟩ to be a new variable of type `dim`. With the `local` variant, the variable is only available within the current group level (or below).

TEXhackers note: `\dim_new:N` is the equivalent to plain TEX's `\newdimen`.

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

`\dim_zero:N` ⟨*dim*⟩

Locally or globally reset ⟨*dim*⟩ to zero. For global variables the global versions should be used.

`\dim_set:Nn`
`\dim_set:Nc`
`\dim_set:cn`
`\dim_gset:Nn`
`\dim_gset:Nc`
`\dim_gset:cn`
`\dim_gset:cc`

`\dim_set:Nn` ⟨*dim*⟩ `{`⟨*dim value*⟩`}`

These functions will set the ⟨*dim*⟩ register to the ⟨*dim value*⟩ value.

`\dim_add:Nn`
`\dim_add:Nc`
`\dim_add:cn`
`\dim_gadd:Nn`
`\dim_gadd:cn`

`\dim_add:Nn` ⟨*dim*⟩ `{`⟨*length*⟩`}`

These functions will add to the ⟨*dim*⟩ register the value ⟨*length*⟩. If the second argument is a ⟨*dim*⟩ register too, the surrounding braces can be left out.

`\dim_sub:Nn`
`\dim_sub:Nc`
`\dim_sub:cn`
`\dim_gsub:Nn`
`\dim_gsub:cn`

`\dim_gsub:Nn` ⟨*dim*⟩ `{`⟨*length*⟩`}`

These functions will subtract from the ⟨*dim*⟩ register the value ⟨*length*⟩. If the second argument is a ⟨*dim*⟩ register too, the surrounding braces can be left out.

`\dim_use:N`
`\dim_use:c`

`\dim_use:N` ⟨*dim*⟩

This function returns the length value kept in ⟨*dim*⟩ in a way suitable for further processing.

TEXhackers note: The function `\dim_use:N` could be implemented directly as the TEX primitive `\tex_the:D` which is also responsible to produce the values for other internal quantities.

We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explanatory.

\dim_show:N
\dim_show:c

\dim_show:N ⟨*skip*⟩

This function pauses the compilation and displays the length value kept in ⟨*skip*⟩ in the console output and log file.

TEXhackers note: The function \dim_show:N could be implemented directly as the TEX primitive \tex_showthe:D which is also responsible to produce the values for other internal quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explaining.

\dim_eval:n

\dim_eval:n {⟨*dim expr*⟩}

Evaluates the value of a dimension expression so that \dim_eval:n {5pt+3pt} puts 8pt back into the input stream. Expandable.

TEXhackers note: This is the ε-TEX primitive \dimexpr turned into a function taking an argument.

\if_dim:w

\if_dim:w ⟨*dimen*$_1$⟩ ⟨*rel*⟩ ⟨*dimen*$_2$⟩ ⟨*true*⟩ \else: ⟨*false*⟩ \fi:

Compare two dimensions. It is recommended to use \dim_eval:n to correctly evaluate and terminate these numbers. ⟨*rel*⟩ is one of <, = or > with catcode 12.

TEXhackers note: This is the TEX primitive \ifdim.

\dim_compare:nNn*TF* ⋆
\dim_compare_p:nNn ⋆

\dim_compare:nNnTF {⟨*dim expr*⟩} ⟨*rel*⟩ {⟨*dim expr*⟩}
{⟨*true*⟩} {⟨*false*⟩}

These functions test two dimension expressions against each other. They are both evaluated by \dim_eval:n. Note that if both expressions are normal dimension variables as in

```
\dim_compare:nNnTF \l_temp_dim < \c_zero_skip {negative}{non-negative}
```

you can safely omit the braces.

TEXhackers note: This is the TEX primitive \ifdim turned into a function.

\dim_while_do:nNnn
\dim_until_do:nNnn
\dim_do_while:nNnn
\dim_do_until:nNnn

\dim_while_do:nNnn ⟨*dim expr*⟩ ⟨*rel*⟩ ⟨*dim expr*⟩ ⟨*code*⟩

\dim_while_do:nNnn tests the dimension expressions and if true performs ⟨*code*⟩ repeatedly while the test remains true. \dim_do_while:nNnn is similar but executes the body first and then performs the check, thus ensuring that the body is executed at least once. The 'until' versions are similar but continue the loop as long as the test is false.

44.2 Variable and constants

\c_max_dim

Constant that denotes the maximum value which can be stored in a ⟨*dim*⟩ register.

\c_zero_dim

Set of constants denoting useful values.

\l_tmpa_dim
\l_tmpb_dim
\l_tmpc_dim
\l_tmpd_dim
\g_tmpa_dim
\g_tmpb_dim

Scratch register for immediate use.

45 Muskips

\muskip_new:N
\muskip_new_local:N

\muskip_new:N ⟨*muskip*⟩

Defines ⟨*muskip*⟩ to be a new variable of type muskip. With the local variant, the variable is only available within the current group level (or below).

TEXhackers note: \muskip_new:N is the equivalent to plain TEX's \newmuskip.

\muskip_set:Nn
\muskip_gset:Nn

\muskip_set:Nn ⟨*muskip*⟩ {⟨*muskip value*⟩}

These functions will set the ⟨*muskip*⟩ register to the ⟨*length*⟩ value.

\muskip_add:Nn
\muskip_gadd:Nn

\muskip_add:Nn ⟨*muskip*⟩ {⟨*length*⟩}

These functions will add to the ⟨*muskip*⟩ register the value ⟨*length*⟩. If the second argument is a ⟨*muskip*⟩ register too, the surrounding braces can be left out.

\muskip_sub:Nn
\muskip_gsub:Nn

\muskip_gsub:Nn ⟨*muskip*⟩ {⟨*length*⟩}

These functions will subtract from the ⟨*muskip*⟩ register the value ⟨*length*⟩. If the second argument is a ⟨*muskip*⟩ register too, the surrounding braces can be left out.

\muskip_use:N

\muskip_use:N ⟨*muskip*⟩

This function returns the length value kept in ⟨*muskip*⟩ in a way suitable for further processing.

TEXhackers note: See note for \dim_use:N.

Part XI

The l3tl package Token Lists

LaTeX3 stores token lists in variables also called 'token lists'. Variables of this type get the suffix tl and functions of this type have the prefix tl. To use a token list variable you simply call the corresponding variable.

Often you find yourself with not a token list variable but an arbitrary token list which has to undergo certain tests. We will *also* prefix these functions with tl. While token list variables are always single tokens, token lists are always surrounded by braces.

46 Functions

\tl_new:N
\tl_new:c
\tl_new:Nn
\tl_new:cn
\tl_new:Nx

\tl_new:Nn ⟨*tl var.*⟩ {⟨*initial token list*⟩}

Defines ⟨*tl var.*⟩ to be a new variable to store a token list. ⟨*initial token list*⟩ is the initial value of ⟨*tl var.*⟩. This makes it possible to assign values to a constant token list variable.

The form `\tl_new:N` initializes the token list variable with an empty value.

`\tl_const:Nn`

`\tl_const:Nn` ⟨*tl var.*⟩ `{`⟨*token list*⟩`}`

Defines ⟨*tl var.*⟩ as a constant expanding to ⟨*token list*⟩. The name of the constant must be free when the constant is created.

`\tl_use:N`
`\tl_use:c`

`\tl_use:N` ⟨*tl var.*⟩

Function that inserts the ⟨*tl var.*⟩ into the processing stream. Instead of `\tl_use:N` simply placing the ⟨*tl var.*⟩ into the input stream is also supported. `\tl_use:c` will complain if the ⟨*tl var.*⟩ hasn't been declared previously!

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

`\tl_show:N` ⟨*tl var.*⟩
`\tl_show:n` `{`⟨*token list*⟩`}`

Function that pauses the compilation and displays the ⟨*tl var.*⟩ or ⟨*token list*⟩ on the console output and in the log file.

`\tl_set:Nn`
`\tl_set:Nc`
`\tl_set:NV`
`\tl_set:No`
`\tl_set:Nv`
`\tl_set:Nf`
`\tl_set:Nx`
`\tl_set:cn`
`\tl_set:co`
`\tl_set:cV`
`\tl_set:cx`
`\tl_gset:Nn`
`\tl_gset:Nc`
`\tl_gset:No`
`\tl_gset:NV`
`\tl_gset:Nv`
`\tl_gset:Nx`
`\tl_gset:cn`
`\tl_gset:cx`

`\tl_set:Nn` ⟨*tl var.*⟩ `{`⟨*token list*⟩`}`

Defines ⟨*tl var.*⟩ to hold the token list ⟨*token list*⟩. Global variants of this command assign the value globally the other variants expand the ⟨*token list*⟩ up to a certain level before the assignment or interpret the ⟨*token list*⟩ as a character list and form a control sequence out of it.

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

`\tl_clear:N` ⟨*tl var.*⟩

The ⟨*tl var.*⟩ is locally or globally cleared. The `c` variants will generate a control sequence name which is then interpreted as ⟨*tl var.*⟩ before clearing.

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

`\tl_clear_new:N` ⟨*tl var.*⟩

These functions check if ⟨*tl var.*⟩ exists. If it does it will be cleared; if it doesn't it will be allocated.

`\tl_put_left:Nn`
`\tl_put_left:NV`
`\tl_put_left:No`
`\tl_put_left:Nx`
`\tl_put_left:cn`
`\tl_put_left:cV`
`\tl_put_left:co`

`\tl_put_left:Nn` ⟨*tl var.*⟩ `{`⟨*token list*⟩`}`

These functions will append ⟨*token list*⟩ to the left of ⟨*tl var.*⟩. ⟨*token list*⟩ might be subject to expansion before assignment.

`\tl_put_right:Nn`
`\tl_put_right:NV`
`\tl_put_right:No`
`\tl_put_right:Nx`
`\tl_put_right:cn`
`\tl_put_right:cV`
`\tl_put_right:co`

`\tl_put_right:Nn` ⟨*tl var.*⟩ `{`⟨*token list*⟩`}`

These functions append ⟨*token list*⟩ to the right of ⟨*tl var.*⟩.

`\tl_gput_left:Nn`
`\tl_gput_left:No`
`\tl_gput_left:NV`
`\tl_gput_left:Nx`
`\tl_gput_left:cn`
`\tl_gput_left:co`
`\tl_gput_left:cV`

`\tl_gput_left:Nn` ⟨*tl var.*⟩ `{`⟨*token list*⟩`}`

These functions will append ⟨*token list*⟩ globally to the left of ⟨*tl var.*⟩.

```
\tl_gput_right:Nn
\tl_gput_right:No
\tl_gput_right:NV
\tl_gput_right:Nx
\tl_gput_right:cn
\tl_gput_right:co
\tl_gput_right:cV
```

`\tl_gput_right:Nn` ⟨*tl var.*⟩ `{`⟨*token list*⟩`}`

These functions will globally append ⟨*token list*⟩ to the right of ⟨*tl var.*⟩.

A word of warning is appropriate here: Token list variables are implemented as macros and as such currently inherit some of the peculiarities of how TEX handles #s in the argument of macros. In particular, the following actions are legal

```
\tl_set:Nn \l_tmpa_tl{##1}
\tl_put_right:Nn \l_tmpa_tl{##2}
\tl_set:No \l_tmpb_tl{\l_tmpa_tl ##3}
```

`x` type expansions where macros being expanded contain #s do not work and will not work until there is an `\expanded` primitive in the engine. If you want them to work you must double #s another level.

```
\tl_set_eq:NN
\tl_set_eq:Nc
\tl_set_eq:cN
\tl_set_eq:cc
\tl_gset_eq:NN
\tl_gset_eq:Nc
\tl_gset_eq:cN
\tl_gset_eq:cc
```

`\tl_set_eq:NN` ⟨*tl var.* $_1$⟩ ⟨*tl var.* $_2$⟩

Fast form for `\tl_set:No` ⟨*tl var.* $_1$⟩ `{`⟨*tl var.* $_2$⟩`}`
when ⟨*tl var.* $_2$⟩ is known to be a variable of type `tl`.

```
\tl_to_str:N
\tl_to_str:c
```

`\tl_to_str:N` ⟨*tl var.*⟩

This function returns the token list kept in ⟨*tl var.*⟩ as a string list with all characters catcoded to ‘other’.

```
\tl_to_str:n
```

`\tl_to_str:n` `{`⟨*token list*⟩`}`

This function turns its argument into a string where all characters have catcode ‘other’.

TEXhackers note: This is the ε-TEX primitive `\detokenize`.

`\tl_rescan:nn` `\tl_rescan:nn {`⟨*catcode setup*⟩`} {`⟨*token list*⟩`}`

Returns the result of re-tokenising ⟨*token list*⟩ with the catcode setup (and whatever other redefinitions) specified. This is useful because the catcodes of characters are 'frozen' when first tokenised; this allows their meaning to be changed even after they've been read as an argument. Also see `\tl_set_rescan:Nnn` below.

TEXhackers note: This is a wrapper around ε-TEX's `\scantokens`.

`\tl_set_rescan:Nnn`
`\tl_set_rescan:Nnx`
`\tl_gset_rescan:Nnn`
`\tl_gset_rescan:Nnx` `\tl_set_rescan:Nnn` ⟨*tl var.*⟩ `{`⟨*catcode setup*⟩`} {`⟨*token list*⟩`}`

Sets ⟨*tl var.*⟩ to the result of re-tokenising ⟨*token list*⟩ with the catcode setup (and whatever other redefinitions) specified.

TEXhackers note: This is a wrapper around ε-TEX's `\scantokens`.

47 Predicates and conditionals

`\tl_if_empty_p:N` ⋆
`\tl_if_empty_p:c` ⋆ `\tl_if_empty_p:N` ⟨*tl var.*⟩

This predicate returns 'true' if ⟨*tl var.*⟩ is 'empty' i.e., doesn't contain any tokens.

`\tl_if_empty:N`*TF* ⋆
`\tl_if_empty:c`*TF* ⋆ `\tl_if_empty:NTF` ⟨*tl var.*⟩ `{`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Execute ⟨*true code*⟩ if ⟨*tl var.*⟩ is empty and ⟨*false code*⟩ if it contains any tokens.

`\tl_if_eq_p:NN` ⋆
`\tl_if_eq_p:cN` ⋆
`\tl_if_eq_p:Nc` ⋆
`\tl_if_eq_p:cc` ⋆ `\tl_if_eq_p:NN` ⟨*tl var.* $_1$⟩ ⟨*tl var.* $_2$⟩

Predicate function which returns 'true' if the two token list variables are identical and 'false' otherwise.

`\tl_if_eq:NN`*TF* ⋆
`\tl_if_eq:cN`*TF* ⋆
`\tl_if_eq:Nc`*TF* ⋆
`\tl_if_eq:cc`*TF* ⋆ `\tl_if_eq:NNTF` ⟨*tl var.* $_1$⟩ ⟨*tl var.* $_2$⟩ `{`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Execute ⟨*true code*⟩ if ⟨*tl var.* $_1$⟩ holds the same token list as ⟨*tl var.* $_2$⟩ and ⟨*false code*⟩ otherwise.

\tl_if_eq:nnTF ⋆
\tl_if_eq:nVTF ⋆
\tl_if_eq:noTF ⋆
\tl_if_eq:VnTF ⋆
\tl_if_eq:onTF ⋆
\tl_if_eq:VVTF ⋆
\tl_if_eq:ooTF ⋆
\tl_if_eq:xxTF ⋆
\tl_if_eq:xnTF ⋆
\tl_if_eq:nxTF ⋆
\tl_if_eq:xVTF ⋆
\tl_if_eq:xoTF ⋆
\tl_if_eq:VxTF ⋆
\tl_if_eq:oxTF ⋆

`\tl_if_eq:nnTF {⟨tlist1⟩} {⟨tlist2⟩} {⟨true code⟩} {⟨false code⟩}`

Execute ⟨*true code*⟩ if the two token lists ⟨*tlist* $_1$⟩ and ⟨*tlist* $_2$⟩ are identical. These functions are expandable if a new enough version of pdfTEX is being used.

\tl_if_eq_p:nn ⋆
\tl_if_eq_p:nV ⋆
\tl_if_eq_p:no ⋆
\tl_if_eq_p:Vn ⋆
\tl_if_eq_p:on ⋆
\tl_if_eq_p:VV ⋆
\tl_if_eq_p:oo ⋆
\tl_if_eq_p:xx ⋆
\tl_if_eq_p:xn ⋆
\tl_if_eq_p:nx ⋆
\tl_if_eq_p:xV ⋆
\tl_if_eq_p:xo ⋆
\tl_if_eq_p:Vx ⋆
\tl_if_eq_p:ox ⋆

`\tl_if_eq_p:nn {⟨tlist1⟩} {⟨tlist2⟩}`

Predicates function which returns 'true' if the two token list are identical and 'false' otherwise. These are only defined if a new enough version of pdfTEX is in use.

\tl_if_empty_p:n ⋆
\tl_if_empty_p:V ⋆
\tl_if_empty_p:o ⋆
\tl_if_empty:nTF
\tl_if_empty:VTF
\tl_if_empty:oTF

`\tl_if_empty:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}`

Execute ⟨*true code*⟩ if ⟨*token list*⟩ doesn't contain any tokens and ⟨*false code*⟩ otherwise.

\tl_if_blank_p:n ⋆
\tl_if_blank:nTF ⋆
\tl_if_blank_p:V ⋆
\tl_if_blank_p:o ⋆
\tl_if_blank:VTF ⋆
\tl_if_blank:oTF ⋆

\tl_if_blank:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Execute ⟨*true code*⟩ if ⟨*token list*⟩ is blank meaning that it is either empty or contains only blank spaces.

\tl_if_single_p:n ⋆
\tl_if_single:nTF ⋆
\tl_if_single_p:N ⋆
\tl_if_single:NTF ⋆

\tl_if_single:NTF {⟨*tl var.*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
\tl_if_single:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Conditional returning true if the token list or the contents of the tl var. consists of a single token only.

Note that an input of 'space'[6] returns ⟨*true*⟩ from this function.

\tl_to_lowercase:n
\tl_to_uppercase:n

\tl_to_lowercase:n {⟨*token list*⟩}

`\tl_to_lowercase:n` converts all tokens in ⟨*token list*⟩ to their lower case representation. Similar for `\tl_to_uppercase:n`.

TEXhackers note: These are the TEX primitives `\lowercase` and `\uppercase` renamed.

48 Working with the contents of token lists

\tl_map_function:nN ⋆
\tl_map_function:NN
\tl_map_function:cN

\tl_map_function:nN {⟨*token list*⟩} ⟨*function*⟩
\tl_map_function:NN ⟨*tl var.*⟩ ⟨*function*⟩

Runs through all elements in a ⟨*token list*⟩ from left to right and places ⟨*function*⟩ in front of each element. As this function will also pick up elements in brace groups, the element is returned with braces and hence ⟨*function*⟩ should be a function with a `:n` suffix even though it may very well only deal with a single token.

This function uses a purely expandable loop function and will stay so as long as ⟨*function*⟩ is expandable too.

[6] But remember any number of consequtive spaces are read as a single space by TEX.

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

`\tl_map_inline:nn {⟨token list⟩} {⟨inline function⟩}`
`\tl_map_inline:Nn ⟨tl var.⟩ {⟨inline function⟩}`

Allows a syntax like `\tl_map_inline:nn {⟨token list⟩} {\token_to_str:N ##1}`. This renders it non-expandable though. Remember to double the #s for each level.

```
\tl_map_variable:nNn
\tl_map_variable:NNn
\tl_map_variable:cNn
```

`\tl_map_variable:nNn {⟨token list⟩} ⟨temp⟩ {⟨action⟩}`
`\tl_map_variable:NNn ⟨tl var.⟩ ⟨temp⟩ {⟨action⟩}`

Assigns ⟨*temp*⟩ to each element on ⟨*token list*⟩ and executes ⟨*action*⟩. As there is an assignment in this process it is not expandable.

TEXhackers note: This is the LaTeX2 function `\@tfor` but with a more sane syntax. Also it works by tail recursion and so is faster as lists grow longer.

```
\tl_map_break:
```

`\tl_map_break:`

For breaking out of a loop. Must not be nested inside a primitive `\if` structure.

```
\tl_reverse:n
\tl_reverse:V
\tl_reverse:o
\tl_reverse:N
```

`\tl_reverse:n {`$\langle token_1\rangle\langle token_2\rangle\ldots\langle token_n\rangle$`}`
`\tl_reverse:N ⟨tl var.⟩`

Reverse the token list (or the token list in the ⟨*tl var.*⟩) to result in $\langle token_n\rangle\ldots\langle token_2\rangle\langle token_1\rangle$. Note that spaces in this token list are gobbled in the process.

Note also that braces are lost in the process of reversing a ⟨*tl var.*⟩. That is,
`\tl_set:Nn \l_tmpa_tl {a{bcd}e} \tl_reverse:N \l_tmpa_tl`
will result in `ebcda`. This behaviour is probably more of a bug than a feature.

```
\tl_elt_count:n ⋆
\tl_elt_count:V ⋆
\tl_elt_count:o ⋆
\tl_elt_count:N ⋆
```

`\tl_elt_count:n {⟨token list⟩}`
`\tl_elt_count:N ⟨tl var.⟩`

Returns the number of elements in the token list. Brace groups encountered count as one element. Note that spaces in this token list are gobbled in the process.

49 Variables and constants

```
\c_job_name_tl
```

Constant that gets the 'job name' assigned when TEX starts.

TEXhackers note: This is the new name for the primitive `\jobname`. It is a constant that is set by TEX and should not be overwritten by the package.

`\c_empty_tl` Constant that is always empty.

TEXhackers note: This was named `\@empty` in LATEX2 and `\empty` in plain TEX.

`\c_space_tl` A space token contained in a token list (compare this with `\char_space_token`). For use where an explicit space is required.

`\l_tmpa_tl`
`\l_tmpb_tl`
`\g_tmpa_tl`
`\g_tmpb_tl` Scratch register for immediate use. They are not used by conditionals or predicate functions. However, it is important to note that you should never rely on such scratch variables unless you fully control the code used between setting them and retrieving their value. Calling code from other modules, or worse allowing arbitrary user input to interfere might result in them not containing what you expect. In that is the case you better define your own scratch variables that are tight to your code by giving them suitable names.

`\l_tl_replace_tl` Internal register used in the replace functions.

`\l_kernel_testa_tl`
`\l_kernel_testb_tl` Registers used for conditional processing if the engine doesn't support arbitrary string comparison. Not for use outside the kernel code!

`\l_kernel_tmpa_tl`
`\l_kernel_tmpb_tl` Scratch registers reserved for other places in kernel code. Not for use outside the kernel code!

`\g_tl_inline_level_int` Internal register used in the inline map functions.

50 Searching for and replacing tokens

`\tl_if_in:NnTF`
`\tl_if_in:cnTF`
`\tl_if_in:nnTF`
`\tl_if_in:VnTF`
`\tl_if_in:onTF` `\tl_if_in:NnTF` ⟨*tl var.*⟩ {⟨*item*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Function that tests if ⟨*item*⟩ is in ⟨*tl var.*⟩. Depending on the result either ⟨*true code*⟩ or ⟨*false code*⟩ is executed. Note that ⟨*item*⟩ cannot contain brace groups nor `#`$_6$ tokens.

`\tl_replace_in:Nnn`
`\tl_replace_in:cnn`
`\tl_greplace_in:Nnn`
`\tl_greplace_in:cnn`

`\tl_replace_in:Nnn` ⟨*tl var.*⟩ `{`⟨*item*$_1$⟩`}` `{`⟨*item*$_2$⟩`}`

Replaces the leftmost occurrence of ⟨*item*$_1$⟩ in ⟨*tl var.*⟩ with ⟨*item*$_2$⟩ if present, otherwise the ⟨*tl var.*⟩ is left untouched. Note that ⟨*item*$_1$⟩ cannot contain brace groups nor `#`$_6$ tokens, and ⟨*item*$_2$⟩ cannot contain `#`$_6$ tokens.

`\tl_replace_all_in:Nnn`
`\tl_replace_all_in:cnn`
`\tl_greplace_all_in:Nnn`
`\tl_greplace_all_in:cnn`

`\tl_replace_all_in:Nnn` ⟨*tl var.*⟩ `{`⟨*item*$_1$⟩`}` `{`⟨*item*$_2$⟩`}`

Replaces *all* occurrences of ⟨*item*$_1$⟩ in ⟨*tl var.*⟩ with ⟨*item*$_2$⟩. Note that ⟨*item*$_1$⟩ cannot contain brace groups nor `#`$_6$ tokens, and ⟨*item*$_2$⟩ cannot contain `#`$_6$ tokens.

`\tl_remove_in:Nn`
`\tl_remove_in:cn`
`\tl_gremove_in:Nn`
`\tl_gremove_in:cn`

`\tl_remove_in:Nn` ⟨*tl var.*⟩ `{`⟨*item*⟩`}`

Removes the leftmost occurrence of ⟨*item*⟩ from ⟨*tl var.*⟩ if present. Note that ⟨*item*⟩ cannot contain brace groups nor `#`$_6$ tokens.

`\tl_remove_all_in:Nn`
`\tl_remove_all_in:cn`
`\tl_gremove_all_in:Nn`
`\tl_gremove_all_in:cn`

`\tl_remove_all_in:Nn` ⟨*tl var.*⟩ `{`⟨*item*⟩`}`

Removes *all* occurrences of ⟨*item*⟩ from ⟨*tl var.*⟩. Note that ⟨*item*⟩ cannot contain brace groups nor `#`$_6$ tokens.

51 Heads or tails?

Here are some functions for grabbing either the head or tail of a list and perform some tests on it.

```
\tl_head:n        ⋆
\tl_head:V        ⋆
\tl_tail:n        ⋆
\tl_tail:V        ⋆
\tl_tail:f        ⋆
\tl_head_i:n      ⋆
\tl_head_iii:n    ⋆
\tl_head_iii:f    ⋆
\tl_head:w        ⋆
\tl_tail:w        ⋆
\tl_head_i:w      ⋆
\tl_head_iii:w    ⋆
```

`\tl_head:n` { ⟨*token*$_1$⟩⟨*token*$_2$⟩...⟨*token*$_n$⟩ }
`\tl_tail:n` { ⟨*token*$_1$⟩⟨*token*$_2$⟩...⟨*token*$_n$⟩ }
`\tl_head:w` ⟨*token*$_1$⟩⟨*token*$_2$⟩...⟨*token*$_n$⟩ `\q_nil`

These functions return either the head or the tail of a list, thus in the above example `\tl_head:n` would return ⟨*token*$_1$⟩ and `\tl_tail:n` would return ⟨*token*$_2$⟩...⟨*token*$_n$⟩. `\tl_head_iii:n` returns the first three tokens. The `:w` versions require some care as they use a delimited argument internally.

TEXhackers note: These are the Lisp functions `car` and `cdr` but with LaTeX3 names.

```
\tl_if_head_eq_meaning_p:nN   ⋆
\tl_if_head_eq_meaning:nNTF   ⋆
```

`\tl_if_head_eq_meaning:nNTF` {⟨*token list*⟩} ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Returns ⟨*true*⟩ if the first token in ⟨*token list*⟩ is equal to ⟨*token*⟩ and ⟨*false*⟩ otherwise. The `meaning` version compares the two tokens with `\if_meaning:w`.

```
\tl_if_head_eq_charcode_p:nN  ⋆
\tl_if_head_eq_charcode_p:fN  ⋆
\tl_if_head_eq_charcode:nNTF  ⋆
\tl_if_head_eq_charcode:fNTF  ⋆
```

`\tl_if_head_eq_charcode:nNTF` {⟨*token list*⟩} ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Returns ⟨*true*⟩ if the first token in ⟨*token list*⟩ is equal to ⟨*token*⟩ and ⟨*false*⟩ otherwise. The `meaning` version compares the two tokens with `\if_charcode:w` but it prevents expansion of them. If you want them to expand, you can use an `f` type expansion first (define `\tl_if_head_eq_charcode:fNTF` or similar).

```
\tl_if_head_eq_catcode_p:nN   ⋆
\tl_if_head_eq_catcode:nNTF   ⋆
```

`\tl_if_head_eq_catcode:nNTF` {⟨*token list*⟩} ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Returns ⟨*true*⟩ if the first token in ⟨*token list*⟩ is equal to ⟨*token*⟩ and ⟨*false*⟩ otherwise. This version uses `\if_catcode:w` for the test but is otherwise identical to the `charcode` version.

Part XII

The l3toks package Token Registers

There is a second form beside token list variables in which LaTeX3 stores token lists, namely the internal TeX token registers. Functions dealing with these registers got the prefix `\toks_`. Unlike token list variables we have an accessing function as one can see below.

The main difference between ⟨*toks*⟩ (token registers) and ⟨*tl var.*⟩ (token list variable) is their behavior regarding expansion. While ⟨*tl vars*⟩ expand fully (i.e., until only unexpandable tokens are left) inside an argument that is subject to expansion (i.e., denoted by `x`) ⟨*toks*⟩'s expand always only up to one level, i.e., passing their contents without further expansion.

There are fewer restrictions on the contents of a token register over a token list variable. So while ⟨*token list*⟩ is used to describe the contents of both of these, bear in mind that slightly different lists of tokens are allowed in each case. The best (only?) example is that a ⟨*toks*⟩ can contain the `#` character (i.e., characters of catcode 6), whereas a ⟨*tl var.*⟩ will require its input to be sanitised before that is possible.

If you're not sure which to use between a ⟨*tl var.*⟩ or a ⟨*toks*⟩, consider what data you're trying to hold. If you're dealing with function parameters involving `#`, or building some sort of data structure then you probably want a ⟨*toks*⟩ (e.g., `l3prop` uses ⟨*toks*⟩ to store its property lists).

If you're storing ad-hoc data for later use (possibly from direct user input) then usually a ⟨*tl var.*⟩ will be what you want.

52 Allocation and use

`\toks_new:N`
`\toks_new:c`
`\toks_new_local:N`
`\toks_new_local:c`

`\toks_new:N` ⟨*toks*⟩

Defines ⟨*toks*⟩ to be a new token list register. With the `local` variant, the variable is only available within the current group level (or below).

TeXhackers note: This is the LaTeX3 allocation for what was called `\newtoks` in plain TeX.

`\toks_use:N`
`\toks_use:c`

`\toks_use:N` ⟨*toks*⟩

Accesses the contents of ⟨*toks*⟩. Contrary to token list variables ⟨*toks*⟩ can't be access simply by calling them directly.

TEXhackers note: Something like `\the` ⟨*toks*⟩.

`\toks_set:Nn`
`\toks_set:NV`
`\toks_set:Nv`
`\toks_set:No`
`\toks_set:Nx`
`\toks_set:Nf`
`\toks_set:cn`
`\toks_set:co`
`\toks_set:cV`
`\toks_set:cv`
`\toks_set:cx`
`\toks_set:cf`

`\toks_set:Nn` ⟨*toks*⟩ `{`⟨*token list*⟩`}`

Defines ⟨*toks*⟩ to hold the token list ⟨*token list*⟩.

TEXhackers note: `\toks_set:Nn` could have been specified in plain TEX by ⟨*toks*⟩ `= {`⟨*token list*⟩`}` but all other functions have no counterpart in plain TEX. Additionally the functions above the global variants described below will check for correct local and global assignments, something that isn't available in plain TEX.

`\toks_gset:Nn`
`\toks_gset:NV`
`\toks_gset:No`
`\toks_gset:Nx`
`\toks_gset:cn`
`\toks_gset:cV`
`\toks_gset:co`
`\toks_gset:cx`

`\toks_gset:Nn` ⟨*toks*⟩ `{`⟨*token list*⟩`}`

Defines ⟨*toks*⟩ to globally hold the token list ⟨*token list*⟩.

`\toks_set_eq:NN`
`\toks_set_eq:cN`
`\toks_set_eq:Nc`
`\toks_set_eq:cc`

`\toks_set_eq:NN` ⟨$toks_1$⟩ ⟨$toks_2$⟩

Set ⟨$toks_1$⟩ to the value of ⟨$toks_2$⟩. Don't try to use `\toks_set:Nn` for this purpose if the second argument is also a token register.

```
\toks_gset_eq:NN
\toks_gset_eq:cN
\toks_gset_eq:Nc
\toks_gset_eq:cc
```

`\toks_gset_eq:NN` ⟨*toks*$_1$⟩ ⟨*toks*$_2$⟩

The ⟨*toks*$_1$⟩ globally set to the value of ⟨*toks*$_2$⟩. Don't try to use `\toks_gset:Nn` for this purpose if the second argument is also a token register.

```
\toks_clear:N
\toks_clear:c
\toks_gclear:N
\toks_gclear:c
```

`\toks_clear:N` ⟨*toks*⟩

The ⟨*toks*⟩ is locally or globally cleared.

```
\toks_use_clear:N
\toks_use_clear:c
\toks_use_gclear:N
\toks_use_gclear:c
```

`\toks_use_clear:N` ⟨*toks*⟩

Accesses the contents of ⟨*toks*⟩ and clears (locally or globally) it afterwards. Actually the clearing operation is done in a way that does not prohibit the access of the following tokens in the input stream with functions stored in the token register. In other words this function is not exactly the same as calling `\toks_use:N` ⟨*toks*⟩ `\toks_clear:N` ⟨*toks*⟩ in sequence.

```
\toks_show:N
\toks_show:c
```

`\toks_show:N` ⟨*toks*⟩

Displays the contents of ⟨*toks*⟩ in the terminal output and log file. `#` signs in the ⟨*toks*⟩ will be shown doubled.

TEXhackers note: Something like `\showthe` ⟨*toks*⟩.

53 Adding to the contents of token registers

```
\toks_put_left:Nn
\toks_put_left:NV
\toks_put_left:No
\toks_put_left:Nx
\toks_put_left:cn
\toks_put_left:cV
\toks_put_left:co
```

`\toks_put_left:Nn` ⟨*toks*⟩ `{`⟨*token list*⟩`}`

These functions will append ⟨*token list*⟩ to the left of ⟨*toks*⟩. Assignment is done locally. If possible append to the right since this operation is faster.

\toks_gput_left:Nn
\toks_gput_left:NV
\toks_gput_left:No
\toks_gput_left:Nx
\toks_gput_left:cn
\toks_gput_left:cV
\toks_gput_left:co

\toks_gput_left:Nn ⟨*toks*⟩ {⟨*token list*⟩}

These functions will append ⟨*token list*⟩ to the left of ⟨*toks*⟩. Assignment is done globally. If possible append to the right since this operation is faster.

\toks_put_right:Nn
\toks_put_right:NV
\toks_put_right:No
\toks_put_right:Nx
\toks_put_right:cV
\toks_put_right:cn
\toks_put_right:co

\toks_put_right:Nn ⟨*toks*⟩ {⟨*token list*⟩}

These functions will append ⟨*token list*⟩ to the right of ⟨*toks*⟩. Assignment is done locally.

\toks_put_right:Nf

\toks_put_right:Nf ⟨*toks*⟩ {⟨*token list*⟩}

Variant of the above. `:Nf` is used by `template.dtx` and will perhaps be moved to that package.

\toks_gput_right:Nn
\toks_gput_right:NV
\toks_gput_right:No
\toks_gput_right:Nx
\toks_gput_right:cn
\toks_gput_right:cV
\toks_gput_right:co

\toks_gput_right:Nn ⟨*toks*⟩ {⟨*token list*⟩}

These functions will append ⟨*token list*⟩ to the right of ⟨*toks*⟩. Assignment is done globally.

54 Predicates and conditionals

\toks_if_empty_p:N ⋆
\toks_if_empty:N*TF* ⋆
\toks_if_empty_p:c ⋆
\toks_if_empty:c*TF* ⋆

\toks_if_empty:NTF ⟨*toks*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Expandable test for whether ⟨*toks*⟩ is empty.

```
\toks_if_eq:NNTF ⋆
\toks_if_eq:NcTF ⋆
\toks_if_eq:cNTF ⋆
\toks_if_eq:ccTF ⋆
\toks_if_eq_p:NN ⋆
\toks_if_eq_p:cN ⋆
\toks_if_eq_p:Nc ⋆
\toks_if_eq_p:cc ⋆
```

`\toks_if_eq:NNTF` ⟨*toks*$_1$⟩ ⟨*toks*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Expandably tests if ⟨*toks*$_1$⟩ and ⟨*toks*$_2$⟩ are equal.

55 Variable and constants

`\c_empty_toks` Constant that is always empty.

```
\l_tmpa_toks
\l_tmpb_toks
\l_tmpc_toks
\g_tmpa_toks
\g_tmpb_toks
\g_tmpc_toks
```

Scratch register for immediate use. They are not used by conditionals or predicate functions.

`\l_tl_replace_toks` A placeholder for contents of functions replacing contents of strings.

Part XIII

The l3seq package Sequences

LaTeX3 implements a data type called 'sequences'. These are special token lists that can be accessed via special function on the 'left'. Appending tokens is possible at both ends. Appended token lists can be accessed only as a union. The token lists that form the individual items of a sequence might contain any tokens except two internal functions that are used to structure sequences (see section internal functions below). It is also possible to map functions on such sequences so that they are executed for every item on the sequence.

All functions that return items from a sequence in some ⟨*tl var.*⟩ assume that the ⟨*tl var.*⟩ is local. See remarks below if you need a global returned value.

The defined functions are not orthogonal in the sense that every possible variation possible is actually available. If you need a new variant use the expansion functions described in the package `l3expan` to build it.

Adding items to the left of a sequence can currently be done with either something like `\seq_put_left:Nn` or with a "stack" function like `\seq_push:Nn` which has the same effect. Maybe one should therefore remove the "left" functions totally.

56 Functions for creating/initialising sequences

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

`\seq_new:N` ⟨*sequence*⟩

Defines ⟨*sequence*⟩ to be a variable of type `seq`.

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

`\seq_clear:N` ⟨*sequence*⟩

These functions locally or globally clear ⟨*sequence*⟩.

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

`\seq_clear_new:N` ⟨*sequence*⟩

These functions locally or globally clear ⟨*sequence*⟩ if it exists or otherwise allocates it.

`\seq_set_eq:NN`
`\seq_set_eq:cN`
`\seq_set_eq:Nc`
`\seq_set_eq:cc`

`\seq_set_eq:NN` ⟨seq_1⟩ ⟨seq_2⟩

Function that locally makes ⟨seq_1⟩ identical to ⟨seq_2⟩.

`\seq_gset_eq:NN`
`\seq_gset_eq:cN`
`\seq_gset_eq:Nc`
`\seq_gset_eq:cc`

`\seq_gset_eq:NN` ⟨seq_1⟩ ⟨seq_2⟩

Function that globally makes ⟨seq_1⟩ identical to ⟨seq_2⟩.

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

`\seq_gconcat:NNN` ⟨*seq*$_1$⟩ ⟨*seq*$_2$⟩ ⟨*seq*$_3$⟩

Function that conatenates ⟨*seq*$_2$⟩ and ⟨*seq*$_3$⟩ and globally assigns the result to ⟨*seq*$_1$⟩.

57 Adding data to sequences

```
\seq_put_left:Nn
\seq_put_left:NV
\seq_put_left:No
\seq_put_left:Nx
\seq_put_left:cn
\seq_put_left:cV
\seq_put_left:co
```

`\seq_put_left:Nn` ⟨*sequence*⟩ ⟨*token list*⟩

Locally appends ⟨*token list*⟩ as a single item to the left of ⟨*sequence*⟩. ⟨*token list*⟩ might get expanded before appending according to the variant.

```
\seq_put_right:Nn
\seq_put_right:NV
\seq_put_right:No
\seq_put_right:Nx
\seq_put_right:cn
\seq_put_right:cV
\seq_put_right:co
```

`\seq_put_right:Nn` ⟨*sequence*⟩ ⟨*token list*⟩

Locally appends ⟨*token list*⟩ as a single item to the right of ⟨*sequence*⟩. ⟨*token list*⟩ might get expanded before appending according to the variant.

```
\seq_gput_left:Nn
\seq_gput_left:NV
\seq_gput_left:No
\seq_gput_left:Nx
\seq_gput_left:cn
\seq_gput_left:cV
\seq_gput_left:co
```

`\seq_gput_left:Nn` ⟨*sequence*⟩ ⟨*token list*⟩

Globally appends ⟨*token list*⟩ as a single item to the left of ⟨*sequence*⟩.

```
\seq_gput_right:Nn
\seq_gput_right:NV
\seq_gput_right:No
\seq_gput_right:Nx
\seq_gput_right:cn
\seq_gput_right:cV
\seq_gput_right:co
```

`\seq_gput_right:Nn` ⟨*sequence*⟩ ⟨*token list*⟩

Globally appends ⟨*token list*⟩ as a single item to the right of ⟨*sequence*⟩.

`\seq_gput_right:Nc` Variant of the above used in the `xor` package. Will probably be moved soon to that package. (Sep 2008)

58 Working with sequences

`\seq_get:NN`
`\seq_get:cN`

`\seq_get:NN` ⟨*sequence*⟩ ⟨*tl var.*⟩

Functions that locally assign the left-most item of ⟨*sequence*⟩ to the token list variable ⟨*tl var.*⟩. Item is not removed from ⟨*sequence*⟩! If you need a global return value you need to code something like this:

```
\seq_get:NN ⟨sequence⟩ \l_tmpa_tl
\tl_gset_eq:NN ⟨global tl var.⟩ \l_tmpa_tl
```

But if this kind of construction is used often enough a separate function should be provided.

`\seq_map_variable:NNn`
`\seq_map_variable:cNn`

`\seq_map_variable:NNn` ⟨*sequence*⟩ ⟨*tl var.*⟩ `{`⟨*code using tl var.*⟩`}`

Every element in ⟨*sequence*⟩ is assigned to ⟨*tl var.*⟩ and then ⟨*code using tl var.*⟩ is executed. The operation is not expandable which means that it can't be used within write operations etc. However, this function can be nested which the others can't.

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

`\seq_map_function:NN` ⟨*sequence*⟩ ⟨*function*⟩

This function applies ⟨*function*⟩ (which must be a function with one argument) to every item of ⟨*sequence*⟩. ⟨*function*⟩ is not executed within a sub-group so that side effects can be achieved locally. The operation is not expandable which means that it can't be used within write operations etc.

In the current implementation the next functions are more efficient and should be preferred.

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

`\seq_map_inline:Nn` ⟨*sequence*⟩ `{`⟨*inline function*⟩`}`

Applies ⟨*inline function*⟩ (which should be the direct coding for a function with one argument (i.e. use #1 as the place holder for this argument)) to every item of ⟨*sequence*⟩. ⟨*inline function*⟩ is not executed within a sub-group so that side effects can be achieved locally. The operation is not expandable which means that it can't be used within write operations etc.

\seq_map_break:
\seq_map_break:n

These functions are used to break out of a mapping function at the point of execution. (Please do not put '\q_stop' inside a ⟨*seq*⟩ that uses these functions.)

\seq_show:N
\seq_show:c

\seq_show:N ⟨*sequence*⟩

Function that pauses the compilation and displays ⟨*seq*⟩ in the terminal output and in the log file. (Usually used for diagnostic purposes.)

\seq_display:N
\seq_display:c

\seq_display:N ⟨*sequence*⟩

As with \seq_show:N but pretty prints the output one line per element.

\seq_remove_duplicates:N
\seq_gremove_duplicates:N

\seq_gremove_duplicates:N ⟨*seq*⟩

Function that removes any duplicate entries in ⟨*seq*⟩.

59 Predicates and conditionals

\seq_if_empty_p:N ⋆
\seq_if_empty_p:c ⋆

\seq_if_empty_p:N ⟨*sequence*⟩

This predicate returns 'true' if ⟨*sequence*⟩ is 'empty' i.e., doesn't contain any items. Note that this is 'false' even if the ⟨*sequence*⟩ only contains a single empty item.

\seq_if_empty:N*TF*
\seq_if_empty:c*TF*

\seq_if_empty:NTF ⟨*sequence*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Set of conditionals that test whether or not a particular ⟨*sequence*⟩ is empty and if so executes either ⟨*true code*⟩ or ⟨*false code*⟩.

\seq_if_in:Nn*TF*
\seq_if_in:NV*TF*
\seq_if_in:cn*TF*
\seq_if_in:cV*TF*
\seq_if_in:co*TF*
\seq_if_in:cx*TF*

\seq_if_in:NnTF ⟨*sequence*⟩ {⟨*item*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Functions that test if ⟨*item*⟩ is in ⟨*sequence*⟩. Depending on the result either ⟨*true code*⟩ or ⟨*false code*⟩ is executed.

60 Internal functions

`\seq_if_empty_err:N` `\seq_if_empty_err:N` ⟨*sequence*⟩

Signals an LaTeX3 error if ⟨*sequence*⟩ is empty.

`\seq_pop_aux:nnNN` `\seq_pop_aux:nnNN` ⟨*assign*$_1$⟩ ⟨*assign*$_2$⟩ ⟨*sequence*⟩ ⟨*tl var.*⟩

Function that assigns the left-most item of ⟨*sequence*⟩ to ⟨*tl var.*⟩ using ⟨*assign*$_1$⟩ and assigns the tail to ⟨*sequence*⟩ using ⟨*assign*$_2$⟩. This function could be used to implement a global return function.

`\seq_get_aux:w`
`\seq_pop_aux:w`
`\seq_put_aux:Nnn`
`\seq_put_aux:w`

Functions used to implement put and get operations. They are not for meant for direct use.

`\seq_elt:w`
`\seq_elt_end:`

Functions (usually used as constants) that separates items within a sequence. They might get special meaning during mapping operations and are not supposed to show up as tokens within an item appended to a sequence.

61 Functions for 'Sequence Stacks'

Special sequences in LaTeX3 are 'stacks' with their usual operations of 'push', 'pop', and 'top'. They are internally implemented as sequences and share some of the functions (like `\seq_new:N` etc.)

`\seq_push:Nn`
`\seq_push:NV`
`\seq_push:No`
`\seq_push:cn`
`\seq_gpush:Nn`
`\seq_gpush:NV`
`\seq_gpush:No`
`\seq_gpush:Nv`
`\seq_gpush:cn`

`\seq_push:Nn` ⟨*stack*⟩ `{`⟨*token list*⟩`}`

Locally or globally pushes ⟨*token list*⟩ as a single item onto the ⟨*stack*⟩.

```
\seq_pop:NN
\seq_pop:cN
\seq_gpop:NN
\seq_gpop:cN
```

\seq_pop:NN ⟨*stack*⟩ ⟨*tl var.*⟩

Functions that assign the top item of ⟨*stack*⟩ to ⟨*tl var.*⟩ and removes it from ⟨*stack*⟩!

```
\seq_top:NN
\seq_top:cN
```

\seq_top:NN ⟨*stack*⟩ ⟨*tl var.*⟩

Functions that locally assign the top item of ⟨*stack*⟩ to the ⟨*tl var.*⟩. Item is *not* removed from ⟨*stack*⟩!

Part XIV

The l3clist package Comma separated lists

LaTeX3 implements a data type called 'clist (comma-lists)'. These are special token lists that can be accessed via special function on the 'left'. Appending tokens is possible at both ends. Appended token lists can be accessed only as a union. The token lists that form the individual items of a comma-list might contain any tokens except for commas that are used to structure comma-lists (braces are need if commas are part of the value). It is also possible to map functions on such comma-lists so that they are executed for every item of the comma-list.

All functions that return items from a comma-list in some ⟨*tl var.*⟩ assume that the ⟨*tl var.*⟩ is local. See remarks below if you need a global returned value.

The defined functions are not orthogonal in the sense that every possible variation possible is actually available. If you need a new variant use the expansion functions described in the package `l3expan` to build it.

Adding items to the left of a comma-list can currently be done with either something like `\clist_put_left:Nn` or with a "stack" function like `\clist_push:Nn` which has the same effect. Maybe one should therefore remove the "left" functions totally.

62 Functions for creating/initialising comma-lists

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

\clist_new:N ⟨*comma-list*⟩

Defines ⟨*comma-list*⟩ to be a variable of type clist.

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

`\clist_clear:N` ⟨*comma-list*⟩

These functions locally or globally clear ⟨*comma-list*⟩.

```
\clist_clear_new:N
\clist_clear_new:c
\clist_gclear_new:N
\clist_gclear_new:c
```

`\clist_clear_new:N` ⟨*comma-list*⟩

These functions locally or globally clear ⟨*comma-list*⟩ if it exists or otherwise allocates it.

```
\clist_set_eq:NN
\clist_set_eq:cN
\clist_set_eq:Nc
\clist_set_eq:cc
```

`\clist_set_eq:NN` ⟨*clist*$_1$⟩ ⟨*clist*$_2$⟩

Function that locally makes ⟨*clist*$_1$⟩ identical to ⟨*clist*$_2$⟩.

```
\clist_gset_eq:NN
\clist_gset_eq:cN
\clist_gset_eq:Nc
\clist_gset_eq:cc
```

`\clist_gset_eq:NN` ⟨*clist*$_1$⟩ ⟨*clist*$_2$⟩

Function that globally makes ⟨*clist*$_1$⟩ identical to ⟨*clist*$_2$⟩.

63 Putting data in

```
\clist_put_left:Nn
\clist_put_left:NV
\clist_put_left:No
\clist_put_left:Nx
\clist_put_left:cn
\clist_put_left:co
\clist_put_left:cV
```

`\clist_put_left:Nn` ⟨*comma-list*⟩ ⟨*token list*⟩

Locally appends ⟨*token list*⟩ as a single item to the left of ⟨*comma-list*⟩. ⟨*token list*⟩ might get expanded before appending according to the variant used.

\clist_put_right:Nn
\clist_put_right:No
\clist_put_right:NV
\clist_put_right:Nx
\clist_put_right:cn
\clist_put_right:co
\clist_put_right:cV

\clist_put_right:Nn ⟨*comma-list*⟩ ⟨*token list*⟩

Locally appends ⟨*token list*⟩ as a single item to the right of ⟨*comma-list*⟩. ⟨*token list*⟩ might get expanded before appending according to the variant used.

\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:Nn ⟨*comma-list*⟩ ⟨*token list*⟩

Globally appends ⟨*token list*⟩ as a single item to the right of ⟨*comma-list*⟩.

\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:Nn ⟨*comma-list*⟩ ⟨*token list*⟩

Globally appends ⟨*token list*⟩ as a single item to the right of ⟨*comma-list*⟩.

64 Getting data out

\clist_use:N
\clist_use:c

\clist_use:N ⟨*clist*⟩

Function that inserts the ⟨*clist*⟩ into the processing stream. Mainly useful if one knows what the ⟨*clist*⟩ contains, e.g., for displaying the content of template parameters.

\clist_show:N
\clist_show:c

\clist_show:N ⟨*clist*⟩

Function that pauses the compilation and displays ⟨*clist*⟩ in the terminal output and in the log file. (Usually used for diagnostic purposes.)

\clist_display:N
\clist_display:c

\clist_display:N ⟨*clist*⟩

As with \clist_show:N but pretty prints the output one line per element.

\clist_get:NN
\clist_get:cN

\clist_get:NN ⟨*comma-list*⟩ ⟨*tl var.*⟩

Functions that locally assign the left-most item of ⟨*comma-list*⟩ to the token list variable ⟨*tl var.*⟩. Item is not removed from ⟨*comma-list*⟩! If you need a global return value you need to code something like this:

```
\clist_get:NN ⟨comma-list⟩ \l_tmpa_tl
\tl_gset_eq:NN ⟨global tl var.⟩ \l_tmpa_tl
```

But if this kind of construction is used often enough a separate function should be provided.

65 Mapping functions

We provide three types of mapping functions, each with their own strengths. The \clist_map_function:NN is expandable whereas \clist_map_inline:Nn type uses ##1 as a placeholder for the current item in ⟨*clist*⟩. Finally we have the \clist_map_variable:NNn type which uses a user-defined variable as placeholder. Both the _inline and _variable versions are nestable.

\clist_map_function:NN
\clist_map_function:cN
\clist_map_function:nN

\clist_map_function:NN ⟨*comma-list*⟩ ⟨*function*⟩

This function applies ⟨*function*⟩ (which must be a function with one argument) to every item of ⟨*comma-list*⟩. ⟨*function*⟩ is not executed within a sub-group so that side effects can be achieved locally. The operation is expandable which means that it can be used within write operations etc.

\clist_map_inline:Nn
\clist_map_inline:cn
\clist_map_inline:nn

\clist_map_inline:Nn ⟨*comma-list*⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ (which should be the direct coding for a function with one argument (i.e. use ##1 as the placeholder for this argument)) to every item of ⟨*comma-list*⟩. ⟨*inline function*⟩ is not executed within a sub-group so that side effects can be achieved locally. The operation is not expandable which means that it can't be used within write operations etc. These functions can be nested.

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

\clist_map_variable:NNn ⟨*comma-list*⟩ ⟨*temp-var*⟩ {⟨*action*⟩}

Assigns ⟨*temp-var*⟩ to each element in ⟨*clist*⟩ and then executes ⟨*action*⟩ which should contain ⟨*temp-var*⟩. As the operation performs an assignment, it is not expandable.

TEXhackers note: These functions resemble the LaTeX 2_ε function `\@for` but does not borrow the somewhat strange syntax.

```
\clist_map_break:
```

\clist_map_break:

For breaking out of a loop. To be used inside `TF` type functions as in the example below.

```
\cs_new_nopar:Npn \test_function:n #1 {
  \intexpr_compare:nTF {#1 > 3} {\clist_map_break:}{``#1''}
}
\clist_map_function:nN {1,2,3,4,5,6,7,8}\test_function:n
```

This would return ``1''``2''``3''.

66 Predicates and conditionals

```
\clist_if_empty_p:N
\clist_if_empty_p:c
```

\clist_if_empty_p:N ⟨*comma-list*⟩

This predicate returns 'true' if ⟨*comma-list*⟩ is 'empty' i.e., doesn't contain any tokens.

```
\clist_if_empty:NTF ⋆
\clist_if_empty:cTF ⋆
```

\clist_if_empty:NTF ⟨*comma-list*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Set of conditionals that test whether or not a particular ⟨*comma-list*⟩ is empty and if so executes either ⟨*true code*⟩ or ⟨*false code*⟩.

```
\clist_if_eq_p:NN ⋆
\clist_if_eq_p:cN ⋆
\clist_if_eq_p:Nc ⋆
\clist_if_eq_p:cc ⋆
```

\clist_if_eq_p:N ⟨*comma-list*$_1$⟩ ⟨*comma-list*$_2$⟩

This predicate returns 'true' if the two comma lists are identical.

```
\clist_if_eq:NNTF ★
\clist_if_eq:cNTF ★
\clist_if_eq:NcTF ★
\clist_if_eq:ccTF ★
```

`\clist_if_eq:NNTF` ⟨*comma-list*$_1$⟩ ⟨*comma-list*$_2$⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Check if ⟨*comma-list*$_1$⟩ and ⟨*comma-list*$_2$⟩ are equal and execute either ⟨*true code*⟩ or ⟨*false code*⟩ accordingly.

```
\clist_if_in:NnTF
\clist_if_in:NVTF
\clist_if_in:NoTF
\clist_if_in:cnTF
\clist_if_in:cVTF
\clist_if_in:coTF
```

`\clist_if_in:NnTF` ⟨*comma-list*⟩ `{`⟨*item*⟩`}` `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Function that tests if ⟨*item*⟩ is in ⟨*comma-list*⟩. Depending on the result either ⟨*true code*⟩ or ⟨*false code*⟩ is executed.

67 Higher level functions

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

`\clist_gconcat:NNN` ⟨*clist*$_1$⟩ ⟨*clist*$_2$⟩ ⟨*clist*$_3$⟩

Function that concatenates ⟨*clist*$_2$⟩ and ⟨*clist*$_3$⟩ and locally or globally assigns the result to ⟨*clist*$_1$⟩.

```
\clist_remove_duplicates:N
\clist_gremove_duplicates:N
```

`\clist_gremove_duplicates:N` ⟨*clist*⟩

Function that removes any duplicate entries in ⟨*clist*⟩.

```
\clist_remove_element:Nn
\clist_gremove_element:Nn
```

`\clist_gremove_element:Nn` ⟨*clist*⟩ ⟨*element*⟩

Function that removes ⟨*element*⟩ from ⟨*clist*⟩, if present.

TEXhackers note: This is similar in concept to `\@removeelement`, except that the syntax is clearer and the initial and final lists have the same name automatically.

68 Functions for 'comma-list stacks'

Special comma-lists in LaTeX3 are 'stacks' with their usual operations of 'push', 'pop', and 'top'. They are internally implemented as comma-lists and share some of the functions (like `\clist_new:N` etc.)

```
\clist_push:Nn
\clist_push:NV
\clist_push:No
\clist_push:cn
\clist_gpush:Nn
\clist_gpush:NV
\clist_gpush:No
\clist_gpush:cn
```

`\clist_push:Nn` ⟨*stack*⟩ `{`⟨*token list*⟩`}`

Locally or globally pushes ⟨*token list*⟩ as a single item onto the ⟨*stack*⟩. ⟨*token list*⟩ might get expanded before the operation.

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

`\clist_pop:NN` ⟨*stack*⟩ ⟨*tl var.*⟩

Functions that assign the top item of ⟨*stack*⟩ to the token list variable ⟨*tl var.*⟩ and removes it from ⟨*stack*⟩!

```
\clist_top:NN
\clist_top:cN
```

`\clist_top:NN` ⟨*stack*⟩ ⟨*tl var.*⟩

Functions that locally assign the top item of ⟨*stack*⟩ to the token list variable ⟨*tl var.*⟩. Item is not removed from ⟨*stack*⟩!

69 Internal functions

```
\clist_if_empty_err:N
```

`\clist_if_empty_err:N` ⟨*comma-list*⟩

Signals an LaTeX3 error if ⟨*comma-list*⟩ is empty.

```
\clist_pop_aux:nnNN
```

`\clist_pop_aux:nnNN` ⟨*assign* $_1$⟩ ⟨*assign* $_2$⟩ ⟨*comma-list*⟩ ⟨*tl var.*⟩

Function that assigns the left-most item of ⟨*comma-list*⟩ to ⟨*tl var.*⟩ using ⟨*assign* $_1$⟩ and assigns the tail to ⟨*comma-list*⟩ using ⟨*assign* $_2$⟩. This function could be used to implement a global return function.

```
\clist_get_aux:w
\clist_pop_aux:w
\clist_pop_auxi:w
\clist_put_aux:NNnnNn
```

Functions used to implement put and get operations. They are not for meant for direct use.

Part XV

The l3prop package
Property Lists

LaTeX3 implements a data structure called a 'property list' which allows arbitrary information to be stored and accessed using keywords rather than numerical indexing.

A property list might contain a set of keys such as `name`, `age`, and `ID`, which each have individual values that can be saved and retrieved.

70 Functions

```
\prop_new:N
\prop_new:c
```

`\prop_new:N` ⟨*prop*⟩

Defines ⟨*prop*⟩ to be a variable of type ⟨*prop*⟩.

```
\prop_clear:N
\prop_clear:c
\prop_gclear:N
\prop_gclear:c
```

`\prop_clear:N` ⟨*prop*⟩

These functions locally or globally clear ⟨*prop*⟩.

\prop_put:Nnn
\prop_put:NnV
\prop_put:NVn
\prop_put:NVV
\prop_put:cnn
\prop_gput:Nnn
\prop_gput:NVn
\prop_gput:Nno
\prop_gput:NnV
\prop_gput:Nnx
\prop_gput:cnn
\prop_gput:ccx

\prop_put:Nnn ⟨*prop*⟩ {⟨*key*⟩} {⟨*token list*⟩}

Locally or globally associates ⟨*token list*⟩ with ⟨*key*⟩ in the ⟨*prop*⟩ ⟨*prop*⟩. If ⟨*key*⟩ has already a meaning within ⟨*prop*⟩ this value is overwritten.

The ⟨*key*⟩ must not contain unescaped # tokens but the ⟨*token list*⟩ may.

\prop_gput_if_new:Nnn

\prop_gput_if_new:Nnn ⟨*prop*⟩ {⟨*key*⟩} {⟨*token list*⟩}

Globally associates ⟨*token list*⟩ with ⟨*key*⟩ in the ⟨*prop*⟩ ⟨*prop*⟩ but only if ⟨*key*⟩ has so far no meaning within ⟨*prop*⟩. Silently ignored if ⟨*key*⟩ is already set in the ⟨*prop*⟩.

\prop_get:NnN
\prop_get:NVN
\prop_get:cnN
\prop_get:cVN
\prop_gget:NnN
\prop_gget:NVN
\prop_gget:cnN
\prop_gget:cVN

\prop_get:NnN ⟨*prop*⟩ {⟨*key*⟩} ⟨*tl var.*⟩

If ⟨*info*⟩ is the information associated with ⟨*key*⟩ in the ⟨*prop*⟩ ⟨*prop*⟩ then the token list variable ⟨*tl var.*⟩ gets ⟨*info*⟩ assigned. Otherwise its value is the special quark \q_no_value. The assignment is done either locally or globally.

\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

\prop_set_eq:NN ⟨*prop* $_1$⟩ ⟨*prop* $_2$⟩

A fast assignment of ⟨*prop*⟩s.

`\prop_get_gdel:NnN` `\prop_get_gdel:NnN` ⟨*prop*⟩ {⟨*key*⟩} ⟨*tl var.*⟩

Like `\prop_get:NnN` but additionally removes ⟨*key*⟩ (and its ⟨*info*⟩) from ⟨*prop*⟩.

`\prop_del:Nn`
`\prop_del:NV`
`\prop_gdel:Nn`
`\prop_gdel:NV`

`\prop_del:Nn` ⟨*prop*⟩ {⟨*key*⟩}

Locally or globally deletes ⟨*key*⟩ and its ⟨*info*⟩ from ⟨*prop*⟩ if found. Otherwise does nothing.

`\prop_map_function:NN` ⋆
`\prop_map_function:cN` ⋆
`\prop_map_function:Nc` ⋆
`\prop_map_function:cc` ⋆

`\prop_map_function:NN` ⟨*prop*⟩ ⟨*function*⟩

Maps ⟨*function*⟩ which should be a function with two arguments (⟨*key*⟩ and ⟨*info*⟩) over every ⟨*key*⟩ ⟨*info*⟩ pair of ⟨*prop*⟩. Property lists do not have any intrinsic "order" when stored. As a result, you should not expect any particular order to apply when using these mapping functions, even with newly-created properly lists.

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

`\prop_map_inline:Nn` ⟨*prop*⟩ {⟨*inline function*⟩}

Just like `\prop_map_function:NN` but with the function of two arguments supplied as inline code. Within ⟨*inline function*⟩ refer to the arguments via `#1` (⟨*key*⟩) and `#2` (⟨*info*⟩). Nestable. Property lists do not have any intrinsic "order" when stored. As a result, you should not expect any particular order to apply when using these mapping functions, even with newly-created properly lists.

`\prop_map_break:`

```
\prop_map_inline:Nn ⟨prop⟩ {
  ... \⟨break test⟩:T {\prop_map_break:} }
```

For breaking out of a loop. To be used inside `TF`-type functions as shown in the example above.

`\prop_show:N`
`\prop_show:c`

`\prop_show:N` ⟨*prop*⟩

Pauses the compilation and shows ⟨*prop*⟩ on the terminal output and in the log file.

`\prop_display:N`
`\prop_display:c`

`\prop_display:N` ⟨*prop*⟩

As with `\prop_show:N` but pretty prints the output one line per property pair.

71 Predicates and conditionals

\prop_if_empty_p:N
\prop_if_empty_p:c

\prop_if_empty_p:N ⟨*prop*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Predicates to test whether or not a particular ⟨*prop*⟩ is empty.

\prop_if_empty:N*TF* ⋆
\prop_if_empty:c*TF* ⋆

\prop_if_empty:NTF ⟨*prop*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Set of conditionals that test whether or not a particular ⟨*prop*⟩ is empty.

\prop_if_eq_p:NN ⋆
\prop_if_eq_p:cN ⋆
\prop_if_eq_p:Nc ⋆
\prop_if_eq_p:cc ⋆
\prop_if_eq:NN*TF* ⋆
\prop_if_eq:cN*TF* ⋆
\prop_if_eq:Nc*TF* ⋆
\prop_if_eq:cc*TF* ⋆

\prop_if_eq:NNF ⟨*prop*$_1$⟩ ⟨*prop*$_2$⟩ {⟨*false code*⟩}

Execute ⟨*false code*⟩ if ⟨*prop*$_1$⟩ doesn't hold the same token list as ⟨*prop*$_2$⟩. Only expandable for new versions of pdfTEX.

\prop_if_in:Nn*TF*
\prop_if_in:NV*TF*
\prop_if_in:No*TF*
\prop_if_in:cn*TF*
\prop_if_in:cc*TF*

\prop_if_in:NnTF ⟨*prop*⟩ {⟨*key*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*key*⟩ is used in ⟨*prop*⟩ and then either executes ⟨*true code*⟩ or ⟨*false code*⟩.

72 Internal functions

\q_prop

Quark used to delimit property lists internally.

\prop_put_aux:w
\prop_put_if_new_aux:w

Internal functions implementing the put operations.

\prop_get_aux:w
\prop_gget_aux:w
\prop_get_del_aux:w
\prop_del_aux:w

Internal functions implementing the get and delete operations.

`\prop_if_in_aux:w` Internal function implementing the key test operation.

`\prop_map_function_aux:w` Internal function implementing the map operations.

`\g_prop_inline_level_int` Integer used in internal name for function used inside `\prop_map_inline:NN`.

`\prop_split_aux:Nnn` `\prop_split_aux:Nnn` ⟨*prop*⟩ ⟨*key*⟩ ⟨*cmd*⟩
Internal function that invokes ⟨*cmd*⟩ with 3 arguments: 1st is the beginning of ⟨*prop*⟩ before ⟨*key*⟩, 2nd is the value associated with ⟨*key*⟩, 3rd is the rest of ⟨*prop*⟩ after ⟨*key*⟩. If there is no key ⟨*key*⟩ in ⟨*prop*⟩, then the 2 arg is `\q_no_value` and the 3rd arg is empty; otherwise the 3rd argument has the two extra tokens ⟨*key*⟩ `\q_no_value` at the end.

This function is used to implement various get operations.

Part XVI

The l3io package Low-level file i/o

Reading and writing from file streams is handled in LaTeX3 using functions with prefixes `\iow_...` (file reading) and `\ior_...` (file writing). Many of the basic functions are very similar, with reading and writing using the same syntax and function concepts. As a result, the reading and writing functions are documented together where this makes sense.

As TeX is limited to 16 input streams and 16 output streams, direct use of the streams by the programmer is not supported in LaTeX3. 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.

Reading from or writing to a file requires a ⟨*stream*⟩ to be used. This is a csname which refers to the file being processed, and is independent of the name of the file (except of course that the file name is needed when the file is opened).

73 Opening and closing streams

```
\iow_new:N
\iow_new:c
\ior_new:N
\ior_new:c
```

`\iow_new:N` ⟨*stream*⟩

Reserves the name ⟨*stream*⟩ for use in accessing a file stream. This operation does not open a raw TeX stream, which is handled internally using a pool and is should not be accessed directly by the programmer.

```
\iow_open:Nn
\iow_open:cn
\ior_open:Nn
\ior_open:cn
```

`\iow_open:Nn` ⟨*stream*⟩ `{`⟨*file name*⟩`}`
`\iow_open:Nn` ⟨*stream*⟩ `{`⟨*file name*⟩`}`

Opens ⟨*file name*⟩ for writing (`\iow_...`) or reading (`\ior_...`) using ⟨*stream*⟩ as the csname by which the file is accessed. If ⟨*stream*⟩ was already open (for either writing or reading) it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately after issuing an `open` instruction. The ⟨*stream*⟩ will remain allocated to ⟨*file name*⟩ until a `close` instruction is given or at the end of the TeX run.

Opening a file for writing will clear any existing content in the file (*i.e.* writing is *not* additive). As the total number of writing streams is limited, it may well be best to save material to be written to an intermediate storage format (for example a token list or toks), and to write the material in one 'shot' from this variable. In this way the file stream is only required for a limited time.

```
\iow_close:N
\iow_close:c
\ior_close:N
\ior_close:c
```

`\iow_close:N` ⟨*stream*⟩
`\ior_close:N` ⟨*stream*⟩

Closes ⟨*stream*⟩, freeing up one of the underlying TeX streams for reuse. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers (the resources here are limited). The name of the ⟨*stream*⟩ will be freed at this stage, to ensure that any further attempts to write to it result in an error.

```
\iow_open_streams:
\iow_open_streams:
```

`\iow_open_streams:`

Displays a list of the file names associated with each open stream: intended for tracking down problems.

73.1 Writing to files

```
\iow_now:Nx
\iow_now:Nn
```

\iow_now:Nx ⟨*stream*⟩ {⟨*tokens*⟩}

\iow_now:Nx immediately writes the expansion of ⟨*tokens*⟩ to the output ⟨*stream*⟩. If the ⟨*stream*⟩ is not open output goes to the terminal. The variant \iow_now:Nn writes out ⟨*tokens*⟩ without any further expansion.

TeXhackers note: These are the equivalent of TeX's \immediate\write with and without expansion control.

```
\iow_log:n
\iow_log:x
\iow_term:n
\iow_term:x
```

\iow_log:x {⟨*tokens*⟩}

These are dedicated functions which write to the log (transcript) file and the terminal, respectively. They are equivalent to using \iow_now:N(n/x) to the streams \c_iow_log_stream and \c_iow_term_stream. The writing takes place immediately.

```
\iow_now_buffer_safe:Nn
\iow_now_buffer_safe:Nx
```

\iow_now_buffer_safe:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

Immediately write ⟨*tokens*⟩ expanded to ⟨*stream*⟩, with every space converted into a newline. This mean that the file can be read back without the danger that very long lines overflow TeX's buffer.

```
\iow_now_when_avail:Nn
\iow_now_when_avail:cn
\iow_now_when_avail:Nx
\iow_now_when_avail:cx
```

\iow_now_when_avail:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

If ⟨*stream*⟩ is open, writes the ⟨*tokens*⟩ to the ⟨*stream*⟩ in the same manner as \iow_now:N(n/x). If the ⟨*stream*⟩ is not open, the ⟨*tokens*⟩ are simply thrown away.

```
\iow_shipout:Nx
\iow_shipout:Nn
```

\iow_shipout:Nx ⟨*stream*⟩ {⟨*tokens*⟩}

Write ⟨*tokens*⟩ to ⟨*stream*⟩ at the point at which the current page is finished. The ⟨*tokens*⟩ are either written unexpanded (\iow_shipout:Nn) or expanded only at the point that the function is used (\iow_shipout:Nx), *i.e.* no expansion takes place when writing to the file.

\iow_shipout_x:Nx
\iow_shipout_x:Nn

`\iow_shipout_x:Nx` ⟨*stream*⟩ `{`⟨*tokens*⟩`}`

Write ⟨*tokens*⟩ to ⟨*stream*⟩ at the point at which the current page is finished. The ⟨*tokens*⟩ are expanded at the time of writing in addition to any expansion at the time of use of the function. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

TeXhackers note: These are the equivalent of TeX's `\write` with and without expansion control at point of use.

\iow_newline: ⋆

`\iow_newline:`

Function to add a new line within the ⟨*tokens*⟩ written to a file. The function has no effect if writing is taking place without expansion (*e.g.* in a `\iow_now:Nn` call).

\iow_char:N ⋆

`\iow_char:N \`⟨*char*⟩
`\iow_char:N \%`

Inserts ⟨*char*⟩ into the output stream. Useful when trying to write difficult characters such as `%`, `{`, `}`, *etc.* in messages, for example:

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

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

73.2 Reading from files

\ior_to:NN
\ior_gto:NN

`\ior_to:NN` ⟨*stream*⟩ ⟨*token list variable*⟩

Functions that reads one or more lines (until an equal number of left and right braces are found) from the input stream ⟨*stream*⟩ and places the result locally or globally into the ⟨*token list variable*⟩. If ⟨*stream*⟩ is not open, input is requested from the terminal.

\ior_if_eof_p:N ⋆
\ior_if_eof:NTF ⋆

`\ior_if_eof:NTF` ⟨*stream*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the end of a ⟨*stream*⟩ has been reached during a reading operation. The test will also return a `true` value if the ⟨*stream*⟩ is not open or the ⟨*file name*⟩ associated with a ⟨*stream*⟩ does not exist at all.

74 Internal functions

```
\iow_raw_new:N
\iow_raw_new:c
\ior_raw_new:N
\ior_raw_new:c
```

\iow_raw_new:N ⟨*stream*⟩

Creates a new low-level ⟨*stream*⟩ for use in subsequent functions. As allocations are made using a pool *do not use this function!*

TEXhackers note: This is LaTeX 2_ε's \newwrite.

```
\if_eof:w ⋆
```

\if_eof:w ⟨*stream*⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

Tests if the end of ⟨*stream*⟩ has been reached during a reading operation.

TEXhackers note: This is the primitive \ifeof.

75 Variables and constants

```
\c_io_streams_tl
```

A list of the positions available for stream allocation (numbers 0 to 15).

```
\c_iow_term_stream
\c_ior_term_stream
\c_iow_log_stream
\c_ior_log_stream
```

Fixed stream numbers for accessing to the log and the terminal. The reading and writing values are the same but are provided so that the meaning is clear.

```
\g_iow_streams_prop
\g_ior_streams_prop
```

Allocation records for streams, linking the stream number to the current name being used for that stream.

```
\g_iow_tmp_stream
\g_ior_tmp_stream
```

Used when creating new streams at the TEX level.

```
\l_iow_stream_int
\l_ior_stream_int
```

Number of stream currently being allocated.

Part XVII

The l3msg package
Communicating with the user

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 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.

76 Creating new messages

All messages have to be created before they can be used. Inside the message text, spaces are *not* ignored. A space where TeX would normally gobble one can be created using `\ `, and a new line with `\\`. New lines may have 'continuation' text added by the output system.

`\msg_new:nnnn`
`\msg_new:nnn`
`\msg_set:nnnn`
`\msg_set:nnn`

`\msg_new:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Creates new ⟨*message*⟩ for ⟨*module*⟩ to produce ⟨*text*⟩ initially and ⟨*more text*⟩ if requested by the user. ⟨*text*⟩ and ⟨*more text*⟩ can use up to four macro parameters (`#1` to `#4`), which are supplied by the message system. At the point where ⟨*message*⟩ is printed, the material supplied for `#1` to `#4` will be subject to an x-type expansion.

An error will be raised by the `new` functions if the message already exists: the `set` functions do not carry any checking. For messages defined using `\msg_new:nnn` or `\msg_set:nnn` LaTeX3 will supply a standard ⟨*more text*⟩ at the point the message is used, if this is required.

77 Message classes

Creating message output requires the message to be given a class.

\msg_class_new:nn
\msg_class_set:nn

`\msg_class_new:nn {⟨class⟩} {⟨code⟩}`

Creates new ⟨*class*⟩ to output a message, using ⟨*code*⟩ to process the message text. The ⟨*class*⟩ should be a text value, while the ⟨*code*⟩ may be any arbitrary material.

The module defines several common message classes. The following describes the standard behaviour of each class if no redirection of the class or message is active. In all cases, the message may be issued supplying 0 to 4 arguments. The code will ensure that there an no errors if the number of arguments supplied here does not match the number in the definition of the message (although of course the sense of the message may be impaired).

\msg_fatal:nnxxxx
\msg_fatal:nnxxx
\msg_fatal:nnxx
\msg_fatal:nnx
\msg_fatal:nn

`\msg_fatal:nnxxxx {⟨module⟩} {⟨name⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ error message ⟨*name*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. After issuing a fatal error the TeX run will halt.

\msg_error:nnxxxx
\msg_error:nnxxx
\msg_error:nnxx
\msg_error:nnx
\msg_error:nn

`\msg_error:nnxxxx {⟨module⟩} {⟨name⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ error message ⟨*name*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions.

TeXhackers note: The standard output here is similar to `\PackageError`.

\msg_warning:nnxxxx
\msg_warning:nnxxx
\msg_warning:nnxx
\msg_warning:nnx
\msg_warning:nn

`\msg_warning:nnxxxx {⟨module⟩} {⟨name⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Prints ⟨*module*⟩ message ⟨*name*⟩ to the terminal, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions.

TeXhackers note: The standard output here is similar to `\PackageWarningNoLine`.

\msg_info:nnxxxx
\msg_info:nnxxx
\msg_info:nnxx
\msg_info:nnx
\msg_info:nn

\msg_info:nnxxxx {⟨*module*⟩} {⟨*name*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Prints ⟨*module*⟩ message ⟨*name*⟩ to the log, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions.

TEXhackers note: The standard output here is similar to \PackageInfoNoLine.

\msg_log:nnxxxx
\msg_log:nnxxx
\msg_log:nnxx
\msg_log:nnx
\msg_log:nn

\msg_log:nnxxxx {⟨*module*⟩} {⟨*name*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Prints ⟨*module*⟩ message ⟨*name*⟩ to the log, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. No continuation text is added.

\msg_trace:nnxxxx
\msg_trace:nnxxx
\msg_trace:nnxx
\msg_trace:nnx
\msg_trace:nn

\msg_trace:nnxxxx {⟨*module*⟩} {⟨*name*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Prints ⟨*module*⟩ message ⟨*name*⟩ to the log, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. No continuation text is added.

\msg_none:nnxxxx
\msg_none:nnxxx
\msg_none:nnxx
\msg_none:nnx
\msg_none:nn

\msg_none:nnxxxx {⟨*module*⟩} {⟨*name*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Does nothing: used for redirecting other message classes. Gobbles arguments given.

78 Redirecting messages

\msg_redirect_class:nn

\msg_redirect_class:nn {⟨*class one*⟩} {⟨*class two*⟩}

Changes the behaviour of messages of ⟨*class one*⟩ so that they are processed using the code for those of ⟨*class two*⟩. Multiple redirections are possible. Redirection to a missing class or infinite loops will raise errors when the messages are used, rather than at the point of redirection.

`\msg_redirect_module:nnn` `\msg_redirect_module:nnn {⟨module⟩} {⟨class one⟩} {⟨class two⟩}`

Redirects message of ⟨*class one*⟩ for ⟨*module*⟩ to act as though they were from ⟨*class two*⟩. Messages of ⟨*class one*⟩ from sources other than ⟨*module*⟩ are not affected by this redirection.

TEXhackers note: This function can be used to make some messages 'silent' by default. For example, all of the `trace` messages of ⟨*module*⟩ could be turned off with:

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

`\msg_redirect_name:nnn` `\msg_redirect_name:nnn {⟨module⟩} {⟨message⟩} {⟨class⟩}`

Redirects a specific ⟨*message*⟩ from a specific ⟨*module*⟩ to act as a member of ⟨*class*⟩ of messages.

TEXhackers note: This function can be used to make a selected message 'silent' without changing global parameters:

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

79 Support functions for output

`\msg_line_context:` `\msg_line_context:`

Prints the text specified in `\c_msg_on_line_tl` followed by the current line in the current input file.

TEXhackers note: This is similar to the text added to messages by LaTeX 2_ε's `\PackageWarning` and `\PackageInfo`.

`\msg_line_number:` `\msg_line_number:`

Prints the current line number in the current input file.

\msg_newline:
\msg_two_newlines:

\msg_newline:

Print one or two newlines with no continuation information.

80 Low-level functions

The low-level functions do not make assumptions about module names. The output functions here produce messages directly, and do not respond to redirection.

\msg_generic_new:nnn
\msg_generic_new:nn
\msg_generic_set:nnn
\msg_generic_set:nn

\msg_generic_new:nnn {⟨*name*⟩} {⟨*text*⟩} {⟨*more text*⟩}

Creates new message ⟨*name*⟩ to produce ⟨*text*⟩ initially and ⟨*more text*⟩ if requested by the user. ⟨*text*⟩ and ⟨*more text*⟩ can use up to four macro parameters (#1 to #4), which are supplied by the message system. Inside ⟨*text*⟩ and ⟨*more text*⟩ spaces are not ignored.

\msg_direct_interrupt:xxxx

\msg_direct_interrupt:xxxxn {⟨*first line*⟩} {⟨*text*⟩} {⟨*continuation*⟩} {⟨*more text*⟩}

Executes a TeX error, interrupting compilation. The ⟨*first line*⟩ is displayed followed by ⟨*text*⟩ and the input prompt. ⟨*more text*⟩ is displays if requested by the user. If ⟨*more text*⟩ is blank a default is supplied. Each line of ⟨*text*⟩ (broken with \\) begins with ⟨*continuation*⟩.

\msg_direct_log:xx
\msg_direct_term:xx

\msg_direct_log:xx {⟨*text*⟩} {⟨*continuation*⟩}

Prints ⟨*text*⟩ to either the log or terminal. New lines (broken with \\) start with ⟨*continuation*⟩.

81 Kernel-specific functions

\msg_kernel_new:nnnn
\msg_kernel_new:nnn
\msg_kernel_set:nnnn
\msg_kernel_set:nnn

\msg_kernel_new:nnnn {⟨*divsion*⟩} {⟨*name*⟩} {⟨*text*⟩} {⟨*more text*⟩}

Creates new kernel message ⟨*name*⟩ to produce ⟨*text*⟩ initially and ⟨*more text*⟩ if requested by the user. ⟨*text*⟩ and ⟨*more text*⟩ can use up to four macro parameters (`#1` to `#4`), which are supplied by the message system. Kernel messages are divided into ⟨*divisions*⟩, roughly equivalent to the LaTeX 2ε package names used.

```
\msg_kernel_fatal:nnxxxx
\msg_kernel_fatal:nnxxx
\msg_kernel_fatal:nnxx
\msg_kernel_fatal:nnx
\msg_kernel_fatal:nn
```

`\msg_kernel_fatal:nnxx {`⟨*division*⟩`} {`⟨*name*⟩`} {`⟨*arg one*⟩`} {`⟨*arg two*⟩`} {`⟨*arg three*⟩`} {`⟨*arg four*⟩`}`

Issues kernel error message ⟨*name*⟩ for ⟨*division*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The TeX run then halts. Cannot be redirected.

```
\msg_kernel_error:nnxxxx
\msg_kernel_error:nnxxx
\msg_kernel_error:nnxx
\msg_kernel_error:nnx
\msg_kernel_error:nn
```

`\msg_kernel_error:nnxx {`⟨*division*⟩`} {`⟨*name*⟩`} {`⟨*arg one*⟩`} {`⟨*arg two*⟩`} {`⟨*arg three*⟩`} {`⟨*arg four*⟩`}`

Issues kernel error message ⟨*name*⟩ for ⟨*division*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. Cannot be redirected.

```
\msg_kernel_warning:nnxxxx
\msg_kernel_warning:nnxxx
\msg_kernel_warning:nnxx
\msg_kernel_warning:nnx
\msg_kernel_warning:nn
```

`\msg_kernel_warning:nnxx {`⟨*division*⟩`} {`⟨*name*⟩`} {`⟨*arg one*⟩`} {`⟨*arg two*⟩`} {`⟨*arg three*⟩`} {`⟨*arg four*⟩`}`

Prints kernel message ⟨*name*⟩ for ⟨*division*⟩ to the terminal, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions.

```
\msg_kernel_info:nnxxxx
\msg_kernel_info:nnxxx
\msg_kernel_info:nnxx
\msg_kernel_info:nnx
\msg_kernel_info:nn
```

`\msg_kernel_info:nxx {`⟨*division*⟩`} {`⟨*name*⟩`} {`⟨*arg one*⟩`} {`⟨*arg two*⟩`} {`⟨*arg three*⟩`} {`⟨*arg four*⟩`}`

Prints kernel message ⟨*name*⟩ for ⟨*division*⟩ to the log, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions.

```
\msg_kernel_bug:x
```

`\msg_kernel_bug:x {`⟨*text*⟩`}`

Short-cut for 'This is a LaTeX bug: check coding' errors.

82 Variables and constants

\c_msg_fatal_tl
\c_msg_error_tl
\c_msg_warning_tl
\c_msg_info_tl

Simple headers for errors. Although these are marked as constants, they could be changed for printing errors in a different language.

\c_msg_coding_error_text_tl
\c_msg_fatal_text_tl
\c_msg_help_text_tl
\c_msg_kernel_bug_text_tl
\c_msg_kernel_bug_more_text_tl
\c_msg_no_info_text_tl
\c_msg_return_text_tl

Various pieces of text for use in messages, which are not changed by the code here although they could be to alter the language. Although these are marked as constants, they could be changed for printing errors in a different language.

\c_msg_on_line_tl

The 'on line' phrase for line numbers. Although marked as a constant, they could be changed for printing errors in a different language.

\c_msg_text_prefix_tl
\c_msg_more_text_prefix_tl

Header information for storing the 'paths' to parts of a message. Although these are marked as constants, they could be changed for printing errors in a different language.

\l_msg_class_tl
\l_msg_current_class_tl
\l_msg_current_module_tl

Information about message method, used for filtering.

\l_msg_names_clist

List of all of the message names defined.

\l_msg_redirect_classes_prop
\l_msg_redirect_names_prop

Re-direction lists containing the class of message to convert an different one.

\l_msg_redirect_classes_clist

List so that filtering does not loop.

Part XVIII

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_`.

83 Generic functions

`\box_new:N`
`\box_new:c`
`\box_new_local:N`
`\box_new_local:c`

`\box_new:N` ⟨*box*⟩

Defines ⟨*box*⟩ to be a new variable of type `box`. With the `local` variant, the variable is only available within the current group level (or below).

TEXhackers note: `\box_new:N` is the equivalent of plain TEX's `\newbox`.

`\if_hbox:N`
`\if_vbox:N`
`\if_box_empty:N`

`\if_hbox:N` ⟨*box*⟩ ⟨*true code*⟩`\else:` ⟨*false code*⟩`\fi:`
`\if_box_empty:N` ⟨*box*⟩ ⟨*true code*⟩`\else:` ⟨*false code*⟩`\fi:`

`\if_hbox:N` and `\if_vbox:N` check if ⟨*box*⟩ is an horizontal or vertical box resp. `\if_box_empty:N` tests if ⟨*box*⟩ is empty (void) and executes `code` according to the test outcome.

TEXhackers note: These are the TEX primitives `\ifhbox`, `\ifvbox` and `\ifvoid`.

`\box_if_horizontal_p:N`
`\box_if_horizontal_p:c`
`\box_if_horizontal:NTF`
`\box_if_horizontal:cTF`

`\box_if_horizontal:NTF` ⟨*box*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if ⟨*box*⟩ is an horizontal box and executes ⟨*code*⟩ accordingly.

\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_vertical:NTF
\box_if_vertical:cTF

\box_if_vertical:NTF ⟨*box*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*box*⟩ is a vertical box and executes ⟨*code*⟩ accordingly.

\box_if_empty_p:N
\box_if_empty_p:c
\box_if_empty:NTF
\box_if_empty:cTF

\box_if_empty:NTF ⟨*box*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*box*⟩ is empty (void) and executes `code` according to the test outcome.

TEXhackers note: `\box_if_empty:NTF` is the LaTeX3 function name for `\ifvoid`.

\box_set_eq:NN
\box_set_eq:cN
\box_set_eq:Nc
\box_set_eq:cc

\box_set_eq:NN ⟨*box*$_1$⟩ ⟨*box*$_2$⟩

Sets ⟨*box*$_1$⟩ equal to ⟨*box*$_2$⟩. Note that this eradicates the contents of ⟨*box*$_2$⟩ afterwards.

\box_gset_eq:NN
\box_gset_eq:cN
\box_gset_eq:Nc
\box_gset_eq:cc

\box_gset_eq:NN ⟨*box*$_1$⟩ ⟨*box*$_2$⟩

Globally sets ⟨*box*$_1$⟩ equal to ⟨*box*$_2$⟩.

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

\box_set_to_last:N ⟨*box*⟩

Sets ⟨*box*⟩ equal to the previous box `\l_last_box` and removes `\l_last_box` from the current list (unless in outer vertical or math mode).

\box_move_right:nn
\box_move_left:nn
\box_move_up:nn
\box_move_down:nn

\box_move_left:nn {⟨*dimen*⟩} {⟨*box function*⟩}

Moves ⟨*box function*⟩ ⟨*dimen*⟩ in the direction specified. ⟨*box function*⟩ is either an operation on a box such as `\box_use:N` or a “raw” box specification like `\vbox:n{xyz}`.

`\box_clear:N`
`\box_clear:c`
`\box_gclear:N`
`\box_gclear:c`

`\box_clear:N` ⟨*box*⟩

Clears ⟨*box*⟩ by setting it to the constant `\c_void_box`. `\box_gclear:N` does it globally.

`\box_use:N`
`\box_use:c`
`\box_use_clear:N`
`\box_use_clear:c`

`\box_use:N` ⟨*box*⟩
`\box_use_clear:N` ⟨*box*⟩

`\box_use:N` puts a copy of ⟨*box*⟩ on the current list while `\box_use_clear:N` puts the box on the current list and then eradicates the contents of it.

TeXhackers note: `\box_use:N` and `\box_use_clear:N` are the TeX primitives `\copy` and `\box` with new (descriptive) names.

`\box_ht:N`
`\box_ht:c`
`\box_dp:N`
`\box_dp:c`
`\box_wd:N`
`\box_wd:c`

`\box_ht:N` ⟨*box*⟩

Returns the height, depth, and width of ⟨*box*⟩ for use in dimension settings.

TeXhackers note: These are the TeX primitives `\ht`, `\dp` and `\wd`.

`\box_show:N`
`\box_show:c`

`\box_show:N` ⟨*box*⟩

Writes the contents of ⟨*box*⟩ to the log file.

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

`\c_empty_box`
`\l_tmpa_box`
`\l_tmpb_box`

`\c_empty_box` is the constantly empty box. The others are scratch boxes.

`\l_last_box`

`\l_last_box` is more or less a read-only box register managed by the engine. It denotes the last box on the current list if there is one, otherwise it is void. You can set other boxes to this box, with the result that the last box on the current list is removed at the same time (so it is with variable with side-effects).

84 Horizontal mode

`\hbox:n` `\hbox:n {⟨contents⟩}`

Places a `hbox` of natural size.

`\hbox_set:Nn`
`\hbox_set:cn`
`\hbox_gset:Nn`
`\hbox_gset:cn` `\hbox_set:Nn ⟨box⟩ {⟨contents⟩}`

Sets ⟨*box*⟩ to be a horizontal mode box containing ⟨*contents*⟩. It has it's natural size. `\hbox_gset:Nn` does it globally.

`\hbox_set_to_wd:Nnn`
`\hbox_set_to_wd:cnn`
`\hbox_gset_to_wd:Nnn`
`\hbox_gset_to_wd:cnn` `\hbox_set_to_wd:Nnn ⟨box⟩ {⟨dimen⟩} {⟨contents⟩}`

Sets ⟨*box*⟩ to contain ⟨*contents*⟩ and have width ⟨*dimen*⟩. `\hbox_gset_to_wd:Nn` does it globally.

`\hbox_to_wd:nn`
`\hbox_to_zero:n` `\hbox_to_wd:nn {⟨dimen⟩} ⟨contents⟩`
`\hbox_to_zero:n ⟨contents⟩`

Places a ⟨*box*⟩ of width ⟨*dimen*⟩ containing ⟨*contents*⟩. `\hbox_to_zero:n` is a shorthand for a width of zero.

`\hbox_overlap_left:n`
`\hbox_overlap_right:n` `\hbox_overlap_left:n ⟨contents⟩`

Places a ⟨*box*⟩ of width zero containing ⟨*contents*⟩ in a way the it overlaps with surrounding material (sticking out to the left or right).

`\hbox_set_inline_begin:N`
`\hbox_set_inline_begin:c`
`\hbox_set_inline_end:`
`\hbox_gset_inline_begin:N`
`\hbox_gset_inline_begin:c`
`\hbox_gset_inline_end:` `\hbox_set_inline_begin:N ⟨box⟩ ⟨contents⟩`
`\hbox_set_inline_end:`

Sets ⟨*box*⟩ to contain ⟨*contents*⟩. This type is useful for use in environment definitions.

`\hbox_unpack:N`
`\hbox_unpack:c`
`\hbox_unpack_clear:N`
`\hbox_unpack_clear:c`

`\hbox_unpack:N` ⟨*box*⟩

`\hbox_unpack:N` unpacks the contents of the ⟨*box*⟩ register and `\hbox_unpack_clear:N` also clears the ⟨*box*⟩ after unpacking it.

TEXhackers note: These are the TEX primitives `\unhcopy` and `\unhbox`.

85 Vertical mode

`\vbox:n`

`\vbox:n {`⟨*contents*⟩`}`

Places a `vbox` of natural size with baseline equal to the baseline of the last line in the box.

`\vbox_set:Nn`
`\vbox_set:cn`
`\vbox_gset:Nn`
`\vbox_gset:cn`

`\vbox_set:Nn` ⟨*box*⟩ `{`⟨*contents*⟩`}`

Sets ⟨*box*⟩ to be a vertical mode box containing ⟨*contents*⟩. It has its natural size. `\vbox_gset:Nn` does it globally.

`\vbox_set_to_ht:Nnn`
`\vbox_set_to_ht:cnn`
`\vbox_gset_to_ht:Nnn`
`\vbox_gset_to_ht:cnn`
`\vbox_gset_to_ht:ccn`

`\vbox_set_to_ht:Nnn` ⟨*box*⟩ `{`⟨*dimen*⟩`}` `{`⟨*contents*⟩`}`

Sets ⟨*box*⟩ to contain ⟨*contents*⟩ and have total height ⟨*dimen*⟩. `\vbox_gset_to_ht:Nn` does it globally.

`\vbox_set_inline_begin:N`
`\vbox_set_inline_end:`
`\vbox_gset_inline_begin:N`
`\vbox_gset_inline_end:`

`\vbox_set_inline_begin:N` ⟨*box*⟩ ⟨*contents*⟩
`\vbox_set_inline_end:`

Sets ⟨*box*⟩ to contain ⟨*contents*⟩. This type is useful for use in environment definitions.

\vbox_set_split_to_ht:NNn

\vbox_set_split_to_ht:NNn ⟨*box*$_1$⟩ ⟨*box*$_2$⟩ {⟨*dimen*⟩}

Sets ⟨*box*$_1$⟩ to contain the top ⟨*dimen*⟩ part of ⟨*box*$_2$⟩.

TeXhackers note: This is the TeX primitive \vsplit.

\vbox_to_ht:nn
\vbox_to_zero:n

\vbox_to_ht:nn {⟨*dimen*⟩} ⟨*contents*⟩
\vbox_to_zero:n ⟨*contents*⟩

Places a ⟨*box*⟩ of size ⟨*dimen*⟩ containing ⟨*contents*⟩.

\vbox_unpack:N
\vbox_unpack:c
\vbox_unpack_clear:N
\vbox_unpack_clear:c

\vbox_unpack:N ⟨*box*⟩

\vbox_unpack:N unpacks the contents of the ⟨*box*⟩ register and \vbox_unpack_clear:N also clears the ⟨*box*⟩ after unpacking it.

TeXhackers note: These are the TeX primitives \unvcopy and \unvbox.

Part XIX

The l3xref package Cross references

\xref_set_label:n

\xref_set_label:n {⟨*name*⟩}

Sets a label in the text. Note that this function does not do anything else than setting the correct labels. In particular, it does not try to fix any spacing around the write node; this is a task for the galley2 module.

\xref_new:nn

\xref_new:nn {⟨*type*⟩} {⟨*value*⟩}

Defines a new cross reference type ⟨*type*⟩. This defines the token list variable \l_xref_curr_⟨*type*⟩_tl with default value ⟨*value*⟩ which gets written fully expanded when \xref_set_label:n is called.

\xref_deferred_new:nn

\xref_deferred_new:nn {⟨*type*⟩} {⟨*value*⟩}

Same as \xref_new:n except for this one, the value written happens when TEX ships out the page. Page numbers use this one obviously.

\xref_get_value:nn ⋆ \xref_get_value:nn {⟨*type*⟩} {⟨*name*⟩}

Extracts the cross reference information of type ⟨*type*⟩ for the label ⟨*name*⟩. This operation is expandable.

Part XX

The l3keyval package Key-value parsing

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.

This module provides the low-level machinery for processing arbitrary key–value lists. The l3keys module provides a higher-level interface for managing run-time settings using key–value input, while other parts of LATEX3 also use key–value input based on l3keyval (for example the xtemplate module).

86 Features of l3keyval

As l3keyval is a low-level module, its functions are restricted to converting a ⟨*keyval list*⟩ into keys and values for further processing. Each key and value (or key alone) has to be processed further by a function provided when l3keyval is called. Typically, this will be *via* one of the \KV_process... functions:

```
\KV_process_space_removal_sanitize:NNn
  \my_processor_function_one:n
  \my_processor_function_two:nn
  { <keyval list> }
```

The two processor functions here handle the cases where there is only a key, and where there is both a key and value, respectively.

l3keyval parses key–value lists in a manner that does not double # tokens or expand any input. The module has processor functions which will sanitize the category codes of = and , tokens (for use in the document body) as well as faster versions which do not do this (for use inside code blocks). Spaces can be removed from each end of the key and value (again for the document body), again with faster code to be used where this is not necessary. Values which are wrapped in braces will have exactly one set removed, meaning that

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

and

```
key = value here,
```

are treated as identical (assuming that space removal is in force). l3keyval

87 Functions for keyval processing

The l3keyval module should be accessed *via* a small set of external functions. These correctly set up the module internals for use by other parts of LaTeX3.

In all cases, two functions have to be supplied by the programmer to apply to the items from the <keyval list> after l3keyval has separated out the entries. The first function should take one argument, and will receive the names of keys for which no value was supplied. The second function should take two arguments: a key name and the associated value.

`\KV_process_space_removal_sanitize:NNn`

`\KV_process_space_removal_sanitize:NNn` ⟨*function* $_1$⟩ ⟨*function* $_2$⟩ {⟨*keyval list*⟩}

Parses the ⟨*keyval list*⟩ splitting it into keys and associated values. Spaces are removed from the ends of both the key and value by this function, and the category codes of non-braced = and , tokens are normalised so that parsing is 'category code safe'. After parsing is completed, ⟨*function* $_1$⟩ is used to process keys without values and ⟨*function* $_2$⟩ deals with keys which have associated values.

`\KV_process_space_removal_no_sanitize:NNn`

`\KV_process_space_removal_no_sanitize:NNn` ⟨*function* $_1$⟩ ⟨*function* $_2$⟩ {⟨*keyval list*⟩}

Parses the ⟨*keyval list*⟩ splitting it into keys and associated values. Spaces are removed from the ends of both the key and value by this function, but category codes are not normalised. After parsing is completed, ⟨*function* $_1$⟩ is used to process keys without values and ⟨*function* $_2$⟩ deals with keys which have associated values.

`\KV_process_no_space_removal_no_sanitize:NNn`

`\KV_process_no_space_removal_no_sanitize:NNn` ⟨*function*$_1$⟩ ⟨*function*$_2$⟩ {⟨*keyval list*⟩}

Parses the ⟨*keyval list*⟩ splitting it into keys and associated values. Spaces are *not* removed from the ends of the key and value, and category codes are *not* normalised. After parsing is completed, ⟨*function*$_1$⟩ is used to process keys without values and ⟨*function*$_2$⟩ deals with keys which have associated values.

`\l_KV_remove_one_level_of_braces_bool` This boolean controls whether or not one level of braces is stripped from the key and value. The default value for this boolean is `true` so that exactly one level of braces is stripped. For certain applications it is desirable to keep the braces in which case the programmer just has to set the boolean false temporarily. Only applicable when spaces are being removed.

88 Internal functions

The remaining functions provided by l3keyval do not have any protection for nesting of one call to the module inside another. They should therefore not be called directly by other modules.

`\KV_parse_no_space_removal_no_sanitize:n` `\KV_parse_no_space_removal_no_sanitize:n` {⟨*keyval li*

Parses the keys and values, passing the results to `\KV_key_no_value_elt:n` and `\KV_key_value_elt:nn` as appropriate. Spaces are not removed in the parsing process and the category codes of `=` and `,` are not normalised.

`\KV_parse_space_removal_no_sanitize:n` `\KV_parse_space_removal_no_sanitize:n` {⟨*keyval list*⟩}

Parses the keys and values, passing the results to `\KV_key_no_value_elt:n` and `\KV_key_value_elt:nn` as appropriate. Spaces are removed in the parsing process from the ends of the key and value, but the category codes of `=` and `,` are not normalised.

`\KV_parse_space_removal_sanitize:n` `\KV_parse_space_removal_sanitize:n` {⟨*keyval list*⟩}

Parses the keys and values, passing the results to `\KV_key_no_value_elt:n` and `\KV_key_value_elt:nn` as appropriate. Spaces are removed in the parsing process from the ends of the key and value and the category codes of `=` and `,` are normalised at the outer level (*i.e.* only unbraced tokens are affected).

`\KV_key_no_value_elt:n`
`\KV_key_value_elt:nn`

`\KV_key_no_value_elt:n` {⟨*key*⟩}
`\KV_key_value_elt:n` {⟨*key*⟩} {⟨*value*⟩}

Used by \KV_parse... functions to further process keys with no values and keys with values, respectively. The standard definitions are error functions: the programmer should provide appropriate definitions for both at point of use.

89 Variables and constants

`\c_KV_single_equal_sign_tl` Constant token list to make finding = faster.

`\l_KV_tmpa_tl`
`\l_KV_tmpb_tl` Scratch token lists.

`\l_KV_parse_tl`
`\l_KV_currkey_tl`
`\l_KV_currval_tl` Token list variables for various parts of the parsed input.

Part XXI

The l3keys package Key–value support

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

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

or

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

For the programmer, the original keyval package gives only the most basic interface for this work. All key macros have to be created one at a time, and as a result the kvoptions and xkeyval packages have been written to extend the ease of creating keys. A very different approach has been provided by the pgfkeys package, which uses a key–value list to generate keys.

The l3keys package is aimed at creating a programming interface for key–value controls in LaTeX3. Keys are created using a key–value interface, in a similar manner to pgfkeys. Each key is created by setting one or more *properties* of the key:

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

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

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

At a document level, `\keys_set:nn` is used within a document function. For LaTeX 2$_\varepsilon$, a generic set up function could be created with

```
\newcommand*\SomePackageSetup[1]{%
  \@nameuse{keys_set:nn}{module}{#1}%
}
```

or to use key–value input as the optional argument for a macro:

```
\newcommand*\SomePackageMacro[2][]{%
  \begingroup
    \@nameuse{keys_set:nn}{module}{#1}%
    % Main code for \SomePackageMacro
  \endgroup
}
```

The same concepts using xparse for LaTeX3 use:

```
\DeclareDocumentCommand \SomePackageSetup { m } {
  \keys_set:nn { module } { #1 }
}
\DeclareDocumentCommand \SomePackageMacro { o m } {
  \group_begin:
    \keys_set:nn { module } { #1 }
    % Main code for \SomePackageMacro
  \group_end:
}
```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As will be discussed in section 91, it is suggested that the character '/' is reserved for sub-division of keys into logical groups. Macros are *not* expanded when creating key names, and so

```
\tl_set:Nn \l_module_tmp_tl { key }
\keys_define:nn { module } {
  \l_module_tmp_tl .code:n = code
}
```

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

90 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 `\keys_define:nn` function does not skip spaces in the input, and does not check the category codes for `,` and `=` tokens. This means that it is intended for use with code blocks and other environments where spaces are ignored.

`.bool_set:N`
`.bool_gset:N` ⟨*key*⟩ `.bool_set:N = ⟨bool⟩`

Defines ⟨*key*⟩ to set ⟨*bool*⟩ to ⟨*value*⟩ (which must be either `true` or `false`). Here, ⟨*bool*⟩ is a LaTeX3 boolean variable (*i.e.* created using `\bool_new:N`). If the variable does not exist, it will be created at the point that the key is set up.

`.choice:` ⟨*key*⟩ `.choice:`

Sets ⟨*key*⟩ to act as a multiple choice key. Each valid choice for ⟨*key*⟩ must then be created, as discussed in section 91.1.

.choice_code:n
.choice_code:x ⟨*key*⟩ .choice_code:n = ⟨*code*⟩

Stores ⟨*code*⟩ for use when .generate_choices:n creates one or more choice sub-keys of the current key. Inside ⟨*code*⟩, \l_keys_choice_tl contains the name of the choice made, and \l_keys_choice_int is the position of the choice in the list given to .generate_choices:n. Choices are discussed in detail in section 91.1.

.code:n
.code:x ⟨*key*⟩ .code:n = ⟨*code*⟩

Stores the ⟨*code*⟩ for execution when ⟨*key*⟩ is called. The ⟨*code*⟩ can include one parameter (#1), which will be the ⟨*value*⟩ given for the ⟨*key*⟩. The .code:x variant will expand ⟨*code*⟩ at the point where the ⟨*key*⟩ is created.

.default:n
.default:V ⟨*key*⟩ .default:n = ⟨*default*⟩

Creates a ⟨*default*⟩ value for ⟨*key*⟩, which is used if no value is given. This will be used if only the key name is given, but not if a blank ⟨*value*⟩ is given:

```
\keys_define:nn { module } {
  key .code:n    = Hello #1,
  key .default:n = World
}
\keys_set:nn { module} {
  key = Fred, % Prints 'Hello Fred'
  key,        % Prints 'Hello World'
  key = ,     % Prints 'Hello '
}
```

TEXhackers note: The ⟨*default*⟩ is stored as a token list variable, and therefore should not contain unescaped # tokens.

.dim_set:N
.dim_set:c
.dim_gset:N
.dim_gset:c ⟨*key*⟩ .dim_set:N = ⟨*dimension*⟩

Sets ⟨*key*⟩ to store the value it is given in ⟨*dimension*⟩. Here, ⟨*dimension*⟩ is a LaTeX3 dim variable (*i.e.* created using \dim_new:N) or a LaTeX 2_ε dimen (*i.e* created using \newdimen). If the variable does not exist, it will be created at the point that the key is set up.

.generate_choices:n ⟨*key*⟩ .generate_choices:n = ⟨*comma list*⟩

Makes ⟨*key*⟩ a multiple choice key, accepting the choices specified in ⟨*comma list*⟩. Each choice will execute code which should previously have been defined using `.choice_code:n` or `.choice_code:x`. Choices are discussed in detail in section 91.1.

`.int_set:N`
`.int_set:c`
`.int_gset:N`
`.int_gset:c`

⟨*key*⟩ `.int_set:N = ⟨integer⟩`

Sets ⟨*key*⟩ to store the value it is given in ⟨*integer*⟩. Here, ⟨*integer*⟩ is a LaTeX3 `int` variable (*i.e.* created using `\int_new:N`) or a LaTeX 2ε `count` (*i.e* created using `\newcount`). If the variable does not exist, it will be created at the point that the key is set up.

`.meta:n`
`.meta:x`

⟨*key*⟩ `.meta:n = ⟨multiple keys⟩`

Makes ⟨*key*⟩ a meta-key, which will set ⟨*multiple keys*⟩ in one go. If ⟨*key*⟩ is given with a value at the time the key is used, then the value will be passed through to the subsidiary ⟨*keys*⟩ for processing (as `#1`).

`.skip_set:N`
`.skip_set:c`
`.skip_gset:N`
`.skip_gset:c`

⟨*key*⟩ `.skip_set:N = ⟨skip⟩`

Sets ⟨*key*⟩ to store the value it is given in ⟨*skip*⟩, which is created if it does not already exist. Here, ⟨*skip*⟩ is a LaTeX3 `skip` variable (*i.e.* created using `\skip_new:N`) or a LaTeX 2ε `skip` (*i.e* created using `\newskip`). If the variable does not exist, it will be created at the point that the key is set up.

`.tl_set:N`
`.tl_set:c`
`.tl_set_x:N`
`.tl_set_x:c`
`.tl_gset:N`
`.tl_gset:c`
`.tl_gset_x:N`
`.tl_gset_x:c`

⟨*key*⟩ `.tl_set:N = ⟨token list variable⟩`

Sets ⟨*key*⟩ to store the value it is given in ⟨*token list variable*⟩, which is created if it does not already exist. Here, ⟨*token list variable*⟩ is a LaTeX3 `tl` variable (*i.e.* created using `\tl_new:N`) or a LaTeX 2ε macro with no arguments (*i.e.* created using `\newcommand` or `\def`). If the variable does not exist, it will be created at the point that the key is set up. The `x` variants perform an `x` expansion at the time the ⟨*value*⟩ passed to the ⟨*key*⟩ is saved to the ⟨*token list variable*⟩.

`.value_forbidden:`
`.value_required:`

⟨*key*⟩ `.value_forbidden:`

Flags for forbidding and requiring a ⟨*value*⟩ for ⟨*key*⟩. Giving a ⟨*value*⟩ for a ⟨*key*⟩ which has the `.value_forbidden:` property set will result in an error. In the same way, if a ⟨*key*⟩ has the `.value_required:` property set then a ⟨*value*⟩ must be given when the ⟨*key*⟩ is used.

91 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 { module } {
  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.

91.1 Multiple choices

Multiple choices are created by setting the `.choice:` property:

```
\keys_define:nn { module } {
  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 choices. Here, the keys can share the same code, and can be rapidly created using the `.choice_code:n` and `.generate_choices:n` properties:

```
\keys_define:nn { module } {
  key .choice_code:n      = {
```

```
    You~gave~choice~‘‘\int_use:N \l_keys_choice_tl’’,~
    which~is~in~position~
    \int_use:N\l_keys_choice_int\space
    in~the~list.
  },
  key .generate_choices:n = {
    choice-a, choice-b, choice-c
  }
}
```

Following common computing practice, `\l_keys_choice_int` is indexed from 0 (as an offset), so that the value of `\l_keys_choice_int` for the first choice in a list will be zero. This means that `\l_keys_choice_int` can be used directly with `\if_case:w` and so on.

`\l_keys_choice_int`
`\l_keys_choice_tl`

Inside the code block for a choice generated using `.generate_choice:`, 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 0.

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 { module } {
  key .choice:n,
  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 `.generate_choices:n` (*i.e.* anything might happen!).

92 Setting keys

`\keys_set:nn`
`\keys_set:nV`
`\keys_set:nv`

`\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. In contrast to \keys_define:nn, this function does check category codes and ignore spaces, and is therefore suitable for user input.

If a key is not known, \keys_set:nn will look for a special unknown key for the same module. This mechanism can be used to create new keys from user input.

```
\keys_define:nn { module } {
  unknown .code:n =
    You~tried~to~set~key~'\l_keys_path_tl'~to~'#1'
}
```

\l_keys_key_tl When processing an unknown key, the name of the key is available as \l_keys_key_tl. Note that this will have been processed using \tl_to_str:N. The value passed to the key (if any) is available as the macro parameter #1.

92.1 Examining keys: internal representation

\keys_if_exist:nn*TF* \keys_if_exist:nnTF {⟨*module*⟩} {⟨*key*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*key*⟩ exists for ⟨*module*⟩, *i.e.* if any code has been defined for ⟨*key*⟩.

TEXhackers note: The function works by testing for the existence of the internal function \keys > ⟨*module*⟩/⟨*key*⟩.cmd:n.

\keys_show:nn \keys_show:nn {⟨*module*⟩} {⟨*key*⟩}

Shows the internal representation of a ⟨*key*⟩.

TEXhackers note: Keys are stored as functions with names of the format \keys > ⟨*module*⟩/⟨*key*⟩.cmd:n.

93 Internal functions

\keys_bool_set:NN \keys_bool_set:NN ⟨*bool*⟩ ⟨*scope*⟩

Creates code to set ⟨*bool*⟩ when ⟨*key*⟩ is given, with setting using ⟨*scope*⟩ (l or g for local or global, respectively). ⟨*bool*⟩ should be a LaTeX3 boolean variable.

\keys_choice_code_store:x \keys_choice_code_store:x ⟨*code*⟩

Stores ⟨*code*⟩ for later use by .generate_code:n.

\keys_choice_make: `\keys_choice_make:`

Makes ⟨*key*⟩ a choice key.

\keys_choices_generate:n `\keys_choices_generate:n {`⟨*comma list*⟩`}`

Makes ⟨*comma list*⟩ choices for ⟨*key*⟩.

\keys_choice_find:n `\keys_choice_find:n {`⟨*choice*⟩`}`

Searches for ⟨*choice*⟩ as a sub-key of ⟨*key*⟩.

\keys_cmd_set:nn
\keys_cmd_set:nx `\keys_cmd_set:nn {`⟨*path*⟩`} {`⟨*code*⟩`}`

Creates a function for ⟨*path*⟩ using ⟨*code*⟩.

\keys_default_set:n
\keys_default_set:V `\keys_default_set:n {`⟨*default*⟩`}`

Sets ⟨*default*⟩ for ⟨*key*⟩.

\keys_define_elt:n
\keys_define_elt:nn `\keys_define_elt:nn {`⟨*key*⟩`} {`⟨*value*⟩`}`

Processing functions for key–value pairs when defining keys.

\keys_define_key:n `\keys_define_key:n {`⟨*key*⟩`}`

Defines ⟨*key*⟩.

\keys_execute: `\keys_execute:`

Executes ⟨*key*⟩ (where the name of the ⟨*key*⟩ will be stored internally).

\keys_execute_unknown: `\keys_execute_unknown:`

Handles unknown ⟨*key*⟩ names.

\keys_if_value_requirement:nTF ⋆ `\keys_if_value_requirement:nTF {`⟨*requirement*⟩`}`
`{`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Check if ⟨*requirement*⟩ applies to ⟨*key*⟩.

\keys_meta_make:n
\keys_meta_make:x `\keys_meta_make:n {`⟨*keys*⟩`}`

Makes ⟨*key*⟩ a meta-key to set ⟨*keys*⟩.

`\keys_property_find:n` \keys_property_find:n {⟨*key*⟩}

Separates ⟨*key*⟩ from ⟨*property*⟩.

`\keys_property_new:nn`
`\keys_property_new_arg:nn` \keys_property_new:nn {⟨*property*⟩} {⟨*code*⟩}

Makes a new ⟨*property*⟩ expanding to ⟨*code*⟩. The `arg` version makes properties with one argument.

`\keys_property_undefine:n` \keys_property_undefine:n {⟨*property*⟩}

Deletes ⟨*property*⟩ of ⟨*key*⟩.

`\keys_set_elt:n`
`\keys_set_elt:nn` \keys_set_elt:nn {⟨*key*⟩} {⟨*value*⟩}

Processing functions for key–value pairs when setting keys.

`\keys_tmp:w` \keys_tmp:w ⟨*args*⟩

Used to store ⟨*code*⟩ to execute a ⟨*key*⟩.

`\keys_value_or_default:n` \keys_value_or_default:n {⟨*value*⟩}

Sets `\l_keys_value_tl` to ⟨*value*⟩, or ⟨*default*⟩ if ⟨*value*⟩ was not given and if ⟨*default*⟩ is available.

`\keys_value_requirement:n` \keys_value_requirement:n {⟨*requirement*⟩}

Sets ⟨*key*⟩ to have ⟨*requirement*⟩ concerning ⟨*value*⟩.

`\keys_variable_set:NnNN`
`\keys_variable_set:cnNN` \keys_variable_set:NnNN ⟨*var*⟩ ⟨*type*⟩ ⟨*scope*⟩ ⟨*expansion*⟩

Sets ⟨*key*⟩ to assign ⟨*value*⟩ to ⟨*variable*⟩. The ⟨*scope*⟩ (blank for local, `g` for global) and ⟨*type*⟩ (`tl`, `int`, etc.) are given explicitly.

94 Variables and constants

`\c_keys_properties_root_tl`
`\c_keys_root_tl` The root paths for keys and properties, used to generate the names of the functions which store these items.

`\c_keys_value_forbidden_tl`
`\c_keys_value_required_tl` Marker text containers: by storing the values the code can make comparisons slightly faster.

`\l_keys_choice_code_tl` Used to transfer code from storage when making multiple choices.

`\l_keys_module_tl`
`\l_keys_path_tl`
`\l_keys_property_tl` Various key paths need to be stored. These are flexible items that are set during the key reading process.

`\l_keys_no_value_bool` A marker for 'no value' as key input.

`\l_keys_value_tl` Holds the currently supplied value, in a token register as there may be `#` tokens.

Part XXII

The l3calc package Infix notation arithmetic in LaTeX3

This is pretty much a straight adaption of the calc package and as such has same syntax for the ⟨*calc expression*⟩. However, there are some noticeable differences.

- The calc expression is expanded fully, which means there are no problems with unfinished conditionals. However, the contents of `\widthof` etc. is not expanded at all. This includes uses in traditional LaTeX as in the array package, which tries to do an `\edef` several times. The code used in l3calc provides self-protection for these cases.

- Muskip registers are supported although they can only be used in `\ratio` if already evaluating a muskip expression. For the other three register types, you can use points.
- All results are rounded, not truncated. More precisely, the primitive TeX operations `\divide` and `\multiply` are not used. The only instance where one will observe an effect is when dividing integers.

This version of l3calc is a now a complete replacement for the original calc package providing the same functionality and will prevent the original calc package from loading.

95 User functions

```
\maxof
\minof
\widthof
\heightof
\depthof
\totalheightof
\ratio
\real
\setlength
\gsetlength
\addtolength
\gaddtolength
\setcounter
\addtocounter
\stepcounter
```

See documentation for LaTeX 2_ε package calc.

```
\calc_maxof:nn
\calc_minof:nn
\calc_widthof:n
\calc_heightof:n
\calc_depthof:n
\calc_totalheightof:n
\calc_ratio:nn
\calc_real:n
\calc_setcounter:nn
\calc_addtocounter:nn
\calc_stepcounter:n
```

Equivalent commands as the above in the expl3 namespace.

\calc_int_set:Nn
\calc_int_gset:Nn
\calc_int_add:Nn
\calc_int_gadd:Nn
\calc_int_sub:Nn
\calc_int_gsub:Nn

\calc_int_set:Nn ⟨*int*⟩ {⟨*calc expression*⟩}

Evaluates ⟨*calc expression*⟩ and either adds or subtracts it from ⟨*int*⟩ or sets ⟨*int*⟩ to it. These operations can also be global.

\calc_dim_set:Nn
\calc_dim_gset:Nn
\calc_dim_add:Nn
\calc_dim_gadd:Nn
\calc_dim_sub:Nn
\calc_dim_gsub:Nn

\calc_dim_set:Nn ⟨*dim*⟩ {⟨*calc expression*⟩}

Evaluates ⟨*calc expression*⟩ and either adds or subtracts it from ⟨*dim*⟩ or sets ⟨*dim*⟩ to it. These operations can also be global.

\calc_skip_set:Nn
\calc_skip_gset:Nn
\calc_skip_add:Nn
\calc_skip_gadd:Nn
\calc_skip_sub:Nn
\calc_skip_gsub:Nn

\calc_skip_set:Nn ⟨*skip*⟩ {⟨*calc expression*⟩}

Evaluates ⟨*calc expression*⟩ and either adds or subtracts it from ⟨*skip*⟩ or sets ⟨*skip*⟩ to it. These operations can also be global.

\calc_muskip_set:Nn
\calc_muskip_gset:Nn
\calc_muskip_add:Nn
\calc_muskip_gadd:Nn
\calc_muskip_sub:Nn
\calc_muskip_gsub:Nn

\calc_muskip_set:Nn ⟨*muskip*⟩ {⟨*calc expression*⟩}

Evaluates ⟨*calc expression*⟩ and either adds or subtracts it from ⟨*muskip*⟩ or sets ⟨*muskip*⟩ to it. These operations can also be global.

\calc_calculate_box_size:nnn

\calc_calculate_box_size:nnn {⟨*dim-set*⟩} {⟨$item_1$⟩ ⟨$item_2$⟩ ... ⟨*item n*⟩} {⟨*contents*⟩}

Sets ⟨*contents*⟩ in a temporary box `\l_tmpa_box`. Then ⟨*dim-set*⟩ is put in front of a loop that inserts +⟨$item_i$⟩ in front of `\l_tmpa_box` and this is evaluated. For instance, if we wanted to determine the total height of the text `xyz` and store it in `\l_tmpa_dim`, we would call it as.

```
\calc_calculate_box_size:nnn
  {\dim_set:Nn\l_tmpa_dim}{\box_ht:N\box_dp:N}{xyz}
```

Similarly, if we wanted the difference between height and depth, we could call it as

```
\calc_calculate_box_size:nnn
  {\dim_set:Nn\l_tmpa_dim}{\box_ht:N{-\box_dp:N}}{xyz}
```

Part XXIII

The l3file package File Loading

96 Loading files

In contrast to the l3io module, which deals with the lowest level of file management, the l3file module provides a higher level interface for handling file contents. This involves providing convenient wrappers around many of the functions in l3io to make them more generally accessible.

It is important to remember that TEX will attempt to locate files using 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.

`\g_file_current_name_tl` Contains the name of the current LaTeX file. This variable should not be modified: it is intended for information only. It will be equal to `\c_job_name_tl` at the start of a LATEX run and will be modified each time a file is read using `\file_input:n`.

`\file_if_exist:nTF`
`\file_if_exist:VTF`

`\file_if_exist:nTF {⟨file name⟩} {⟨true code⟩} {⟨false code⟩}`

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by `\file_path_include:n`). The branching versions then leave either ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

`\file_input:n`
`\file_input:V`

`\file_input:n {⟨file name⟩}`

Searches for ⟨*file name*⟩ in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional LaTeX source. All files read are recorded for information and the file name stack is updated by this function.

`\file_path_include:n`

`\file_path_include:n {⟨path⟩}`

Adds ⟨*path*⟩ to the list of those used to search for files by the `\file_input:n` and `\file_if_exist:n` function. The assignment is local.

`\file_path_remove:n`

`\file_path_remove:n {⟨path⟩}`

Removes ⟨*path*⟩ from the list of those used to search for files by the `\file_input:n` and `\file_if_exist:n` function. The assignment is local.

`\file_list:`

`\file_list:`

This function will list all files loaded using `\file_input:n` in the log file.

Part XXIV

Implementation

97 l3names implementation

This is the base part of LaTeX3 defining things like `catcode`s and redefining the TeX primitives, as well as setting up the code to load expl3 modules in LaTeX 2_ε.

97.1 Internal functions

`\ExplSyntaxStatus`
`\ExplSyntaxPopStack`
`\ExplSyntaxStack`

Functions used to track the state of the catcode regime.

`\@pushfilename`
`\@popfilename`

Re-definitions of LaTeX's file-loading functions to support `\ExplSyntax`.

97.2 Package loading

Before anything else, check that we're using ε-TeX; no point continuing otherwise.

```
⟨*initex | package⟩
\begingroup
\def\firstoftwo#1#2{#1}
\def\secondoftwo#1#2{#2}
\def\etexmissingerror{Not running under e-TeX}
\def\etexmissinghelp{%
  This package requires e-TeX.^^J%
  Try compiling the document with ‘elatex’ instead of ‘latex’.^^J%
  When using pdfTeX, try ‘pdfelatex’ instead of ‘pdflatex’%
}%
\expandafter\ifx\csname eTeXversion\endcsname\relax
  \expandafter\secondoftwo\else\expandafter\firstoftwo\fi
    {\endgroup}{%
⟨initex⟩      \expandafter\errhelp\expandafter{\etexmissinghelp}%
⟨initex⟩      \expandafter\errmessage\expandafter{\etexmissingerror}%
⟨package⟩      \PackageError{l3names}{\etexmissingerror}{\etexmissinghelp}%
      \endgroup
      \endinput
    }
⟨/initex | package⟩
```

97.3 Catcode assignments

Catcodes for begingroup, endgroup, macro parameter, superscript, and tab, are all assigned before the start of the documented code. (See the beginning of `l3names.dtx`.)

Reason for `\endlinechar=32` is that a line ending with a backslash will be interpreted as the token `\␣` which seems most natural and since spaces are ignored it works as we intend elsewhere.

Before we do this we must however record the settings for the catcode regime as it was when we start changing it.

```
⟨*initex | package⟩
\protected\edef\ExplSyntaxOff{
  \unexpanded{\ifodd \ExplSyntaxStatus\relax
  \def\ExplSyntaxStatus{0}
  }
  \catcode  126=\the \catcode 126 \relax
  \catcode  32=\the \catcode 32 \relax
  \catcode  9=\the \catcode 9 \relax
  \endlinechar  =\the \endlinechar \relax
  \catcode  95=\the \catcode 95 \relax
  \catcode  58=\the \catcode 58 \relax
  \noexpand\fi
```

```
}
\catcode126=10\relax  % tilde is a space char.
\catcode32=9\relax    % space is ignored
\catcode9=9\relax     % tab also ignored
\endlinechar=32\relax % endline is space
\catcode95=11\relax   % underscore letter
\catcode58=11\relax   % colon letter
```

97.4 Setting up primitive names

Here is the function that renames TeX's primitives.

Normally the old name is left untouched, but the possibility of undefining the original names is made available by docstrip and package options. If nothing else, this gives a way of checking what 'old code' a package depends on...

If the package option 'removeoldnames' is used then some trick code is run after the end of this file, to skip past the code which has been inserted by LaTeX 2ε to manage the file name stack, this code would break if run once the TeX primitives have been undefined. (What a surprise!) **The option has been temporarily disabled.**

To get things started, give a new name for `\let`.

```
\let \tex_let:D \let
⟨/initex | package⟩
```

and now an internal function to possibly remove the old name: for the moment.

```
⟨*initex⟩
\long \def \name_undefine:N #1 {
  \tex_let:D #1 \c_undefined
}
⟨/initex⟩

⟨*package⟩
\DeclareOption{removeoldnames}{
  \long\def\name_undefine:N#1{
    \tex_let:D#1\c_undefined}}

\DeclareOption{keepoldnames}{
  \long\def\name_undefine:N#1{}}

\ExecuteOptions{keepoldnames}

\ProcessOptions
⟨/package⟩
```

The internal function to give the new name and possibly undefine the old name.

```
⟨*initex | package⟩
```

```
\long \def \name_primitive:NN #1#2 {
  \tex_let:D #2 #1
  \name_undefine:N #1
}
```

97.5 Reassignment of primitives

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.

```
\name_primitive:NN \                    \tex_space:D
\name_primitive:NN \/                   \tex_italiccor:D
\name_primitive:NN \-                   \tex_hyphen:D
```

Now all the other primitives.

```
\name_primitive:NN \let                 \tex_let:D
\name_primitive:NN \def                 \tex_def:D
\name_primitive:NN \edef                \tex_edef:D
\name_primitive:NN \gdef                \tex_gdef:D
\name_primitive:NN \xdef                \tex_xdef:D
\name_primitive:NN \chardef             \tex_chardef:D
\name_primitive:NN \countdef            \tex_countdef:D
\name_primitive:NN \dimendef            \tex_dimendef:D
\name_primitive:NN \skipdef             \tex_skipdef:D
\name_primitive:NN \muskipdef           \tex_muskipdef:D
\name_primitive:NN \mathchardef         \tex_mathchardef:D
\name_primitive:NN \toksdef             \tex_toksdef:D
\name_primitive:NN \futurelet           \tex_futurelet:D
\name_primitive:NN \advance             \tex_advance:D
\name_primitive:NN \divide              \tex_divide:D
\name_primitive:NN \multiply            \tex_multiply:D
\name_primitive:NN \font                \tex_font:D
\name_primitive:NN \fam                 \tex_fam:D
\name_primitive:NN \global              \tex_global:D
\name_primitive:NN \long                \tex_long:D
\name_primitive:NN \outer               \tex_outer:D
\name_primitive:NN \setlanguage         \tex_setlanguage:D
\name_primitive:NN \globaldefs          \tex_globaldefs:D
\name_primitive:NN \afterassignment     \tex_afterassignment:D
\name_primitive:NN \aftergroup          \tex_aftergroup:D
\name_primitive:NN \expandafter         \tex_expandafter:D
\name_primitive:NN \noexpand            \tex_noexpand:D
\name_primitive:NN \begingroup          \tex_begingroup:D
\name_primitive:NN \endgroup            \tex_endgroup:D
\name_primitive:NN \halign              \tex_halign:D
\name_primitive:NN \valign              \tex_valign:D
\name_primitive:NN \cr                  \tex_cr:D
```

```
\name_primitive:NN \crcr                  \tex_crcr:D
\name_primitive:NN \noalign               \tex_noalign:D
\name_primitive:NN \omit                  \tex_omit:D
\name_primitive:NN \span                  \tex_span:D
\name_primitive:NN \tabskip               \tex_tabskip:D
\name_primitive:NN \everycr               \tex_everycr:D
\name_primitive:NN \if                    \tex_if:D
\name_primitive:NN \ifcase                \tex_ifcase:D
\name_primitive:NN \ifcat                 \tex_ifcat:D
\name_primitive:NN \ifnum                 \tex_ifnum:D
\name_primitive:NN \ifodd                 \tex_ifodd:D
\name_primitive:NN \ifdim                 \tex_ifdim:D
\name_primitive:NN \ifeof                 \tex_ifeof:D
\name_primitive:NN \ifhbox                \tex_ifhbox:D
\name_primitive:NN \ifvbox                \tex_ifvbox:D
\name_primitive:NN \ifvoid                \tex_ifvoid:D
\name_primitive:NN \ifx                   \tex_ifx:D
\name_primitive:NN \iffalse               \tex_iffalse:D
\name_primitive:NN \iftrue                \tex_iftrue:D
\name_primitive:NN \ifhmode               \tex_ifhmode:D
\name_primitive:NN \ifmmode               \tex_ifmmode:D
\name_primitive:NN \ifvmode               \tex_ifvmode:D
\name_primitive:NN \ifinner               \tex_ifinner:D
\name_primitive:NN \else                  \tex_else:D
\name_primitive:NN \fi                    \tex_fi:D
\name_primitive:NN \or                    \tex_or:D
\name_primitive:NN \immediate             \tex_immediate:D
\name_primitive:NN \closeout              \tex_closeout:D
\name_primitive:NN \openin                \tex_openin:D
\name_primitive:NN \openout               \tex_openout:D
\name_primitive:NN \read                  \tex_read:D
\name_primitive:NN \write                 \tex_write:D
\name_primitive:NN \closein               \tex_closein:D
\name_primitive:NN \newlinechar           \tex_newlinechar:D
\name_primitive:NN \input                 \tex_input:D
\name_primitive:NN \endinput              \tex_endinput:D
\name_primitive:NN \inputlineno           \tex_inputlineno:D
\name_primitive:NN \errmessage            \tex_errmessage:D
\name_primitive:NN \message               \tex_message:D
\name_primitive:NN \show                  \tex_show:D
\name_primitive:NN \showthe               \tex_showthe:D
\name_primitive:NN \showbox               \tex_showbox:D
\name_primitive:NN \showlists             \tex_showlists:D
\name_primitive:NN \errhelp               \tex_errhelp:D
\name_primitive:NN \errorcontextlines     \tex_errorcontextlines:D
\name_primitive:NN \tracingcommands       \tex_tracingcommands:D
\name_primitive:NN \tracinglostchars      \tex_tracinglostchars:D
\name_primitive:NN \tracingmacros         \tex_tracingmacros:D
\name_primitive:NN \tracingonline         \tex_tracingonline:D
\name_primitive:NN \tracingoutput         \tex_tracingoutput:D
```

```
\name_primitive:NN \tracingpages          \tex_tracingpages:D
\name_primitive:NN \tracingparagraphs     \tex_tracingparagraphs:D
\name_primitive:NN \tracingrestores       \tex_tracingrestores:D
\name_primitive:NN \tracingstats          \tex_tracingstats:D
\name_primitive:NN \pausing               \tex_pausing:D
\name_primitive:NN \showboxbreadth        \tex_showboxbreadth:D
\name_primitive:NN \showboxdepth          \tex_showboxdepth:D
\name_primitive:NN \batchmode             \tex_batchmode:D
\name_primitive:NN \errorstopmode         \tex_errorstopmode:D
\name_primitive:NN \nonstopmode           \tex_nonstopmode:D
\name_primitive:NN \scrollmode            \tex_scrollmode:D
\name_primitive:NN \end                   \tex_end:D
\name_primitive:NN \csname                \tex_csname:D
\name_primitive:NN \endcsname             \tex_endcsname:D
\name_primitive:NN \ignorespaces          \tex_ignorespaces:D
\name_primitive:NN \relax                 \tex_relax:D
\name_primitive:NN \the                   \tex_the:D
\name_primitive:NN \mag                   \tex_mag:D
\name_primitive:NN \language              \tex_language:D
\name_primitive:NN \mark                  \tex_mark:D
\name_primitive:NN \topmark               \tex_topmark:D
\name_primitive:NN \firstmark             \tex_firstmark:D
\name_primitive:NN \botmark               \tex_botmark:D
\name_primitive:NN \splitfirstmark        \tex_splitfirstmark:D
\name_primitive:NN \splitbotmark          \tex_splitbotmark:D
\name_primitive:NN \fontname              \tex_fontname:D
\name_primitive:NN \escapechar            \tex_escapechar:D
\name_primitive:NN \endlinechar           \tex_endlinechar:D
\name_primitive:NN \mathchoice            \tex_mathchoice:D
\name_primitive:NN \delimiter             \tex_delimiter:D
\name_primitive:NN \mathaccent            \tex_mathaccent:D
\name_primitive:NN \mathchar              \tex_mathchar:D
\name_primitive:NN \mskip                 \tex_mskip:D
\name_primitive:NN \radical               \tex_radical:D
\name_primitive:NN \vcenter               \tex_vcenter:D
\name_primitive:NN \mkern                 \tex_mkern:D
\name_primitive:NN \above                 \tex_above:D
\name_primitive:NN \abovewithdelims       \tex_abovewithdelims:D
\name_primitive:NN \atop                  \tex_atop:D
\name_primitive:NN \atopwithdelims        \tex_atopwithdelims:D
\name_primitive:NN \over                  \tex_over:D
\name_primitive:NN \overwithdelims        \tex_overwithdelims:D
\name_primitive:NN \displaystyle          \tex_displaystyle:D
\name_primitive:NN \textstyle             \tex_textstyle:D
\name_primitive:NN \scriptstyle           \tex_scriptstyle:D
\name_primitive:NN \scriptscriptstyle     \tex_scriptscriptstyle:D
\name_primitive:NN \nonscript             \tex_nonscript:D
\name_primitive:NN \eqno                  \tex_eqno:D
\name_primitive:NN \leqno                 \tex_leqno:D
\name_primitive:NN \abovedisplayshortskip \tex_abovedisplayshortskip:D
```

```
\name_primitive:NN \abovedisplayskip      \tex_abovedisplayskip:D
\name_primitive:NN \belowdisplayshortskip \tex_belowdisplayshortskip:D
\name_primitive:NN \belowdisplayskip      \tex_belowdisplayskip:D
\name_primitive:NN \displaywidowpenalty   \tex_displaywidowpenalty:D
\name_primitive:NN \displayindent         \tex_displayindent:D
\name_primitive:NN \displaywidth          \tex_displaywidth:D
\name_primitive:NN \everydisplay          \tex_everydisplay:D
\name_primitive:NN \predisplaysize        \tex_predisplaysize:D
\name_primitive:NN \predisplaypenalty     \tex_predisplaypenalty:D
\name_primitive:NN \postdisplaypenalty    \tex_postdisplaypenalty:D
\name_primitive:NN \mathbin               \tex_mathbin:D
\name_primitive:NN \mathclose             \tex_mathclose:D
\name_primitive:NN \mathinner             \tex_mathinner:D
\name_primitive:NN \mathop                \tex_mathop:D
\name_primitive:NN \displaylimits         \tex_displaylimits:D
\name_primitive:NN \limits                \tex_limits:D
\name_primitive:NN \nolimits              \tex_nolimits:D
\name_primitive:NN \mathopen              \tex_mathopen:D
\name_primitive:NN \mathord               \tex_mathord:D
\name_primitive:NN \mathpunct             \tex_mathpunct:D
\name_primitive:NN \mathrel               \tex_mathrel:D
\name_primitive:NN \overline              \tex_overline:D
\name_primitive:NN \underline             \tex_underline:D
\name_primitive:NN \left                  \tex_left:D
\name_primitive:NN \right                 \tex_right:D
\name_primitive:NN \binoppenalty          \tex_binoppenalty:D
\name_primitive:NN \relpenalty            \tex_relpenalty:D
\name_primitive:NN \delimitershortfall    \tex_delimitershortfall:D
\name_primitive:NN \delimiterfactor       \tex_delimiterfactor:D
\name_primitive:NN \nulldelimiterspace    \tex_nulldelimiterspace:D
\name_primitive:NN \everymath             \tex_everymath:D
\name_primitive:NN \mathsurround          \tex_mathsurround:D
\name_primitive:NN \medmuskip             \tex_medmuskip:D
\name_primitive:NN \thinmuskip            \tex_thinmuskip:D
\name_primitive:NN \thickmuskip           \tex_thickmuskip:D
\name_primitive:NN \scriptspace           \tex_scriptspace:D
\name_primitive:NN \noboundary            \tex_noboundary:D
\name_primitive:NN \accent                \tex_accent:D
\name_primitive:NN \char                  \tex_char:D
\name_primitive:NN \discretionary         \tex_discretionary:D
\name_primitive:NN \hfil                  \tex_hfil:D
\name_primitive:NN \hfilneg               \tex_hfilneg:D
\name_primitive:NN \hfill                 \tex_hfill:D
\name_primitive:NN \hskip                 \tex_hskip:D
\name_primitive:NN \hss                   \tex_hss:D
\name_primitive:NN \vfil                  \tex_vfil:D
\name_primitive:NN \vfilneg               \tex_vfilneg:D
\name_primitive:NN \vfill                 \tex_vfill:D
\name_primitive:NN \vskip                 \tex_vskip:D
\name_primitive:NN \vss                   \tex_vss:D
```

```
\name_primitive:NN \unskip               \tex_unskip:D
\name_primitive:NN \kern                 \tex_kern:D
\name_primitive:NN \unkern               \tex_unkern:D
\name_primitive:NN \hrule                \tex_hrule:D
\name_primitive:NN \vrule                \tex_vrule:D
\name_primitive:NN \leaders              \tex_leaders:D
\name_primitive:NN \cleaders             \tex_cleaders:D
\name_primitive:NN \xleaders             \tex_xleaders:D
\name_primitive:NN \lastkern             \tex_lastkern:D
\name_primitive:NN \lastskip             \tex_lastskip:D
\name_primitive:NN \indent               \tex_indent:D
\name_primitive:NN \par                  \tex_par:D
\name_primitive:NN \noindent             \tex_noindent:D
\name_primitive:NN \vadjust              \tex_vadjust:D
\name_primitive:NN \baselineskip         \tex_baselineskip:D
\name_primitive:NN \lineskip             \tex_lineskip:D
\name_primitive:NN \lineskiplimit        \tex_lineskiplimit:D
\name_primitive:NN \clubpenalty          \tex_clubpenalty:D
\name_primitive:NN \widowpenalty         \tex_widowpenalty:D
\name_primitive:NN \exhyphenpenalty      \tex_exhyphenpenalty:D
\name_primitive:NN \hyphenpenalty        \tex_hyphenpenalty:D
\name_primitive:NN \linepenalty          \tex_linepenalty:D
\name_primitive:NN \doublehyphendemerits \tex_doublehyphendemerits:D
\name_primitive:NN \finalhyphendemerits  \tex_finalhyphendemerits:D
\name_primitive:NN \adjdemerits          \tex_adjdemerits:D
\name_primitive:NN \hangafter            \tex_hangafter:D
\name_primitive:NN \hangindent           \tex_hangindent:D
\name_primitive:NN \parshape             \tex_parshape:D
\name_primitive:NN \hsize                \tex_hsize:D
\name_primitive:NN \lefthyphenmin        \tex_lefthyphenmin:D
\name_primitive:NN \righthyphenmin       \tex_righthyphenmin:D
\name_primitive:NN \leftskip             \tex_leftskip:D
\name_primitive:NN \rightskip            \tex_rightskip:D
\name_primitive:NN \looseness            \tex_looseness:D
\name_primitive:NN \parskip              \tex_parskip:D
\name_primitive:NN \parindent            \tex_parindent:D
\name_primitive:NN \uchyph               \tex_uchyph:D
\name_primitive:NN \emergencystretch     \tex_emergencystretch:D
\name_primitive:NN \pretolerance         \tex_pretolerance:D
\name_primitive:NN \tolerance            \tex_tolerance:D
\name_primitive:NN \spaceskip            \tex_spaceskip:D
\name_primitive:NN \xspaceskip           \tex_xspaceskip:D
\name_primitive:NN \parfillskip          \tex_parfillskip:D
\name_primitive:NN \everypar             \tex_everypar:D
\name_primitive:NN \prevgraf             \tex_prevgraf:D
\name_primitive:NN \spacefactor          \tex_spacefactor:D
\name_primitive:NN \shipout              \tex_shipout:D
\name_primitive:NN \vsize                \tex_vsize:D
\name_primitive:NN \interlinepenalty     \tex_interlinepenalty:D
\name_primitive:NN \brokenpenalty        \tex_brokenpenalty:D
```

```
\name_primitive:NN \topskip                \tex_topskip:D
\name_primitive:NN \maxdeadcycles          \tex_maxdeadcycles:D
\name_primitive:NN \maxdepth               \tex_maxdepth:D
\name_primitive:NN \output                 \tex_output:D
\name_primitive:NN \deadcycles             \tex_deadcycles:D
\name_primitive:NN \pagedepth              \tex_pagedepth:D
\name_primitive:NN \pagestretch            \tex_pagestretch:D
\name_primitive:NN \pagefilstretch         \tex_pagefilstretch:D
\name_primitive:NN \pagefillstretch        \tex_pagefillstretch:D
\name_primitive:NN \pagefilllstretch       \tex_pagefilllstretch:D
\name_primitive:NN \pageshrink             \tex_pageshrink:D
\name_primitive:NN \pagegoal               \tex_pagegoal:D
\name_primitive:NN \pagetotal              \tex_pagetotal:D
\name_primitive:NN \outputpenalty          \tex_outputpenalty:D
\name_primitive:NN \hoffset                \tex_hoffset:D
\name_primitive:NN \voffset                \tex_voffset:D
\name_primitive:NN \insert                 \tex_insert:D
\name_primitive:NN \holdinginserts         \tex_holdinginserts:D
\name_primitive:NN \floatingpenalty        \tex_floatingpenalty:D
\name_primitive:NN \insertpenalties        \tex_insertpenalties:D
\name_primitive:NN \lower                  \tex_lower:D
\name_primitive:NN \moveleft               \tex_moveleft:D
\name_primitive:NN \moveright              \tex_moveright:D
\name_primitive:NN \raise                  \tex_raise:D
\name_primitive:NN \copy                   \tex_copy:D
\name_primitive:NN \lastbox                \tex_lastbox:D
\name_primitive:NN \vsplit                 \tex_vsplit:D
\name_primitive:NN \unhbox                 \tex_unhbox:D
\name_primitive:NN \unhcopy                \tex_unhcopy:D
\name_primitive:NN \unvbox                 \tex_unvbox:D
\name_primitive:NN \unvcopy                \tex_unvcopy:D
\name_primitive:NN \setbox                 \tex_setbox:D
\name_primitive:NN \hbox                   \tex_hbox:D
\name_primitive:NN \vbox                   \tex_vbox:D
\name_primitive:NN \vtop                   \tex_vtop:D
\name_primitive:NN \prevdepth              \tex_prevdepth:D
\name_primitive:NN \badness                \tex_badness:D
\name_primitive:NN \hbadness               \tex_hbadness:D
\name_primitive:NN \vbadness               \tex_vbadness:D
\name_primitive:NN \hfuzz                  \tex_hfuzz:D
\name_primitive:NN \vfuzz                  \tex_vfuzz:D
\name_primitive:NN \overfullrule           \tex_overfullrule:D
\name_primitive:NN \boxmaxdepth            \tex_boxmaxdepth:D
\name_primitive:NN \splitmaxdepth          \tex_splitmaxdepth:D
\name_primitive:NN \splittopskip           \tex_splittopskip:D
\name_primitive:NN \everyhbox              \tex_everyhbox:D
\name_primitive:NN \everyvbox              \tex_everyvbox:D
\name_primitive:NN \nullfont               \tex_nullfont:D
\name_primitive:NN \textfont               \tex_textfont:D
\name_primitive:NN \scriptfont             \tex_scriptfont:D
```

```
\name_primitive:NN \scriptscriptfont     \tex_scriptscriptfont:D
\name_primitive:NN \fontdimen            \tex_fontdimen:D
\name_primitive:NN \hyphenchar           \tex_hyphenchar:D
\name_primitive:NN \skewchar             \tex_skewchar:D
\name_primitive:NN \defaulthyphenchar    \tex_defaulthyphenchar:D
\name_primitive:NN \defaultskewchar      \tex_defaultskewchar:D
\name_primitive:NN \number               \tex_number:D
\name_primitive:NN \romannumeral         \tex_romannumeral:D
\name_primitive:NN \string               \tex_string:D
\name_primitive:NN \lowercase            \tex_lowercase:D
\name_primitive:NN \uppercase            \tex_uppercase:D
\name_primitive:NN \meaning              \tex_meaning:D
\name_primitive:NN \penalty              \tex_penalty:D
\name_primitive:NN \unpenalty            \tex_unpenalty:D
\name_primitive:NN \lastpenalty          \tex_lastpenalty:D
\name_primitive:NN \special              \tex_special:D
\name_primitive:NN \dump                 \tex_dump:D
\name_primitive:NN \patterns             \tex_patterns:D
\name_primitive:NN \hyphenation          \tex_hyphenation:D
\name_primitive:NN \time                 \tex_time:D
\name_primitive:NN \day                  \tex_day:D
\name_primitive:NN \month                \tex_month:D
\name_primitive:NN \year                 \tex_year:D
\name_primitive:NN \jobname              \tex_jobname:D
\name_primitive:NN \everyjob             \tex_everyjob:D
\name_primitive:NN \count                \tex_count:D
\name_primitive:NN \dimen                \tex_dimen:D
\name_primitive:NN \skip                 \tex_skip:D
\name_primitive:NN \toks                 \tex_toks:D
\name_primitive:NN \muskip               \tex_muskip:D
\name_primitive:NN \box                  \tex_box:D
\name_primitive:NN \wd                   \tex_wd:D
\name_primitive:NN \ht                   \tex_ht:D
\name_primitive:NN \dp                   \tex_dp:D
\name_primitive:NN \catcode              \tex_catcode:D
\name_primitive:NN \delcode              \tex_delcode:D
\name_primitive:NN \sfcode               \tex_sfcode:D
\name_primitive:NN \lccode               \tex_lccode:D
\name_primitive:NN \uccode               \tex_uccode:D
\name_primitive:NN \mathcode             \tex_mathcode:D
```

Since LaTeX3 requires at least the ε-TeX extensions, we also rename the additional primitives. These are all given the prefix `\etex_`.

```
\name_primitive:NN \ifdefined            \etex_ifdefined:D
\name_primitive:NN \ifcsname             \etex_ifcsname:D
\name_primitive:NN \unless               \etex_unless:D
\name_primitive:NN \eTeXversion          \etex_eTeXversion:D
\name_primitive:NN \eTeXrevision         \etex_eTeXrevision:D
\name_primitive:NN \marks                \etex_marks:D
```

```
\name_primitive:NN \topmarks              \etex_topmarks:D
\name_primitive:NN \firstmarks            \etex_firstmarks:D
\name_primitive:NN \botmarks              \etex_botmarks:D
\name_primitive:NN \splitfirstmarks       \etex_splitfirstmarks:D
\name_primitive:NN \splitbotmarks         \etex_splitbotmarks:D
\name_primitive:NN \unexpanded            \etex_unexpanded:D
\name_primitive:NN \detokenize            \etex_detokenize:D
\name_primitive:NN \scantokens            \etex_scantokens:D
\name_primitive:NN \showtokens            \etex_showtokens:D
\name_primitive:NN \readline              \etex_readline:D
\name_primitive:NN \tracingassigns        \etex_tracingassigns:D
\name_primitive:NN \tracingscantokens     \etex_tracingscantokens:D
\name_primitive:NN \tracingnesting        \etex_tracingnesting:D
\name_primitive:NN \tracingifs            \etex_tracingifs:D
\name_primitive:NN \currentiflevel        \etex_currentiflevel:D
\name_primitive:NN \currentifbranch       \etex_currentifbranch:D
\name_primitive:NN \currentiftype         \etex_currentiftype:D
\name_primitive:NN \tracinggroups         \etex_tracinggroups:D
\name_primitive:NN \currentgrouplevel     \etex_currentgrouplevel:D
\name_primitive:NN \currentgrouptype      \etex_currentgrouptype:D
\name_primitive:NN \showgroups            \etex_showgroups:D
\name_primitive:NN \showifs               \etex_showifs:D
\name_primitive:NN \interactionmode       \etex_interactionmode:D
\name_primitive:NN \lastnodetype          \etex_lastnodetype:D
\name_primitive:NN \iffontchar            \etex_iffontchar:D
\name_primitive:NN \fontcharht            \etex_fontcharht:D
\name_primitive:NN \fontchardp            \etex_fontchardp:D
\name_primitive:NN \fontcharwd            \etex_fontcharwd:D
\name_primitive:NN \fontcharic            \etex_fontcharic:D
\name_primitive:NN \parshapeindent        \etex_parshapeindent:D
\name_primitive:NN \parshapelength        \etex_parshapelength:D
\name_primitive:NN \parshapedimen         \etex_parshapedimen:D
\name_primitive:NN \numexpr               \etex_numexpr:D
\name_primitive:NN \dimexpr               \etex_dimexpr:D
\name_primitive:NN \glueexpr              \etex_glueexpr:D
\name_primitive:NN \muexpr                \etex_muexpr:D
\name_primitive:NN \gluestretch           \etex_gluestretch:D
\name_primitive:NN \glueshrink            \etex_glueshrink:D
\name_primitive:NN \gluestretchorder      \etex_gluestretchorder:D
\name_primitive:NN \glueshrinkorder       \etex_glueshrinkorder:D
\name_primitive:NN \gluetomu              \etex_gluetomu:D
\name_primitive:NN \mutoglue              \etex_mutoglue:D
\name_primitive:NN \lastlinefit           \etex_lastlinefit:D
\name_primitive:NN \interlinepenalties    \etex_interlinepenalties:D
\name_primitive:NN \clubpenalties         \etex_clubpenalties:D
\name_primitive:NN \widowpenalties        \etex_widowpenalties:D
\name_primitive:NN \displaywidowpenalties \etex_displaywidowpenalties:D
\name_primitive:NN \middle                \etex_middle:D
\name_primitive:NN \savinghyphcodes       \etex_savinghyphcodes:D
\name_primitive:NN \savingvdiscards       \etex_savingvdiscards:D
```

```
\name_primitive:NN \pagediscards          \etex_pagediscards:D
\name_primitive:NN \splitdiscards         \etex_splitdiscards:D
\name_primitive:NN \TeXXETstate           \etex_TeXXETstate:D
\name_primitive:NN \beginL                \etex_beginL:D
\name_primitive:NN \endL                  \etex_endL:D
\name_primitive:NN \beginR                \etex_beginR:D
\name_primitive:NN \endR                  \etex_endR:D
\name_primitive:NN \predisplaydirection   \etex_predisplaydirection:D
\name_primitive:NN \everyeof              \etex_everyeof:D
\name_primitive:NN \protected             \etex_protected:D
```

All major distributions use pdfε-TEX as engine so we add these names as well. Since the pdfTEX team has been very good at prefixing most primitives with `pdf` (so far only five do not start with `pdf`) we do not give then a double `pdf` prefix. The list below covers pdfTEXv 1.30.4.

```
%% integer registers:
\name_primitive:NN \pdfoutput             \pdf_output:D
\name_primitive:NN \pdfminorversion       \pdf_minorversion:D
\name_primitive:NN \pdfcompresslevel      \pdf_compresslevel:D
\name_primitive:NN \pdfdecimaldigits      \pdf_decimaldigits:D
\name_primitive:NN \pdfimageresolution    \pdf_imageresolution:D
\name_primitive:NN \pdfpkresolution       \pdf_pkresolution:D
\name_primitive:NN \pdftracingfonts       \pdf_tracingfonts:D
\name_primitive:NN \pdfuniqueresname      \pdf_uniqueresname:D
\name_primitive:NN \pdfadjustspacing      \pdf_adjustspacing:D
\name_primitive:NN \pdfprotrudechars      \pdf_protrudechars:D
\name_primitive:NN \efcode                \pdf_efcode:D
\name_primitive:NN \lpcode                \pdf_lpcode:D
\name_primitive:NN \rpcode                \pdf_rpcode:D
\name_primitive:NN \pdfforcepagebox       \pdf_forcepagebox:D
\name_primitive:NN \pdfoptionalwaysusepdfpagebox \pdf_optionalwaysusepdfpagebox:D
\name_primitive:NN \pdfinclusionerrorlevel\pdf_inclusionerrorlevel:D
\name_primitive:NN \pdfoptionpdfinclusionerrorlevel \pdf_optionpdfinclusionerrorlevel:D
\name_primitive:NN \pdfimagehicolor       \pdf_imagehicolor:D
\name_primitive:NN \pdfimageapplygamma    \pdf_imageapplygamma:D
\name_primitive:NN \pdfgamma              \pdf_gamma:D
\name_primitive:NN \pdfimagegamma         \pdf_imagegamma:D
%% dimen registers:
\name_primitive:NN \pdfhorigin            \pdf_horigin:D
\name_primitive:NN \pdfvorigin            \pdf_vorigin:D
\name_primitive:NN \pdfpagewidth          \pdf_pagewidth:D
\name_primitive:NN \pdfpageheight         \pdf_pageheight:D
\name_primitive:NN \pdflinkmargin         \pdf_linkmargin:D
\name_primitive:NN \pdfdestmargin         \pdf_destmargin:D
\name_primitive:NN \pdfthreadmargin       \pdf_threadmargin:D
%% token registers:
\name_primitive:NN \pdfpagesattr          \pdf_pagesattr:D
\name_primitive:NN \pdfpageattr           \pdf_pageattr:D
\name_primitive:NN \pdfpageresources      \pdf_pageresources:D
```

```
\name_primitive:NN \pdfpkmode            \pdf_pkmode:D
%% expandable commands:
\name_primitive:NN \pdftexrevision       \pdf_texrevision:D
\name_primitive:NN \pdftexbanner         \pdf_texbanner:D
\name_primitive:NN \pdfcreationdate      \pdf_creationdate:D
\name_primitive:NN \pdfpageref           \pdf_pageref:D
\name_primitive:NN \pdfxformname         \pdf_xformname:D
\name_primitive:NN \pdffontname          \pdf_fontname:D
\name_primitive:NN \pdffontobjnum        \pdf_fontobjnum:D
\name_primitive:NN \pdffontsize          \pdf_fontsize:D
\name_primitive:NN \pdfincludechars      \pdf_includechars:D
\name_primitive:NN \leftmarginkern       \pdf_leftmarginkern:D
\name_primitive:NN \rightmarginkern      \pdf_rightmarginkern:D
\name_primitive:NN \pdfescapestring      \pdf_escapestring:D
\name_primitive:NN \pdfescapename        \pdf_escapename:D
\name_primitive:NN \pdfescapehex         \pdf_escapehex:D
\name_primitive:NN \pdfunescapehex       \pdf_unescapehex:D
\name_primitive:NN \pdfstrcmp            \pdf_strcmp:D
\name_primitive:NN \pdfuniformdeviate    \pdf_uniformdeviate:D
\name_primitive:NN \pdfnormaldeviate     \pdf_normaldeviate:D
\name_primitive:NN \pdfmdfivesum         \pdf_mdfivesum:D
\name_primitive:NN \pdffilemoddate       \pdf_filemoddate:D
\name_primitive:NN \pdffilesize          \pdf_filesize:D
\name_primitive:NN \pdffiledump          \pdf_filedump:D
%% read-only integers:
\name_primitive:NN \pdftexversion        \pdf_texversion:D
\name_primitive:NN \pdflastobj           \pdf_lastobj:D
\name_primitive:NN \pdflastxform         \pdf_lastxform:D
\name_primitive:NN \pdflastximage        \pdf_lastximage:D
\name_primitive:NN \pdflastximagepages   \pdf_lastximagepages:D
\name_primitive:NN \pdflastannot         \pdf_lastannot:D
\name_primitive:NN \pdflastxpos          \pdf_lastxpos:D
\name_primitive:NN \pdflastypos          \pdf_lastypos:D
\name_primitive:NN \pdflastdemerits      \pdf_lastdemerits:D
\name_primitive:NN \pdfelapsedtime       \pdf_elapsedtime:D
\name_primitive:NN \pdfrandomseed        \pdf_randomseed:D
\name_primitive:NN \pdfshellescape       \pdf_shellescape:D
%% general commands:
\name_primitive:NN \pdfobj               \pdf_obj:D
\name_primitive:NN \pdfrefobj            \pdf_refobj:D
\name_primitive:NN \pdfxform             \pdf_xform:D
\name_primitive:NN \pdfrefxform          \pdf_refxform:D
\name_primitive:NN \pdfximage            \pdf_ximage:D
\name_primitive:NN \pdfrefximage         \pdf_refximage:D
\name_primitive:NN \pdfannot             \pdf_annot:D
\name_primitive:NN \pdfstartlink         \pdf_startlink:D
\name_primitive:NN \pdfendlink           \pdf_endlink:D
\name_primitive:NN \pdfoutline           \pdf_outline:D
\name_primitive:NN \pdfdest              \pdf_dest:D
\name_primitive:NN \pdfthread            \pdf_thread:D
```

```
\name_primitive:NN \pdfstartthread        \pdf_startthread:D
\name_primitive:NN \pdfendthread          \pdf_endthread:D
\name_primitive:NN \pdfsavepos            \pdf_savepos:D
\name_primitive:NN \pdfinfo               \pdf_info:D
\name_primitive:NN \pdfcatalog            \pdf_catalog:D
\name_primitive:NN \pdfnames              \pdf_names:D
\name_primitive:NN \pdfmapfile            \pdf_mapfile:D
\name_primitive:NN \pdfmapline            \pdf_mapline:D
\name_primitive:NN \pdffontattr           \pdf_fontattr:D
\name_primitive:NN \pdftrailer            \pdf_trailer:D
\name_primitive:NN \pdffontexpand         \pdf_fontexpand:D
%%\name_primitive:NN \vadjust [<pre spec>] <filler> { <vertical mode material> } (h, m)
\name_primitive:NN \pdfliteral            \pdf_literal:D
%%\name_primitive:NN \special <pdfspecial spec>
\name_primitive:NN \pdfresettimer         \pdf_resettimer:D
\name_primitive:NN \pdfsetrandomseed      \pdf_setrandomseed:D
\name_primitive:NN \pdfnoligatures        \pdf_noligatures:D
```

We're ignoring XeTeX and LuaTeX right now except for a check whether they're in use:

```
\name_primitive:NN \XeTeXversion          \xetex_version:D
\name_primitive:NN \directlua             \luatex_directlua:D
```

XeTeX adds `\strcmp` to the set of primitives, with the same implementation as `\pdfstrcmp` but a different name. To avoid having to worry about this later, the same internal name is used.

```
\etex_ifdefined:D \strcmp
  \etex_ifdefined:D \xetex_version:D
    \name_primitive:NN \strcmp \pdf_strcmp:D
  \tex_fi:D
\tex_fi:D
```

97.6 expl3 code switches

`\ExplSyntaxOn` `\ExplSyntaxOff` `\ExplSyntaxStatus`

Here we define functions that are used to turn on and off the special conventions used in the kernel of LaTeX3.

First of all, the space, tab and the return characters will all be ignored inside LaTeX3 code, the latter because endline is set to a space instead. When space characters are needed in LaTeX3 code the ~ character will be used for that purpose.

Specification of the desired behavior:

- ExplSyntax can be either On or Off.
- The On switch is ⟨*null*⟩ if ExplSyntax is on.
- The Off switch is ⟨*null*⟩ if ExplSyntax is off.

- If the On switch is issued and not ⟨*null*⟩, it records the current catcode scheme just prior to it being issued.
- An Off switch restores the catcode scheme to what it was just prior to the previous On switch.

```
\etex_protected:D \tex_def:D \ExplSyntaxOn {
  \tex_ifodd:D \ExplSyntaxStatus \tex_relax:D
  \tex_else:D
    \etex_protected:D \tex_edef:D \ExplSyntaxOff {
      \etex_unexpanded:D{
        \tex_ifodd:D \ExplSyntaxStatus \tex_relax:D
          \tex_def:D \ExplSyntaxStatus{0}
      }
      \tex_catcode:D  126=\tex_the:D \tex_catcode:D 126 \tex_relax:D
      \tex_catcode:D  32=\tex_the:D \tex_catcode:D 32 \tex_relax:D
      \tex_catcode:D  9=\tex_the:D \tex_catcode:D 9 \tex_relax:D
      \tex_endlinechar:D  =\tex_the:D \tex_endlinechar:D \tex_relax:D
      \tex_catcode:D  95=\tex_the:D \tex_catcode:D 95 \tex_relax:D
      \tex_catcode:D  58=\tex_the:D \tex_catcode:D 58 \tex_relax:D
      \tex_noexpand:D \tex_fi:D
    }
    \tex_def:D \ExplSyntaxStatus { 1 }
    \tex_catcode:D  126=10 \tex_relax:D % tilde is a space char.
    \tex_catcode:D   32=9  \tex_relax:D % space is ignored
    \tex_catcode:D    9=9  \tex_relax:D % tab also ignored
    \tex_endlinechar:D =32 \tex_relax:D % endline is space
    \tex_catcode:D   95=11 \tex_relax:D % underscore letter
    \tex_catcode:D   58=11 \tex_relax:D % colon letter
  \tex_fi:D
}
```

At this point we better set the status.

```
\tex_def:D \ExplSyntaxStatus { 1 }
```

`\ExplSyntaxNamesOn`
`\ExplSyntaxNamesOff`

Sometimes we need to be able to use names from the kernel of LaTeX3 without adhering it's conventions according to space characters. These macros provide the necessary settings.

```
\etex_protected:D \tex_def:D \ExplSyntaxNamesOn {
  \tex_catcode:D `\_=11\tex_relax:D
  \tex_catcode:D `\:=11\tex_relax:D
}
\etex_protected:D \tex_def:D \ExplSyntaxNamesOff {
  \tex_catcode:D `\_=8\tex_relax:D
  \tex_catcode:D `\:=12\tex_relax:D
}
```

97.7 Package loading

\GetIdInfo
\filedescription
\filename
\fileversion
\fileauthor
\filedate
\filenameext
\filetimestamp
\GetIdInfoAuxi:w
\GetIdInfoAuxii:w
\GetIdInfoAuxCVS:w
\GetIdInfoAuxSVN:w

Extract all information from a cvs or svn field. The formats are slightly different but at least the information is in the same positions so we check in the date format so see if it contains a / after the four-digit year. If it does it is cvs else svn and we extract information. To be on the safe side we ensure that spaces in the argument are seen.

```
\etex_protected:D \tex_def:D \GetIdInfo {
  \tex_begingroup:D
  \tex_catcode:D   32=10 \tex_relax:D % needed? Probably for now.
  \GetIdInfoMaybeMissing:w
}

\etex_protected:D \tex_def:D\GetIdInfoMaybeMissing:w$#1$#2{
  \tex_def:D \l_kernel_tmpa_tl {#1}
  \tex_def:D \l_kernel_tmpb_tl {Id}
  \tex_ifx:D \l_kernel_tmpa_tl \l_kernel_tmpb_tl
    \tex_def:D \l_kernel_tmpa_tl {
      \tex_endgroup:D
      \tex_def:D\filedescription{#2}
      \tex_def:D\filename      {[unknown~name]}
      \tex_def:D\fileversion   {000}
      \tex_def:D\fileauthor    {[unknown~author]}
      \tex_def:D\filedate      {0000/00/00}
      \tex_def:D\filenameext   {[unknown~ext]}
      \tex_def:D\filetimestamp {[unknown~timestamp]}
    }
  \tex_else:D
    \tex_def:D \l_kernel_tmpa_tl {\GetIdInfoAuxi:w$#1${#2}}
  \tex_fi:D
  \l_kernel_tmpa_tl
}

\etex_protected:D \tex_def:D\GetIdInfoAuxi:w$#1~#2.#3~#4~#5~#6~#7~#8$#9{
  \tex_endgroup:D
  \tex_def:D\filename{#2}
  \tex_def:D\fileversion{#4}
  \tex_def:D\filedescription{#9}
  \tex_def:D\fileauthor{#7}
  \GetIdInfoAuxii:w #5\tex_relax:D
  #3\tex_relax:D#5\tex_relax:D#6\tex_relax:D
}

\etex_protected:D \tex_def:D\GetIdInfoAuxii:w #1#2#3#4#5#6\tex_relax:D{
  \tex_ifx:D#5/
    \tex_expandafter:D\GetIdInfoAuxCVS:w
  \tex_else:D
    \tex_expandafter:D\GetIdInfoAuxSVN:w
  \tex_fi:D
}
```

```
\etex_protected:D \tex_def:D\GetIdInfoAuxCVS:w #1,v\tex_relax:D
                              #2\tex_relax:D#3\tex_relax:D{
  \tex_def:D\filedate{#2}
  \tex_def:D\filenameext{#1}
  \tex_def:D\filetimestamp{#3}
```

When creating the format we want the information in the log straight away.

```
⟨initex⟩\tex_immediate:D\tex_write:D-1
⟨initex⟩  {\filename;~ v\fileversion,~\filedate;~\filedescription}
}
\etex_protected:D \tex_def:D\GetIdInfoAuxSVN:w #1\tex_relax:D#2-#3-#4
                              \tex_relax:D#5Z\tex_relax:D{
  \tex_def:D\filenameext{#1}
  \tex_def:D\filedate{#2/#3/#4}
  \tex_def:D\filetimestamp{#5}
⟨-package⟩\tex_immediate:D\tex_write:D-1
⟨-package⟩  {\filename;~ v\fileversion,~\filedate;~\filedescription}
}
⟨/initex | package⟩
```

Finally some corrections in the case we are running over LaTeX 2ε.

We want to set things up so that experimental packages and regular packages can coexist with the former using the LaTeX3 programming catcode settings. Since it cannot be the task of the end user to know how a package is constructed under the hood we make it so that the experimental packages have to identify themselves. As an example it can be done as

```
\RequirePackage{l3names}
\ProvidesExplPackage{agent}{2007/08/28}{007}{bonding module}
```

or by using the `\file`⟨*field*⟩ informations from `\GetIdInfo` as the packages in this distribution do like this:

```
\RequirePackage{l3names}
\GetIdInfo$Id: l3names.dtx 1853 2010-03-21 09:11:08Z joseph $
         {L3 Experimental Box module}
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
```

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

First up is the identification. Rather trivial as we don't allow for options just yet.

```
⟨*package⟩
\etex_protected:D \tex_def:D \ProvidesExplPackage#1#2#3#4{
  \ProvidesPackage{#1}[#2~v#3~#4]
  \ExplSyntaxOn
```

```
}
\etex_protected:D \tex_def:D \ProvidesExplClass#1#2#3#4{
  \ProvidesClass{#1}[#2~v#3~#4]
  \ExplSyntaxOn
}
\etex_protected:D \tex_def:D \ProvidesExplFile#1#2#3#4{
  \ProvidesFile{#1}[#2~v#3~#4]
  \ExplSyntaxOn
}
```

`\@pushfilename`
`\@popfilename`

The idea behind the code is to record whether or not the LaTeX3 syntax is on or off when about to load a file with class or package extension. This status stored in the parameter `\ExplSyntaxStatus` and set by `\ExplSyntaxOn` and `\ExplSyntaxOff` to `1` and `0` respectively is pushed onto the stack `\ExplSyntaxStack`. Then the catcodes are set back to normal, the file loaded with its options and finally the stack is popped again. The whole thing is a bit problematical. So let's take a look at what the desired behavior is: A package or class which declares itself of Expl type by using `\ProvidesExplClass` or `\ProvidesExplPackage` should automatically ensure the correct catcode scheme as soon as the identification part is over. Similarly, a package or class which uses the traditional `\ProvidesClass` or `\ProvidesPackage` commands should go back to the traditional catcode scheme. An example:

```
\RequirePackage{l3names}
\ProvidesExplPackage{foobar}{2009/05/07}{0.1}{Foobar package}
\cs_new:Npn \foo_bar:nn #1#2 {#1,#2}
...
\RequirePackage{array}
...
\cs_new:Npn \foo_bar:nnn #1#2#3 {#3,#2,#1}
```

Inside the array package, everything should behave as normal under traditional LaTeX but as soon as we are back at the top level, we should use the new catcode regime.

Whenever LaTeX inputs a package file or similar, it calls upon `\@pushfilename` to push the name, the extension and the catcode of @ of the file it was currently processing onto a file name stack. Similarly, after inputting such a file, this file name stack is popped again and the catcode of @ is set to what it was before. If it is a package within package, @ maintains catcode 11 whereas if it is package within document preamble @ is reset to what it was in the preamble (which is usually catcode 12). We wish to adopt a similar technique. Every time an Expl package or class is declared, they will issue an ExplSyntaxOn. Then whenever we are about to load another file, we will first push this status onto a stack and then turn it off again. Then when done loading a file, we pop the stack and if ExplSyntax was On right before, so should it be now. The only problem with this is that we cannot guarantee that we get to the file name stack very early on. Therefore, if the ExplSyntaxStack is empty when trying to pop it, we ensure to turn ExplSyntax off again.

\@pushfilename is prepended with a small function pushing the current ExplSyntaxStatus (true/false) onto a stack. Then the current catcode regime is recorded and ExplSyntax is switched off.

\@popfilename is appended with a function for popping the ExplSyntax stack. However, chances are we didn't get to hook into the file stack early enough so LaTeX might try to pop the file name stack while the ExplSyntaxStack is empty. If the latter is empty, we just switch off ExplSyntax.

```
\tex_edef:D \@pushfilename{
  \etex_unexpanded:D{
    \tex_edef:D \ExplSyntaxStack{ \ExplSyntaxStatus \ExplSyntaxStack }
    \ExplSyntaxOff
  }
  \etex_unexpanded:D\tex_expandafter:D{\@pushfilename }
}
\tex_edef:D \@popfilename{
  \etex_unexpanded:D\tex_expandafter:D{\@popfilename
    \tex_if:D 2\ExplSyntaxStack 2
      \ExplSyntaxOff
    \tex_else:D
      \tex_expandafter:D\ExplSyntaxPopStack\ExplSyntaxStack\q_nil
    \tex_fi:D
  }
}
```

\ExplSyntaxPopStack
\ExplSyntaxStack

Popping the stack is simple: Take the first token which is either 0 (false) or 1 (true) and test if it is odd. Save the rest. The stack is initially empty set to 0 signalling that before l3names was loaded, the ExplSyntax was off.

```
\etex_protected:D\tex_def:D\ExplSyntaxPopStack#1#2\q_nil{
  \tex_def:D\ExplSyntaxStack{#2}
  \tex_ifodd:D#1\tex_relax:D
    \ExplSyntaxOn
  \tex_else:D
    \ExplSyntaxOff
  \tex_fi:D
}
\tex_def:D \ExplSyntaxStack{0}
```

97.8 Finishing up

A few of the 'primitives' assigned above have already been stolen by LaTeX, so assign them by hand to the saved real primitive.

```
\tex_let:D\tex_input:D         \@@input
\tex_let:D\tex_underline:D     \@@underline
\tex_let:D\tex_end:D           \@@end
```

```
\tex_let:D\tex_everymath:D    \frozen@everymath
\tex_let:D\tex_everydisplay:D \frozen@everydisplay
\tex_let:D\tex_italiccor:D    \@@italiccorr
\tex_let:D\tex_hyphen:D       \@@hyph
```

TEX has a nasty habit of inserting a command with the name `\par` so we had better make sure that that command at least has a definition.

```
\tex_let:D\par          \tex_par:D
```

This is the end for `l3names` when used on top of LATEX 2_ε:

```
\tex_ifx:D\name_undefine:N\@gobble
  \tex_def:D\name_pop_stack:w{}
\tex_else:D
```

But if traditional TEX code is disabled, do this…

As mentioned above, The LATEX 2_ε package mechanism will insert some code to handle the filename stack, and reset the package options, this code will die if the TEX primitives have gone, so skip past it and insert some equivalent code that will work.

First a version of `\ProvidesPackage` that can cope.

```
\tex_def:D\ProvidesPackage{
  \tex_begingroup:D
  \ExplSyntaxOff
  \package_provides:w}

\tex_def:D\package_provides:w#1#2[#3]{
  \tex_endgroup:D
  \tex_immediate:D\tex_write:D-1{Package:~#1#2~#3}
  \tex_expandafter:D\tex_xdef:D
    \tex_csname:D ver@#1.sty\tex_endcsname:D{#1}}
```

In this case the catcode preserving stack is not maintained and `\ExplSyntaxOn` conventions stay in force once on. You'll need to turn then off explicitly with `\ExplSyntaxOff` (although as currently built on 2e, nothing except very experimental code will run in this mode!) Also note that `\RequirePackage` is a simple definition, just for one file, with no options.

```
\tex_def:D\name_pop_stack:w#1\relax{%
  \ExplSyntaxOff
  \tex_expandafter:D\@p@pfilename\@currnamestack\@nil
  \tex_let:D\default@ds\@unknownoptionerror
  \tex_global:D\tex_let:D\ds@\@empty
  \tex_global:D\tex_let:D\@declaredoptions\@empty}

\tex_def:D\@p@pfilename#1#2#3#4\@nil{%
  \tex_gdef:D\@currname{#1}%
  \tex_gdef:D\@currext{#2}%
  \tex_catcode:D`\@#3%
  \tex_gdef:D\@currnamestack{#4}}
```

```
  \tex_def:D\NeedsTeXFormat#1{}
  \tex_def:D\RequirePackage#1{
    \tex_expandafter:D\tex_ifx:D
      \tex_csname:D ver@#1.sty\tex_endcsname:D\tex_relax:D
        \ExplSyntaxOn
        \tex_input:D#1.sty\tex_relax:D
    \tex_fi:D}
\tex_fi:D
```

The `\futurelet` just forces the special end of file marker to vanish, so the argument of `\name_pop_stack:w` does not cause an end-of-file error. (Normally I use `\expandafter` for this trick, but here the next token is in fact `\let` and that may be undefined.)

```
\tex_futurelet:D\name_tmp:\name_pop_stack:w
```

expl3 dependency checks We want the expl3 bundle to be loaded 'as one'; this command is used to ensure that one of the `l3` packages isn't loaded on its own.

```
⟨*!initex⟩
\etex_protected:D\tex_def:D \package_check_loaded_expl: {
  \@ifpackageloaded{expl3}{}{
    \PackageError{expl3}{Cannot~load~the~expl3~modules~separately}{
      The~expl3~modules~cannot~be~loaded~separately;\MessageBreak
      please~\protect\usepackage{expl3}~instead.
    }
  }
}
⟨/!initex⟩
```

```
⟨/package⟩
```

97.9 Showing memory usage

This section is from some old code from 1993; it'd be good to work out how it should be used in our code today.

During the development of the LaTeX3 kernel we need to be able to keep track of the memory usage. Therefore we generate empty pages while loading the kernel code, just to be able to check the memory usage.

```
⟨*showmemory⟩
\g_trace_statistics_status=2\scan_stop:
\cs_set_nopar:Npn\showMemUsage{
  \if_horizontal_mode:
     \tex_errmessage:D{Wrong~ mode~ H:~ something~ triggered~
                       hmode~ above}
  \else:
     \tex_message:D{Mode ~ okay}
```

```
  \fi:
  \tex_shipout:D\hbox:w{}
}
\showMemUsage
⟨/showmemory⟩
```

98 l3basics implementation

We need l3names to get things going but we actually need it very early on, so it is loaded at the very top of the file `l3basics.dtx`. Also, most of the code below won't run until l3expan has been loaded.

98.1 Renaming some TeX primitives (again)

`\cs_set_eq:NwN`

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.[7]

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
\tex_let:D \cs_set_eq:NwN            \tex_let:D
```

`\if_true:`
`\if_false:`
`\or:`
`\else:`
`\fi:`
`\reverse_if:N`
`\if:w`
`\if_bool:N`
`\if_predicate:w`
`\if_charcode:w`
`\if_catcode:w`

Then some conditionals.

```
\cs_set_eq:NwN   \if_true:           \tex_iftrue:D
\cs_set_eq:NwN   \if_false:          \tex_iffalse:D
\cs_set_eq:NwN   \or:                \tex_or:D
\cs_set_eq:NwN   \else:              \tex_else:D
\cs_set_eq:NwN   \fi:                \tex_fi:D
\cs_set_eq:NwN   \reverse_if:N       \etex_unless:D
\cs_set_eq:NwN   \if:w               \tex_if:D
\cs_set_eq:NwN   \if_bool:N          \tex_ifodd:D
\cs_set_eq:NwN   \if_predicate:w     \tex_ifodd:D
\cs_set_eq:NwN   \if_charcode:w      \tex_if:D
\cs_set_eq:NwN   \if_catcode:w       \tex_ifcat:D
```

`\if_meaning:w`

```
\cs_set_eq:NwN   \if_meaning:w       \tex_ifx:D
```

[7] This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the "tex. . . D" name in the cases where no good alternative exists.

\if_mode_math:
\if_mode_horizontal:
\if_mode_vertical:
\if_mode_inner:

TeX lets us detect some if its modes.

```
\cs_set_eq:NwN   \if_mode_math:       \tex_ifmmode:D
\cs_set_eq:NwN   \if_mode_horizontal: \tex_ifhmode:D
\cs_set_eq:NwN   \if_mode_vertical:   \tex_ifvmode:D
\cs_set_eq:NwN   \if_mode_inner:      \tex_ifinner:D
```

\if_cs_exist:N
\if_cs_exist:w

```
\cs_set_eq:NwN   \if_cs_exist:N      \etex_ifdefined:D
\cs_set_eq:NwN   \if_cs_exist:w      \etex_ifcsname:D
```

\exp_after:wN
\exp_not:N
\exp_not:n

The three `\exp_` functions are used in the l3expan module where they are described.

```
\cs_set_eq:NwN   \exp_after:wN       \tex_expandafter:D
\cs_set_eq:NwN   \exp_not:N          \tex_noexpand:D
\cs_set_eq:NwN   \exp_not:n          \etex_unexpanded:D
```

\iow_shipout_x:Nn
\token_to_meaning:N
\token_to_str:N
\token_to_str:c
\cs:w
\cs_end:
\cs_meaning:N
\cs_meaning:c
\cs_show:N
\cs_show:c

```
\cs_set_eq:NwN   \iow_shipout_x:Nn    \tex_write:D
\cs_set_eq:NwN   \token_to_meaning:N \tex_meaning:D
\cs_set_eq:NwN   \token_to_str:N     \tex_string:D
\cs_set_eq:NwN   \cs:w               \tex_csname:D
\cs_set_eq:NwN   \cs_end:            \tex_endcsname:D
\cs_set_eq:NwN   \cs_meaning:N       \tex_meaning:D
\tex_def:D \cs_meaning:c {\exp_args:Nc\cs_meaning:N}
\cs_set_eq:NwN   \cs_show:N          \tex_show:D
\tex_def:D \cs_show:c {\exp_args:Nc\cs_show:N}
\tex_def:D \token_to_str:c {\exp_args:Nc\token_to_str:N}
```

\scan_stop:
\group_begin:
\group_end:

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.

```
\cs_set_eq:NwN   \scan_stop:         \tex_relax:D
\cs_set_eq:NwN   \group_begin:       \tex_begingroup:D
\cs_set_eq:NwN   \group_end:         \tex_endgroup:D
```

\group_execute_after:N

```
\cs_set_eq:NwN \group_execute_after:N \tex_aftergroup:D
```

\pref_global:D
\pref_long:D
\pref_protected:D

```
\cs_set_eq:NwN   \pref_global:D      \tex_global:D
\cs_set_eq:NwN   \pref_long:D        \tex_long:D
\cs_set_eq:NwN   \pref_protected:D   \etex_protected:D
```

98.2 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

\cs_set_nopar:Npn
\cs_set_nopar:Npx
\cs_set:Npn
\cs_set:Npx
\cs_set_protected_nopar:Npn
\cs_set_protected_nopar:Npx
\cs_set_protected:Npn
\cs_set_protected:Npx

All assignment functions in LaTeX3 should be naturally robust; after all, the TeX primitives for assignments are and it can be a cause of problems if others aren't.

```
\cs_set_eq:NwN   \cs_set_nopar:Npn           \tex_def:D
\cs_set_eq:NwN   \cs_set_nopar:Npx           \tex_edef:D
\pref_protected:D \cs_set_nopar:Npn \cs_set:Npn {
  \pref_long:D \cs_set_nopar:Npn
}
\pref_protected:D \cs_set_nopar:Npn \cs_set:Npx {
  \pref_long:D \cs_set_nopar:Npx
}
\pref_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npn {
  \pref_protected:D \cs_set_nopar:Npn
}
\pref_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npx {
  \pref_protected:D \cs_set_nopar:Npx
}
\cs_set_protected_nopar:Npn \cs_set_protected:Npn {
  \pref_protected:D \pref_long:D \cs_set_nopar:Npn
}
\cs_set_protected_nopar:Npn \cs_set_protected:Npx {
  \pref_protected:D \pref_long:D \cs_set_nopar:Npx
}
```

\cs_gset_nopar:Npn
\cs_gset_nopar:Npx
\cs_gset:Npn
\cs_gset:Npx
\cs_gset_protected_nopar:Npn
\cs_gset_protected_nopar:Npx
\cs_gset_protected:Npn
\cs_gset_protected:Npx

Global versions of the above functions.

```
\cs_set_eq:NwN   \cs_gset_nopar:Npn          \tex_gdef:D
\cs_set_eq:NwN   \cs_gset_nopar:Npx          \tex_xdef:D
\cs_set_protected_nopar:Npn \cs_gset:Npn {
  \pref_long:D \cs_gset_nopar:Npn
}
\cs_set_protected_nopar:Npn \cs_gset:Npx {
  \pref_long:D \cs_gset_nopar:Npx
}
\cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npn {
  \pref_protected:D \cs_gset_nopar:Npn
}
\cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npx {
  \pref_protected:D \cs_gset_nopar:Npx
}
\cs_set_protected_nopar:Npn \cs_gset_protected:Npn {
  \pref_protected:D \pref_long:D \cs_gset_nopar:Npn
}
\cs_set_protected_nopar:Npn \cs_gset_protected:Npx {
  \pref_protected:D \pref_long:D \cs_gset_nopar:Npx
}
```

98.3 Selecting tokens

\use:c This macro grabs its argument and returns a csname from it.

```
\cs_set:Npn \use:c #1 { \cs:w#1\cs_end: }
```

\use:x Fully expands its argument and passes it to the input stream. Uses \cs_tmp: as a scratch register but does not affect it.

```
\cs_set_protected:Npn \use:x #1 {
  \group_begin:
    \cs_set:Npx \cs_tmp: {#1}
    \exp_args:wN
  \group_end:
  \cs_tmp:
}
```

\use:n \use:nn \use:nnn \use:nnnn These macro grabs its arguments and returns it back to the input (with outer braces removed). \use:n is defined earlier for bootstrapping.

```
\cs_set:Npn \use:n   #1    {#1}
\cs_set:Npn \use:nn   #1#2     {#1#2}
\cs_set:Npn \use:nnn  #1#2#3   {#1#2#3}
\cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}
```

\use_i:nn \use_ii:nn These macros are needed to provide functions with true and false cases, as introduced by Michael some time ago. By using \exp_after:wN \use_i:nn \else: constructions it is possible to write code where the true or false case is able to access the following tokens from the input stream, which is not possible if the \c_true_bool syntax is used.

```
\cs_set:Npn \use_i:nn  #1#2 {#1}
\cs_set:Npn \use_ii:nn #1#2 {#2}
```

\use_i:nnn \use_ii:nnn \use_iii:nnn \use_i:nnnn \use_ii:nnnn \use_iii:nnnn \use_iv:nnnn \use_i_ii:nnn We also need something for picking up arguments from a longer list.

```
\cs_set:Npn \use_i:nnn    #1#2#3{#1}
\cs_set:Npn \use_ii:nnn   #1#2#3{#2}
\cs_set:Npn \use_iii:nnn  #1#2#3{#3}
\cs_set:Npn \use_i:nnnn   #1#2#3#4{#1}
\cs_set:Npn \use_ii:nnnn  #1#2#3#4{#2}
\cs_set:Npn \use_iii:nnnn #1#2#3#4{#3}
\cs_set:Npn \use_iv:nnnn  #1#2#3#4{#4}
\cs_set:Npn \use_i_ii:nnn #1#2#3{#1#2}
```

\use_none_delimit_by_q_nil:w \use_none_delimit_by_q_stop:w e_delimit_by_q_recursion_stop:w Functions that gobble everything until they see either \q_nil or \q_stop resp.

```
\cs_set:Npn \use_none_delimit_by_q_nil:w #1\q_nil{}
\cs_set:Npn \use_none_delimit_by_q_stop:w #1\q_stop{}
\cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop {}
```

`\use_i_delimit_by_q_nil:nw`
`\use_i_delimit_by_q_stop:nw`
`_delimit_by_q_recursion_stop:nw`

Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

```
\cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2\q_nil{#1}
\cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2\q_stop{#1}
\cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw #1#2 \q_recursion_stop {#1}
```

`\use_i_after_fi:nw`
`\use_i_after_else:nw`
`\use_i_after_or:nw`
`\use_i_after_orelse:nw`

Returns the first argument after ending the conditional.

```
\cs_set:Npn \use_i_after_fi:nw #1\fi:{\fi: #1}
\cs_set:Npn \use_i_after_else:nw #1\else:#2\fi:{\fi: #1}
\cs_set:Npn \use_i_after_or:nw #1\or: #2\fi: {\fi:#1}
\cs_set:Npn \use_i_after_orelse:nw #1 #2#3\fi: {\fi:#1}
```

98.4 Gobbling tokens from input

`\use_none:n`
`\use_none:nn`
`\use_none:nnn`
`\use_none:nnnn`
`\use_none:nnnnn`
`\use_none:nnnnnn`
`\use_none:nnnnnnn`
`\use_none:nnnnnnnn`
`\use_none:nnnnnnnnn`

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of `n`'s following the `:` in the name. Although defining `\use_none:nnn` and above as separate calls of `\use_none:n` and `\use_none:nn` is slightly faster, 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.

```
\cs_set:Npn \use_none:n #1{}
\cs_set:Npn \use_none:nn #1#2{}
\cs_set:Npn \use_none:nnn #1#2#3{}
\cs_set:Npn \use_none:nnnn #1#2#3#4{}
\cs_set:Npn \use_none:nnnnn #1#2#3#4#5{}
\cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6{}
\cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7{}
\cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8{}
\cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9{}
```

98.5 Expansion control from l3expan

`\exp_args:Nc`

Moved here for now as it is going to be used right away.

```
\cs_set:Npn \exp_args:Nc #1#2{\exp_after:wN#1\cs:w#2\cs_end:}
```

98.6 Conditional processing and definitions

Underneath any predicate function (`_p`) or other conditional forms (`TF`, etc.) is a built-in logic saying that it after all of the testing and processing must return the ⟨*state*⟩ this leaves TEX in. Therefore, a simple user interface could be something like

```
\if_meaning:w #1#2   \prg_return_true:  \else:
```

```
      \if_meaning:w #1#3  \prg_return_true: \else:
      \prg_return_false:
  \fi: \fi:
```

Usually, a TEX programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the TEX programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

`\prg_return_true:`
`\prg_return_false:`

These break statements put TEX in a ⟨*true*⟩ or ⟨*false*⟩ state. The idea is that the expansion of `\tex_romannumeral:D \c_zero` is ⟨*null*⟩ so we set off a `\tex_romannumeral:D`. It will on its way expand any `\else:` or `\fi:` that are waiting to be discarded anyway before finally arriving at the `\c_zero` we will place right after the conditional. After this expansion has terminated, we issue either `\if_true:` or `\if_false:` to put TEX in the correct state.

```
\cs_set:Npn \prg_return_true: { \exp_after:wN\if_true:\tex_romannumeral:D }
\cs_set:Npn \prg_return_false: {\exp_after:wN\if_false:\tex_romannumeral:D }
```

An extended state space could instead utilize `\tex_ifcase:D`:

```
\cs_set:Npn \prg_return_true: {
  \exp_after:wN\tex_ifcase:D \exp_after:wN \c_zero \tex_romannumeral:D
}
\cs_set:Npn \prg_return_false: {
  \exp_after:wN\tex_ifcase:D \exp_after:wN \c_one \tex_romannumeral:D
}
\cs_set:Npn \prg_return_error: {
  \exp_after:wN\tex_ifcase:D \exp_after:wN \c_two \tex_romannumeral:D
}
```

`\prg_set_conditional:Npnn`
`\prg_new_conditional:Npnn`
`_set_protected_conditional:Npnn`
`_new_protected_conditional:Npnn`

The user functions for the types using parameter text from the programmer. Call aux function to grab parameters, split the base function into name and signature and then use, e.g., `\cs_set:Npn` to define it with.

```
\cs_set_protected:Npn \prg_set_conditional:Npnn #1{
  \prg_get_parm_aux:nw{
    \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
    \cs_set:Npn {parm}
  }
}
\cs_set_protected:Npn \prg_new_conditional:Npnn #1{
  \prg_get_parm_aux:nw{
    \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
    \cs_new:Npn {parm}
  }
```

```
}
\cs_set_protected:Npn \prg_set_protected_conditional:Npnn #1{
  \prg_get_parm_aux:nw{
    \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
    \cs_set_protected:Npn {parm}
  }
}
\cs_set_protected:Npn \prg_new_protected_conditional:Npnn #1{
  \prg_get_parm_aux:nw{
    \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
    \cs_new_protected:Npn {parm}
  }
}
```

\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
g_set_protected_conditional:Nnn
g_new_protected_conditional:Nnn

The user functions for the types automatically inserting the correct parameter text based on the signature. Call aux function after calculating number of arguments, split the base function into name and signature and then use, e.g., `\cs_set:Npn` to define it with.

```
\cs_set_protected:Npn \prg_set_conditional:Nnn #1{
  \exp_args:Nnf \prg_get_count_aux:nn{
    \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
    \cs_set:Npn {count}
  }{\cs_get_arg_count_from_signature:N #1}
}
\cs_set_protected:Npn \prg_new_conditional:Nnn #1{
  \exp_args:Nnf \prg_get_count_aux:nn{
    \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
    \cs_new:Npn {count}
  }{\cs_get_arg_count_from_signature:N #1}
}

\cs_set_protected:Npn \prg_set_protected_conditional:Nnn #1{
  \exp_args:Nnf \prg_get_count_aux:nn{
    \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
    \cs_set_protected:Npn {count}
  }{\cs_get_arg_count_from_signature:N #1}
}

\cs_set_protected:Npn \prg_new_protected_conditional:Nnn #1{
  \exp_args:Nnf \prg_get_count_aux:nn{
    \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
    \cs_new_protected:Npn {count}
  }{\cs_get_arg_count_from_signature:N #1}
}
```

\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn

The obvious setting-equal functions.

```
\cs_set_protected:Npn \prg_set_eq_conditional:NNn #1#2#3 {
  \prg_set_eq_conditional_aux:NNNn \cs_set_eq:cc #1#2 {#3}
```

```
}
\cs_set_protected:Npn \prg_new_eq_conditional:NNn #1#2#3 {
  \prg_set_eq_conditional_aux:NNNn \cs_new_eq:cc #1#2 {#3}
}
```

\prg_get_parm_aux:nw
\prg_get_count_aux:nn

For the Npnn type we must grab the parameter text before continuing. We make this a very generic function that takes one argument before reading everything up to a left brace. Something similar for the Nnn type.

```
\cs_set:Npn \prg_get_count_aux:nn #1#2 {#1{#2}}
\cs_set:Npn \prg_get_parm_aux:nw #1#2#{#1{#2}}
```

e_conditional_parm_aux:nnNNnnnn
enerate_conditional_parm_aux:nw

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. For the time being, we do not use this piece of information but could well throw an error. The fourth argument is how to define this function, the fifth is the text `parm` or `count` for which version to use to define the functions, the sixth is the parameters to use (possibly empty) or number of arguments, the seventh is the list of forms to define, the eight is the replacement text which we will augment when defining the forms.

```
\cs_set:Npn \prg_generate_conditional_aux:nnNNnnnn #1#2#3#4#5#6#7#8{
  \prg_generate_conditional_aux:nnw{#5}{
    #4{#1}{#2}{#6}{#8}
  }#7,?, \q_recursion_stop
}
```

Looping through the list of desired forms. First is the text `parm` or `count`, second is five arguments packed together and third is the form. Use text and form to call the correct type.

```
\cs_set:Npn \prg_generate_conditional_aux:nnw #1#2#3,{
  \if:w ?#3
    \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
  \fi:
  \use:c{prg_generate_#3_form_#1:Nnnnn} #2
  \prg_generate_conditional_aux:nnw{#1}{#2}
}
```

\prg_generate_p_form_parm:Nnnnn
prg_generate_TF_form_parm:Nnnnn
\prg_generate_T_form_parm:Nnnnn
\prg_generate_F_form_parm:Nnnnn

How to generate the various forms. The `parm` types here takes the following arguments: 1: how to define (an N-type), 2: name, 3: signature, 4: parameter text (or empty), 5: replacement.

```
\cs_set:Npn \prg_generate_p_form_parm:Nnnnn #1#2#3#4#5{
  \exp_args:Nc #1 {#2_p:#3}#4{#5 \c_zero
    \exp_after:wN\c_true_bool\else:\exp_after:wN\c_false_bool\fi:
  }
}
```

```
\cs_set:Npn \prg_generate_TF_form_parm:Nnnnn #1#2#3#4#5{
  \exp_args:Nc#1 {#2:#3TF}#4{#5 \c_zero
    \exp_after:wN \use_i:nn \else: \exp_after:wN \use_ii:nn \fi:
  }
}
\cs_set:Npn \prg_generate_T_form_parm:Nnnnn #1#2#3#4#5{
  \exp_args:Nc#1 {#2:#3T}#4{#5 \c_zero
    \else:\exp_after:wN\use_none:nn\fi:\use:n
  }
}
\cs_set:Npn \prg_generate_F_form_parm:Nnnnn #1#2#3#4#5{
  \exp_args:Nc#1 {#2:#3F}#4{#5 \c_zero
    \exp_after:wN\use_none:nn\fi:\use:n
  }
}
```

prg_generate_p_form_count:Nnnnn
rg_generate_TF_form_count:Nnnnn
rg_generate_T_form_count:Nnnnn
prg_generate_F_form_count:Nnnnn

How to generate the various forms. The `count` types here use a number to insert the correct parameter text, otherwise like the `parm` functions above.

```
\cs_set:Npn \prg_generate_p_form_count:Nnnnn #1#2#3#4#5{
  \cs_generate_from_arg_count:cNnn {#2_p:#3} #1 {#4}{#5 \c_zero
    \exp_after:wN\c_true_bool\else:\exp_after:wN\c_false_bool\fi:
  }
}
\cs_set:Npn \prg_generate_TF_form_count:Nnnnn #1#2#3#4#5{
  \cs_generate_from_arg_count:cNnn {#2:#3TF} #1 {#4}{#5 \c_zero
    \exp_after:wN\use_i:nn\else:\exp_after:wN\use_ii:nn\fi:
  }
}
\cs_set:Npn \prg_generate_T_form_count:Nnnnn #1#2#3#4#5{
  \cs_generate_from_arg_count:cNnn {#2:#3T} #1 {#4}{#5 \c_zero
        \else:\exp_after:wN\use_none:nn\fi:\use:n
  }
}
\cs_set:Npn \prg_generate_F_form_count:Nnnnn #1#2#3#4#5{
  \cs_generate_from_arg_count:cNnn {#2:#3F} #1 {#4}{#5 \c_zero
    \exp_after:wN\use_none:nn\fi:\use:n
  }
}
```

prg_set_eq_conditional_aux:NNNn
prg_set_eq_conditional_aux:NNNw

```
\cs_set:Npn \prg_set_eq_conditional_aux:NNNn #1#2#3#4 {
  \prg_set_eq_conditional_aux:NNNw #1#2#3#4,?,\q_recursion_stop
}
```

Manual clist loop over argument `#4`.

```
\cs_set:Npn \prg_set_eq_conditional_aux:NNNw #1#2#3#4, {
  \if:w ? #4 \scan_stop:
```

```
    \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
  \fi:
  #1 {
    \exp_args:NNc \cs_split_function:NN #2 {prg_conditional_form_#4:nnn}
  }{
    \exp_args:NNc \cs_split_function:NN #3 {prg_conditional_form_#4:nnn}
  }
  \prg_set_eq_conditional_aux:NNNw #1{#2}{#3}
}
```

```
\cs_set:Npn \prg_conditional_form_p:nnn  #1#2#3 {#1_p:#2}
\cs_set:Npn \prg_conditional_form_TF:nnn #1#2#3 {#1:#2TF}
\cs_set:Npn \prg_conditional_form_T:nnn  #1#2#3 {#1:#2T}
\cs_set:Npn \prg_conditional_form_F:nnn  #1#2#3 {#1:#2F}
```

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`
`\c_false_bool`

Here are the canonical boolean values.

```
\tex_chardef:D \c_true_bool = 1~
\tex_chardef:D \c_false_bool = 0~
```

98.7 Dissecting a control sequence

`\cs_to_str:N`
`\cs_to_str_aux:w`

This converts a control sequence into the character string of its name, removing the leading escape character. This turns out to be a non-trivial matter as there a different cases:

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of `\tex_escapechar:D` is negative;
- when the escape character is a space.

The route chosen is this: If `\token_to_str:N \a` produces a non-space escape char, then this will produce two tokens. If the escape char is non-printable, only one token is produced. If the escape char is a space, then a space token plus one token character token is produced. If we augment the result of this expansion with the letters `ax` we get the following three scenarios (with $\langle X \rangle$ being a printable non-space escape character):

- $\langle X \rangle$`aax`

- `aax`
- ` aax`

In the second and third case, putting an auxiliary function in front reading undelimited arguments will treat them the same, removing the space token for us automatically. Therefore, if we test the second and third argument of what such a function reads, in case 1 we will get true and in cases 2 and 3 we will get false. If we choose to optimize for the usual case of a printable escape char, we can do it like this (again getting TeX to remove the leading space for us):

```
\cs_set_nopar:Npn \cs_to_str:N {
  \if:w \exp_after:wN \cs_str_aux:w\token_to_str:N \a ax\q_nil
  \else:
    \exp_after:wN \exp_after:wN\exp_after:wN \use_ii:nn
  \fi:
  \exp_after:wN \use_none:n \token_to_str:N
}
\cs_set:Npn \cs_str_aux:w #1#2#3#4\q_nil{#2#3}
```

`\cs_split_function:NN`
`\cs_split_function_aux:w`
`\cs_split_function_auxii:w`

This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean ⟨*true*⟩ or ⟨*false*⟩ is returned with ⟨*true*⟩ for when there is a colon in the function and ⟨*false*⟩ if there is not. Lastly, the second argument of `\cs_split_function:NN` is supposed to be a function taking three variables, one for name, one for signature, and one for the boolean. For example, `\cs_split_function:NN\foo_bar:cnx\use_i:nnn` as input becomes `\use_i:nnn {foo_bar}{cnx}\c_true_bool`.

Can't use a literal `:` because it has the wrong catcode here, so it's transformed from `@` with `\tex_lowercase:D`.

```
\group_begin:
  \tex_lccode:D  `\@ = `\: \scan_stop:
  \tex_catcode:D `\@ = 12~
\tex_lowercase:D {
  \group_end:
```

First ensure that we actually get a properly evaluated str as we don't know how many expansions `\cs_to_str:N` requires. Insert extra colon to catch the error cases.

```
\cs_set:Npn \cs_split_function:NN #1#2{
  \exp_after:wN \cs_split_function_aux:w
    \tex_romannumeral:D -`\q \cs_to_str:N #1 @a \q_nil #2
}
```

If no colon in the name, `#2` is `a` with catcode 11 and `#3` is empty. If colon in the name, then either `#2` is a colon or the first letter of the signature. The letters here have catcode 12. If a colon was given we need to a) split off the colon and quark at the end and b) ensure we return the name, signature and boolean true We can't use \quark_if_no_value:NTF yet but this is very safe anyway as all tokens have catcode 12.

```
\cs_set:Npn \cs_split_function_aux:w #1@#2#3\q_nil#4{
  \if_meaning:w a#2
    \exp_after:wN \use_i:nn
  \else:
    \exp_after:wN\use_ii:nn
  \fi:
  {#4{#1}{}\c_false_bool}
  {\cs_split_function_auxii:w#2#3\q_nil #4{#1}}
}
\cs_set:Npn \cs_split_function_auxii:w #1@a\q_nil#2#3{
  #2{#3}{#1}\c_true_bool
}
```

End of lowercase

```
}
```

\cs_get_function_name:N
\cs_get_function_signature:N

Now returning the name is trivial: just discard the last two arguments. Similar for signature.

```
\cs_set:Npn \cs_get_function_name:N #1 {
  \cs_split_function:NN #1\use_i:nnn
}
\cs_set:Npn \cs_get_function_signature:N #1 {
  \cs_split_function:NN #1\use_ii:nnn
}
```

98.8 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 \tex_relax:D token. A control sequence is said to be *free* (to be defined) if it does not already exist and also meets the requirement that it does not contain a D signature. The reasoning behind this is that most of the time, a check for a free control sequence is when we wish to make a new control sequence and we do not want to let the user define a new "do not use" control sequence.

\cs_if_exist_p:N
\cs_if_exist_p:c
\cs_if_exist:N*TF*
\cs_if_exist:c*TF*

Two versions for checking existence. For the N form we firstly check for \tex_relax:D and then if it is in the hash table. There is no problem when inputting something like \else: or \fi: as TeX will only ever skip input in case the token tested against is \tex_relax:D.

```
\prg_set_conditional:Npnn \cs_if_exist:N #1 {p,TF,T,F}{
  \if_meaning:w #1\tex_relax:D
    \prg_return_false:
  \else:
```

```
    \if_cs_exist:N #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  \fi:
}
```

For the c form we firstly check if it is in the hash table and then for \tex_relax:D 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.

```
\prg_set_conditional:Npnn \cs_if_exist:c #1 {p,TF,T,F}{
  \if_cs_exist:w #1 \cs_end:
    \exp_after:wN \use_i:nn
  \else:
    \exp_after:wN \use_ii:nn
  \fi:
  {
    \exp_after:wN \if_meaning:w \cs:w #1\cs_end: \tex_relax:D
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
  }
  \prg_return_false:
}
```

\cs_if_do_not_use_p:N
\cs_if_do_not_use_aux:nnN

```
\cs_set:Npn \cs_if_do_not_use_p:N #1{
  \cs_split_function:NN #1 \cs_if_do_not_use_aux:nnN
}
\cs_set:Npn \cs_if_do_not_use_aux:nnN #1#2#3{
  \exp_after:wN\str_if_eq_p:nn \token_to_str:N D {#2}
}
```

\cs_if_free_p:N
\cs_if_free_p:c
\cs_if_free:N*TF*
\cs_if_free:c*TF*

The simple implementation is one using the boolean expression parser: If it is exists or is do not use, then return false.

```
\prg_set_conditional:Npnn \cs_if_free:N #1{p,TF,T,F}{
  \bool_if:nTF {\cs_if_exist_p:N #1 || \cs_if_do_not_use_p:N #1}
  {\prg_return_false:}{\prg_return_true:}
}
```

However, this functionality may not be available this early on. We do something similar: The numerical values of true and false is one and zero respectively, which we can use.

The problem again here is that the token we are checking may in fact be something that can disturb the scanner, so we have to be careful. We would like to do minimal evaluation so we ensure this.

```
\prg_set_conditional:Npnn \cs_if_free:N #1{p,TF,T,F}{
  \tex_ifnum:D \cs_if_exist_p:N #1 =\c_zero
    \exp_after:wN \use_i:nn
  \else:
    \exp_after:wN \use_ii:nn
  \fi:
  {
    \tex_ifnum:D \cs_if_do_not_use_p:N #1 =\c_zero
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
  \prg_return_false:
}
\cs_set_nopar:Npn \cs_if_free_p:c{\exp_args:Nc\cs_if_free_p:N}
\cs_set_nopar:Npn \cs_if_free:cTF{\exp_args:Nc\cs_if_free:NTF}
\cs_set_nopar:Npn \cs_if_free:cT{\exp_args:Nc\cs_if_free:NT}
\cs_set_nopar:Npn \cs_if_free:cF{\exp_args:Nc\cs_if_free:NF}
```

98.9 Defining and checking (new) functions

`\c_minus_one`
`\c_zero`
`\c_sixteen`

We need the constants `\c_minus_one` and `\c_sixteen` now for writing information to the log and the terminal and `\c_zero` which is used by some functions in the l3alloc module. The rest are defined in the l3int module – at least for the ones that can be defined with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the l3int module is required but it can't be used until the allocation has been set up properly! The actual allocation mechanism is in l3alloc and as TeX wants to reserve count registers 0–9, the first available one is 10 so we use that for `\c_minus_one`.

```
⟨*!initex⟩
\cs_set_eq:NwN \c_minus_one\m@ne
⟨/!initex⟩
⟨*!package⟩
\tex_countdef:D \c_minus_one = 10 ~
\c_minus_one = -1 ~
⟨/!package⟩
\tex_chardef:D \c_sixteen = 16~
\tex_chardef:D \c_zero = 0~
```

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`
`\iow_term:x`

We define a routine to write only to the log file. And a similar one for writing to both the log file and the terminal.

```
\cs_set_protected_nopar:Npn \iow_log:x {
  \tex_immediate:D \iow_shipout_x:Nn \c_minus_one
}
\cs_set_protected_nopar:Npn \iow_term:x {
  \tex_immediate:D \iow_shipout_x:Nn \c_sixteen
}
```

`\msg_kernel_bug:x`

This will show internal errors.

```
\cs_set_protected_nopar:Npn \msg_kernel_bug:x #1 {
  \iow_term:x { This~is~a~LaTeX~bug:~check~coding! }
  \tex_errmessage:D {#1}
}
```

`\cs_record_meaning:N`

This macro will be used later on for tracing purposes. But we need some more modules to define it, so we just give some dummy definition here.

```
⟨*trace⟩
\cs_set:Npn \cs_record_meaning:N #1{}
⟨/trace⟩
```

`\chk_if_free_cs:N`
`\chk_if_free_cs:c`

This command is called by `\cs_new_nopar:Npn` and `\cs_new_eq:NN` etc. to make sure that the argument sequence is not already in use. If it is, an error is signalled. It checks if ⟨*csname*⟩ is undefined or `\scan_stop:`. Otherwise an error message is issued. We have to make sure we don't put the argument into the conditional processing since it may be an `\if...` type function!

```
\cs_set_protected_nopar:Npn \chk_if_free_cs:N #1{
  \cs_if_free:NF #1
  {
    \msg_kernel_bug:x {Command~name~`\token_to_str:N #1'~
                       already~defined!~
                       Current~meaning:~\token_to_meaning:N #1
                      }
  }
⟨*trace⟩
  \cs_record_meaning:N#1
%       \iow_term:x{Defining~\token_to_str:N #1~on~%}
  \iow_log:x{Defining~\token_to_str:N #1~on~
                   line~\tex_the:D \tex_inputlineno:D}
⟨/trace⟩
}
```

```
\cs_set_protected_nopar:Npn \chk_if_free_cs:c {
  \exp_args:Nc \chk_if_free_cs:N
}
```

\chk_if_exist_cs:N
\chk_if_exist_cs:c

This function issues a warning message when the control sequence in its argument does not exist.

```
\cs_set_protected_nopar:Npn \chk_if_exist_cs:N #1 {
  \cs_if_exist:NF #1
  {
    \msg_kernel_bug:x {Command~ `\token_to_str:N #1'~
                      not~ yet~ defined!}
  }
}
\cs_set_protected_nopar:Npn \chk_if_exist_cs:c {
  \exp_args:Nc \chk_if_exist_cs:N
}
```

\str_if_eq_p:nn
\str_if_eq_p_aux:w

Takes 2 lists of characters as arguments and expands into `\c_true_bool` if they are equal, and `\c_false_bool` otherwise. Note that in the current implementation spaces in these strings are ignored.[8]

```
\prg_set_conditional:Npnn \str_if_eq:nn #1#2{p}{
  \str_if_eq_p_aux:w #1\scan_stop:\\#2\scan_stop:\\
}
\cs_set_nopar:Npn \str_if_eq_p_aux:w #1#2\\#3#4\\{
  \if_meaning:w#1#3
    \if_meaning:w#1\scan_stop:\prg_return_true: \else:
    \if_meaning:w#3\scan_stop:\prg_return_false: \else:
    \str_if_eq_p_aux:w #2\\#4\\\fi:\fi:
  \else:\prg_return_false: \fi:}
```

\cs_if_eq_name_p:NN

An application of the above function, already streamlined for speed, so I put it in here. It takes two control sequences as arguments and expands into true iff they have the same name. We make it long in case one of them is `\par`!

```
\prg_set_conditional:Npnn \cs_if_eq_name:NN #1#2{p}{
  \exp_after:wN\exp_after:wN
  \exp_after:wN\str_if_eq_p_aux:w
  \exp_after:wN\token_to_str:N
  \exp_after:wN#1
  \exp_after:wN\scan_stop:
  \exp_after:wN\\
  \token_to_str:N#2\scan_stop:\\}
```

\str_if_eq_var_p:nf
\str_if_eq_var_start:nnN
\str_if_eq_var_stop:w

A variant of `\str_if_eq_p:nn` which has the advantage of obeying spaces in at least the second argument. See l3quark for an application. From the hand of David Kastrup with slight modifications to make it fit with the remainder of the expl3 language.

[8] This is a function which could use `\tlist_compare:xx`.

The macro builds a string of `\if:w \fi:` pairs from the first argument. The idea is to turn the comparison of `ab` and `cde` into

```
\tex_number:D
  \if:w \scan_stop: \if:w b\if:w a cde\scan_stop: '\fi: \fi: \fi:
 13
```

The `'` is important here. If all tests are true, the `'` is read as part of the number in which case the returned number is `13` in octal notation so `\tex_number:D` returns `11`. If one test returns false the `'` is never seen and then we get just `13`. We wrap the whole process in an external `\if:w` in order to make it return either `\c_true_bool` or `\c_false_bool` since some parts of l3prg expect a predicate to return one of these two tokens.

```
\prg_set_conditional:Npnn \str_if_eq_var:nf #1#2 {p} {
  \if:w \tex_number:D\str_if_eq_var_start:nnN{}{}#1\scan_stop:{#2}
}
\cs_set_nopar:Npn\str_if_eq_var_start:nnN#1#2#3{
  \if:w#3\scan_stop:\exp_after:wN\str_if_eq_var_stop:w\fi:
  \str_if_eq_var_start:nnN{\if:w#3#1}{#2\fi:}
}
\cs_set:Npn\str_if_eq_var_stop:w\str_if_eq_var_start:nnN#1#2#3{
  #1#3\scan_stop:'#213~\prg_return_true:\else:\prg_return_false:\fi:
}
```

98.10 More new definitions

`\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`

These are like `\cs_set_nopar:Npn` and `\cs_set_eq:NN`, but they first check that the argument command is not already in use. You may use `\pref_global:D`, `\pref_long:D`, `\pref_protected:D`, and `\tex_outer:D` as prefixes.

```
\cs_set:Npn \cs_tmp:w #1#2{
  \cs_set_protected_nopar:Npn #1 ##1 {
   \chk_if_free_cs:N ##1
   #2 ##1
 }
}
\cs_tmp:w \cs_new_nopar:Npn \cs_set_nopar:Npn
\cs_tmp:w \cs_new_nopar:Npx\cs_set_nopar:Npx
\cs_tmp:w \cs_new:Npn \cs_set:Npn
\cs_tmp:w \cs_new:Npx\cs_set:Npx
\cs_tmp:w \cs_new_protected_nopar:Npn \cs_set_protected_nopar:Npn
\cs_tmp:w \cs_new_protected_nopar:Npx\cs_set_protected_nopar:Npx
\cs_tmp:w \cs_new_protected:Npn \cs_set_protected:Npn
\cs_tmp:w \cs_new_protected:Npx\cs_set_protected:Npx
```

`\cs_gnew_nopar:Npn`
`\cs_gnew_nopar:Npx`
`\cs_gnew:Npn`
`\cs_gnew:Npx`
`\cs_gnew_protected_nopar:Npn`
`\cs_gnew_protected_nopar:Npx`
`\cs_gnew_protected:Npn`
`\cs_gnew_protected:Npx`

Global versions of the above functions.

```
\cs_tmp:w \cs_gnew_nopar:Npn \cs_gset_nopar:Npn
\cs_tmp:w \cs_gnew_nopar:Npx \cs_gset_nopar:Npx
\cs_tmp:w \cs_gnew:Npn \cs_gset:Npn
\cs_tmp:w \cs_gnew:Npx \cs_gset:Npx
\cs_tmp:w \cs_gnew_protected_nopar:Npn \cs_gset_protected_nopar:Npn
\cs_tmp:w \cs_gnew_protected_nopar:Npx \cs_gset_protected_nopar:Npx
\cs_tmp:w \cs_gnew_protected:Npn \cs_gset_protected:Npn
\cs_tmp:w \cs_gnew_protected:Npx \cs_gset_protected:Npx
```

`\cs_set_nopar:cpn`
`\cs_set_nopar:cpx`
`\cs_gset_nopar:cpn`
`\cs_gset_nopar:cpx`
`\cs_new_nopar:cpn`
`\cs_new_nopar:cpx`
`\cs_gnew_nopar:cpn`
`\cs_gnew_nopar:cpx`

Like `\cs_set_nopar:Npn` and `\cs_new_nopar:Npn`, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the `c` stands for csname argument, see the expansion module). Global versions are also provided.

`\cs_set_nopar:cpn`⟨*string*⟩⟨*rep-text*⟩ will turn ⟨*string*⟩ into a csname and then assign ⟨*rep-text*⟩ to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```
\cs_set:Npn \cs_tmp:w #1#2{
  \cs_new_nopar:Npn #1 { \exp_args:Nc #2 }
}
\cs_tmp:w  \cs_set_nopar:cpn  \cs_set_nopar:Npn
\cs_tmp:w \cs_set_nopar:cpx  \cs_set_nopar:Npx
\cs_tmp:w \cs_gset_nopar:cpn  \cs_gset_nopar:Npn
\cs_tmp:w \cs_gset_nopar:cpx  \cs_gset_nopar:Npx
\cs_tmp:w \cs_new_nopar:cpn  \cs_new_nopar:Npn
\cs_tmp:w \cs_new_nopar:cpx  \cs_new_nopar:Npx
\cs_tmp:w \cs_gnew_nopar:cpn  \cs_gnew_nopar:Npn
\cs_tmp:w \cs_gnew_nopar:cpx  \cs_gnew_nopar:Npx
```

`\cs_set:cpn`
`\cs_set:cpx`
`\cs_gset:cpn`
`\cs_gset:cpx`
`\cs_new:cpn`
`\cs_new:cpx`
`\cs_gnew:cpn`
`\cs_gnew:cpx`

Variants of the `\cs_set:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```
\cs_tmp:w \cs_set:cpn  \cs_set:Npn
\cs_tmp:w \cs_set:cpx  \cs_set:Npx
\cs_tmp:w \cs_gset:cpn  \cs_gset:Npn
\cs_tmp:w \cs_gset:cpx  \cs_gset:Npx
\cs_tmp:w \cs_new:cpn   \cs_new:Npn
\cs_tmp:w \cs_new:cpx  \cs_new:Npx
\cs_tmp:w \cs_gnew:cpn  \cs_gnew:Npn
\cs_tmp:w \cs_gnew:cpx  \cs_gnew:Npx
```

`\cs_set_protected_nopar:cpn`
`\cs_set_protected_nopar:cpx`
`\cs_gset_protected_nopar:cpn`
`\cs_gset_protected_nopar:cpx`
`\cs_new_protected_nopar:cpn`
`\cs_new_protected_nopar:cpx`
`\cs_gnew_protected_nopar:cpn`
`\cs_gnew_protected_nopar:cpx`

Variants of the `\cs_set_protected_nopar:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```
\cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
\cs_tmp:w \cs_set_protected_nopar:cpx  \cs_set_protected_nopar:Npx
\cs_tmp:w \cs_gset_protected_nopar:cpn  \cs_gset_protected_nopar:Npn
\cs_tmp:w \cs_gset_protected_nopar:cpx  \cs_gset_protected_nopar:Npx
```

```
\cs_tmp:w \cs_new_protected_nopar:cpn   \cs_new_protected_nopar:Npn
\cs_tmp:w \cs_new_protected_nopar:cpx  \cs_new_protected_nopar:Npx
\cs_tmp:w \cs_gnew_protected_nopar:cpn  \cs_gnew_protected_nopar:Npn
\cs_tmp:w \cs_gnew_protected_nopar:cpx  \cs_gnew_protected_nopar:Npx
```

\cs_set_protected:cpn
\cs_set_protected:cpx
\cs_gset_protected:cpn
\cs_gset_protected:cpx
\cs_new_protected:cpn
\cs_new_protected:cpx
\cs_gnew_protected:cpn
\cs_gnew_protected:cpx

Variants of the `\cs_set_protected:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```
\cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
\cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
\cs_tmp:w \cs_gset_protected:cpn  \cs_gset_protected:Npn
\cs_tmp:w \cs_gset_protected:cpx  \cs_gset_protected:Npx
\cs_tmp:w \cs_new_protected:cpn  \cs_new_protected:Npn
\cs_tmp:w \cs_new_protected:cpx  \cs_new_protected:Npx
\cs_tmp:w \cs_gnew_protected:cpn \cs_gnew_protected:Npn
\cs_tmp:w \cs_gnew_protected:cpx  \cs_gnew_protected:Npx
```

\use_0_parameter:
\use_1_parameter:
\use_2_parameter:
\use_3_parameter:
\use_4_parameter:
\use_5_parameter:
\use_6_parameter:
\use_7_parameter:
\use_8_parameter:
\use_9_parameter:

For using parameters, i.e., when you need to define a function to process three parameters. See xparse for an application.

```
\cs_set_nopar:cpn{use_0_parameter:}{}
\cs_set_nopar:cpn{use_1_parameter:}{{##1}}
\cs_set_nopar:cpn{use_2_parameter:}{{##1}{##2}}
\cs_set_nopar:cpn{use_3_parameter:}{{##1}{##2}{##3}}
\cs_set_nopar:cpn{use_4_parameter:}{{##1}{##2}{##3}{##4}}
\cs_set_nopar:cpn{use_5_parameter:}{{##1}{##2}{##3}{##4}{##5}}
\cs_set_nopar:cpn{use_6_parameter:}{{##1}{##2}{##3}{##4}{##5}{##6}}
\cs_set_nopar:cpn{use_7_parameter:}{{##1}{##2}{##3}{##4}{##5}{##6}{##7}}
\cs_set_nopar:cpn{use_8_parameter:}{
  {##1}{##2}{##3}{##4}{##5}{##6}{##7}{##8}}
\cs_set_nopar:cpn{use_9_parameter:}{
  {##1}{##2}{##3}{##4}{##5}{##6}{##7}{##8}{##9}}
```

98.11 Copying definitions

\cs_set_eq:NN
\cs_set_eq:cN
\cs_set_eq:Nc
\cs_set_eq:cc

These macros allow us to copy the definition of a control sequence to another control sequence.

The `=` sign allows us to define funny char tokens like `=` itself or `␣` with this function. For the definition of `\c_space_chartok{~}` to work we need the `~` after the `=`.

`\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it long in order to throw an 'already defined' error rather than 'runaway argument'.

The `c` variants are not protected in order for their arguments to be constructed in the correct context.

```
\cs_set_protected:Npn \cs_set_eq:NN #1 { \cs_set_eq:NwN #1=~ }
```

```
\cs_set_protected_nopar:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
\cs_set_protected_nopar:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
\cs_set_protected_nopar:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
```

\cs_new_eq:NN
\cs_new_eq:cN
\cs_new_eq:Nc
\cs_new_eq:cc

```
\cs_new_protected:Npn \cs_new_eq:NN #1 {
  \chk_if_free_cs:N #1
  \cs_set_eq:NN #1
}
\cs_new_protected_nopar:Npn \cs_new_eq:cN { \exp_args:Nc  \cs_new_eq:NN }
\cs_new_protected_nopar:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
\cs_new_protected_nopar:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }
```

\cs_gset_eq:NN
\cs_gset_eq:cN
\cs_gset_eq:Nc
\cs_gset_eq:cc

```
\cs_new_protected:Npn \cs_gset_eq:NN { \pref_global:D  \cs_set_eq:NN }
\cs_new_protected_nopar:Npn \cs_gset_eq:Nc { \exp_args:NNc  \cs_gset_eq:NN }
\cs_new_protected_nopar:Npn \cs_gset_eq:cN { \exp_args:Nc   \cs_gset_eq:NN }
\cs_new_protected_nopar:Npn \cs_gset_eq:cc { \exp_args:Ncc  \cs_gset_eq:NN }
```

\cs_gnew_eq:NN
\cs_gnew_eq:cN
\cs_gnew_eq:Nc
\cs_gnew_eq:cc

```
\cs_new_protected:Npn \cs_gnew_eq:NN #1 {
  \chk_if_free_cs:N #1
  \pref_global:D \cs_set_eq:NN #1
}
\cs_new_protected_nopar:Npn \cs_gnew_eq:cN { \exp_args:Nc  \cs_gnew_eq:NN }
\cs_new_protected_nopar:Npn \cs_gnew_eq:Nc { \exp_args:NNc \cs_gnew_eq:NN }
\cs_new_protected_nopar:Npn \cs_gnew_eq:cc { \exp_args:Ncc \cs_gnew_eq:NN }
```

98.12 Undefining functions

\cs_undefine:N
\cs_undefine:c
\cs_gundefine:N
\cs_gundefine:c

The following function is used to free the main memory from the definition of some function that isn't in use any longer.

```
\cs_new_protected_nopar:Npn \cs_undefine:N  #1 {
  \cs_set_eq:NN #1    \c_undefined:D
}
\cs_new_protected_nopar:Npn \cs_undefine:c  #1 {
  \cs_set_eq:cN {#1}  \c_undefined:D
}
\cs_new_protected_nopar:Npn \cs_gundefine:N #1 {
  \cs_gset_eq:NN #1   \c_undefined:D
}
\cs_new_protected_nopar:Npn \cs_gundefine:c #1 {
  \cs_gset_eq:cN {#1} \c_undefined:D
}
```

98.13 Engine specific definitions

\c_xetex_is_engine_bool
\c_luatex_is_engine_bool
\xetex_if_engine:TF
\luatex_if_engine:TF

In some cases it will be useful to know which engine we're running. Don't provide a `_p` predicate because the `_bool` is used for the same thing.

```
\if_cs_exist:N \xetex_version:D
  \cs_new_eq:NN \c_xetex_is_engine_bool \c_true_bool
\else:
  \cs_new_eq:NN \c_xetex_is_engine_bool \c_false_bool
\fi:
\prg_new_conditional:Npnn \xetex_if_engine: {TF,T,F} {
  \if_bool:N \c_xetex_is_engine_bool
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

```
\if_cs_exist:N \luatex_directlua:D
  \cs_new_eq:NN \c_luatex_is_engine_bool \c_true_bool
\else:
  \cs_new_eq:NN \c_luatex_is_engine_bool \c_false_bool
\fi:
\prg_set_conditional:Npnn \xetex_if_engine: {TF,T,F}{
  \if_bool:N \c_xetex_is_engine_bool \prg_return_true:
  \else: \prg_return_false: \fi:
}
\prg_set_conditional:Npnn \luatex_if_engine: {TF,T,F}{
  \if_bool:N \c_luatex_is_engine_bool \prg_return_true:
  \else: \prg_return_false: \fi:
}
```

98.14 Scratch functions

\prg_do_nothing:

I don't think this function belongs here, but one place is as good as any other. I want to use this function when I want to express 'no operation'. It is for example used in templates where depending on the users settings we have to either select an function that does something, or one that does nothing.

```
\cs_new_nopar:Npn \prg_do_nothing: {}
```

98.15 Defining functions from a given number of arguments

_get_arg_count_from_signature:N
rg_count_from_signature_aux:nnN
rg_count_from_signature_auxii:w

Counting the number of tokens in the signature, i.e., the number of arguments the function should take. If there is no signature, we return that there is −1 arguments to signal an error. Otherwise we insert the string `9876543210` after the signature. If the signature is empty, the number we want is 0 so we remove the first nine tokens and return the tenth. Similarly, if the signature is `nnn` we want to remove the nine tokens `nnn987654` and return 3. Therefore, we simply remove the first nine tokens and then return the tenth.

```
\cs_set:Npn \cs_get_arg_count_from_signature:N #1{
  \cs_split_function:NN #1 \cs_get_arg_count_from_signature_aux:nnN
}
\cs_set:Npn \cs_get_arg_count_from_signature_aux:nnN #1#2#3{
  \if_predicate:w #3 % \bool_if:NTF here
    \exp_after:wN \use_i:nn
  \else:
    \exp_after:wN\use_ii:nn
  \fi:
  {
    \exp_after:wN \cs_get_arg_count_from_signature_auxii:w
      \use_none:nnnnnnnnn  #2 9876543210\q_nil
  }
  {-1}
}
\cs_set:Npn \cs_get_arg_count_from_signature_auxii:w #1#2\q_nil{#1}
```

A variant form we need right away.

```
\cs_set_nopar:Npn \cs_get_arg_count_from_signature:c {
  \exp_args:Nc \cs_get_arg_count_from_signature:N
}
```

cs_generate_from_arg_count:NNnn
ate_from_arg_count_error_msg:Nn

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

```
\cs_set:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4{
  \tex_ifcase:D \etex_numexpr:D #3\tex_relax:D
    \use_i_after_orelse:nw{#2#1}
  \or:
    \use_i_after_orelse:nw{#2#1 ##1}
  \or:
    \use_i_after_orelse:nw{#2#1 ##1##2}
  \or:
    \use_i_after_orelse:nw{#2#1 ##1##2##3}
  \or:
    \use_i_after_orelse:nw{#2#1 ##1##2##3##4}
  \or:
    \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5}
  \or:
    \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6}
  \or:
    \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7}
  \or:
```

```
    \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7##8}
  \or:
    \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7##8##9}
  \else:
    \use_i_after_fi:nw{
      \cs_generate_from_arg_count_error_msg:Nn#1{#3}
      \use_none:n % to remove replacement text
    }
  \fi:
  {#4}
}
```

A variant form we need right away.

```
\cs_set_nopar:Npn \cs_generate_from_arg_count:cNnn {
  \exp_args:Nc \cs_generate_from_arg_count:NNnn
}
```

The error message. Elsewhere we use the value of -1 to signal a missing colon in a function, so provide a hint for help on this.

```
\cs_set:Npn \cs_generate_from_arg_count_error_msg:Nn #1#2 {
  \msg_kernel_bug:x {
    You're~ trying~ to~ define~ the~ command~ `\token_to_str:N #1'~
    with~ \use:n{\tex_the:D\etex_numexpr:D #2\tex_relax:D} ~
    arguments~ but~ I~ only~ allow~ 0-9~arguments.~Perhaps~you~
    forgot~to~use~a~colon~in~the~function~name?~
    I~ can~ probably~ not~ help~ you~ here
  }
}
```

98.16 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.

\cs_set:Nn
\cs_set:Nx
\cs_set_nopar:Nn
\cs_set_nopar:Nx
\cs_set_protected:Nn
\cs_set_protected:Nx
\cs_set_protected_nopar:Nn
\cs_set_protected_nopar:Nx
\cs_gset:Nn
\cs_gset:Nx
\cs_gset_nopar:Nn
\cs_gset_nopar:Nx
\cs_gset_protected:Nn
\cs_gset_protected:Nx
\cs_gset_protected_nopar:Nn
\cs_gset_protected_nopar:Nx

We want to define `\cs_set:Nn` as

```
\cs_set_protected:Npn \cs_set:Nn #1#2{
  \cs_generate_from_arg_count:NNnn #1\cs_set:Npn
    {\cs_get_arg_count_from_signature:N #1}{#2}
}
```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```
\cs_set:Npn \cs_tmp:w #1#2#3{
  \cs_set_protected:cpx {cs_#1:#2}##1##2{
    \exp_not:N \cs_generate_from_arg_count:NNnn ##1
    \exp_after:wN \exp_not:N \cs:w cs_#1:#3 \cs_end:
      {\exp_not:N\cs_get_arg_count_from_signature:N ##1}{##2}
  }
}
```

Then we define the 32 variants beginning with N.

```
\cs_tmp:w {set}{Nn}{Npn}
\cs_tmp:w {set}{Nx}{Npx}
\cs_tmp:w {set_nopar}{Nn}{Npn}
\cs_tmp:w {set_nopar}{Nx}{Npx}
\cs_tmp:w {set_protected}{Nn}{Npn}
\cs_tmp:w {set_protected}{Nx}{Npx}
\cs_tmp:w {set_protected_nopar}{Nn}{Npn}
\cs_tmp:w {set_protected_nopar}{Nx}{Npx}
\cs_tmp:w {gset}{Nn}{Npn}
\cs_tmp:w {gset}{Nx}{Npx}
\cs_tmp:w {gset_nopar}{Nn}{Npn}
\cs_tmp:w {gset_nopar}{Nx}{Npx}
\cs_tmp:w {gset_protected}{Nn}{Npn}
\cs_tmp:w {gset_protected}{Nx}{Npx}
\cs_tmp:w {gset_protected_nopar}{Nn}{Npn}
\cs_tmp:w {gset_protected_nopar}{Nx}{Npx}
```

\cs_new:Nn
\cs_new:Nx
\cs_new_nopar:Nn
\cs_new_nopar:Nx
\cs_new_protected:Nn
\cs_new_protected:Nx
\cs_new_protected_nopar:Nn
\cs_new_protected_nopar:Nx
\cs_gnew:Nn
\cs_gnew:Nx
\cs_gnew_nopar:Nn
\cs_gnew_nopar:Nx
\cs_gnew_protected:Nn
\cs_gnew_protected:Nx
\cs_gnew_protected_nopar:Nn
\cs_gnew_protected_nopar:Nx

```
\cs_tmp:w {new}{Nn}{Npn}
\cs_tmp:w {new}{Nx}{Npx}
\cs_tmp:w {new_nopar}{Nn}{Npn}
\cs_tmp:w {new_nopar}{Nx}{Npx}
\cs_tmp:w {new_protected}{Nn}{Npn}
\cs_tmp:w {new_protected}{Nx}{Npx}
\cs_tmp:w {new_protected_nopar}{Nn}{Npn}
\cs_tmp:w {new_protected_nopar}{Nx}{Npx}
\cs_tmp:w {gnew}{Nn}{Npn}
\cs_tmp:w {gnew}{Nx}{Npx}
\cs_tmp:w {gnew_nopar}{Nn}{Npn}
\cs_tmp:w {gnew_nopar}{Nx}{Npx}
\cs_tmp:w {gnew_protected}{Nn}{Npn}
\cs_tmp:w {gnew_protected}{Nx}{Npx}
\cs_tmp:w {gnew_protected_nopar}{Nn}{Npn}
\cs_tmp:w {gnew_protected_nopar}{Nx}{Npx}
```

Then something similar for the c variants.

```
\cs_set_protected:Npn \cs_set:cn #1#2{
```

```
  \cs_generate_from_arg_count:cNnn {#1}\cs_set:Npn
    {\cs_get_arg_count_from_signature:c {#1}}{#2}
}
```

```
\cs_set:Npn \cs_tmp:w #1#2#3{
  \cs_set_protected:cpx {cs_#1:#2}##1##2{
    \exp_not:N\cs_generate_from_arg_count:cNnn {##1}
    \exp_after:wN \exp_not:N \cs:w cs_#1:#3 \cs_end:
      {\exp_not:N\cs_get_arg_count_from_signature:c {##1}}{##2}
  }
}
```

\cs_set:cn
\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

The 32 c variants.

```
\cs_tmp:w {set}{cn}{Npn}
\cs_tmp:w {set}{cx}{Npx}
\cs_tmp:w {set_nopar}{cn}{Npn}
\cs_tmp:w {set_nopar}{cx}{Npx}
\cs_tmp:w {set_protected}{cn}{Npn}
\cs_tmp:w {set_protected}{cx}{Npx}
\cs_tmp:w {set_protected_nopar}{cn}{Npn}
\cs_tmp:w {set_protected_nopar}{cx}{Npx}
\cs_tmp:w {gset}{cn}{Npn}
\cs_tmp:w {gset}{cx}{Npx}
\cs_tmp:w {gset_nopar}{cn}{Npn}
\cs_tmp:w {gset_nopar}{cx}{Npx}
\cs_tmp:w {gset_protected}{cn}{Npn}
\cs_tmp:w {gset_protected}{cx}{Npx}
\cs_tmp:w {gset_protected_nopar}{cn}{Npn}
\cs_tmp:w {gset_protected_nopar}{cx}{Npx}
```

\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
\cs_gnew:cn
\cs_gnew:cx
\cs_gnew_nopar:cn
\cs_gnew_nopar:cx
\cs_gnew_protected:cn
\cs_gnew_protected:cx
\cs_gnew_protected_nopar:cn
\cs_gnew_protected_nopar:cx

```
\cs_tmp:w {new}{cn}{Npn}
\cs_tmp:w {new}{cx}{Npx}
\cs_tmp:w {new_nopar}{cn}{Npn}
\cs_tmp:w {new_nopar}{cx}{Npx}
\cs_tmp:w {new_protected}{cn}{Npn}
\cs_tmp:w {new_protected}{cx}{Npx}
\cs_tmp:w {new_protected_nopar}{cn}{Npn}
\cs_tmp:w {new_protected_nopar}{cx}{Npx}
\cs_tmp:w {gnew}{cn}{Npn}
\cs_tmp:w {gnew}{cx}{Npx}
\cs_tmp:w {gnew_nopar}{cn}{Npn}
\cs_tmp:w {gnew_nopar}{cx}{Npx}
\cs_tmp:w {gnew_protected}{cn}{Npn}
\cs_tmp:w {gnew_protected}{cx}{Npx}
\cs_tmp:w {gnew_protected_nopar}{cn}{Npn}
\cs_tmp:w {gnew_protected_nopar}{cx}{Npx}
```

\cs_if_eq_p:NN
\cs_if_eq_p:cN
\cs_if_eq_p:Nc
\cs_if_eq_p:cc
\cs_if_eq:NN*TF*
\cs_if_eq:cN*TF*
\cs_if_eq:Nc*TF*
\cs_if_eq:cc*TF*

Check if two control sequences are identical.

```
\prg_set_conditional:Npnn \cs_if_eq:NN #1#2{p,TF,T,F}{
  \if_meaning:w #1#2
  \prg_return_true: \else: \prg_return_false: \fi:
}
\cs_new_nopar:Npn \cs_if_eq_p:cN {\exp_args:Nc  \cs_if_eq_p:NN}
\cs_new_nopar:Npn \cs_if_eq:cNTF {\exp_args:Nc  \cs_if_eq:NNTF}
\cs_new_nopar:Npn \cs_if_eq:cNT  {\exp_args:Nc  \cs_if_eq:NNT}
\cs_new_nopar:Npn \cs_if_eq:cNF  {\exp_args:Nc  \cs_if_eq:NNF}
\cs_new_nopar:Npn \cs_if_eq_p:Nc {\exp_args:NNc \cs_if_eq_p:NN}
\cs_new_nopar:Npn \cs_if_eq:NcTF {\exp_args:NNc \cs_if_eq:NNTF}
\cs_new_nopar:Npn \cs_if_eq:NcT  {\exp_args:NNc \cs_if_eq:NNT}
\cs_new_nopar:Npn \cs_if_eq:NcF  {\exp_args:NNc \cs_if_eq:NNF}
\cs_new_nopar:Npn \cs_if_eq_p:cc {\exp_args:Ncc \cs_if_eq_p:NN}
\cs_new_nopar:Npn \cs_if_eq:ccTF {\exp_args:Ncc \cs_if_eq:NNTF}
\cs_new_nopar:Npn \cs_if_eq:ccT  {\exp_args:Ncc \cs_if_eq:NNT}
\cs_new_nopar:Npn \cs_if_eq:ccF  {\exp_args:Ncc \cs_if_eq:NNF}
```

```
⟨/initex | package⟩
```

```
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

99 l3expan implementation

99.1 Internal functions and variables

\exp_after:wN

\exp_after:wN ⟨*token*$_1$⟩ ⟨*token*$_2$⟩

This will expand ⟨*token*$_2$⟩ once before processing ⟨*token*$_1$⟩. This is similar to \exp_args:No except that no braces are put around the result of expanding ⟨*token*$_2$⟩.

TEXhackers note: This is the primitive \expandafter which was renamed to fit into the naming conventions of LATEX3.

\l_exp_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.

\exp_eval_register:N ⋆
\exp_eval_register:c ⋆

\exp_eval_register:N ⟨*register*⟩

These functions evaluates a register as part of a `V` or `v` expansion (respectively). A register might exist as one of two things: A parameter-less non-long, non-protected macro or a built-in TEX register such as `\count`.

`\exp_eval_error_msg:w` `\exp_eval_error_msg:w` ⟨*register*⟩

Used to generate an error message if a variable called as part of a `v` or `V` expansion is defined as `\scan_stop:`. This typically indicates that an incorrect cs name has been used.

`\n::`
`\N::`
`\c::`
`\o::`
`\f::`
`\x::`
`\v::`
`\V::`
`\:::` `\cs_set_nopar:Npn \exp_args:Ncof {\::c\::o\::f\:::}`

Internal forms for the base expansion types.

99.2 Module code

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

`\exp_after:wN`
`\exp_not:N`
`\exp_not:n`

These are defined in l3basics.

```
⟨*bootstrap⟩
\cs_set_eq:NwN   \exp_after:wN       \tex_expandafter:D
\cs_set_eq:NwN   \exp_not:N          \tex_noexpand:D
\cs_set_eq:NwN   \exp_not:n          \etex_unexpanded:D
⟨/bootstrap⟩
```

99.3 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 LaTeX3 names for `\cs_set_nopar:Npx` at some point, and so is never going to be expandable.[9])

The definition of expansion functions with this technique happens in section 99.5. In section 99.4 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

`\l_exp_tl` We need a scratch token list variable. We don't use `tl` methods so that l3expan can be loaded earlier.

```
\cs_new_nopar:Npn \l_exp_tl {}
```

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 `\::`$\langle Z\rangle$ 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.

`\exp_arg_next:nnn`
`\exp_arg_next_nobrace: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 auxilliary 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.

```
\cs_new:Npn\exp_arg_next:nnn#1#2#3{
  #2\:::{#3{#1}}
}
\cs_new:Npn\exp_arg_next_nobrace:nnn#1#2#3{
  #2\:::{#3#1}
}
```

`\:::` The end marker is just another name for the identity function.

```
\cs_new:Npn\:::#1{#1}
```

`\::n` This function is used to skip an argument that doesn't need to be expanded.

```
\cs_new:Npn\::n#1\:::#2#3{
  #1\:::{#2{#3}}
}
```

[9]However, some primitives have certain characteristics that means that their arguments undergo an `x` type expansion but the primitive is in fact still expandable. We shall make it very clear when such a function is expandable.

`\::N` This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
\cs_new:Npn\::N#1\:::#2#3{
  #1\:::{#2#3}
}
```

`\::c` This function is used to skip an argument that is turned into as control sequence without expansion.

```
\cs_new:Npn\::c#1\:::#2#3{
  \exp_after:wN\exp_arg_next_nobrace:nnn\cs:w #3\cs_end:{#1}{#2}
}
```

`\::o` This function is used to expand an argument once.

```
\cs_new:Npn\::o#1\:::#2#3{
  \exp_after:wN\exp_arg_next:nnn\exp_after:wN{#3}{#1}{#2}
}
```

`\::f`
`\exp_stop_f:` This function is used to expand a token list until the first unexpandable token is found. The underlying `\tex_romannumeral:D -'0` expands everything in its way to find something terminating the number and thereby expands the function in front of it. This scanning procedure is terminated once the expansion hits something non-expandable or a space. We introduce `\exp_stop_f:` to mark such an end of expansion marker; in case the scanner hits a number, this number also terminates the scanning and is left untouched. In the example shown earlier the scanning was stopped once TEX had fully expanded `\cs_set_eq:Nc \aaa {b \l_tmpa_tl b}` into `\cs_set_eq:NwN \aaa = \blurb` which then turned out to contain the non-expandable token `\cs_set_eq:NwN`. Since the expansion of `\tex_romannumeral:D -'0` is ⟨*null*⟩, we wind up with a fully expanded list, only TEX has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```
\cs_new:Npn\::f#1\:::#2#3{
  \exp_after:wN\exp_arg_next:nnn
  \exp_after:wN{\tex_romannumeral:D -'0 #3}
  {#1}{#2}
}
\cs_new_nopar:Npn \exp_stop_f: {~}
```

`\::x` This function is used to expand an argument fully. We could use the new expandable primitive `\expanded` here, but we don't want to create incompatibilities between engines.

```
  \cs_new_protected:Npn \::x #1 \::: #2#3 {
    \cs_set_nopar:Npx \l_exp_tl {{#3}}
    \exp_after:wN \exp_arg_next:nnn \l_exp_tl {#1}{#2}
  }
```

\::v
\::V

These functions return the value of a register, i.e., one of `tl`, `num`, `int`, `skip`, `dim` and `muskip`. The `V` version expects a single token whereas `v` like `c` creates a csname from its argument given in braces and then evaluates it as if it was a `V`. The sequence `\tex_romannumeral:D -‘0` sets off an `f` type expansion. The argument is returned in braces.

```
\cs_new:Npn \::V#1\:::#2#3{
  \exp_after:wN\exp_arg_next:nnn
  \exp_after:wN{
    \tex_romannumeral:D -‘0
    \exp_eval_register:N #3
  }
  {#1}{#2}
}
\cs_new:Npn \::v#1\:::#2#3{
  \exp_after:wN\exp_arg_next:nnn
  \exp_after:wN{
    \tex_romannumeral:D -‘0
    \exp_eval_register:c {#3}
  }
  {#1}{#2}
}
```

\exp_eval_register:N
\exp_eval_register:c
\exp_eval_error_msg:w

This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in TEX register such as `\count`. For the TEX registers we have to utilize a `\tex_the:D` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\tex_the:D` 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 `\tex_relax:D`.

```
\cs_set_nopar:Npn \exp_eval_register:N #1{
  \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 `\tex_relax:D`. 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.

```
    \if_meaning:w \tex_relax:D #1
      \exp_eval_error_msg:w
    \fi:
```

The next bit requires some explanation. The function must be initiated by the sequence `\tex_romannumeral:D -‘0` and we want to terminate this expansion chain by inserting an `\exp_stop_f:` token. However, we have to expand the register `#1` before we do that. If it is a TEX register, we need to execute the sequence `\exp_after:wN\exp_stop_f:\tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN\exp_stop_f: #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```
  \else:
    \exp_after:wN \use_i_ii:nnn
  \fi:
  \exp_after:wN \exp_stop_f: \tex_the:D #1
}
\cs_set_nopar:Npn \exp_eval_register:c #1{
  \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.
\exp_eval_error_msg:w ...erroneous variable used!

l.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}
```

```
\group_begin:%
\tex_catcode:D‘\!=11\tex_relax:D%
\tex_catcode:D‘\ =11\tex_relax:D%
\cs_gset:Npn\exp_eval_error_msg:w#1\tex_the:D#2{%
\fi:\fi:\erroneous variable used!}%
\group_end:%
```

99.4 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 `\pref_global:D` for example. This together with the fact that the ‘general’ concept above is slower means that we should convert whenever possible and perhaps remove all remaining occurences by hand-encoding in the end.

`\exp_args:No`
`\exp_args:NNo`
`\exp_args:NNNo`

```
\cs_new:Npn \exp_args:No #1#2{\exp_after:wN#1\exp_after:wN{#2}}
\cs_new:Npn \exp_args:NNo #1#2#3{\exp_after:wN#1\exp_after:wN#2
  \exp_after:wN{#3}}
\cs_new:Npn \exp_args:NNNo #1#2#3#4{\exp_after:wN#1\exp_after:wN#2
  \exp_after:wN#3\exp_after:wN{#4}}
```

\exp_args:Nc
\exp_args:cc
\exp_args:NNc
\exp_args:Ncc
\exp_args:Nccc

Here are the functions that turn their argument into csnames but are expandable.

```
\cs_set:Npn \exp_args:Nc #1#2{\exp_after:wN#1\cs:w#2\cs_end:}
\cs_new:Npn \exp_args:cc #1#2{\cs:w #1\exp_after:wN\cs_end:\cs:w #2\cs_end:}
\cs_new:Npn \exp_args:NNc #1#2#3{\exp_after:wN#1\exp_after:wN#2
    \cs:w#3\cs_end:}
\cs_new:Npn \exp_args:Ncc #1#2#3{\exp_after:wN#1
    \cs:w#2\exp_after:wN\cs_end:\cs:w#3\cs_end:}
\cs_new:Npn \exp_args:Nccc #1#2#3#4{\exp_after:wN#1
    \cs:w#2\exp_after:wN\cs_end:\cs:w#3\exp_after:wN
      \cs_end:\cs:w #4\cs_end:}
```

\exp_args:Nco

If we force that the third argument always has braces, we could implement this function with less tokens and only two arguments.

```
\cs_new:Npn \exp_args:Nco #1#2#3{\exp_after:wN#1\cs:w#2\exp_after:wN
     \cs_end:\exp_after:wN{#3}}
```

99.5 Definitions with the 'general' technique

\exp_args:Nf
\exp_args:NV
\exp_args:Nv
\exp_args:Nx

```
\cs_set_nopar:Npn \exp_args:Nf {\::f\:::}
\cs_set_nopar:Npn \exp_args:Nv {\::v\:::}
\cs_set_nopar:Npn \exp_args:NV {\::V\:::}
\cs_set_protected_nopar:Npn \exp_args:Nx {\::x\:::}
```

\exp_args:NNV
\exp_args:NNv
\exp_args:NNf
\exp_args:NNx
\exp_args:NVV
\exp_args:Ncx
\exp_args:Nfo
\exp_args:Nff
\exp_args:Ncf
\exp_args:Nco
\exp_args:Nnf
\exp_args:Nno
\exp_args:NnV
\exp_args:Nnx
\exp_args:Noo
\exp_args:Noc
\exp_args:Nox
\exp_args:Nxo
\exp_args:Nxx

Here are the actual function definitions, using the helper functions above.

```
\cs_set_nopar:Npn \exp_args:NNf {\::N\::f\:::}
\cs_set_nopar:Npn \exp_args:NNv {\::N\::v\:::}
\cs_set_nopar:Npn \exp_args:NNV {\::N\::V\:::}
\cs_set_protected_nopar:Npn \exp_args:NNx {\::N\::x\:::}

\cs_set_protected_nopar:Npn \exp_args:Ncx {\::c\::x\:::}
\cs_set_nopar:Npn \exp_args:Nfo {\::f\::o\:::}
\cs_set_nopar:Npn \exp_args:Nff {\::f\::f\:::}
\cs_set_nopar:Npn \exp_args:Ncf {\::c\::f\:::}
\cs_set_nopar:Npn \exp_args:Nnf {\::n\::f\:::}
\cs_set_nopar:Npn \exp_args:Nno {\::n\::o\:::}
\cs_set_nopar:Npn \exp_args:NnV {\::n\::V\:::}
\cs_set_protected_nopar:Npn \exp_args:Nnx {\::n\::x\:::}

\cs_set_nopar:Npn \exp_args:Noc {\::o\::c\:::}
\cs_set_nopar:Npn \exp_args:Noo {\::o\::o\:::}
\cs_set_protected_nopar:Npn \exp_args:Nox {\::o\::x\:::}

\cs_set_nopar:Npn \exp_args:NVV {\::V\::V\:::}

```

```
\cs_set_protected_nopar:Npn \exp_args:Nxo {\::x\::o\:::}
\cs_set_protected_nopar:Npn \exp_args:Nxx {\::x\::x\:::}
```

\exp_args:Ncco
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:NcNc
\exp_args:NcNo
\exp_args:NNno
\exp_args:NNNV
\exp_args:Nnno
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Nooo
\exp_args:Noox
\exp_args:Nnnc
\exp_args:NNnx
\exp_args:NNoo
\exp_args:NNox

```
\cs_set_nopar:Npn \exp_args:NNNV {\::N\::N\::V\:::}

\cs_set_nopar:Npn \exp_args:NNno {\::N\::n\::o\:::}
\cs_set_protected_nopar:Npn \exp_args:NNnx {\::N\::n\::x\:::}
\cs_set_nopar:Npn \exp_args:NNoo {\::N\::o\::o\:::}
\cs_set_protected_nopar:Npn \exp_args:NNox {\::N\::o\::x\:::}

\cs_set_nopar:Npn \exp_args:Nnnc {\::n\::n\::c\:::}
\cs_set_nopar:Npn \exp_args:Nnno {\::n\::n\::o\:::}
\cs_set_protected_nopar:Npn \exp_args:Nnnx {\::n\::n\::x\:::}
\cs_set_protected_nopar:Npn \exp_args:Nnox {\::n\::o\::x\:::}

\cs_set_nopar:Npn \exp_args:NcNc {\::c\::N\::c\:::}
\cs_set_nopar:Npn \exp_args:NcNo {\::c\::N\::o\:::}
\cs_set_nopar:Npn \exp_args:Ncco {\::c\::c\::o\:::}
\cs_set_nopar:Npn \exp_args:Ncco {\::c\::c\::o\:::}
\cs_set_protected_nopar:Npn \exp_args:Nccx {\::c\::c\::x\:::}
\cs_set_protected_nopar:Npn \exp_args:Ncnx {\::c\::n\::x\:::}

\cs_set_protected_nopar:Npn \exp_args:Noox {\::o\::o\::x\:::}
\cs_set_nopar:Npn \exp_args:Nooo {\::o\::o\::o\:::}
```

99.6 Preventing expansion

\exp_not:o
\exp_not:f
\exp_not:v
\exp_not:V

```
\cs_new:Npn\exp_not:o#1{\exp_not:n\exp_after:wN{#1}}
\cs_new:Npn\exp_not:f#1{
  \exp_not:n\exp_after:wN{\tex_romannumeral:D -`0 #1}
}
\cs_new:Npn\exp_not:v#1{
  \exp_not:n\exp_after:wN{\tex_romannumeral:D -`0 \exp_eval_register:c {#1}}
}
\cs_new:Npn\exp_not:V#1{
  \exp_not:n\exp_after:wN{\tex_romannumeral:D -`0 \exp_eval_register:N #1}
}
```

\exp_not:c A helper function.

```
\cs_new:Npn\exp_not:c#1{\exp_after:wN\exp_not:N\cs:w#1\cs_end:}
```

99.7 Defining function variants

\cs_generate_variant:Nn
\cs_generate_variant_aux:nnNn
\cs_generate_variant_aux:nnw
\cs_generate_variant_aux:N

#1 : Base form of a function; e.g., \tl_set:Nn

#2 : One or more variant argument specifiers; e.g., {Nx,c,cx}

Split up the original base function to grab its name and signature consisting of k letters. Then we wish to iterate through the list of variant argument specifiers, and for each one 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. For example, for a base function \tl_set:Nn which needs a c variant form, we want the new signature to be cn.

```
\cs_new_protected:Npn \cs_generate_variant:Nn #1 {
  \chk_if_exist_cs:N #1
  \cs_split_function:NN #1 \cs_generate_variant_aux:nnNn
}
```

We discard the boolean and then set off a loop through the desired variant forms.

```
\cs_set:Npn \cs_generate_variant_aux:nnNn #1#2#3#4{
  \cs_generate_variant_aux:nnw {#1}{#2} #4,?,\q_recursion_stop
}
```

Next is the real work to be done. We now have 1: base name, 2: base signature, 3: beginning of variant signature. To construct the new csname and the \exp_args:Ncc form, we need the variant signature. In our example, we wanted to discard the first two letters of the base signature because the variant form started with cc. This is the same as putting first cc in the signature and then \use_none:nn followed by the base signature NNn. We therefore call a small loop that outputs an n for each letter in the variant signature and use this to call the correct \use_none: variant. Firstly though, we check whether to terminate the loop.

```
\cs_set:Npn \cs_generate_variant_aux:nnw #1 #2 #3, {
  \if:w ? #3
    \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
  \fi:
```

Then check if the variant form has already been defined.

```
  \cs_if_free:cTF {
    #1:#3\use:c {use_none:\cs_generate_variant_aux:N #3 ?}#2
  }
  {
```

If not, then define it and then additionally check if the \exp_args:N form needed is defined.

```
    \_cs_generate_variant_aux:ccpx { #1 : #2 }
    {
      #1:#3 \use:c{use_none:\cs_generate_variant_aux:N #3 ?}#2
```

```
    }
    {
      \exp_not:c { exp_args:N #3} \exp_not:c {#1:#2}
    }
    \cs_generate_internal_variant:n {#3}
  }
```

Otherwise tell that it was already defined.

```
  {
    \iow_log:x{
      Variant~\token_to_str:c {
        #1:#3\use:c {use_none:\cs_generate_variant_aux:N #3 ?}#2
      }~already~defined;~ not~ changing~ it~on~line~
      \tex_the:D \tex_inputlineno:D
    }
  }
```

Recurse.

```
  \cs_generate_variant_aux:nnw{#1}{#2}
}
```

The small loop for defining the required number of ns. Break when seeing a ?.

```
\cs_set:Npn \cs_generate_variant_aux:N #1{
  \if:w ?#1 \exp_after:wN\use_none:nn \fi: n \cs_generate_variant_aux:N
}
```

_cs_generate_variant_aux:Ncpx
_cs_generate_variant_aux:ccpx
_cs_generate_variant_aux:w

The idea here is to pick up protected parent functions, using the nature of the meaning string that they generate. The test here is almost the same as `\tl_if_empty:nTF`, but has to be hard-coded as that function is not yet available and because it has to match both long and short macros.

```
\group_begin:
  \tex_lccode:D `\Z = `\d \scan_stop:
  \tex_lccode:D `\? =`\\ \scan_stop:
  \tex_catcode:D `\P = 12 \scan_stop:
  \tex_catcode:D `\R = 12 \scan_stop:
  \tex_catcode:D `\O = 12 \scan_stop:
  \tex_catcode:D `\T = 12 \scan_stop:
  \tex_catcode:D `\E = 12 \scan_stop:
  \tex_catcode:D `\C = 12 \scan_stop:
  \tex_catcode:D `\Z = 12 \scan_stop:
\tex_lowercase:D {
  \group_end:
  \cs_new_nopar:Npn \_cs_generate_variant_aux:Ncpx #1
    {
      \exp_after:wN \_cs_generate_variant_aux:w
        \tex_meaning:D #1 ? PROTECTEZ \q_nil
```

```
    }
  \cs_new_nopar:Npn \_cs_generate_variant_aux:ccpx
    { \exp_args:Nc \_cs_generate_variant_aux:Ncpx}
  \cs_new:Npn \_cs_generate_variant_aux:w
    #1 ? PROTECTEZ #2 \q_nil
    {
      \exp_after:wN \tex_ifx:D \exp_after:wN
        \q_nil \etex_detokenize:D {#1} \q_nil
        \exp_after:wN \cs_new_protected_nopar:cpx
      \tex_else:D
        \exp_after:wN \cs_new_nopar:cpx
      \tex_fi:D
    }
}
```

\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`

```
\cs_new_protected:Npn \cs_generate_internal_variant:n #1 {
  \cs_if_free:cT { exp_args:N #1 }{
```

We use `new` to log the definition if we have to make one.

```
    \cs_new:cpx { exp_args:N #1 }
                { \cs_generate_internal_variant_aux:n #1 : }
  }
}
```

generate_internal_variant_aux:n

This command grabs char by char outputting `\::#1` (not expanded further) until we see a `:`. That colon is in fact also turned into `\:::` so that the required structure for `\exp_args...` commands is correctly terminated.

```
\cs_new:Npn \cs_generate_internal_variant_aux:n #1 {
  \exp_not:c{::#1}
  \if_meaning:w #1 :
    \exp_after:wN \use_none:n
  \fi:
  \cs_generate_internal_variant_aux:n
}
```

99.8 Last-unbraced versions

\exp_arg_last_unbraced:nn
\::f_unbraced
\::o_unbraced
\::V_unbraced
\::v_unbraced

There are a few places where the last argument needs to be available unbraced. First some helper macros.

```
\cs_new:Npn \exp_arg_last_unbraced:nn #1#2 { #2#1 }
\cs_new:Npn \::f_unbraced \:::#1#2 {
  \exp_after:wN \exp_arg_last_unbraced:nn
```

```
  \exp_after:wN { \tex_romannumeral:D -`0 #2 } {#1}
}
\cs_new:Npn \::o_unbraced \:::#1#2 {
  \exp_after:wN \exp_arg_last_unbraced:nn \exp_after:wN {#2 }{#1}
}
\cs_new:Npn \::V_unbraced \:::#1#2 {
  \exp_after:wN \exp_arg_last_unbraced:nn
  \exp_after:wN { \tex_romannumeral:D -`0 \exp_eval_register:N #2 } {#1}
}
\cs_new:Npn \::v_unbraced \:::#1#2 {
  \exp_after:wN \exp_arg_last_unbraced:nn
  \exp_after:wN {
    \tex_romannumeral:D -`0 \exp_eval_register:c {#2}
  } {#1}
}
```

\exp_last_unbraced:NV
\exp_last_unbraced:Nv
\exp_last_unbraced:Nf
\exp_last_unbraced:NcV
\exp_last_unbraced:NNo
\exp_last_unbraced:NNV
\exp_last_unbraced:NNNo

Now the business end.

```
\cs_new_nopar:Npn \exp_last_unbraced:Nf { \::f_unbraced \::: }
\cs_new_nopar:Npn \exp_last_unbraced:NV { \::V_unbraced \::: }
\cs_new_nopar:Npn \exp_last_unbraced:Nv { \::v_unbraced \::: }
\cs_new_nopar:Npn \exp_last_unbraced:NcV {
  \::c \::V_unbraced \:::
}
\cs_new:Npn \exp_last_unbraced:NNo #1#2#3 {
  \exp_after:wN #1 \exp_after:wN #2 #3
}
\cs_new_nopar:Npn \exp_last_unbraced:NNV {
  \::N \::V_unbraced \:::
}
\cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4 {
  \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4
}
```

```
⟨/initex | package⟩
```

Show token usage:

```
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

100 l3prg implementation

100.1 Variables

\l_tmpa_bool \g_tmpa_bool

Reserved booleans.

`\g_prg_inline_level_int` Global variable to track the nesting of the stepwise inline loop.

100.2 Module code

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

`\prg_return_true:`
`\prg_return_false:`
`\prg_set_conditional:Npnn`
`\prg_new_conditional:Npnn`
`_set_protected_conditional:Npnn`
`_new_protected_conditional:Npnn`
`\prg_set_conditional:Nnn`
`\prg_new_conditional:Nnn`
`g_set_protected_conditional:Nnn`
`g_new_protected_conditional:Nnn`
`\prg_set_eq_conditional:NNn`
`\prg_new_eq_conditional:NNn`

These are all defined in l3basics, as they are needed "early". This is just a reminder that that is the case!

100.3 Choosing modes

`\mode_if_vertical_p:`
`\mode_if_vertical:TF`

For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the `\c_zero` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```
\prg_set_conditional:Npnn \mode_if_vertical: {p,TF,T,F}{
  \if_mode_vertical:
  \prg_return_true: \else: \prg_return_false: \fi:
}
```

`\mode_if_horizontal_p:`
`\mode_if_horizontal:TF`

For testing horizontal mode.

```
\prg_set_conditional:Npnn \mode_if_horizontal: {p,TF,T,F}{
  \if_mode_horizontal:
  \prg_return_true: \else: \prg_return_false: \fi:
}
```

`\mode_if_inner_p:`
`\mode_if_inner:TF`

For testing inner mode.

```
\prg_set_conditional:Npnn \mode_if_inner: {p,TF,T,F}{
  \if_mode_inner:
  \prg_return_true: \else: \prg_return_false: \fi:
}
```

`\mode_if_math_p:`
`\mode_if_math:TF`

For testing math mode. Uses the kern-save `\scan_align_safe_stop:`.

```
\prg_set_conditional:Npnn \mode_if_math: {p,TF,T,F}{
  \scan_align_safe_stop: \if_mode_math:
  \prg_return_true: \else: \prg_return_false: \fi:
}
```

Alignment safe grouping and scanning

`\group_align_safe_begin:`
`\group_align_safe_end:`

TeX's alignment structures present many problems. As Knuth says himself in *TeX: The Program*: "It's sort of a miracle whenever `\halign` or `\valign` work, [...]" One problem relates to commands that internally issues a `\cr` but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a `&` with category code 4 we will get some sort of weird error message because the underlying `\tex_futurelet:D` will store the token at the end of the alignment template. This could be a $\&_4$ giving a message like `! Misplaced \cr.` or even worse: it could be the `\endtemplate` token causing even more trouble! To solve this we have to open a special group so that TeX still thinks it's on safe ground but at the same time we don't want to introduce any brace group that may find its way to the output. The following functions help with this by using code documented only in Appendix D of *The TeXbook*...

```
\cs_new_nopar:Npn \group_align_safe_begin: {
  \if_false:{\fi:\if_num:w`}=\c_zero\fi:}
\cs_new_nopar:Npn \group_align_safe_end:   {\if_num:w`{=\c_zero}\fi:}
```

`\scan_align_safe_stop:`

When TeX is in the beginning of an align cell (right after the `\cr`) it is in a somewhat strange mode as it is looking ahead to find an `\tex_omit:D` or `\tex_noalign:D` and hasn't looked at the preamble yet. Thus an `\tex_ifmmode:D` test will always fail unless we insert `\scan_stop:` to stop TeX's scanning ahead. On the other hand we don't want to insert a `\scan_stop:` every time as that will destroy kerning between letters[10] Unfortunately there is no way to detect if we're in the beginning of an alignment cell as they have different characteristics depending 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 iff 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.

```
\cs_new_nopar:Npn \scan_align_safe_stop: {
  \intexpr_compare:nNnT \etex_currentgrouptype:D = \c_six
  {
    \intexpr_compare:nNnF \etex_lastnodetype:D = \c_zero
    {
      \intexpr_compare:nNnF \etex_lastnodetype:D = \c_seven
        \scan_stop:
    }
  }
}
```

[10] Unless we enforce an extra pass with an appropriate value of `\pretolerance`.

100.4 Producing n copies

\prg_replicate:nn
\prg_replicate_aux:N
\prg_replicate_first_aux:N

This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input `25` result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with `25` then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. Finally we must ensure that the cascade comes to a peaceful end so we make it so that the original csname TeX is creating is simply `\prg_do_nothing:` expanding to nothing.

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. 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.

```
\cs_new_nopar:Npn \prg_replicate:nn #1{
  \cs:w prg_do_nothing:
  \exp_after:wN\prg_replicate_first_aux:N
  \tex_romannumeral:D -`\q \intexpr_eval:n{#1} \cs_end:
  \cs_end:
}
\cs_new_nopar:Npn \prg_replicate_aux:N#1{
  \cs:w prg_replicate_#1:n\prg_replicate_aux:N
}
\cs_new_nopar:Npn \prg_replicate_first_aux:N#1{
  \cs:w prg_replicate_first_#1:n\prg_replicate_aux:N
}
```

Then comes all the functions that do the hard work of inserting all the copies.

```
\cs_new_nopar:Npn      \prg_replicate_ :n #1{}% no, this is not a typo!
\cs_new:cpn {prg_replicate_0:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1#1}}
\cs_new:cpn {prg_replicate_1:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1#1}#1}
\cs_new:cpn {prg_replicate_2:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1#1}#1#1}
\cs_new:cpn {prg_replicate_3:n}#1{
  \cs_end:{#1#1#1#1#1#1#1#1#1#1}#1#1#1}
\cs_new:cpn {prg_replicate_4:n}#1{
  \cs_end:{#1#1#1#1#1#1#1#1#1#1}#1#1#1#1}
\cs_new:cpn {prg_replicate_5:n}#1{
```

```
  \cs_end:{#1#1#1#1#1#1#1#1#1#1}#1#1#1#1#1}
\cs_new:cpn {prg_replicate_6:n}#1{
  \cs_end:{#1#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1}
\cs_new:cpn {prg_replicate_7:n}#1{
  \cs_end:{#1#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1#1}
\cs_new:cpn {prg_replicate_8:n}#1{
  \cs_end:{#1#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1#1#1}
\cs_new:cpn {prg_replicate_9:n}#1{
  \cs_end:{#1#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1#1#1#1}
```

Users shouldn't ask for something to be replicated once or even not at all but...

```
\cs_new:cpn {prg_replicate_first_-:n}#1{\cs_end: \ERROR }
\cs_new:cpn {prg_replicate_first_0:n}#1{\cs_end: }
\cs_new:cpn {prg_replicate_first_1:n}#1{\cs_end: #1}
\cs_new:cpn {prg_replicate_first_2:n}#1{\cs_end: #1#1}
\cs_new:cpn {prg_replicate_first_3:n}#1{\cs_end: #1#1#1}
\cs_new:cpn {prg_replicate_first_4:n}#1{\cs_end: #1#1#1#1}
\cs_new:cpn {prg_replicate_first_5:n}#1{\cs_end: #1#1#1#1#1}
\cs_new:cpn {prg_replicate_first_6:n}#1{\cs_end: #1#1#1#1#1#1}
\cs_new:cpn {prg_replicate_first_7:n}#1{\cs_end: #1#1#1#1#1#1#1}
\cs_new:cpn {prg_replicate_first_8:n}#1{\cs_end: #1#1#1#1#1#1#1#1}
\cs_new:cpn {prg_replicate_first_9:n}#1{\cs_end: #1#1#1#1#1#1#1#1#1}
```

\prg_stepwise_function:nnnN
\prg_stepwise_function_incr:nnnN
\prg_stepwise_function_decr:nnnN

A stepwise function. Firstly we check the direction of the steps #2 since that will depend on which test we should use. If the step is positive we use a greater than test, otherwise a less than test. If the test comes out true exit, otherwise perform #4, add the step to #1 and try again with this new value of #1.

```
\cs_new:Npn \prg_stepwise_function:nnnN #1#2{
  \intexpr_compare:nNnTF{#2}<\c_zero
  {\exp_args:Nf\prg_stepwise_function_decr:nnnN }
  {\exp_args:Nf\prg_stepwise_function_incr:nnnN }
  {\intexpr_eval:n{#1}}{#2}
}
\cs_new:Npn \prg_stepwise_function_incr:nnnN #1#2#3#4{
  \intexpr_compare:nNnF {#1}>{#3}
  {
    #4{#1}
    \exp_args:Nf \prg_stepwise_function_incr:nnnN
    {\intexpr_eval:n{#1 + #2}}
    {#2}{#3}{#4}
  }
}
\cs_new:Npn \prg_stepwise_function_decr:nnnN #1#2#3#4{
  \intexpr_compare:nNnF {#1}<{#3}
  {
    #4{#1}
    \exp_args:Nf \prg_stepwise_function_decr:nnnN
    {\intexpr_eval:n{#1 + #2}}
```

```
    {#2}{#3}{#4}
  }
}
```

\g_prg_inline_level_int
\prg_stepwise_inline:nnnn
\prg_stepwise_inline_decr:nnnn
\prg_stepwise_inline_incr:nnnn

This function uses the same approach as for instance \clist_map_inline:Nn to allow arbitrary nesting. First construct the special function and then call an auxiliary one which just carries the newly constructed csname. Must make assignments global when we maintain our own stack.

```
\int_new:N\g_prg_inline_level_int
\cs_new_protected:Npn\prg_stepwise_inline:nnnn #1#2#3#4{
  \int_gincr:N \g_prg_inline_level_int
  \cs_gset_nopar:cpn{prg_stepwise_inline_\int_use:N\g_prg_inline_level_int :n}##1{#4}
  \intexpr_compare:nNnTF {#2}<\c_zero
  {\exp_args:Ncf \prg_stepwise_inline_decr:Nnnn }
  {\exp_args:Ncf \prg_stepwise_inline_incr:Nnnn }
  {prg_stepwise_inline_\int_use:N\g_prg_inline_level_int :n}
  {\intexpr_eval:n{#1}} {#2} {#3}
  \int_gdecr:N \g_prg_inline_level_int
}
\cs_new:Npn \prg_stepwise_inline_incr:Nnnn #1#2#3#4{
  \intexpr_compare:nNnF {#2}>{#4}
  {
    #1{#2}
    \exp_args:NNf \prg_stepwise_inline_incr:Nnnn #1
    {\intexpr_eval:n{#2 + #3}} {#3}{#4}
  }
}
\cs_new:Npn \prg_stepwise_inline_decr:Nnnn #1#2#3#4{
  \intexpr_compare:nNnF {#2}<{#4}
  {
    #1{#2}
    \exp_args:NNf \prg_stepwise_inline_decr:Nnnn #1
    {\intexpr_eval:n{#2 + #3}} {#3}{#4}
  }
}
```

\prg_stepwise_variable:nnnNn
rg_stepwise_variable_decr:nnnNn
rg_stepwise_variable_incr:nnnNn

Almost the same as above. Just store the value in #4 and execute #5.

```
\cs_new_protected:Npn \prg_stepwise_variable:nnnNn #1#2 {
  \intexpr_compare:nNnTF {#2}<\c_zero
  {\exp_args:Nf\prg_stepwise_variable_decr:nnnNn}
  {\exp_args:Nf\prg_stepwise_variable_incr:nnnNn}
  {\intexpr_eval:n{#1}}{#2}
}
\cs_new_protected:Npn \prg_stepwise_variable_incr:nnnNn #1#2#3#4#5 {
  \intexpr_compare:nNnF {#1}>{#3}
  {
    \cs_set_nopar:Npn #4{#1} #5
```

```
    \exp_args:Nf \prg_stepwise_variable_incr:nnnNn
    {\intexpr_eval:n{#1 + #2}}{#2}{#3}#4{#5}
  }
}
\cs_new_protected:Npn \prg_stepwise_variable_decr:nnnNn #1#2#3#4#5 {
  \intexpr_compare:nNnF {#1}<{#3}
  {
    \cs_set_nopar:Npn #4{#1} #5
    \exp_args:Nf \prg_stepwise_variable_decr:nnnNn
    {\intexpr_eval:n{#1 + #2}}{#2}{#3}#4{#5}
  }
}
```

100.5 Booleans

For normal booleans we set them to either `\c_true_bool` or `\c_false_bool` and then use `\if_bool:N` to choose the right branch. The functions return either the TF, T, or F case *after* ending the `\if_bool:N`. We only define the `N` versions here as the `c` versions can easily be constructed with the expansion module.

`\bool_new:N`
`\bool_new:c`
`\bool_set_true:N`
`\bool_set_true:c`
`\bool_set_false:N`
`\bool_set_false:c`
`\bool_gset_true:N`
`\bool_gset_true:c`
`\bool_gset_false:N`
`\bool_gset_false:c`

Defining and setting a boolean is easy.

```
\cs_new_protected_nopar:Npn \bool_new:N #1   { \cs_new_eq:NN #1 \c_false_bool }
\cs_new_protected_nopar:Npn \bool_new:c #1   { \cs_new_eq:cN {#1} \c_false_bool }
\cs_new_protected_nopar:Npn \bool_set_true:N   #1 { \cs_set_eq:NN  #1 \c_true_bool }
\cs_new_protected_nopar:Npn \bool_set_true:c   #1 { \cs_set_eq:cN  {#1} \c_true_bool }
\cs_new_protected_nopar:Npn \bool_set_false:N  #1 { \cs_set_eq:NN  #1 \c_false_bool }
\cs_new_protected_nopar:Npn \bool_set_false:c  #1 { \cs_set_eq:cN  {#1} \c_false_bool }
\cs_new_protected_nopar:Npn \bool_gset_true:N   #1 { \cs_gset_eq:NN  #1 \c_true_bool }
\cs_new_protected_nopar:Npn \bool_gset_true:c   #1 { \cs_gset_eq:cN  {#1} \c_true_bool }
\cs_new_protected_nopar:Npn \bool_gset_false:N  #1 { \cs_gset_eq:NN  #1 \c_false_bool }
\cs_new_protected_nopar:Npn \bool_gset_false:c  #1 { \cs_gset_eq:cN  {#1} \c_false_bool }
```

`\bool_set_eq:NN`
`\bool_set_eq:Nc`
`\bool_set_eq:cN`
`\bool_set_eq:cc`
`\bool_gset_eq:NN`
`\bool_gset_eq:Nc`
`\bool_gset_eq:cN`
`\bool_gset_eq:cc`

Setting a boolean to another is also pretty easy.

```
\cs_new_eq:NN \bool_set_eq:NN \cs_set_eq:NN
\cs_new_eq:NN \bool_set_eq:Nc \cs_set_eq:Nc
\cs_new_eq:NN \bool_set_eq:cN \cs_set_eq:cN
\cs_new_eq:NN \bool_set_eq:cc \cs_set_eq:cc
\cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
\cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
\cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
\cs_new_eq:NN \bool_gset_eq:cc \cs_gset_eq:cc
```

`\l_tmpa_bool`
`\g_tmpa_bool`

A few booleans just if you need them.

```
\bool_new:N \l_tmpa_bool
\bool_new:N \g_tmpa_bool
```

\bool_if_p:N
\bool_if_p:c
\bool_if:NTF
\bool_if:cTF

Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

```
\prg_set_conditional:Npnn \bool_if:N #1 {p,TF,T,F}{
  \if_bool:N #1 \prg_return_true: \else: \prg_return_false: \fi:
}
\cs_generate_variant:Nn \bool_if_p:N {c}
\cs_generate_variant:Nn \bool_if:NTF {c}
\cs_generate_variant:Nn \bool_if:NT {c}
\cs_generate_variant:Nn \bool_if:NF {c}
```

\bool_while_do:Nn
\bool_while_do:cn
\bool_until_do:Nn
\bool_until_do:cn

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.

```
\cs_new:Npn \bool_while_do:Nn #1 #2 {
  \bool_if:NT #1 {#2 \bool_while_do:Nn #1 {#2}}
}
\cs_generate_variant:Nn \bool_while_do:Nn {c}

\cs_new:Npn \bool_until_do:Nn #1 #2 {
  \bool_if:NF #1 {#2 \bool_until_do:Nn #1 {#2}}
}
\cs_generate_variant:Nn \bool_until_do:Nn {c}
```

\bool_do_while:Nn
\bool_do_while:cn
\bool_do_until:Nn
\bool_do_until:cn

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.

```
\cs_new:Npn \bool_do_while:Nn #1 #2 {
  #2 \bool_if:NT #1 {\bool_do_while:Nn #1 {#2}}
}
\cs_generate_variant:Nn \bool_do_while:Nn {c}

\cs_new:Npn \bool_do_until:Nn #1 #2 {
  #2 \bool_if:NF #1 {\bool_do_until:Nn #1 {#2}}
}
\cs_generate_variant:Nn \bool_do_until:Nn {c}
```

100.6 Parsing boolean expressions

\bool_if_p:n
\bool_if:nTF
\bool_get_next:N
\bool_cleanup:N
\bool_choose:NN
\bool_!:w
\bool_Not:w
\bool_Not:w
\bool_(:w
\bool_p:w
\bool_8_1:w
\bool_I_1:w
\bool_8_0:w
\bool_I_0:w
\bool_)_0:w
\bool_)_1:w
\bool_S_0:w

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, insert a negating function (if-even in this case) and call GetNext.
- If none of the above, start evaluating a new expression by reinserting the token found (this is supposed to be a predicate function) in front of Eval.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

⟨*true*⟩And Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

⟨*false*⟩And Current truth value is false, logical And seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return ⟨*false*⟩.

⟨*true*⟩Or Current truth value is true, logical Or seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return ⟨*true*⟩.

⟨*false*⟩Or Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

⟨*true*⟩Close Current truth value is true, Close seen, return ⟨*true*⟩.

⟨*false*⟩Close Current truth value is false, Close seen, return ⟨*false*⟩.

We introduce an additional Stop operation with the following semantics:

⟨*true*⟩Stop Current truth value is true, return ⟨*true*⟩.

⟨*false*⟩Stop Current truth value is false, return ⟨*false*⟩.

The reasons for this follow below.

Now for how these works in practice. The canonical true and false values have numerical values 1 and 0 respectively. We evaluate this using the primitive `\tex_number:D` operation. First we issue a `\group_align_safe_begin:` as we are using `&&` as syntax shorthand for the And operation and we need to hide it for TEX. We also need to finish this special 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.

```
\cs_set:Npn \bool_if_p:n #1{
  \group_align_safe_begin:
  \bool_get_next:N ( #1 )S
}
```

The GetNext operation. We make it a switch: If not a `!` or `(`, we assume it is a predicate.

```
\cs_set:Npn \bool_get_next:N #1{
  \use:c {
    bool_
    \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
    :w
  } #1
}
```

The Not operation. Discard the token read and reverse the truth value of the next expression using `\intexpr_if_even_p:n` if there are brackets, otherwise reverse the logic here and now.

```
\cs_set:cpn { bool_!:w } #1#2 {
  \if_meaning:w ( #2
    \exp_after:wN \bool_Not:w
  \else:
    \if_meaning:w ! #2
      \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
    \else:
      \exp_after:wN \exp_after:wN \exp_after:wN \bool_Not:N
    \fi:
  \fi:
  #2
}
\cs_new:Npn \bool_Not:w {
  \exp_after:wN \intexpr_if_even_p:n \tex_number:D \bool_get_next:N
}
\cs_new:Npn \bool_Not:N #1 {
  \exp_after:wN \bool_p:w
  \if_meaning:w #1 \c_true_bool
    \c_false_bool
  \else:
    \c_true_bool
  \fi:
}
```

The Open operation. Discard the token read and start a sub-expression.

```
\cs_set:cpn {bool_(:w}#1{
  \exp_after:wN  \bool_cleanup:N \tex_number:D \bool_get_next:N
}
```

Otherwise just evaluate the predicate and look for And, Or or Close afterward.

```
\cs_set:cpn {bool_p:w}{\exp_after:wN  \bool_cleanup:N \tex_number:D  }
```

This cleanup function can be omitted once predicates return their true/false booleans outside the conditionals.

```
\cs_new_nopar:Npn \bool_cleanup:N #1{
  \exp_after:wN \bool_choose:NN \exp_after:wN #1
  \int_to_roman:w-`\q
}
```

Branching the six way switch.

```
\cs_new_nopar:Npn \bool_choose:NN #1#2{  \use:c{bool_#2_#1:w} }
```

Continues scanning. Must remove the second `&` or `|`.

```
\cs_new_nopar:cpn{bool_&_1:w}&{\bool_get_next:N}
\cs_new_nopar:cpn{bool_|_0:w}|{\bool_get_next:N}
```

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.

```
\cs_new_nopar:cpn{bool_)_0:w}{ \c_false_bool }
\cs_new_nopar:cpn{bool_)_1:w}{ \c_true_bool }
\cs_new_nopar:cpn{bool_S_0:w}{\group_align_safe_end: \c_false_bool }
\cs_new_nopar:cpn{bool_S_1:w}{\group_align_safe_end: \c_true_bool }
```

When the truth value has already been decided, we have to throw away the remainder of the current group as we are doing minimal evaluation. This is slightly tricky as there are no braces so we have to play match the `()` manually.

```
\cs_set:cpn{bool_&_0:w}&{\bool_eval_skip_to_end:Nw \c_false_bool}
\cs_set:cpn{bool_|_1:w}|{\bool_eval_skip_to_end:Nw \c_true_bool}
```

\bool_eval_skip_to_end:Nw
\bool_eval_skip_to_end_aux:Nw
\bool_eval_skip_to_end_auxii: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.

This whole operation could be made a lot simpler if we were allowed to do simple pattern matching. With a new enough pdfTEX one can do that sort of thing to test for existence of particular tokens.

```
\cs_set:Npn \bool_eval_skip_to_end:Nw #1#2){
  \bool_eval_skip_to_end_aux:Nw #1 #2(\q_no_value\q_nil{#2}
}
```

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

```
\cs_set:Npn \bool_eval_skip_to_end_aux:Nw #1#2(#3#4\q_nil#5{
  \quark_if_no_value:NTF #3
  { #1 }
  { \bool_eval_skip_to_end_auxii:Nw #1 #5 }
}
```

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!

```
\cs_set:Npn \bool_eval_skip_to_end_auxii:Nw #1#2(#3){
  \bool_eval_skip_to_end:Nw #1#3 )
}
```

\bool_set:Nn
\bool_set:cn
\bool_gset:Nn
\bool_gset:cn

This function evaluates a boolean expression and assigns the first argument the meaning \c_true_bool or \c_false_bool.

```
\cs_new:Npn \bool_set:Nn #1#2 {\tex_chardef:D #1 = \bool_if_p:n {#2}}
\cs_new:Npn \bool_gset:Nn #1#2 {
  \tex_global:D \tex_chardef:D #1 =  \bool_if_p:n {#2}
}
\cs_generate_variant:Nn \bool_set:Nn {c}
\cs_generate_variant:Nn \bool_gset:Nn {c}
```

\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.

```
\cs_new:Npn \bool_not_p:n #1{ \bool_if_p:n{!(#1)} }
```

\bool_xor_p:nn

Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```
\cs_new:Npn \bool_xor_p:nn #1#2 {
  \intexpr_compare:nNnTF {\bool_if_p:n { #1 }} = {\bool_if_p:n { #2 }}
  {\c_false_bool}{\c_true_bool}
}
```

```
\prg_set_conditional:Npnn \bool_if:n  #1 {TF,T,F}{
  \if_predicate:w \bool_if_p:n{#1}
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\bool_while_do:nn
\bool_until_do:nn
\bool_do_while:nn
\bool_do_until:nn

#1 : Predicate test
#2 : Code to execute

```
\cs_new:Npn \bool_while_do:nn #1#2 {
  \bool_if:nT {#1} { #2 \bool_while_do:nn {#1}{#2} }
}
\cs_new:Npn \bool_until_do:nn #1#2 {
  \bool_if:nF {#1} { #2 \bool_until_do:nn {#1}{#2} }
}
\cs_new:Npn \bool_do_while:nn #1#2 {
  #2 \bool_if:nT {#1} { \bool_do_while:nn {#1}{#2} }
}
\cs_new:Npn \bool_do_until:nn #1#2 {
  #2 \bool_if:nF {#1} { \bool_do_until:nn {#1}{#2} }
}
```

100.7 Case switch

\prg_case_int:nnn
\prg_case_int_aux:nnn

This case switch is in reality quite simple. It takes three arguments:

1. An integer expression you wish to find.
2. A list of pairs of {⟨*integer expr*⟩} {⟨*code*⟩}. The list can be as long as is desired and ⟨*integer expr*⟩ can be negative.
3. The code to be executed if the value wasn't found.

We don't need the else case here yet, so leave it dangling in the input stream.

```
\cs_new:Npn \prg_case_int:nnn #1 #2 {
```

We will be parsing on #1 for each step so we might as well evaluate it first in case it is complicated.

```
  \exp_args:Nf \prg_case_int_aux:nnn { \intexpr_eval:n{#1}} #2
```

The ? below is just so there are enough arguments when we reach the end. And it made you look. ;-)

```
  \q_recursion_tail ? \q_recursion_stop
}
\cs_new:Npn \prg_case_int_aux:nnn #1#2#3{
```

If we reach the end, return the else case. We just remove braces.

```
  \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}
```

Otherwise we compare (which evaluates #2 for us)

```
  \intexpr_compare:nNnTF{#1}={#2}
```

If true, we want to remove the remainder of the list, the else case and then execute the code specified. \prg_end_case:nw {#3} does just that in one go. This means f style expansion works the way one wants it to work.

```
  { \prg_end_case:nw {#3} }
  { \prg_case_int_aux:nnn {#1}}
}
```

\prg_case_dim:nnn
\prg_case_dim_aux:nnn

Same as \prg_case_dim:nnn except it is for ⟨*dim*⟩ registers.

```
\cs_new:Npn \prg_case_dim:nnn #1 #2 {
  \exp_args:No \prg_case_dim_aux:nnn {\dim_use:N \dim_eval:n{#1}} #2
  \q_recursion_tail ? \q_recursion_stop
}
\cs_new:Npn \prg_case_dim_aux:nnn #1#2#3{
  \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}
  \dim_compare:nNnTF{#1}={#2}
  { \prg_end_case:nw {#3} }
  { \prg_case_dim_aux:nnn {#1}}
}
```

\prg_case_str:nnn
\prg_case_str_aux:nnn

Same as \prg_case_dim:nnn except it is for strings.

```
\cs_new:Npn \prg_case_str:nnn #1 #2 {
  \prg_case_str_aux:nnn {#1} #2
  \q_recursion_tail ? \q_recursion_stop
}
\cs_new:Npn \prg_case_str_aux:nnn #1#2#3{
  \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}
  \tl_if_eq:xxTF{#1}{#2}
  { \prg_end_case:nw {#3} }
  { \prg_case_str_aux:nnn {#1}}
}
```

\prg_case_tl:Nnn
\prg_case_tl_aux:NNn

Same as \prg_case_dim:nnn except it is for token list variables.

```
\cs_new:Npn \prg_case_tl:Nnn #1 #2 {
  \prg_case_tl_aux:NNn #1 #2
  \q_recursion_tail ? \q_recursion_stop
}
\cs_new:Npn \prg_case_tl_aux:NNn #1#2#3{
  \quark_if_recursion_tail_stop_do:Nn #2{\use:n}
  \tl_if_eq:NNTF #1 #2
  {  \prg_end_case:nw {#3} }
  { \prg_case_tl_aux:NNn #1}
}
```

\prg_end_case:nw

Ending a case switch is always performed the same way so we optimize for this. #1 is the code to execute, #2 the remainder, and #3 the dangling else case.

```
\cs_new:Npn \prg_end_case:nw #1#2\q_recursion_stop#3{#1}
```

100.8 Sorting

\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.

```
\cs_new_protected_nopar:Npn \prg_define_quicksort:nnn #1#2#3 {
  \cs_set:cpx{#1_quicksort:n}##1{
    \exp_not:c{#1_quicksort_start_partition:w} ##1
    \exp_not:n{#2\q_nil#3\q_stop}
  }
  \cs_set:cpx{#1_quicksort_braced:n}##1{
    \exp_not:c{#1_quicksort_start_partition_braced:n} ##1
    \exp_not:N\q_nil\exp_not:N\q_stop
  }
  \cs_set:cpx {#1_quicksort_start_partition:w} #2 ##1 #3{
    \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
    \exp_not:c{#1_quicksort_do_partition_i:nnnw} {##1}{}{}
  }
  \cs_set:cpx {#1_quicksort_start_partition_braced:n} ##1 {
    \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
    \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn} {##1}{}{}
  }
```

Now for doing the partitions.

```
  \cs_set:cpx {#1_quicksort_do_partition_i:nnnw} ##1##2##3 #2 ##4 #3 {
    \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
    {
      \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
      \exp_not:c{#1_quicksort_partition_greater_ii:nnnn}
      \exp_not:c{#1_quicksort_partition_less_ii:nnnn}
    }
    {##1}{##2}{##3}{##4}
  }
  \cs_set:cpx {#1_quicksort_do_partition_i_braced:nnnn} ##1##2##3##4 {
    \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
    {
      \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
      \exp_not:c{#1_quicksort_partition_greater_ii_braced:nnnn}
      \exp_not:c{#1_quicksort_partition_less_ii_braced:nnnn}
    }
    {##1}{##2}{##3}{##4}
  }
  \cs_set:cpx {#1_quicksort_do_partition_ii:nnnw} ##1##2##3 #2 ##4 #3 {
    \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
    {
      \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
      \exp_not:c{#1_quicksort_partition_less_i:nnnn}
      \exp_not:c{#1_quicksort_partition_greater_i:nnnn}
    }
```

```
    {##1}{##2}{##3}{##4}
  }
  \cs_set:cpx {#1_quicksort_do_partition_ii_braced:nnnn} ##1##2##3##4 {
    \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
    {
      \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
      \exp_not:c{#1_quicksort_partition_less_i_braced:nnnn}
      \exp_not:c{#1_quicksort_partition_greater_i_braced:nnnn}
    }
    {##1}{##2}{##3}{##4}
  }
```

This part of the code handles the two branches in each sorting. Again we will also have to do it braced.

```
  \cs_set:cpx {#1_quicksort_partition_less_i:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##2}{{##4}##3}}
  \cs_set:cpx {#1_quicksort_partition_less_ii:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##3{##4}}}
  \cs_set:cpx {#1_quicksort_partition_greater_i:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{{##4}##2}{##3}}
  \cs_set:cpx {#1_quicksort_partition_greater_ii:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2{##4}}{##3}}
  \cs_set:cpx {#1_quicksort_partition_less_i_braced:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##2}{{##4}##3}}
  \cs_set:cpx {#1_quicksort_partition_less_ii_braced:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2}{##3{##4}}}
  \cs_set:cpx {#1_quicksort_partition_greater_i_braced:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{{##4}##2}{##3}}
  \cs_set:cpx {#1_quicksort_partition_greater_ii_braced:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2{##4}}{##3}}
```

Finally, the big kahuna! This is where the sub-lists are sorted.

```
  \cs_set:cpx {#1_do_quicksort_braced:nnnnw} ##1##2##3##4\q_stop {
    \exp_not:c{#1_quicksort_braced:n}{##2}
    \exp_not:c{#1_quicksort_function:n}{##1}
    \exp_not:c{#1_quicksort_braced:n}{##3}
  }
}
```

`\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.

```
\prg_define_quicksort:nnn {prg}{}{}
```

`\prg_quicksort_function:n`
`\prg_quicksort_compare:nnTF`

```
\cs_set:Npn \prg_quicksort_function:n {\ERROR}
\cs_set:Npn \prg_quicksort_compare:nnTF {\ERROR}
```

100.9 Variable type and scope

\prg_variable_get_scope:N
\prg_variable_get_scope_aux: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.

```
\group_begin:
  \tex_lccode:D `\& = `\g \tex_relax:D
  \tex_catcode:D `\& = \c_twelve \tex_relax:D
\tl_to_lowercase:n {
  \group_end:
  \cs_new_nopar:Npn \prg_variable_get_scope:N #1 {
    \exp_last_unbraced:Nf \prg_variable_get_scope_aux:w
    { \cs_to_str:N #1 \exp_stop_f: \q_nil }
  }
  \cs_new_nopar:Npn \prg_variable_get_scope_aux:w #1#2 \q_nil {
    \token_if_eq_meaning:NNT & #1 {g}
  }
}
\group_begin:
  \tex_lccode:D `\& = `\_ \tex_relax:D
  \tex_catcode:D `\& = \c_twelve \tex_relax:D
\tl_to_lowercase:n {
  \group_end:
  \cs_new_nopar:Npn \prg_variable_get_type:N #1 {
    \exp_after:wN \p;rg_variable_get_type_aux:w
      \token_to_str:N #1 & a \q_nil
  }
  \cs_new_nopar:Npn \prg_variable_get_type_aux:w #1 & #2#3 \q_nil {
    \token_if_eq_meaning:NNTF a #2 {
      #1
    }{
      \prg_variable_get_type_aux:w #2#3 \q_nil
    }
  }
}
```

That's it (for now).

```
⟨/initex | package⟩
```

```
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

101 l3quark implementation

We start by ensuring that the required packages are loaded. We check for `l3expan` since this a basic package that is essential for use of any higher-level package.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

`\quark_new:N`

Allocate a new quark.

```
\cs_new_protected_nopar:Npn \quark_new:N #1 { \tl_const:Nn #1 {#1} }
```

`\q_stop`
`\q_no_value`
`\q_nil`

`\q_stop` is often used as a marker in parameter text, `\q_no_value` is the canonical missing value, and `\q_nil` represents a nil pointer in some data structures.

```
\quark_new:N \q_stop
\quark_new:N \q_no_value
\quark_new:N \q_nil
```

`\q_error`
`\q_mark`

We need two additional quarks. `\q_error` delimits the end of the computation for purposes of error recovery. `\q_mark` is used in parameter text when we need a scanning boundary that is distinct from `\q_stop`.

```
\quark_new:N\q_error
\quark_new:N\q_mark
```

`\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.

```
\quark_new:N\q_recursion_tail
\quark_new:N\q_recursion_stop
```

`\quark_if_recursion_tail_stop:n`
`\quark_if_recursion_tail_stop:N`
`\quark_if_recursion_tail_stop:o`

When doing recursions it is easy to spend a lot of time testing if we found the end marker. To avoid this, we use a recursion end marker every time we do this kind of task. Also, if the recursion end marker is found, we wrap things up and finish.

```
\cs_new:Npn \quark_if_recursion_tail_stop:n #1 {
  \exp_after:wN\if_meaning:w
    \quark_if_recursion_tail_aux:w #1?\q_nil\q_recursion_tail\q_recursion_tail
    \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
  \fi:
}
```

```
\cs_new:Npn \quark_if_recursion_tail_stop:N #1 {
  \if_meaning:w#1\q_recursion_tail
    \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
  \fi:
}
\cs_generate_variant:Nn \quark_if_recursion_tail_stop:n {o}
```

rk_if_recursion_tail_stop_do:nn
rk_if_recursion_tail_stop_do:Nn
rk_if_recursion_tail_stop_do:on

```
\cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1#2 {
  \exp_after:wN\if_meaning:w
    \quark_if_recursion_tail_aux:w #1?\q_nil\q_recursion_tail\q_recursion_tail
    \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
  \else:
    \exp_after:wN\use_none:n
  \fi:
  {#2}
}
\cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1#2 {
  \if_meaning:w #1\q_recursion_tail
    \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
  \else:
    \exp_after:wN\use_none:n
  \fi:
  {#2}
}
\cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn {on}
```

\quark_if_recursion_tail_aux:w

```
\cs_new:Npn \quark_if_recursion_tail_aux:w #1#2 \q_nil \q_recursion_tail {#1}
```

\quark_if_no_value_p:N
\quark_if_no_value_p:n
\quark_if_no_value:N*TF*
\quark_if_no_value:n*TF*

Here we test if we found a special quark as the first argument. We better start with `\q_no_value` as the first argument since the whole thing may otherwise loop if `#1` is wrongly given a string like `aabc` instead of a single token.[11]

```
\prg_new_conditional:Nnn \quark_if_no_value:N {p,TF,T,F} {
  \if_meaning:w \q_no_value #1
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

We also provide an `n` type. If run under a sufficiently new pdfε-TeX, it uses a built-in primitive for string comparisons, otherwise it uses the slower `\str_if_eq_var_p:nf` function. In the latter case it would be faster to use a temporary token list variable but it would render the function non-expandable. Using the pdfε-TeX primitive is the preferred approach. Note that we have to add a manual space token in the first part of the comparison, otherwise it is gobbled by `\str_if_eq_var_p:nf`. The reason for using this

[11] It may still loop in special circumstances however!

function instead of \str_if_eq_p:nn is that a sequence like ␣\q_no_value will test equal to \q_no_value using the latter test function and unfortunately this example turned up in one application.

```
\cs_if_exist:cTF {pdf_strcmp:D}
{
  \prg_new_conditional:Nnn \quark_if_no_value:n {p,TF,T,F} {
    \if_num:w \pdf_strcmp:D
        {\exp_not:N \q_no_value}
        {\exp_not:n{#1}} = \c_zero
      \prg_return_true: \else: \prg_return_false:
    \fi:
  }
}
{
  \prg_new_conditional:Nnn \quark_if_no_value:n {p,TF,T,F} {
    \exp_args:NNo
    \if_predicate:w  \str_if_eq_var_p:nf
        {\token_to_str:N\q_no_value\c_space_tl}
        {\tl_to_str:n{#1}}
      \prg_return_true: \else: \prg_return_false:
    \fi:
  }
}
```

\quark_if_nil_p:N
\quark_if_nil:N*TF*

A function to check for the presence of \q_nil.

```
\prg_new_conditional:Nnn \quark_if_nil:N {p,TF,T,F} {
  \if_meaning:w \q_nil #1 \prg_return_true: \else: \prg_return_false: \fi:
}
```

\quark_if_nil_p:n
\quark_if_nil_p:V
\quark_if_nil_p:o
\quark_if_nil:n*TF*
\quark_if_nil:V*TF*
\quark_if_nil:o*TF*

A function to check for the presence of \q_nil.

```
\cs_if_exist:cTF {pdf_strcmp:D} {
  \prg_new_conditional:Nnn \quark_if_nil:n {p,TF,T,F} {
    \if_num:w \pdf_strcmp:D
        {\exp_not:N \q_nil}
        {\exp_not:n{#1}} = \c_zero
      \prg_return_true: \else: \prg_return_false:
    \fi:
  }
}
{
  \prg_new_conditional:Nnn \quark_if_nil:n {p,TF,T,F} {
    \exp_args:NNo
    \if_predicate:w \str_if_eq_var_p:nf
        {\token_to_str:N\q_nil\c_space_tl}
        {\tl_to_str:n{#1}}
      \prg_return_true: \else: \prg_return_false:
    \fi:
```

```
  }
}
\cs_generate_variant:Nn \quark_if_nil_p:n {V}
\cs_generate_variant:Nn \quark_if_nil:nTF {V}
\cs_generate_variant:Nn \quark_if_nil:nT  {V}
\cs_generate_variant:Nn \quark_if_nil:nF  {V}
\cs_generate_variant:Nn \quark_if_nil_p:n {o}
\cs_generate_variant:Nn \quark_if_nil:nTF {o}
\cs_generate_variant:Nn \quark_if_nil:nT  {o}
\cs_generate_variant:Nn \quark_if_nil:nF  {o}
```

Show token usage:

```

⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

102 l3token implementation

102.1 Documentation of internal functions

`\l_peek_true_tl`
`\l_peek_false_tl`

These token list variables are used internally when choosing either the true or false branches of a test.

`\l_peek_search_tl`

Used to store `\l_peek_search_token`.

`\peek_tmp:w`

Scratch function used to gobble tokens from the input stream.

`\l_peek_true_aux_tl`
`\c_peek_true_remove_next_tl`

These token list variables are used internally when choosing either the true or false branches of a test.

`\peek_ignore_spaces_execute_branches:`
`\peek_ignore_spaces_aux:`

Functions used to ignore space tokens in the input stream.

102.2 Module code

First a few required packages to get this going.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

102.3 Character tokens

\char_set_catcode:w
\char_set_catcode:nn
\char_value_catcode:w
\char_value_catcode:n
\char_show_value_catcode:w
\char_show_value_catcode:n

```
\cs_new_eq:NN \char_set_catcode:w \tex_catcode:D
\cs_new_protected_nopar:Npn \char_set_catcode:nn #1#2 {
  \char_set_catcode:w #1 = \intexpr_eval:w #2\intexpr_eval_end:
}
\cs_new_nopar:Npn \char_value_catcode:w { \int_use:N \tex_catcode:D }
\cs_new_nopar:Npn \char_value_catcode:n #1 {
  \char_value_catcode:w \intexpr_eval:w #1\intexpr_eval_end:
}
\cs_new_nopar:Npn \char_show_value_catcode:w {
  \tex_showthe:D \tex_catcode:D
}
\cs_new_nopar:Npn \char_show_value_catcode:n #1 {
  \char_show_value_catcode:w \intexpr_eval:w #1\intexpr_eval_end:
}
```

\char_make_escape:N
\char_make_begin_group:N
\char_make_end_group:N
\char_make_math_shift:N
\char_make_alignment: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
\char_make_invalid:N

```
\cs_new_protected_nopar:Npn \char_make_escape:N           #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_begin_group:N      #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_end_group:N        #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_math_shift:N       #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_alignment:N        #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_end_line:N         #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_parameter:N        #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_math_superscript:N #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_math_subscript:N   #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_ignore:N           #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_space:N            #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_letter:N           #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_other:N            #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_active:N           #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_comment:N          #1 { \char_set_catcode:nn {`#1} {\c_
\cs_new_protected_nopar:Npn \char_make_invalid:N          #1 { \char_set_catcode:nn {`#1} {\c_
```

\char_make_escape:n
\char_make_begin_group:n
\char_make_end_group:n
\char_make_math_shift:n
\char_make_alignment: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
\char_make_invalid:n

```
\cs_new_protected_nopar:Npn \char_make_escape:n          #1 { \char_set_catcode:nn {#1} {\c_z
\cs_new_protected_nopar:Npn \char_make_begin_group:n     #1 { \char_set_catcode:nn {#1} {\c_o
\cs_new_protected_nopar:Npn \char_make_end_group:n       #1 { \char_set_catcode:nn {#1} {\c_t
\cs_new_protected_nopar:Npn \char_make_math_shift:n      #1 { \char_set_catcode:nn {#1} {\c_t
\cs_new_protected_nopar:Npn \char_make_alignment:n       #1 { \char_set_catcode:nn {#1} {\c_f
\cs_new_protected_nopar:Npn \char_make_end_line:n        #1 { \char_set_catcode:nn {#1} {\c_f
\cs_new_protected_nopar:Npn \char_make_parameter:n       #1 { \char_set_catcode:nn {#1} {\c_s
\cs_new_protected_nopar:Npn \char_make_math_superscript:n #1 { \char_set_catcode:nn {#1} {\c_s
\cs_new_protected_nopar:Npn \char_make_math_subscript:n  #1 { \char_set_catcode:nn {#1} {\c_e
\cs_new_protected_nopar:Npn \char_make_ignore:n          #1 { \char_set_catcode:nn {#1} {\c_n
\cs_new_protected_nopar:Npn \char_make_space:n           #1 { \char_set_catcode:nn {#1} {\c_t
\cs_new_protected_nopar:Npn \char_make_letter:n          #1 { \char_set_catcode:nn {#1} {\c_e
\cs_new_protected_nopar:Npn \char_make_other:n           #1 { \char_set_catcode:nn {#1} {\c_t
\cs_new_protected_nopar:Npn \char_make_active:n          #1 { \char_set_catcode:nn {#1} {\c_t
\cs_new_protected_nopar:Npn \char_make_comment:n         #1 { \char_set_catcode:nn {#1} {\c_f
\cs_new_protected_nopar:Npn \char_make_invalid:n         #1 { \char_set_catcode:nn {#1} {\c_f
```

\char_set_mathcode:w
\char_set_mathcode:nn
\char_gset_mathcode:w
\char_gset_mathcode:nn
\char_value_mathcode:w
\char_value_mathcode:n
\char_show_value_mathcode:w
\char_show_value_mathcode:n

Math codes.

```
\cs_new_eq:NN \char_set_mathcode:w \tex_mathcode:D
\cs_new_protected_nopar:Npn \char_set_mathcode:nn #1#2 {
  \char_set_mathcode:w #1 = \intexpr_eval:w #2\intexpr_eval_end:
}
\cs_new_protected_nopar:Npn \char_gset_mathcode:w { \pref_global:D \tex_mathcode:D }
\cs_new_protected_nopar:Npn \char_gset_mathcode:nn #1#2 {
  \char_gset_mathcode:w #1 = \intexpr_eval:w #2\intexpr_eval_end:
}
\cs_new_nopar:Npn \char_value_mathcode:w { \int_use:N \tex_mathcode:D }
\cs_new_nopar:Npn \char_value_mathcode:n #1 {
  \char_value_mathcode:w \intexpr_eval:w #1\intexpr_eval_end:
}
\cs_new_nopar:Npn \char_show_value_mathcode:w { \tex_showthe:D \tex_mathcode:D }
\cs_new_nopar:Npn \char_show_value_mathcode:n #1 {
  \char_show_value_mathcode:w \intexpr_eval:w #1\intexpr_eval_end:
}
```

\char_set_lccode:w
\char_set_lccode:nn
\char_value_lccode:w
\char_value_lccode:n
\char_show_value_lccode:w
\char_show_value_lccode:n

```
\cs_new_eq:NN \char_set_lccode:w \tex_lccode:D
\cs_new_protected_nopar:Npn \char_set_lccode:nn #1#2{
  \char_set_lccode:w #1 = \intexpr_eval:w #2\intexpr_eval_end:
}
\cs_new_nopar:Npn \char_value_lccode:w {\int_use:N\tex_lccode:D}
\cs_new_nopar:Npn \char_value_lccode:n #1{\char_value_lccode:w
  \intexpr_eval:w #1\intexpr_eval_end:}
\cs_new_nopar:Npn \char_show_value_lccode:w {\tex_showthe:D\tex_lccode:D}
\cs_new_nopar:Npn \char_show_value_lccode:n #1{
  \char_show_value_lccode:w \intexpr_eval:w #1\intexpr_eval_end:}
```

\char_set_uccode:w
\char_set_uccode:nn
\char_value_uccode:w
\char_value_uccode:n
\char_show_value_uccode:w
\char_show_value_uccode:n

```
\cs_new_eq:NN \char_set_uccode:w \tex_uccode:D
\cs_new_protected_nopar:Npn \char_set_uccode:nn #1#2{
  \char_set_uccode:w #1 = \intexpr_eval:w #2\intexpr_eval_end:
}
\cs_new_nopar:Npn \char_value_uccode:w {\int_use:N\tex_uccode:D}
\cs_new_nopar:Npn \char_value_uccode:n #1{\char_value_uccode:w
  \intexpr_eval:w #1\intexpr_eval_end:}
\cs_new_nopar:Npn \char_show_value_uccode:w {\tex_showthe:D\tex_uccode:D}
\cs_new_nopar:Npn \char_show_value_uccode:n #1{
  \char_show_value_uccode:w \intexpr_eval:w #1\intexpr_eval_end:}
```

\char_set_sfcode:w
\char_set_sfcode:nn
\char_value_sfcode:w
\char_value_sfcode:n
\char_show_value_sfcode:w
\char_show_value_sfcode:n

```
\cs_new_eq:NN \char_set_sfcode:w \tex_sfcode:D
\cs_new_protected_nopar:Npn \char_set_sfcode:nn #1#2 {
  \char_set_sfcode:w #1 = \intexpr_eval:w #2\intexpr_eval_end:
}
\cs_new_nopar:Npn \char_value_sfcode:w { \int_use:N \tex_sfcode:D }
\cs_new_nopar:Npn \char_value_sfcode:n #1 {
  \char_value_sfcode:w \intexpr_eval:w #1\intexpr_eval_end:
}
\cs_new_nopar:Npn \char_show_value_sfcode:w { \tex_showthe:D \tex_sfcode:D }
\cs_new_nopar:Npn \char_show_value_sfcode:n #1 {
  \char_show_value_sfcode:w \intexpr_eval:w #1\intexpr_eval_end:
}
```

102.4 Generic tokens

\token_new:Nn

Creates a new token. (Will: why can't this just be `\cs_new_eq:NN \token_new:Nn \cs_gnew_eq:NN`? Seriously, that doesn't work!)

```
\cs_new_protected_nopar:Npn \token_new:Nn #1#2 {\cs_gnew_eq:NN #1#2}
```

\c_group_begin_token
\c_group_end_token
\c_math_shift_token
\c_alignment_tab_token
\c_parameter_token
\c_math_superscript_token
\c_math_subscript_token
\c_space_token
\c_letter_token
\c_other_char_token
\c_active_char_token

We define these useful tokens. We have to do it by hand with the brace tokens for obvious reasons.

```
\cs_new_eq:NN \c_group_begin_token {
\cs_new_eq:NN \c_group_end_token }
\group_begin:
\char_set_catcode:nn{`\*}{3}
\token_new:Nn \c_math_shift_token {*}
\char_set_catcode:nn{`\*}{4}
\token_new:Nn \c_alignment_tab_token {*}
\token_new:Nn \c_parameter_token {#}
\token_new:Nn \c_math_superscript_token {^}
\char_set_catcode:nn{`\*}{8}
```

```
\token_new:Nn \c_math_subscript_token {*}
\token_new:Nn \c_space_token {~}
\token_new:Nn \c_letter_token {a}
\token_new:Nn \c_other_char_token {1}
\char_set_catcode:nn{`\*}{13}
\cs_gset_nopar:Npn \c_active_char_token {\exp_not:N*}
\group_end:
```

\token_if_group_begin_p:N
\token_if_group_begin:NTF

Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.

```
\prg_new_conditional:Nnn \token_if_group_begin:N {p,TF,T,F} {
  \if_catcode:w \exp_not:N #1\c_group_begin_token
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_group_end_p:N
\token_if_group_end:NTF

Check if token is a end group token. We use the constant `\c_group_end_token` for this.

```
\prg_new_conditional:Nnn \token_if_group_end:N {p,TF,T,F} {
  \if_catcode:w \exp_not:N #1\c_group_end_token
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_math_shift_p:N
\token_if_math_shift:NTF

Check if token is a math shift token. We use the constant `\c_math_shift_token` for this.

```
\prg_new_conditional:Nnn \token_if_math_shift:N {p,TF,T,F} {
  \if_catcode:w \exp_not:N #1\c_math_shift_token
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_alignment_tab_p:N
\token_if_alignment_tab:NTF

Check if token is an alignment tab token. We use the constant `\c_alignment_tab_token` for this.

```
\prg_new_conditional:Nnn \token_if_alignment_tab:N {p,TF,T,F} {
  \if_catcode:w \exp_not:N #1\c_alignment_tab_token
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_parameter_p:N
\token_if_parameter:NTF

Check if token is a parameter token. We use the constant `\c_parameter_token` for this. We have to trick TeX a bit to avoid an error message.

```
\prg_new_conditional:Nnn \token_if_parameter:N {p,TF,T,F} {
  \exp_after:wN\if_catcode:w \cs:w c_parameter_token\cs_end:\exp_not:N #1
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_math_superscript_p:N
\token_if_math_superscript:N*TF*

Check if token is a math superscript token. We use the constant \c_math_superscript_token for this.

```
\prg_new_conditional:Nnn \token_if_math_superscript:N {p,TF,T,F} {
  \if_catcode:w \exp_not:N #1\c_math_superscript_token
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_math_subscript_p:N
\token_if_math_subscript:N*TF*

Check if token is a math subscript token. We use the constant \c_math_subscript_token for this.

```
\prg_new_conditional:Nnn \token_if_math_subscript:N {p,TF,T,F} {
  \if_catcode:w \exp_not:N #1\c_math_subscript_token
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_space_p:N
\token_if_space:N*TF*

Check if token is a space token. We use the constant \c_space_token for this.

```
\prg_new_conditional:Nnn \token_if_space:N {p,TF,T,F} {
  \if_catcode:w \exp_not:N #1\c_space_token
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_letter_p:N
\token_if_letter:N*TF*

Check if token is a letter token. We use the constant \c_letter_token for this.

```
\prg_new_conditional:Nnn \token_if_letter:N {p,TF,T,F} {
  \if_catcode:w \exp_not:N #1\c_letter_token
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_other_char_p:N
\token_if_other_char:N*TF*

Check if token is an other char token. We use the constant \c_other_char_token for this.

```
\prg_new_conditional:Nnn \token_if_other_char:N {p,TF,T,F} {
  \if_catcode:w \exp_not:N #1\c_other_char_token
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_active_char_p:N
\token_if_active_char:N*TF*

Check if token is an active char token. We use the constant \c_active_char_token for this.

```
\prg_new_conditional:Nnn \token_if_active_char:N {p,TF,T,F} {
  \if_catcode:w \exp_not:N #1\c_active_char_token
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_eq_meaning_p:NN
\token_if_eq_meaning:NN*TF*

Check if the tokens #1 and #2 have same meaning.

```
\prg_new_conditional:Nnn \token_if_eq_meaning:NN {p,TF,T,F} {
  \if_meaning:w  #1  #2
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_eq_catcode_p:NN
\token_if_eq_catcode:NNTF

Check if the tokens #1 and #2 have same category code.

```
\prg_new_conditional:Nnn \token_if_eq_catcode:NN {p,TF,T,F} {
  \if_catcode:w \exp_not:N #1 \exp_not:N #2
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_eq_charcode_p:NN
\token_if_eq_charcode:NNTF

Check if the tokens #1 and #2 have same character code.

```
\prg_new_conditional:Nnn \token_if_eq_charcode:NN {p,TF,T,F} {
  \if_charcode:w \exp_not:N #1 \exp_not:N #2
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\token_if_macro_p:N
\token_if_macro:NTF
\token_if_macro_p_aux:w

When a token is a macro, \token_to_meaning:N will always output something like \long macro:#1->#1 so we simply check to see if the meaning contains ->. Argument #2 in the code below will be empty if the string -> isn't present, proof that the token was not a macro (which is why we reverse the emptiness test). However this function will fail on its own auxiliary function (and a few other private functions as well) but that should certainly never be a problem!

```
\prg_new_conditional:Nnn \token_if_macro:N {p,TF,T,F} {
  \exp_after:wN \token_if_macro_p_aux:w \token_to_meaning:N #1 -> \q_nil
}
\cs_new_nopar:Npn \token_if_macro_p_aux:w #1 -> #2 \q_nil{
  \if_predicate:w \tl_if_empty_p:n{#2}
    \prg_return_false: \else: \prg_return_true: \fi:
}
```

\token_if_cs_p:N
\token_if_cs:NTF

Check if token has same catcode as a control sequence. We use \scan_stop: for this.

```
\prg_new_conditional:Nnn \token_if_cs:N {p,TF,T,F} {
  \if_predicate:w \token_if_eq_catcode_p:NN \scan_stop: #1
    \prg_return_true: \else: \prg_return_false: \fi:}
```

\token_if_expandable_p:N
\token_if_expandable:NTF

Check if token is expandable. We use the fact that TeX will temporarily convert \exp_not:N ⟨*token*⟩ into \scan_stop: if ⟨*token*⟩ is expandable.

```
\prg_new_conditional:Nnn \token_if_expandable:N {p,TF,T,F} {
  \cs_if_exist:NTF #1 {
    \exp_after:wN \if_meaning:w \exp_not:N #1 #1
      \prg_return_false: \else: \prg_return_true: \fi:
  } {
    \prg_return_false:
  }
}
```

\token_if_chardef_p:N
\token_if_mathchardef_p:N
\token_if_int_register_p:N
\token_if_skip_register_p:N
\token_if_dim_register_p:N
\token_if_toks_register_p:N
\token_if_protected_macro_p:N
\token_if_long_macro_p:N
ken_if_protected_long_macro_p:N
\token_if_chardef:NTF
\token_if_mathchardef:NTF
\token_if_long_macro:NTF
\token_if_protected_macro:NTF
ken_if_protected_long_macro:NTF
\token_if_dim_register:NTF
\token_if_skip_register:NTF
\token_if_int_register:NTF
\token_if_toks_register:NTF
\token_if_chardef_p_aux:w
\token_if_mathchardef_p_aux:w
\token_if_int_register_p_aux:w
\token_if_skip_register_p_aux:w
\token_if_dim_register_p_aux:w
\token_if_toks_register_p_aux:w
oken_if_protected_macro_p_aux:w
\token_if_long_macro_p_aux:w
if_protected_long_macro_p_aux:w

Most of these functions have to check the meaning of the token in question so we need to do some checkups on which characters are output by `\token_to_meaning:N`. As usual, these characters have catcode 12 so we must do some serious substitutions in the code below…

```
\group_begin:
  \char_set_lccode:nn {`\T}{`\T}
  \char_set_lccode:nn {`\F}{`\F}
  \char_set_lccode:nn {`\X}{`\n}
  \char_set_lccode:nn {`\Y}{`\t}
  \char_set_lccode:nn {`\Z}{`\d}
  \char_set_lccode:nn {`\?}{`\\}
  \tl_map_inline:nn{\X\Y\Z\M\C\H\A\R\O\U\S\K\I\P\L\G\P\E}
    {\char_set_catcode:nn {`#1}{12}}
```

We convert the token list to lowercase and restore the catcode and lowercase code changes.

```
\tl_to_lowercase:n{
  \group_end:
```

First up is checking if something has been defined with `\tex_chardef:D` or `\tex_mathchardef:D`. This is easy since TeX thinks of such tokens as hexadecimal so it stores them as `\char"`⟨*hex number*⟩ or `\mathchar"`⟨*hex number*⟩.

```
\prg_new_conditional:Nnn \token_if_chardef:N {p,TF,T,F} {
  \exp_after:wN \token_if_chardef_aux:w
  \token_to_meaning:N #1?CHAR"\q_nil
}
\cs_new_nopar:Npn \token_if_chardef_aux:w #1?CHAR"#2\q_nil{
  \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
}

\prg_new_conditional:Nnn \token_if_mathchardef:N {p,TF,T,F} {
  \exp_after:wN \token_if_mathchardef_aux:w
  \token_to_meaning:N #1?MAYHCHAR"\q_nil
}
\cs_new_nopar:Npn \token_if_mathchardef_aux:w #1?MAYHCHAR"#2\q_nil{
  \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
}
```

Integer registers are a little more difficult since they expand to `\count`⟨*number*⟩ and there is also a primitive `\countdef`. So we have to check for that primitive as well.

```
\prg_new_conditional:Nnn \token_if_int_register:N {p,TF,T,F} {
  \if_meaning:w \tex_countdef:D #1
    \prg_return_false:
  \else:
    \exp_after:wN \token_if_int_register_aux:w
      \token_to_meaning:N #1?COUXY\q_nil
  \fi:
```

```
}
\cs_new_nopar:Npn \token_if_int_register_aux:w #1?COUXY#2\q_nil{
  \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
}
```

Skip registers are done the same way as the integer registers.

```
\prg_new_conditional:Nnn \token_if_skip_register:N {p,TF,T,F} {
  \if_meaning:w \tex_skipdef:D #1
    \prg_return_false:
  \else:
    \exp_after:wN \token_if_skip_register_aux:w
    \token_to_meaning:N #1?SKIP\q_nil
  \fi:
}
\cs_new_nopar:Npn \token_if_skip_register_aux:w #1?SKIP#2\q_nil{
  \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
}
```

Dim registers. No news here

```
\prg_new_conditional:Nnn \token_if_dim_register:N {p,TF,T,F} {
  \if_meaning:w \tex_dimendef:D #1
    \c_false_bool
  \else:
    \exp_after:wN \token_if_dim_register_aux:w
    \token_to_meaning:N #1?ZIMEX\q_nil
  \fi:
}
\cs_new_nopar:Npn \token_if_dim_register_aux:w #1?ZIMEX#2\q_nil{
  \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
}
```

Toks registers.

```
\prg_new_conditional:Nnn \token_if_toks_register:N {p,TF,T,F} {
  \if_meaning:w \tex_toksdef:D #1
    \prg_return_false:
  \else:
    \exp_after:wN \token_if_toks_register_aux:w
    \token_to_meaning:N #1?YOKS\q_nil
  \fi:
}
\cs_new_nopar:Npn \token_if_toks_register_aux:w #1?YOKS#2\q_nil{
  \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
}
```

Protected macros.

```
\prg_new_conditional:Nnn \token_if_protected_macro:N {p,TF,T,F} {
  \exp_after:wN \token_if_protected_macro_aux:w
```

```
  \token_to_meaning:N #1?PROYECYEZ~MACRO\q_nil
}
\cs_new_nopar:Npn \token_if_protected_macro_aux:w #1?PROYECYEZ~MACRO#2\q_nil{
  \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
}
```

Long macros.

```
\prg_new_conditional:Nnn \token_if_long_macro:N {p,TF,T,F} {
  \exp_after:wN \token_if_long_macro_aux:w
  \token_to_meaning:N #1?LOXG~MACRO\q_nil
}
\cs_new_nopar:Npn \token_if_long_macro_aux:w #1?LOXG~MACRO#2\q_nil{
  \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
}
```

Finally protected long macros where we for once don't have to add an extra test since there is no primitive for the combined prefixes.

```
\prg_new_conditional:Nnn \token_if_protected_long_macro:N {p,TF,T,F} {
  \exp_after:wN \token_if_protected_long_macro_aux:w
  \token_to_meaning:N #1?PROYECYEZ?LOXG~MACRO\q_nil
}
\cs_new_nopar:Npn \token_if_protected_long_macro_aux:w #1
  ?PROYECYEZ?LOXG~MACRO#2\q_nil{
  \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
}
```

Finally the `\tl_to_lowercase:n` ends!

```
}
```

We do not provide a function for testing if a control sequence is "outer" since we don't use that in LaTeX3.

et_prefix_arg_replacement_aux:w
`\token_get_prefix_spec:N`
`\token_get_arg_spec:N`
`\token_get_replacement_spec:N`

In the xparse package 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.

```
\group_begin:
\char_set_lccode:nn {`\?}{`\:}
\char_set_catcode:nn{`\M}{12}
\char_set_catcode:nn{`\A}{12}
\char_set_catcode:nn{`\C}{12}
\char_set_catcode:nn{`\R}{12}
\char_set_catcode:nn{`\O}{12}
\tl_to_lowercase:n{
```

```
  \group_end:
  \cs_new_nopar:Npn \token_get_prefix_arg_replacement_aux:w #1MACRO?#2->#3\q_nil#4{
    #4{#1}{#2}{#3}
  }
  \cs_new_nopar:Npn\token_get_prefix_spec:N #1{
    \token_if_macro:NTF #1{
      \exp_after:wN \token_get_prefix_arg_replacement_aux:w
      \token_to_meaning:N #1\q_nil\use_i:nnn
    }{\scan_stop:}
  }
  \cs_new_nopar:Npn\token_get_arg_spec:N #1{
    \token_if_macro:NTF #1{
      \exp_after:wN \token_get_prefix_arg_replacement_aux:w
      \token_to_meaning:N #1\q_nil\use_ii:nnn
    }{\scan_stop:}
  }
  \cs_new_nopar:Npn\token_get_replacement_spec:N #1{
    \token_if_macro:NTF #1{
      \exp_after:wN \token_get_prefix_arg_replacement_aux:w
      \token_to_meaning:N #1\q_nil\use_iii:nnn
    }{\scan_stop:}
  }
}
```

Useless code: because we can!

\token_if_primitive_p:N
\token_if_primitive_p_aux:N
\token_if_primitive:NTF

It is rather hard to determine if a token is a primitive. First we can check if it is a control sequence or active character. If either, we check if it is a macro. Then we can go through a tedious process of testing for different register types. . . I don't actually think this function is useful but you never know.

```
\prg_new_conditional:Nnn \token_if_primitive:N {p,TF,T,F} {
  \if_predicate:w \token_if_cs_p:N #1
    \if_predicate:w \token_if_macro_p:N #1
      \prg_return_false:
    \else:
      \token_if_primitive_p_aux:N #1
    \fi:
  \else:
    \if_predicate:w \token_if_active_char_p:N #1
      \if_predicate:w \token_if_macro_p:N #1
        \prg_return_false:
      \else:
        \token_if_primitive_p_aux:N #1
      \fi:
    \else:
      \prg_return_false:
    \fi:
  \fi:
```

```
}
\cs_new_nopar:Npn \token_if_primitive_p_aux:N #1{
  \if_predicate:w \token_if_chardef_p:N #1 \c_false_bool
  \else:
    \if_predicate:w \token_if_mathchardef_p:N #1 \prg_return_false:
    \else:
      \if_predicate:w \token_if_int_register_p:N #1 \prg_return_false:
      \else:
        \if_predicate:w \token_if_skip_register_p:N #1 \prg_return_false:
        \else:
          \if_predicate:w \token_if_dim_register_p:N #1 \prg_return_false:
          \else:
            \if_predicate:w   \token_if_toks_register_p:N #1 \prg_return_false:
            \else:
```

We made it!

```
              \prg_return_true:
            \fi:
          \fi:
        \fi:
      \fi:
    \fi:
  \fi:
}
```

102.5 Peeking ahead at the next token

\l_peek_token
\g_peek_token
\l_peek_search_token

We define some other tokens which will initially be the character `?`.

```
\token_new:Nn \l_peek_token {?}
\token_new:Nn \g_peek_token {?}
\token_new:Nn \l_peek_search_token {?}
```

\peek_after:NN
\peek_gafter:NN

`\peek_after:NN` takes two argument where the first is a function acting on `\l_peek_token` and the second is the next token in the input stream which `\l_peek_token` is set equal to. `\peek_gafter:NN` does the same globally to `\g_peek_token`.

```
\cs_new_protected_nopar:Npn \peek_after:NN {\tex_futurelet:D \l_peek_token }
\cs_new_protected_nopar:Npn \peek_gafter:NN {
  \pref_global:D \tex_futurelet:D \g_peek_token
}
```

For normal purposes there are four main 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.

The generic functions will take four arguments: The token to search for, the test function to run on it and the true/false cases. The general algorithm is this:

1. Store the token to search for in `\l_peek_search_token`.
2. In order to avoid doubling of hash marks where it seems unnatural we put the ⟨*true*⟩ and ⟨*false*⟩ cases through an `x` type expansion but using `\exp_not:n` to avoid any expansion. This has the same effect as putting it through a ⟨*toks*⟩ register but is faster. Also put in a special alignment safe group end.
3. Put in an alignment safe group begin.
4. Peek ahead and call the function which will act on the next token in the input stream.

`\l_peek_true_tl` `\l_peek_false_tl` Two dedicated token list variables that store the true and false cases.

```
\tl_new:N \l_peek_true_tl
\tl_new:N \l_peek_false_tl
```

`\peek_tmp:w` Scratch function used for storing the token to be removed if found.

```
\cs_new_nopar:Npn \peek_tmp:w {}
```

`\l_peek_search_tl` We also use this token list variable for storing the token we want to compare. This turns out to be useful.

```
\tl_new:N \l_peek_search_tl
```

`\peek_token_generic:NNTF`
#1 : the function to execute (obey or ignore spaces, etc.),
#2 : the special token we're looking for.

```
\cs_new_protected:Npn \peek_token_generic:NNTF #1#2#3#4 {
  \cs_set_eq:NN \l_peek_search_token #2
  \tl_set:Nn \l_peek_search_tl {#2}
  \tl_set:Nn \l_peek_true_tl  { \group_align_safe_end: #3 }
  \tl_set:Nn \l_peek_false_tl { \group_align_safe_end: #4 }
  \group_align_safe_begin:
    \peek_after:NN #1
}
\cs_new_protected:Npn \peek_token_generic:NNT #1#2#3 {
  \peek_token_generic:NNTF #1#2 {#3} {}
}
\cs_new_protected:Npn \peek_token_generic:NNF #1#2#3 {
  \peek_token_generic:NNTF #1#2 {} {#3}
}
```

\peek_token_remove_generic:NNTF If we want to be able to remove any character from the input stream we might as well do it the same way for all characters so we define this as little differently from above.

```
\cs_new_protected:Npn \peek_token_remove_generic:NNTF #1#2#3#4 {
  \cs_set_eq:NN \l_peek_search_token #2
  \tl_set:Nn \l_peek_search_tl {#2}
  \tl_set:Nn \l_peek_true_aux_tl {#3}
  \tl_set_eq:NN \l_peek_true_tl \c_peek_true_remove_next_tl
  \tl_set:Nn \l_peek_false_tl {\group_align_safe_end: #4}
  \group_align_safe_begin:
    \peek_after:NN #1
}
\cs_new:Npn \peek_token_remove_generic:NNT #1#2#3 {
  \peek_token_remove_generic:NNTF #1#2 {#3} {}
}
\cs_new:Npn \peek_token_remove_generic:NNF #1#2#3 {
  \peek_token_remove_generic:NNTF #1#2 {} {#3}
}
```

\l_peek_true_aux_tl
\c_peek_true_remove_next_tl

Two token list variables to help with removing the character from the input stream.

```
\tl_new:N \l_peek_true_aux_tl
\tl_const:Nn \c_peek_true_remove_next_tl {\group_align_safe_end:
  \tex_afterassignment:D \l_peek_true_aux_tl \cs_set_eq:NN \peek_tmp:w
}
```

\peek_execute_branches_meaning:
\peek_execute_branches_catcode:
peek_execute_branches_charcode:
xecute_branches_charcode_aux:NN

There are three major tests between tokens in TEX: meaning, catcode and charcode. Hence we define three basic test functions that set in after the ignoring phase is over and done with.

```
\cs_new_nopar:Npn \peek_execute_branches_meaning: {
  \if_meaning:w \l_peek_token \l_peek_search_token
    \exp_after:wN \l_peek_true_tl
  \else:
    \exp_after:wN \l_peek_false_tl
  \fi:
}
\cs_new_nopar:Npn \peek_execute_branches_catcode: {
  \if_catcode:w \exp_not:N\l_peek_token \exp_not:N\l_peek_search_token
    \exp_after:wN \l_peek_true_tl
  \else:
    \exp_after:wN \l_peek_false_tl
  \fi:
}
```

For the charcode version we do things a little differently. We want to check the token directly but if we do this we face problems if the next thing in the input stream is a braced group or a space token. The braced group would be read as a complete argument and the space would be gobbled by TEX's argument reading routines. Hence we test for both of these and if one of them is found we just execute the false result directly since no one should ever try to use the `charcode` function for searching for `\c_group_begin_token` or `\c_space_token`. The same is true for `\c_group_end_token`, as this can only occur if the function is at the end of a group.

```
\cs_new_nopar:Npn \peek_execute_branches_charcode: {
  \bool_if:nTF {
    \token_if_eq_catcode_p:NN \l_peek_token \c_group_begin_token ||
    \token_if_eq_catcode_p:NN \l_peek_token \c_group_end_token ||
    \token_if_eq_meaning_p:NN \l_peek_token \c_space_token
  }
  { \l_peek_false_tl  }
```

Otherwise we call a small auxiliary function that just grabs the next token. We can do that because it really is a single token; we just have insert it again afterwards. Also we stored the token we were looking for in the token list variable `\l_peek_search_tl` so we unpack it again for this function.

```
  { \exp_after:wN \peek_execute_branches_charcode_aux:NN \l_peek_search_tl }
}
```

Then we just do the usual `\if_charcode:w` comparison. We also remember to insert `#2` again after executing the true or false branches.

```
\cs_new:Npn \peek_execute_branches_charcode_aux:NN #1#2{
  \if_charcode:w \exp_not:N #1\exp_not:N#2
    \exp_after:wN \l_peek_true_tl
  \else:
    \exp_after:wN \l_peek_false_tl
  \fi:
  #2
}
```

\peek_def_aux:nnnn
\peek_def_aux_ii:nnnnn

This function aids defining conditional variants without too much repeated code. I hope that it doesn't detract too much from the readability.

```
\cs_new_nopar:Npn \peek_def_aux:nnnn #1#2#3#4 {
  \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { TF }
  \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { T  }
  \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { F  }
}
\cs_new_protected_nopar:Npn \peek_def_aux_ii:nnnnn #1#2#3#4#5 {
  \cs_new_nopar:cpx { #1 #5 } {
    \tl_if_empty:nF {#2} {
      \exp_not:n { \cs_set_eq:NN \peek_execute_branches: #2 }
    }
    \exp_not:c { #3 #5 }
    \exp_not:n { #4 }
  }
}
```

\peek_meaning:NTF Here we use meaning comparison with \if_meaning:w.

```
\peek_def_aux:nnnn
  { peek_meaning:N }
  {}
  { peek_token_generic:NN }
  { \peek_execute_branches_meaning: }
```

\peek_meaning_ignore_spaces:NTF

```
\peek_def_aux:nnnn
  { peek_meaning_ignore_spaces:N }
  { \peek_execute_branches_meaning: }
  { peek_token_generic:NN }
  { \peek_ignore_spaces_execute_branches: }
```

\peek_meaning_remove:NTF

```
\peek_def_aux:nnnn
  { peek_meaning_remove:N }
  {}
  { peek_token_remove_generic:NN }
  { \peek_execute_branches_meaning: }
```

eaning_remove_ignore_spaces:NTF

```
\peek_def_aux:nnnn
  { peek_meaning_remove_ignore_spaces:N }
  { \peek_execute_branches_meaning: }
  { peek_token_remove_generic:NN }
  { \peek_ignore_spaces_execute_branches: }
```

\peek_catcode:NTF Here we use catcode comparison with \if_catcode:w.

```
\peek_def_aux:nnnn
  { peek_catcode:N }
  {}
  { peek_token_generic:NN }
  { \peek_execute_branches_catcode: }
```

\peek_catcode_ignore_spaces:NTF

```
\peek_def_aux:nnnn
  { peek_catcode_ignore_spaces:N }
  { \peek_execute_branches_catcode: }
  { peek_token_generic:NN }
  { \peek_ignore_spaces_execute_branches: }
```

\peek_catcode_remove:NTF

```
\peek_def_aux:nnnn
  { peek_catcode_remove:N }
  {}
  { peek_token_remove_generic:NN }
  { \peek_execute_branches_catcode: }
```

atcode_remove_ignore_spaces:NTF

```
\peek_def_aux:nnnn
  { peek_catcode_remove_ignore_spaces:N }
  { \peek_execute_branches_catcode: }
  { peek_token_remove_generic:NN }
  { \peek_ignore_spaces_execute_branches: }
```

\peek_charcode:NTF Here we use charcode comparison with `\if_charcode:w`.

```
\peek_def_aux:nnnn
  { peek_charcode:N }
  {}
  { peek_token_generic:NN }
  { \peek_execute_branches_charcode: }
```

peek_charcode_ignore_spaces:NTF

```
\peek_def_aux:nnnn
  { peek_charcode_ignore_spaces:N }
  { \peek_execute_branches_charcode: }
  { peek_token_generic:NN }
  { \peek_ignore_spaces_execute_branches: }
```

\peek_charcode_remove:NTF

```
\peek_def_aux:nnnn
  { peek_charcode_remove:N }
  {}
  { peek_token_remove_generic:NN }
  { \peek_execute_branches_charcode: }
```

arcode_remove_ignore_spaces:NTF

```
\peek_def_aux:nnnn
  { peek_charcode_remove_ignore_spaces:N }
  { \peek_execute_branches_charcode: }
  { peek_token_remove_generic:NN }
  { \peek_ignore_spaces_execute_branches:}
```

`\peek_ignore_spaces_aux:`
`ignore_spaces_execute_branches:`

Throw away a space token and search again. We could define this in a more devious way where the auxiliary function gobbles the space token but then what do we do if we decide that a certain function should ignore more than one specific token? For example someone might find it interesting to define a `\peek_` function that ignores `a`'s and `b`'s! Or maybe different kinds of "funny spaces"... Therefore I have decided to use this version which uses `\tex_afterassignment:D` to call the auxiliary function after the next token has been removed by `\cs_set_eq:NN`. That way it is easily extensible.

```
\cs_new_nopar:Npn \peek_ignore_spaces_aux: {
  \peek_after:NN \peek_ignore_spaces_execute_branches:
}
\cs_new_protected_nopar:Npn \peek_ignore_spaces_execute_branches: {
  \token_if_eq_meaning:NNTF \l_peek_token \c_space_token
  {  \tex_afterassignment:D \peek_ignore_spaces_aux:
     \cs_set_eq:NN \peek_tmp:w
  }
  \peek_execute_branches:
}
```

⟨/initex | package⟩

⟨*showmemory⟩

```
\showMemUsage
```

⟨/showmemory⟩

103 l3int implementation

103.1 Internal functions and variables

`\int_advance:w` — `\int_advance:w` ⟨*int register*⟩ ⟨*optional 'by'*⟩ ⟨*number*⟩ ⟨*space*⟩

Increments the count register by the specified amount.

TEXhackers note: This is TEX's `\advance`.

`\int_convert_number_to_letter:n` ⋆ — `\int_convert_number_to_letter:n` {⟨*integer expression*⟩}

Internal function for turning a number for a different base into a letter or digit.

`\int_pre_eval_one_arg:Nn`
`\int_pre_eval_two_args:Nnn`

`\int_pre_eval_one_arg:Nn` ⟨*function*⟩ {⟨*integer expression*⟩}
`\int_pre_eval_one_arg:Nnn` ⟨*function*⟩ {⟨*int* $expr_1$⟩} {⟨*int* $expr_2$⟩}

These are expansion helpers; they evaluate their integer expressions before handing them off to the specified ⟨*function*⟩.

\int_get_sign_and_digits:n ⋆	
\int_get_sign:n ⋆	
\int_get_digits:n ⋆	\int_get_sign_and_digits:n {⟨*number*⟩}

From an argument that may or may not include a + or - sign, these functions expand to the respective components of the number.

103.2 Module loading and primitives definitions

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

\int_to_roman:w
\int_to_number:w
\int_advance:w

A new name for the primitives.

```
\cs_new_eq:NN \int_to_roman:w \tex_romannumeral:D
\cs_new_eq:NN \int_to_number:w \tex_number:D
\cs_new_eq:NN \int_advance:w \tex_advance:D
```

Functions that support LaTeX's user accessible counters should be added here, too. But first the internal counters.

103.3 Allocation and setting

\int_new:N
\int_new:c
\int_new_local:N
\int_new_local:c

For the LaTeX3 format:

```
⟨*initex⟩
\alloc_new:nnnN {int} {11} {\c_max_register_int} \tex_countdef:D
⟨/initex⟩
```

For 'l3in2e':

```
⟨*package⟩
\cs_new_protected_nopar:Npn \int_new:N #1 {
  \chk_if_free_cs:N #1
  \newcount #1
}
\cs_new_protected_nopar:Npn \int_new_local:N #1 {
```

```
  \chk_if_free_cs:N #1
  \int_compare:nNnTF
    \tex_currentgrouplevel:D = 0
    \newcount \loccount
  #1
}
⟨/package⟩
```

```
\cs_generate_variant:Nn \int_new:N {c}
\cs_generate_variant:Nn \int_new_local:N {c}
```

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

Setting counters is again something that I would like to make uniform at the moment to get a better overview.

```
\cs_new_protected_nopar:Npn \int_set:Nn #1#2{#1 \intexpr_eval:w #2\intexpr_eval_end:
⟨*check⟩
\chk_local_or_pref_global:N #1
⟨/check⟩
}
\cs_new_protected_nopar:Npn \int_gset:Nn {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨-check⟩ \pref_global:D
   \int_set:Nn }
\cs_generate_variant:Nn\int_set:Nn  {cn}
\cs_generate_variant:Nn\int_gset:Nn {cn}
```

\int_incr:N
\int_decr:N
\int_gincr:N
\int_gdecr:N
\int_incr:c
\int_decr:c
\int_gincr:c
\int_gdecr:c

Incrementing and decrementing of integer registers is done with the following functions.

```
\cs_new_protected_nopar:Npn \int_incr:N #1{\int_advance:w#1\c_one
⟨*check⟩
        \chk_local_or_pref_global:N #1
⟨/check⟩
}
\cs_new_protected_nopar:Npn \int_decr:N #1{\int_advance:w#1\c_minus_one
⟨*check⟩
        \chk_local_or_pref_global:N #1
⟨/check⟩
}
\cs_new_protected_nopar:Npn \int_gincr:N {
```

We make sure that a local variable is not updated globally by changing the internal test (i.e. \chk_local_or_pref_global:N) before making the assignment. This is done by \pref_global_chk: which also issues the necessary \pref_global:D. This is not very efficient, but this code will be only included for debugging purposes. Using \pref_global:D in front of the local function is better in the production versions.

```
⟨*check⟩
```

```
    \pref_global_chk:
⟨/check⟩
⟨-check⟩ \pref_global:D
    \int_incr:N}
\cs_new_protected_nopar:Npn \int_gdecr:N {
⟨*check⟩
    \pref_global_chk:
⟨/check⟩
⟨-check⟩ \pref_global:D
    \int_decr:N}
```

With the `\int_add:Nn` functions we can shorten the above code. If this makes it too slow . . .

```
\cs_set_protected_nopar:Npn \int_incr:N #1{\int_add:Nn#1\c_one}
\cs_set_protected_nopar:Npn \int_decr:N #1{\int_add:Nn#1\c_minus_one}
\cs_set_protected_nopar:Npn \int_gincr:N #1{\int_gadd:Nn#1\c_one}
\cs_set_protected_nopar:Npn \int_gdecr:N #1{\int_gadd:Nn#1\c_minus_one}
```

```
\cs_generate_variant:Nn \int_incr:N {c}
\cs_generate_variant:Nn \int_decr:N {c}
\cs_generate_variant:Nn \int_gincr:N {c}
\cs_generate_variant:Nn \int_gdecr:N {c}
```

`\int_zero:N`
`\int_zero:c`
`\int_gzero:N`
`\int_gzero:c`

Functions that reset an ⟨*int*⟩ register to zero.

```
\cs_new_protected_nopar:Npn \int_zero:N  #1 {#1=\c_zero}
\cs_generate_variant:Nn \int_zero:N {c}
```

```
\cs_new_protected_nopar:Npn \int_gzero:N #1 {\pref_global:D #1=\c_zero}
\cs_generate_variant:Nn \int_gzero:N {c}
```

`\int_add:Nn`
`\int_add:cn`
`\int_gadd:Nn`
`\int_gadd:cn`
`\int_sub:Nn`
`\int_sub:cn`
`\int_gsub:Nn`
`\int_gsub:cn`

Adding and substracting to and from a counter . . . We should think of using these functions

```
\cs_new_protected_nopar:Npn \int_add:Nn #1#2{
```

We need to say `by` in case the first argument is a register accessed by its number, e.g., `\count23`. Not that it should ever happen but. . .

```
     \int_advance:w #1 by \intexpr_eval:w #2\intexpr_eval_end:
⟨*check⟩
     \chk_local_or_pref_global:N #1
⟨/check⟩
}
\cs_new_nopar:Npn \int_sub:Nn #1#2{
     \int_advance:w #1-\intexpr_eval:w #2\intexpr_eval_end:
⟨*check⟩
\chk_local_or_pref_global:N #1
⟨/check⟩
```

```
}
\cs_new_protected_nopar:Npn \int_gadd:Nn {
⟨*check⟩
  \pref_global_chk:
⟨/check⟩
⟨-check⟩ \pref_global:D
  \int_add:Nn }
\cs_new_protected_nopar:Npn \int_gsub:Nn {
⟨*check⟩
  \pref_global_chk:
⟨/check⟩
⟨-check⟩ \pref_global:D
  \int_sub:Nn }
\cs_generate_variant:Nn \int_add:Nn  {cn}
\cs_generate_variant:Nn \int_gadd:Nn {cn}
\cs_generate_variant:Nn \int_sub:Nn  {cn}
\cs_generate_variant:Nn \int_gsub:Nn {cn}
```

`\int_use:N`
`\int_use:c`

Here is how counters are accessed:

```
\cs_new_eq:NN \int_use:N \tex_the:D
\cs_new_nopar:Npn \int_use:c #1{\int_use:N \cs:w#1\cs_end:}
```

`\int_show:N`
`\int_show:c`

Diagnostics.

```
\cs_new_eq:NN  \int_show:N \tex_showthe:D
\cs_new_nopar:Npn \int_show:c {\exp_args:Nc \int_show:N }
```

`\int_to_arabic:n`

Nothing exciting here.

```
\cs_new_nopar:Npn \int_to_arabic:n #1{ \intexpr_eval:n{#1}}
```

`\int_roman_lcuc_mapping:Nnn`

Using TEX's built-in feature for producing roman numerals has some surprising features. One is the the characters resulting from `\int_to_roman:w` have category code 12 so they may fail in certain comparison tests. Therefore we use a mapping from the character TEX produces to the character we actually want which will give us letters with category code 11.

```
\cs_new_protected_nopar:Npn \int_roman_lcuc_mapping:Nnn #1#2#3{
  \cs_set_nopar:cpn {int_to_lc_roman_#1:}{#2}
  \cs_set_nopar:cpn {int_to_uc_roman_#1:}{#3}
}
```

Here are the default mappings. I haven't found any examples of say Turkish doing the mapping `i \i I` but at least there is a possibility for it if needed. Note: I have now asked a Turkish person and he tells me they do the `i I` mapping.

```
\int_roman_lcuc_mapping:Nnn i i I
```

```
\int_roman_lcuc_mapping:Nnn v v V
\int_roman_lcuc_mapping:Nnn x x X
\int_roman_lcuc_mapping:Nnn l l L
\int_roman_lcuc_mapping:Nnn c c C
\int_roman_lcuc_mapping:Nnn d d D
\int_roman_lcuc_mapping:Nnn m m M
```

For the delimiter we cheat and let it gobble its arguments instead.

```
\int_roman_lcuc_mapping:Nnn Q \use_none:nn \use_none:nn
```

`\int_to_roman:n`
`\int_to_Roman:n`
`\int_to_roman_lcuc:NN`

The commands for producing the lower and upper case roman numerals run a loop on one character at a time and also carries some information for upper or lower case with it. We put it through `\intexpr_eval:n` first which is safer and more flexible.

```
\cs_new_nopar:Npn \int_to_roman:n #1 {
  \exp_after:wN \int_to_roman_lcuc:NN \exp_after:wN l
    \int_to_roman:w \intexpr_eval:n {#1} Q
}
\cs_new_nopar:Npn \int_to_Roman:n #1 {
  \exp_after:wN \int_to_roman_lcuc:NN \exp_after:wN u
    \int_to_roman:w \intexpr_eval:n {#1} Q
}
\cs_new_nopar:Npn \int_to_roman_lcuc:NN #1#2{
  \use:c {int_to_#1c_roman_#2:}
  \int_to_roman_lcuc:NN #1
}
```

`nt_convert_number_with_rule:nnN`

This is our major workhorse for conversions. `#1` is the number we want converted, `#2` is the base number, and `#3` is the function converting the number. This function expects to receive a non-negative integer and as such is ideal for something using `\if_case:w` internally.

The basic example is this: We want to convert the number 50 (`#1`) into an alphabetic equivalent `ax`. For the English language our list contains 26 elements so this is our argument `#2` while the function `#3` just turns `1` into `a`, `2` into `b`, etc. Hence our goal is to turn 50 into the sequence `#3{1}#1{24}` so what we do is to first divide 50 by 26 and truncating the result returning 1. Then before we execute this we call the function again but this time on the result of the remainder of the division. This goes on until the remainder is less than or equal to the base number where we just call the function `#3` directly on the number.

We do a little pre-expansion of the arguments below as they otherwise have a tendency to grow quite large.

```
\cs_set_nopar:Npn \int_convert_number_with_rule:nnN #1#2#3{
  \intexpr_compare:nNnTF {#1}>{#2}
  {
    \exp_args:Nf \int_convert_number_with_rule:nnN
      { \intexpr_div_truncate:nn {#1-1}{#2} }{#2}
      #3
```

Note that we have to nudge our modulus function so it won't return 0 as that wouldn't work with `\if_case:w` when that expects a positive number to produce a letter.

```
    \exp_args:Nf #3 { \intexpr_eval:n{1+\intexpr_mod:nn {#1-1}{#2}} }
  }
  { \exp_args:Nf #3{ \intexpr_eval:n{#1} } }
}
```

As can be seen it is even simpler to convert to number systems that contain 0, since then we don't have to add or subtract 1 here and there.

`_alph_default_conversion_rule:n`
`_Alph_default_conversion_rule:n`

Now we just set up a default conversion rule. Ideally every language should have one such rule, as say in Danish there are 29 letters in the alphabet.

```
\cs_new_nopar:Npn \int_alph_default_conversion_rule:n #1{
  \if_case:w #1
    \or: a\or: b\or: c\or: d\or: e\or: f
    \or: g\or: h\or: i\or: j\or: k\or: l
    \or: m\or: n\or: o\or: p\or: q\or: r
    \or: s\or: t\or: u\or: v\or: w\or: x
    \or: y\or: z
  \fi:
}
\cs_new_nopar:Npn \int_Alph_default_conversion_rule:n #1{
  \if_case:w #1
    \or: A\or: B\or: C\or: D\or: E\or: F
    \or: G\or: H\or: I\or: J\or: K\or: L
    \or: M\or: N\or: O\or: P\or: Q\or: R
    \or: S\or: T\or: U\or: V\or: W\or: X
    \or: Y\or: Z
  \fi:
}
```

`\int_to_alph:n`
`\int_to_Alph:n`

The actual functions are just instances of the generic function. The second argument of `\int_convert_number_with_rule:nnN` should of course match the number of `\or:`s in the conversion rule.

```
\cs_new_nopar:Npn \int_to_alph:n #1{
  \int_convert_number_with_rule:nnN {#1}{26}
    \int_alph_default_conversion_rule:n
}
\cs_new_nopar:Npn \int_to_Alph:n #1{
  \int_convert_number_with_rule:nnN {#1}{26}
    \int_Alph_default_conversion_rule:n
}
```

`\int_to_symbol:n`

Turning a number into a symbol is also easy enough.

```
\cs_new_nopar:Npn \int_to_symbol:n #1{
```

```
  \mode_if_math:TF
  {
    \int_convert_number_with_rule:nnN {#1}{9}
      \int_symbol_math_conversion_rule:n
  }
  {
    \int_convert_number_with_rule:nnN {#1}{9}
      \int_symbol_text_conversion_rule:n
  }
}
```

`t_symbol_math_conversion_rule:n`
`t_symbol_text_conversion_rule:n`

Nothing spectacular here.

```
\cs_new_nopar:Npn \int_symbol_math_conversion_rule:n #1 {
  \if_case:w #1
    \or: *
    \or: \dagger
    \or: \ddagger
    \or: \mathsection
    \or: \mathparagraph
    \or: \|
    \or: **
    \or: \dagger\dagger
    \or: \ddagger\ddagger
  \fi:
}
\cs_new_nopar:Npn \int_symbol_text_conversion_rule:n #1 {
  \if_case:w #1
    \or: \textasteriskcentered
    \or: \textdagger
    \or: \textdaggerdbl
    \or: \textsection
    \or: \textparagraph
    \or: \textbardbl
    \or: \textasteriskcentered\textasteriskcentered
    \or: \textdagger\textdagger
    \or: \textdaggerdbl\textdaggerdbl
  \fi:
}
```

`\l_tmpa_int`
`\l_tmpb_int`
`\l_tmpc_int`
`\g_tmpa_int`
`\g_tmpb_int`

We provide four local and two global scratch counters, maybe we need more or less.

```
\int_new:N \l_tmpa_int
\int_new:N \l_tmpb_int
\int_new:N \l_tmpc_int
\int_new:N \g_tmpa_int
\int_new:N \g_tmpb_int
```

`\int_pre_eval_one_arg:Nn`
`\int_pre_eval_two_args:Nnn`

These are handy when handing down values to other functions. All they do is evaluate the number in advance.

```
\cs_set_nopar:Npn \int_pre_eval_one_arg:Nn #1#2{
  \exp_args:Nf#1{\intexpr_eval:n{#2}}}
\cs_set_nopar:Npn \int_pre_eval_two_args:Nnn #1#2#3{
  \exp_args:Nff#1{\intexpr_eval:n{#2}}{\intexpr_eval:n{#3}}
}
```

103.4 Defining constants

\int_const:Nn

As stated, most constants can be defined as \tex_chardef:D or \tex_mathchardef:D but that's engine dependent.

```
\cs_new_protected_nopar:Npn \int_const:Nn #1#2 {
  \intexpr_compare:nTF { #2 > \c_minus_one }
    {
      \intexpr_compare:nTF { #2 > \c_max_register_int }
        {
          \int_new:N #1
          \int_gset:Nn #1 {#2}
        }
        {
          \chk_if_free_cs:N #1
            \tex_global:D \tex_mathchardef:D #1 = \intexpr_eval:n {#2}
        }
    }
    {
      \int_new:N #1
      \int_gset:Nn #1 {#2}
    }
}
```

\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_hundred_one
\c_twohundred_fifty_five
\c_twohundred_fifty_six
\c_thousand
\c_ten_thousand
\c_ten_thousand_one
\c_ten_thousand_two
\c_ten_thousand_three

And the usual constants, others are still missing. Please, make every constant a real constant at least for the moment. We can easily convert things in the end when we have found what constants are used in critical places and what not.

```
 %% \tex_countdef:D \c_minus_one = 10 \scan_stop:
 %% \c_minus_one = -1 \scan_stop:        %% in l3basics
%\int_const:Nn \c_zero   {0}
\int_const:Nn \c_one    {1}
\int_const:Nn \c_two    {2}
\int_const:Nn \c_three  {3}
\int_const:Nn \c_four   {4}
\int_const:Nn \c_five   {5}
\int_const:Nn \c_six    {6}
\int_const:Nn \c_seven  {7}
\int_const:Nn \c_eight  {8}
\int_const:Nn \c_nine   {9}
\int_const:Nn \c_ten      {10}
\int_const:Nn \c_eleven   {11}
```

```
\int_const:Nn \c_twelve   {12}
\int_const:Nn \c_thirteen {13}
\int_const:Nn \c_fourteen {14}
\int_const:Nn \c_fifteen  {15}
 %% \tex_chardef:D \c_sixteen    = 16\scan_stop: %% in l3basics
\int_const:Nn \c_thirty_two {32}
```

The next one may seem a little odd (obviously!) but is useful when dealing with logical operators.

```
\int_const:Nn \c_hundred_one          {101}
\int_const:Nn \c_twohundred_fifty_five{255}
\int_const:Nn \c_twohundred_fifty_six {256}
\int_const:Nn \c_thousand             {1000}
\int_const:Nn \c_ten_thousand         {10000}
\int_const:Nn \c_ten_thousand_one     {10001}
\int_const:Nn \c_ten_thousand_two     {10002}
\int_const:Nn \c_ten_thousand_three   {10003}
\int_const:Nn \c_ten_thousand_four    {10004}
\int_const:Nn \c_twenty_thousand      {20000}
```

\c_max_int

The largest number allowed is $2^{31} - 1$

```
\int_const:Nn \c_max_int {2147483647}
```

103.5 Scanning and conversion

Conversion between different numbering schemes requires meticulous work. A number can be preceeded by any number of + and/or -. We define a generic function which will return the sign and/or the remainder.

\int_get_sign_and_digits:n
\int_get_sign:n
\int_get_digits:n
nt_get_sign_and_digits_aux:nNNN
nt_get_sign_and_digits_aux:oNNN

A number may be preceeded by any number of +s and -s. Start out by assuming we have a positive number.

```
\cs_new_nopar:Npn \int_get_sign_and_digits:n #1{
  \int_get_sign_and_digits_aux:nNNN {#1} \c_true_bool \c_true_bool \c_true_bool
}
\cs_new_nopar:Npn \int_get_sign:n #1{
  \int_get_sign_and_digits_aux:nNNN {#1} \c_true_bool \c_true_bool \c_false_bool
}
\cs_new_nopar:Npn \int_get_digits:n #1{
  \int_get_sign_and_digits_aux:nNNN {#1} \c_true_bool \c_false_bool \c_true_bool
}
```

Now check the first character in the string. Only a - can change if a number is positive or negative, hence we reverse the boolean governing this. Then gobble the - and start over.

```
\cs_new_nopar:Npn \int_get_sign_and_digits_aux:nNNN #1#2#3#4{
  \tl_if_head_eq_charcode:fNTF {#1} -
  {
    \bool_if:NTF #2
    { \int_get_sign_and_digits_aux:oNNN {\use_none:n #1} \c_false_bool #3#4 }
    { \int_get_sign_and_digits_aux:oNNN {\use_none:n #1} \c_true_bool  #3#4 }
  }
```

The other cases are much simpler since we either just have to gobble the `+` or exit immediately and insert the correct sign.

```
  {
    \tl_if_head_eq_charcode:fNTF {#1} +
    { \int_get_sign_and_digits_aux:oNNN {\use_none:n #1} #2#3#4}
    {
```

The boolean `#3` is for printing the sign while `#4` is for printing the digits.

```
      \bool_if:NT #3 { \bool_if:NF #2 - }
      \bool_if:NT #4 {#1}
    }
  }
}
\cs_generate_variant:Nn  \int_get_sign_and_digits_aux:nNNN {oNNN}
```

`\int_convert_from_base_ten:nn`
`t_convert_from_base_ten_aux:nnn`
`t_convert_from_base_ten_aux:non`
`t_convert_from_base_ten_aux:fon`

`#1` is the base 10 number to be converted to base `#2`. We split off the sign first, print if if there and then convert only the number. Since this is supposedly a base 10 number we can let TEX do the reading of `+` and `-`.

```
\cs_set_nopar:Npn \int_convert_from_base_ten:nn#1#2{
  \intexpr_compare:nNnTF {#1}<\c_zero
  {
    - \int_convert_from_base_ten_aux:nfn {}
    {  \intexpr_eval:n {-#1} }
  }
  {
    \int_convert_from_base_ten_aux:nfn {}
    {  \intexpr_eval:n {#1} }
  }
  {#2}
}
```

The algorithm runs like this:

1. If the number ⟨*num*⟩ is greater than ⟨*base*⟩, calculate modulus of ⟨*num*⟩ and ⟨*base*⟩ and carry that over for next round. The remainder is calculated as a truncated division of ⟨*num*⟩ and ⟨*base*⟩. Start over with these new values.

2. If ⟨*num*⟩ is less than or equal to ⟨*base*⟩ convert it to the correct symbol, print the previously calculated digits and exit.

#1 is the carried over result, #2 the remainder and #3 the base number.

```
\cs_new_nopar:Npn \int_convert_from_base_ten_aux:nnn#1#2#3{
  \intexpr_compare:nNnTF {#2}<{#3}
  { \int_convert_number_to_letter:n{#2} #1 }
  {
    \int_convert_from_base_ten_aux:ffn
    {
      \int_convert_number_to_letter:n {\intexpr_mod:nn {#2}{#3}}
      #1
    }
    { \intexpr_div_truncate:nn{#2}{#3}}
    {#3}
  }
}
\cs_generate_variant:Nn \int_convert_from_base_ten_aux:nnn {nfn}
\cs_generate_variant:Nn \int_convert_from_base_ten_aux:nnn {ffn}
```

\int_convert_number_to_letter:n Turning a number for a different base into a letter or digit.

```
\cs_set_nopar:Npn \int_convert_number_to_letter:n #1{
  \if_case:w \intexpr_eval:w #1-10\intexpr_eval_end:
  \exp_after:wN A \or: \exp_after:wN B \or:
  \exp_after:wN C \or: \exp_after:wN D \or: \exp_after:wN E \or:
  \exp_after:wN F \or: \exp_after:wN G \or: \exp_after:wN H \or:
  \exp_after:wN I \or: \exp_after:wN J \or: \exp_after:wN K \or:
  \exp_after:wN L \or: \exp_after:wN M \or: \exp_after:wN N \or:
  \exp_after:wN O \or: \exp_after:wN P \or: \exp_after:wN Q \or:
  \exp_after:wN R \or: \exp_after:wN S \or: \exp_after:wN T \or:
  \exp_after:wN U \or: \exp_after:wN V \or: \exp_after:wN W \or:
  \exp_after:wN X \or: \exp_after:wN Y \or: \exp_after:wN Z \else:
  \use_i_after_fi:nw{ #1 }\fi: }
```

\int_convert_to_base_ten:nn #1 is the number, #2 is its base. First we get the sign, then use only the digits/letters from it and pass that onto a new function.

```
\cs_set_nopar:Npn \int_convert_to_base_ten:nn #1#2 {
  \intexpr_eval:n{
    \int_get_sign:n{#1}
    \exp_args:Nf\int_convert_to_base_ten_aux:nn {\int_get_digits:n{#1}}{#2}
  }
}
```

This is an intermediate function to get things started.

```
\cs_new_nopar:Npn \int_convert_to_base_ten_aux:nn #1#2{
  \int_convert_to_base_ten_auxi:nnN {0}{#2} #1 \q_nil
}
```

Here we check each letter/digit and calculate the next number. `#1` is the previously calculated result (to be multiplied by the base), `#2` is the base and `#3` is the next letter/digit to be added.

```
\cs_new_nopar:Npn \int_convert_to_base_ten_auxi:nnN#1#2#3{
  \quark_if_nil:NTF #3
  {#1}
  {\exp_args:Nf\int_convert_to_base_ten_auxi:nnN
    {\intexpr_eval:n{ #1*#2+\int_convert_letter_to_number:N #3} }
    {#2}
  }
}
```

This is for turning a letter or digit into a number. This function also takes care of handling lowercase and uppercase letters. Hence `a` is turned into `11` and so is `A`.

```
\cs_set_nopar:Npn \int_convert_letter_to_number:N #1{
  \intexpr_compare:nNnTF{`#1}<{58}{#1}
  {
    \intexpr_eval:n{ `#1 -
      \intexpr_compare:nNnTF{`#1}<{91}{ 55 }{ 87 }
    }
  }
}
```

Needed from the tl module:

```
\int_new:N \g_tl_inline_level_int
```

```
⟨/initex | package⟩
```

Show token usage:

```
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

104 l3intexpr implementation

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

\if_num:w
\if_case:w

Here are the remaining primitives for number comparisons and expressions.

```
\cs_new_eq:NN \if_num:w           \tex_ifnum:D
\cs_new_eq:NN \if_case:w          \tex_ifcase:D
```

\intexpr_value:w
\intexpr_eval:n
\intexpr_eval:w
\intexpr_eval_end:
\if_intexpr_compare:w
\if_intexpr_odd:w
\if_intexpr_case:w

Here are the remaining primitives for number comparisons and expressions.

```
\cs_set_eq:NN \intexpr_value:w \tex_number:D
\cs_set_eq:NN \intexpr_eval:w \etex_numexpr:D
\cs_set_protected:Npn \intexpr_eval_end: {\tex_relax:D}
\cs_set_eq:NN \if_intexpr_compare:w \tex_ifnum:D
\cs_set_eq:NN \if_intexpr_odd:w \tex_ifodd:D
\cs_set_eq:NN \if_intexpr_case:w \tex_ifcase:D
\cs_set:Npn \intexpr_eval:n #1{
  \intexpr_value:w \intexpr_eval:w #1\intexpr_eval_end:
}
```

\intexpr_compare_p:n
\intexpr_compare:nTF

Comparison tests using a simple syntax where only one set of braces is required and additional operators such as != and >= are supported. First some notes on the idea behind this. We wish to support writing code like

```
\intexpr_compare_p:n { 5 + \l_tmpa_int != 4 - \l_tmpb_int }
```

In other words, we want to somehow add the missing \intexpr_eval:w where required. We can start evaluating from the left using \intexpr:w, and we know that since the relation symbols <, >, = and ! are not allowed in such expressions, they will terminate the expression. Therefore, we first let TEX evaluate this left hand side of the (in)equality.

```
\prg_set_conditional:Npnn \intexpr_compare:n #1{p,TF,T,F}{
  \exp_after:wN \intexpr_compare_auxi:w \intexpr_value:w
    \intexpr_eval:w #1\q_stop
}
```

Then the next step is to figure out which relation we should use, so we have to somehow get rid of the first evaluation so that we can see what stopped it. \tex_romannumeral:D is handy here since its expansion given a non-positive number is ⟨*null*⟩. We therefore simply check if the first token of the left hand side evaluation is a minus. If not, we insert it and issue \tex_romannumeral:D, thereby ridding us of the left hand side evaluation. We do however save it for later.

```
\cs_set:Npn \intexpr_compare_auxi:w #1#2\q_stop{
   \exp_after:wN   \intexpr_compare_auxii:w \tex_romannumeral:D
   \if:w #1- \else: -\fi: #1#2 \q_stop #1#2 \q_nil
}
```

This leaves the first relation symbol in front and assuming the right hand side has been input, at least one other token as well. We support the following forms: =, <, > and the extended `!=`, `==`, `<=` and `>=`. All the extended forms have an extra `=` so we check if that is present as well. Then use specific function to perform the test.

```
\cs_set:Npn \intexpr_compare_auxii:w #1#2#3\q_stop{
   \use:c{
     intexpr_compare_
     #1  \if_meaning:w =#2 =  \fi:
     :w}
}
```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument. Equality is easy:

```
\cs_set:cpn {intexpr_compare_=:w} #1=#2\q_nil{
  \if_intexpr_compare:w #1=\intexpr_eval:w #2 \intexpr_eval_end:
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

So is the one using `==` – we just have to use `==` in the parameter text.

```
\cs_set:cpn {intexpr_compare_==:w} #1==#2\q_nil{
  \if_intexpr_compare:w #1=\intexpr_eval:w #2 \intexpr_eval_end:
  \prg_return_true: \else: \prg_return_false: \fi:
}
```

Not equal is just about reversing the truth value.

```
\cs_set:cpn {intexpr_compare_!=:w} #1!=#2\q_nil{
  \if_intexpr_compare:w #1=\intexpr_eval:w #2 \intexpr_eval_end:
  \prg_return_false: \else: \prg_return_true: \fi:
}
```

Less than and greater than are also straight forward.

```
\cs_set:cpn {intexpr_compare_<:w} #1<#2\q_nil{
  \if_intexpr_compare:w #1<\intexpr_eval:w #2 \intexpr_eval_end:
  \prg_return_true: \else: \prg_return_false: \fi:
}
\cs_set:cpn {intexpr_compare_>:w} #1>#2\q_nil{
  \if_intexpr_compare:w #1>\intexpr_eval:w #2 \intexpr_eval_end:
  \prg_return_true: \else: \prg_return_false: \fi:
}
```

The less than or equal operation is just the opposite of the greater than operation. Vice versa for less than or equal.

```
\cs_set:cpn {intexpr_compare_<=:w} #1<=#2\q_nil{
  \if_intexpr_compare:w #1>\intexpr_eval:w #2 \intexpr_eval_end:
  \prg_return_false: \else: \prg_return_true: \fi:
}
\cs_set:cpn {intexpr_compare_>=:w} #1>=#2\q_nil{
```

```
  \if_intexpr_compare:w #1<\intexpr_eval:w #2 \intexpr_eval_end:
  \prg_return_false: \else: \prg_return_true: \fi:
}
```

\intexpr_compare_p:nNn
\intexpr_compare:nNnTF

More efficient but less natural in typing.

```
\prg_set_conditional:Npnn \intexpr_compare:nNn #1#2#3{p,TF,T,F}{
  \if_intexpr_compare:w \intexpr_eval:w #1 #2 \intexpr_eval:w #3
  \intexpr_eval_end:
  \prg_return_true: \else: \prg_return_false: \fi:
}
```

\intexpr_max:nn
\intexpr_min:nn
\intexpr_abs:n

Functions for min, max, and absolute value.

```
\cs_set:Npn \intexpr_abs:n #1{
  \intexpr_value:w
  \if_intexpr_compare:w \intexpr_eval:w #1<\c_zero
    -
  \fi:
  \intexpr_eval:w #1\intexpr_eval_end:
}
\cs_set:Npn \intexpr_max:nn #1#2{
  \intexpr_value:w \intexpr_eval:w
    \if_intexpr_compare:w
      \intexpr_eval:w #1>\intexpr_eval:w #2\intexpr_eval_end:
      #1
    \else:
      #2
    \fi:
  \intexpr_eval_end:
}
\cs_set:Npn \intexpr_min:nn #1#2{
  \intexpr_value:w \intexpr_eval:w
    \if_intexpr_compare:w
      \intexpr_eval:w #1<\intexpr_eval:w #2\intexpr_eval_end:
      #1
    \else:
      #2
    \fi:
  \intexpr_eval_end:
}
```

\intexpr_div_truncate:nn
\intexpr_div_round:nn
\intexpr_mod:nn

As \intexpr_eval:w rounds the result of a division we also provide a version that truncates the result.

Initial version didn't work correctly with eTEX's implementation.

```
%\cs_set:Npn \intexpr_div_truncate_raw:nn #1#2 {
%  \intexpr_eval:n{ (2*#1 - #2) / (2* #2) }
%}
```

New version by Heiko:

```
\cs_set:Npn \intexpr_div_truncate:nn #1#2 {
  \intexpr_value:w \intexpr_eval:w
    \if_intexpr_compare:w \intexpr_eval:w #1 = \c_zero
      0
    \else:
      (#1
      \if_intexpr_compare:w \intexpr_eval:w #1 < \c_zero
        \if_intexpr_compare:w \intexpr_eval:w #2 < \c_zero
          -( #2 +
        \else:
          +( #2 -
        \fi:
      \else:
        \if_intexpr_compare:w \intexpr_eval:w #2 < \c_zero
          +( #2 +
        \else:
          -( #2 -
        \fi:
      \fi:
      1)/2)
    \fi:
    /(#2)
  \intexpr_eval_end:
}
```

For the sake of completeness:

```
\cs_set:Npn \intexpr_div_round:nn #1#2 {\intexpr_eval:n{(#1)/(#2)}}
```

Finally there's the modulus operation.

```
\cs_set:Npn \intexpr_mod:nn #1#2 {
  \intexpr_value:w
    \intexpr_eval:w
    #1 - \intexpr_div_truncate:nn {#1}{#2} * (#2)
    \intexpr_eval_end:
}
```

`\intexpr_if_odd_p:n`
`\intexpr_if_odd:nTF`
`\intexpr_if_even_p:n`
`\intexpr_if_even:nTF`

A predicate function.

```
\prg_set_conditional:Npnn \intexpr_if_odd:n #1 {p,TF,T,F} {
  \if_intexpr_odd:w \intexpr_eval:w #1\intexpr_eval_end:
    \prg_return_true: \else: \prg_return_false: \fi:
}
\prg_set_conditional:Npnn \intexpr_if_even:n #1 {p,TF,T,F} {
  \if_intexpr_odd:w \intexpr_eval:w #1\intexpr_eval_end:
    \prg_return_false: \else: \prg_return_true: \fi:
}
```

\intexpr_while_do:nn
\intexpr_until_do:nn
\intexpr_do_while:nn
\intexpr_do_until: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.

```
\cs_set:Npn \intexpr_while_do:nn #1#2{
  \intexpr_compare:nT {#1}{#2 \intexpr_while_do:nn {#1}{#2}}
}
\cs_set:Npn \intexpr_until_do:nn #1#2{
  \intexpr_compare:nF {#1}{#2 \intexpr_until_do:nn {#1}{#2}}
}
\cs_set:Npn \intexpr_do_while:nn #1#2{
  #2 \intexpr_compare:nT {#1}{\intexpr_do_while:nNnn {#1}{#2}}
}
\cs_set:Npn \intexpr_do_until:nn #1#2{
  #2 \intexpr_compare:nF {#1}{\intexpr_do_until:nn {#1}{#2}}
}
```

\intexpr_while_do:nNnn
\intexpr_until_do:nNnn
\intexpr_do_while:nNnn
\intexpr_do_until:nNnn

As above but not using the more natural syntax.

```
\cs_set:Npn \intexpr_while_do:nNnn #1#2#3#4{
  \intexpr_compare:nNnT {#1}#2{#3}{#4 \intexpr_while_do:nNnn {#1}#2{#3}{#4}}
}
\cs_set:Npn \intexpr_until_do:nNnn #1#2#3#4{
  \intexpr_compare:nNnF {#1}#2{#3}{#4 \intexpr_until_do:nNnn {#1}#2{#3}{#4}}
}
\cs_set:Npn \intexpr_do_while:nNnn #1#2#3#4{
  #4 \intexpr_compare:nNnT {#1}#2{#3}{\intexpr_do_while:nNnn {#1}#2{#3}{#4}}
}
\cs_set:Npn \intexpr_do_until:nNnn #1#2#3#4{
  #4 \intexpr_compare:nNnF {#1}#2{#3}{\intexpr_do_until:nNnn {#1}#2{#3}{#4}}
}
```

\c_max_register_int

This is here as this particular integer is needed both in package mode and to bootstrap l3alloc

```
\tex_mathchardef:D \c_max_register_int = 32767 \scan_stop:
```

⟨/initex | package⟩

105 l3skip implementation

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

105.1 Skip registers

\skip_new:N
\skip_new:c
\skip_new_local:N
\skip_new_local:c

Allocation of a new internal registers.

```
⟨*initex⟩
\alloc_new:nnnN {skip} \c_zero \c_max_register_int \tex_skipdef:D
⟨/initex⟩
⟨*package⟩
\cs_new_protected_nopar:Npn \skip_new:N #1 {
  \chk_if_free_cs:N #1
  \newskip #1
}
\cs_new_protected_nopar:Npn \skip_new_local:N #1 {
  \chk_if_free_cs:N #1
  \int_compare:nNnTF
    \tex_currentgrouplevel:D = 0
    \newskip \locskip
  #1
}
⟨/package⟩
\cs_generate_variant:Nn \skip_new:N {c}
\cs_generate_variant:Nn \skip_new_local:N {c}
```

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

Setting skips is again something that I would like to make uniform at the moment to get a better overview.

```
\cs_new_protected_nopar:Npn \skip_set:Nn #1#2 {
  #1\skip_eval:n{#2}
⟨*check⟩
\chk_local_or_pref_global:N #1
⟨/check⟩
}
\cs_new_protected_nopar:Npn \skip_gset:Nn {
⟨*check⟩
  \pref_global_chk:
⟨/check⟩
⟨-check⟩ \pref_global:D
  \skip_set:Nn
}
\cs_generate_variant:Nn \skip_set:Nn  {cn}
\cs_generate_variant:Nn \skip_gset:Nn  {cn}
```

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

Reset the register to zero.

```
\cs_new_protected_nopar:Npn \skip_zero:N #1{
  #1\c_zero_skip \scan_stop:
⟨*check⟩
  \chk_local_or_pref_global:N #1
⟨/check⟩
```

```
}
\cs_new_protected_nopar:Npn \skip_gzero:N {
```

We make sure that a local variable is not updated globally by changing the internal test (i.e. `\chk_local_or_pref_global:N`) before making the assignment. This is done by `\pref_global_chk:` which also issues the necessary `\pref_global:D`. This is not very efficient, but this code will be only included for debugging purposes. Using `\pref_global:D` in front of the local function is better in the production versions.

```
⟨*check⟩
  \pref_global_chk:
⟨/check⟩
⟨-check⟩ \pref_global:D
  \skip_zero:N
}
\cs_generate_variant:Nn \skip_zero:N {c}
\cs_generate_variant:Nn \skip_gzero:N {c}
```

`\skip_add:Nn`
`\skip_add:cn`
`\skip_gadd:Nn`
`\skip_gadd:cn`
`\skip_sub:Nn`
`\skip_gsub:Nn`

Adding and subtracting to and from <skip>s

```
\cs_new_protected_nopar:Npn \skip_add:Nn #1#2 {
```

We need to say `by` in case the first argment is a register accessed by its number, e.g., `\skip23`.

```
  \tex_advance:D#1 by \skip_eval:n{#2}
⟨*check⟩
  \chk_local_or_pref_global:N #1
⟨/check⟩
}
\cs_generate_variant:Nn \skip_add:Nn {cn}

\cs_new_protected_nopar:Npn \skip_sub:Nn #1#2{
  \tex_advance:D#1-\skip_eval:n{#2}
⟨*check⟩
  \chk_local_or_pref_global:N #1
⟨/check⟩
}

\cs_new_protected_nopar:Npn \skip_gadd:Nn {
⟨*check⟩
  \pref_global_chk:
⟨/check⟩
⟨-check⟩ \pref_global:D
  \skip_add:Nn
}
\cs_generate_variant:Nn \skip_gadd:Nn {cn}

\cs_new_nopar:Npn \skip_gsub:Nn {
⟨*check⟩
```

```
  \pref_global_chk:
⟨/check⟩
⟨-check⟩ \pref_global:D
  \skip_sub:Nn
}
```

\skip_horizontal:N
\skip_horizontal:c
\skip_horizontal:n
\skip_vertical:N
\skip_vertical:c
\skip_vertical:n

Inserting skips.

```
\cs_new_eq:NN  \skip_horizontal:N \tex_hskip:D
\cs_generate_variant:Nn \skip_horizontal:N {c}
\cs_new_nopar:Npn \skip_horizontal:n #1 { \skip_horizontal:N \skip_eval:n{#1} }
\cs_new_eq:NN  \skip_vertical:N \tex_vskip:D
\cs_generate_variant:Nn \skip_vertical:N {c}
\cs_new_nopar:Npn \skip_vertical:n #1 { \skip_vertical:N \skip_eval:n{#1} }
```

\skip_use:N
\skip_use:c

Here is how skip registers are accessed:

```
\cs_new_eq:NN \skip_use:N \tex_the:D
\cs_generate_variant:Nn \skip_use:N {c}
```

\skip_show:N
\skip_show:c

Diagnostics.

```
\cs_new_eq:NN  \skip_show:N \tex_showthe:D
\cs_new_nopar:Npn \skip_show:c #1 { \skip_show:N \cs:w #1 \cs_end: }
```

\skip_eval:n

Evaluating a calc expression.

```
\cs_new_protected_nopar:Npn \skip_eval:n #1 { \etex_glueexpr:D #1 \scan_stop: }
```

\l_tmpa_skip
\l_tmpb_skip
\l_tmpc_skip
\g_tmpa_skip
\g_tmpb_skip

We provide three local and two global scratch registers, maybe we need more or less.

```
%%\chk_if_free_cs:N \l_tmpa_skip
%%\tex_skipdef:D\l_tmpa_skip 255  %currently taken up by \skip@
\skip_new:N \l_tmpa_skip
\skip_new:N \l_tmpb_skip
\skip_new:N \l_tmpc_skip
\skip_new:N \g_tmpa_skip
\skip_new:N \g_tmpb_skip
```

\c_zero_skip
\c_max_skip

```
⟨*!package⟩
\skip_new:N \c_zero_skip
\skip_set:Nn \c_zero_skip {0pt}
\skip_new:N \c_max_skip
\skip_set:Nn \c_max_skip {16383.99999pt}
⟨/!package⟩
⟨*!initex⟩
\cs_set_eq:NN \c_zero_skip \z@
\cs_set_eq:NN \c_max_skip  \maxdimen
⟨/!initex⟩
```

\skip_if_infinite_glue_p:n
\skip_if_infinite_glue:nTF

With ε-TEX we all of a sudden get access to a lot information we should otherwise consider ourselves lucky to get. One is the stretch and shrink components of a skip register and the order or those components. \skip_if_infinite_glue:nTF tests it directly by looking at the stretch and shrink order. If either of the predicate functions return ⟨*true*⟩ \bool_if:nTF will return ⟨*true*⟩ and the logic test will take the true branch.

```
\prg_new_conditional:Nnn \skip_if_infinite_glue:n {p,TF,T,F} {
  \bool_if:nTF {
      \intexpr_compare_p:nNn {\etex_gluestretchorder:D #1 } > \c_zero ||
      \intexpr_compare_p:nNn {\etex_glueshrinkorder:D  #1 } > \c_zero
  } {\prg_return_true:} {\prg_return_false:}
}
```

p_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 global.

```
\cs_new_nopar:Npn \skip_split_finite_else_action:nnNN #1#2#3#4{
  \skip_if_infinite_glue:nTF {#1}
  {
    #3 = \c_zero_skip
    #4 = \c_zero_skip
    #2
  }
  {
    #3 = \etex_gluestretch:D #1 \scan_stop:
    #4 = \etex_glueshrink:D  #1 \scan_stop:
  }
}
```

105.2 Dimen registers

\dim_new:N
\dim_new:c
\dim_new_local:N
\dim_new_local:c

Allocating ⟨*dim*⟩ registers...

```
⟨*initex⟩
\alloc_new:nnnN {dim} \c_zero \c_max_register_int \tex_dimendef:D
⟨/initex⟩
⟨*package⟩
\cs_new_protected_nopar:Npn \dim_new:N #1 {
  \chk_if_free_cs:N #1
  \newdimen #1
}
\cs_new_protected_nopar:Npn \dim_new_local:N #1 {
  \chk_if_free_cs:N #1
  \int_compare:nNnTF
    \tex_currentgrouplevel:D = 0
    \newdimen \locdimen
```

```
  #1
}
⟨/package⟩
\cs_generate_variant:Nn \dim_new:N {c}
\cs_generate_variant:Nn \dim_new_local:N {c}
```

\dim_set:Nn
\dim_set:cn
\dim_set:Nc
\dim_gset:Nn
\dim_gset:cn
\dim_gset:Nc
\dim_gset:cc

We add `\dim_eval:n` in order to allow simple arithmetic and a space just for those using `\dimen1` or alike. See OR!

```
\cs_new_protected_nopar:Npn \dim_set:Nn #1#2 { #1~ \dim_eval:n{#2} }
\cs_generate_variant:Nn \dim_set:Nn {cn,Nc}

\cs_new_protected_nopar:Npn \dim_gset:Nn { \pref_global:D \dim_set:Nn  }
\cs_generate_variant:Nn \dim_gset:Nn {cn,Nc,cc}
```

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

Resetting.

```
\cs_new_protected_nopar:Npn \dim_zero:N #1 { #1\c_zero_skip }
\cs_generate_variant:Nn \dim_zero:N {c}

\cs_new_protected_nopar:Npn \dim_gzero:N { \pref_global:D \dim_zero:N  }
\cs_generate_variant:Nn \dim_gzero:N {c}
```

\dim_add:Nn
\dim_add:cn
\dim_add:Nc
\dim_gadd:Nn
\dim_gadd:cn

Addition.

```
\cs_new_protected_nopar:Npn \dim_add:Nn #1#2{
```

We need to say `by` in case the first argment is a register accessed by its number, e.g., `\dimen23`.

```
    \tex_advance:D#1 by \dim_eval:n{#2}\scan_stop:
}
\cs_generate_variant:Nn \dim_add:Nn {cn,Nc}

\cs_new_protected_nopar:Npn \dim_gadd:Nn { \pref_global:D \dim_add:Nn  }
\cs_generate_variant:Nn \dim_gadd:Nn {cn}
```

\dim_sub:Nn
\dim_sub:cn
\dim_sub:Nc
\dim_gsub:Nn
\dim_gsub:cn

Subtracting.

```
\cs_new_protected_nopar:Npn \dim_sub:Nn #1#2 { \tex_advance:D#1-#2\scan_stop: }
\cs_generate_variant:Nn \dim_sub:Nn {cn,Nc}

\cs_new_protected_nopar:Npn \dim_gsub:Nn { \pref_global:D \dim_sub:Nn  }
\cs_generate_variant:Nn \dim_gsub:Nn {cn}
```

\dim_use:N
\dim_use:c

Accessing a ⟨*dim*⟩.

```
\cs_new_eq:NN \dim_use:N \tex_the:D
\cs_generate_variant:Nn \dim_use:N {c}
```

\dim_show:N
\dim_show:c

Diagnostics.

```
\cs_new_eq:NN  \dim_show:N \tex_showthe:D
\cs_new_nopar:Npn \dim_show:c #1 { \dim_show:N \cs:w #1 \cs_end: }
```

\l_tmpa_dim
\l_tmpb_dim
\l_tmpc_dim
\l_tmpd_dim
\g_tmpa_dim
\g_tmpb_dim

Some scratch registers.

```
\dim_new:N \l_tmpa_dim
\dim_new:N \l_tmpb_dim
\dim_new:N \l_tmpc_dim
\dim_new:N \l_tmpd_dim
\dim_new:N \g_tmpa_dim
\dim_new:N \g_tmpb_dim
```

\c_zero_dim
\c_max_dim

Just aliases.

```
\cs_new_eq:NN \c_zero_dim \c_zero_skip
\cs_new_eq:NN \c_max_dim  \c_max_skip
```

\dim_eval:n

Evaluating a calc expression.

```
\cs_new_protected_nopar:Npn \dim_eval:n #1 { \etex_dimexpr:D #1 \scan_stop: }
```

\if_dim:w

The comparison primitive.

```
\cs_new_eq:NN \if_dim:w \tex_ifdim:D
```

\dim_compare_p:nNn
\dim_compare:nNn*TF*

```
\prg_new_conditional:Nnn \dim_compare:nNn {p,TF,T,F} {
  \if_dim:w \dim_eval:n {#1} #2 \dim_eval:n {#3}
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

\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.

```
\cs_new_nopar:Npn \dim_while_do:nNnn #1#2#3#4{
  \dim_compare:nNnT {#1}#2{#3}{#4 \dim_while_do:nNnn {#1}#2{#3}{#4}}
}
\cs_new_nopar:Npn \dim_until_do:nNnn #1#2#3#4{
  \dim_compare:nNnF {#1}#2{#3}{#4 \dim_until_do:nNnn {#1}#2{#3}{#4}}
}
\cs_new_nopar:Npn \dim_do_while:nNnn #1#2#3#4{
  #4 \dim_compare:nNnT {#1}#2{#3}{\dim_do_while:nNnn {#1}#2{#3}{#4}}
}
\cs_new_nopar:Npn \dim_do_until:nNnn #1#2#3#4{
  #4 \dim_compare:nNnF {#1}#2{#3}{\dim_do_until:nNnn {#1}#2{#3}{#4}}
}
```

105.3 Muskips

\muskip_new:N
\muskip_new_local:N

And then we add muskips.

```
⟨*initex⟩
\alloc_new:nnnN {muskip} \c_zero \c_max_register_int \tex_muskipdef:D
⟨/initex⟩
⟨*package⟩
\cs_new_protected_nopar:Npn \muskip_new:N #1 {
  \chk_if_free_cs:N #1
  \newmuskip #1
}
\cs_new_protected_nopar:Npn \muskip_new_local:N #1 {
  \chk_if_free_cs:N #1
  \int_compare:nNnTF
    \tex_currentgrouplevel:D = 0
    \newmuskip \locmuskip
  #1
}
⟨/package⟩
```

\muskip_set:Nn
\muskip_gset:Nn
\muskip_add:Nn
\muskip_gadd:Nn
\muskip_sub:Nn
\muskip_gsub:Nn

Simple functions for muskips.

```
\cs_new_protected_nopar:Npn \muskip_set:Nn#1#2{#1\etex_muexpr:D#2\scan_stop:}
\cs_new_protected_nopar:Npn \muskip_gset:Nn{\pref_global:D\muskip_set:Nn}
\cs_new_protected_nopar:Npn \muskip_add:Nn#1#2{\tex_advance:D#1\etex_muexpr:D#2\scan_stop:}
\cs_new_protected_nopar:Npn \muskip_gadd:Nn{\pref_global:D\muskip_add:Nn}
\cs_new_protected_nopar:Npn \muskip_sub:Nn#1#2{\tex_advance:D#1-\etex_muexpr:D#2\scan_stop:}
\cs_new_protected_nopar:Npn \muskip_gsub:Nn{\pref_global:D\muskip_sub:Nn}
```

\muskip_use:N

Accessing a ⟨*muskip*⟩.

```
\cs_new_eq:NN \muskip_use:N \tex_the:D
```

```
⟨/initex | package⟩
```

106 l3tl implementation

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
```

```
⟨*initex | package⟩
```

A token list variable is a control sequence that holds tokens. The interface is similar to that for token registers, but beware that the behavior vis á vis `\cs_set_nopar:Npx` etc. . . . is different. (You see this comes from Denys' implementation.)

106.1 Functions

\tl_new:N
\tl_new:c
\tl_new:Nn
\tl_new:cn
\tl_new:Nx

We provide one allocation function (which checks that the name is not used) and two clear functions that locally or globally clear the token list. The allocation function has two arguments to specify an initial value. This is the only way to give values to constants.

```
\cs_new_protected:Npn \tl_new:Nn #1#2{
  \chk_if_free_cs:N #1
```

If checking we don't allow constants to be defined.

```
⟨*check⟩
  \chk_var_or_const:N #1
⟨/check⟩
```

Otherwise any variable type is allowed.

```
  \cs_gset_nopar:Npn #1{#2}
}
\cs_generate_variant:Nn \tl_new:Nn {cn}
\cs_new_protected:Npn \tl_new:Nx #1#2{
  \chk_if_free_cs:N #1
⟨check⟩ \chk_var_or_const:N #1
  \cs_gset_nopar:Npx #1{#2}
}
\cs_new_protected_nopar:Npn \tl_new:N #1{\tl_new:Nn #1{}}
\cs_new_protected_nopar:Npn \tl_new:c #1{\tl_new:cn {#1}{}}
```

\tl_const:Nn

For creating constant token lists: there is not actually anything here that cannot be achieved using \tl_new:N and \tl_set:Nn

```
\cs_new_protected:Npn \tl_const:Nn #1#2 {
  \tl_new:N   #1
  \tl_gset:Nn #1 {#2}
}
```

\tl_use:N
\tl_use:c

Perhaps this should just be enabled when checking?

```
\cs_new_nopar:Npn \tl_use:N #1 {
  \if_meaning:w #1 \tex_relax:D
```

If ⟨*tl var.*⟩ equals \tex_relax:D it is probably stemming from a \cs:w...\cs_end: that was created by mistake somewhere.

```
      \msg_kernel_bug:x {Token~list~variable~ `\token_to_str:N #1'~
                         has~ an~ erroneous~ structure!}
  \else:
    \exp_after:wN #1
  \fi:
}
\cs_generate_variant:Nn \tl_use:N {c}
```

\tl_show:N
\tl_show:c
\tl_show:n

Showing a ⟨*tl var.*⟩ is just \showing it and I don't really care about checking that it's malformed at this stage.

```
\cs_new_nopar:Npn \tl_show:N #1 { \cs_show:N #1 }
\cs_generate_variant:Nn \tl_show:N {c}
\cs_set_eq:NN \tl_show:n \etex_showtokens:D
```

\tl_set:Nn
\tl_set:NV
\tl_set:Nv
\tl_set:No
\tl_set:Nf
\tl_set:Nx
\tl_set:cn
\tl_set:cV
\tl_set:cv
\tl_set:co
\tl_set:cx
\tl_gset:Nn
\tl_gset:NV
\tl_gset:Nv
\tl_gset:No
\tl_gset:Nf
\tl_gset:Nx
\tl_gset:cn
\tl_gset:cV
\tl_gset:cv
\tl_gset:cx

By using \exp_not:n token list variables can contain # tokens.

```
\cs_new_protected:Npn \tl_set:Nn #1#2 {
  \cs_set_nopar:Npx #1 { \exp_not:n {#2} }
}
\cs_new_protected:Npn \tl_set:Nx #1#2 {
  \cs_set_nopar:Npx #1 {#2}
}
\cs_new_protected:Npn \tl_gset:Nn #1#2 {
  \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
}
\cs_new_protected:Npn \tl_gset:Nx #1#2 {
  \cs_gset_nopar:Npx #1 {#2}
}
\cs_generate_variant:Nn \tl_set:Nn  { NV }
\cs_generate_variant:Nn \tl_set:Nn  { Nv }
\cs_generate_variant:Nn \tl_set:Nn  { No }
\cs_generate_variant:Nn \tl_set:Nn  { Nf }
\cs_generate_variant:Nn \tl_set:Nn  { cV }
\cs_generate_variant:Nn \tl_set:Nn  { c }
\cs_generate_variant:Nn \tl_set:Nn  { cv }
\cs_generate_variant:Nn \tl_set:Nn  { co }
\cs_generate_variant:Nn \tl_set:Nx  { c }
\cs_generate_variant:Nn \tl_gset:Nn { NV }
\cs_generate_variant:Nn \tl_gset:Nn { Nv }
\cs_generate_variant:Nn \tl_gset:Nn { No }
\cs_generate_variant:Nn \tl_gset:Nn { Nf }
\cs_generate_variant:Nn \tl_gset:Nn { c }
\cs_generate_variant:Nn \tl_gset:Nn { cV }
\cs_generate_variant:Nn \tl_gset:Nn { cv }
\cs_generate_variant:Nn \tl_gset:Nx { c }
```

\tl_set_eq:NN
\tl_set_eq:Nc
\tl_set_eq:cN
\tl_set_eq:cc
\tl_gset_eq:NN
\tl_gset_eq:Nc
\tl_gset_eq:cN
\tl_gset_eq:cc

For setting token list variables equal to each other. First checking:

```
⟨*check⟩
\cs_new_protected_nopar:Npn \tl_set_eq:NN #1#2{
  \chk_exist_cs:N #1  \cs_set_eq:NN #1#2
  \chk_local_or_pref_global:N #1  \chk_var_or_const:N #2
}
\cs_new_protected_nopar:Npn \tl_gset_eq:NN #1#2{
  \chk_exist_cs:N #1  \cs_gset_eq:NN #1#2
  \chk_global:N #1  \chk_var_or_const:N #2
}
⟨/check⟩
```

Non-checking versions are easy.

```
⟨*!check⟩
\cs_new_eq:NN \tl_set_eq:NN  \cs_set_eq:NN
\cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
⟨/!check⟩
```

The rest again with the expansion module.

```
\cs_generate_variant:Nn \tl_set_eq:NN {Nc,c,cc}
\cs_generate_variant:Nn \tl_gset_eq:NN {Nc,c,cc}
```

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

Clearing a token list variable.

```
\cs_new_protected_nopar:Npn \tl_clear:N #1{\tl_set_eq:NN #1\c_empty_tl}
\cs_generate_variant:Nn \tl_clear:N {c}
\cs_new_protected_nopar:Npn \tl_gclear:N #1{\tl_gset_eq:NN #1\c_empty_tl}
\cs_generate_variant:Nn \tl_gclear:N {c}
```

\tl_clear_new:N
\tl_clear_new:c

These macros check whether a token list exists. If it does it is cleared, if it doesn't it is allocated.

```
⟨*check⟩
\cs_new_protected_nopar:Npn \tl_clear_new:N #1{
  \chk_var_or_const:N #1
  \if_predicate:w \cs_if_exist_p:N #1
    \tl_clear:N #1
  \else:
    \tl_new:N #1
  \fi:
}
⟨/check⟩
⟨-check⟩\cs_new_eq:NN \tl_clear_new:N \tl_clear:N
\cs_generate_variant:Nn \tl_clear_new:N {c}
```

\tl_gclear_new:N
\tl_gclear_new:c

These are the global versions of the above.

```
⟨*check⟩
\cs_new_protected_nopar:Npn \tl_gclear_new:N #1{
  \chk_var_or_const:N #1
  \if_predicate:w \cs_if_exist_p:N #1
    \tl_gclear:N #1
  \else:
    \tl_new:N #1
  \fi:}
⟨/check⟩
⟨-check⟩\cs_new_eq:NN \tl_gclear_new:N \tl_gclear:N
\cs_generate_variant:Nn \tl_gclear_new:N {c}
```

\tl_put_right:Nn
\tl_put_right:NV
\tl_put_right:Nv
\tl_put_right:No
\tl_put_right:Nx
\tl_put_right:cn
\tl_put_right:cV
\tl_put_right:cv
\tl_put_right:cx
\tl_gput_right:Nn
\tl_gput_right:NV
\tl_gput_right:Nv
\tl_gput_right:No
\tl_gput_right:Nx
\tl_gput_right:cn
\tl_gput_right:cV
\tl_gput_right:cv
\tl_gput_right:co
\tl_gput_right:cx

Adding to one end of a token list is done partially using hand tuned functions for performance reasons.

```
\cs_new_protected:Npn \tl_put_right:Nn #1#2 {
  \cs_set_nopar:Npx #1 { \exp_not:V #1 \exp_not:n {#2} }
}
\cs_new_protected:Npn \tl_put_right:NV #1#2 {
  \cs_set_nopar:Npx #1 { \exp_not:V #1 \exp_not:V #2 }
}
\cs_new_protected:Npn \tl_put_right:Nv #1#2 {
  \cs_set_nopar:Npx #1 { \exp_not:V #1 \exp_not:v {#2} }
}
\cs_new_protected:Npn \tl_put_right:Nx #1#2 {
  \cs_set_nopar:Npx #1 { \exp_not:V #1 #2 }
}
\cs_new_protected:Npn \tl_put_right:No #1#2 {
  \cs_set_nopar:Npx #1 { \exp_not:V #1 \exp_not:o {#2} }
}
\cs_new_protected:Npn \tl_gput_right:Nn #1#2 {
  \cs_gset_nopar:Npx #1 { \exp_not:V #1 \exp_not:n {#2} }
}
\cs_new_protected:Npn \tl_gput_right:NV #1#2 {
  \cs_gset_nopar:Npx #1 { \exp_not:V #1 \exp_not:V #2 }
}
\cs_new_protected:Npn \tl_gput_right:Nv #1#2 {
  \cs_gset_nopar:Npx #1 { \exp_not:V #1 \exp_not:v {#2} }
}
\cs_new_protected:Npn \tl_gput_right:No #1#2 {
  \cs_gset_nopar:Npx #1 { \exp_not:V #1 \exp_not:o {#2} }
}
\cs_new_protected:Npn \tl_gput_right:Nx #1#2 {
  \cs_gset_nopar:Npx #1 { \exp_not:V #1 #2 }
}
\cs_generate_variant:Nn \tl_put_right:Nn  { c }
\cs_generate_variant:Nn \tl_put_right:NV  { c }
\cs_generate_variant:Nn \tl_put_right:Nv  { c }
\cs_generate_variant:Nn \tl_put_right:Nx  { c }
\cs_generate_variant:Nn \tl_gput_right:Nn { c }
\cs_generate_variant:Nn \tl_gput_right:NV { c }
\cs_generate_variant:Nn \tl_gput_right:Nv { c }
\cs_generate_variant:Nn \tl_gput_right:No { c }
\cs_generate_variant:Nn \tl_gput_right:Nx { c }
```

\tl_put_left:Nn
\tl_put_left:NV
\tl_put_left:Nv
\tl_put_left:No
\tl_put_left:Nx
\tl_put_left:cn
\tl_put_left:cV
\tl_put_left:cv
\tl_put_left:cx
\tl_gput_left:Nn
\tl_gput_left:NV
\tl_gput_left:Nv
\tl_gput_left:No
\tl_gput_left:Nx
\tl_gput_left:cn
\tl_gput_left:cV

Adding to the left is basically the same as putting on the right.

```
\cs_new_protected:Npn \tl_put_left:Nn #1#2 {
  \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:V #1  }
}
\cs_new_protected:Npn \tl_put_left:NV #1#2 {
  \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:V #1 }
```

```
}
\cs_new_protected:Npn \tl_put_left:Nv #1#2 {
  \cs_set_nopar:Npx #1 { \exp_not:v {#2} \exp_not:V #1 }
}
\cs_new_protected:Npn \tl_put_left:Nx #1#2 {
  \cs_set_nopar:Npx #1 { #2 \exp_not:V #1 }
}
\cs_new_protected:Npn \tl_put_left:No #1#2 {
  \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:V #1 }
}
\cs_new_protected:Npn \tl_gput_left:Nn #1#2 {
  \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:V #1 }
}
\cs_new_protected:Npn \tl_gput_left:NV #1#2 {
  \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:V #1 }
}
\cs_new_protected:Npn \tl_gput_left:Nv #1#2 {
  \cs_gset_nopar:Npx #1 { \exp_not:v {#2} \exp_not:V #1 }
}
\cs_new_protected:Npn \tl_gput_left:No #1#2 {
  \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:V #1 }
}
\cs_new_protected:Npn \tl_gput_left:Nx #1#2 {
  \cs_gset_nopar:Npx #1 { #2 \exp_not:V #1 }
}
\cs_generate_variant:Nn \tl_put_left:Nn  { c }
\cs_generate_variant:Nn \tl_put_left:NV  { c }
\cs_generate_variant:Nn \tl_put_left:Nv  { c }
\cs_generate_variant:Nn \tl_put_left:Nx  { c }
\cs_generate_variant:Nn \tl_gput_left:Nn { c }
\cs_generate_variant:Nn \tl_gput_left:NV { c }
\cs_generate_variant:Nn \tl_gput_left:Nv { c }
\cs_generate_variant:Nn \tl_gput_left:Nx { c }
```

\tl_gset:Nc
\tl_set:Nc

These two functions are included because they are necessary in Denys' implementations. The :Nc convention (see the expansion module) is very unusual at first sight, but it works nicely over all modules, so we would like to keep it.

Construct a control sequence on the fly from #2 and save it in #1.

```
\cs_new_protected_nopar:Npn \tl_gset:Nc {
⟨*check⟩
  \pref_global_chk:
⟨/check⟩
⟨-check⟩ \pref_global:D
  \tl_set:Nc}
```

\pref_global_chk: will turn the variable check in \tl_set:No into a global check.

```
\cs_new_protected_nopar:Npn \tl_set:Nc #1#2{\tl_set:No #1{\cs:w#2\cs_end:}}
```

106.2 Variables and constants

\c_job_name_tl

Inherited from the expl3 name for the primitive: this needs to actually contain the text of the jobname rather than the name of the primitive, of course.

```
\tl_new:N \c_job_name_tl
\tl_set:Nx \c_job_name_tl { \tex_jobname:D }
```

\c_empty_tl

Two constants which are often used.

```
\tl_const:Nn \c_empty_tl { }
```

\c_space_tl

A space as a token list (as opposed to as a character).

```
\tl_const:Nn \c_space_tl { ~ }
```

\g_tmpa_tl
\g_tmpb_tl

Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

```
\tl_new:N \g_tmpa_tl
\tl_new:N \g_tmpb_tl
```

\l_kernel_testa_tl
\l_kernel_testb_tl

Local temporaries. These are the ones for test routines. This means that one can safely use other temporaries when calling test routines.

```
\tl_new:N \l_kernel_testa_tl
\tl_new:N \l_kernel_testb_tl
```

\l_tmpa_tl
\l_tmpb_tl

These are local temporary token list variables. Be sure not to assume that the value you put into them will survive for long—see discussion above.

```
\tl_new:N \l_tmpa_tl
\tl_new:N \l_tmpb_tl
```

\l_kernel_tmpa_tl
\l_kernel_tmpb_tl

These are local temporary token list variables reserved for use by the kernel. They should not be used by other modules.

```
\tl_new:N \l_kernel_tmpa_tl
\tl_new:N \l_kernel_tmpb_tl
```

106.3 Predicates and conditionals

We also provide a few conditionals, both in expandable form (with `\c_true_bool`) and in 'brace-form', the latter are denoted by `TF` at the end, as explained elsewhere.

\tl_if_empty_p:N
\tl_if_empty_p:c
\tl_if_empty:N*TF*
\tl_if_empty:c*TF*

These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

```
\prg_set_conditional:Npnn \tl_if_empty:N #1 {p,TF,T,F} {
  \if_meaning:w #1 \c_empty_tl
    \prg_return_true: \else: \prg_return_false: \fi:
}
\cs_generate_variant:Nn \tl_if_empty_p:N {c}
\cs_generate_variant:Nn \tl_if_empty:NTF {c}
\cs_generate_variant:Nn \tl_if_empty:NT  {c}
\cs_generate_variant:Nn \tl_if_empty:NF  {c}
```

\tl_if_eq_p:NN
\tl_if_eq_p:Nc
\tl_if_eq_p:cN
\tl_if_eq_p:cc
\tl_if_eq:NN*TF*
\tl_if_eq:Nc*TF*
\tl_if_eq:cN*TF*
\tl_if_eq:cc*TF*

Returns `\c_true_bool` iff the two token list variables are equal.

```
\prg_new_conditional:Npnn \tl_if_eq:NN #1#2 {p,TF,T,F} {
  \if_meaning:w #1 #2 \prg_return_true: \else: \prg_return_false: \fi:
}
\cs_generate_variant:Nn \tl_if_eq_p:NN {Nc,c,cc}
\cs_generate_variant:Nn \tl_if_eq:NNTF {Nc,c,cc}
\cs_generate_variant:Nn \tl_if_eq:NNT  {Nc,c,cc}
\cs_generate_variant:Nn \tl_if_eq:NNF  {Nc,c,cc}
```

\tl_if_empty_p:n
\tl_if_empty_p:V
\tl_if_empty_p:o
\tl_if_empty:n*TF*
\tl_if_empty:V*TF*
\tl_if_empty:o*TF*

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.

```
\prg_new_conditional:Npnn \tl_if_empty:n #1 {p,TF,T,F} {
  \exp_after:wN \if_meaning:w \exp_after:wN \q_nil \tl_to_str:n {#1} \q_nil
    \prg_return_true: \else: \prg_return_false: \fi:
}
\cs_generate_variant:Nn \tl_if_empty_p:n {V}
\cs_generate_variant:Nn \tl_if_empty:nTF {V}
\cs_generate_variant:Nn \tl_if_empty:nT  {V}
\cs_generate_variant:Nn \tl_if_empty:nF  {V}
\cs_generate_variant:Nn \tl_if_empty_p:n {o}
\cs_generate_variant:Nn \tl_if_empty:nTF {o}
\cs_generate_variant:Nn \tl_if_empty:nT  {o}
\cs_generate_variant:Nn \tl_if_empty:nF  {o}
```

\tl_if_blank_p:n
\tl_if_blank_p:V
\tl_if_blank_p:o
\tl_if_blank:n*TF*
\tl_if_blank:V*TF*
\tl_if_blank:o*TF*
\tl_if_blank_p_aux:w

This is based on the answers in "Around the Bend No 2" but is safer as the tests listed there all have one small flaw: If the input in the test is two tokens with the same meaning as the internal delimiter, they will fail since one of them is mistaken for the actual delimiter. In our version below we make sure to pass the input through `\tl_to_str:n` which ensures that all the tokens are converted to catcode 12. However we use an `a` with catcode 11 as delimiter so we can *never* get into the same problem as the solutions in "Around the Bend No 2".

```
\prg_new_conditional:Npnn \tl_if_blank:n #1 {p,TF,T,F} {
  \exp_after:wN \tl_if_blank_p_aux:w \tl_to_str:n {#1} aa..\q_nil
}
\cs_new:Npn \tl_if_blank_p_aux:w #1#2 a #3#4 \q_nil {
  \if_meaning:w #3 #4 \prg_return_true: \else: \prg_return_false: \fi:
}
\cs_generate_variant:Nn \tl_if_blank_p:n {V}
\cs_generate_variant:Nn \tl_if_blank:nTF {V}
\cs_generate_variant:Nn \tl_if_blank:nT  {V}
\cs_generate_variant:Nn \tl_if_blank:nF  {V}
\cs_generate_variant:Nn \tl_if_blank_p:n {o}
\cs_generate_variant:Nn \tl_if_blank:nTF {o}
\cs_generate_variant:Nn \tl_if_blank:nT  {o}
\cs_generate_variant:Nn \tl_if_blank:nF  {o}
```

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

If the argument is a single token. 'Space' is considered 'true'.

```
\prg_new_conditional:Nnn \tl_if_single:n {p,TF,T,F} {
  \tl_if_empty:nTF {#1}
    {\prg_return_false:}
    {
      \tl_if_blank:nTF {#1}
        {\prg_return_true:}
        {
          \_tl_if_single_aux:w #1 \q_nil
        }
    }
}
```

Use `\exp_after:wN` below I know what I'm doing. Use `\exp_args:NV` or `\exp_args_unbraced:NV` for more flexibility in your own code.

```
\prg_new_conditional:Nnn \tl_if_single:N {p,TF,T,F} {
  \tl_if_empty:NTF #1
    {\prg_return_false:}
    {
      \exp_after:wN \tl_if_blank:nTF #1
        {\prg_return_true:}
        {
          \exp_after:wN \_tl_if_single_aux:w #1 \q_nil
        }
    }
}

\cs_new:Npn \_tl_if_single_aux:w #1#2 \q_nil {
  \tl_if_empty:nTF {#2} \prg_return_true: \prg_return_false:
}
```

\tl_if_eq:xx*TF*
\tl_if_eq:nn*TF*
\tl_if_eq:VV*TF*
\tl_if_eq:oo*TF*
\tl_if_eq:xn*TF*
\tl_if_eq:nx*TF*
\tl_if_eq:on*TF*
\tl_if_eq:no*TF*
\tl_if_eq:Vn*TF*
\tl_if_eq:nV*TF*
\tl_if_eq:xV*TF*
\tl_if_eq:xo*TF*
\tl_if_eq:Vx*TF*
\tl_if_eq:ox*TF*
\tl_if_eq_p:xx
\tl_if_eq_p:nn
\tl_if_eq_p:VV
\tl_if_eq_p:oo
\tl_if_eq_p:xn
\tl_if_eq_p:nx
\tl_if_eq_p:Vn
\tl_if_eq_p:on
\tl_if_eq_p:nV
\tl_if_eq_p:no
\tl_if_eq_p:xV
\tl_if_eq_p:Vx
\tl_if_eq_p:xo
\tl_if_eq_p:ox

Test if two token lists are identical. pdfTEX contains a most interesting primitive for expandable string comparison so we make use of it if available. Presumable it will be in the final version.

Firstly we give it an appropriate name. Note that this primitive actually performs an `x` type expansion but it is still expandable! Hence we must program these functions backwards to add `\exp_not:n`. We provide the combinations for the types `n`, `o` and `x`.

```
\cs_new_eq:NN \tl_compare:xx \pdf_strcmp:D
\cs_new:Npn \tl_compare:nn #1#2{
  \tl_compare:xx{\exp_not:n{#1}}{\exp_not:n{#2}}
}
\cs_new:Npn \tl_compare:nx #1{
  \tl_compare:xx{\exp_not:n{#1}}
}
\cs_new:Npn \tl_compare:xn #1#2{
  \tl_compare:xx{#1}{\exp_not:n{#2}}
}
\cs_new:Npn \tl_compare:nV #1#2 {
  \tl_compare:xx { \exp_not:n {#1} } { \exp_not:V #2 }
}
\cs_new:Npn \tl_compare:no #1#2{
  \tl_compare:xx{\exp_not:n{#1}}{\exp_not:n\exp_after:wN{#2}}
}
\cs_new:Npn \tl_compare:Vn #1#2 {
  \tl_compare:xx { \exp_not:V #1 } { \exp_not:n {#2} }
}
\cs_new:Npn \tl_compare:on #1#2{
  \tl_compare:xx{\exp_not:n\exp_after:wN{#1}}{\exp_not:n{#2}}
}
\cs_new:Npn \tl_compare:VV #1#2 {
  \tl_compare:xx { \exp_not:V #1 } { \exp_not:V #2 }
}
\cs_new:Npn \tl_compare:oo #1#2{
  \tl_compare:xx{\exp_not:n\exp_after:wN{#1}}{\exp_not:n\exp_after:wN{#2}}
}
\cs_new:Npn \tl_compare:xV #1#2 {
  \tl_compare:xx {#1} { \exp_not:V #2 }
}
\cs_new:Npn \tl_compare:xo #1#2{
  \tl_compare:xx{#1}{\exp_not:n\exp_after:wN{#2}}
}
\cs_new:Npn \tl_compare:Vx #1#2 {
  \tl_compare:xx { \exp_not:V #1 } {#2}
}
\cs_new:Npn \tl_compare:ox #1#2{
  \tl_compare:xx{\exp_not:n\exp_after:wN{#1}}{#2}
}
```

Since we have a lot of basically identical functions to define we define one to define the rest. Unfortunately we aren't quite set up to use the new `\tl_map_inline:nn` function yet.

```
\cs_set_nopar:Npn \tl_tmp:w #1 {
  \tl_set:Nx \l_kernel_tmpa_tl {
    \exp_not:N \prg_new_conditional:Npnn \exp_not:c {tl_if_eq:#1}
      ####1 ####2 {p,TF,T,F} {
        \exp_not:N \tex_ifnum:D
        \exp_not:c {tl_compare:#1} {####1}{####2}
        \exp_not:n{ =\c_zero \prg_return_true: \else: \prg_return_false: \fi: }
    }
  }
  \l_kernel_tmpa_tl
}
\tl_tmp:w{xx}  \tl_tmp:w{nx}  \tl_tmp:w{ox} \tl_tmp:w{Vx}
\tl_tmp:w{xn}  \tl_tmp:w{nn}  \tl_tmp:w{on} \tl_tmp:w{Vn}
\tl_tmp:w{xo}  \tl_tmp:w{no}  \tl_tmp:w{oo}
\tl_tmp:w{xV}  \tl_tmp:w{nV}  \tl_tmp:w{VV}
```

However all of this only makes sense if we actually have that primitive. Therefore we disable it again if it is not there and define `\tl_if_eq:nn` the old fashioned (and unexpandable) way.

In some cases below, since arbitrary token lists could be being used in this function, you can't assume (as token list variables usually do) that there won't be any # tokens. Therefore, `\tl_set:Nx` and `\exp_not:n` is used instead of plain `\tl_set:Nn`.

```
\cs_if_exist:cF{pdf_strcmp:D}{
  \prg_set_protected_conditional:Npnn \tl_if_eq:nn #1#2 {TF,T,F} {
    \tl_set:Nx \l_kernel_testa_tl {\exp_not:n{#1}}
    \tl_set:Nx \l_kernel_testb_tl {\exp_not:n{#2}}
    \if_meaning:w\l_kernel_testa_tl \l_kernel_testb_tl
      \prg_return_true: \else: \prg_return_false:
    \fi:
  }
  \prg_set_protected_conditional:Npnn \tl_if_eq:nV #1#2 {TF,T,F} {
    \tl_set:Nx \l_kernel_testa_tl { \exp_not:n {#1} }
    \tl_set:Nx \l_kernel_testb_tl { \exp_not:V #2 }
    \if_meaning:w \l_kernel_testa_tl \l_kernel_testb_tl
      \prg_return_true: \else: \prg_return_false:
    \fi:
  }
  \prg_set_protected_conditional:Npnn \tl_if_eq:no #1#2 {TF,T,F} {
    \tl_set:Nx \l_kernel_testa_tl {\exp_not:n{#1}}
    \tl_set:Nx \l_kernel_testb_tl {\exp_not:o{#2}}
    \if_meaning:w\l_kernel_testa_tl \l_kernel_testb_tl
      \prg_return_true: \else: \prg_return_false:
    \fi:
  }
  \prg_set_protected_conditional:Npnn \tl_if_eq:nx #1#2 {TF,T,F} {
```

```
  \tl_set:Nx \l_kernel_testa_tl {\exp_not:n{#1}}
  \tl_set:Nx \l_kernel_testb_tl {#2}
  \if_meaning:w\l_kernel_testa_tl \l_kernel_testb_tl
    \prg_return_true: \else: \prg_return_false:
  \fi:
}
\prg_set_protected_conditional:Npnn \tl_if_eq:Vn #1#2 {TF,T,F} {
  \tl_set:Nx \l_kernel_testa_tl { \exp_not:V #1 }
  \tl_set:Nx \l_kernel_testb_tl { \exp_not:n{#2} }
  \if_meaning:w \l_kernel_testa_tl \l_kernel_testb_tl
    \prg_return_true: \else: \prg_return_false:
  \fi:
}
\prg_set_protected_conditional:Npnn \tl_if_eq:on #1#2 {TF,T,F} {
  \tl_set:Nx \l_kernel_testa_tl {\exp_not:o{#1}}
  \tl_set:Nx \l_kernel_testb_tl {\exp_not:n{#2}}
  \if_meaning:w\l_kernel_testa_tl \l_kernel_testb_tl
    \prg_return_true: \else: \prg_return_false:
  \fi:
}
\prg_set_protected_conditional:Npnn \tl_if_eq:VV #1#2 {TF,T,F} {
  \tl_set:Nx \l_kernel_testa_tl { \exp_not:V #1 }
  \tl_set:Nx \l_kernel_testb_tl { \exp_not:V #2 }
  \if_meaning:w \l_kernel_testa_tl \l_kernel_testb_tl
    \prg_return_true: \else: \prg_return_false:
  \fi:
}
\prg_set_protected_conditional:Npnn \tl_if_eq:oo #1#2 {TF,T,F} {
  \tl_set:Nx \l_kernel_testa_tl {\exp_not:o{#1}}
  \tl_set:Nx \l_kernel_testb_tl {\exp_not:o{#2}}
  \if_meaning:w\l_kernel_testa_tl \l_kernel_testb_tl
    \prg_return_true: \else: \prg_return_false:
  \fi:
}
\prg_set_protected_conditional:Npnn \tl_if_eq:Vx #1#2 {TF,T,F} {
  \tl_set:Nx \l_kernel_testa_tl { \exp_not:V #1 }
  \tl_set:Nx \l_kernel_testb_tl {#2}
  \if_meaning:w \l_kernel_testa_tl \l_kernel_testb_tl
    \prg_return_true: \else: \prg_return_false:
  \fi:
}
\prg_set_protected_conditional:Npnn \tl_if_eq:ox #1#2 {TF,T,F} {
  \tl_set:Nx \l_kernel_testa_tl {\exp_not:o{#1}}
  \tl_set:Nx \l_kernel_testb_tl {#2}
  \if_meaning:w\l_kernel_testa_tl \l_kernel_testb_tl
    \prg_return_true: \else: \prg_return_false:
  \fi:
}
\prg_set_protected_conditional:Npnn \tl_if_eq:xn #1#2 {TF,T,F} {
  \tl_set:Nx \l_kernel_testa_tl {#1}
```

```
    \tl_set:Nx \l_kernel_testb_tl {\exp_not:n{#2}}
    \if_meaning:w\l_kernel_testa_tl \l_kernel_testb_tl
      \prg_return_true: \else: \prg_return_false:
    \fi:
  }
  \prg_set_protected_conditional:Npnn \tl_if_eq:xV #1#2 {TF,T,F} {
    \tl_set:Nx \l_kernel_testa_tl {#1}
    \tl_set:Nx \l_kernel_testb_tl { \exp_not:V #2 }
    \if_meaning:w \l_kernel_testa_tl \l_kernel_testb_tl
      \prg_return_true: \else: \prg_return_false:
    \fi:
  }
  \prg_set_protected_conditional:Npnn \tl_if_eq:xo #1#2 {TF,T,F} {
    \tl_set:Nx \l_kernel_testa_tl {#1}
    \tl_set:Nx \l_kernel_testb_tl {\exp_not:o{#2}}
    \if_meaning:w\l_kernel_testa_tl \l_kernel_testb_tl
      \prg_return_true: \else: \prg_return_false:
    \fi:
  }
  \prg_set_protected_conditional:Npnn \tl_if_eq:xx #1#2 {TF,T,F} {
    \tl_set:Nx \l_kernel_testa_tl {#1}
    \tl_set:Nx \l_kernel_testb_tl {#2}
    \if_meaning:w\l_kernel_testa_tl \l_kernel_testb_tl
      \prg_return_true: \else: \prg_return_false:
    \fi:
  }
}
```

106.4 Working with the contents of token lists

`\tl_to_lowercase:n`
`\tl_to_uppercase:n`

Just some names for a few primitives.

```
\cs_new_eq:NN \tl_to_lowercase:n \tex_lowercase:D
\cs_new_eq:NN \tl_to_uppercase:n \tex_uppercase:D
```

`\tl_to_str:n`

Another name for a primitive.

```
\cs_new_eq:NN \tl_to_str:n \etex_detokenize:D
```

`\tl_to_str:N`
`\tl_to_str:c`
`\tl_to_str_aux:w`

These functions return the replacement text of a token list as a string list with all characters catcoded to ‘other’.

```
\cs_new_nopar:Npn \tl_to_str:N {\exp_after:wN\tl_to_str_aux:w
  \token_to_meaning:N}
\cs_new_nopar:Npn \tl_to_str_aux:w #1>{}
\cs_generate_variant:Nn \tl_to_str:N {c}
```

\tl_map_function:nN
\tl_map_function:NN
\tl_map_function:cN
\tl_map_function_aux: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.

```
\cs_new:Npn \tl_map_function:nN #1#2{
  \tl_map_function_aux:Nn #2 #1 \q_recursion_tail \q_recursion_stop
}
\cs_new_nopar:Npn \tl_map_function:NN #1#2{
  \exp_after:wN \tl_map_function_aux:Nn
  \exp_after:wN #2 #1 \q_recursion_tail \q_recursion_stop
}
\cs_new:Npn \tl_map_function_aux:Nn #1#2{
  \quark_if_recursion_tail_stop:n{#2}
  #1{#2} \tl_map_function_aux:Nn  #1
}
\cs_generate_variant:Nn \tl_map_function:NN {cN}
```

\tl_map_inline:nn
\tl_map_inline:Nn
\tl_map_inline:cn
\tl_map_inline_aux:n
\g_tl_inline_level_int

The inline functions are straight forward by now. We use a little trick with the counter \g_tl_inline_level_int to make them nestable. We can also make use of \tl_map_function:Nn from before.

```
\cs_new_protected:Npn \tl_map_inline:nn #1#2{
  \int_gincr:N \g_tl_inline_level_int
  \cs_gset:cpn {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
  ##1{#2}
  \exp_args:Nc \tl_map_function_aux:Nn
  {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
  #1 \q_recursion_tail\q_recursion_stop
  \int_gdecr:N \g_tl_inline_level_int
}
\cs_new_protected:Npn \tl_map_inline:Nn #1#2{
  \int_gincr:N \g_tl_inline_level_int
  \cs_gset:cpn {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
  ##1{#2}
  \exp_last_unbraced:NcV \tl_map_function_aux:Nn
  {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
  #1 \q_recursion_tail\q_recursion_stop
  \int_gdecr:N \g_tl_inline_level_int
}
\cs_generate_variant:Nn \tl_map_inline:Nn {c}
```

\tl_map_variable:nNn
\tl_map_variable:NNn
\tl_map_variable:cNn

\tl_map_variable:nNn ⟨*token list*⟩ ⟨*temp*⟩ ⟨*action*⟩ assigns ⟨*temp*⟩ to each element and executes ⟨*action*⟩.

```
\cs_new_protected:Npn \tl_map_variable:nNn #1#2#3{
  \tl_map_variable_aux:Nnn #2 {#3} #1 \q_recursion_tail \q_recursion_stop
}
```

Next really has to be v/V args

```
\cs_new_protected_nopar:Npn \tl_map_variable:NNn {\exp_args:No \tl_map_variable:nNn}
\cs_generate_variant:Nn \tl_map_variable:NNn {c}
```

\tl_map_variable_aux:NnN

The general loop. Assign the temp variable #1 to the current item #3 and then check if that's the stop marker. If it is, break the loop. If not, execute the action #2 and continue.

```
\cs_new_protected:Npn \tl_map_variable_aux:Nnn #1#2#3{
  \tl_set:Nn #1{#3}
  \quark_if_recursion_tail_stop:N #1
  #2 \tl_map_variable_aux:Nnn #1{#2}
}
```

\tl_map_break:

The break statement.

```
\cs_new_eq:NN \tl_map_break: \use_none_delimit_by_q_recursion_stop:w
```

\tl_reverse:n
\tl_reverse:V
\tl_reverse:o
\tl_reverse_aux:nN

Reversal of a token list is done by taking one token at a time and putting it in front of the ones before it.

```
\cs_new:Npn \tl_reverse:n #1{
  \tl_reverse_aux:nN {} #1 \q_recursion_tail\q_recursion_stop
}
\cs_new:Npn \tl_reverse_aux:nN #1#2{
  \quark_if_recursion_tail_stop_do:nn {#2}{ #1 }
  \tl_reverse_aux:nN {#2#1}
}
\cs_generate_variant:Nn \tl_reverse:n {V,o}
```

\tl_reverse:N

This reverses the list, leaving \exp_stop_f: in front, which in turn is removed by the f expansion which comes to a halt.

```
\cs_new_protected_nopar:Npn \tl_reverse:N #1 {
  \tl_set:Nf #1 { \tl_reverse:o { #1 \exp_stop_f: } }
}
```

\tl_elt_count:n
\tl_elt_count:V
\tl_elt_count:o
\tl_elt_count:N

Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. \tl_elt_count_aux:n grabs the element and replaces it by +1. The 0 to ensure it works on an empty list.

```
\cs_new:Npn \tl_elt_count:n #1{
  \intexpr_eval:n {
    0 \tl_map_function:nN {#1} \tl_elt_count_aux:n
  }
}
\cs_generate_variant:Nn \tl_elt_count:n {V,o}
\cs_new_nopar:Npn \tl_elt_count:N #1{
  \intexpr_eval:n {
    0 \tl_map_function:NN #1 \tl_elt_count_aux:n
  }
}
```

\tl_num_elt_count_aux:n

Helper function for counting elements in a token list.

```
\cs_new:Npn \tl_elt_count_aux:n #1 { + 1 }
```

\tl_set_rescan:Nnn
\tl_gset_rescan:Nnn

These functions store the {⟨*token list*⟩} in ⟨*tl var.*⟩ after redefining catcodes, etc., in argument #2.

#1 : ⟨*tl var.*⟩
#2 : {⟨*catcode setup, etc.*⟩}
#3 : {⟨*token list*⟩}

```
\cs_new_protected:Npn \tl_set_rescan:Nnn  { \tl_set_rescan_aux:NNnn \tl_set:Nn  }
\cs_new_protected:Npn \tl_gset_rescan:Nnn { \tl_set_rescan_aux:NNnn \tl_gset:Nn }
```

\tl_set_rescan_aux:NNnn

This macro uses a trick to extract an unexpanded token list after it's rescanned with `\etex_scantokens:D`. This technique was first used (as far as I know) by Heiko Oberdiek in his catchfile package, albeit for real files rather than the 'fake' `\scantokens` one.

The basic problem arises because `\etex_scantokens:D` emulates a file read, which inserts an EOF marker into the expansion; the simplistic
`\exp_args:NNo \cs_set:Npn \tmp:w { \etex_scantokens:D {some text} }`
unfortunately doesn't work, calling the error:
`! File ended while scanning definition of \tmp:w.`
(LuaTEX works around this problem with its `\scantextokens` primitive.)

Usually, we'd define `\etex_everyeof:D` to be `\exp_not:N` to gobble the EOF marker, but since we're not expanding the token list, it gets left in there and we have the same basic problem.

Instead, we define `\etex_everyeof:D` to contain a marker that's impossible to occur within the scanned text; that is, the same char twice with different catcodes. (For some reason, we *don't* need to insert a `\exp_not:N` token after it to prevent the EOF marker to expand. Anyone know why?)

A helper function is can be used to save the token list delimited by the special marker, keeping the catcode redefinitions hidden away in a group.

\c_two_ats_with_two_catcodes_tl

A tl with two @ characters with two different catcodes. Used as a special marker for delimited text.

```
\group_begin:
  \tex_lccode:D `\A = `\@ \scan_stop:
  \tex_lccode:D `\B = `\@ \scan_stop:
  \tex_catcode:D `\A = 8 \scan_stop:
  \tex_catcode:D `\B = 3 \scan_stop:
\tl_to_lowercase:n {
  \group_end:
  \tl_const:Nn \c_two_ats_with_two_catcodes_tl { A B }
}
```

#1 : `\tl_set` function
#2 : ⟨*tl var.*⟩
#3 : {⟨*catcode setup, etc.*⟩}
#4 : {⟨*token list*⟩}

Note that if you change `\etex_everyeof:D` in #3 then you'd better do it correctly!

```
\cs_new_protected:Npn \tl_set_rescan_aux:NNnn #1#2#3#4 {
  \group_begin:
    \toks_set:NV \etex_everyeof:D \c_two_ats_with_two_catcodes_tl
    \tex_endlinechar:D = \c_minus_one
    #3
    \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#4}
    \exp_args:NNNV
  \group_end:
  #1 #2 \l_tmpa_tl
}
```

`\tl_rescan_aux:w`

```
\exp_after:wN \cs_set:Npn
\exp_after:wN \tl_rescan_aux:w
\exp_after:wN #
\exp_after:wN 1 \c_two_ats_with_two_catcodes_tl {
  \tl_set:Nn \l_tmpa_tl {#1}
}
```

`\tl_set_rescan:Nnx`
`\tl_gset_rescan:Nnx`

These functions store the full expansion of {⟨*token list*⟩} in ⟨*tl var.*⟩ after redefining catcodes, etc., in argument #2.

#1 : ⟨*tl var.*⟩
#2 : {⟨*catcode setup, etc.*⟩}
#3 : {⟨*token list*⟩}

The expanded versions are much simpler because the `\etex_scantokens:D` can occur within the expansion.

```
\cs_new_protected:Npn \tl_set_rescan:Nnx #1#2#3 {
  \group_begin:
    \etex_everyeof:D { \exp_not:N }
    \tex_endlinechar:D = \c_minus_one
    #2
    \tl_set:Nx \l_kernel_tmpa_tl { \etex_scantokens:D {#3} }
    \exp_args:NNNV
  \group_end:
  \tl_set:Nn #1 \l_kernel_tmpa_tl
}
```

Globally is easier again:

```
\cs_new_protected:Npn \tl_gset_rescan:Nnx #1#2#3 {
  \group_begin:
    \etex_everyeof:D { \exp_not:N }
    \tex_endlinechar:D = \c_minus_one
    #2
    \tl_gset:Nx #1 { \etex_scantokens:D {#3} }
  \group_end:
}
```

`\tl_rescan:nn` The inline wrapper for `\etex_scantokens:D`.

`#1` : Catcode changes (etc.)
`#2` : Token list to re-tokenise

```
\cs_new_protected:Npn \tl_rescan:nn #1#2 {
  \group_begin:
    \toks_set:NV \etex_everyeof:D \c_two_ats_with_two_catcodes_tl
    \tex_endlinechar:D = \c_minus_one
    #1
    \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#2}
  \exp_args:NV \group_end:
  \l_tmpa_tl
}
```

106.5 Checking for and replacing tokens

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

See the replace functions for further comments. In this part we don't care too much about brace stripping since we are not interested in passing on the tokens which are split off in the process.

```
\prg_new_protected_conditional:Npnn \tl_if_in:Nn #1#2 {TF,T,F} {
  \cs_set:Npn \tl_tmp:w ##1 #2 ##2 \q_stop {
    \quark_if_no_value:nTF {##2} {\prg_return_false:} {\prg_return_true:}
  }
  \exp_after:wN \tl_tmp:w #1 #2 \q_no_value \q_stop
}
\cs_generate_variant:Nn \tl_if_in:NnTF {c}
\cs_generate_variant:Nn \tl_if_in:NnT  {c}
\cs_generate_variant:Nn \tl_if_in:NnF  {c}
```

`\tl_if_in:nnTF`
`\tl_if_in:VnTF`
`\tl_if_in:onTF`

```
\prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 {TF,T,F} {
  \cs_set:Npn \tl_tmp:w ##1 #2 ##2 \q_stop {
    \quark_if_no_value:nTF {##2} {\prg_return_false:} {\prg_return_true:}
  }
  \tl_tmp:w #1 #2 \q_no_value \q_stop
}
```

```
\cs_generate_variant:Nn \tl_if_in:nnTF {V}
\cs_generate_variant:Nn \tl_if_in:nnT  {V}
\cs_generate_variant:Nn \tl_if_in:nnF  {V}
\cs_generate_variant:Nn \tl_if_in:nnTF {o}
\cs_generate_variant:Nn \tl_if_in:nnT  {o}
\cs_generate_variant:Nn \tl_if_in:nnF  {o}
```

_l_tl_replace_tl
\tl_replace_in:Nnn
\tl_replace_in:cnn
\tl_greplace_in:Nnn
\tl_greplace_in:cnn
_tl_replace_in_aux:NNnn

The concept here is that only the first occurrence should be replaced. The first step is to define an auxiliary which will match the appropriate item, with a trailing marker. If the last token is the marker there is nothing to do, otherwise replace the token and clean up (hence the second use of `\_tl_tmp:w`). To prevent loosing braces or spaces there are a couple of empty groups and the strange-looking `\use:n`.

```
\tl_new:N \_l_tl_replace_tl
\cs_new_protected_nopar:Npn \tl_replace_in:Nnn {
  \_tl_replace_in_aux:NNnn \tl_set_eq:NN
}
\cs_new_protected:Npn \_tl_replace_in_aux:NNnn #1#2#3#4 {
  \cs_set:Npn \_tl_tmp:w ##1 #3 ##2 \q_stop
    {
      \quark_if_no_value:nF {##2}
        {
          \tl_set:No \_l_tl_replace_tl { ##1 #4 }
          \cs_set:Npn \_tl_tmp:w ####1 #3 \q_no_value {
            \tl_put_right:No \_l_tl_replace_tl {####1}
          }
          \_tl_tmp:w \prg_do_nothing: ##2
          #1 #2 \_l_tl_replace_tl
        }
    }
  \use:n
    {
      \exp_after:wN \_tl_tmp:w \exp_after:wN
        \prg_do_nothing:
    }
  #2 #3 \q_no_value \q_stop
}
\cs_new_protected_nopar:Npn \tl_greplace_in:Nnn {
  \_tl_replace_in_aux:NNnn \tl_gset_eq:NN
}
\cs_generate_variant:Nn \tl_replace_in:Nnn  { c }
\cs_generate_variant:Nn \tl_greplace_in:Nnn { c }
```

\tl_replace_all_in:Nnn
\tl_replace_all_in:Nnn
\tl_greplace_all_in:cnn
\tl_greplace_all_in:cnn
_tl_replace_all_in_aux:NNnn

A similar approach here but with a loop built in.

```
\cs_new_protected_nopar:Npn \tl_replace_all_in:Nnn {
  \_tl_replace_all_in_aux:NNnn \tl_set_eq:NN
}
\cs_new_protected_nopar:Npn \tl_greplace_all_in:Nnn {
```

```
  \_tl_replace_all_in_aux:NNnn \tl_gset_eq:NN
}
\cs_new_protected:Npn \_tl_replace_all_in_aux:NNnn #1#2#3#4 {
  \tl_clear:N \_l_tl_replace_tl
  \cs_set:Npn \_tl_tmp:w ##1 #3 ##2 \q_stop
    {
      \quark_if_no_value:nTF {##2}
        { \tl_put_right:No \_l_tl_replace_tl {##1} }
        {
          \tl_put_right:No \_l_tl_replace_tl { ##1 #4 }
          \_tl_tmp:w \prg_do_nothing: ##2 \q_stop
        }
    }
  \use:n
    {
      \exp_after:wN \_tl_tmp:w \exp_after:wN
        \prg_do_nothing:
    }
  #2 #3 \q_no_value \q_stop
  #1 #2 \_l_tl_replace_tl
}
\cs_generate_variant:Nn \tl_replace_all_in:Nnn  { c }
\cs_generate_variant:Nn \tl_greplace_all_in:Nnn { c }
```

`\tl_remove_in:Nn`
`\tl_remove_in:cn`
`\tl_gremove_in:Nn`
`\tl_gremove_in:cn`

Next comes a series of removal functions. I have just implemented them as subcases of the replace functions for now (I'm lazy).

```
\cs_new_protected:Npn \tl_remove_in:Nn  #1#2{\tl_replace_in:Nnn #1{#2}{}}
\cs_new_protected:Npn \tl_gremove_in:Nn #1#2{\tl_greplace_in:Nnn #1{#2}{}}
\cs_generate_variant:Nn \tl_remove_in:Nn {cn}
\cs_generate_variant:Nn \tl_gremove_in:Nn {cn}
```

`\tl_remove_all_in:Nn`
`\tl_remove_all_in:cn`
`\tl_gremove_all_in:Nn`
`\tl_gremove_all_in:cn`

Same old, same old.

```
\cs_new_protected:Npn \tl_remove_all_in:Nn #1#2{
  \tl_replace_all_in:Nnn #1{#2}{}
}
\cs_new_protected:Npn \tl_gremove_all_in:Nn #1#2{
  \tl_greplace_all_in:Nnn #1{#2}{}
}
\cs_generate_variant:Nn \tl_remove_all_in:Nn {cn}
\cs_generate_variant:Nn \tl_gremove_all_in:Nn {cn}
```

106.6 Heads or tails?

`\tl_head:n`
`\tl_head:V`
`\tl_head_i:n`
`\tl_tail:n`
`\tl_tail:V`
`\tl_tail:f`
`\tl_head_iii:n`
`\tl_head_iii:f`
`\tl_head:w`
`\tl_head_i:w`
`\tl_tail:w`
`\tl_head_iii:w`

These functions pick up either the head or the tail of a list. `\tl_head_iii:n` returns the first three items on a list.

```
\cs_new:Npn \tl_head:n #1{\tl_head:w #1\q_nil}
\cs_new_eq:NN \tl_head_i:n \tl_head:n
\cs_new:Npn \tl_tail:n #1{\tl_tail:w #1\q_nil}
\cs_generate_variant:Nn \tl_tail:n {f}
\cs_new:Npn \tl_head_iii:n #1{\tl_head_iii:w #1\q_nil}
\cs_generate_variant:Nn \tl_head_iii:n {f}
\cs_new_eq:NN \tl_head:w \use_i_delimit_by_q_nil:nw
\cs_new_eq:NN \tl_head_i:w \tl_head:w
\cs_new:Npn \tl_tail:w #1#2\q_nil{#2}
\cs_new:Npn \tl_head_iii:w #1#2#3#4\q_nil{#1#2#3}
\cs_generate_variant:Nn \tl_head:n { V }
\cs_generate_variant:Nn \tl_tail:n { V }
```

\tl_if_head_eq_meaning_p:nN
\tl_if_head_eq_meaning:nNTF
\tl_if_head_eq_charcode_p:nN
\tl_if_head_eq_charcode_p:fN
\tl_if_head_eq_charcode:nNTF
\tl_if_head_eq_charcode:fNTF
\tl_if_head_eq_catcode_p:nN
\tl_if_head_eq_catcode:nNTF

When we want to check if the first token of a list equals something specific it is usually either to see if it is a control sequence or a character. Hence we make two different functions as the internal test is different. `\tl_if_head_meaning_eq:nNTF` uses `\if_meaning:w` and will consider the tokens $\mathtt{b}_{11}$ and $\mathtt{b}_{12}$ different. `\tl_if_head_char_eq:nNTF` on the other hand only compares character codes so would regard $\mathtt{b}_{11}$ and $\mathtt{b}_{12}$ as equal but would also regard two primitives as equal.

```
\prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 {p,TF,T,F} {
  \exp_after:wN \if_meaning:w \tl_head:w #1 \q_nil #2
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

For the charcode and catcode versions we insert `\exp_not:N` in front of both tokens. If you need them to expand fully as TeX does itself with these you can use an `f` type expansion.

```
\prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 {p,TF,T,F} {
  \exp_after:wN \if:w \exp_after:wN \exp_not:N
      \tl_head:w #1 \q_nil \exp_not:N #2
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

Actually the default is already an `f` type expansion.

```
%% \cs_new:Npn \tl_if_head_eq_charcode_p:fN #1#2{
%%    \exp_after:wN\if_charcode:w \tl_head:w #1\q_nil\exp_not:N#2
%%      \c_true_bool
%%    \else:
%%      \c_false_bool
%%    \fi:
%% }
%% \def_long_test_function_new:npn {tl_if_head_eq_charcode:fN}#1#2{
%%   \if_predicate:w \tl_if_head_eq_charcode_p:fN {#1}#2}
```

These `:fN` variants are broken; temporary patch:

```
\cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN {f}
```

```
\cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF {f}
\cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT  {f}
\cs_generate_variant:Nn \tl_if_head_eq_charcode:nNF  {f}
```

And now catcodes:

```
\prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1#2 {p,TF,T,F} {
  \exp_after:wN \if_catcode:w \exp_after:wN \exp_not:N
      \tl_head:w #1 \q_nil \exp_not:N #2
    \prg_return_true: \else: \prg_return_false: \fi:
}
```

Show token usage:

```
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

107 l3toks implementation

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

107.1 Allocation and use

\toks_new:N
\toks_new:c
\toks_new_local:N
\toks_new_local:c

Allocates a new token register.

```
⟨*initex⟩
\alloc_new:nnnN {toks} \c_zero \c_max_register_int \tex_toksdef:D
⟨/initex⟩
```

```
⟨*package⟩
\cs_new_protected_nopar:Npn \toks_new:N #1 {
  \chk_if_free_cs:N #1
  \newtoks #1
}
\cs_new_protected_nopar:Npn \toks_new_local:N #1 {
  \chk_if_free_cs:N #1
  \int_compare:nNnTF
    \tex_currentgrouplevel:D = 0
    \newtoks \loctoks
```

```
  #1
}
⟨/package⟩
```

```
\cs_generate_variant:Nn \toks_new:N {c}
\cs_generate_variant:Nn \toks_new_local:N {c}
```

\toks_use:N
\toks_use:c

This function returns the contents of a token register.

```
\cs_new_eq:NN  \toks_use:N \tex_the:D
\cs_generate_variant:Nn \toks_use:N {c}
```

\toks_set:Nn
\toks_set:NV
\toks_set:Nv
\toks_set:No
\toks_set:Nx
\toks_set:Nf
\toks_set:cn
\toks_set:co
\toks_set:cV
\toks_set:cv
\toks_set:cx
\toks_set:cf

`\toks_set:Nn`⟨*toks*⟩⟨*stuff*⟩ stores ⟨*stuff*⟩ without expansion in ⟨*toks*⟩. `\toks_set:No` and `\toks_set:Nx` expand ⟨*stuff*⟩ once and fully.

```
⟨*check⟩
\cs_new_protected_nopar:Npn \toks_set:Nn #1 { \chk_local:N #1 #1 }
\cs_generate_variant:Nn \toks_set:Nn {No,Nf}
⟨/check⟩
```

If we don't check if ⟨*toks*⟩ is a local register then the `\toks_set:Nn` function has nothing to do. We implement `\toks_set:No/d/f` by hand when not checking because this is going to be used *extensively* in keyval processing! TODO: (Will) Can we get some numbers published on how necessary this is? On the other hand I'm happy to believe Morten :)

```
⟨*!check⟩
\cs_new_eq:NN       \toks_set:Nn \prg_do_nothing:
\cs_new_protected:Npn \toks_set:NV #1#2 {
  #1 \exp_after:wN { \int_to_roman:w -`0 \exp_eval_register:N #2 }
}
\cs_new_protected:Npn \toks_set:Nv #1#2 {
  #1 \exp_after:wN { \int_to_roman:w -`0 \exp_eval_register:c {#2} }
}
\cs_new_protected:Npn \toks_set:No #1#2 { #1 \exp_after:wN {#2} }
\cs_new_protected:Npn \toks_set:Nf #1#2 {
  #1 \exp_after:wN { \int_to_roman:w -`0#2 }
}
⟨/!check⟩
```

```
\cs_generate_variant:Nn \toks_set:Nn {Nx,cn,cV,cv,co,cx,cf}
```

\toks_gset:Nn
\toks_gset:NV
\toks_gset:No
\toks_gset:Nx
\toks_gset:cn
\toks_gset:cV
\toks_gset:co
\toks_gset:cx

These functions are the global variants of the above.

```
⟨check⟩\cs_new_protected_nopar:Npn \toks_gset:Nn #1 { \chk_global:N #1 \pref_global:D #1 }
⟨!check⟩\cs_new_eq:NN \toks_gset:Nn \pref_global:D
\cs_generate_variant:Nn \toks_gset:Nn {NV,No,Nx,cn,cV,co,cx}
```

\toks_set_eq:NN
\toks_set_eq:Nc
\toks_set_eq:cN
\toks_set_eq:cc
\toks_gset_eq:NN
\toks_gset_eq:Nc
\toks_gset_eq:cN
\toks_gset_eq:cc

`\toks_set_eq:NN`⟨*toks1*⟩⟨*toks2*⟩ copies the contents of ⟨*toks2*⟩ in ⟨*toks1*⟩.

```
⟨*check⟩
\cs_new_protected_nopar:Npn\toks_set_eq:NN #1#2 {
  \chk_local:N #1
  \chk_var_or_const:N #2
  #1 #2
}
\cs_new_protected_nopar:Npn\toks_gset_eq:NN #1#2 {
  \chk_global:N #1
  \chk_var_or_const:N #2
  \pref_global:D #1 #2
}
⟨/check⟩
⟨*!check⟩
\cs_new_eq:NN \toks_set_eq:NN  \prg_do_nothing:
\cs_new_eq:NN \toks_gset_eq:NN \pref_global:D
⟨/!check⟩
\cs_generate_variant:Nn \toks_set_eq:NN  {Nc,cN,cc}
\cs_generate_variant:Nn \toks_gset_eq:NN  {Nc,cN,cc}
```

\toks_clear:N
\toks_gclear:N
\toks_clear:c
\toks_gclear:c

These functions clear a token register, either locally or globally.

```
\cs_new_protected_nopar:Npn \toks_clear:N #1 {
  #1\c_empty_toks
⟨check⟩\chk_local_or_pref_global:N #1
}

\cs_new_protected_nopar:Npn \toks_gclear:N {
⟨check⟩  \pref_global_chk:
⟨!check⟩  \pref_global:D
  \toks_clear:N
}

\cs_generate_variant:Nn \toks_clear:N  {c}
\cs_generate_variant:Nn \toks_gclear:N {c}
```

\toks_use_clear:N
\toks_use_clear:c
\toks_use_gclear:N
\toks_use_gclear:c

These functions clear a token register (locally or globally) after returning the contents.

They make sure that clearing the register does not interfere with following tokens. In other words, the contents of the register might operate on what follows in the input stream.

```
\cs_new_protected_nopar:Npn \toks_use_clear:N #1 {
  \exp_last_unbraced:NNV \toks_clear:N #1 #1
}

\cs_new_protected_nopar:Npn \toks_use_gclear:N {
⟨check⟩  \pref_global_chk:
⟨!check⟩  \pref_global:D
  \toks_use_clear:N
}
```

```
\cs_generate_variant:Nn \toks_use_clear:N  {c}
\cs_generate_variant:Nn \toks_use_gclear:N {c}
```

\toks_show:N
\toks_show:c

This function shows the contents of a token register on the terminal. TODO: this is not pretty when the argument is a control sequence that doesn't exist!

```
\cs_new_eq:NN           \toks_show:N \tex_showthe:D
\cs_generate_variant:Nn \toks_show:N {c}
```

107.2 Adding to token registers' contents

\toks_put_left:Nn
\toks_put_left:NV
\toks_put_left:No
\toks_put_left:Nx
\toks_put_left:cn
\toks_put_left:cV
\toks_put_left:co
\toks_gput_left:Nn
\toks_gput_left:NV
\toks_gput_left:No
\toks_gput_left:Nx
\toks_gput_left:cn
\toks_gput_left:cV
\toks_gput_left:co
\toks_put_left_aux:w

`\toks_put_left:Nn` ⟨*toks*⟩⟨*stuff*⟩ adds the tokens of *stuff* on the 'left-side' of the token register ⟨*toks*⟩. `\toks_put_left:No` does the same, but expands the tokens once. We need to look out for brace stripping so we add a token, which is then later removed.

```
\cs_new_protected_nopar:Npn \toks_put_left:Nn #1 {
  \exp_after:wN \toks_put_left_aux:w \exp_after:wN \q_mark
    \toks_use:N #1 \q_stop #1
}
```

```
\cs_generate_variant:Nn \toks_put_left:Nn  {NV,No,Nx,cn,co,cV}
```

```
\cs_new_protected_nopar:Npn \toks_gput_left:Nn {
⟨check⟩  \pref_global_chk:
⟨!check⟩  \pref_global:D
  \toks_put_left:Nn
}
```

```
\cs_generate_variant:Nn \toks_gput_left:Nn  {NV,No,Nx,cn,cV,co}
```

A helper function for `\toks_put_left:Nn`. Its arguments are subsequently the tokens of ⟨*stuff*⟩, the token register ⟨*toks*⟩ and the current contents of ⟨*toks*⟩. We make sure to remove the token we inserted earlier.

```
\cs_new:Npn \toks_put_left_aux:w #1\q_stop #2#3 {
  #2 \exp_after:wN { \use_i:nn {#3} #1 }
⟨check⟩  \chk_local_or_pref_global:N #2
}
```

\toks_put_right:Nn
\toks_put_right:NV
\toks_put_right:No
\toks_put_right:Nx
\toks_put_right:cn
\toks_put_right:cV
\toks_put_right:co
\toks_gput_right:Nn
\toks_gput_right:NV
\toks_gput_right:No
\toks_gput_right:Nx
\toks_gput_right:cn
\toks_gput_right:cV
\toks_gput_right:co

These macros add a list of tokens to the right of a token register.

```
\cs_new_protected:Npn \toks_put_right:Nn #1#2 {
  #1 \exp_after:wN { \toks_use:N #1 #2 }
⟨check⟩  \chk_local_or_pref_global:N #1
}
```

```
\cs_new_protected_nopar:Npn \toks_gput_right:Nn {
⟨check⟩  \pref_global_chk:
⟨!check⟩  \pref_global:D
  \toks_put_right:Nn
}
```

A couple done by hand for speed.

```
⟨check⟩\cs_generate_variant:Nn \toks_put_right:Nn {No}
⟨*!check⟩
\cs_new_protected:Npn \toks_put_right:NV #1#2 {
  #1 \exp_after:wN \exp_after:wN \exp_after:wN {
      \exp_after:wN \toks_use:N \exp_after:wN #1
      \int_to_roman:w -`0 \exp_eval_register:N #2
  }
}
\cs_new_protected:Npn \toks_put_right:No #1#2 {
  #1 \exp_after:wN \exp_after:wN \exp_after:wN {
     \exp_after:wN \toks_use:N \exp_after:wN #1 #2
  }
}
⟨/!check⟩
\cs_generate_variant:Nn \toks_put_right:Nn {Nx,cn,cV,co}
\cs_generate_variant:Nn \toks_gput_right:Nn {NV,No,Nx,cn,cV,co}
```

`\toks_put_right:Nf`

We implement `\toks_put_right:Nf` by hand because I think I might use it in the l3keyval module in which case it is going to be used a lot.

```
⟨check⟩\cs_generate_variant:Nn \toks_put_right:Nn {Nf}
⟨*!check⟩
\cs_new_protected:Npn \toks_put_right:Nf #1#2 {
  #1 \exp_after:wN \exp_after:wN \exp_after:wN {
      \exp_after:wN \toks_use:N \exp_after:wN #1 \int_to_roman:w -`0#2
  }
}
⟨/!check⟩
```

107.3 Predicates and conditionals

`\toks_if_empty_p:N`
`\toks_if_empty_p:c`
`\toks_if_empty:NTF`
`\toks_if_empty:cTF`

`\toks_if_empty:NTF`⟨*toks*⟩⟨*true code*⟩⟨*false code*⟩ tests if a token register is empty and executes either ⟨*true code*⟩ or ⟨*false code*⟩. This test had the advantage of being expandable. Otherwise one has to do an `x` type expansion in order to prevent problems with parameter tokens.

```
\prg_new_conditional:Nnn \toks_if_empty:N {p,TF,T,F} {
  \tl_if_empty:VTF #1 {\prg_return_true:} {\prg_return_false:}
}
```

```
\cs_generate_variant:Nn \toks_if_empty_p:N {c}
\cs_generate_variant:Nn \toks_if_empty:NTF {c}
\cs_generate_variant:Nn \toks_if_empty:NT  {c}
\cs_generate_variant:Nn \toks_if_empty:NF  {c}
```

\toks_if_eq_p:NN
\toks_if_eq_p:cN
\toks_if_eq_p:Nc
\toks_if_eq_p:cc
\toks_if_eq:NN*TF*
\toks_if_eq:Nc*TF*
\toks_if_eq:cN*TF*
\toks_if_eq:cc*TF*

This function test whether two token registers have the same contents.

```
\prg_new_conditional:Nnn \toks_if_eq:NN {p,TF,T,F} {
  \tl_if_eq:xxTF {\toks_use:N #1} {\toks_use:N #2}
      {\prg_return_true:} {\prg_return_false:}
}
\cs_generate_variant:Nn \toks_if_eq_p:NN {Nc,c,cc}
\cs_generate_variant:Nn \toks_if_eq:NNTF {Nc,c,cc}
\cs_generate_variant:Nn \toks_if_eq:NNT  {Nc,c,cc}
\cs_generate_variant:Nn \toks_if_eq:NNF  {Nc,c,cc}
```

107.4 Variables and constants

\l_tmpa_toks
\l_tmpb_toks
\l_tmpc_toks
\g_tmpa_toks
\g_tmpb_toks
\g_tmpc_toks

Some scratch registers …

```
\tex_toksdef:D \l_tmpa_toks = 255\scan_stop:
⟨initex⟩\seq_put_right:Nn \g_toks_allocation_seq {255}
\toks_new:N \l_tmpb_toks
\toks_new:N \l_tmpc_toks
\toks_new:N \g_tmpa_toks
\toks_new:N \g_tmpb_toks
\toks_new:N \g_tmpc_toks
```

\c_empty_toks

And here is a constant, which is a (permanently) empty token register.

```
\toks_new:N \c_empty_toks
```

\l_tl_replace_toks

And here is one for tl vars. Can't define it there as the allocation isn't set up at that point.

```
\toks_new:N \l_tl_replace_toks
```

```
⟨/initex | package⟩
```

Show token usage:

```
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

108 l3seq implementation

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
```

A sequence is a control sequence whose top-level expansion is of the form '\seq_elt:w ⟨$text_1$⟩ \seq_elt_end: ... \seq_elt:w ⟨$text_n$⟩ ...'. We use explicit delimiters instead of braces around ⟨*text*⟩ to allow efficient searching for an item in the sequence.

\seq_elt:w
\seq_elt_end:

We allocate the delimiters and make them errors if executed.

```
⟨*initex | package⟩
\cs_new:Npn \seq_elt:w {\ERROR}
\cs_new:Npn \seq_elt_end: {\ERROR}
```

108.1 Allocating and initialisation

\seq_new:N
\seq_new:c

Sequences are implemented using token lists.

```
\cs_new_eq:NN \seq_new:N \tl_new:N
\cs_new_eq:NN \seq_new:c \tl_new:c
```

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

Clearing a sequence is the same as clearing a token list.

```
\cs_new_eq:NN \seq_clear:N  \tl_clear:N
\cs_new_eq:NN \seq_clear:c  \tl_clear:c
\cs_new_eq:NN \seq_gclear:N \tl_gclear:N
\cs_new_eq:NN \seq_gclear:c \tl_gclear:c
```

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

Clearing a sequence is the same as clearing a token list.

```
\cs_new_eq:NN \seq_clear_new:N  \tl_clear_new:N
\cs_new_eq:NN \seq_clear_new:c  \tl_clear_new:c
\cs_new_eq:NN \seq_gclear_new:N \tl_gclear_new:N
\cs_new_eq:NN \seq_gclear_new:c \tl_gclear_new:c
```

\seq_set_eq:NN
\seq_set_eq:Nc
\seq_set_eq:cN
\seq_set_eq:cc

We can set one seq equal to another.

```
\cs_new_eq:NN \seq_set_eq:NN \cs_set_eq:NN
\cs_new_eq:NN \seq_set_eq:cN \cs_set_eq:cN
\cs_new_eq:NN \seq_set_eq:Nc \cs_set_eq:Nc
\cs_new_eq:NN \seq_set_eq:cc \cs_set_eq:cc
```

`\seq_gset_eq:NN`
`\seq_gset_eq:cN`
`\seq_gset_eq:Nc`
`\seq_gset_eq:cc`

And of course globally which seems to be needed far more often.[12]

```
\cs_new_eq:NN \seq_gset_eq:NN \cs_gset_eq:NN
\cs_new_eq:NN \seq_gset_eq:cN \cs_gset_eq:cN
\cs_new_eq:NN \seq_gset_eq:Nc \cs_gset_eq:Nc
\cs_new_eq:NN \seq_gset_eq:cc \cs_gset_eq:cc
```

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

`\seq_gconcat:NNN` ⟨*seq 1*⟩ ⟨*seq 2*⟩ ⟨*seq 3*⟩ will globally assign ⟨*seq 1*⟩ the concatenation of ⟨*seq 2*⟩ and ⟨*seq 3*⟩.

```
\cs_new_protected_nopar:Npn \seq_gconcat:NNN #1#2#3 {
  \tl_gset:Nx #1 { \exp_not:V #2 \exp_not:V #3 }
}
\cs_generate_variant:Nn \seq_gconcat:NNN {ccc}
```

108.2 Predicates and conditionals

`\seq_if_empty_p:N`
`\seq_if_empty_p:c`
`\seq_if_empty:NTF`
`\seq_if_empty:cTF`

A predicate which evaluates to `\c_true_bool` iff the sequence is empty.

```
\prg_new_eq_conditional:NNn \seq_if_empty:N \tl_if_empty:N {p,TF,T,F}
\prg_new_eq_conditional:NNn \seq_if_empty:c \tl_if_empty:c {p,TF,T,F}
```

`\seq_if_empty_err:N`

Signals an error if the sequence is empty.

```
\cs_new_nopar:Npn \seq_if_empty_err:N #1 {
  \if_meaning:w #1 \c_empty_tl
```

As I said before, I don't think we need to provide checks for this kind of error, since it is a severe internal macro package error that can not be produced by the user directly. Can it? So the next line of code should be probably removed. (Will: I have no idea what this comment means.)

```
    \tl_clear:N \l_kernel_testa_tl % catch prefixes
    \msg_kernel_bug:x {Empty~sequence~`\token_to_str:N#1'}
  \fi:
}
```

`\seq_if_in:NnTF`
`\seq_if_in:NVTF`
`\seq_if_in:cnTF`
`\seq_if_in:cVTF`
`\seq_if_in:coTF`
`\seq_if_in:cxTF`

`\seq_if_in:NnTF` ⟨*seq*⟩⟨*item*⟩ ⟨*true case*⟩ ⟨*false case*⟩ will check whether ⟨*item*⟩ is in ⟨*seq*⟩ and then either execute the ⟨*true case*⟩ or the ⟨*false case*⟩. ⟨*true case*⟩ and ⟨*false case*⟩ may contain incomplete `\if_charcode:w` statements.

Note that `##2` in the definition below for `\seq_tmp:w` contains exactly one token which we can compare with `\q_no_value`.

```
\prg_new_protected_conditional:Nnn \seq_if_in:Nn {TF,T,F} {
  \cs_set:Npn \seq_tmp:w ##1 \seq_elt:w #2 \seq_elt_end: ##2##3 \q_stop {
    \if_meaning:w \q_no_value ##2
```

[12]To save a bit of space these functions could be made identical to those from the tl or clist module.

```
      \prg_return_false: \else: \prg_return_true: \fi:
  }
  \exp_after:wN \seq_tmp:w #1 \seq_elt:w #2 \seq_elt_end: \q_no_value \q_stop
}
```

```
\cs_generate_variant:Nn \seq_if_in:NnTF { NV, cV, co, c, cx}
\cs_generate_variant:Nn \seq_if_in:NnT  { NV, cV, co, c, cx}
\cs_generate_variant:Nn \seq_if_in:NnF  { NV, cV, co, c, cx}
```

108.3 Getting data out

`\seq_get:NN`
`\seq_get:cN`
`\seq_get_aux:w`

`\seq_get:NN` ⟨*sequence*⟩⟨*cmd*⟩ defines ⟨*cmd*⟩ to be the left-most element of ⟨*sequence*⟩.

```
\cs_new_protected_nopar:Npn \seq_get:NN #1 {
  \seq_if_empty_err:N #1
  \exp_after:wN \seq_get_aux:w #1 \q_stop
}
\cs_new_protected:Npn \seq_get_aux:w \seq_elt:w #1 \seq_elt_end: #2 \q_stop #3 {
  \tl_set:Nn #3 {#1}
}
\cs_generate_variant:Nn \seq_get:NN {c}
```

`\seq_pop_aux:nnNN`
`\seq_pop_aux:w`

`\seq_pop_aux:nnNN` ⟨*def*$_1$⟩ ⟨*def*$_2$⟩ ⟨*sequence*⟩ ⟨*cmd*⟩ assigns the left_most element of ⟨*sequence*⟩ to ⟨*cmd*⟩ using ⟨*def*$_2$⟩, and assigns the tail of ⟨*sequence*⟩ to ⟨*sequence*⟩ using ⟨*def*$_1$⟩.

```
\cs_new_protected:Npn \seq_pop_aux:nnNN #1#2#3 {
  \seq_if_empty_err:N #3
  \exp_after:wN \seq_pop_aux:w #3 \q_stop #1#2#3
}
\cs_new_protected:Npn \seq_pop_aux:w
    \seq_elt:w #1 \seq_elt_end: #2\q_stop #3#4#5#6 {
  #3 #5 {#2}
  #4 #6 {#1}
}
```

`\seq_show:N`
`\seq_show:c`

```
\cs_new_eq:NN \seq_show:N \tl_show:N
\cs_new_eq:NN \seq_show:c \tl_show:c
```

`\seq_display:N`
`\seq_display:c`

```
\cs_new_protected_nopar:Npn \seq_display:N #1 {
  \iow_term:x { Sequence~\token_to_str:N #1~contains~
                  the~elements~(without~outer~braces): }
  \toks_clear:N \l_tmpa_toks
  \seq_map_inline:Nn #1 {
```

```
      \toks_if_empty:NF  \l_tmpa_toks {
        \toks_put_right:Nx \l_tmpa_toks {^^J>~}
      }
      \toks_put_right:Nx \l_tmpa_toks {
        \c_space_tl \iow_char:N \{ \exp_not:n {##1} \iow_char:N \}
      }
    }
    \toks_show:N \l_tmpa_toks
}
\cs_generate_variant:Nn \seq_display:N {c}
```

108.4 Putting data in

\seq_put_aux:Nnn
\seq_put_aux:w

\seq_put_aux:Nnn ⟨*sequence*⟩ ⟨*left*⟩ ⟨*right*⟩ adds the elements specified by ⟨*left*⟩ to the left of ⟨*sequence*⟩, and those specified by ⟨*right*⟩ to the right.

```
\cs_new_protected:Npn \seq_put_aux:Nnn #1 {
  \exp_after:wN \seq_put_aux:w #1 \q_stop #1
}
\cs_new_protected:Npn \seq_put_aux:w #1\q_stop #2#3#4 { \tl_set:Nn #2 {#3#1#4} }
```

\seq_put_left:Nn
\seq_put_left:NV
\seq_put_left:No
\seq_put_left:Nx
\seq_put_left:cn
\seq_put_left:cV
\seq_put_left:co
\seq_put_right:Nn
\seq_put_right:No
\seq_put_right:NV
\seq_put_right:Nx
\seq_put_right:cn
\seq_put_right:cV
\seq_put_right:co

Here are the usual operations for adding to the left and right.

```
\cs_new_protected:Npn \seq_put_left:Nn #1#2 {
  \seq_put_aux:Nnn #1 {\seq_elt:w #2\seq_elt_end:} {}
}
```

We can't put in a \prg_do_nothing: instead of {} above since this argument is passed literally (and we would end up with many \prg_do_nothing:s inside the sequences).

```
\cs_generate_variant:Nn \seq_put_left:Nn {NV,No,Nx,c,cV,co}

\cs_new_protected:Npn \seq_put_right:Nn #1#2{
        \seq_put_aux:Nnn #1{}{\seq_elt:w #2\seq_elt_end:}}

\cs_generate_variant:Nn \seq_put_right:Nn {NV,No,Nx,c,cV,co}
```

\seq_gput_left:Nn
\seq_gput_left:NV
\seq_gput_left:No
\seq_gput_left:Nx
\seq_gput_left:cn
\seq_gput_left:cV
\seq_gput_left:co
\seq_gput_right:Nn
\seq_gput_right:NV
\seq_gput_right:No
\seq_gput_right:Nx
\seq_gput_right:cn
\seq_gput_right:cV
\seq_gput_right:co

```
\cs_new_protected:Npn \seq_gput_left:Nn {
⟨*check⟩
  \pref_global_chk:
⟨/check⟩
⟨*!check⟩
  \pref_global:D
⟨/!check⟩
  \seq_put_left:Nn
}
```

```
\cs_new_protected:Npn \seq_gput_right:Nn {
⟨*check⟩
  \pref_global_chk:
⟨/check⟩
⟨*!check⟩
  \pref_global:D
⟨/!check⟩
  \seq_put_right:Nn
}
```

```
\cs_generate_variant:Nn \seq_gput_left:Nn {NV,No,Nx,c,cV,co}
```

```
\cs_generate_variant:Nn \seq_gput_right:Nn {NV,No,Nx,c,cV,co}
```

`\seq_gput_right:Nc` TODO: move to `xor` (Sep 2008)

```
\cs_generate_variant:Nn \seq_gput_right:Nn {Nc}
```

108.5 Mapping

`\seq_map_variable:NNn`
`\seq_map_variable:cNn`
`\seq_map_variable_aux:Nnw`

Nothing spectacular here.

```
\cs_new_protected:Npn \seq_map_variable_aux:Nnw #1#2 \seq_elt:w #3 \seq_elt_end: {
  \tl_set:Nn #1 {#3}
  \quark_if_nil:NT #1 \seq_map_break:
  #2
  \seq_map_variable_aux:Nnw #1{#2}
}
\cs_new_protected:Npn \seq_map_variable:NNn #1#2#3 {
  \tl_set:Nn #2 {\seq_map_variable_aux:Nnw #2{#3}}
  \exp_after:wN #2 #1 \seq_elt:w \q_nil\seq_elt_end: \q_stop
}
\cs_generate_variant:Nn \seq_map_variable:NNn {c}
```

`\seq_map_break:`
`\seq_map_break:n`

Terminate a mapping function at the point of execution. The latter takes an argument to be executed after cleaning up the map.

```
\cs_new_eq:NN \seq_map_break: \use_none_delimit_by_q_stop:w
\cs_new_eq:NN \seq_map_break:n \use_i_delimit_by_q_stop:nw
```

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

`\seq_map_function:NN` ⟨*sequence*⟩ ⟨*cmd*⟩ applies ⟨*cmd*⟩ to each element of ⟨*sequence*⟩, from left to right. Since we don't have braces, this implementation is not very efficient. It might be better to say that ⟨*cmd*⟩ must be a function with one argument that is delimited by `\seq_elt_end:`.

```
\cs_new_protected_nopar:Npn \seq_map_function:NN #1#2 {
  \cs_set:Npn \seq_elt:w ##1 \seq_elt_end: {#2{##1}}
  #1 \use_none:n \q_stop
```

```
  \cs_set_eq:NN \seq_elt:w \ERROR
}
\cs_generate_variant:Nn \seq_map_function:NN {c}
```

\seq_map_inline:Nn
\seq_map_inline:cn

When no braces are used, this version of mapping seems more natural.

```
\cs_new_protected_nopar:Npn \seq_map_inline:Nn #1#2 {
  \cs_set:Npn \seq_elt:w ##1 \seq_elt_end: {#2}
  #1 \use_none:n \q_stop
  \cs_set_eq:NN \seq_elt:w \ERROR
}
\cs_generate_variant:Nn \seq_map_inline:Nn {c}
```

108.6 Manipulation

\l_clist_remove_clist

A common scratch space for the removal routines.

```
\seq_new:N \l_seq_remove_seq
```

\seq_remove_duplicates_aux:NN
\seq_remove_duplicates_aux:n
\seq_remove_duplicates:N
\seq_gremove_duplicates:N

Copied from `\clist_remove_duplicates`.

```
\cs_new_protected:Npn \seq_remove_duplicates_aux:NN #1#2 {
  \seq_clear:N \l_seq_remove_seq
  \seq_map_function:NN #2 \seq_remove_duplicates_aux:n
  #1 #2 \l_seq_remove_seq
}
\cs_new_protected:Npn \seq_remove_duplicates_aux:n #1 {
  \seq_if_in:NnF \l_seq_remove_seq {#1} {
    \seq_put_right:Nn \l_seq_remove_seq {#1}
  }
}

\cs_new_protected_nopar:Npn \seq_remove_duplicates:N {
  \seq_remove_duplicates_aux:NN \seq_set_eq:NN
}
\cs_new_protected_nopar:Npn \seq_gremove_duplicates:N {
  \seq_remove_duplicates_aux:NN \seq_gset_eq:NN
}
```

108.7 Sequence stacks

\seq_push:Nn
\seq_push:NV
\seq_push:No
\seq_push:cn
\seq_pop:NN
\seq_pop:cN

Since sequences can be used as stacks, we ought to have both 'push' and 'pop'. In most cases they are nothing more then new names for old functions.

```
\cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
\cs_new_eq:NN \seq_push:NV \seq_put_left:NV
\cs_new_eq:NN \seq_push:No \seq_put_left:No
```

```
\cs_new_eq:NN \seq_push:cn \seq_put_left:cn
\cs_new_protected_nopar:Npn \seq_pop:NN { \seq_pop_aux:nnNN \tl_set:Nn \tl_set:Nn }
\cs_generate_variant:Nn \seq_pop:NN {c}
```

\seq_gpush:Nn
\seq_gpush:NV
\seq_gpush:No
\seq_gpush:cn
\seq_gpush:Nv
\seq_gpop:NN
\seq_gpop:cN

I don't agree with Denys that one needs only local stacks, actually I believe that one will probably need the functions here more often. In case of `\seq_gpop:NN` the value is nevertheless returned locally.

```
\cs_new_eq:NN  \seq_gpush:Nn \seq_gput_left:Nn
\cs_new_protected_nopar:Npn \seq_gpop:NN  { \seq_pop_aux:nnNN \tl_gset:Nn \tl_set:Nn }
\cs_generate_variant:Nn \seq_gpush:Nn {NV,No,c,Nv}
\cs_generate_variant:Nn  \seq_gpop:NN  {c}
```

\seq_top:NN
\seq_top:cN

Looking at the top element of the stack without removing it is done with this operation.

```
\cs_new_eq:NN \seq_top:NN \seq_get:NN
\cs_new_eq:NN \seq_top:cN \seq_get:cN
```

```
⟨/initex | package⟩
```

Show token usage:

```
⟨*showmemory⟩
%\showMemUsage
⟨/showmemory⟩
```

109 l3clist implementation

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

109.1 Allocation and initialisation

\clist_new:N
\clist_new:c

Comma-Lists are implemented using token lists.

```
\cs_new_eq:NN \clist_new:N \tl_new:N
\cs_generate_variant:Nn \clist_new:N {c}
```

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

Clearing a comma-list is the same as clearing a token list.

```
\cs_new_eq:NN \clist_clear:N  \tl_clear:N
\cs_generate_variant:Nn \clist_clear:N {c}
\cs_new_eq:NN \clist_gclear:N \tl_gclear:N
\cs_generate_variant:Nn \clist_gclear:N {c}
```

\clist_clear_new:N
\clist_clear_new:c
\clist_gclear_new:N
\clist_gclear_new:c

Clearing a comma-list is the same as clearing a token list.

```
\cs_new_eq:NN \clist_clear_new:N  \tl_clear_new:N
\cs_generate_variant:Nn \clist_clear_new:N {c}
\cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
\cs_generate_variant:Nn \clist_gclear_new:N {c}
```

\clist_set_eq:NN
\clist_set_eq:cN
\clist_set_eq:Nc
\clist_set_eq:cc

We can set one ⟨*clist*⟩ equal to another.

```
\cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
\cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
\cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
\cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
```

\clist_gset_eq:NN
\clist_gset_eq:cN
\clist_gset_eq:Nc
\clist_gset_eq:cc

An of course globally which seems to be needed far more often.

```
\cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
\cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

109.2 Predicates and conditionals

\clist_if_empty_p:N
\clist_if_empty_p:c
\clist_if_empty:N*TF*
\clist_if_empty:c*TF*

```
\prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N {p,TF,T,F}
\prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c {p,TF,T,F}
```

\clist_if_empty_err:N

Signals an error if the comma-list is empty.

```
\cs_new_protected_nopar:Npn \clist_if_empty_err:N #1 {
  \if_meaning:w #1 \c_empty_tl
    \tl_clear:N \l_kernel_testa_tl % catch prefixes
    \msg_kernel_bug:x {Empty~comma-list~`\token_to_str:N #1'}
  \fi:
}
```

\clist_if_eq_p:NN
\clist_if_eq_p:Nc
\clist_if_eq_p:cN
\clist_if_eq_p:cc
\clist_if_eq:NN*TF*
\clist_if_eq:cN*TF*
\clist_if_eq:Nc*TF*
\clist_if_eq:cc*TF*

Returns `\c_true` iff the two comma-lists are equal.

```
\prg_new_eq_conditional:NNn \clist_if_eq:NN \tl_if_eq:NN {p,TF,T,F}
\prg_new_eq_conditional:NNn \clist_if_eq:cN \tl_if_eq:cN {p,TF,T,F}
\prg_new_eq_conditional:NNn \clist_if_eq:Nc \tl_if_eq:Nc {p,TF,T,F}
\prg_new_eq_conditional:NNn \clist_if_eq:cc \tl_if_eq:cc {p,TF,T,F}
```

\clist_if_in:NnTF
\clist_if_in:NVTF
\clist_if_in:NoTF
\clist_if_in:cnTF
\clist_if_in:cVTF
\clist_if_in:coTF

`\clist_if_in:NnTF` ⟨*clist*⟩⟨*item*⟩ ⟨*true case*⟩ ⟨*false case*⟩ will check whether ⟨*item*⟩ is in ⟨*clist*⟩ and then either execute the ⟨*true case*⟩ or the ⟨*false case*⟩. ⟨*true case*⟩ and ⟨*false case*⟩ may contain incomplete `\if_charcode:w` statements.

```
\prg_new_protected_conditional:Nnn \clist_if_in:Nn {TF,T,F} {
  \cs_set:Npn \clist_tmp:w ##1,#2,##2##3 \q_stop {
    \if_meaning:w \q_no_value ##2
      \prg_return_false: \else: \prg_return_true: \fi:
  }
  \exp_last_unbraced:NNo \clist_tmp:w , #1 , #2 , \q_no_value \q_stop
}
```

```
\cs_generate_variant:Nn \clist_if_in:NnTF {NV,No,cn,cV,co}
\cs_generate_variant:Nn \clist_if_in:NnT {NV,No,cn,cV,co}
\cs_generate_variant:Nn \clist_if_in:NnF {NV,No,cn,cV,co}
```

109.3 Retrieving data

\clist_use:N
\clist_use:c

Using a ⟨*clist*⟩ is just executing it but if ⟨*clist*⟩ equals `\scan_stop:` it is probably stemming from a `\cs:w ... \cs_end:` that was created by mistake somewhere.

```
\cs_new_nopar:Npn \clist_use:N #1 {
  \if_meaning:w #1 \scan_stop:
    \msg_kernel_bug:x {
      Comma~list~ `\token_to_str:N #1'~ has~ an~ erroneous~ structure!}
  \else:
    \exp_after:wN #1
  \fi:
}
\cs_generate_variant:Nn \clist_use:N {c}
```

\clist_get:NN
\clist_get:cN
\clist_get_aux:w

`\clist_get:NN` ⟨*comma-list*⟩⟨*cmd*⟩ defines ⟨*cmd*⟩ to be the left-most element of ⟨*comma-list*⟩.

```
\cs_new_protected_nopar:Npn \clist_get:NN #1 {
  \clist_if_empty_err:N #1
  \exp_after:wN \clist_get_aux:w #1,\q_stop
}
```

```
\cs_new_protected:Npn \clist_get_aux:w  #1,#2\q_stop #3 { \tl_set:Nn #3{#1} }
```

```
\cs_generate_variant:Nn \clist_get:NN {cN}
```

\clist_pop_aux:nnNN
\clist_pop_aux:w
\clist_pop_auxi:w

`\clist_pop_aux:nnNN` ⟨def_1⟩ ⟨def_2⟩ ⟨*comma-list*⟩ ⟨*cmd*⟩ assigns the left-most element of ⟨*comma-list*⟩ to ⟨*cmd*⟩ using ⟨def_2⟩, and assigns the tail of ⟨*comma-list*⟩ to ⟨*comma-list*⟩ using ⟨def_1⟩.

```
\cs_new_protected:Npn \clist_pop_aux:nnNN #1#2#3 {
  \clist_if_empty_err:N #3
  \exp_after:wN \clist_pop_aux:w #3,\q_nil\q_stop #1#2#3
}
```

After the assignmnets below, if there was only one element in the original clist, it now contains only `\q_nil`.

```
\cs_new_protected:Npn \clist_pop_aux:w  #1,#2\q_stop #3#4#5#6 {
   #4 #6 {#1}
   #3 #5 {#2}
   \quark_if_nil:NTF #5 { #3 #5 {} }{ \clist_pop_auxi:w #2 #3#5 }
}
```

```
\cs_new:Npn \clist_pop_auxi:w #1,\q_nil #2#3 { #2#3{#1} }
```

`\clist_show:N`
`\clist_show:c`

```
\cs_new_eq:NN \clist_show:N \tl_show:N
\cs_new_eq:NN \clist_show:c \tl_show:c
```

`\clist_display:N`
`\clist_display:c`

```
\cs_new_protected_nopar:Npn \clist_display:N #1 {
  \iow_term:x { Comma-list~\token_to_str:N #1~contains~
                the~elements~(without~outer~braces): }
  \toks_clear:N \l_tmpa_toks
  \clist_map_inline:Nn #1 {
    \toks_if_empty:NF  \l_tmpa_toks {
      \toks_put_right:Nx \l_tmpa_toks {^^J>~}
    }
    \toks_put_right:Nx \l_tmpa_toks {
      \c_space_tl \iow_char:N \{ \exp_not:n {##1} \iow_char:N \}
    }
  }
  \toks_show:N \l_tmpa_toks
}
\cs_generate_variant:Nn \clist_display:N {c}
```

109.4 Storing data

`\clist_put_aux:NNnnNn` The generic put function. When adding we have to distinguish between an empty ⟨*clist*⟩ and one that contains at least one item (otherwise we accumulate commas).

MH says: Perhaps we should make sure that empty arguments don't get on the stack as that is probably a mistake. That's what I've implemented here. Since `\tl_if_empty:nF` is expandable prefixes are still allowed.

```
\cs_new_protected:Npn \clist_put_aux:NNnnNn #1#2#3#4#5#6 {
  \clist_if_empty:NTF #5 { #1 #5 {#6} } {
    \tl_if_empty:nF {#6} { #2 #5{#3#6#4} }
  }
}
```

\clist_put_left:Nn
\clist_put_left:NV
\clist_put_left:No
\clist_put_left:Nx
\clist_put_left:cn
\clist_put_left:cV
\clist_put_left:co

The operations for adding to the left.

```
\cs_new_protected_nopar:Npn \clist_put_left:Nn {
  \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_left:Nn {} ,
}
\cs_generate_variant:Nn \clist_put_left:Nn {NV,No,Nx,cn,cV,co}
```

\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

Global versions.

```
\cs_new_protected_nopar:Npn \clist_gput_left:Nn {
  \clist_put_aux:NNnnNn \tl_gset:Nn \tl_gput_left:Nn {} ,
}
\cs_generate_variant:Nn \clist_gput_left:Nn {NV,No,Nx,cn,cV,co}
```

\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

Adding something to the right side is almost the same.

```
\cs_new_protected_nopar:Npn \clist_put_right:Nn {
  \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_right:Nn , {}
}
\cs_generate_variant:Nn \clist_put_right:Nn {NV,No,Nx,cn,cV,co}
```

\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

And here the global variants.

```
\cs_new_protected_nopar:Npn \clist_gput_right:Nn {
  \clist_put_aux:NNnnNn \tl_gset:Nn \tl_gput_right:Nn , {}
}
\cs_generate_variant:Nn \clist_gput_right:Nn {NV,No,Nx,cn,cV,co}
```

109.5 Mapping

\clist_map_function:NN
\clist_map_function:cN
\clist_map_function:nN

\clist_map_function:NN ⟨*comma-list*⟩ ⟨*cmd*⟩ applies ⟨*cmd*⟩ to each element of ⟨*comma-list*⟩, from left to right.

```
\cs_new_nopar:Npn \clist_map_function:NN #1#2 {
  \clist_if_empty:NF #1 {
    \exp_after:wN \clist_map_function_aux:Nw
    \exp_after:wN #2 #1 , \q_recursion_tail , \q_recursion_stop
  }
}
\cs_generate_variant:Nn \clist_map_function:NN {cN}

\cs_new:Npn \clist_map_function:nN #1#2 {
  \tl_if_blank:nF {#1} {
    \clist_map_function_aux:Nw #2 #1 , \q_recursion_tail , \q_recursion_stop
  }
}
```

\clist_map_function_aux:Nw The general loop. Tests if we hit the first stop marker and exits if we did. If we didn't, place the function #1 in front of the element #2, which is surrounded by braces.

```
\cs_new:Npn \clist_map_function_aux:Nw #1#2,{
  \quark_if_recursion_tail_stop:n{#2}
  #1{#2}
  \clist_map_function_aux:Nw #1
}
```

\clist_map_break: The break statement is easy. Same as in other modules, gobble everything up to the special recursion stop marker.

```
\cs_new_eq:NN \clist_map_break: \use_none_delimit_by_q_recursion_stop:w
```

\clist_map_inline:Nn
\clist_map_inline:cn
\clist_map_inline:nn

The inline type is faster but not expandable. In order to make it nestable, we use a counter to keep track of the nesting level so that all of the functions called have distict names. A simpler approach would of course be to use grouping and thus the save stack but then you lose the ability to do things locally.

A funny little thing occured in one document: The command setting up the first call of \clist_map_inline:Nn was used in a tabular cell and the inline code used \\ so the loop broke as soon as this happened. Lesson to be learned from this: If you wish to have group like structure but not using the groupings of TeX, then do every operation globally.

```
\int_new:N \g_clist_inline_level_int
\cs_new_protected:Npn \clist_map_inline:Nn #1#2 {
  \clist_if_empty:NF #1 {
    \int_gincr:N \g_clist_inline_level_int
    \cs_gset:cpn {clist_map_inline_ \int_use:N \g_clist_inline_level_int :n}
    ##1{#2}
```

It is a lot more efficient to carry over the special function rather than constructing the same csname over and over again, so we just do it once. We reuse \clist_map_function_aux:Nw for the actual loop.

```
    \exp_last_unbraced:NcV \clist_map_function_aux:Nw
    {clist_map_inline_ \int_use:N \g_clist_inline_level_int :n}
    #1 , \q_recursion_tail , \q_recursion_stop
    \int_gdecr:N \g_clist_inline_level_int
  }
}
\cs_generate_variant:Nn \clist_map_inline:Nn {c}
\cs_new_protected:Npn \clist_map_inline:nn #1#2 {
  \tl_if_empty:nF {#1} {
    \int_gincr:N \g_clist_inline_level_int
    \cs_gset:cpn {clist_map_inline_ \int_use:N \g_clist_inline_level_int :n}
    ##1{#2}
    \exp_args:Nc \clist_map_function_aux:Nw
```

```
    {clist_map_inline_ \int_use:N \g_clist_inline_level_int :n}
    #1 , \q_recursion_tail , \q_recursion_stop
    \int_gdecr:N \g_clist_inline_level_int
  }
}
```

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

\clist_map_variable:NNn ⟨*comma-list*⟩ ⟨*temp*⟩ ⟨*action*⟩ assigns ⟨*temp*⟩ to each element and executes ⟨*action*⟩.

```
\cs_new_protected:Npn \clist_map_variable:nNn #1#2#3 {
  \tl_if_empty:nF {#1} {
    \clist_map_variable_aux:Nnw #2 {#3} #1
    , \q_recursion_tail , \q_recursion_stop
  }
}
```

Something for v/V

```
\cs_new_protected_nopar:Npn \clist_map_variable:NNn {\exp_args:No \clist_map_variable:nNn}
\cs_generate_variant:Nn\clist_map_variable:NNn {cNn}
```

\clist_map_variable_aux:Nnw

The general loop. Assign the temp variable #1 to the current item #3 and then check if that's the stop marker. If it is, break the loop. If not, execute the action #2 and continue.

```
\cs_new_protected:Npn \clist_map_variable_aux:Nnw #1#2#3,{
  \cs_set_nopar:Npn #1{#3}
  \quark_if_recursion_tail_stop:N #1
  #2 \clist_map_variable_aux:Nnw #1{#2}
}
```

109.6 Higher level functions

\clist_concat_aux:NNNN
\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc

\clist_gconcat:NNN ⟨*clist 1*⟩ ⟨*clist 2*⟩ ⟨*clist 3*⟩ will globally assign ⟨*clist 1*⟩ the concatenation of ⟨*clist 2*⟩ and ⟨*clist 3*⟩.

Again the situation is a bit more complicated because of the use of commas between items, so if either list is empty we have to avoid adding a comma.

```
\cs_new_protected_nopar:Npn \clist_concat_aux:NNNN #1#2#3#4 {
  \tl_set:No \l_tmpa_tl {#3}
  \tl_set:No \l_tmpb_tl {#4}
  #1 #2 {
    \exp_not:V \l_tmpa_tl
    \tl_if_empty:NF \l_tmpa_tl { \tl_if_empty:NF \l_tmpb_tl , }
    \exp_not:V \l_tmpb_tl
  }
}
\cs_new_protected_nopar:Npn \clist_concat:NNN  { \clist_concat_aux:NNNN \tl_set:Nx  }
\cs_new_protected_nopar:Npn \clist_gconcat:NNN { \clist_concat_aux:NNNN \tl_gset:Nx }
\cs_generate_variant:Nn \clist_concat:NNN {ccc}
\cs_generate_variant:Nn \clist_gconcat:NNN {ccc}
```

\l_clist_remove_clist

A common scratch space for the removal routines.

```
\clist_new:N \l_clist_remove_clist
```

\clist_remove_duplicates_aux:NN
\clist_remove_duplicates_aux:n
\clist_remove_duplicates:N
\clist_gremove_duplicates:N

Removing duplicate entries in a ⟨*clist*⟩ is fairly straight forward. We use a temporary variable and then go through the list from left to right. For each element check if the element is already present in the list.

```
\cs_new_protected:Npn \clist_remove_duplicates_aux:NN #1#2 {
  \clist_clear:N \l_clist_remove_clist
  \clist_map_function:NN #2 \clist_remove_duplicates_aux:n
  #1 #2 \l_clist_remove_clist
}
\cs_new_protected:Npn \clist_remove_duplicates_aux:n #1 {
  \clist_if_in:NnF \l_clist_remove_clist {#1} {
    \clist_put_right:Nn \l_clist_remove_clist {#1}
  }
}
```

The high level functions are just for telling if it should be a local or global setting.

```
\cs_new_protected_nopar:Npn \clist_remove_duplicates:N {
  \clist_remove_duplicates_aux:NN \clist_set_eq:NN
}
\cs_new_protected_nopar:Npn \clist_gremove_duplicates:N {
  \clist_remove_duplicates_aux:NN \clist_gset_eq:NN
}
```

\clist_remove_element:Nn
\clist_gremove_element:Nn
\clist_remove_element_aux:NNn
\clist_remove_element_aux:n

The same general idea is used for removing elements: the parent functions just set things up for the internal ones.

```
\cs_new_protected_nopar:Npn \clist_remove_element:Nn {
  \clist_remove_element_aux:NNn \clist_set_eq:NN
}
\cs_new_protected_nopar:Npn \clist_gremove_element:Nn {
  \clist_remove_element_aux:NNn \clist_gset_eq:NN
}
\cs_new_protected:Npn \clist_remove_element_aux:NNn #1#2#3 {
  \clist_clear:N \l_clist_remove_clist
  \cs_set:Npn \clist_remove_element_aux:n ##1 {
    \tl_if_eq:nnF {#3} {##1} {
      \clist_put_right:Nn \l_clist_remove_clist {##1}
    }
  }
  \clist_map_function:NN #2 \clist_remove_element_aux:n
  #1 #2 \l_clist_remove_clist
}
\cs_new:Npn \clist_remove_element_aux:n #1 { }
```

109.7 Stack operations

We build stacks from comma-lists, but here we put the specific functions together.

\clist_push:Nn
\clist_push:No
\clist_push:NV
\clist_push:cn
\clist_pop:NN
\clist_pop:cN

Since comma-lists can be used as stacks, we ought to have both 'push' and 'pop'. In most cases they are nothing more then new names for old functions.

```
\cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
\cs_new_eq:NN \clist_push:NV \clist_put_left:NV
\cs_new_eq:NN \clist_push:No \clist_put_left:No
\cs_new_eq:NN \clist_push:cn \clist_put_left:cn
\cs_new_protected_nopar:Npn \clist_pop:NN {\clist_pop_aux:nnNN \tl_set:Nn \tl_set:Nn}
\cs_generate_variant:Nn \clist_pop:NN {cN}
```

\clist_gpush:Nn
\clist_gpush:No
\clist_gpush:NV
\clist_gpush:cn
\clist_gpop:NN
\clist_gpop:cN

I don't agree with Denys that one needs only local stacks, actually I believe that one will probably need the functions here more often. In case of `\clist_gpop:NN` the value is nevertheless returned locally.

```
\cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
\cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
\cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
\cs_generate_variant:Nn \clist_gpush:Nn {cn}

\cs_new_protected_nopar:Npn \clist_gpop:NN {\clist_pop_aux:nnNN \tl_gset:Nn \tl_set:Nn}
\cs_generate_variant:Nn \clist_gpop:NN {cN}
```

\clist_top:NN
\clist_top:cN

Looking at the top element of the stack without removing it is done with this operation.

```
\cs_new_eq:NN \clist_top:NN \clist_get:NN
\cs_new_eq:NN \clist_top:cN \clist_get:cN

⟨/initex | package⟩
```

Show token usage:

```
⟨*showmemory⟩
%\showMemUsage
⟨/showmemory⟩
```

110 l3prop implementation

A property list is a token register whose contents is of the form

$$\backslash q\_prop \langle key_1 \rangle \backslash q\_prop \{\langle info_1 \rangle\} \ldots \backslash q\_prop \langle key_n \rangle \backslash q\_prop \{\langle info_n \rangle\}$$

The property ⟨*key*⟩s and ⟨*info*⟩s might be arbitrary token lists; each ⟨*info*⟩ is surrounded by braces.

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

\q_prop The separator between ⟨*key*⟩s and ⟨*info*⟩s and ⟨*key*⟩s.

```
\quark_new:N\q_prop
```

To get values from property-lists, token lists should be passed to the appropriate functions.

110.1 Functions

\prop_new:N
\prop_new:c

Property lists are implemented as token registers.

```
\cs_new_eq:NN \prop_new:N \toks_new:N
\cs_new_eq:NN \prop_new:c \toks_new:c
```

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

The same goes for clearing a property list, either locally or globally.

```
\cs_new_eq:NN \prop_clear:N  \toks_clear:N
\cs_new_eq:NN \prop_clear:c  \toks_clear:c
\cs_new_eq:NN \prop_gclear:N \toks_gclear:N
\cs_new_eq:NN \prop_gclear:c \toks_gclear:c
```

\prop_set_eq:NN
\prop_set_eq:Nc
\prop_set_eq:cN
\prop_set_eq:cc
\prop_gset_eq:NN
\prop_gset_eq:Nc
\prop_gset_eq:cN
\prop_gset_eq:cc

This makes two ⟨*prop*⟩s have the same contents.

```
\cs_new_eq:NN \prop_set_eq:NN  \toks_set_eq:NN
\cs_new_eq:NN \prop_set_eq:Nc  \toks_set_eq:Nc
\cs_new_eq:NN \prop_set_eq:cN  \toks_set_eq:cN
\cs_new_eq:NN \prop_set_eq:cc  \toks_set_eq:cc
\cs_new_eq:NN \prop_gset_eq:NN \toks_gset_eq:NN
\cs_new_eq:NN \prop_gset_eq:Nc \toks_gset_eq:Nc
\cs_new_eq:NN \prop_gset_eq:cN \toks_gset_eq:cN
\cs_new_eq:NN \prop_gset_eq:cc \toks_gset_eq:cc
```

\prop_show:N
\prop_show:c

Show on the console the raw contents of a property list's token register.

```
\cs_new_eq:NN \prop_show:N \toks_show:N
\cs_new_eq:NN \prop_show:c \toks_show:c
```

`\prop_display:N`
`\prop_display:c`

Pretty print the contents of a property list on the console.

```
\cs_new_protected_nopar:Npn \prop_display:N #1 {
  \iow_term:x { Property-list~\token_to_str:N #1~contains~
                the~pairs~(without~outer~braces): }
  \toks_clear:N \l_tmpa_toks
  \prop_map_inline:Nn #1 {
    \toks_if_empty:NF  \l_tmpa_toks {
      \toks_put_right:Nx \l_tmpa_toks {^^J>~}
    }
    \toks_put_right:Nx \l_tmpa_toks {
      \c_space_tl \iow_char:N \{ \exp_not:n {##1} \iow_char:N \} \c_space_tl
      \c_space_tl => \c_space_tl
      \c_space_tl \iow_char:N \{ \exp_not:n {##2} \iow_char:N \}
    }
  }
  \toks_show:N \l_tmpa_toks
}
\cs_generate_variant:Nn \prop_display:N {c}
```

`\prop_split_aux:Nnn`

`\prop_split_aux:Nnn`⟨*prop*⟩⟨*key*⟩⟨*cmd*⟩ invokes ⟨*cmd*⟩ with 3 arguments: 1st is the beginning of ⟨*prop*⟩ before ⟨*key*⟩, 2nd is the value associated with ⟨*key*⟩, 3rd is the rest of ⟨*prop*⟩ after ⟨*key*⟩. If there is no property ⟨*key*⟩ in ⟨*prop*⟩, then the 2nd argument will be `\q_no_value` and the 3rd argument is empty; otherwise the 3rd argument has the extra tokens `\q_prop` ⟨*key*⟩ `\q_prop \q_no_value` at the end.

```
\cs_new_protected:Npn \prop_split_aux:Nnn #1#2#3{
  \cs_set:Npn \prop_tmp:w ##1 \q_prop #2 \q_prop ##2##3 \q_stop {
    #3 {##1}{##2}{##3}
  }
  \exp_after:wN \prop_tmp:w \toks_use:N #1 \q_prop #2 \q_prop \q_no_value \q_stop
}
```

`\prop_get:NnN`
`\prop_get:NVN`
`\prop_get:cnN`
`\prop_get:cVN`
`\prop_get_aux:w`

`\prop_get:NnN` ⟨*prop*⟩⟨*key*⟩⟨*tl var.*⟩ defines ⟨*tl var.*⟩ to be the value associated with ⟨*key*⟩ in ⟨*prop*⟩, `\q_no_value` if not found.

```
\cs_new_protected:Npn \prop_get:NnN #1#2 {
  \prop_split_aux:Nnn #1{#2}\prop_get_aux:w
}
\cs_new_protected:Npn \prop_get_aux:w #1#2#3#4 { \tl_set:Nn #4 {#2} }

\cs_generate_variant:Nn \prop_get:NnN { NVN, cnN, cVN }
```

`\prop_gget:NnN`
`\prop_gget:NVN`
`\prop_gget:NnN`
`\prop_gget:NVN`
`\prop_gget_aux:w`

The global version of the previous function.

```
\cs_new_protected:Npn \prop_gget:NnN #1#2{
  \prop_split_aux:Nnn #1{#2}\prop_gget_aux:w}
\cs_new_protected:Npn \prop_gget_aux:w #1#2#3#4{\tl_gset:Nx#4{\exp_not:n{#2}}}

\cs_generate_variant:Nn \prop_gget:NnN { NVN, cnN, cVN }
```

`\prop_get_gdel:NnN` `\prop_get_del_aux:w` `\prop_get_gdel:NnN` is the same as `\prop_get:NnN` but the ⟨*key*⟩ and its value are afterwards globally removed from ⟨*property_list*⟩. One probably also needs the local variants or only the local one, or... We decide this later.

```
\cs_new_protected:Npn \prop_get_gdel:NnN #1#2#3{
  \prop_split_aux:Nnn #1{#2}{\prop_get_del_aux:w #3{\toks_gset:Nn #1}{#2}}}
\cs_new_protected:Npn \prop_get_del_aux:w #1#2#3#4#5#6{
  \tl_set:Nn #1 {#5}
  \quark_if_no_value:NF #1 {
    \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#3\q_prop\q_no_value {#2{#4##1}}
    \prop_tmp:w #6}
}
```

`\prop_put:Nnn` `\prop_put:NnV` `\prop_put:NVn` `\prop_put:NVV` `\prop_put:cnn` `\prop_gput:Nnn` `\prop_gput:NVn` `\prop_gput:NnV` `\prop_gput:Nno` `\prop_gput:Nnx` `\prop_gput:cnn` `\prop_gput:ccx` `\prop_put_aux:w`

`\prop_put:Nnn` {⟨*prop*⟩} {⟨*key*⟩} {⟨*info*⟩} adds/changes the value associated with ⟨*key*⟩ in ⟨*prop*⟩ to ⟨*info*⟩.

```
\cs_new_protected:Npn \prop_put:Nnn #1#2{
  \prop_split_aux:Nnn #1{#2}{
    \prop_clear:N #1
    \prop_put_aux:w {\toks_put_right:Nn #1}{#2}
  }
}

\cs_new_protected:Npn \prop_gput:Nnn #1#2{
  \prop_split_aux:Nnn #1{#2}{
    \prop_gclear:N #1
    \prop_put_aux:w {\toks_gput_right:Nn #1}{#2}
  }
}

\cs_new_protected:Npn \prop_put_aux:w #1#2#3#4#5#6{
  #1{\q_prop#2\q_prop{#6}#3}
  \tl_if_empty:nF{#5}
  {
    \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#2\q_prop\q_no_value {#1{##1}}
    \prop_tmp:w #5
  }
}

\cs_generate_variant:Nn \prop_put:Nnn { NnV, NVn, NVV, cnn }

\cs_generate_variant:Nn \prop_gput:Nnn {NVn,NnV,Nno,Nnx,Nox,cnn,ccx}
```

`\prop_del:Nn` `\prop_del:NV` `\prop_gdel:NV` `\prop_gdel:Nn` `\prop_del_aux:w`

`\prop_del:Nn` ⟨*prop*⟩⟨*key*⟩ deletes the entry for ⟨*key*⟩ in ⟨*prop*⟩, if any.

```
\cs_new_protected:Npn \prop_del:Nn #1#2{
  \prop_split_aux:Nnn #1{#2}{\prop_del_aux:w {\toks_set:Nn #1}{#2}}}
\cs_new_protected:Npn \prop_gdel:Nn #1#2{
  \prop_split_aux:Nnn #1{#2}{\prop_del_aux:w {\toks_gset:Nn #1}{#2}}}
\cs_new_protected:Npn \prop_del_aux:w #1#2#3#4#5{
```

```
  \cs_set_nopar:Npn \prop_tmp:w {#4}
  \quark_if_no_value:NF \prop_tmp:w {
    \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#2\q_prop\q_no_value {#1{#3##1}}
    \prop_tmp:w #5
  }
}
\cs_generate_variant:Nn \prop_del:Nn { NV }
\cs_generate_variant:Nn \prop_gdel:Nn { NV }
%
```

`\prop_gput_if_new:Nnn`
`\prop_put_if_new_aux:w`

`\prop_gput_if_new:Nnn` ⟨*prop*⟩ ⟨*key*⟩ ⟨*info*⟩ is equivalent to

```
\prop_if_in:NnTF ⟨prop⟩⟨key⟩
    {}%
    {\prop_gput:Nnn
        ⟨property_list⟩
        ⟨key⟩
        ⟨info⟩}
```

Here we go (listening to Porgy & Bess in a recording with Ella F. and Louis A. which makes writing macros sometimes difficult; I find myself humming instead of working):

```
\cs_new_protected:Npn \prop_gput_if_new:Nnn #1#2{
  \prop_split_aux:Nnn #1{#2}{\prop_put_if_new_aux:w #1{#2}}}
\cs_new_protected:Npn \prop_put_if_new_aux:w #1#2#3#4#5#6{
  \tl_if_empty:nT {#5}{#1{\q_prop#2\q_prop{#6}#3}}}
```

110.2 Predicates and conditionals

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

This conditional takes a ⟨*prop*⟩ as its argument and evaluates either the true or the false case, depending on whether or not ⟨*prop*⟩ contains any properties.

```
\prg_new_eq_conditional:NNn \prop_if_empty:N \toks_if_empty:N {p,TF,T,F}
\prg_new_eq_conditional:NNn \prop_if_empty:c \toks_if_empty:c {p,TF,T,F}
```

`\prop_if_eq_p:NN`
`\prop_if_eq_p:cN`
`\prop_if_eq_p:Nc`
`\prop_if_eq_p:cc`
`\prop_if_eq:NNTF`
`\prop_if_eq:NcTF`
`\prop_if_eq:cNTF`
`\prop_if_eq:ccTF`

These functions test whether two property lists are equal.

```
\prg_new_eq_conditional:NNn \prop_if_eq:NN \toks_if_eq:NN {p,TF,T,F}
\prg_new_eq_conditional:NNn \prop_if_eq:cN \toks_if_eq:cN {p,TF,T,F}
\prg_new_eq_conditional:NNn \prop_if_eq:Nc \toks_if_eq:Nc {p,TF,T,F}
\prg_new_eq_conditional:NNn \prop_if_eq:cc \toks_if_eq:cc {p,TF,T,F}
```

`\prop_if_in:NnTF`
`\prop_if_in:NVTF`
`\prop_if_in:NoTF`
`\prop_if_in:cnTF`
`\prop_if_in:ccTF`
`\prop_if_in_aux:w`

`\prop_if_in:NnTF` ⟨*property_list*⟩ ⟨*key*⟩ ⟨*true_case*⟩ ⟨*false_case*⟩ will check whether or not ⟨*key*⟩ is on the ⟨*property_list*⟩ and then select either the true or false case.

```
\prg_new_protected_conditional:Nnn \prop_if_in:Nn {TF,T,F} {
  \prop_split_aux:Nnn #1 {#2} {\prop_if_in_aux:w}
```

```
}
\cs_new_nopar:Npn \prop_if_in_aux:w #1#2#3 {
  \quark_if_no_value:nTF {#2} {\prg_return_false:} {\prg_return_true:}
}

\cs_generate_variant:Nn \prop_if_in:NnTF {NV,No,cn,cc}
\cs_generate_variant:Nn \prop_if_in:NnT  {NV,No,cn,cc}
\cs_generate_variant:Nn \prop_if_in:NnF  {NV,No,cn,cc}
```

110.3 Mapping functions

`\prop_map_function:NN`
`\prop_map_function:cN`
`\prop_map_function:Nc`
`\prop_map_function:cc`
`\prop_map_function_aux:w`

Maps a function on every entry in the property list. The function must take 2 arguments: a key and a value.

First, some failed attempts:

```
\cs_new_nopar:Npn \prop_map_function:NN #1#2{
  \exp_after:wN \prop_map_function_aux:w
  \exp_after:wN #2 \toks_use:N #1 \q_prop{}\q_prop \q_no_value \q_stop
}
\cs_new_nopar:Npn \prop_map_function_aux:w #1\q_prop#2\q_prop#3{
  \if_predicate:w \tl_if_empty_p:n{#2}
    \exp_after:wN \prop_map_break:
  \fi:
  #1{#2}{#3}
  \prop_map_function_aux:w #1
}
```

problem with the above implementation is that an empty key stops the mapping but all other functions in the module allow the use of empty keys (as one value)

```
\cs_set_nopar:Npn \prop_map_function:NN #1#2{
  \exp_after:wN \prop_map_function_aux:w
  \exp_after:wN #2 \toks_use:N #1 \q_prop \q_no_value \q_prop \q_no_value
}
\cs_set_nopar:Npn \prop_map_function_aux:w #1\q_prop#2\q_prop#3{
  \quark_if_no_value:nF{#2}
    {
     #1{#2}{#3}
     \prop_map_function_aux:w #1
    }
}
```

problem with the above implementation is that `\quark_if_no_value:nF` is fairly slow and if `\quark_if_no_value:NF` is used instead we have to do an assignment thus making the mapping not expandable (is that important?)

Here's the current version of the code:

```
\cs_set_nopar:Npn \prop_map_function:NN #1#2 {
    \exp_after:wN \prop_map_function_aux:w
    \exp_after:wN #2 \toks_use:N #1 \q_prop \q_nil \q_prop \q_no_value \q_stop
}
\cs_set:Npn \prop_map_function_aux:w #1 \q_prop #2 \q_prop #3 {
  \if_meaning:w \q_nil #2
    \exp_after:wN \prop_map_break:
  \fi:
  #1{#2}{#3}
  \prop_map_function_aux:w #1
}
```

(potential) problem with the above implementation is that it will return true if #2 contains more than just \q_nil thus executing whatever follows. Claim: this can't happen :-) so we should be ok

```
\cs_generate_variant:Nn \prop_map_function:NN {c,Nc,cc}
```

\prop_map_inline:Nn
\prop_map_inline:cn
\g_prop_inline_level_int

The inline functions are straight forward. It takes longer to test if the list is empty than to run it on an empty list so we don't waste time doing that.

```
\int_new:N \g_prop_inline_level_int
\cs_new_protected_nopar:Npn \prop_map_inline:Nn #1#2 {
  \int_gincr:N \g_prop_inline_level_int
  \cs_gset:cpn {prop_map_inline_ \int_use:N \g_prop_inline_level_int :n}
    ##1##2{#2}
  \prop_map_function:Nc #1
    {prop_map_inline_ \int_use:N \g_prop_inline_level_int :n}
  \int_gdecr:N \g_prop_inline_level_int
}

\cs_generate_variant:Nn\prop_map_inline:Nn {cn}
```

\prop_map_break:

The break statement.

```
\cs_new_eq:NN \prop_map_break: \use_none_delimit_by_q_stop:w

⟨/initex | package⟩
```

Show token usage:

```
⟨*showmemory⟩
%\showMemUsage
⟨/showmemory⟩
```

111 l3io implementation

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

111.1 Variables and constants

\c_iow_term_stream
\c_ior_term_stream
\c_iow_log_stream
\c_ior_log_stream

Here we allocate two output streams for writing to the transcript file only (\c_iow_log_stream) and to both the terminal and transcript file (\c_iow_term_stream). Both can be used to read from and have equivalent \c_ior versions.

```
\cs_new_eq:NN \c_iow_term_stream \c_sixteen
\cs_new_eq:NN \c_ior_term_stream \c_sixteen
\cs_new_eq:NN \c_iow_log_stream  \c_minus_one
\cs_new_eq:NN \c_ior_log_stream  \c_minus_one
```

\c_iow_streams_tl
\c_ior_streams_tl

The list of streams available, by number.

```
\tl_const:Nn \c_iow_streams_tl
  {
    \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
  }
\cs_new_eq:NN \c_ior_streams_tl \c_iow_streams_tl
```

\g_iow_streams_prop
\g_ior_streams_prop

The allocations for streams are stored in property lists, which are set up to have a 'full' set of allocations from the start. In package mode, a few slots are always taken, so these are blocked off from use.

```
\prop_new:N \g_iow_streams_prop
\prop_new:N \g_ior_streams_prop
⟨/initex | package⟩
⟨*package⟩
\prop_put:Nnn \g_iow_streams_prop { 0 } { LaTeX2e~reserved }
\prop_put:Nnn \g_iow_streams_prop { 1 } { LaTeX2e~reserved }
\prop_put:Nnn \g_iow_streams_prop { 2 } { LaTeX2e~reserved }
\prop_put:Nnn \g_ior_streams_prop { 0 } { LaTeX2e~reserved }
⟨/package⟩
⟨*initex | package⟩
```

`\l_iow_stream_int`
`\l_ior_stream_int`

Used to track the number allocated to the stream being created: this is taken from the property list but does alter.

```
\int_new:N \l_iow_stream_int
\cs_new_eq:NN \l_ior_stream_int \l_iow_stream_int
```

111.2 Stream management

`\iow_raw_new:N`
`\ior_raw_new:N`

The lowest level for stream management is actually creating raw TEX streams. As these are very limited (even with ε-TEX) this should not be addressed directly.

```
⟨/initex | package⟩
⟨*initex⟩
\alloc_setup_type:nnn { iow } \c_zero \c_sixteen
\cs_new_protected_nopar:Npn \iow_raw_new:N #1 {
  \alloc_reg:NnNN g { iow } \tex_chardef:D #1
}
\alloc_setup_type:nnn { ior } \c_zero \c_sixteen
\cs_new_protected_nopar:Npn \ior_raw_new:N #1 {
  \alloc_reg:NnNN g { ior } \tex_chardef:D #1
}
⟨/initex⟩
⟨*package⟩
\cs_set_eq:NN \iow_raw_new:N \newwrite
\cs_set_eq:NN \ior_raw_new:N \newread
⟨/package⟩
⟨*initex | package⟩
\cs_generate_variant:Nn \iow_raw_new:N { c }
\cs_generate_variant:Nn \ior_raw_new:N { c }
```

`\iow_new:N`
`\iow_new:c`
`\ior_new:N`
`\ior_new:c`

These are not needed but are included for consistency with other variable types.

```
\cs_new_protected_nopar:Npn \iow_new:N #1 {
  \cs_gnew_eq:NN #1 \c_iow_log_stream
}
\cs_generate_variant:Nn \iow_new:N { c }
\cs_new_protected_nopar:Npn \ior_new:N #1 {
```

```
  \cs_gnew_eq:NN #1 \c_ior_log_stream
}
\cs_generate_variant:Nn \ior_new:N { c }
```

`\iow_open:Nn` `\iow_open:cn` `\ior_open:Nn` `\ior_open:cn` In both cases, opening a stream starts with a call to the closing function: this is safest. There is then a loop through the allocation number list to find the first free stream number. When one is found the allocation can take place, the information can be stored and finally the file can actually be opened.

```
\cs_new_protected_nopar:Npn \iow_open:Nn #1#2 {
  \iow_close:N #1
  \int_set:Nn \l_iow_stream_int { \c_sixteen }
  \tl_map_function:NN \c_iow_streams_tl \iow_alloc_write:n
  \intexpr_compare:nTF { \l_iow_stream_int = \c_sixteen }
    { \msg_kernel_error:nn { iow } { streams-exhausted } }
    {
      \iow_stream_alloc:N #1
      \prop_gput:NVn \g_iow_streams_prop \l_iow_stream_int {#2}
      \tex_immediate:D \tex_openout:D #1#2 \scan_stop:
    }
}
\cs_generate_variant:Nn \iow_open:Nn { c }
\cs_new_protected_nopar:Npn \ior_open:Nn #1#2 {
  \ior_close:N #1
  \int_set:Nn \l_ior_stream_int { \c_sixteen }
  \tl_map_function:NN \c_ior_streams_tl \ior_alloc_read:n
  \intexpr_compare:nTF { \l_ior_stream_int = \c_sixteen }
    { \msg_kernel_error:nn { ior } { streams-exhausted } }
    {
      \ior_stream_alloc:N  #1
      \prop_gput:NVn \g_ior_streams_prop \l_ior_stream_int {#2}
      \tex_openin:D #1#2 \scan_stop:
    }
}
\cs_generate_variant:Nn \ior_open:Nn { c }
```

`\iow_alloc_write:n` `\ior_alloc_read:n` These functions are used to see if a particular stream is available. The property list contains file names for streams in use, so any unused ones are for the taking.

```
\cs_new_protected_nopar:Npn \iow_alloc_write:n #1 {
  \prop_if_in:NnF \g_iow_streams_prop {#1}
    {
      \int_set:Nn \l_iow_stream_int {#1}
      \tl_map_break:
    }
}
\cs_new_protected_nopar:Npn \ior_alloc_read:n #1 {
  \prop_if_in:NnF \g_iow_streams_prop {#1}
    {
```

```
      \int_set:Nn \l_ior_stream_int {#1}
      \tl_map_break:
    }
}
```

\iow_stream_alloc:N
\ior_stream_alloc:N
\iow_stream_alloc_aux:
\ior_stream_alloc_aux:
\g_iow_tmp_stream
\g_ior_tmp_stream

Allocating a raw stream is much easier in initex mode than for the package. For the format, all streams will be allocated by l3io and so there is a simple check to see if a raw stream is actually available. On the other hand, for the package there will be non-managed streams. So if the managed one is not open, a check is made to see if some other managed stream is available before deciding to open a new one. If a new one is needed, we get the number allocated by LaTeX 2_ε to get 'back on track' with allocation.

```
\cs_new_protected_nopar:Npn \iow_stream_alloc:N #1 {
  \cs_if_exist:cTF { g_iow_ \int_use:N \l_iow_stream_int _stream }
    { \cs_gset_eq:Nc #1 { g_iow_ \int_use:N \l_iow_stream_int _stream } }
    {
⟨/initex | package⟩
⟨*package⟩
      \iow_stream_alloc_aux:
      \intexpr_compare:nT { \l_iow_stream_int = \c_sixteen }
        {
          \iow_raw_new:N \g_iow_tmp_stream
          \int_set:Nn \l_iow_stream_int { \g_iow_tmp_stream }
          \cs_gset_eq:cN
            { g_iow_ \int_use:N \l_iow_stream_int _stream }
            \g_iow_tmp_stream
        }
⟨/package⟩
⟨*initex⟩
      \iow_raw_new:c { g_iow_ \int_use:N \l_iow_stream_int _stream }
⟨/initex⟩
⟨*initex | package⟩
      \cs_gset_eq:Nc #1 { g_iow_ \int_use:N \l_iow_stream_int _stream }
    }
}
⟨/initex | package⟩
⟨*package⟩
\cs_new_protected_nopar:Npn \iow_stream_alloc_aux: {
  \int_incr:N \l_iow_stream_int
  \intexpr_compare:nT
    { \l_iow_stream_int < \c_sixteen }
    {
      \cs_if_exist:cTF { g_iow_ \int_use:N \l_iow_stream_int _stream }
        {
          \prop_if_in:NVT \g_iow_streams_prop \l_iow_stream_int
            { \iow_stream_alloc_aux: }
        }
        { \iow_stream_alloc_aux: }
    }
```

```
}
⟨/package⟩
⟨*initex | package⟩
\cs_new_protected_nopar:Npn \ior_stream_alloc:N #1 {
  \cs_if_exist:cTF { g_ior_ \int_use:N \l_ior_stream_int _stream }
    { \cs_gset_eq:Nc #1 { g_ior_ \int_use:N \l_ior_stream_int _stream } }
    {
⟨/initex | package⟩
⟨*package⟩
      \ior_stream_alloc_aux:
      \intexpr_compare:nT { \l_ior_stream_int = \c_sixteen }
        {
          \ior_raw_new:N \g_ior_tmp_stream
          \int_set:Nn \l_ior_stream_int { \g_ior_tmp_stream }
          \cs_gset_eq:cN
            { g_ior_ \int_use:N \l_iow_stream_int _stream }
            \g_ior_tmp_stream
        }
⟨/package⟩
⟨*initex⟩
      \ior_raw_new:c { g_ior_ \int_use:N \l_ior_stream_int _stream }
⟨/initex⟩
⟨*initex | package⟩
      \cs_gset_eq:Nc #1 { g_ior_ \int_use:N \l_ior_stream_int _stream }
    }
}
⟨/initex | package⟩
⟨*package⟩
\cs_new_protected_nopar:Npn \ior_stream_alloc_aux: {
  \int_incr:N \l_ior_stream_int
  \intexpr_compare:nT
    { \l_ior_stream_int < \c_sixteen }
    {
       \cs_if_exist:cTF { g_ior_ \int_use:N \l_ior_stream_int _stream }
         {
           \prop_if_in:NVT \g_ior_streams_prop \l_ior_stream_int
             { \ior_stream_alloc_aux: }
         }
         { \ior_stream_alloc_aux: }
    }
}
⟨/package⟩
⟨*initex | package⟩
```

\iow_close:N
\iow_close:c
\iow_close:N
\iow_close:c

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.

```
\cs_new_protected_nopar:Npn \iow_close:N #1 {
  \cs_if_exist:NT #1
```

```
    {
      \intexpr_compare:nF { #1 = \c_minus_one }
        {
          \tex_immediate:D \tex_closeout:D #1
          \prop_gdel:NV \g_iow_streams_prop #1
          \cs_gundefine:N #1
        }
    }
}
\cs_generate_variant:Nn \iow_close:N { c }
\cs_new_protected_nopar:Npn \ior_close:N #1 {
  \cs_if_exist:NT #1
    {
      \intexpr_compare:nF { #1 = \c_minus_one }
        {
          \tex_closeout:D #1
          \prop_gdel:NV \g_ior_streams_prop #1
          \cs_gundefine:N #1
        }
    }
}
\cs_generate_variant:Nn \ior_close:N { c }
```

`\iow_open_streams:`
`\ior_open_streams:`

Simply show the property lists.

```
\cs_new_protected_nopar:Npn \iow_open_streams: {
  \prop_display:N \g_iow_streams_prop
}
\cs_new_protected_nopar:Npn \ior_open_streams: {
  \prop_display:N \g_ior_streams_prop
}
```

Text for the error messages.

```
\msg_kernel_new:nnnn { iow } { streams-exhausted }
  {Output streams exhausted}
  {%
    TeX can only open up to 16 output streams at one time.\\%
    All 16 are currently in use, and something wanted to open
    another one.%
  }
\msg_kernel_new:nnnn { ior } { streams-exhausted }
  {Input streams exhausted}
  {%
    TeX can only open up to 16 input streams at one time.\\%
    All 16 are currently in use, and something wanted to open
    another one.%
  }
```

111.3 Immediate writing

\iow_now:Nx

An abbreviation for an often used operation, which immediately writes its second argument expanded to the output stream.

```
\cs_new_protected_nopar:Npn \iow_now:Nx { \tex_immediate:D \iow_shipout_x:Nn }
```

\iow_now:Nn

This routine writes the second argument onto the output stream without expansion. If this stream isn't open, the output goes to the terminal instead. If the first argument is no output stream at all, we get an internal error.

```
\cs_new_protected_nopar:Npn \iow_now:Nn #1#2 {
  \iow_now:Nx #1 { \exp_not:n {#2} }
}
```

\iow_log:n
\iow_log:x
\iow_term:n
\iow_term:x

Now we redefine two functions for which we needed a definition very early on.

```
\cs_set_protected_nopar:Npn \iow_log:x  { \iow_now:Nx \c_iow_log_stream  }
\cs_new_protected_nopar:Npn \iow_log:n  { \iow_now:Nn \c_iow_log_stream  }
\cs_set_protected_nopar:Npn \iow_term:x { \iow_now:Nx \c_iow_term_stream }
\cs_new_protected_nopar:Npn \iow_term:n { \iow_now:Nn \c_iow_term_stream }
```

\iow_now_when_avail:Nn
\iow_now_when_avail:cn
\iow_now_when_avail:Nx
\iow_now_when_avail:cx

For writing only if the stream requested is open at all.

```
\cs_new_protected_nopar:Npn \iow_now_when_avail:Nn #1 {
  \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nn #1 }
}
\cs_generate_variant:Nn \iow_now_when_avail:Nn { c }
\cs_new_protected_nopar:Npn \iow_now_when_avail:Nx #1 {
  \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nx #1 }
}
\cs_generate_variant:Nn \iow_now_when_avail:Nx { c }
```

\iow_now_buffer_safe:Nn
\iow_now_buffer_safe:Nx
_now_buffer_safe_expanded_aux:w

Another type of writing onto an output stream is used for potentially long token sequences. We break the output lines at every blank in the second argument. This avoids the problem of buffer overflow when reading back, or badly broken lines on systems with limited file records. The only thing we have to take care of, is the danger of two blanks in succession since these get converted into a \par when we read the stuff back. But this can happen only if things like two spaces find their way into the second argument. Usually, multiple spaces are removed by TeX's scanner.

```
\cs_new_protected_nopar:Npn \iow_now_buffer_safe:Nn {
  \iow_now_buffer_safe_aux:w \iow_now:Nx
}
\cs_new_protected_nopar:Npn \iow_now_buffer_safe:Nx {
  \iow_now_buffer_safe_aux:w \iow_now:Nn
}
\cs_new_protected_nopar:Npn \iow_now_buffer_safe_aux:w #1#2#3 {
  \group_begin: \tex_newlinechar:D`\ #1#2 {#3} \group_end:
}
```

111.4 Deferred writing

\iow_shipout_x:Nn
\iow_shipout_x:Nx

First the easy part, this is the primitive.

```
\cs_set_eq:NN \iow_shipout_x:Nn \tex_write:D
\cs_generate_variant:Nn \iow_shipout_x:Nn  {Nx }
```

\iow_shipout:Nn
\iow_shipout:Nx

With ε-TeX available deferred writing is easy.

```
\cs_new_protected_nopar:Npn \iow_shipout:Nn #1#2 {
  \iow_shipout_x:Nn #1 { \exp_not:n {#2} }
}
\cs_generate_variant:Nn \iow_shipout:Nn { Nx }
```

112 Special characters for writing

\iow_newline:

Global variable holding the character that forces a new line when something is written to an output stream.

```
\cs_new_nopar:Npn \iow_newline: { ^^J }
```

\iow_char:N

Function to write any escaped char to an output stream.

```
\cs_new:Npn \iow_char:N #1 { \cs_to_str:N #1 }
```

112.1 Reading input

\if_eof:w

A simple primitive renaming.

```
\cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

\ior_if_eof_p:N
\ior_if_eof:N*TF*

To test if some particular input stream is exhausted the following conditional is provided. As the pool model means that closed streams are undefined control sequences, the test has two parts.

```
\prg_new_conditional:Nnn \ior_if_eof:N { p , TF , T , F } {
  \cs_if_exist:NTF #1
    { \tex_ifeof:D #1 \prg_return_true: \else: \prg_return_false: \fi: }
    { \prg_return_true: }
}
```

\ior_to:NN
\ior_gto:NN

And here we read from files.

```
\cs_new_protected_nopar:Npn \ior_to:NN #1#2 {
  \tex_read:D #1 to #2
}
\cs_new_protected_nopar:Npn \ior_gto:NN {
  \pref_global:D \ior_to:NN
}
⟨/initex | package⟩
```

113 l3msg implementation

The usual lead-off.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

LaTeX is handling context, so the TeX "noise" is turned down.

```
\int_set:Nn \tex_errorcontextlines:D { \c_minus_one }
```

113.1 Variables and constants

\c_msg_fatal_tl
\c_msg_error_tl
\c_msg_warning_tl
\c_msg_info_tl

Header information.

```
\tl_const:Nn \c_msg_fatal_tl   { Fatal~Error }
\tl_const:Nn \c_msg_error_tl   { Error }
\tl_const:Nn \c_msg_warning_tl { Warning }
\tl_const:Nn \c_msg_info_tl    { Info }
```

\c_msg_coding_error_text_tl
\c_msg_fatal_text_tl
\c_msg_help_text_tl
\c_msg_kernel_bug_text_tl
\c_msg_kernel_bug_more_text_tl
\c_msg_no_info_text_tl
\c_msg_return_text_tl

Simple pieces of text for messages.

```
\tl_const:Nn \c_msg_coding_error_text_tl {
  This~is~a~coding~error.
  \msg_two_newlines:
}
\tl_const:Nn \c_msg_fatal_text_tl {
  This~is~a~fatal~error:~LaTeX~will~abort
}
\tl_const:Nn \c_msg_help_text_tl {
  For~immediate~help~type~H~<return>
}
\tl_const:Nn \c_msg_kernel_bug_text_tl {
  This~is~a~LaTeX~bug:~check~coding!
}
\tl_const:Nn \c_msg_kernel_bug_more_text_tl {
  There~is~a~coding~bug~somewhere~around~here.
  \msg_newline:
  This~probably~needs~examining~by~an~expert.
  \c_msg_return_text_tl
}
\tl_const:Nn \c_msg_no_info_text_tl {
  LaTeX~does~not~know~anything~more~about~this~error,~sorry.
  \c_msg_return_text_tl
}
```

```
\tl_const:Nn \c_msg_return_text_tl {
  \msg_two_newlines:
  Try~typing~<return>~to~proceed.
  \msg_newline:
  If~that~doesn't~work,~type~X~<return>~to~quit
}
```

`\c_msg_hide_tl<spaces>` An empty variable with a number of (category code 11) spaces at the end of its name. This is used to push the TeX part of an error message "off the screen".

No indentation here as ␣ is a letter!

```
\group_begin:
\char_make_letter:N\ %
\tl_to_lowercase:n{%
\group_end:%
\tl_const:Nn%
\c_msg_hide_tl                                                                                    %
{}%
}%
```

`\c_msg_on_line_tl` Text for "on line".

```
\tl_const:Nn \c_msg_on_line_tl { on~line }
```

`\c_msg_text_prefix_tl`
`\c_msg_more_text_prefix_tl`

Prefixes for storage areas.

```
\tl_const:Nn \c_msg_text_prefix_tl      { msg_text ~>~ }
\tl_const:Nn \c_msg_more_text_prefix_tl { msg_text_more ~>~ }
```

`\l_msg_class_tl`
`\l_msg_current_class_tl`
`\l_msg_current_module_tl`

For holding the current message method and that for redirection.

```
\tl_new:N \l_msg_class_tl
\tl_new:N \l_msg_current_class_tl
\tl_new:N \l_msg_current_module_tl
```

`\l_msg_names_clist` Lists used for filtering.

```
\clist_new:N \l_msg_names_clist
```

`\l_msg_redirect_classes_prop`
`\l_msg_redirect_names_prop`

For filtering messages, a list of all messages and of those which have to be modified is required.

```
\prop_new:N \l_msg_redirect_classes_prop
\prop_new:N \l_msg_redirect_names_prop
```

`\l_msg_redirect_classes_clist` To prevent an infinite loop.

```
\clist_new:N \l_msg_redirect_classes_clist
```

`\l_msg_tmp_tl` A scratch variable.

```
\tl_new:N \l_msg_tmp_tl
```

113.2 Output helper functions

\msg_line_number:
\msg_line_context:

For writing the line number nicely.

```
\cs_new_nopar:Npn \msg_line_number: {
  \toks_use:N \tex_inputlineno:D
}
\cs_new_nopar:Npn \msg_line_context: {
  \c_msg_on_line_tl
  \c_space_tl
  \msg_line_number:
}
```

\msg_newline:
\msg_two_newlines:

Always forces a new line.

```
\cs_new_nopar:Npn \msg_newline:      { ^^J }
\cs_new_nopar:Npn \msg_two_newlines: { ^^J ^^J }
```

113.3 Generic functions

The lowest level functions make no assumptions about modules, *etc.*

\msg_generic_new:nnn
\msg_generic_new:nn

Creating a new message is basically the same as the non-checking version, and so after a check everything hands over.

```
\cs_new_protected_nopar:Npn \msg_generic_new:nnn #1 {
  \chk_if_free_cs:c { \c_msg_text_prefix_tl #1 :xxxx }
  \msg_generic_set:nnn {#1}
}
\cs_new_protected_nopar:Npn \msg_generic_new:nn #1 {
  \chk_if_free_cs:c { \c_msg_text_prefix_tl #1 :xxxx }
  \msg_generic_set:nn {#1}
}
```

\msg_generic_set:nnn
\msg_generic_set:nn
\msg_generic_set_clist:n

Creating a message is quite simple. There must be a short text part, while the longer text may or may not be available.

```
\cs_new_protected_nopar:Npn \msg_generic_set:nnn #1#2#3 {
  \msg_generic_set_clist:n {#1}
  \cs_set:cpn { \c_msg_text_prefix_tl #1 :xxxx } ##1##2##3##4 {#2}
  \cs_set:cpn { \c_msg_more_text_prefix_tl #1 :xxxx } ##1##2##3##4 {#3}
}
\cs_new_protected_nopar:Npn \msg_generic_set:nn #1#2 {
  \msg_generic_set_clist:n {#1}
  \cs_set:cpn { \c_msg_text_prefix_tl #1 :xxxx } ##1##2##3##4 {#2}
  \cs_set_eq:cN { \c_msg_more_text_prefix_tl #1 :xxxx } \c_undefined
}
\cs_new_protected_nopar:Npn \msg_generic_set_clist:n #1 {
```

```
  \clist_if_in:NnF \l_msg_names_clist { // #1 / } {
    \clist_put_right:Nn \l_msg_names_clist { // #1 / }
  }
}
```

\msg_direct_interrupt:xxxx
\msg_direct_interrupt:n

The low-level interruption macro is rather opaque, unfortunately. The idea here is to create a a message which hides all of TEX's own information by filling the output up with spaces. To achieve this, spaces have to be letters: hence no indentation. The odd \c_msg_hide_tl<spaces> actually does the hiding: it is the large run of spaces in the name that is important here. The meaning of \\ is altered so that the explanation text is a simple run whilst the initial error has line-continuation shown.

```
\group_begin:
  \char_set_lccode:nn {`\&} {`\ } % {
  \char_set_lccode:w `\} = `\ \scan_stop:
  \char_make_active:N \&
  \char_make_letter:N\ %
\tl_to_lowercase:n{%
\group_end:%
\cs_new_protected:Npn\msg_direct_interrupt:xxxx#1#2#3#4{%
\group_begin:%
\cs_set_eq:NN\\\msg_newline:%
\cs_set_eq:NN\ \c_space_tl%
\msg_direct_interrupt_aux:n{#4}%
\cs_set_nopar:Npn\\{\msg_newline:#3}%
\tex_errhelp:D\l_msg_tmp_tl%
\cs_set:Npn&{%
\tex_errmessage:D{%
#1\msg_newline:%
#2\msg_two_newlines:%
\c_msg_help_text_tl%
\c_msg_hide_tl                                                                 %
}%
}%
&%
\group_end:%
}%
}%
\cs_new_protected:Npn \msg_direct_interrupt_aux:n #1 {
  \tl_if_empty:nTF {#1} {
    \tl_set:Nx \l_msg_tmp_tl { { \c_msg_no_info_text_tl } }
  }{
    \tl_set:Nx \l_msg_tmp_tl { {#1} }
  }
}
```

\msg_direct_log:xx
\msg_direct_term:xx

Printing to the log or terminal without a stop is rather easier.

```
\cs_new_protected:Npn \msg_direct_log:xx #1#2 {
```

```
  \group_begin:
    \cs_set:Npn \\ { \msg_newline: #2 }
    \cs_set_eq:NN \ \c_space_tl
    \iow_log:x { #1 \msg_newline: }
  \group_end:
}
\cs_new_protected:Npn \msg_direct_term:xx #1#2 {
  \group_begin:
    \cs_set:Npn \\ { \msg_newline: #2 }
    \cs_set_eq:NN \ \c_space_tl
    \iow_term:x { #1 \msg_newline: }
  \group_end:
}
```

113.4 General functions

The main functions for messaging are built around the separation of module from the message name. These have short names as they will be widely used.

\msg_new:nnnn
\msg_new:nnn
\msg_set:nnnn
\msg_set:nnn

For making messages: all aliases.

```
\cs_new_protected_nopar:Npn \msg_new:nnnn #1#2 {
  \msg_generic_new:nnn { #1 / #2 }
}
\cs_new_protected_nopar:Npn \msg_new:nnn #1#2 {
  \msg_generic_new:nn { #1 / #2 }
}
\cs_new_protected_nopar:Npn \msg_set:nnnn #1#2 {
  \msg_generic_set:nnn { #1 / #2 }
}
\cs_new_protected_nopar:Npn \msg_set:nnn #1#2 {
  \msg_generic_set:nn { #1 / #2 }
}
```

\msg_class_new:nn
\msg_class_set:nn

Creating a new class produces three new functions, with varying numbers of arguments. The \msg_class_loop:n function is set up so that redirection will work as desired.

```
\cs_new_protected_nopar:Npn \msg_class_new:nn #1 {
  \chk_if_free_cs:c { msg_ #1 :nnxxxx }
  \prop_new:c { l_msg_redirect_ #1 _prop }
  \msg_class_set:nn {#1}
}
\cs_new_protected_nopar:Npn \msg_class_set:nn #1#2 {
  \prop_clear:c { l_msg_redirect_ #1 _prop }
  \cs_set_protected:cpn { msg_ #1 :nnxxxx } ##1##2##3##4##5##6 {
    \msg_use:nnnnxxxx {#1} {#2} {##1} {##2} {##3} {##4} {##5} {##6}
  }
  \cs_set_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5 {
```

```
    \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { }
  }
  \cs_set_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4 {
    \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { }
  }
  \cs_set_protected:cpx { msg_ #1 :nnx } ##1##2##3 {
    \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { }
  }
  \cs_set_protected:cpx { msg_ #1 :nn } ##1##2 {
    \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} { } { } { } { }
  }
}
```

\msg_use:nnnnxxxx

The main message-using macro creates two auxiliary functions: one containing the code for the message, and the second a loop function. There is then a hand-off to the system for checking if redirection is needed.

```
\cs_new_protected:Npn \msg_use:nnnnxxxx #1#2#3#4#5#6#7#8 {
  \cs_set_nopar:Npn \msg_use_code: {
    \clist_clear:N \l_msg_redirect_classes_clist
    #2
  }
  \cs_set:Npn \msg_use_loop:n ##1 {
    \clist_if_in:NnTF \l_msg_redirect_classes_clist {#1} {
      \msg_kernel_error:nn { msg } { redirect-loop } {#1}
    }{
      \clist_put_right:Nn \l_msg_redirect_classes_clist {#1}
      \cs_if_exist:cTF { msg_ ##1 :nnxxxx } {
        \use:c { msg_ ##1 :nnxxxx } {#3} {#4} {#5} {#6} {#7} {#8}
      }{
        \msg_kernel_error:nnx { msg } { message-class-unknown } {##1}
      }
    }
  }
  \cs_if_exist:cTF { \c_msg_text_prefix_tl #3 / #4 :xxxx } {
    \msg_use_aux:nnn {#1} {#3} {#4}
  }{
    \msg_kernel_error:nnxx { msg } { message-unknown } {#3} {#4}
  }
}
```

\msg_use_code:
\msg_use_loop:

Blank definitions are initially created for these functions.

```
\cs_new_nopar:Npn \msg_use_code: { }
\cs_new:Npn \msg_use_loop:n #1 { }
```

\msg_use_aux:nnn

The first auxiliary macro looks for a match by name: the most restrictive check.

```
\cs_new_protected_nopar:Npn \msg_use_aux:nnn #1#2#3 {
```

```
  \tl_set:Nn \l_msg_current_class_tl  {#1}
  \tl_set:Nn \l_msg_current_module_tl {#2}
  \prop_if_in:NnTF \l_msg_redirect_names_prop { // #2 / #3 / } {
     \msg_use_loop_check:nn { names } { // #2 / #3 / }
  }{
    \msg_use_aux:nn {#1} {#2}
  }
}
```

`\msg_use_aux:nn` The second function checks for general matches by module or for all modules.

```
\cs_new_protected_nopar:Npn \msg_use_aux:nn #1#2 {
  \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } {#2} {
    \msg_use_loop_check:nn {#1} {#2}
  }{
    \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } { * } {
      \msg_use_loop_check:nn {#1} { * }
    }{
      \msg_use_code:
    }
  }
}
```

`\msg_use_loop_check:nn` When checking whether to loop, the same code is needed in a few places.

```
\cs_new_protected:Npn \msg_use_loop_check:nn #1#2 {
  \prop_get:cnN { l_msg_redirect_ #1 _prop } {#2} \l_msg_class_tl
  \tl_if_eq:NNTF \l_msg_current_class_tl \l_msg_class_tl {
    \msg_use_code:
  }{
    \msg_use_loop:n { \l_msg_class_tl }
  }
}
```

`\msg_fatal:nnxxxx`
`\msg_fatal:nnxxx`
`\msg_fatal:nnxx`
`\msg_fatal:nnx`
`\msg_fatal:nn`

For fatal errors, after the error message TEX bails out.

```
\msg_class_new:nn { fatal } {
  \msg_direct_interrupt:xxxx
    { \c_msg_fatal_tl \msg_two_newlines: }
    {
      ( \c_msg_fatal_tl ) \c_space_tl
      \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
    }
    { ( \c_msg_fatal_tl ) \c_space_tl }
    { \c_msg_fatal_text_tl }
  \tex_end:D
}
```

`\msg_error:nnxxxx`
`\msg_error:nnxxx`
`\msg_error:nnxx`
`\msg_error:nnx`
`\msg_error:nn`

For an error, the interrupt routine is called, then any recovery code is tried.

```
\msg_class_new:nn { error } {
  \msg_direct_interrupt:xxxx
    { #1~\c_msg_error_tl \msg_newline: }
    {
      ( #1 ) \c_space_tl
      \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
    }
    { ( #1 ) \c_space_tl }
    {
      \cs_if_exist:cTF { \c_msg_more_text_prefix_tl #1 / #2 :xxxx }
        {
          \use:c { \c_msg_more_text_prefix_tl #1 / #2 :xxxx }
            {#3} {#4} {#5} {#6}
        }
        { \c_msg_no_info_text_tl }
    }
}
```

\msg_warning:nnxxxx
\msg_warning:nnxxx
\msg_warning:nnxx
\msg_warning:nnx
\msg_warning:nn

Warnings are printed to the terminal.

```
\msg_class_new:nn { warning } {
  \msg_direct_term:xx {
    \c_space_tl #1 ~ \c_msg_warning_tl :~
    \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
  }
  { ( #1 ) \c_space_tl \c_space_tl }
}
```

\msg_info:nnxxxx
\msg_info:nnxxx
\msg_info:nnxx
\msg_info:nnx
\msg_info:nn

Information only goes into the log.

```
\msg_class_new:nn { info } {
  \msg_direct_log:xx {
    \c_space_tl #1~\c_msg_info_tl :~
    \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
  }
  { ( #1 ) \c_space_tl \c_space_tl }
}
```

\msg_log:nnxxxx
\msg_log:nnxxx
\msg_log:nnxx
\msg_log:nnx
\msg_log:nn

“Log” data is very similar to information, but with no extras added.

```
\msg_class_new:nn { log } {
  \msg_direct_log:xx {
    \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
  }
  { }
}
```

\msg_trace:nnxxxx
\msg_trace:nnxxx
\msg_trace:nnxx
\msg_trace:nnx
\msg_trace:nn

Trace data is the same as log data, more or less

```
\msg_class_new:nn { trace } {
  \msg_direct_log:xx {
    \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
  }
  { }
}
```

\msg_none:nnxxxx
\msg_none:nnxxx
\msg_none:nnxx
\msg_none:nnx
\msg_none:nn

The none message type is needed so that input can be gobbled.

```
\msg_class_new:nn { none } { }
```

113.5 Redirection functions

\msg_redirect_class:nn

Converts class one into class two.

```
\cs_new_protected_nopar:Npn \msg_redirect_class:nn #1#2 {
  \prop_put:cnn { l_msg_redirect_ #1 _prop } { * } {#2}
}
```

\msg_redirect_module:nnn

For when all messages of a class should be altered for a given module.

```
\cs_new_protected_nopar:Npn \msg_redirect_module:nnn #1#2#3 {
  \prop_put:cnn { l_msg_redirect_ #2 _prop } {#1} {#3}
}
```

\msg_redirect_name:nnn

Named message will always use the given class.

```
\cs_new_protected_nopar:Npn \msg_redirect_name:nnn #1#2#3 {
  \prop_put:Nnn \l_msg_redirect_names_prop { // #1 / #2 / } {#3}
}
```

113.6 Kernel-specific functions

\msg_kernel_new:nnnn
\msg_kernel_new:nnn
\msg_kernel_set:nnnn
\msg_kernel_set:nnn

The kernel needs some messages of its own. These are created using pre-built functions. Two functions are provided: one more general and one which only has the short text part.

```
\cs_new_protected_nopar:Npn \msg_kernel_new:nnnn #1#2 {
  \msg_new:nnnn { LaTeX } { #1 / #2 }
}
\cs_new_protected_nopar:Npn \msg_kernel_new:nnn #1#2 {
  \msg_new:nnn { LaTeX } { #1 / #2 }
}
\cs_new_protected_nopar:Npn \msg_kernel_set:nnnn #1#2 {
  \msg_set:nnnn { LaTeX } { #1 / #2 }
}
\cs_new_protected_nopar:Npn \msg_kernel_set:nnn #1#2 {
  \msg_set:nnn { LaTeX } { #1 / #2 }
}
```

\msg_kernel_classes_new:n

Quickly make the fewer-arguments versions.

```
\cs_new_protected_nopar:Npn \msg_kernel_classes_new:n #1 {
  \cs_new_protected:cpx { msg_kernel_ #1 :nnxxx } ##1##2##3##4##5
  {
    \exp_not:c { msg_kernel_ #1 :nnxxxx }
      {##1} {##2} {##3} {##4} {##5} { }
  }
  \cs_new_protected:cpx { msg_kernel_ #1 :nnxx } ##1##2##3##4
  {
    \exp_not:c { msg_kernel_ #1 :nnxxxx }
      {##1} {##2} {##3} {##4} { } { }
  }
  \cs_new_protected:cpx { msg_kernel_ #1 :nnx } ##1##2##3
  {
    \exp_not:c { msg_kernel_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { }
  }
  \cs_new_protected:cpx { msg_kernel_ #1 :nn } ##1##2
  {
    \exp_not:c { msg_kernel_ #1 :nnxxxx } {##1} {##2} { } { } { } { }
  }
}
```

\msg_kernel_fatal:nnxxxx
\msg_kernel_fatal:nnxxx
\msg_kernel_fatal:nnxx
\msg_kernel_fatal:nnx
\msg_kernel_fatal:nn

Fatal kernel errors cannot be re-defined.

```
\cs_new_protected:Npn \msg_kernel_fatal:nnxxxx #1#2#3#4#5#6 {
  \msg_direct_interrupt:xxxx
    { \c_msg_fatal_tl \msg_two_newlines: }
    {
      ( LaTeX ) \c_space_tl
      \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 :xxxx }
        {#3} {#4} {#5} {#6}
    }
    { ( LaTeX ) \c_space_tl}
    { \c_msg_fatal_text_tl }
  \tex_end:D
}
\msg_kernel_classes_new:n { fatal }
```

\msg_kernel_error:nnxxxx
\msg_kernel_error:nnxxx
\msg_kernel_error:nnxx
\msg_kernel_error:nnx
\msg_kernel_error:nn

Neither can kernel errors.

```
\cs_new_protected:Npn \msg_kernel_error:nnxxxx #1#2#3#4#5#6 {
  \msg_direct_interrupt:xxxx
    { LaTeX~\c_msg_error_tl \msg_newline: }
    {
      ( LaTeX ) \c_space_tl
      \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 :xxxx }
        {#3} {#4} {#5} {#6}
    }
    { ( LaTeX ) \c_space_tl }
```

```
    {
      \cs_if_exist:cTF
        { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 :xxxx }
        {
          \use:c { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 :xxxx }
            {#3} {#4} {#5} {#6}
        }
        { \c_msg_no_info_text_tl }
    }
}
\msg_kernel_classes_new:n { error }
```

`\msg_kernel_warning:nnxxxx`
`\msg_kernel_warning:nnxxx`
`\msg_kernel_warning:nnxx`
`\msg_kernel_warning:nnx`
`\msg_kernel_warning:nn`
`\msg_kernel_info:nnxxxx`
`\msg_kernel_info:nnxxx`
`\msg_kernel_info:nnxx`
`\msg_kernel_info:nnx`
`\msg_kernel_info:nn`

Life is much more simple for warnings and information messages, as these are just shortcuts to the standard classes.

```
\cs_new_protected_nopar:Npn \msg_kernel_warning:nnxxxx #1#2 {
  \msg_warning:nnxxxx { LaTeX } { #1 / #2 }
}
\msg_kernel_classes_new:n { warning }
\cs_new_protected_nopar:Npn \msg_kernel_info:nnxxxx #1#2 {
  \msg_info:nnxxxx { LaTeX } { #1 / #2 }
}
\msg_kernel_classes_new:n { info }
```

Error messages needed to actually implement the message system itself.

```
\msg_kernel_new:nnnn { msg } { message-unknown }
  { Unknown~message~'#2'~for~module~'#1'.}
  {
    \c_msg_coding_error_text_tl
    LaTeX~was~asked~to~display~a~message~called~'#2'\\
    by~the~module~'#1'~module:~this~message~does~not~exist.
    \c_msg_return_text_tl
  }
\msg_kernel_new:nnnn { msg } { message-class-unknown }
  { Unknown~message~class~'#1'. }
  {
    LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\\
    this~was~never~defined.

    \c_msg_return_text_tl
  }
\msg_kernel_new:nnnn { msg } { redirect-loop }
  { Message~redirection~loop~for~message~class~'#1'. }
  {
    LaTeX~has~been~asked~to~redirect~messages~in~an~infinite~loop.\\
    The~original~message~here~has~been~lost.
    \c_msg_return_text_tl
  }
```

\msg_kernel_bug:x The LaTeX coding bug error gets re-visited here.

```
\cs_set_protected:Npn \msg_kernel_bug:x #1 {
  \msg_direct_interrupt:xxxx
    { \c_msg_kernel_bug_text_tl }
    { !~#1 }
    { ! }
    { \c_msg_kernel_bug_more_text_tl }
}
```

```
⟨/initex | package⟩
```

114 l3box implementation

Announce and ensure that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

The code in this module is very straight forward so I'm not going to comment it very extensively.

114.1 Generic boxes

\box_new:N
\box_new:c
\box_new_local:N
\box_new_local:c

Defining a new ⟨*box*⟩ register.

```
⟨*initex⟩
\alloc_new:nnnN {box} \c_zero \c_max_register_int \tex_mathchardef:D
```

Now, remember that \box255 has a special role in TeX, it shouldn't be allocated…

```
\seq_put_right:Nn \g_box_allocation_seq {255}
⟨/initex⟩
```

When we run on top of LaTeX, we just use its allocation mechanism.

```
⟨*package⟩
\cs_new_protected:Npn \box_new:N #1 {
  \chk_if_free_cs:N #1
  \newbox #1
}
\cs_new_protected:Npn \box_new_local:N #1 {
  \chk_if_free_cs:N #1
  \int_compare:nNnTF
```

```
      \tex_currentgrouplevel:D = 0
      \newbox \locbox
    #1
}
⟨/package⟩
```

```
\cs_generate_variant:Nn \box_new:N {c}
\cs_generate_variant:Nn \box_new_local:N {c}
```

`\if_hbox:N`
`\if_vbox:N`
`\if_box_empty:N`

The primitives for testing if a ⟨*box*⟩ is empty/void or which type of box it is.

```
\cs_new_eq:NN \if_hbox:N       \tex_ifhbox:D
\cs_new_eq:NN \if_vbox:N       \tex_ifvbox:D
\cs_new_eq:NN \if_box_empty:N  \tex_ifvoid:D
```

`\box_if_horizontal_p:N`
`\box_if_horizontal_p:c`
`\box_if_vertical_p:N`
`\box_if_vertical_p:c`
`\box_if_horizontal:N`*TF*
`\box_if_horizontal:c`*TF*
`\box_if_vertical:N`*TF*
`\box_if_vertical:c`*TF*

```
\prg_new_conditional:Nnn \box_if_horizontal:N {p,TF,T,F} {
  \tex_ifhbox:D #1 \prg_return_true: \else: \prg_return_false: \fi:
}
\prg_new_conditional:Nnn \box_if_vertical:N {p,TF,T,F} {
  \tex_ifvbox:D #1 \prg_return_true: \else: \prg_return_false: \fi:
}
\cs_generate_variant:Nn \box_if_horizontal_p:N {c}
\cs_generate_variant:Nn \box_if_horizontal:NTF {c}
\cs_generate_variant:Nn \box_if_horizontal:NT  {c}
\cs_generate_variant:Nn \box_if_horizontal:NF  {c}
\cs_generate_variant:Nn \box_if_vertical_p:N {c}
\cs_generate_variant:Nn \box_if_vertical:NTF {c}
\cs_generate_variant:Nn \box_if_vertical:NT  {c}
\cs_generate_variant:Nn \box_if_vertical:NF  {c}
```

`\box_if_empty_p:N`
`\box_if_empty_p:c`
`\box_if_empty:N`*TF*
`\box_if_empty:c`*TF*

Testing if a ⟨*box*⟩ is empty/void.

```
\prg_new_conditional:Nnn \box_if_empty:N {p,TF,T,F} {
  \tex_ifvoid:D #1 \prg_return_true: \else: \prg_return_false: \fi:
}
\cs_generate_variant:Nn \box_if_empty_p:N {c}
\cs_generate_variant:Nn \box_if_empty:NTF {c}
\cs_generate_variant:Nn \box_if_empty:NT  {c}
\cs_generate_variant:Nn \box_if_empty:NF  {c}
```

`\box_set_eq:NN`
`\box_set_eq:cN`
`\box_set_eq:Nc`
`\box_set_eq:cc`

Assigning the contents of a box to be another box. This clears the second box globally (that's how TEX does it).

```
\cs_new_protected_nopar:Npn \box_set_eq:NN #1#2 {\tex_setbox:D #1 \tex_box:D #2}
\cs_generate_variant:Nn \box_set_eq:NN {cN,Nc,cc}
```

\box_gset_eq:NN
\box_gset_eq:cN
\box_gset_eq:Nc
\box_gset_eq:cc

Global version of the above.

```
\cs_new_protected_nopar:Npn \box_gset_eq:NN {\pref_global:D\box_set_eq:NN}
\cs_generate_variant:Nn \box_gset_eq:NN {cN,Nc,cc}
```

\l_last_box

A different name for this read-only primitive.

```
\cs_new_eq:NN \l_last_box \tex_lastbox:D
```

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

Set a box to the previous box.

```
\cs_new_protected_nopar:Npn \box_set_to_last:N #1{\tex_setbox:D#1\l_last_box}
\cs_generate_variant:Nn \box_set_to_last:N {c}
\cs_new_protected_nopar:Npn \box_gset_to_last:N {\pref_global:D \box_set_to_last:N}
\cs_generate_variant:Nn \box_gset_to_last:N {c}
```

\box_move_left:nn
\box_move_right:nn
\box_move_up:nn
\box_move_down:nn

Move box material in different directions.

```
\cs_new:Npn \box_move_left:nn #1#2{\tex_moveleft:D\dim_eval:n{#1}{#2}}
\cs_new:Npn \box_move_right:nn #1#2{\tex_moveright:D\dim_eval:n{#1}{#2}}
\cs_new:Npn \box_move_up:nn #1#2{\tex_raise:D\dim_eval:n{#1}{#2}}
\cs_new:Npn \box_move_down:nn #1#2{\tex_lower:D\dim_eval:n{#1}{#2}}
```

\box_clear:N
\box_clear:c
\box_gclear:N
\box_gclear:c

Clear a ⟨*box*⟩ register.

```
\cs_new_protected_nopar:Npn \box_clear:N #1{\box_set_eq:NN #1 \c_empty_box }
\cs_generate_variant:Nn \box_clear:N {c}
\cs_new_protected_nopar:Npn \box_gclear:N {\pref_global:D\box_clear:N}
\cs_generate_variant:Nn \box_gclear:N {c}
```

\box_ht:N
\box_ht:c
\box_dp:N
\box_dp:c
\box_wd:N
\box_wd:c

Accessing the height, depth, and width of a ⟨*box*⟩ register.

```
\cs_new_eq:NN \box_ht:N \tex_ht:D
\cs_new_eq:NN \box_dp:N \tex_dp:D
\cs_new_eq:NN \box_wd:N \tex_wd:D
\cs_generate_variant:Nn \box_ht:N {c}
\cs_generate_variant:Nn \box_dp:N {c}
\cs_generate_variant:Nn \box_wd:N {c}
```

\box_use_clear:N
\box_use_clear:c
\box_use:N
\box_use:c

Using a ⟨*box*⟩. These are just TeX primitives with meaningful names.

```
\cs_new_eq:NN \box_use_clear:N \tex_box:D
\cs_generate_variant:Nn \box_use_clear:N {c}
\cs_new_eq:NN \box_use:N \tex_copy:D
\cs_generate_variant:Nn \box_use:N {c}
```

\box_show:N
\box_show:c

Show the contents of a box and write it into the log file.

```
\cs_set_eq:NN \box_show:N \tex_showbox:D
\cs_generate_variant:Nn \box_show:N {c}
```

\c_empty_box
\l_tmpa_box
\l_tmpb_box

We allocate some ⟨*box*⟩ registers here (and borrow a few from LaTeX).

```
⟨package⟩\cs_set_eq:NN \c_empty_box \voidb@x
⟨package⟩\cs_new_eq:NN \l_tmpa_box \@tempboxa
⟨initex⟩\box_new:N \c_empty_box
⟨initex⟩\box_new:N \l_tmpa_box
\box_new:N \l_tmpb_box
```

114.2 Vertical boxes

\vbox:n

Put a vertical box directly into the input stream.

```
\cs_new_protected_nopar:Npn \vbox:n {\tex_vbox:D \scan_stop:}
```

\vbox_set:Nn
\vbox_set:cn
\vbox_gset:Nn
\vbox_gset:cn

Storing material in a vertical box with a natural height.

```
\cs_new_protected:Npn \vbox_set:Nn #1#2 {\tex_setbox:D #1 \tex_vbox:D {#2}}
\cs_generate_variant:Nn \vbox_set:Nn {cn}
\cs_new_protected_nopar:Npn \vbox_gset:Nn  {\pref_global:D \vbox_set:Nn}
\cs_generate_variant:Nn \vbox_gset:Nn {cn}
```

\vbox_set_to_ht:Nnn
\vbox_set_to_ht:cnn
\vbox_gset_to_ht:Nnn
\vbox_gset_to_ht:cnn
\vbox_gset_to_ht:ccn

Storing material in a vertical box with a specified height.

```
\cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3 {
  \tex_setbox:D #1 \tex_vbox:D to #2 {#3}
}
\cs_generate_variant:Nn \vbox_set_to_ht:Nnn {cnn}
\cs_new_protected_nopar:Npn \vbox_gset_to_ht:Nnn { \pref_global:D \vbox_set_to_ht:Nnn }
\cs_generate_variant:Nn \vbox_gset_to_ht:Nnn {cnn,ccn}
```

\vbox_set_inline_begin:N
\vbox_set_inline_end:
\vbox_gset_inline_begin:N
\vbox_gset_inline_end:

Storing material in a vertical box. This type is useful in environment definitions.

```
\cs_new_protected_nopar:Npn \vbox_set_inline_begin:N  #1 {
  \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }
\cs_new_eq:NN \vbox_set_inline_end: \c_group_end_token
\cs_new_protected_nopar:Npn \vbox_gset_inline_begin:N {
  \pref_global:D \vbox_set_inline_begin:N }
\cs_new_eq:NN \vbox_gset_inline_end: \c_group_end_token
```

\vbox_to_ht:nn
\vbox_to_zero:n

Put a vertical box directly into the input stream.

```
\cs_new_protected:Npn \vbox_to_ht:nn #1#2{\tex_vbox:D to \dim_eval:n{#1}{#2}}
\cs_new_protected:Npn \vbox_to_zero:n #1 {\tex_vbox:D to \c_zero_dim {#1}}
```

\vbox_set_split_to_ht:NNn

Splitting a vertical box in two.

```
\cs_new_protected_nopar:Npn \vbox_set_split_to_ht:NNn #1#2#3{
  \tex_setbox:D #1 \tex_vsplit:D #2 to #3
}
```

\vbox_unpack:N
\vbox_unpack:c
\vbox_unpack_clear:N
\vbox_unpack_clear:c

Unpacking a box and if requested also clear it.

```
\cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
\cs_generate_variant:Nn \vbox_unpack:N {c}
\cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D
\cs_generate_variant:Nn \vbox_unpack_clear:N {c}
```

114.3 Horizontal boxes

\hbox:n

Put a horizontal box directly into the input stream.

```
\cs_new_protected_nopar:Npn \hbox:n {\tex_hbox:D \scan_stop:}
```

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn

Assigning the contents of a box to be another box. This clears the second box globally (that's how TeX does it).

```
\cs_new_protected:Npn \hbox_set:Nn #1#2 {\tex_setbox:D #1 \tex_hbox:D {#2}}
\cs_generate_variant:Nn \hbox_set:Nn {cn}
\cs_new_protected_nopar:Npn \hbox_gset:Nn  {\pref_global:D \hbox_set:Nn}
\cs_generate_variant:Nn \hbox_gset:Nn {cn}
```

\hbox_set_to_wd:Nnn
\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn

Storing material in a horizontal box with a specified width.

```
\cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3 {
  \tex_setbox:D #1 \tex_hbox:D to \dim_eval:n{#2} {#3}
}
\cs_generate_variant:Nn \hbox_set_to_wd:Nnn {cnn}
\cs_new_protected_nopar:Npn \hbox_gset_to_wd:Nnn {\pref_global:D \hbox_set_to_wd:Nnn }
\cs_generate_variant:Nn \hbox_gset_to_wd:Nnn {cnn}
```

\hbox_set_inline_begin:N
\hbox_set_inline_begin:c
\hbox_set_inline_end:
\hbox_gset_inline_begin:N
\hbox_gset_inline_begin:c
\hbox_gset_inline_end:

Storing material in a horizontal box. This type is useful in environment definitions.

```
\cs_new_protected_nopar:Npn \hbox_set_inline_begin:N  #1 {
  \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token
}
\cs_generate_variant:Nn \hbox_set_inline_begin:N {c}
\cs_new_eq:NN \hbox_set_inline_end: \c_group_end_token
\cs_new_protected_nopar:Npn \hbox_gset_inline_begin:N {
  \pref_global:D \hbox_set_inline_begin:N
}
\cs_generate_variant:Nn \hbox_gset_inline_begin:N {c}
\cs_new_eq:NN \hbox_gset_inline_end: \c_group_end_token
```

\hbox_to_wd:nn
\hbox_to_zero:n

Put a horizontal box directly into the input stream.

```
\cs_new_protected:Npn \hbox_to_wd:nn #1#2 {\tex_hbox:D to #1 {#2}}
\cs_new_protected:Npn \hbox_to_zero:n #1 {\tex_hbox:D to \c_zero_skip {#1}}
```

`\hbox_overlap_left:n`
`\hbox_overlap_right:n`

Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

```
\cs_new_protected:Npn \hbox_overlap_left:n  #1 {\hbox_to_zero:n {\tex_hss:D #1}}
\cs_new_protected:Npn \hbox_overlap_right:n #1 {\hbox_to_zero:n {#1 \tex_hss:D}}
```

`\hbox_unpack:N`
`\hbox_unpack:c`
`\hbox_unpack_clear:N`
`\hbox_unpack_clear:c`

Unpacking a box and if requested also clear it.

```
\cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
\cs_generate_variant:Nn \hbox_unpack:N {c}
\cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
\cs_generate_variant:Nn \hbox_unpack_clear:N {c}
```

```
⟨/initex | package⟩
```

```
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

115 l3xref implementation

115.1 Internal functions and variables

`\g_xref_all_curr_immediate_fields_prop`
`\g_xref_all_curr_deferred_fields_prop`

What they say they are :)

`\xref_write`

A stream for writing cross references, although they are not required to be in a separate file.

`\xref_define_label:nn`

`\xref_define_label:nn {⟨name⟩} {⟨plist contents⟩}`

Define the property list for each label; used internally by `\xref_set_label:n`.

115.2 Module code

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

There are two kinds of information, namely information which is *immediate* like a section title and then there's *deferred* information like page numbers. Each reference type belong to one of these categories, which we save internally as the property lists `\g_xref_all_curr_immediate_fields_prop` and `\g_xref_all_curr_deferred_fields_prop` and the reference type ⟨*xyz*⟩ exists as the key-info pair `\xref_`⟨*xyz*⟩`_key {\l_xref_curr_`⟨*xyz*⟩`_tl}` on one of these lists. This way each new entry type is just added as another key-info pair.

When the cross references are generated at the beginning of the document each will turn into a control sequence. Thus `\label{mylab}` will internally refer to the property list `\g_xref_mylab_prop`.

The extraction of values from this property list can be done in several different ways but we want to keep the operation expandable. Therefore we use a dedicated function for each type of cross reference, which looks like this:

```
\xref_get_value_xyz_aux:w -> #1 \xref_xyz_key #2#3\q_nil{#2}
```

This will throw away all the bits we don't need. In case `xyz` is the first on the `mylab` property list `#1` is empty, if it's the last key-info pair `#3` is empty. The value of the field can be extracted with the function `\xref_get_value:nn` where the first argument is the type and the second the label name so here it would be `\xref_get_value:nn {xyz} {mylab}`.

`_all_curr_immediate_fields_prop`
`f_all_curr_deferred_fields_prop`

The two main property lists for storing information. They contain key-info pairs for all known types.

```
\prop_new:N \g_xref_all_curr_immediate_fields_prop
\prop_new:N \g_xref_all_curr_deferred_fields_prop
```

`\xref_new:nn`
`\xref_deferred_new:nn`
`\xref_new_aux:nnn`

Setting up a new cross reference type is fairly straight forward when we follow the game plan mentioned earlier.

```
\cs_new_nopar:Npn \xref_new:nn {\xref_new_aux:nnn{immediate}}
\cs_new_nopar:Npn \xref_deferred_new:nn {\xref_new_aux:nnn{deferred}}
\cs_new_nopar:Npn \xref_new_aux:nnn #1#2#3{
```

First put the new type in the relevant property list.

```
  \prop_gput:ccx {g_xref_all_curr_ #1 _fields_prop}
  { xref_ #2 _key }
  { \exp_not:c {l_xref_curr_#2_tl }}
```

Then define the key to be a protected macro.[13]

```
  \cs_set_protected_nopar:cpn { xref_#2_key }{}
  \tl_new:cn{l_xref_curr_#2_tl}{#3}
```

[13]We could also set it equal to `\scan_stop:` but this just feels "cleaner".

Now for the function extracting the value of a reference. We could do this with a simple `\prop_if_in` thing put since we want to do things in an expandable way we make a separate grabber for each type—this is also faster. The grabber function can be defined by using an intricate construction of `\exp_after:wN` and other goodies but I prefer readable code. The end result for the input `xyz` is

```
\cs_set_nopar:Npn\xref_get_value_xyz_aux:w #1\xref_xyz_key #2#3\q_nil{#2}
```

```
  \toks_set:Nx \l_tmpa_toks {
    \exp_not:n { \cs_set_nopar:cpn {xref_get_value_#2_aux:w} ##1 }
    \exp_not:N \q_prop
    \exp_not:c { xref_#2_key }
    \exp_not:N \q_prop
  }
  \toks_use:N \l_tmpa_toks ##2 ##3\q_nil {##2}
}
```

`\xref_get_value:nn` Getting the correct value for a given label-type pair is a matter of connecting the correct grabber functions and property list.

```
\cs_new_nopar:Npn \xref_get_value:nn #1#2 {
  \cs_if_exist:cTF{g_xref_#2_prop}
  {
```

This next expansion may look a little weird but it isn't if you think about it!

```
    \exp_args:NcNc \exp_after:wN {xref_get_value_#1_aux:w}
    \toks_use:N {g_xref_#2_prop}
```

Better put in the stop marker.

```
    \q_nil
  }
  {??}
}
```

Temporary! We expand the property list and so we can't have the `\q_prop` marker just expand!

```
\cs_set_nopar:Npn \exp_after:cc #1#2 {
  \exp_after:wN \exp_after:wN
  \cs:w #1\exp_after:wN\cs_end: \cs:w #2\cs_end:
}
\cs_set_protected:Npn \q_prop {\q_prop}
```

`\xref_define_label:nn`
`\xref_define_label_aux:nn` Define the property list for each label. We better do this in two steps because the special catcode regime is in effect and since some of the info fields are very likely to contain actual text, we better make sure spaces aren't ignored! As for the meaning of other characters

then it is a possibility to also have a field containing catcode instructions which can then be activated with \etex_scantokens:D.

```
\cs_new_protected_nopar:Npn \xref_define_label:nn {
  \group_begin:
    \char_set_catcode:nn {`\ }\c_ten
    \xref_define_label_aux:nn
}
```

If the label is already taken we have a multiply defined label and we should do something about it. For now we don't do anything spectacular.

```
\cs_new_nopar:Npn \xref_define_label_aux:nn #1#2 {
    \cs_if_free:cTF{g_xref_#1_prop}
    {\prop_new:c{g_xref_#1_prop}}{\WARNING}
    \toks_gset:cn{g_xref_#1_prop}{#2}
  \group_end:
}
```

\xref_set_label:n Then the generic command for setting a label. We expand the immediate labels fully before calling the write function but make sure the deferred fields aren't expanded just yet. Due to property lists being implemented as token list registers we must expand the 'immediate' fields twice.

```
\cs_set_nopar:Npn \xref_set_label:n #1{
  \cs_set_nopar:Npx \xref_tmp:w{\toks_use:N\g_xref_all_curr_immediate_fields_prop}
  \exp_args:NNx\iow_shipout_x:Nn \xref_write{
    \xref_define_label:nn {#1} {
      \xref_tmp:w
      \toks_use:N \g_xref_all_curr_deferred_fields_prop
    }
  }
}
```

That's it (for now).

```
⟨/initex | package⟩
```

```
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

116 l3xref test file

```
⟨*testfile⟩
\documentclass{article}
\usepackage{l3xref}
\ExplSyntaxOn
```

```
\cs_set_nopar:Npn \startrecording {\iow_open:Nn \xref_write {\jobname.xref}}
\cs_set_nopar:Npn \DefineCrossReferences {
  \group_begin:
    \ExplSyntaxNamesOn
    \InputIfFileExists{\jobname.xref}{}{}
  \group_end:
}
\AtBeginDocument{\DefineCrossReferences\startrecording}

\xref_new:nn {name}{}
\cs_set_nopar:Npn \setname{\tl_set:Nn\l_xref_curr_name_tl}
\cs_set_nopar:Npn \getname{\xref_get_value:nn{name}}

\xref_deferred_new:nn {page}{\thepage}
\cs_set_nopar:Npn \getpage{\xref_get_value:nn{page}}

\xref_deferred_new:nn {valuepage}{\number\value{page}}
\cs_set_nopar:Npn \getvaluepage{\xref_get_value:nn{valuepage}}

\cs_set_eq:NN \setlabel \xref_set_label:n

\ExplSyntaxOff
\begin{document}
\pagenumbering{roman}

Text\setname{This is a name}\setlabel{testlabel1}.  More
text\setname{This is another name}\setlabel{testlabel2}. \clearpage

Text\setname{This is a third name}\setlabel{testlabel3}. More
text\setname{Hello World!}\setlabel{testlabel4}. \clearpage

\pagenumbering{arabic}

Text\setname{Name 5}\setlabel{testlabel5}. More text\setname{Name
  6}\setlabel{testlabel6}. \clearpage

Text\setname{Name 7}\setlabel{testlabel 7}. More text\setname{Name
  8}\setlabel{testlabel8}. \clearpage

Now let's extract some values. \getname{testlabel1} on page
\getpage{testlabel1} with value \getvaluepage{testlabel1}.

Now let's extract some values. \getname{testlabel 7} on page
\getpage{testlabel 7} with value \getvaluepage{testlabel 7}.
\end{document}
⟨/testfile⟩
```

117 l3keyval implementation

\KV_sanitize_outerlevel_active_equals:N \KV_sanitize_outerlevel_active_commas:N

`\KV_sanitize_outerlevel_active_equals:N` ⟨*tl var.*⟩

Replaces catcode other = and , within a ⟨*tl var.*⟩ with active characters.

\KV_remove_surrounding_spaces:nw \KV_remove_surrounding_spaces_auxi:w ⋆

`\KV_remove_surrounding_spaces:nw` ⟨*tl*⟩ ⟨*token list*⟩ `\q_ni`
`\KV_remove_surrounding_spaces_auxi:w` ⟨*token list*⟩ ␣Q3

Removes a possible leading space plus a possible ending space from a ⟨*token list*⟩. The first version (which is not used in the code) stores it in ⟨*tl*⟩.

\KV_add_value_element:w \KV_set_key_element:w

`\KV_set_key_element:w` ⟨*token list*⟩ `\q_nil`
`\KV_add_value_element:w \q_stop` ⟨*token list*⟩ `\q_nil`

Specialised functions to strip spaces from their input and set the token registers `\l_KV_currkey_tl` or `\l_KV_currval_tl` respectively.

\KV_split_key_value_current:w \KV_split_key_value_space_removal:w \KV_split_key_value_space_removal_detect_error:wTF \KV_split_key_value_no_space_removal:w

`\KV_split_key_value_current:w ...`

These functions split keyval lists into chunks depending which sanitising method is being used. `\KV_split_key_value_current:w` is `\cs_set_eq:NN` to whichever is appropriate.

117.1 Module code

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

`\l_KV_tmpa_tl`
`\l_KV_tmpb_tl`
`\c_KV_single_equal_sign_tl`

Various useful things.

```
\tl_new:N  \l_KV_tmpa_tl
\tl_new:N  \l_KV_tmpb_tl
\tl_const:Nn \c_KV_single_equal_sign_tl { = }
```

\l_KV_parse_tl
\l_KV_currkey_tl
\l_KV_currval_tl

Some more useful things.

```
\tl_new:N \l_KV_parse_tl
\tl_new:N \l_KV_currkey_tl
\tl_new:N \l_KV_currval_tl
```

\l_KV_level_int

This is used to track how deeply nested calls to the keyval processor are, so that the correct functions are always in use.

```
\int_new:N \l_KV_level_int
```

remove_one_level_of_braces_bool

A boolean to control

```
\bool_new:N \l_KV_remove_one_level_of_braces_bool
\bool_set_true:N \l_KV_remove_one_level_of_braces_bool
```

cess_space_removal_sanitize:NNn
s_space_removal_no_sanitize:NNn
o_space_removal_no_sanitize:NNn
\KV_process_aux:NNNn

The wrapper function takes care of assigning the appropriate `elt` functions before and after the parsing step. In that way there is no danger of a mistake with the wrong functions being used.

```
\cs_new_protected_nopar:Npn \KV_process_space_removal_sanitize:NNn {
  \KV_process_aux:NNNn \KV_parse_space_removal_sanitize:n
}
\cs_new_protected_nopar:Npn \KV_process_space_removal_no_sanitize:NNn {
  \KV_process_aux:NNNn \KV_parse_space_removal_no_sanitize:n
}
\cs_new_protected_nopar:Npn \KV_process_no_space_removal_no_sanitize:NNn {
  \KV_process_aux:NNNn \KV_parse_no_space_removal_no_sanitize:n
}
\cs_new_protected:Npn \KV_process_aux:NNNn #1#2#3#4 {
  \cs_set_eq:cN
    { KV_key_no_value_elt_ \int_use:N \l_KV_level_int :n }
    \KV_key_no_value_elt:n
  \cs_set_eq:cN
    { KV_key_value_elt_ \int_use:N \l_KV_level_int :nn }
    \KV_key_value_elt:nn
  \cs_set_eq:NN \KV_key_no_value_elt:n #2
  \cs_set_eq:NN \KV_key_value_elt:nn #3
  \int_incr:N \l_KV_level_int
  #1 {#4}
  \int_decr:N \l_KV_level_int
  \cs_set_eq:Nc \KV_key_no_value_elt:n
    { KV_key_no_value_elt_ \int_use:N \l_KV_level_int :n }
  \cs_set_eq:Nc \KV_key_value_elt:nn
    { KV_key_value_elt_ \int_use:N \l_KV_level_int :nn }
}
```

tize_outerlevel_active_equals:N
tize_outerlevel_active_commas:N

Some functions for sanitizing top level equals and commas. Replace $=_{13}$ and $,_{13}$ with $=_{12}$ and $,_{12}$ resp.

```
\group_begin:
\char_set_catcode:nn{`\=}{13}
\char_set_catcode:nn{`\,}{13}
\char_set_lccode:nn{`\8}{`\=}
\char_set_lccode:nn{`\9}{`\,}
\tl_to_lowercase:n{\group_end:
\cs_new_protected_nopar:Npn \KV_sanitize_outerlevel_active_equals:N #1{
  \tl_replace_all_in:Nnn #1 = 8
}
\cs_new_nopar:Npn \KV_sanitize_outerlevel_active_commas:N #1{
  \tl_replace_all_in:Nnn #1 , 9
}
}
```

KV_remove_surrounding_spaces:nw
emove_surrounding_spaces_auxi:w
move_surrounding_spaces_auxii:w
\KV_set_key_element:w
\KV_add_value_element:w

The macro `\KV_remove_surrounding_spaces:nw` removes a possible leading space plus a possible ending space from its second argument and stores it in the token register `#1`.

Based on Around the Bend No. 15 but with some enhancements. For instance, this definition is purely expandable.

We use a funny token $\mathtt{Q}_3$ as a delimiter.

```
\group_begin:
\char_set_catcode:nn{`\Q}{3}
```

```
\cs_gnew:Npn\KV_remove_surrounding_spaces:nw#1#2\q_nil{
```

The idea in this processing is to use a Q with strange catcode to remove a trailing space. But first, how to get this expansion going?

If you have read the fine print in the l3expan module, you'll know that the `f` type expansion will expand until the first non-expandable token is seen and if this token is a space, it will be gobbled. Sounds useful for removing a leading space but we also need to make sure that it does nothing but removing that space! Therefore we prepend the argument to be trimmed with an `\exp_not:N`. Now why is that? `\exp_not:N` in itself is an expandable command so will allow the `f` expansion to continue. If the first token in the argument to be trimmed is a space, it will be gobbled and the expansion stop. If the first token isn't a space, the `\exp_not:N` turns it temporarily into `\scan_stop:` which is unexpandable. The processing stops but the token following directly after `\exp_not:N` is now back to normal.

The function here allows you to insert arbitrary functions in the first argument but they should all be with an `f` type expansion. For the application in this module, we use `\tl_set:Nf`.

Once the expansion has been kick-started, we apply `\KV_remove_surrounding_spaces_auxi:w` to the replacement text of `#2`, adding a leading `\exp_not:N`. Note that no braces are stripped off of the original argument.

```
  #1{\KV_remove_surrounding_spaces_auxi:w \exp_not:N#2Q~Q}
}
```

\KV_remove_surrounding_spaces_auxi:w removes a trailing space if present, then calls \KV_remove_surrounding_spaces_auxii:w to clean up any leftover bizarre Qs. In order for \KV_remove_surrounding_spaces_auxii:w to work properly we need to put back a Q first.

```
\cs_gnew:Npn\KV_remove_surrounding_spaces_auxi:w#1~Q{
  \KV_remove_surrounding_spaces_auxii:w #1 Q
}
```

Now all that is left to do is remove a leading space which should be taken care of by the function used to initiate the expansion. Simply return the argument before the funny Q.

```
\cs_gnew:Npn\KV_remove_surrounding_spaces_auxii:w#1Q#2{#1}
```

Here are some specialized versions of the above. They do exactly what we want in one go. First trim spaces from the value and then put the result surrounded in braces onto \l_KV_parse_tl.

```
\cs_gnew_protected:Npn\KV_add_value_element:w\q_stop#1\q_nil{
  \tl_set:Nf\l_KV_currval_tl {
    \KV_remove_surrounding_spaces_auxi:w \exp_not:N#1Q~Q
  }
  \tl_put_right:No\l_KV_parse_tl{
    \exp_after:wN { \l_KV_currval_tl }
  }
}
```

When storing the key we firstly remove spaces plus the prepended \q_no_value.

```
\cs_gnew_protected:Npn\KV_set_key_element:w#1\q_nil{
  \tl_set:Nf\l_KV_currkey_tl
  {
    \exp_last_unbraced:NNo \KV_remove_surrounding_spaces_auxi:w
      \exp_not:N \use_none:n #1Q~Q
  }
```

Afterwards we gobble an extra level of braces if that's what we are asked to do.

```
  \bool_if:NT \l_KV_remove_one_level_of_braces_bool
  {
    \exp_args:NNo \tl_set:No \l_KV_currkey_tl {
      \exp_after:wN \KV_add_element_aux:w  \l_KV_currkey_tl \q_nil
    }
  }
}
\group_end:
```

\KV_add_element_aux:w A helper function for fixing braces around keys and values.

```
\cs_new:Npn \KV_add_element_aux:w#1\q_nil{#1}
```

Parse a list of keyvals, put them into list form with entries like `\KV_key_no_value_elt:n{key1}` and `\KV_key_value_elt:nn{key2}{val2}`.

`\KV_parse_sanitize_aux:n` The slow parsing algorithm sanitizes active commas and equal signs at the top level first. Then uses `#1` as inspector of each element in the comma list.

```
\cs_new_protected:Npn \KV_parse_sanitize_aux:n #1 {
  \group_begin:
    \tl_clear:N \l_KV_parse_tl
    \tl_set:Nn \l_KV_tmpa_tl {#1}
    \KV_sanitize_outerlevel_active_equals:N \l_KV_tmpa_tl
    \KV_sanitize_outerlevel_active_commas:N \l_KV_tmpa_tl
    \exp_last_unbraced:NNV \KV_parse_elt:w \q_no_value
      \l_KV_tmpa_tl , \q_nil ,
```

We evaluate the parsed keys and values outside the group so the token register is restored to its previous value.

```
  \exp_after:wN \group_end:
  \l_KV_parse_tl
}
```

`\KV_parse_no_sanitize_aux:n` Like above but we don't waste time sanitizing. This is probably the one we will use for preamble parsing where catcodes of `=` and `,` are as expected!

```
\cs_new_protected:Npn \KV_parse_no_sanitize_aux:n #1{
  \group_begin:
    \tl_clear:N \l_KV_parse_tl
    \KV_parse_elt:w \q_no_value #1 , \q_nil ,
  \exp_after:wN \group_end:
  \l_KV_parse_tl
}
```

`\KV_parse_elt:w` This function will always have a `\q_no_value` stuffed in as the rightmost token in `#1`. In case there was a blank entry in the comma separated list we just run it again. The `\use_none:n` makes sure to gobble the quark `\q_no_value`. A similar test is made to check if we hit the end of the recursion.

```
\cs_set:Npn \KV_parse_elt:w #1,{
  \tl_if_blank:oTF{\use_none:n #1}
  { \KV_parse_elt:w \q_no_value }
  {
    \quark_if_nil:oF {\use_ii:nn #1 }
```

If we made it to here we can start parsing the key and value. When done try, try again.

```
    {
      \KV_split_key_value_current:w #1==\q_nil
      \KV_parse_elt:w \q_no_value
```

```
    }
  }
}
```

\KV_split_key_value_current:w

The function called to split the keys and values.

```
\cs_new:Npn \KV_split_key_value_current:w {\ERROR}
```

We provide two functions for splitting keys and values. The reason being that most of the time, we should probably be in the special coding regime where spaces are ignored. Hence it makes no sense to spend time searching for extra space tokens and we can do the settings directly. When comparing these two versions (neither doing any sanitizing) the `no_space_removal` version is more than 40% faster than `space_removal`.

It is up to functions like `\DeclareTemplate` to check which catcode regime is active and then pick up the version best suited for it.

split_key_value_space_removal:w
_space_removal_detect_error:wTF
c_key_value_space_removal_aux:w

The code below removes extraneous spaces around the keys and values plus one set of braces around the entire value.

Unlike the version to be used when spaces are ignored, this one only grabs the key which is everything up to the first = and save the rest for closer inspection. Reason is that if a user has entered `mykey={{myval}},` then the outer braces have already been removed before we even look at what might come after the key. So this is slightly more tedious (but only slightly) but at least it always removes only one level of braces.

```
\cs_new_protected:Npn \KV_split_key_value_space_removal:w #1 = #2\q_nil{
```

First grab the key.

```
  \KV_set_key_element:w#1\q_nil
```

Then we start checking. If only a key was entered, `#2` contains `=` and nothing else, so we test for that first.

```
  \tl_set:Nn\l_KV_tmpa_tl{#2}
  \tl_if_eq:NNTF\l_KV_tmpa_tl\c_KV_single_equal_sign_tl
```

Then we just insert the default key.

```
  {
    \tl_put_right:No\l_KV_parse_tl{
      \exp_after:wN \KV_key_no_value_elt:n
      \exp_after:wN {\l_KV_currkey_tl}
    }
  }
```

Otherwise we must take a closer look at what is left. The remainder of the original list up to the comma is now stored in `#2` plus an additional `==`, which wasn't gobbled during the initial reading of arguments. If there is an error then we can see at least one more = so we call an auxiliary function to check for this.

```
  {
    \KV_split_key_value_space_removal_detect_error:wTF#2\q_no_value\q_nil
    {\KV_split_key_value_space_removal_aux:w \q_stop #2}
    { \msg_kernel_error:nn { keyval } { misplaced-equals-sign } }
  }
}
```

The error test.

```
\cs_new_protected:Npn
  \KV_split_key_value_space_removal_detect_error:wTF#1=#2#3\q_nil{
    \tl_if_head_eq_meaning:nNTF{#3}\q_no_value
}
```

Now we can start extracting the value. Recall that #1 here starts with \q_stop so all braces are still there! First we try to see how much is left if we gobble three brace groups from #1. If #1 is empty or blank, all three quarks are gobbled. If #1 consists of exactly one token or brace group, only the latter quark is left.

```
\cs_new:Npn \KV_val_preserve_braces:NnN #1#2#3{{#2}}
\cs_new_protected:Npn\KV_split_key_value_space_removal_aux:w #1=={
  \tl_set:Nx\l_KV_tmpa_tl{\exp_not:o{\use_none:nnn#1\q_nil\q_nil}}
  \tl_put_right:No\l_KV_parse_tl{
    \exp_after:wN \KV_key_value_elt:nn
    \exp_after:wN {\l_KV_currkey_tl}
  }
```

If there a blank space or nothing at all, \l_KV_tmpa_tl is now completely empty.

```
    \tl_if_empty:NTF\l_KV_tmpa_tl
```

We just put an empty value on the stack.

```
  { \tl_put_right:Nn\l_KV_parse_tl{{}} }
  {
```

If there was exactly one brace group or token in #1, \l_KV_tmpa_tl is now equal to \q_nil. Then we can just pick it up as the second argument of #1. This will also take care of any spaces which might surround it.

```
    \quark_if_nil:NTF\l_KV_tmpa_tl
    {
      \bool_if:NTF \l_KV_remove_one_level_of_braces_bool
      {
        \tl_put_right:No\l_KV_parse_tl{
          \exp_after:wN{\use_ii:nnn #1\q_nil}
        }
      }
```

```
      {
        \tl_put_right:No\l_KV_parse_tl{
          \exp_after:wN{\KV_val_preserve_braces:NnN #1\q_nil}
        }
      }
    }
```

Otherwise we grab the value.

```
    { \KV_add_value_element:w #1\q_nil }
  }
}
```

it_key_value_no_space_removal:w

This version is for when in the special coding regime where spaces are ignored so there is no need to do any fancy space hacks, however fun they may be. Since there are no spaces, a set of braces around a value is automatically stripped by TeX.

```
\cs_new_protected:Npn \KV_split_key_value_no_space_removal:w #1#2=#3=#4\q_nil{
  \tl_set:Nn\l_KV_tmpa_tl{#4}
  \tl_if_empty:NTF \l_KV_tmpa_tl
  {
    \tl_put_right:Nn\l_KV_parse_tl{\KV_key_no_value_elt:n{#2}}
  }
  {
    \tl_if_eq:NNTF\c_KV_single_equal_sign_tl\l_KV_tmpa_tl
    {
      \tl_put_right:Nn\l_KV_parse_tl{\KV_key_value_elt:nn{#2}{#3}}
    }
    { \msg_kernel_error:nn { keyval } { misplaced-equals-sign } }
  }
}
```

\KV_key_no_value_elt:n
\KV_key_value_elt:nn

```
\cs_new:Npn \KV_key_no_value_elt:n #1{\ERROR}
\cs_new:Npn \KV_key_value_elt:nn #1#2{\ERROR}
```

_no_space_removal_no_sanitize:n

Finally we can put all the things together. `\KV_parse_no_space_removal_no_sanitize:n` is the version that disallows unmatched conditional and does no space removal.

```
\cs_new_protected_nopar:Npn \KV_parse_no_space_removal_no_sanitize:n {
  \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_no_space_removal:w
  \KV_parse_no_sanitize_aux:n
}
```

_parse_space_removal_sanitize:n
rse_space_removal_no_sanitize:n

The other varieties can be defined in a similar manner. For the version needed at the document level, we can use this one.

```
\cs_new_protected_nopar:Npn \KV_parse_space_removal_sanitize:n {
```

```
  \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_space_removal:w
  \KV_parse_sanitize_aux:n
}
```

For preamble use by the non-programmer this is probably best.

```
\cs_new_protected_nopar:Npn \KV_parse_space_removal_no_sanitize:n {
  \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_space_removal:w
  \KV_parse_no_sanitize_aux:n
}
```

```
\msg_kernel_new:nnnn { keyval } { misplaced-equals-sign }
  {Misplaced~equals~sign~in~key--value~input~\msg_line_context:}
  {
    I~am~trying~to~read~some~key--value~input~but~found~two~equals~
    signs\\%
    without~a~comma~between~them.
  }
```

```
⟨/initex | package⟩
```

```
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

The usual preliminaries.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

117.1.1 Variables and constants

\c_keys_root_tl
\c_keys_properties_root_tl

Where the keys are really stored.

```
\tl_const:Nn \c_keys_root_tl { keys~>~ }
\tl_const:Nn \c_keys_properties_root_tl { keys_properties }
```

\c_keys_value_forbidden_tl
\c_keys_value_required_tl

Two marker token lists.

```
\tl_const:Nn \c_keys_value_forbidden_tl { forbidden }
\tl_const:Nn \c_keys_value_required_tl { required }
```

\l_keys_choice_int
\l_keys_choice_tl

Used for the multiple choice system.

```
\int_new:N \l_keys_choice_int
\tl_new:N \l_keys_choice_tl
```

`\l_keys_choice_code_tl` When creating multiple choices, the code is stored here.

```
\tl_new:N \l_keys_choice_code_tl
```

`\l_keys_key_tl`
`\l_keys_path_tl`
`\l_keys_property_tl`

Storage for the current key name and the path of the key (key name plus module name).

```
\tl_new:N \l_keys_key_tl
\tl_new:N \l_keys_path_tl
\tl_new:N \l_keys_property_tl
```

`\l_keys_module_tl` The module for an entire set of keys.

```
\tl_new:N \l_keys_module_tl
```

`\l_keys_no_value_bool` To indicate that no value has been given.

```
\bool_new:N \l_keys_no_value_bool
```

`\l_keys_value_tl` A token variable for the given value.

```
\tl_new:N \l_keys_value_tl
```

117.1.2 Internal functions

`\keys_bool_set:NN` Boolean keys are really just choices, but all done by hand.

```
\cs_new_protected_nopar:Npn \keys_bool_set:NN #1#2 {
  \keys_cmd_set:nx { \l_keys_path_tl / true } {
    \exp_not:c { bool_ #2 set_true:N }
      \exp_not:N #1
  }
  \keys_cmd_set:nx { \l_keys_path_tl / false } {
    \exp_not:N \use:c
      { bool_ #2 set_false:N }
      \exp_not:N #1
  }
  \keys_choice_make:
  \cs_if_exist:NF #1 {
    \bool_new:N #1
  }
  \keys_default_set:n { true }
}
```

`\keys_choice_code_store:x` The code for making multiple choices is stored in a token list as there should not be any # tokens.

```
\cs_new_protected:Npn \keys_choice_code_store:x #1 {
  \tl_set:cx { \c_keys_root_tl \l_keys_path_tl .choice_code_tl } {#1}
}
```

\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.

```
\cs_new_protected_nopar:Npn \keys_choice_find:n #1 {
  \keys_execute_aux:nn { \l_keys_path_tl / #1 } {
    \keys_execute_aux:nn { \l_keys_path_tl / unknown } { }
  }
}
```

\keys_choice_make: To make a choice from a key, two steps: set the code, and set the unknown key.

```
\cs_new_protected_nopar:Npn \keys_choice_make: {
  \keys_cmd_set:nn { \l_keys_path_tl } {
    \keys_choice_find:n {##1}
  }
  \keys_cmd_set:nn { \l_keys_path_tl / unknown } {
    \msg_kernel_error:nnxx { keys } { choice-unknown }
      { \l_keys_path_tl } {##1}
  }
}
```

\keys_choices_generate:n
\keys_choices_generate_aux:n

Creating multiple-choices means setting up the "indicator" code, then applying whatever the user wanted.

```
\cs_new_protected:Npn \keys_choices_generate:n #1 {
  \keys_choice_make:
  \int_zero:N \l_keys_choice_int
  \cs_if_exist:cTF {
    \c_keys_root_tl \l_keys_path_tl .choice_code_tl
  } {
    \tl_set:Nv \l_keys_choice_code_tl {
      \c_keys_root_tl \l_keys_path_tl .choice_code_tl
    }
  }{
    \msg_kernel_error:nnx { keys } { generate-choices-before-code }
      { \l_keys_path_tl }
  }
  \clist_map_function:nN {#1} \keys_choices_generate_aux:n
}
\cs_new_protected_nopar:Npn \keys_choices_generate_aux:n #1 {
  \keys_cmd_set:nx { \l_keys_path_tl / #1 } {
    \exp_not:n { \tl_set:Nn \l_keys_choice_tl } {#1}
    \exp_not:n { \int_set:Nn \l_keys_choice_int }
      { \int_use:N \l_keys_choice_int }
    \exp_not:V \l_keys_choice_code_tl
  }
  \int_incr:N \l_keys_choice_int
}
```

\keys_cmd_set:nn
\keys_cmd_set:nx
\keys_cmd_set_aux:n

Creating a new command means setting properties and then creating a function with the correct number of arguments.

```
\cs_new_protected:Npn \keys_cmd_set:nn #1#2 {
  \keys_cmd_set_aux:n {#1}
  \cs_generate_from_arg_count:cNnn { \c_keys_root_tl #1 .cmd:n }
    \cs_set:Npn 1 {#2}
}
\cs_new_protected:Npn \keys_cmd_set:nx #1#2 {
  \keys_cmd_set_aux:n {#1}
  \cs_generate_from_arg_count:cNnn { \c_keys_root_tl #1 .cmd:n }
    \cs_set:Npx 1 {#2}
}
\cs_new_protected_nopar:Npn \keys_cmd_set_aux:n #1 {
  \keys_property_undefine:n { #1 .default_tl }
  \cs_if_free:cT { \c_keys_root_tl #1 .req_tl }
    { \tl_new:c { \c_keys_root_tl #1 .req_tl } }
  \tl_clear:c { \c_keys_root_tl #1 .req_tl }
}
```

\keys_default_set:n
\keys_default_set:V

Setting a default value is easy.

```
\cs_new_protected:Npn \keys_default_set:n #1 {
  \cs_if_free:cT { \c_keys_root_tl \l_keys_path_tl .default_tl }
    { \tl_new:c { \c_keys_root_tl \l_keys_path_tl .default_tl } }
  \tl_set:cn { \c_keys_root_tl \l_keys_path_tl .default_tl } {#1}
}
\cs_generate_variant:Nn \keys_default_set:n { V }
```

\keys_define:nn

The main key-defining function mainly sets up things for l3keyval to use.

```
\cs_new_protected:Npn \keys_define:nn #1#2 {
  \tl_set:Nn \l_keys_module_tl {#1}
  \KV_process_no_space_removal_no_sanitize:NNn
    \keys_define_elt:n \keys_define_elt:nn {#2}
}
```

\keys_define_elt:n
\keys_define_elt:nn

The element processors for defining keys.

```
\cs_new_protected_nopar:Npn \keys_define_elt:n #1 {
  \bool_set_true:N \l_keys_no_value_bool
  \keys_define_elt_aux:nn {#1} { }
}
\cs_new_protected:Npn \keys_define_elt:nn #1#2 {
  \bool_set_false:N \l_keys_no_value_bool
  \keys_define_elt_aux:nn {#1} {#2}
}
```

\keys_define_elt_aux:nn

The auxiliary function does most of the work.

```
\cs_new_protected:Npn \keys_define_elt_aux:nn #1#2 {
  \keys_property_find:n {#1}
  \cs_set_eq:Nc \keys_tmp:w
    { \c_keys_properties_root_tl \l_keys_property_tl }
  \cs_if_exist:NTF \keys_tmp:w {
    \keys_define_key:n {#2}
  }{
    \msg_kernel_error:nnxx { keys } { property-unknown }
      { \l_keys_property_tl } { \l_keys_path_tl }
  }
}
```

`\keys_define_key:n`

Defining a new key means finding the code for the appropriate property then running it. As properties have signatures, a check can be made for required values without needing anything set explicitly.

```
\cs_new_protected:Npn \keys_define_key:n #1 {
  \bool_if:NTF \l_keys_no_value_bool {
    \intexpr_compare:nTF {
      \exp_args:Nc \cs_get_arg_count_from_signature:N
        { \l_keys_property_tl } = \c_zero
    } {
      \keys_tmp:w
    }{
      \msg_kernel_error:nnxx { key } { property-requires-value }
        { \l_keys_property_tl } { \l_keys_path_tl }
    }
  }{
    \keys_tmp:w {#1}
  }
}
```

`\keys_execute:`
`\keys_execute_unknown:`
`\keys_execute_aux:nn`

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!

```
\cs_new_protected_nopar:Npn \keys_execute: {
  \keys_execute_aux:nn { \l_keys_path_tl } {
    \keys_execute_unknown:
  }
}
\cs_new_protected_nopar:Npn \keys_execute_unknown: {
  \keys_execute_aux:nn { \l_keys_module_tl / unknown } {
    \msg_kernel_error:nnxx { keys } { key-unknown } { \l_keys_path_tl }
      { \l_keys_module_tl }
  }
}
```

If there is only one argument required, it is wrapped in braces so that everything is passed through properly. On the other hand, if more than one is needed it is down to the user to have put things in correctly! The use of `\q_keys_stop` here means that arguments do not run away (hence the nine empty groups), but that the module can clean up the spare groups at the end of executing the key.

```
\cs_new_protected_nopar:Npn \keys_execute_aux:nn #1#2 {
  \cs_set_eq:Nc \keys_tmp:w { \c_keys_root_tl #1 .cmd:n }
  \cs_if_exist:NTF \keys_tmp:w {
    \exp_args:NV \keys_tmp:w \l_keys_value_tl
  }{
    #2
  }
}
```

`\keys_if_exist:nnTF` A check for the existance of a key. This works by looking for the command function for the key (which ends `.cmd:n`).

```
\prg_set_conditional:Nnn \keys_if_exist:nn {TF,T,F} {
  \cs_if_exist:cTF { \c_keys_root_tl #1 / #2 .cmd:n } {
    \prg_return_true:
  }{
    \prg_return_false:
  }
}
```

`\keys_if_value_requirement:nTF` To test if a value is required or forbidden. Only one version is needed, so done by hand.

```
\cs_new_nopar:Npn \keys_if_value_requirement:nTF #1 {
  \tl_if_eq:ccTF { c_keys_value_ #1 _tl } {
    \c_keys_root_tl \l_keys_path_tl .req_tl
  }
}
```

`\keys_meta_make:n`
`\keys_meta_make:x`

To create a met-key, simply set up to pass data through.

```
\cs_new_protected_nopar:Npn \keys_meta_make:n #1 {
  \exp_last_unbraced:NNo \keys_cmd_set:nn \l_keys_path_tl
    \exp_after:wN { \exp_after:wN \keys_set:nn  \exp_after:wN { \l_keys_module_tl } {#1} }
}
\cs_new_protected_nopar:Npn \keys_meta_make:x #1 {
  \keys_cmd_set:nx { \l_keys_path_tl } {
    \exp_not:N \keys_set:nn { \l_keys_module_tl } {#1}
  }
}
```

`\keys_property_find:n`
`\keys_property_find_aux:n`
`\keys_property_find_aux:w`

Searching for a property means finding the last "." in the input, and storing the text before and after it.

```
\cs_new_protected_nopar:Npn \keys_property_find:n #1 {
  \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / }
```

```
  \tl_if_in:nnTF {#1} {.} {
    \keys_property_find_aux:n {#1}
  }{
    \msg_kernel_error:nnx { keys } { key-no-property } {#1}
  }
}
\cs_new_protected_nopar:Npn \keys_property_find_aux:n #1 {
  \keys_property_find_aux:w #1 \q_stop
}
\cs_new_protected_nopar:Npn \keys_property_find_aux:w #1 . #2 \q_stop {
  \tl_if_in:nnTF {#2} { . } {
    \tl_set:Nx \l_keys_path_tl {
      \l_keys_path_tl \tl_to_str:n {#1} .
    }
    \keys_property_find_aux:w #2 \q_stop
  }{
    \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl \tl_to_str:n {#1} }
    \tl_set:Nn \l_keys_property_tl { . #2 }
  }
}
```

`\keys_property_new:nn`
`\keys_property_new_arg:nn`

Creating a new property is simply a case of making the correctly-named function.

```
\cs_new_nopar:Npn \keys_property_new:nn #1#2 {
  \cs_new:cpn { \c_keys_properties_root_tl #1 } {#2}
}
\cs_new_protected_nopar:Npn \keys_property_new_arg:nn #1#2 {
  \cs_new:cpn { \c_keys_properties_root_tl #1 } ##1 {#2}
}
```

`\keys_property_undefine:n`

Removing a property means undefining it.

```
\cs_new_protected_nopar:Npn \keys_property_undefine:n #1 {
  \cs_set_eq:cN { \c_keys_root_tl #1 } \c_undefined
}
```

`\keys_set:nn`
`\keys_set:nV`
`\keys_set:nv`

The main setting function just does the set up to get l3keyval to do the hard work.

```
\cs_new_protected:Npn \keys_set:nn #1#2 {
  \tl_set:Nn \l_keys_module_tl {#1}
  \KV_process_space_removal_sanitize:NNn
    \keys_set_elt:n \keys_set_elt:nn {#2}
}
\cs_generate_variant:Nn \keys_set:nn { nV, nv }
```

`\keys_set_elt:n`
`\keys_set_elt:nn`

The two element processors are almost identical, and pass the data through to the underlying auxiliary, which does the work.

```
\cs_new_protected_nopar:Npn \keys_set_elt:n #1 {
```

```
  \bool_set_true:N \l_keys_no_value_bool
  \keys_set_elt_aux:nn {#1} { }
}
\cs_new_protected:Npn \keys_set_elt:nn #1#2 {
  \bool_set_false:N \l_keys_no_value_bool
  \keys_set_elt_aux:nn {#1} {#2}
}
```

\keys_set_elt_aux:nn
\keys_set_elt_aux:

First, set the current path and add a default if needed. There are then checks to see if the a value is required or forbidden. If everything passes, move on to execute the code.

```
\cs_new_protected:Npn \keys_set_elt_aux:nn #1#2 {
  \tl_set:Nx \l_keys_key_tl { \tl_to_str:n {#1} }
  \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / \l_keys_key_tl }
  \keys_value_or_default:n {#2}
  \keys_if_value_requirement:nTF { required } {
    \bool_if:NTF \l_keys_no_value_bool {
      \msg_kernel_error:nnx  { keys } { value-required }
        { \l_keys_path_tl }
    }{
      \keys_set_elt_aux:
    }
  }{
    \keys_set_elt_aux:
  }
}
\cs_new_protected_nopar:Npn \keys_set_elt_aux: {
  \keys_if_value_requirement:nTF { forbidden } {
    \bool_if:NTF \l_keys_no_value_bool {
      \keys_execute:
    }{
      \msg_kernel_error:nnxx { keys } { value-forbidden }
        { \l_keys_path_tl } { \tl_use:N \l_keys_value_tl }
    }
  }{
    \keys_execute:
  }
}
```

\keys_show:nn Showing a key is just a question of using the correct name.

```
\cs_new_nopar:Npn \keys_show:nn #1#2 {
  \cs_show:c { \c_keys_root_tl #1 / \tl_to_str:n {#2} .cmd:n }
}
```

\keys_tmp:w This scratch function is used to actually execute keys.

```
\cs_new:Npn \keys_tmp:w {}
```

`\keys_value_or_default:n` If a value is given, return it as `#1`, otherwise send a default if available.

```
\cs_new_protected:Npn \keys_value_or_default:n #1 {
  \tl_set:Nn \l_keys_value_tl  {#1}
  \bool_if:NT \l_keys_no_value_bool {
    \cs_if_exist:cT { \c_keys_root_tl \l_keys_path_tl .default_tl } {
      \tl_set:Nv \l_keys_value_tl  {
        \c_keys_root_tl \l_keys_path_tl .default_tl
      }
    }
  }
}
```

`\keys_value_requirement:n` Values can be required or forbidden by having the appropriate marker set.

```
\cs_new_protected_nopar:Npn \keys_value_requirement:n #1 {
  \tl_set_eq:cc { \c_keys_root_tl \l_keys_path_tl .req_tl }
    { c_keys_value_ #1 _tl }
}
```

`\keys_variable_set:NnNN`
`\keys_variable_set:cnNN` Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed.

```
\cs_new_protected_nopar:Npn \keys_variable_set:NnNN #1#2#3#4 {
  \cs_if_exist:NF #1 {
    \use:c { #2 _new:N } #1
  }
  \keys_cmd_set:nx { \l_keys_path_tl } {
    \exp_not:c { #2 _ #3 set:N #4 } \exp_not:N #1 {##1}
  }
}
\cs_generate_variant:Nn \keys_variable_set:NnNN { c }
```

117.1.3 Properties

`.bool_set:N`
`.bool_gset:N` One function for this.

```
\keys_property_new_arg:nn { .bool_set:N } {
  \keys_bool_set:NN #1 { }
}
\keys_property_new_arg:nn { .bool_gset:N } {
  \keys_bool_set:NN #1 n
}
```

`.choice:` Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```
\keys_property_new:nn { .choice: } {
  \keys_choice_make:
}
```

`.choice_code:n` `.choice_code:x` Storing the code for choices, using `\exp_not:n` to avoid needing two internal functions.

```
\keys_property_new_arg:nn { .choice_code:n } {
  \keys_choice_code_store:x { \exp_not:n {#1} }
}
\keys_property_new_arg:nn { .choice_code:x } {
  \keys_choice_code_store:x {#1}
}
```

`.code:n` `.code:x` Creating code is simply a case of passing through to the underlying `set` function.

```
\keys_property_new_arg:nn { .code:n } {
  \keys_cmd_set:nn { \l_keys_path_tl } {#1}
}
\keys_property_new_arg:nn { .code:x } {
  \keys_cmd_set:nx { \l_keys_path_tl } {#1}
}
```

`.default:n` `.default:V` Expansion is left to the internal functions.

```
\keys_property_new_arg:nn { .default:n } {
  \keys_default_set:n  {#1}
}
\keys_property_new_arg:nn { .default:V } {
  \keys_default_set:V #1
}
```

`.dim_set:N` `.dim_set:c` `.dim_gset:N` `.dim_gset:c` Setting a variable is very easy: just pass the data along.

```
\keys_property_new_arg:nn { .dim_set:N } {
  \keys_variable_set:NnNN #1 { dim } { } n
}
\keys_property_new_arg:nn { .dim_set:c } {
  \keys_variable_set:cnNN {#1} { dim } { } n
}
\keys_property_new_arg:nn { .dim_gset:N } {
  \keys_variable_set:NnNN #1 { dim } g n
}
\keys_property_new_arg:nn { .dim_gset:c } {
  \keys_variable_set:cnNN {#1} { dim } g n
}
```

`.generate_choices:n` Making choices is easy.

```
\keys_property_new_arg:nn { .generate_choices:n } {
  \keys_choices_generate:n {#1}
}
```

.int_set:N
.int_set:c
.int_gset:N
.int_gset:c

Setting a variable is very easy: just pass the data along.

```
\keys_property_new_arg:nn { .int_set:N } {
  \keys_variable_set:NnNN #1 { int } { } n
}
\keys_property_new_arg:nn { .int_set:c } {
  \keys_variable_set:cnNN {#1} { int } { } n
}
\keys_property_new_arg:nn { .int_gset:N } {
  \keys_variable_set:NnNN #1 { int } g n
}
\keys_property_new_arg:nn { .int_gset:c } {
  \keys_variable_set:cnNN {#1} { int } g n
}
```

.meta:n
.meta:x

Making a meta is handled internally.

```
\keys_property_new_arg:nn { .meta:n } {
  \keys_meta_make:n {#1}
}
\keys_property_new_arg:nn { .meta:x } {
  \keys_meta_make:x {#1}
}
```

.skip_set:N
.skip_set:c
.skip_gset:N
.skip_gset:c

Setting a variable is very easy: just pass the data along.

```
\keys_property_new_arg:nn { .skip_set:N } {
  \keys_variable_set:NnNN #1 { skip } { } n
}
\keys_property_new_arg:nn { .skip_set:c } {
  \keys_variable_set:cnNN {#1} { skip } { } n
}
\keys_property_new_arg:nn { .skip_gset:N } {
  \keys_variable_set:NnNN #1 { skip } g n
}
\keys_property_new_arg:nn { .skip_gset:c } {
  \keys_variable_set:cnNN {#1} { skip } g n
}
```

.tl_set:N
.tl_set:c
.tl_set_x:N
.tl_set_x:c
.tl_gset:N
.tl_gset:c
.tl_gset_x:N
.tl_gset_x:c

Setting a variable is very easy: just pass the data along.

```
\keys_property_new_arg:nn { .tl_set:N } {
  \keys_variable_set:NnNN #1 { tl } { } n
}
\keys_property_new_arg:nn { .tl_set:c } {
  \keys_variable_set:cnNN {#1} { tl } { } n
}
\keys_property_new_arg:nn { .tl_set_x:N } {
  \keys_variable_set:NnNN #1 { tl } { } x
}
```

```
\keys_property_new_arg:nn { .tl_set_x:c } {
  \keys_variable_set:cnNN {#1} { tl } { } x
}
\keys_property_new_arg:nn { .tl_gset:N } {
  \keys_variable_set:NnNN #1 { tl } g n
}
\keys_property_new_arg:nn { .tl_gset:c } {
  \keys_variable_set:cnNN {#1} { tl } g n
}
\keys_property_new_arg:nn { .tl_gset_x:N } {
  \keys_variable_set:NnNN #1 { tl } g x
}
\keys_property_new_arg:nn { .tl_gset_x:c } {
  \keys_variable_set:cnNN {#1} { tl } g x
}
```

`.value_forbidden:`
`.value_required:`

These are very similar, so both call the same function.

```
\keys_property_new:nn { .value_forbidden: } {
  \keys_value_requirement:n { forbidden }
}
\keys_property_new:nn { .value_required: } {
  \keys_value_requirement:n { required }
}
```

117.1.4 Messages

For when there is a need to complain.

```
\msg_kernel_new:nnnn { keys } { choice-unknown }
  { Choice~'#2'~unknown~for~key~'#1'. }
  {
    The~key~'#1'~takes~a~limited~number~of~values.\\
    The~input~given,~'#2',~is~not~on~the~list~accepted.
  }
\msg_kernel_new:nnnn { keys } { generate-choices-before-code }
  { No~code~available~to~generate~choices~for~key~'#1'. }
  {
    \l_msg_coding_error_text_tl
    Before~using~.generate_choices:n~the~code~should~be~defined\\%
    with~.choice_code:n~or~.choice_code:x.
  }
\msg_kernel_new:nnnn { keys } { key-no-property }
  { No~property~given~in~definition~of~key~'#1'. }
  {
    \c_msg_coding_error_text_tl
    Inside~\token_to_str:N \keys_define:nn \c_space_tl each~key~name
    needs~a~property:  \\
    ~ ~ #1 .<property> \\
```

```
    LaTeX~did~not~find~a~'.'~to~indicate~the~start~of~a~property.
  }
\msg_kernel_new:nnnn { keys } { key-unknown }
  { The~key~'#1'~is~unknown~and~is~being~ignored. }
  {
    The~module~'#2'~does~not~have~a~key~called~#1'.\\
    Check~that~you~have~spelled~the~key~name~correctly.
  }
\msg_kernel_new:nnnn { keys } { property-requires-value }
  { The property '#1' requires a value. }
  {
    \l_msg_coding_error_text_tl
    LaTeX~was~asked~to~set~property~'#2'~for~key~'#1'.\\
    No~value~was~given~for~the~property,~and~one~is~required.
  }
\msg_kernel_new:nnnn { keys } { property-unknown }
  { The key property '#1' is unknown. }
  {
    \l_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':\\
    this~property~is~not~defined.
  }
\msg_kernel_new:nnnn { keys } { value-forbidden }
  { The~key~'#1'~does~not~taken~a~value. }
  {
    The~key~'#1'~should~be~given~without~a~value.\\
    LaTeX~will~ignore~the~given~value~'#2'.
  }
\msg_kernel_new:nnnn { keys } { value-required }
  { The~key~'#1'~requires~a~value. }
  {
    The~key~'#1'~must~have~a~value.\\
    No~value~was~present:~the~key~will~be~ignored.
  }
```

⟨/initex | package⟩

118 l3calc implementation

118.1 Variables

```
\l_calc_expression_tl
\g_calc_A_register
\l_calc_B_register
\l_calc_current_type_int
\g_calc_A_int
\l_calc_B_int
\l_calc_C_int
\g_calc_A_dim
\l_calc_B_dim
\l_calc_C_dim
\g_calc_A_skip
\l_calc_B_skip
\l_calc_C_skip
\g_calc_A_muskip
\l_calc_B_muskip
\l_calc_C_muskip
```

Internal registers.

118.2 Internal functions

```
\calc_assign_generic:NNNNnn
\calc_pre_scan:N
\calc_open:w
\calc_init_B:
\calc_numeric:
\calc_close:
\calc_post_scan:N
\calc_multiply:N
\calc_divide:N
\calc_generic_add_or_subtract:N
\calc_add:
\calc_subtract:
\calc_add_A_to_B:
\calc_subtract_A_from_B:
\calc_generic_multiply_or_divide:N
\calc_multiply_B_by_A:
\calc_divide_B_by_A:
\calc_multiply:
\calc_divide:
\calc_textsize:Nn
\calc_ratio_multiply:nn
\calc_ratio_divide:nn
\calc_real_evaluate:nn
\calc_real_multiply:n
\calc_real_divide:n
\calc_maxmin_operation:Nnn
\calc_maxmin_generic:Nnn
\calc_maxmin_div_or_mul:NNnn
\calc_maxmin_multiply:
\calc_maxmin_multiply:
\calc_error:N
\calc_chk_document_counter:nn
```

Awaiting better documentation :)

118.3 Module code

Since this is basically a re-worked version of the calc package, I haven't bothered with too many comments except for in the places where this package differs. This may (and should) change at some point.

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
```

```
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

`\l_calc_expression_tl`
`\g_calc_A_register`
`\l_calc_B_register`
`\l_calc_current_type_int`

Here we define some registers and pointers we will need.

```
\tl_new:N \l_calc_expression_tl
\cs_new_nopar:Npn \g_calc_A_register{}
\cs_new_nopar:Npn \l_calc_B_register{}
\int_new:N \l_calc_current_type_int
```

`\g_calc_A_int`
`\l_calc_B_int`
`\l_calc_C_int`
`\g_calc_A_dim`
`\l_calc_B_dim`
`\l_calc_C_dim`
`\g_calc_A_skip`
`\l_calc_B_skip`
`\l_calc_C_skip`
`\g_calc_A_muskip`
`\l_calc_B_muskip`
`\l_calc_C_muskip`

For each type of register we will need three registers to do our manipulations.

```
\int_new:N \g_calc_A_int
\int_new:N \l_calc_B_int
\int_new:N \l_calc_C_int
\dim_new:N \g_calc_A_dim
\dim_new:N \l_calc_B_dim
\dim_new:N \l_calc_C_dim
\skip_new:N \g_calc_A_skip
\skip_new:N \l_calc_B_skip
\skip_new:N \l_calc_C_skip
\muskip_new:N \g_calc_A_muskip
\muskip_new:N \l_calc_B_muskip
\muskip_new:N \l_calc_C_muskip
```

`\calc_assign_generic:NNNNnn`

The generic function. `#1` is a number denoting which type we are doing. (0=int, 1=dim, 2=skip, 3=muskip), `#2` = temp register A, `#3` = temp register B, `#4` is a function acting on `#5` which is the register to be set. `#6` is the calc expression. We do a little extra work so that `\real` and `\ratio` can still be used by the user.

```
\cs_new:Npn \calc_assign_generic:NNNNnn#1#2#3#4#5#6{
  \cs_set_eq:NN\g_calc_A_register#2
  \cs_set_eq:NN\l_calc_B_register#3
  \int_set:Nn \l_calc_current_type_int {#1}
  \group_begin:
    \cs_set_eq:NN \real \calc_real:n
    \cs_set_eq:NN \ratio\calc_ratio:nn
    \tl_set:Nx\l_calc_expression_tl{#6}
    \exp_after:wN
  \group_end:
  \exp_after:wN\calc_open:w\exp_after:wN(\l_calc_expression_tl !
  \pref_global:D\g_calc_A_register\l_calc_B_register
  \group_end:
  #4{#5}\l_calc_B_register
}
```

A simpler version relying on `\real` and `\ratio` having our definition is

```
\cs_new:Npn \calc_assign_generic:NNNNnn#1#2#3#4#5#6{
  \cs_set_eq:NN\g_calc_A_register#2\cs_set_eq:NN\l_calc_B_register#3
  \int_set:Nn \l_calc_current_type_int {#1}
  \tl_set:Nx\l_calc_expression_tl{#6}
  \exp_after:wN\calc_open:w\exp_after:wN(\l_calc_expression_tl !
  \pref_global:D\g_calc_A_register\l_calc_B_register
  \group_end:
  #4{#5}\l_calc_B_register
}
```

\calc_int_set:Nn
\calc_int_gset:Nn
\calc_int_add:Nn
\calc_int_gadd:Nn
\calc_int_sub:Nn
\calc_int_gsub:Nn

Here are the individual versions for the different register types. First integer registers.

```
\cs_new_nopar:Npn\calc_int_set:Nn{
  \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_set:Nn
}
\cs_new_nopar:Npn\calc_int_gset:Nn{
  \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_gset:Nn
}
\cs_new_nopar:Npn\calc_int_add:Nn{
  \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_add:Nn
}
\cs_new_nopar:Npn\calc_int_gadd:Nn{
  \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_gadd:Nn
}
\cs_new_nopar:Npn\calc_int_sub:Nn{
  \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_sub:Nn
}
\cs_new_nopar:Npn\calc_int_gsub:Nn{
  \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_gsub:Nn
}
```

\calc_dim_set:Nn
\calc_dim_gset:Nn
\calc_dim_add:Nn
\calc_dim_gadd:Nn
\calc_dim_sub:Nn
\calc_dim_gsub:Nn

Dimens.

```
\cs_new_nopar:Npn\calc_dim_set:Nn{
  \calc_assign_generic:NNNNnn\c_one\g_calc_A_dim\l_calc_B_dim\dim_set:Nn
}
\cs_new_nopar:Npn\calc_dim_gset:Nn{
  \calc_assign_generic:NNNNnn\c_one\g_calc_A_dim\l_calc_B_dim\dim_gset:Nn
}
\cs_new_nopar:Npn\calc_dim_add:Nn{
  \calc_assign_generic:NNNNnn\c_one\g_calc_A_dim\l_calc_B_dim\dim_add:Nn
}
\cs_new_nopar:Npn\calc_dim_gadd:Nn{
  \calc_assign_generic:NNNNnn\c_one\g_calc_A_dim\l_calc_B_dim\dim_gadd:Nn
}
\cs_new_nopar:Npn\calc_dim_sub:Nn{
  \calc_assign_generic:NNNNnn\c_one\g_calc_A_int\l_calc_B_int\dim_sub:Nn
}
\cs_new_nopar:Npn\calc_dim_gsub:Nn{
```

```
  \calc_assign_generic:NNNNnn\c_one\g_calc_A_int\l_calc_B_int\dim_gsub:Nn
}
```

\calc_skip_set:Nn
\calc_skip_gset:Nn
\calc_skip_add:Nn
\calc_skip_gadd:Nn
\calc_skip_sub:Nn
\calc_skip_gsub:Nn

Skips.

```
\cs_new_nopar:Npn\calc_skip_set:Nn{
  \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_set:Nn
}
\cs_new_nopar:Npn\calc_skip_gset:Nn{
  \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_gset:Nn
}
\cs_new_nopar:Npn\calc_skip_add:Nn{
  \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_add:Nn
}
\cs_new_nopar:Npn\calc_skip_gadd:Nn{
  \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_gadd:Nn
}
\cs_new_nopar:Npn\calc_skip_sub:Nn{
  \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_sub:Nn
}
\cs_new_nopar:Npn\calc_skip_gsub:Nn{
  \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_gsub:Nn
}
```

\calc_muskip_set:Nn
\calc_muskip_gset:Nn
\calc_muskip_add:Nn
\calc_muskip_gadd:Nn
\calc_muskip_sub:Nn
\calc_muskip_gsub:Nn

Muskips.

```
\cs_new_nopar:Npn\calc_muskip_set:Nn{
  \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
    \muskip_set:Nn
}
\cs_new_nopar:Npn\calc_muskip_gset:Nn{
  \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
    \muskip_gset:Nn
}
\cs_new_nopar:Npn\calc_muskip_add:Nn{
  \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
    \muskip_add:Nn
}
\cs_new_nopar:Npn\calc_muskip_gadd:Nn{
  \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
    \muskip_gadd:Nn
}
\cs_new_nopar:Npn\calc_muskip_sub:Nn{
  \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
    \muskip_add:Nn
}
\cs_new_nopar:Npn\calc_muskip_gsub:Nn{
  \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
    \muskip_gadd:Nn
}
```

\calc_pre_scan:N In case we found one of the special operations, this should just be executed.

```
\cs_new_nopar:Npn \calc_pre_scan:N #1{
  \if_meaning:w(#1
    \exp_after:wN\calc_open:w
  \else:
    \if_meaning:w \calc_textsize:Nn #1
    \else:
      \if_meaning:w \calc_maxmin_operation:Nnn #1
      \else:
```

\calc_numeric: uses a primitive assignment so doesn't care about these dangling \fi:s.

```
        \calc_numeric:
      \fi:
    \fi:
  \fi:
  #1}
```

\calc_open:w

```
\cs_new_nopar:Npn \calc_open:w({
  \group_begin:\group_execute_after:N\calc_init_B:
  \group_begin:\group_execute_after:N\calc_init_B:
  \calc_pre_scan:N
}
```

\calc_init_B:
\calc_numeric:
\calc_close:

```
\cs_new_nopar:Npn\calc_init_B:{\l_calc_B_register\g_calc_A_register}
\cs_new_nopar:Npn\calc_numeric:{
  \tex_afterassignment:D\calc_post_scan:N
  \pref_global:D\g_calc_A_register
}
\cs_new_nopar:Npn\calc_close:{
  \group_end:\pref_global:D\g_calc_A_register\l_calc_B_register
  \group_end:\pref_global:D\g_calc_A_register\l_calc_B_register
  \calc_post_scan:N}
```

\calc_post_scan:N Look at what token we have and decide where to go.

```
\cs_new_nopar:Npn\calc_post_scan:N#1{
  \if_meaning:w#1!\cs_set_eq:NN\calc_next:w\group_end: \else:
    \if_meaning:w#1+\cs_set_eq:NN\calc_next:w\calc_add: \else:
      \if_meaning:w#1-\cs_set_eq:NN\calc_next:w\calc_subtract: \else:
        \if_meaning:w#1*\cs_set_eq:NN\calc_next:w\calc_multiply:N \else:
          \if_meaning:w#1/\cs_set_eq:NN\calc_next:w\calc_divide:N \else:
            \if_meaning:w#1)\cs_set_eq:NN\calc_next:w\calc_close: \else:
              \if_meaning:w#1\scan_stop:\cs_set_eq:NN\calc_next:w\calc_post_scan:N
              \else:
```

If we get here, there is an error but let's also disable `\calc_next:w` since it is otherwise undefined. No need to give extra errors just for that.

```
          \cs_set_eq:NN \calc_next:w \prg_do_nothing:
          \calc_error:N#1
         \fi:
        \fi:
       \fi:
      \fi:
     \fi:
    \fi:
  \fi:
  \calc_next:w}
```

`\calc_multiply:N`
`\calc_divide:N`

The switches for multiplication and division.

```
\cs_new_nopar:Npn \calc_multiply:N #1{
  \if_meaning:w \calc_maxmin_operation:Nnn #1
    \cs_set_eq:NN \calc_next:w \calc_maxmin_multiply:
  \else:
    \if_meaning:w \calc_ratio_multiply:nn #1
      \cs_set_eq:NN \calc_next:w \calc_ratio_multiply:nn
    \else:
      \if_meaning:w \calc_real_evaluate:nn #1
        \cs_set_eq:NN \calc_next:w \calc_real_multiply:n
      \else:
        \cs_set_nopar:Npn \calc_next:w{\calc_multiply: #1}
      \fi:
    \fi:
  \fi:
  \calc_next:w
}
\cs_new_nopar:Npn \calc_divide:N #1{
  \if_meaning:w \calc_maxmin_operation:Nnn #1
    \cs_set_eq:NN \calc_next:w \calc_maxmin_divide:
  \else:
    \if_meaning:w \calc_ratio_multiply:nn #1
      \cs_set_eq:NN \calc_next:w \calc_ratio_divide:nn
    \else:
      \if_meaning:w \calc_real_evaluate:nn #1
        \cs_set_eq:NN \calc_next:w \calc_real_divide:n
      \else:
        \cs_set_nopar:Npn \calc_next:w{\calc_divide: #1}
      \fi:
    \fi:
  \fi:
  \calc_next:w
}
```

`\calc_generic_add_or_subtract:N`
`\calc_add:`
`\calc_subtract:`
`\calc_add_A_to_B:`
`\calc_subtract_A_from_B:`

Here is how we add and subtract.

```
\cs_new_nopar:Npn\calc_generic_add_or_subtract:N#1{
  \group_end:
  \pref_global:D\g_calc_A_register\l_calc_B_register\group_end:
  \group_begin:\group_execute_after:N#1\group_begin:
  \group_execute_after:N\calc_init_B:
  \calc_pre_scan:N}
\cs_new_nopar:Npn\calc_add:{\calc_generic_add_or_subtract:N\calc_add_A_to_B:}
\cs_new_nopar:Npn\calc_subtract:{
  \calc_generic_add_or_subtract:N\calc_subtract_A_from_B:}
```

Don't use `\tex_advance:D` since it allows overflows.

```
\cs_new_nopar:Npn\calc_add_A_to_B:{
  \l_calc_B_register
  \if_case:w\l_calc_current_type_int
  \etex_numexpr:D\or:
  \etex_dimexpr:D\or:
  \etex_glueexpr:D\or:
  \etex_muexpr:D\fi:
  \l_calc_B_register + \g_calc_A_register\scan_stop:
}
\cs_new_nopar:Npn\calc_subtract_A_from_B:{
  \l_calc_B_register
  \if_case:w\l_calc_current_type_int
  \etex_numexpr:D\or:
  \etex_dimexpr:D\or:
  \etex_glueexpr:D\or:
  \etex_muexpr:D\fi:
  \l_calc_B_register - \g_calc_A_register\scan_stop:
}
```

`lc_generic_multiply_or_divide:N`
`\calc_multiply_B_by_A:`
`\calc_divide_B_by_A:`
`\calc_multiply:`
`\calc_divide:`

And here is how we multiply and divide. Note that we do not use the primitive TEX operations but the expandable operations provided by ε-TEX. This means that all results are rounded not truncated!

```
\cs_new_nopar:Npn\calc_generic_multiply_or_divide:N#1{
  \group_end:
  \group_begin:
  \cs_set_eq:NN\g_calc_A_register\g_calc_A_int
  \cs_set_eq:NN\l_calc_B_register\l_calc_B_int
  \int_zero:N \l_calc_current_type_int
  \group_execute_after:N#1\calc_pre_scan:N
}
\cs_new_nopar:Npn\calc_multiply_B_by_A:{
  \l_calc_B_register
  \if_case:w\l_calc_current_type_int
  \etex_numexpr:D\or:
  \etex_dimexpr:D\or:
  \etex_glueexpr:D\or:
  \etex_muexpr:D\fi:
```

```
  \l_calc_B_register*\g_calc_A_int\scan_stop:
}
\cs_new_nopar:Npn\calc_divide_B_by_A:{
  \l_calc_B_register
  \if_case:w\l_calc_current_type_int
  \etex_numexpr:D\or:
  \etex_dimexpr:D\or:
  \etex_glueexpr:D\or:
  \etex_muexpr:D\fi:
  \l_calc_B_register/\g_calc_A_int\scan_stop:
}
\cs_new_nopar:Npn\calc_multiply:{
  \calc_generic_multiply_or_divide:N\calc_multiply_B_by_A:}
\cs_new_nopar:Npn\calc_divide:{
  \calc_generic_multiply_or_divide:N\calc_divide_B_by_A:}
```

\calc_calculate_box_size:nnn
\calc_calculate_box_size_aux:n

Put something in a box and measure it. #1 is a list of \box_ht:N etc., #2 should be \dim_set:Nn⟨*dim register*⟩ or \dim_gset:Nn⟨*dim register*⟩ and #3 is the contents.

```
\cs_new:Npn \calc_calculate_box_size:nnn #1#2#3{
  \hbox_set:Nn \l_tmpa_box {{#3}}
  #2{\c_zero_dim \tl_map_function:nN{#1}\calc_calculate_box_size_aux:n}
}
```

Helper for calculating the final dimension.

```
\cs_set_nopar:Npn \calc_calculate_box_size_aux:n#1{ + #1\l_tmpa_box}
```

\calc_textsize:Nn

Now we can define \calc_textsize:Nn.

```
\cs_set_protected:Npn \calc_textsize:Nn#1#2{
  \group_begin:
  \cs_set_eq:NN\calc_widthof_aux:n\box_wd:N
  \cs_set_eq:NN\calc_heightof_aux:n\box_ht:N
  \cs_set_eq:NN\calc_depthof_aux:n\box_dp:N
  \cs_set_nopar:Npn\calc_totalheightof_aux:n{\box_ht:N\box_dp:N}
  \exp_args:No\calc_calculate_box_size:nnn{#1}
  {\dim_gset:Nn\g_calc_A_register}
```

Restore the four user commands here since there might be a recursive call.

```
  {
    \cs_set_eq:NN \calc_depthof_aux:n \calc_depthof_auxi:n
    \cs_set_eq:NN \calc_widthof_aux:n \calc_widthof_auxi:n
    \cs_set_eq:NN \calc_heightof_aux:n \calc_heightof_auxi:n
    \cs_set_eq:NN \calc_totalheightof_aux:n \calc_totalheightof_auxi:n
    #2
  }
  \group_end:
  \calc_post_scan:N
}
```

\calc_ratio_multiply:nn
\calc_ratio_divide:nn

Evaluate a ratio. If we were already evaluation a ⟨*muskip*⟩ register, the ratio is probably also done with this type and we'll have to convert them to regular points.

```
\cs_set_protected:Npn\calc_ratio_multiply:nn#1#2{
  \group_end:\group_begin:
  \if_num:w\l_calc_current_type_int < \c_three
    \calc_dim_set:Nn\l_calc_B_int{#1}
    \calc_dim_set:Nn\l_calc_C_int{#2}
  \else:
    \calc_dim_muskip:Nn{\l_calc_B_int\etex_mutoglue:D}{#1}
    \calc_dim_muskip:Nn{\l_calc_C_int\etex_mutoglue:D}{#2}
  \fi:
```

Then store the ratio as a fraction, which we just pass on.

```
  \cs_gset_nopar:Npx\calc_calculated_ratio:{
    \int_use:N\l_calc_B_int/\int_use:N\l_calc_C_int
  }
  \group_end:
```

Here we set the new value of `\l_calc_B_register` and remember to evaluate it as the correct type. Note that the intermediate calculation is a scaled operation (meaning the intermediate value is 64-bit) so we don't get into trouble when first multiplying by a large number and then dividing.

```
  \l_calc_B_register
  \if_case:w\l_calc_current_type_int
  \etex_numexpr:D\or:
  \etex_dimexpr:D\or:
  \etex_glueexpr:D\or:
  \etex_muexpr:D\fi:
  \l_calc_B_register*\calc_calculated_ratio:\scan_stop:
  \group_begin:
  \calc_post_scan:N}
```

Division is just flipping the arguments around.

```
\cs_new:Npn \calc_ratio_divide:nn#1#2{\calc_ratio_multiply:nn{#2}{#1}}
```

\calc_real_evaluate:nn
\calc_real_multiply:n
\calc_real_divide:n

Although we could define the `\real` function as a subcase of `\ratio`, this is horribly inefficient since we just want to convert the decimal to a fraction.

```
\cs_new_protected_nopar:Npn\calc_real_evaluate:nn #1#2{
  \group_end:
  \l_calc_B_register
  \if_case:w\l_calc_current_type_int
  \etex_numexpr:D\or:
  \etex_dimexpr:D\or:
  \etex_glueexpr:D\or:
  \etex_muexpr:D\fi:
```

```
  \l_calc_B_register *
    \tex_number:D \dim_eval:n{#1pt}/
    \tex_number:D\dim_eval:n{#2pt}
  \scan_stop:
  \group_begin:
  \calc_post_scan:N}
\cs_new_nopar:Npn \calc_real_multiply:n #1{\calc_real_evaluate:nn{#1}{1}}
\cs_new_nopar:Npn \calc_real_divide:n {\calc_real_evaluate:nn{1}}
```

\calc_maxmin_operation:Nnn
\calc_maxmin_generic:Nnn
\calc_maxmin_div_or_mul:NNnn
\calc_maxmin_multiply:
\calc_maxmin_multiply:

The max and min functions.

```
\cs_set_protected:Npn\calc_maxmin_operation:Nnn#1#2#3{
  \group_begin:
  \calc_maxmin_generic:Nnn#1{#2}{#3}
  \group_end:
  \calc_post_scan:N
}
```

`#1` is either `>` or `<` and was expanded into this initially.

```
\cs_new_protected:Npn \calc_maxmin_generic:Nnn#1#2#3{
  \group_begin:
  \if_case:w\l_calc_current_type_int
    \calc_int_set:Nn\l_calc_C_int{#2}%
    \calc_int_set:Nn\l_calc_B_int{#3}%
    \pref_global:D\g_calc_A_register
    \if_num:w\l_calc_C_int#1\l_calc_B_int
    \l_calc_C_int\else:\l_calc_B_int\fi:
  \or:
    \calc_dim_set:Nn\l_calc_C_dim{#2}%
    \calc_dim_set:Nn\l_calc_B_dim{#3}%
    \pref_global:D\g_calc_A_register
    \if_dim:w\l_calc_C_dim#1\l_calc_B_dim
    \l_calc_C_dim\else:\l_calc_B_dim\fi:
  \or:
    \calc_skip_set:Nn\l_calc_C_skip{#2}%
    \calc_skip_set:Nn\l_calc_B_skip{#3}%
    \pref_global:D\g_calc_A_register
    \if_dim:w\l_calc_C_skip#1\l_calc_B_skip
    \l_calc_C_skip\else:\l_calc_B_skip\fi:
  \else:
    \calc_muskip_set:Nn\l_calc_C_muskip{#2}%
    \calc_muskip_set:Nn\l_calc_B_muskip{#3}%
    \pref_global:D\g_calc_A_register
    \if_dim:w\l_calc_C_muskip#1\l_calc_B_muskip
    \l_calc_C_muskip\else:\l_calc_B_muskip\fi:
  \fi:
  \group_end:
}
\cs_new:Npn\calc_maxmin_div_or_mul:NNnn#1#2#3#4{
```

```
  \group_end:
  \group_begin:
  \int_zero:N\l_calc_current_type_int
  \group_execute_after:N#1
  \calc_maxmin_generic:Nnn#2{#3}{#4}
  \group_end:
  \group_begin:
  \calc_post_scan:N
}
\cs_new_nopar:Npn\calc_maxmin_multiply:{
  \calc_maxmin_div_or_mul:NNnn\calc_multiply_B_by_A:}
\cs_new_nopar:Npn\calc_maxmin_divide:  {
  \calc_maxmin_div_or_mul:NNnn\calc_divide_B_by_A:}
```

\calc_error:N

The error message.

```
\cs_new_nopar:Npn\calc_error:N#1{
  \PackageError{calc}
  {`\token_to_str:N#1'~ invalid~ at~ this~ point}
  {I~ expected~ to~ see~ one~ of:~ +~ -~ *~ /~ )}
}
```

118.4 Higher level commands

The various operations allowed.

\calc_maxof:nn
\calc_minof:nn
\maxof
\minof

Max and min operations

```
\cs_new:Npn \calc_maxof:nn#1#2{
  \calc_maxmin_operation:Nnn > \exp_not:n{{#1}{#2}}
}
\cs_new:Npn \calc_minof:nn#1#2{
  \calc_maxmin_operation:Nnn < \exp_not:n{{#1}{#2}}
}
\cs_set_eq:NN \maxof \calc_maxof:nn
\cs_set_eq:NN \minof \calc_minof:nn
```

\calc_widthof:n
\calc_widthof_aux:n
\calc_widthof_auxi:n
\calc_heightof:n
\calc_heightof_aux:n
\calc_heightof_auxi:n
\calc_depthof:n
\calc_depthof_aux:n
\calc_depthof_auxi:n
\calc_totalheightof:n
\calc_totalheightof_aux:n
\calc_totalheightof_auxi:n

Text dimension commands.

```
\cs_new:Npn \calc_widthof:n#1{
  \calc_textsize:Nn \exp_not:N\calc_widthof_aux:n\exp_not:n{{#1}}
}
\cs_new:Npn \calc_heightof:n#1{
  \calc_textsize:Nn \exp_not:N\calc_heightof_aux:n\exp_not:n{{#1}}
}
\cs_new:Npn \calc_depthof:n#1{
  \calc_textsize:Nn \exp_not:N\calc_depthof_aux:n\exp_not:n{{#1}}
}
```

```
\cs_new:Npn \calc_totalheightof:n#1{
  \calc_textsize:Nn \exp_not:N\calc_totalheightof_aux:n \exp_not:n{{#1}}
}
\cs_new:Npn \calc_widthof_aux:n #1{
  \exp_not:N\calc_widthof_aux:n\exp_not:n{{#1}}
}
\cs_new_eq:NN \calc_widthof_auxi:n \calc_widthof_aux:n
\cs_new:Npn \calc_depthof_aux:n #1{
  \exp_not:N\calc_depthof_aux:n\exp_not:n{{#1}}
}
\cs_new_eq:NN \calc_depthof_auxi:n \calc_depthof_aux:n
\cs_new:Npn \calc_heightof_aux:n #1{
  \exp_not:N\calc_heightof_aux:n\exp_not:n{{#1}}
}
\cs_new_eq:NN \calc_heightof_auxi:n \calc_heightof_aux:n
\cs_new:Npn \calc_totalheightof_aux:n #1{
  \exp_not:N\calc_totalheightof_aux:n\exp_not:n{{#1}}
}
\cs_new_eq:NN \calc_totalheightof_auxi:n \calc_totalheightof_aux:n
```

`\calc_ratio:nn`
`\calc_real:n`

Ratio and real.

```
\cs_new:Npn \calc_ratio:nn#1#2{
  \calc_ratio_multiply:nn\exp_not:n{{#1}{#2}}}
\cs_new_nopar:Npn \calc_real:n {\calc_real_evaluate:nn}
```

We can implement real and ratio without actually using these names. We'll see.

`\widthof`
`\heightof`
`\depthof`
`\totalheightof`
`\ratio`
`\real`

User commands.

```
\cs_set_eq:NN \depthof\calc_depthof:n
\cs_set_eq:NN \widthof\calc_widthof:n
\cs_set_eq:NN \heightof\calc_heightof:n
\cs_set_eq:NN \totalheightof\calc_totalheightof:n
%%\cs_set_eq:NN \ratio\calc_ratio:nn
%%\cs_set_eq:NN \real\calc_real:n
```

`\setlength`
`\gsetlength`
`\addtolength`
`\gaddtolength`

```
\cs_set_protected_nopar:Npn \setlength{\calc_skip_set:Nn}
\cs_set_protected_nopar:Npn \gsetlength{\calc_skip_gset:Nn}
\cs_set_protected_nopar:Npn \addtolength{\calc_skip_add:Nn}
\cs_set_protected_nopar:Npn \gaddtolength{\calc_skip_gadd:Nn}
```

`\calc_setcounter:nn`
`\calc_addtocounter:nn`
`\calc_stepcounter:n`
`\setcounter`
`\addtocounter`
`\stepcounter`
`\calc_chk_document_counter:nn`

Document commands for $\LaTeX\,2_\varepsilon$ counters. Also add support for amstext. Note that when l3breqn is used, `\mathchoice` will no longer need this switch as the argument is only executed once.

```
⟨/initex | package⟩
⟨*package⟩
\newif\iffirstchoice@    \firstchoice@true
⟨/package⟩
⟨*initex | package⟩
\cs_set_protected_nopar:Npn \calc_setcounter:nn#1#2{
  \calc_chk_document_counter:nn{#1}{
    \exp_args:Nc\calc_int_gset:Nn {c@#1}{#2}
  }
}
\cs_set_protected_nopar:Npn \calc_addtocounter:nn#1#2{
⟨/initex | package⟩
⟨*package⟩
  \iffirstchoice@
⟨/package⟩
⟨*initex | package⟩
  \calc_chk_document_counter:nn{#1}{
    \exp_args:Nc\calc_int_gadd:Nn {c@#1}{#2}
  }
⟨/initex | package⟩
⟨*package⟩
  \fi:
⟨/package⟩
⟨*initex | package⟩
}
\cs_set_protected_nopar:Npn \calc_stepcounter:n#1{
⟨/initex | package⟩
⟨*package⟩
  \iffirstchoice@
⟨/package⟩
⟨*initex | package⟩
  \calc_chk_document_counter:nn{#1}{
    \int_gincr:c {c@#1}
    \group_begin:
      \cs_set_eq:NN \@elt\@stpelt \use:c{cl@#1}
    \group_end:
  }
⟨/initex | package⟩
⟨*package⟩
  \fi:
⟨/package⟩
⟨*initex | package⟩
}
\cs_new_nopar:Npn \calc_chk_document_counter:nn#1{
  \cs_if_free:cTF{c@#1}{\@nocounterr {#1}}
}
\cs_set_eq:NN \setcounter \calc_setcounter:nn
\cs_set_eq:NN \addtocounter \calc_addtocounter:nn
\cs_set_eq:NN \stepcounter \calc_stepcounter:n
⟨/initex | package⟩
```

```
⟨*package⟩
\AtBeginDocument{
  \cs_set_eq:NN \setcounter \calc_setcounter:nn
  \cs_set_eq:NN \addtocounter \calc_addtocounter:nn
  \cs_set_eq:NN \stepcounter \calc_stepcounter:n
}
```

Prevent the usual calc from loading.

```
\cs_set_nopar:cpn{ver@calc.sty}{2005/08/06}
⟨/package⟩
```

```
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

119 l3file implementation

The usual lead-off.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\package_check_loaded_expl:
⟨/package⟩
⟨*initex | package⟩
```

\g_file_current_name_tl
\g_file_stack_seq

The name of the current file should be available at all times.

```
\tl_new:N \g_file_current_name_tl
\seq_new:N \g_file_stack_seq
```

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 LaTeX2e.

```
⟨/initex | package⟩
⟨*initex⟩
\toks_put_right:Nn \tex_everyjob:D {
  \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
}
⟨/initex⟩
⟨*package⟩
\tl_gset_eq:NN \g_file_current_name_tl \@currname
⟨/package⟩
⟨*initex | package⟩
```

\g_file_record_seq

The total list of files used is recorded separately from the stack.

```
\seq_new:N \g_file_record_seq
```

The current file name should be included in the file list!

```
⟨/initex | package⟩
⟨*initex⟩
\toks_put_right:Nn \tex_everyjob:D {
  \seq_gput_right:NV \g_file_record_seq \g_file_current_name_tl
}
⟨/initex⟩
⟨*initex | package⟩
```

`\l_file_search_path_seq`

The current search path.

```
\seq_new:N \l_file_search_path_seq
```

`\l_file_search_path_saved_seq`

The current search path has to be saved for package use.

```
⟨/initex | package⟩
⟨*package⟩
\seq_new:N \l_file_search_path_saved_seq
⟨/package⟩
⟨*initex | package⟩
```

`\l_file_name_tl`
`\g_file_test_stream`
`\file_if_exist:nTF`
`\file_if_exist:VTF`
`\file_if_exist_aux:n`

Checking if a file exists takes place in two parts. First, look on the TeX path, then look on the LaTeX path. The token list `\l_file_name_tl` is used as a marker for finding the file, and is also needed by `\file_input:n`.

```
\tl_new:N \l_file_name_tl
\prg_new_protected_conditional:Nnn \file_if_exist:n { T , F , TF } {
  \ior_open:Nn \g_file_test_stream {#1}
  \ior_if_eof:NTF \g_file_test_stream
    { \file_if_exist_path_aux:n {#1} }
    {
      \ior_close:N \g_file_test_stream
      \tl_set:Nn \l_file_name_tl {#1}
      \prg_return_true:
    }
}
\cs_new_protected_nopar:Npn \file_if_exist_path_aux:n #1 {
  \tl_clear:N \l_file_name_tl
⟨/initex | package⟩
⟨*package⟩
  \cs_if_exist:NT \input@path
    {
      \seq_set_eq:NN \l_file_search_path_saved_seq
        \l_file_search_path_seq
      \clist_map_inline:Nn \input@path
        { \seq_put_right:Nn \l_file_search_path_seq {##1} }
    }
⟨/package⟩
```

```
⟨*initex | package⟩
  \seq_map_inline:Nn \l_file_search_path_seq
    {
      \ior_open:Nn \g_file_test_stream { ##1 #1 }
      \ior_if_eof:NF \g_file_test_stream
        {
          \tl_set:Nn \l_file_name_tl { ##1 #1 }
          \seq_map_break:
        }
    }
⟨/initex | package⟩
⟨*package⟩
  \cs_if_exist:NT \input@path
    {
      \seq_set_eq:NN \l_file_search_path_seq
        \l_file_search_path_saved_seq
    }
⟨/package⟩
⟨*initex | package⟩
  \ior_close:N \g_file_test_stream
  \tl_if_empty:NTF \l_file_name_tl
    { \prg_return_false: }
    { \prg_return_true: }
}
\cs_generate_variant:Nn \file_if_exist:nT  { V }
\cs_generate_variant:Nn \file_if_exist:nF  { V }
\cs_generate_variant:Nn \file_if_exist:nTF { V }
```

\file_input:n
\file_input:V

Most of the work is done by the file test above.

```
\cs_new_protected_nopar:Npn \file_input:n #1 {
  \file_if_exist:nT {#1}
    {
⟨/initex | package⟩
⟨*package⟩
      \@addtofilelist {#1}
⟨/package⟩
⟨*initex | package⟩
      \seq_gpush:NV \g_file_stack_seq \g_file_current_name_tl
      \tl_gset:Nn \g_file_current_name_tl {#1}
      \tex_expandafter:D \tex_input:D \l_file_name_tl ~
      \seq_gpop:NN \g_file_stack_seq \g_file_current_name_tl
    }
}
\cs_generate_variant:Nn \file_input:n { V }
```

\file_path_include:n
\file_path_remove:n

Wrapper functions to manage the search path.

```
\cs_new_protected_nopar:Npn \file_path_include:n #1 {
  \seq_put_right:Nn \l_file_search_path_seq {#1}
```

```
  \seq_remove_duplicates:N \l_file_search_path_seq
}
\cs_new_protected_nopar:Npn \file_path_remove:n #1 {
  \seq_remove_element:Nn \l_file_search_path_seq {#1}
}
```

\file_list:
\file_list_aux:n

A function to list all files used to the log.

```
\cs_new_protected_nopar:Npn \file_list: {
  \seq_remove_duplicates:N \g_file_record_seq
  \iow_log:n { *~File~List~* }
  \seq_map_function:NN \g_file_record_seq \file_list_aux:n
  \iow_log:n { ************* }
}
\cs_new_protected_nopar:Npn \file_list_aux:n #1 { \iow_log:n {#1} }
```

When used as a package, there is a need to hold onto the standard file list as well as the new one here.

```
⟨/initex | package⟩
⟨*package⟩
\AtBeginDocument{
  \clist_map_inline:Nn \@filelist
    { \seq_put_right:Nn \g_file_record_seq {#1} }
}
⟨/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

\! . . . 1531
\ . . . 2128, 2129, 2141, 2142, 5281, 5283
* . . . 2368, 2370, 2374, 2379
\, . . . 5880, 5882
\- . . . 63
\.bool_gset:N . . . 6315
\.bool_set:N . . . 6315
\.choice: . . . 6321
\.choice_code:n . . . 6324
\.choice_code:x . . . 6324
\.code:n . . . 6330
\.code:x . . . 6330
\.default:V . . . 6336
\.default:n . . . 6336
\.dim_gset:N . . . 6342
\.dim_gset:c . . . 6342
\.dim_set:N . . . 6342
\.dim_set:c . . . 6342
\.generate_choices:n . . . 6354
\.int_gset:N . . . 6357
\.int_gset:c . . . 6357
\.int_set:N . . . 6357
\.int_set:c . . . 6357
\.meta:n . . . 6369
\.meta:x . . . 6369
\.skip_gset:N . . . 6375
\.skip_gset:c . . . 6375
\.skip_set:N . . . 6375
\.skip_set:c . . . 6375
\.tl_gset:N . . . 6387
\.tl_gset:c . . . 6387
\.tl_gset_x:N . . . 6387
\.tl_gset_x:c . . . 6387
\.tl_set:N . . . 6387
\.tl_set:c . . . 6387
\.tl_set_x:N . . . 6387
\.tl_set_x:c . . . 6387
\.value_forbidden: . . . 6411
\.value_required: . . . 6411
\/ . . . 62
\: . . . 588, 592, 1021, 2549
\::: . . . *183*, 1473, 1476, 1478, 1478–1480, 1482, 1483, 1485, 1488, 1491, 1497, 1501, 1509, 1552–1559, 1561–1568, 1570–1572, 1574, 1576–1578, 1580–1583, 1585–1588, 1590–1595, 1597, 1598, 1691, 1695, 1698, 1702, 1708–1710, 1712, 1718
\::N . . . *183*, 1482, 1482, 1556–1559, 1578, 1580–1583, 1590, 1591, 1718
\::V . . . *183*, 1501, 1501, 1554, 1558, 1567, 1574, 1578
\::V_unbraced 1690, 1698, 1709, 1712, 1718
\::c . . . *183*, 1485, 1485, 1561, 1564, 1570, 1585, 1590–1595, 1712
\::f *183*, 1491, 1491, 1552, 1556, 1562–1565
\::f_unbraced . . . 1690, 1691, 1708
\::n . . . *183*, 1479, 1479, 1565–1568, 1580, 1581, 1585–1588, 1595
\::o . . . *183*, 1488, 1488, 1562, 1566, 1570–1572, 1576, 1580, 1582, 1583, 1586, 1588, 1591–1593, 1597, 1598
\::o_unbraced . . . 1690, 1695
\::v . . . *183*, 1501, 1509, 1553, 1557
\::v_unbraced . . . 1690, 1702, 1710
\::x . . . *183*, 1497, 1497, 1555, 1559, 1561, 1568, 1572, 1576, 1577, 1581, 1583, 1587, 1588, 1594, 1595, 1597
\= . . . 5879, 5881
\? . . . 1649, 2462, 2549
\@ . . . 718, 1021, 1022, 4018, 4019
\@@end . . . 691
\@@hyph . . . 695
\@@input . . . 689
\@@italiccorr . . . 694
\@@underline . . . 690
\@addtofilelist . . . 7035
\@currext . . . 717
\@currname . . . 716, 6963
\@currnamestack . . . 711, 719
\@declaredoptions . . . 714
\@elt . . . 6921
\@empty . . . 713, 714
\@filelist . . . 7062

\@gobble 697
\@ifpackageloaded 731
\@nil 711, 715
\@nocounterr 6931
\@p@pfilename 711, 715
\@popfilename *136*, 664, 671, 672
\@pushfilename *136*, 664, 664, 669
\@stpelt 6921
\@tempboxa 5662
\@unknownoptionerror 712
\\ 1150, 1152, 1156, 1164, 1165, 1649, 2462, 5131, 5138, 5289, 5292, 5315, 5322, 5556, 5563, 5571, 6032, 6420, 6427, 6435, 6436, 6442, 6449, 6456, 6462, 6468
\{ 4440, 4618, 4797, 4799
\} 4440, 4618, 4797, 4799, 5282
_ 587, 591, 2141
_cs_generate_variant_aux:Ncpx 1647, 1659, 1665
_cs_generate_variant_aux:ccpx 1625, 1647, 1664
_cs_generate_variant_aux:w 1647, 1661, 1666
_l_tl_replace_tl . . 4090, 4090, 4099, 4101, 4104, 4126, 4130, 4132, 4142
_tl_if_single_aux:w . . . 3770, 3781, 3785
_tl_replace_all_in_aux:NNnn 4119, 4120, 4123, 4125
_tl_replace_in_aux:NNnn 4090, 4092, 4094, 4115
_tl_tmp:w 4095, 4100, 4103, 4109, 4127, 4133, 4138
\| 2967

Numbers

\8 5881
\9 5882

\␣ 61, 1532, 5165, 5229, 5281, 5282, 5284, 5290, 5316, 5323, 5765

A

\A 2463, 2551, 4018, 4020
\a 1013
\above 182
\abovedisplayshortskip 195
\abovedisplayskip 196
\abovewithdelims 183
\accent 233
\addtocounter *133*, 6887, 6934, 6940
\addtolength *133*, 6883, 6885
\adjdemerits 270
\advance 77
\afterassignment 87
\aftergroup 88
\alloc_new:nnnN 2792, 3306, 3428, 3494, 4207, 5590
\alloc_reg:NnNN 4956, 4960
\alloc_setup_type:nnn 4954, 4958
\AtBeginDocument 5798, 6938, 7061
\atop 184
\atopwithdelims 185

B

\B 4019, 4021
\badness 332
\baselineskip 260
\batchmode 153
\begin 5813
\begingroup 2, 91
\beginL 445
\beginR 447
\belowdisplayshortskip 197
\belowdisplayskip 198
\binoppenalty 221
\bool_(:w 1919
\bool_)_0:w 1919
\bool_)_1:w 1919
\bool_8_0:w 1919
\bool_8_1:w 1919
\bool_:w 1919
\bool_choose:NN 1919, 1958, 1961
\bool_cleanup:N . . 1919, 1954, 1956, 1957
\bool_do_until:cn *36*, 1911
\bool_do_until:Nn *36*, 1911, 1915, 1916, 1918
\bool_do_until:nn . . *39*, 1996, 2005, 2006
\bool_do_while:cn *36*, 1911
\bool_do_while:Nn *36*, 1911, 1911, 1912, 1914
\bool_do_while:nn . . *39*, 1996, 2002, 2003
\bool_eval_skip_to_end:Nw 1968, 1969, 1970, 1970, 1979
\bool_eval_skip_to_end_aux:Nw 1970, 1971, 1973
\bool_eval_skip_to_end_auxii:Nw 1970, 1976, 1978
\bool_get_next:N 1919, 1921, 1923, 1943, 1954, 1962, 1963
\bool_gset:cn *38*, 1981

.bool_gset:N 124
\bool_gset:Nn 38, 1981, 1982, 1986
\bool_gset_eq:cc 36, 1886, 1893
\bool_gset_eq:cN 36, 1886, 1892
\bool_gset_eq:Nc 36, 1886, 1891
\bool_gset_eq:NN 36, 1886, 1890
\bool_gset_false:c 36, 1876, 1885
\bool_gset_false:N 36, 1876, 1884
\bool_gset_true:c 36, 1876, 1883
\bool_gset_true:N 36, 1876, 1882
\bool_I_0:w 1919
\bool_I_1:w 1919
\bool_if:cTF 36, 1896
\bool_if:N 1896
\bool_if:n 1992
\bool_if:NF 1902, 1908, 1916, 3065
\bool_if:nF 2000, 2006
\bool_if:NT 1901, 1904, 1912, 3065, 3066, 5914, 6294
\bool_if:nT 1997, 2003
\bool_if:NTF 36, 1306, 1896, 1900, 3057, 5985, 6162, 6266, 6278
\bool_if:nTF 37, 1919, 2678, 3410
\bool_if_p:c 36, 1896
\bool_if_p:N 36, 1896, 1899
\bool_if_p:n 37, 1919, 1919, 1981, 1983, 1987, 1989, 1993
\bool_new:c 35, 1876, 1877
\bool_new:N 35, 1876, 1876, 1894, 1895, 5850, 6056, 6070
\bool_Not:N 1937, 1945
\bool_Not:w 1919, 1932, 1942
\bool_not_p:n 38, 1987, 1987
\bool_p:w 1919, 1946
\bool_S_0:w 1919
\bool_S_1:w 1919
\bool_set:cn 38, 1981
.bool_set:N 124
\bool_set:Nn 38, 1981, 1981, 1985
\bool_set_eq:cc 36, 1886, 1889
\bool_set_eq:cN 36, 1886, 1888
\bool_set_eq:Nc 36, 1886, 1887
\bool_set_eq:NN 36, 1886, 1886
\bool_set_false:c 36, 1876, 1881
\bool_set_false:N 36, 1876, 1880, 6147, 6258
\bool_set_true:c 36, 1876, 1879
\bool_set_true:N 36, 1876, 1878, 5851, 6143, 6254
\bool_until_do:cn 36, 1903
\bool_until_do:Nn 36, 1903, 1907, 1908, 1910
\bool_until_do:nn . . 39, 1996, 1999, 2000
\bool_while_do:cn 36, 1903
\bool_while_do:Nn 36, 1903, 1903, 1904, 1906
\bool_while_do:nn . . 39, 1996, 1996, 1997
\bool_xor_p:nn 38, 1988, 1988
\botmark 168
\botmarks 394
\box 376
\box_clear:c 115, 5645
\box_clear:N 115, 5645, 5645–5647
\box_dp:c 115, 5649
\box_dp:N 115, 5649, 5650, 5653, 6731, 6732
\box_gclear:c 115, 5645
\box_gclear:N 115, 5645, 5647, 5648
\box_gset_eq:cc 114, 5634
\box_gset_eq:cN 114, 5634
\box_gset_eq:Nc 114, 5634
\box_gset_eq:NN . . . 114, 5634, 5634, 5635
\box_gset_to_last:c 114, 5637
\box_gset_to_last:N 114, 5637, 5639, 5640
\box_ht:c 115, 5649
\box_ht:N 115, 5649, 5649, 5652, 6730, 6732
\box_if_empty:cTF 114, 5625
\box_if_empty:N 5625
\box_if_empty:NF 5631
\box_if_empty:NT 5630
\box_if_empty:NTF 114, 5625, 5629
\box_if_empty_p:c 114, 5625
\box_if_empty_p:N 114, 5625, 5628
\box_if_horizontal:cTF 113, 5611
\box_if_horizontal:N 5611
\box_if_horizontal:NF 5620
\box_if_horizontal:NT 5619
\box_if_horizontal:NTF . 113, 5611, 5618
\box_if_horizontal_p:c 113, 5611
\box_if_horizontal_p:N . 113, 5611, 5617
\box_if_vertical:cTF 114, 5611
\box_if_vertical:N 5614
\box_if_vertical:NF 5624
\box_if_vertical:NT 5623
\box_if_vertical:NTF . . . 114, 5611, 5622
\box_if_vertical_p:c 114, 5611
\box_if_vertical_p:N . . . 114, 5611, 5621
\box_move_down:nn 114, 5641, 5644
\box_move_left:nn 114, 5641, 5641
\box_move_right:nn 114, 5641, 5642
\box_move_up:nn 114, 5641, 5643
\box_new:c 113, 5589
\box_new:N 113, 5589, 5594, 5606, 5663–5665
\box_new_local:c 113, 5589

\box_new_local:N . . 113, 5589, 5598, 5607
\box_set_eq:cc 114, 5632
\box_set_eq:cN 114, 5632
\box_set_eq:Nc 114, 5632
\box_set_eq:NN 114, 5632, 5632–5634, 5645
\box_set_to_last:c 114, 5637
\box_set_to_last:N . 114, 5637, 5637–5639
\box_show:c 115, 5659
\box_show:N 115, 5659, 5659, 5660
\box_use:c 115, 5655
\box_use:N 115, 5655, 5657, 5658
\box_use_clear:c 115, 5655
\box_use_clear:N . . 115, 5655, 5655, 5656
\box_wd:c 115, 5649
\box_wd:N 115, 5649, 5651, 5654, 6729
\boxmaxdepth 338
\brokenpenalty 295

C

\C 1655, 2463, 2552
\c_active_char_token 48, 2365, 2380, 2423
\c_alignment_tab_token 48, 2365, 2371, 2395
\c_eight . 58, 2292, 2308, 3014, 3024, 4930
\c_eleven 58, 2295, 2311, 3014, 3027, 4933
\c_empty_box . . 115, 5645, 5661, 5661, 5663
\c_empty_tl 77, 3599, 3601, 3712, 3712, 3723, 4397, 4563
\c_empty_toks 84, 4266, 4358, 4358
\c_false_bool 11, 954, 974, 1010, 1011, 1035, 1282, 1291, 1876, 1877, 1880, 1881, 1884, 1885, 1948, 1964, 1966, 1968, 1990, 2505, 2599, 3049, 3052, 3058, 4183
\c_fifteen 58, 2299, 2315, 3014, 3031, 4937
\c_five . . 58, 2289, 2305, 3014, 3021, 4927
\c_four . . 58, 2288, 2304, 3014, 3020, 4926
\c_fourteen 58, 2298, 2314, 3014, 3030, 4936
\c_group_begin_token 48, 2365, 2365, 2383, 2679, 5678, 5704
\c_group_end_token . . . 48, 2365, 2366, 2387, 2680, 5679, 5682, 5707, 5712
\c_hundred_one 58, 3014, 3034
\c_io_streams_tl 105
\c_ior_log_stream . 105, 4916, 4919, 4975
\c_ior_streams_tl 4920, 4939, 4994
\c_ior_term_stream 105, 4916, 4917
\c_iow_log_stream 105, 4916, 4918, 4971, 5146, 5147
\c_iow_streams_tl 4920, 4920, 4939, 4981
\c_iow_term_stream 105, 4916, 4916, 5148, 5149
\c_job_name_tl 76, 3710, 3710, 3711
\c_keys_properties_root_tl 132, 6045, 6046, 6153, 6239, 6242
\c_keys_root_tl 132, 6045, 6045, 6075, 6095, 6098, 6117, 6122, 6127–6129, 6132–6134, 6188, 6196, 6204, 6245, 6289, 6295, 6297, 6303
\c_keys_value_forbidden_tl 132, 6047, 6047
\c_keys_value_required_tl 132, 6047, 6048
\c_KV_single_equal_sign_tl 122, 5843, 5845, 5956, 6007
\c_letter_token 48, 2365, 2377, 2415
\c_luatex_is_engine_bool 26, 1279, 1289, 1291, 1298
\c_math_shift_token . 48, 2365, 2369, 2391
\c_math_subscript_token 48, 2365, 2375, 2407
\c_math_superscript_token 48, 2365, 2373, 2403
\c_max_dim 68, 3473, 3474
\c_max_int 58, 3044, 3044
\c_max_register_int 58, 2792, 2999, 3297, 3297, 3306, 3428, 3494, 4207, 5590
\c_max_skip 65, 3399, 3402, 3403, 3407, 3474
\c_minus_one . 15, 58, 1099, 1100, 1103, 1104, 1109, 2827, 2845, 2847, 2997, 3014, 3014, 3015, 4029, 4045, 4055, 4063, 4918, 4919, 5101, 5113, 5194
\c_msg_coding_error_text_tl 112, 5199, 5199, 5555, 6433
\c_msg_error_tl 112, 5195, 5196, 5428, 5526
\c_msg_fatal_text_tl 112, 5199, 5203, 5423, 5520
\c_msg_fatal_tl 112, 5195, 5195, 5417, 5419, 5422, 5513
\c_msg_help_text_tl 112, 5199, 5206, 5298
\c_msg_hide_tl 5233, 5299
\c_msg_hide_tl<spaces> 5228
\c_msg_info_tl 112, 5195, 5198, 5452
\c_msg_kernel_bug_more_text_tl 112, 5199, 5212, 5580
\c_msg_kernel_bug_text_tl 112, 5199, 5209, 5577
\c_msg_more_text_prefix_tl 112, 5237, 5238, 5268, 5273, 5435, 5437, 5535, 5537

\c_msg_no_info_text_tl *112*, 5199, 5218, 5308, 5440, 5540
\c_msg_on_line_tl . *112*, 5236, 5236, 5251
\c_msg_return_text_tl *112*, 5199, 5216, 5220, 5222, 5558, 5566, 5573
\c_msg_text_prefix_tl *112*, 5237, 5237, 5258, 5262, 5267, 5272, 5379, 5420, 5431, 5446, 5453, 5459, 5465, 5516, 5529
\c_msg_warning_tl . *112*, 5195, 5197, 5445
\c_nine . . *58*, 2293, 2309, 3014, 3025, 4931
\c_one *58*, 2285, 2301, 2822, 2844, 2846, 3014, 3017, 4923, 6528, 6531, 6534, 6537, 6540, 6543
\c_other_char_token . *48*, 2365, 2378, 2419
\c_parameter_token *48*, 2365, 2372
\c_peek_true_remove_next_tl *214*, 2648, 2659, 2660
\c_seven *58*, 1757, 2291, 2307, 3014, 3023, 4929
\c_six *58*, 1753, 2290, 2306, 3014, 3022, 4928
\c_sixteen *15*, *58*, 1099, 1106, 1112, 3014, 3032, 4916, 4917, 4954, 4958, 4980, 4982, 4993, 4995, 5025, 5046, 5065, 5086
\c_space_tl *77*, 2224, 2246, 3713, 3713, 4440, 4618, 4797–4799, 5252, 5290, 5316, 5323, 5419, 5422, 5430, 5433, 5445, 5448, 5452, 5455, 5515, 5519, 5528, 5532, 6434
\c_space_token *48*, 2365, 2376, 2411, 2681, 2772
\c_ten *58*, 2294, 2310, 3014, 3026, 4932, 5765
\c_ten_thousand *58*, 3014, 3038
\c_ten_thousand_four *58*, 3014, 3042
\c_ten_thousand_one *58*, 3014, 3039
\c_ten_thousand_three . . . *58*, 3014, 3041
\c_ten_thousand_two *58*, 3014, 3040
\c_thirteen *58*, 2297, 2313, 3014, 3029, 4935
\c_thirty_two *58*, 3014, 3033
\c_thousand *58*, 3014, 3037
\c_three *58*, 2287, 2303, 3014, 3019, 4925, 6564, 6568, 6572, 6576, 6580, 6584, 6747
\c_true_bool . *11*, 954, 974, 1010, 1010, 1039, 1280, 1289, 1878, 1879, 1882, 1883, 1947, 1950, 1965, 1967, 1969, 1990, 3046, 3049, 3052, 3059, 4181
\c_twelve *58*, 2129, 2142, 2296, 2312, 3014, 3028, 4934
\c_twenty_thousand *58*, 3014, 3043
\c_two *58*, 2286, 2302, 3014, 3018, 4924, 6546, 6549, 6552, 6555, 6558, 6561
\c_two_ats_with_two_catcodes_tl 4017, 4024, 4028, 4039, 4062
\c_twohundred_fifty_five . *58*, 3014, 3035
\c_twohundred_fifty_six . *58*, 3014, 3036
\c_undefined 44, 50, 5273, 6245
\c_undefined:D . . . 1268, 1271, 1274, 1277
\c_xetex_is_engine_bool *26*, 1279, 1280, 1282, 1285, 1294
\c_zero *15*, *58*, 953, 958, 963, 968, 973, 978, 983, 988, 1081, 1087, 1099, 1107, 1750, 1751, 1755, 1804, 1831, 1855, 2215, 2237, 2284, 2300, 2852, 2854, 3014, 3016, 3072, 3206, 3236, 3240, 3241, 3247, 3306, 3411, 3412, 3428, 3494, 3834, 4207, 4922, 4954, 4958, 5590, 6165, 6510, 6513, 6516, 6519, 6522, 6525
\c_zero_dim *68*, 3473, 3473, 5684, 6724
\c_zero_skip *65*, 3339, 3399, 3400, 3401, 3406, 3418, 3419, 3449, 3473, 5714
\calc_add: *354*, 6616, 6665, 6671
\calc_add_A_to_B: . *354*, 6665, 6671, 6674
\calc_addtocounter:nn *133*, 6887, 6897, 6934, 6940
\calc_assign_generic:NNNNnn *354*, 6494, 6494, 6510, 6513, 6516, 6519, 6522, 6525, 6528, 6531, 6534, 6537, 6540, 6543, 6546, 6549, 6552, 6555, 6558, 6561, 6564, 6568, 6572, 6576, 6580, 6584
\calc_calculate_box_size:nnn *134*, 6722, 6722, 6733
\calc_calculate_box_size_aux:n 6722, 6724, 6726
\calc_calculated_ratio: 6754, 6764
\calc_chk_document_counter:nn *354*, 6887, 6893, 6903, 6918, 6930
\calc_close: *354*, 6605, 6610, 6620
\calc_depthof:n . . . *133*, 6846, 6852, 6877
\calc_depthof_aux:n 6731, 6736, 6846, 6853, 6862, 6863, 6865
\calc_depthof_auxi:n . . . 6736, 6846, 6865
\calc_dim_add:Nn *134*, 6527, 6533
\calc_dim_gadd:Nn *134*, 6527, 6536
\calc_dim_gset:Nn *134*, 6527, 6530
\calc_dim_gsub:Nn *134*, 6527, 6542
\calc_dim_muskip:Nn 6751, 6752

\calc_dim_set:Nn *134*, 6527, 6527, 6748, 6749, 6799, 6800
\calc_dim_sub:Nn *134*, 6527, 6539
\calc_divide: *354*, 6659, 6692, 6720
\calc_divide:N *354*, 6619, 6633, 6649
\calc_divide_B_by_A: *354*, 6692, 6709, 6721, 6832
\calc_error:N *354*, 6624, 6833, 6833
\calc_generic_add_or_subtract:N *354*, 6665, 6665, 6671, 6673
\calc_generic_multiply_or_divide:N *354*, 6692, 6692, 6719, 6721
\calc_heightof:n . . *133*, 6846, 6849, 6879
\calc_heightof_aux:n 6730, 6738, 6846, 6850, 6866, 6867, 6869
\calc_heightof_auxi:n . . 6738, 6846, 6869
\calc_init_B: *354*, 6601, 6602, 6605, 6605, 6669
\calc_int_add:Nn *134*, 6509, 6515
\calc_int_gadd:Nn . *134*, 6509, 6518, 6904
\calc_int_gset:Nn . *134*, 6509, 6512, 6894
\calc_int_gsub:Nn *134*, 6509, 6524
\calc_int_set:Nn *134*, 6509, 6509, 6793, 6794
\calc_int_sub:Nn *134*, 6509, 6521
\calc_maxmin_div_or_mul:NNnn *354*, 6784, 6819, 6830, 6832
\calc_maxmin_divide: 6651, 6831
\calc_maxmin_generic:Nnn *354*, 6784, 6786, 6790, 6824
\calc_maxmin_multiply: *354*, 6635, 6784, 6829
\calc_maxmin_operation:Nnn *354*, 6593, 6634, 6650, 6784, 6784, 6839, 6842
\calc_maxof:nn *133*, 6838, 6838, 6844
\calc_minof:nn *133*, 6838, 6841, 6845
\calc_multiply: . . . *354*, 6643, 6692, 6718
\calc_multiply:N . . *354*, 6618, 6633, 6633
\calc_multiply_B_by_A: *354*, 6692, 6700, 6719, 6830
\calc_muskip_add:Nn *134*, 6563, 6571
\calc_muskip_gadd:Nn . . . *134*, 6563, 6575
\calc_muskip_gset:Nn . . . *134*, 6563, 6567
\calc_muskip_gsub:Nn . . . *134*, 6563, 6583
\calc_muskip_set:Nn *134*, 6563, 6563, 6811, 6812
\calc_muskip_sub:Nn *134*, 6563, 6579
\calc_next:w 6615–6621, 6623, 6632, 6635, 6638, 6641, 6643, 6647, 6651, 6654, 6657, 6659, 6663
\calc_numeric: *354*, 6595, 6605, 6606
\calc_open:w . . *354*, 6504, 6589, 6600, 6600
\calc_post_scan:N *354*, 6607, 6613, 6614, 6614, 6621, 6743, 6766, 6781, 6788, 6827
\calc_pre_scan:N *354*, 6587, 6587, 6603, 6670, 6698
\calc_ratio:nn *133*, 6500, 6874, 6874, 6881
\calc_ratio_divide:nn *354*, 6654, 6745, 6767
\calc_ratio_multiply:nn . . *354*, 6637, 6638, 6653, 6745, 6745, 6767, 6875
\calc_real:n . . *133*, 6499, 6874, 6876, 6882
\calc_real_divide:n *354*, 6657, 6768, 6783
\calc_real_evaluate:nn . . . *354*, 6640, 6656, 6768, 6768, 6782, 6783, 6876
\calc_real_multiply:n *354*, 6641, 6768, 6782
\calc_setcounter:nn *133*, 6887, 6892, 6933, 6939
\calc_skip_add:Nn . *134*, 6545, 6551, 6885
\calc_skip_gadd:Nn . *134*, 6545, 6554, 6886
\calc_skip_gset:Nn . *134*, 6545, 6548, 6884
\calc_skip_gsub:Nn *134*, 6545, 6560
\calc_skip_set:Nn *134*, 6545, 6545, 6805, 6806, 6883
\calc_skip_sub:Nn *134*, 6545, 6557
\calc_stepcounter:n *133*, 6887, 6912, 6935, 6941
\calc_subtract: . . . *354*, 6617, 6665, 6672
\calc_subtract_A_from_B: *354*, 6665, 6673, 6683
\calc_textsize:Nn *354*, 6591, 6727, 6727, 6847, 6850, 6853, 6856
\calc_totalheightof:n *133*, 6846, 6855, 6880
\calc_totalheightof_aux:n 6732, 6739, 6846, 6856, 6870, 6871, 6873
\calc_totalheightof_auxi:n 6739, 6846, 6873
\calc_widthof:n . . . *133*, 6846, 6846, 6878
\calc_widthof_aux:n 6729, 6737, 6846, 6847, 6858, 6859, 6861
\calc_widthof_auxi:n . . . 6737, 6846, 6861
\catcode . 26–28, 30, 31, 34–36, 38, 39, 380
\char 234
\char_gset_mathcode:nn . . *48*, 2316, 2321
\char_gset_mathcode:w *48*, 2316, 2320, 2322
\char_make_active:N . *46*, 2284, 2297, 5283
\char_make_active:n *46*, 2300, 2313
\char_make_alignment:N . . *46*, 2284, 2288
\char_make_alignment:n . . *46*, 2300, 2304
\char_make_begin_group:N . *46*, 2284, 2285
\char_make_begin_group:n . *46*, 2300, 2301

\char_make_comment:N 46, 2284, 2298
\char_make_comment:n 46, 2300, 2314
\char_make_end_group:N . . 46, 2284, 2286
\char_make_end_group:n . . 46, 2300, 2302
\char_make_end_line:N . . . 46, 2284, 2289
\char_make_end_line:n . . . 46, 2300, 2305
\char_make_escape:N 46, 2284, 2284
\char_make_escape:n 46, 2300, 2300
\char_make_ignore:N 46, 2284, 2293
\char_make_ignore:n 46, 2300, 2309
\char_make_invalid:N 46, 2284, 2299
\char_make_invalid:n 46, 2300, 2315
\char_make_letter:N 46, 2284, 2295, 5229, 5284
\char_make_letter:n 46, 2300, 2311
\char_make_math_shift:N . 46, 2284, 2287
\char_make_math_shift:n . 46, 2300, 2303
\char_make_math_subscript:N 46, 2284, 2292
\char_make_math_subscript:n 46, 2300, 2308
\char_make_math_superscript:N 46, 2284, 2291
\char_make_math_superscript:n 46, 2300, 2307
\char_make_other:N 46, 2284, 2296
\char_make_other:n 46, 2300, 2312
\char_make_parameter:N . . 46, 2284, 2290
\char_make_parameter:n . . 46, 2300, 2306
\char_make_space:N 46, 2284, 2294
\char_make_space:n 46, 2300, 2310
\char_set_catcode:nn 45, 2270, 2271, 2284–2315, 2368, 2370, 2374, 2379, 2464, 2550–2554, 5765, 5879, 5880, 5892
\char_set_catcode:w . 45, 2270, 2270, 2272
\char_set_lccode:nn . . 47, 2332, 2333, 2457–2462, 2549, 5281, 5881, 5882
\char_set_lccode:w 47, 2332, 2332, 2334, 5282
\char_set_mathcode:nn . . . 48, 2316, 2317
\char_set_mathcode:w 48, 2316, 2316, 2318
\char_set_sfcode:nn 47, 2352, 2353
\char_set_sfcode:w . . 47, 2352, 2352, 2354
\char_set_uccode:nn 47, 2342, 2343
\char_set_uccode:w . . 47, 2342, 2342, 2344
\char_show_value_catcode:n 45, 2270, 2281
\char_show_value_catcode:w 45, 2270, 2278, 2282
\char_show_value_lccode:n 47, 2332, 2340
\char_show_value_lccode:w 47, 2332, 2339, 2341
\char_show_value_mathcode:n 48, 2316, 2329
\char_show_value_mathcode:w 48, 2316, 2328, 2330
\char_show_value_sfcode:n 47, 2352, 2361
\char_show_value_sfcode:w 47, 2352, 2360, 2362
\char_show_value_uccode:n 47, 2342, 2350
\char_show_value_uccode:w 47, 2342, 2349, 2351
\char_value_catcode:n . . . 45, 2270, 2275
\char_value_catcode:w 45, 2270, 2274, 2276
\char_value_lccode:n 47, 2332, 2337
\char_value_lccode:w 47, 2332, 2336, 2337
\char_value_mathcode:n . . 48, 2316, 2325
\char_value_mathcode:w 48, 2316, 2324, 2326
\char_value_sfcode:n 47, 2352, 2357
\char_value_sfcode:w 47, 2352, 2356, 2358
\char_value_uccode:n 47, 2342, 2347
\char_value_uccode:w 47, 2342, 2346, 2347
\chardef 69
\chk_exist_cs:N 3585, 3589
\chk_global:N 3590, 4244, 4254
\chk_if_exist_cs:c 11, 1139, 1146
\chk_if_exist_cs:N 11, 1139, 1139, 1147, 1611
\chk_if_free_cs:c 11, 1121, 1136, 5258, 5262, 5340
\chk_if_free_cs:N 11, 1121, 1121, 1137, 1178, 1250, 1261, 2796, 2800, 3005, 3310, 3314, 3392, 3432, 3436, 3498, 3502, 3524, 3532, 4211, 4215, 5595, 5599
\chk_local:N 4227, 4249
\chk_local_or_pref_global:N . . 2811, 2824, 2829, 2859, 2865, 3326, 3341, 3356, 3363, 3586, 4267, 4301, 4305
\chk_var_or_const:N 3526, 3533, 3586, 3590, 3605, 3617, 4250, 4255
.choice: 124
.choice_code:n 125
.choice_code:x 125
\cleaders 252
\clearpage 5817, 5820, 5825, 5828
\clist_clear:c 91, 4544
\clist_clear:N 91, 4544, 4544, 4545, 4715, 4737, 5364
\clist_clear_new:c 91, 4548

\clist_clear_new:N . . 91, 4548, 4548, 4549
\clist_concat:ccc 95, 4700
\clist_concat:NNN . . 95, 4700, 4709, 4711
\clist_concat_aux:NNNN 4700, 4700, 4709, 4710
\clist_display:c 93, 4609
\clist_display:N . . . 93, 4609, 4609, 4623
\clist_gclear:c 91, 4544
\clist_gclear:N 91, 4544, 4546, 4547
\clist_gclear_new:c 91, 4548
\clist_gclear_new:N . 91, 4548, 4550, 4551
\clist_gconcat:ccc 95, 4700
\clist_gconcat:NNN . . 95, 4700, 4710, 4712
\clist_get:cN 93, 4591, 4760
\clist_get:NN . . 93, 4591, 4591, 4596, 4759
\clist_get_aux:w . . . 97, 4591, 4593, 4595
\clist_gpop:cN 96, 4753
\clist_gpop:NN 96, 4753, 4757, 4758
\clist_gpush:cn 96, 4753
\clist_gpush:Nn 96, 4753, 4753, 4756
\clist_gpush:No 96, 4753, 4755
\clist_gpush:NV 96, 4753, 4754
\clist_gput_left:cn 92, 4633
\clist_gput_left:co 92, 4633
\clist_gput_left:cV 92, 4633
\clist_gput_left:Nn 92, 4633, 4633, 4636, 4753
\clist_gput_left:No 92, 4633, 4755
\clist_gput_left:NV 92, 4633, 4754
\clist_gput_left:Nx 92, 4633
\clist_gput_right:cn 92, 4641
\clist_gput_right:co 92, 4641
\clist_gput_right:cV 92, 4641
\clist_gput_right:Nn 92, 4641, 4641, 4644
\clist_gput_right:No 92, 4641
\clist_gput_right:NV 92, 4641
\clist_gput_right:Nx 92, 4641
\clist_gremove_duplicates:N 95, 4714, 4727
\clist_gremove_element:Nn 95, 4730, 4733
\clist_gset_eq:cc 91, 4556, 4559
\clist_gset_eq:cN 91, 4556, 4557
\clist_gset_eq:Nc 91, 4556, 4558
\clist_gset_eq:NN 91, 4556, 4556, 4728, 4734
\clist_if_empty:c 4561
\clist_if_empty:cTF 94, 4560
\clist_if_empty:N 4560
\clist_if_empty:NF 4646, 4665
\clist_if_empty:NTF 94, 4560, 4625
\clist_if_empty_err:N 96, 4562, 4562, 4592, 4598
\clist_if_empty_p:c 94, 4560
\clist_if_empty_p:N 94, 4560
\clist_if_eq:cc 4571
\clist_if_eq:ccTF 95, 4568
\clist_if_eq:cN 4569
\clist_if_eq:cNTF 95, 4568
\clist_if_eq:Nc 4570
\clist_if_eq:NcTF 95, 4568
\clist_if_eq:NN 4568
\clist_if_eq:NNTF 95, 4568
\clist_if_eq_p:cc 94, 4568
\clist_if_eq_p:cN 94, 4568
\clist_if_eq_p:Nc 94, 4568
\clist_if_eq_p:NN 94, 4568
\clist_if_in:cnTF 95, 4572
\clist_if_in:coTF 95, 4572
\clist_if_in:cVTF 95, 4572
\clist_if_in:Nn 4572
\clist_if_in:NnF 4581, 4720, 5276
\clist_if_in:NnT 4580
\clist_if_in:NnTF . . 95, 4572, 4579, 5368
\clist_if_in:NoTF 95, 4572
\clist_if_in:NVTF 95, 4572
\clist_map_break: 94, 4662, 4662
\clist_map_function:cN 93, 4645
\clist_map_function:NN 93, 4645, 4645, 4651, 4716, 4743
\clist_map_function:nN 93, 4645, 4652, 6104
\clist_map_function_aux:Nw . . . 4647, 4654, 4657, 4657, 4660, 4669, 4681
\clist_map_inline:cn 93, 4663
\clist_map_inline:Nn 93, 4613, 4663, 4664, 4675, 6999, 7062
\clist_map_inline:nn 93, 4663, 4676
\clist_map_variable:cNn 94, 4687
\clist_map_variable:NNn 94, 4687, 4693, 4694
\clist_map_variable:nNn 94, 4687, 4687, 4693
\clist_map_variable_aux:Nnw 4689, 4695, 4695, 4698
\clist_new:c 90, 4542
\clist_new:N 90, 4542, 4542, 4543, 4713, 5242, 5245
\clist_pop:cN 96, 4747
\clist_pop:NN 96, 4747, 4751, 4752
\clist_pop_aux:nnNN 96, 4597, 4597, 4751, 4757

\clist_pop_aux:w . . . *97*, 4597, 4599, 4601
\clist_pop_auxi:w . . *97*, 4597, 4604, 4606
\clist_push:cn *96*, 4747, 4750
\clist_push:Nn *96*, 4747, 4747
\clist_push:No *96*, 4747, 4749
\clist_push:NV *96*, 4747, 4748
\clist_put_aux:NNnnNn *97*, 4624, 4624, 4630, 4634, 4638, 4642
\clist_put_left:cn *91*, 4629, 4750
\clist_put_left:co *91*, 4629
\clist_put_left:cV *91*, 4629
\clist_put_left:Nn *91*, 4629, 4629, 4632, 4747
\clist_put_left:No *91*, 4629, 4749
\clist_put_left:NV *91*, 4629, 4748
\clist_put_left:Nx *91*, 4629
\clist_put_right:cn *92*, 4637
\clist_put_right:co *92*, 4637
\clist_put_right:cV *92*, 4637
\clist_put_right:Nn *92*, 4637, 4637, 4640, 4721, 4740, 5277, 5371
\clist_put_right:No *92*, 4637
\clist_put_right:NV *92*, 4637
\clist_put_right:Nx *92*, 4637
\clist_remove_duplicates:N *95*, 4714, 4724
\clist_remove_duplicates_aux:n 4714, 4716, 4719
\clist_remove_duplicates_aux:NN 4714, 4714, 4725, 4728
\clist_remove_element:Nn . *95*, 4730, 4730
\clist_remove_element_aux:n 4730, 4738, 4743, 4746
\clist_remove_element_aux:NNn 4730, 4731, 4734, 4736
\clist_set_eq:cc *91*, 4552, 4555
\clist_set_eq:cN *91*, 4552, 4553
\clist_set_eq:Nc *91*, 4552, 4554
\clist_set_eq:NN *91*, 4552, 4552, 4725, 4731
\clist_show:c *92*, 4607, 4608
\clist_show:N *92*, 4607, 4607
\clist_tmp:w 4573, 4577
\clist_top:cN *96*, 4759, 4760
\clist_top:NN *96*, 4759, 4759
\clist_use:c *92*, 4582
\clist_use:N *92*, 4582, 4582, 4590
\closein 128
\closeout 123
\clubpenalties 436
\clubpenalty 263
.code:n *125*
.code:x *125*
\copy 320
\count 371
\countdef 70
\cr 95
\crcr 96
\cs:w *11*, 781, 784, 838, 879, 1066, 1365, 1404, 1486, 1528, 1541, 1542, 1544, 1546, 1548–1550, 1609, 1763, 1769, 1772, 2399, 2885, 3390, 3466, 3709, 5760
\cs_end: *11*, 781, 785, 838, 879, 1060, 1066, 1365, 1404, 1486, 1528, 1541, 1542, 1544, 1546, 1548, 1549, 1551, 1609, 1765, 1766, 1775–1777, 1779, 1781, 1783, 1785, 1787, 1789, 1791–1802, 2399, 2885, 3390, 3466, 3709, 5760
\cs_generate_from_arg_count:cNnn 973, 978, 983, 988, 1350, 1403, 6117, 6122
\cs_generate_from_arg_count:NNnn 1321, 1321, 1351, 1364
\cs_generate_from_arg_count_error_msg:Nn 1321, 1344, 1353
\cs_generate_internal_variant:n *28*, 1632, 1677, 1677
\cs_generate_internal_variant_aux:n 1680, 1683, 1683, 1688
\cs_generate_variant:Nn *27*, 1610, 1610, 1899–1902, 1906, 1910, 1914, 1918, 1985, 1986, 2186, 2204, 2252–2259, 2807, 2808, 2820, 2821, 2848–2851, 2853, 2855, 2880–2883, 3070, 3096, 3097, 3321, 3322, 3336, 3337, 3351, 3352, 3359, 3373, 3382, 3385, 3388, 3443, 3444, 3446, 3448, 3450, 3452, 3456, 3458, 3460, 3462, 3464, 3530, 3550, 3552, 3566–3582, 3597, 3598, 3600, 3602, 3614, 3625, 3656–3664, 3695–3702, 3726–3729, 3733–3736, 3741–3748, 3755–3762, 3949, 3961, 3980, 3985, 3999, 4008, 4075–4077, 4084–4089, 4117, 4118, 4144, 4145, 4148, 4149, 4156, 4157, 4161, 4163, 4168, 4169, 4188–4191, 4222, 4223, 4225, 4228, 4243, 4246, 4263, 4264, 4274, 4275, 4284, 4285, 4287, 4292, 4298, 4312, 4326–4328, 4339–4342, 4347–4350, 4393, 4409–4411, 4419, 4445, 4453, 4456,

4475–4477, 4488, 4496, 4502, 4525, 4528, 4529, 4543, 4545, 4547, 4549, 4551, 4579–4581, 4590, 4596, 4623, 4632, 4636, 4640, 4644, 4651, 4675, 4694, 4711, 4712, 4752, 4756, 4758, 4804, 4815, 4819, 4848, 4849, 4861, 4862, 4880–4882, 4894, 4904, 4968, 4969, 4973, 4977, 4990, 5003, 5109, 5121, 5153, 5157, 5168, 5172, 5606, 5607, 5617–5624, 5628–5631, 5633, 5635, 5638, 5640, 5646, 5648, 5652–5654, 5656, 5658, 5660, 5668, 5670, 5674, 5676, 5689, 5691, 5694, 5696, 5700, 5702, 5706, 5711, 5718, 5720, 6136, 6252, 6314, 7027–7029, 7044
\cs_generate_variant_aux:N 1610, 1622, 1627, 1637, 1644, 1645
\cs_generate_variant_aux:nnNn 1610, 1612, 1614
\cs_generate_variant_aux:nnw 1610, 1615, 1617, 1642
\cs_get_arg_count_from_signature:c 1318, 1405
\cs_get_arg_count_from_signature:N 25, 910, 916, 923, 930, 1302, 1302, 1319, 1366, 6164
\cs_get_arg_count_from_signature_aux:nnN 1302, 1303, 1305
\cs_get_arg_count_from_signature_auxii:w 1302, 1312, 1317
\cs_get_function_name:N . 25, 1042, 1042
\cs_get_function_signature:N 25, 1042, 1045
\cs_gnew:cn 18, 1424
\cs_gnew:cpn 17, 1209, 1215
\cs_gnew:cpx 17, 1209, 1216
\cs_gnew:cx 18, 1424
\cs_gnew:Nn 18, 1385
\cs_gnew:Npn 17, 1190, 1192, 1215, 5893, 5896, 5899
\cs_gnew:Npx 17, 1190, 1193, 1216
\cs_gnew:Nx 18, 1385
\cs_gnew_eq:cc 23, 1260, 1266
\cs_gnew_eq:cN 23, 1260, 1264
\cs_gnew_eq:Nc 23, 1260, 1265
\cs_gnew_eq:NN 23, 1260, 1260, 1264–1266, 2364, 4971, 4975
\cs_gnew_nopar:cn 19, 1424
\cs_gnew_nopar:cpn 17, 1198, 1207
\cs_gnew_nopar:cpx 17, 1198, 1208
\cs_gnew_nopar:cx 19, 1424
\cs_gnew_nopar:Nn 19, 1385
\cs_gnew_nopar:Npn . . 17, 1190, 1190, 1207
\cs_gnew_nopar:Npx . . 17, 1190, 1191, 1208
\cs_gnew_nopar:Nx 19, 1385
\cs_gnew_protected:cn 19, 1424
\cs_gnew_protected:cpn . . 17, 1225, 1231
\cs_gnew_protected:cpx . . 17, 1225, 1232
\cs_gnew_protected:cx 19, 1424
\cs_gnew_protected:Nn 19, 1385
\cs_gnew_protected:Npn 17, 1190, 1196, 1231, 5900, 5908
\cs_gnew_protected:Npx 17, 1190, 1197, 1232
\cs_gnew_protected:Nx 19, 1385
\cs_gnew_protected_nopar:cn . . . 19, 1424
\cs_gnew_protected_nopar:cpn 18, 1217, 1223
\cs_gnew_protected_nopar:cpx 18, 1217, 1224
\cs_gnew_protected_nopar:cx . . . 19, 1424
\cs_gnew_protected_nopar:Nn . . . 19, 1385
\cs_gnew_protected_nopar:Npn 18, 1190, 1194, 1223
\cs_gnew_protected_nopar:Npx 18, 1190, 1195, 1224
\cs_gnew_protected_nopar:Nx . . . 19, 1385
\cs_gset:cn 22, 1408
\cs_gset:cpn 20, 1209, 1211, 3964, 3973, 4667, 4679, 4898
\cs_gset:cpx 20, 1209, 1212
\cs_gset:cx 22, 1408
\cs_gset:Nn 22, 1362
\cs_gset:Npn 20, 818, 820, 1192, 1211, 1533
\cs_gset:Npx 20, 818, 823, 1193, 1212
\cs_gset:Nx 22, 1362
\cs_gset_eq:cc . 23, 1256, 1259, 1893, 4389
\cs_gset_eq:cN . . 23, 1256, 1258, 1277, 1883, 1885, 1892, 4387, 5029, 5069
\cs_gset_eq:Nc 23, 1256, 1257, 1891, 4388, 5020, 5038, 5060, 5078
\cs_gset_eq:NN 23, 1256, 1256–1259, 1274, 1882, 1884, 1890, 3589, 3595, 4386
\cs_gset_nopar:cn 22, 1408
\cs_gset_nopar:cpn . . 20, 1198, 1203, 1830
\cs_gset_nopar:cpx 20, 1198, 1204
\cs_gset_nopar:cx 22, 1408
\cs_gset_nopar:Nn 22, 1362
\cs_gset_nopar:Npn 20, 818, 818, 821, 827, 833, 1190, 1203, 2380, 3528

\cs_gset_nopar:Npx . *20*, 818, 819, 824, 830, 836, 1191, 1204, 3534, 3561, 3564, 3642, 3645, 3648, 3651, 3654, 3681, 3684, 3687, 3690, 3693, 6754
\cs_gset_nopar:Nx *22*, 1362
\cs_gset_protected:cn *22*, 1408
\cs_gset_protected:cpn . . *21*, 1225, 1227
\cs_gset_protected:cpx . . *21*, 1225, 1228
\cs_gset_protected:cx *22*, 1408
\cs_gset_protected:Nn *22*, 1362
\cs_gset_protected:Npn *21*, 818, 832, 1196, 1227
\cs_gset_protected:Npx *21*, 818, 835, 1197, 1228
\cs_gset_protected:Nx *22*, 1362
\cs_gset_protected_nopar:cn . . . *23*, 1408
\cs_gset_protected_nopar:cpn *21*, 1217, 1219
\cs_gset_protected_nopar:cpx *21*, 1217, 1220
\cs_gset_protected_nopar:cx . . . *23*, 1408
\cs_gset_protected_nopar:Nn . . . *23*, 1362
\cs_gset_protected_nopar:Npn *21*, 818, 826, 1194, 1219
\cs_gset_protected_nopar:Npx *21*, 818, 829, 1195, 1220
\cs_gset_protected_nopar:Nx . . . *23*, 1362
\cs_gundefine:c *23*, 1267, 1276
\cs_gundefine:N *23*, 1267, 1273, 5105, 5117
\cs_if_do_not_use_aux:nnN 1074, 1075, 1077
\cs_if_do_not_use_p:N *10*, 1074, 1074, 1087
\cs_if_eq:ccF 1455
\cs_if_eq:ccT 1454
\cs_if_eq:ccTF *10*, 1440, 1453
\cs_if_eq:cNF 1447
\cs_if_eq:cNT 1446
\cs_if_eq:cNTF *10*, 1440, 1445
\cs_if_eq:NcF 1451
\cs_if_eq:NcT 1450
\cs_if_eq:NcTF *10*, 1440, 1449
\cs_if_eq:NN 1440
\cs_if_eq:NNF 1447, 1451, 1455
\cs_if_eq:NNT 1446, 1450, 1454
\cs_if_eq:NNTF . *10*, 1440, 1445, 1449, 1453
\cs_if_eq_name:NN 1158
\cs_if_eq_name_p:NN *10*, 1158
\cs_if_eq_p:cc *10*, 1440, 1452
\cs_if_eq_p:cN *10*, 1440, 1444
\cs_if_eq_p:Nc *10*, 1440, 1448
\cs_if_eq_p:NN . *10*, 1440, 1444, 1448, 1452
\cs_if_exist:c 1059
\cs_if_exist:cF 3843
\cs_if_exist:cT 6295
\cs_if_exist:cTF *10*, 1048, 2210, 2233, 5019, 5048, 5059, 5088, 5372, 5379, 5435, 5534, 5750, 6094, 6196
\cs_if_exist:N 1048
\cs_if_exist:NF 1140, 6069, 6307
\cs_if_exist:NT . . 5099, 5111, 6995, 7015
\cs_if_exist:NTF *10*, 1048, 2449, 5177, 6154, 6189
\cs_if_exist_p:c *10*, 1048
\cs_if_exist_p:N *10*, 1048, 1081, 3606, 3618
\cs_if_free:cF 1098
\cs_if_free:cT . . . 1097, 1678, 6127, 6132
\cs_if_free:cTF *10*, 1080, 1096, 1621, 5769, 6931
\cs_if_free:N 1080
\cs_if_free:NF 1098, 1122
\cs_if_free:NT 1097
\cs_if_free:NTF *10*, 1080, 1096, 5151, 5155
\cs_if_free_p:c *10*, 1080, 1095
\cs_if_free_p:N *10*, 1080, 1095
\cs_meaning:c *12*, 781, 787
\cs_meaning:N *12*, 781, 786, 787
\cs_new:cn *18*, 1424
\cs_new:cpn *17*, 1209, 1213, 1775–1778, 1780, 1782, 1784, 1786, 1788, 1790, 1792–1802, 6239, 6242
\cs_new:cpx *17*, 1209, 1214, 1679
\cs_new:cx *18*, 1424
\cs_new:Nn *18*, 1385
\cs_new:Npn *17*, 891, 915, 1176, 1184, 1213, 1472, 1475, 1478, 1479, 1482, 1485, 1488, 1491, 1501, 1509, 1536, 1537, 1539, 1542, 1543, 1545, 1547, 1550, 1599, 1600, 1603, 1606, 1609, 1666, 1683, 1690, 1691, 1695, 1698, 1702, 1714, 1720, 1803, 1809, 1818, 1838, 1846, 1903, 1907, 1911, 1915, 1942, 1945, 1981, 1982, 1987, 1988, 1996, 1999, 2002, 2005, 2008, 2012, 2018, 2022, 2028, 2032, 2038, 2042, 2048, 2175, 2181, 2187, 2196, 2205, 2653, 2656, 2686, 3752, 3785, 3789, 3792, 3795, 3798, 3801, 3804, 3807, 3810, 3813, 3816, 3819, 3822, 3825, 3950, 3957, 3992, 3995, 4003, 4014, 4158, 4160, 4162, 4166, 4167, 4179, 4299, 4370,

4371, 4606, 4652, 4657, 4746, 5174, 5386, 5641–5644, 5922, 5952, 5973, 6014, 6015, 6291, 6494, 6722, 6767, 6819, 6838, 6841, 6846, 6849, 6852, 6855, 6858, 6862, 6866, 6870, 6874
\cs_new:Npx *17*, 1176, 1185, 1214
\cs_new:Nx *18*, 1385
\cs_new_eq:cc *23*, 936, 1249, 1255
\cs_new_eq:cN *23*, 1249, 1253, 1877
\cs_new_eq:Nc *23*, 1249, 1254
\cs_new_eq:NN *23*, 1249, 1249, 1253–1255, 1280, 1282, 1289, 1291, 1876, 1886–1893, 2270, 2316, 2332, 2342, 2352, 2365, 2366, 2788–2790, 2884, 2886, 3146, 3147, 3381, 3384, 3387, 3389, 3463, 3465, 3473, 3474, 3476, 3515, 3594, 3595, 3613, 3624, 3788, 3943–3945, 3991, 4159, 4164, 4165, 4224, 4231, 4245, 4260, 4261, 4286, 4372–4389, 4429, 4430, 4489, 4490, 4520–4523, 4526, 4530, 4531, 4542, 4544, 4546, 4548, 4550, 4552–4559, 4607, 4608, 4662, 4747–4750, 4753–4755, 4759, 4760, 4772–4787, 4905, 4916–4919, 4939, 4951, 5175, 5608–5610, 5636, 5649–5651, 5655, 5657, 5662, 5679, 5682, 5688, 5690, 5707, 5712, 5717, 5719, 6861, 6865, 6869, 6873
\cs_new_nopar:cn *19*, 1424
\cs_new_nopar:cpn *17*, 1198, 1205, 1962–1967
\cs_new_nopar:cpx *17*, 1198, 1206, 1673, 2700
\cs_new_nopar:cx *19*, 1424
\cs_new_nopar:Nn *19*, 1385
\cs_new_nopar:Npn *17*, 1176, 1182, 1199, 1205, 1301, 1444–1455, 1471, 1496, 1659, 1664, 1708–1711, 1717, 1749, 1751, 1752, 1762, 1768, 1771, 1774, 1957, 1961, 2132, 2136, 2145, 2149, 2274, 2275, 2278, 2281, 2324, 2325, 2328, 2329, 2336, 2337, 2339, 2340, 2346, 2347, 2349, 2350, 2356, 2357, 2360, 2361, 2441, 2471, 2478, 2489, 2500, 2511, 2522, 2529, 2536, 2543, 2557, 2560, 2566, 2572, 2598, 2628, 2663, 2670, 2677, 2694, 2768, 2862, 2885, 2887, 2888, 2901, 2905, 2909, 2923, 2932, 2941, 2945, 2949, 2960, 2973, 3045, 3048, 3051, 3054, 3083, 3116, 3119, 3374, 3383, 3386, 3390, 3415, 3466, 3481, 3484, 3487, 3490, 3542, 3551, 3946, 3948, 3953, 4009, 4396, 4582, 4645, 4877, 5173, 5247, 5250, 5255, 5256, 5385, 5733–5735, 5749, 5768, 5887, 6202, 6238, 6288, 6479, 6480, 6509, 6512, 6515, 6518, 6521, 6524, 6527, 6530, 6533, 6536, 6539, 6542, 6545, 6548, 6551, 6554, 6557, 6560, 6563, 6567, 6571, 6575, 6579, 6583, 6587, 6600, 6605, 6606, 6610, 6614, 6633, 6649, 6665, 6671, 6672, 6674, 6683, 6692, 6700, 6709, 6718, 6720, 6782, 6783, 6829, 6831, 6833, 6876, 6930
\cs_new_nopar:Npx . . *17*, 1176, 1183, 1206
\cs_new_nopar:Nx *19*, 1385
\cs_new_protected:cn *19*, 1424
\cs_new_protected:cpn . . . *17*, 1225, 1229
\cs_new_protected:cpx *17*, 1225, 1230, 5492, 5497, 5502, 5506
\cs_new_protected:cx *19*, 1424
\cs_new_protected:Nn *19*, 1385
\cs_new_protected:Npn *17*, 903, 929, 1176, 1188, 1229, 1249, 1256, 1260, 1497, 1610, 1677, 1828, 1854, 1860, 1868, 2630, 2638, 2641, 2644, 3523, 3531, 3538, 3554, 3557, 3560, 3563, 3626, 3629, 3632, 3635, 3638, 3641, 3644, 3647, 3650, 3653, 3665, 3668, 3671, 3674, 3677, 3680, 3683, 3686, 3689, 3692, 3962, 3971, 3981, 3986, 4015, 4016, 4026, 4042, 4052, 4060, 4094, 4125, 4146, 4147, 4150, 4153, 4232, 4235, 4238, 4239, 4303, 4314, 4320, 4330, 4416, 4420, 4424, 4446, 4449, 4450, 4454, 4457, 4466, 4478, 4484, 4504, 4509, 4595, 4597, 4601, 4624, 4664, 4676, 4687, 4695, 4714, 4719, 4736, 4805, 4811, 4814, 4816, 4818, 4820, 4822, 4828, 4834, 4840, 4850, 4852, 4854, 4864, 4866, 5287, 5306, 5313, 5320, 5362, 5407, 5511, 5524, 5594, 5598, 5667, 5671, 5683, 5684, 5693, 5697, 5713–5716, 5861, 5923, 5934, 5953, 5969, 5974, 6000, 6074, 6091, 6115, 6120, 6131, 6137, 6146, 6150, 6161, 6247, 6257, 6261, 6292, 6790
\cs_new_protected:Npx *17*, 1176, 1189, 1230
\cs_new_protected:Nx *19*, 1385
\cs_new_protected_nopar:cn *19*, 1424

\cs_new_protected_nopar:cpn *18*, 1217, 1221
\cs_new_protected_nopar:cpx *18*, 1217, 1222, 1671
\cs_new_protected_nopar:cx *19*, 1424
\cs_new_protected_nopar:Nn *19*, 1385
\cs_new_protected_nopar:Npn *18*, 1176, 1186, 1221, 1253–1255, 1257–1259, 1264–1267, 1270, 1273, 1276, 1876–1885, 2049, 2167, 2271, 2284–2315, 2317, 2320, 2321, 2333, 2343, 2353, 2364, 2622, 2623, 2699, 2771, 2795, 2799, 2809, 2814, 2822, 2827, 2832, 2838, 2852, 2854, 2856, 2868, 2874, 2889, 2996, 3309, 3313, 3323, 3329, 3338, 3344, 3353, 3360, 3366, 3391, 3431, 3435, 3445, 3447, 3449, 3451, 3453, 3457, 3459, 3461, 3475, 3497, 3501, 3509–3514, 3536, 3537, 3584, 3588, 3599, 3601, 3604, 3616, 3703, 3709, 3984, 4000, 4091, 4114, 4119, 4122, 4210, 4214, 4227, 4244, 4248, 4253, 4265, 4269, 4276, 4279, 4288, 4293, 4307, 4390, 4412, 4431, 4491, 4497, 4514, 4517, 4524, 4527, 4562, 4591, 4609, 4629, 4633, 4637, 4641, 4693, 4700, 4709, 4710, 4724, 4727, 4730, 4733, 4751, 4757, 4788, 4896, 4955, 4959, 4970, 4974, 4978, 4991, 5004, 5011, 5018, 5043, 5058, 5083, 5098, 5110, 5122, 5125, 5142, 5143, 5147, 5149, 5150, 5154, 5158, 5161, 5164, 5169, 5181, 5184, 5257, 5261, 5265, 5270, 5275, 5327, 5330, 5333, 5336, 5339, 5344, 5387, 5396, 5470, 5473, 5476, 5479, 5482, 5485, 5488, 5491, 5544, 5548, 5632, 5634, 5637, 5639, 5645, 5647, 5666, 5669, 5675, 5677, 5680, 5685, 5692, 5695, 5701, 5703, 5708, 5763, 5852, 5855, 5858, 5884, 6016, 6020, 6024, 6058, 6077, 6082, 6106, 6125, 6142, 6176, 6181, 6187, 6207, 6211, 6216, 6224, 6227, 6241, 6244, 6253, 6276, 6302, 6306, 6768, 6991, 7030, 7045, 7049, 7052, 7058
\cs_new_protected_nopar:Npx *18*, 1176, 1187, 1222
\cs_new_protected_nopar:Nx *19*, 1385
\cs_record_meaning:N *15*, 1118, 1119, 1130
\cs_set:cn *21*, 1408
\cs_set:cpn *20*, 1209, 1209, 1930, 1953, 1956, 1968, 1969, 3171, 3175, 3179, 3183, 3187, 3191, 3195, 5267, 5268, 5272
\cs_set:cpx *20*, 1209, 1210, 2050, 2054, 2058, 2062, 2066, 2075, 2084, 2093, 2102, 2104, 2106, 2108, 2110, 2112, 2114, 2116, 2118
\cs_set:cx *21*, 1408
\cs_set:Nn *21*, 1362
\cs_set:Npn *20*, 798, 800, 838, 846–881, 885, 909, 938–940, 945, 952, 957, 962, 967, 972, 977, 982, 987, 992, 995, 1006–1009, 1019, 1025, 1029, 1038, 1042, 1045, 1074, 1077, 1119, 1173, 1176, 1184, 1198, 1209, 1302, 1305, 1317, 1321, 1353, 1362, 1401, 1541, 1614, 1617, 1644, 1919, 1923, 1970, 1973, 1978, 2125, 2126, 3154, 3161, 3165, 3204, 3211, 3221, 3231, 3234, 3258, 3259, 3273, 3276, 3279, 3282, 3285, 3288, 3291, 3294, 4036, 4070, 4079, 4095, 4100, 4127, 4403, 4492, 4498, 4573, 4738, 4806, 4887, 5294, 5315, 5322, 5367, 5941, 6118
\cs_set:Npx *20*, 798, 803, 841, 1185, 1210, 6123
\cs_set:Nx *21*, 1362
\cs_set_eq:cc *23*, 933, 1245, 1248, 1889, 4385
\cs_set_eq:cN *23*, 1245, 1246, 1271, 1879, 1881, 1888, 4383, 5273, 5862, 5865, 6245
\cs_set_eq:Nc *23*, 1245, 1247, 1887, 4384, 5873, 5875, 6152, 6188
\cs_set_eq:NN *23*, 1245, 1245–1248, 1251, 1256, 1262, 1268, 1878, 1880, 1886, 2631, 2645, 2661, 2702, 2774, 3148, 3149, 3151–3153, 3406, 3407, 3553, 3585, 3594, 4382, 4494, 4500, 4964, 4965, 5167, 5289, 5290, 5316, 5323, 5659, 5661, 5810, 5868, 5869, 6017, 6021, 6025, 6495, 6496, 6499, 6500, 6615–6621, 6623, 6635, 6638, 6641, 6651, 6654, 6657, 6695, 6696, 6729–6731, 6736–6739, 6844, 6845, 6877–6882, 6921, 6933–6935, 6939–6941
\cs_set_eq:NwN *24*, 753, 759–786, 788, 791–799, 818, 819, 1100, 1245, 1467–1469

\cs_set_nopar:cn *22*, 1408
\cs_set_nopar:cpn *20*, 1198, 1201, 1233–1241, 1243, 2890, 2891, 5742, 6943
\cs_set_nopar:cpx *20*, 1198, 1202
\cs_set_nopar:cx *22*, 1408
\cs_set_nopar:Nn *22*, 1362
\cs_set_nopar:Npn *20*, 742, 798, 798, 800, 801, 803, 806, 807, 809, 813, 1012, 1095–1098, 1152, 1169, 1182, 1201, 1318, 1350, 1517, 1527, 1552–1554, 1556–1558, 1562–1567, 1570, 1571, 1574, 1578, 1580, 1582, 1585, 1586, 1590–1593, 1598, 1863, 1871, 2913, 2991, 2993, 3071, 3098, 3110, 3127, 3828, 4696, 4825, 4844, 4855, 4857, 4883, 5292, 5363, 5758, 5774, 5791, 5792, 5801, 5802, 5805, 5808, 6643, 6659, 6726, 6732
\cs_set_nopar:Npx . *20*, 798, 799, 804, 810, 816, 1183, 1202, 1498, 3555, 3558, 3627, 3630, 3633, 3636, 3639, 3666, 3669, 3672, 3675, 3678, 5775
\cs_set_nopar:Nx *22*, 1362
\cs_set_protected:cn *22*, 1408
\cs_set_protected:cpn *21*, 1225, 1225, 5346
\cs_set_protected:cpx *21*, 1225, 1226, 1363, 1402, 5349, 5352, 5355, 5358
\cs_set_protected:cx *22*, 1408
\cs_set_protected:Nn *22*, 1362
\cs_set_protected:Npn *21*, 798, 812, 839, 882, 888, 894, 897, 900, 906, 912, 919, 922, 926, 932, 935, 1188, 1225, 1245, 3150, 5575, 5762, 6727, 6745, 6784
\cs_set_protected:Npx *21*, 798, 815, 1189, 1226
\cs_set_protected:Nx *22*, 1362
\cs_set_protected_nopar:cn *22*, 1408
\cs_set_protected_nopar:cpn *21*, 1217, 1217, 5739
\cs_set_protected_nopar:cpx *21*, 1217, 1218
\cs_set_protected_nopar:cx *22*, 1408
\cs_set_protected_nopar:Nn *22*, 1362
\cs_set_protected_nopar:Npn *21*, 798, 806, 812, 815, 820, 823, 826, 829, 832, 835, 1108, 1111, 1114, 1121, 1136, 1139, 1146, 1177, 1186, 1217, 1246–1248, 1555, 1559, 1561, 1568, 1572, 1576, 1577, 1581, 1583, 1587, 1588, 1594, 1595, 1597, 2844–2847, 5146, 5148, 6883–6886, 6892, 6897, 6912
\cs_set_protected_nopar:Npx *21*, 798, 809, 1187, 1218
\cs_set_protected_nopar:Nx *22*, 1362
\cs_show:c *12*, 781, 789, 6289
\cs_show:N *12*, 781, 788, 789, 3551
\cs_split_function:NN *25*, 884, 890, 896, 902, 908, 914, 921, 928, 1000, 1002, 1020, 1025, 1043, 1046, 1075, 1303, 1612
\cs_split_function_aux:w 1020, 1026, 1029
\cs_split_function_auxii:w 1020, 1036, 1038
\cs_str_aux:w 1013, 1019
\cs_tmp: 841, 844
\cs_tmp:w 1176, 1182–1198, 1201–1232, 1362, 1369–1401, 1408–1439
\cs_to_str:N *4*, *24*, 1012, 1012, 1027, 2134, 5174
\cs_to_str_aux:w 1012
\cs_undefine:c *23*, 1267, 1270
\cs_undefine:N *23*, 1267, 1267
\csname 11, 158
\currentgrouplevel 410
\currentgrouptype 411
\currentifbranch 407
\currentiflevel 406
\currentiftype 408

D

\d 1648, 2461
\dagger 2963, 2969
\day 366
\ddagger 2964, 2970
\deadcycles 300
\DeclareOption 48, 51
\def 3–6, 24, 43, 49, 52, 57, 65
\def_long_test_function_new:npn . . 4186
.default:n *125*
.default:V *125*
\default@ds 712
\defaulthyphenchar 350
\defaultskewchar 351
\DefineCrossReferences 5792, 5798
\delcode 381
\delimiter 175
\delimiterfactor 224
\delimitershortfall 223

\depthof *133*, 6877, 6877
\detokenize 398
\dim_add:cn *66*, 3453
\dim_add:Nc *66*, 3453
\dim_add:Nn *66*, 3453, 3453, 3456, 3457, 6534
\dim_compare:nNn 3477
\dim_compare:nNnF 3485, 3491
\dim_compare:nNnT 3482, 3488
\dim_compare:nNnTF *67*, 2024, 3477
\dim_compare_p:nNn *67*, 3477
\dim_do_until:nNnn . . *68*, 3481, 3490, 3491
\dim_do_while:nNnn . . *68*, 3481, 3487, 3488
\dim_eval:n *67*, 2019, 3445, 3454, 3475, 3475, 3478, 5641–5644, 5683, 5698, 6777, 6778
\dim_gadd:cn *66*, 3453
\dim_gadd:Nn . . . *66*, 3453, 3457, 3458, 6537
.dim_gset:c *125*
\dim_gset:cc *66*, 3445
\dim_gset:cn *66*, 3445
.dim_gset:N *125*
\dim_gset:Nc *66*, 3445
\dim_gset:Nn *66*, 3445, 3447, 3448, 6531, 6734
\dim_gsub:cn *66*, 3459
\dim_gsub:Nn . . . *66*, 3459, 3461, 3462, 6543
\dim_gzero:c *66*, 3449
\dim_gzero:N *66*, 3449, 3451, 3452
\dim_new:c *65*, 3427
\dim_new:N *65*, 3427, 3431, 3443, 3467–3472, 6485–6487
\dim_new_local:c *65*, 3427
\dim_new_local:N . . . *65*, 3427, 3435, 3444
.dim_set:c *125*
\dim_set:cn *66*, 3445
.dim_set:N *125*
\dim_set:Nc *66*, 3445
\dim_set:Nn *66*, 3445, 3445–3447, 6528
\dim_show:c *67*, 3465, 3466
\dim_show:N *67*, 3465, 3465, 3466
\dim_sub:cn *66*, 3459
\dim_sub:Nc *66*, 3459
\dim_sub:Nn *66*, 3459, 3459–3461, 6540
\dim_until_do:nNnn . . *68*, 3481, 3484, 3485
\dim_use:c *66*, 3463
\dim_use:N *66*, 2019, 3463, 3463, 3464
\dim_while_do:nNnn . . *68*, 3481, 3481, 3482
\dim_zero:c *66*, 3449
\dim_zero:N *66*, 3449, 3449–3451
\dimen 372
\dimendef 71
\dimexpr 425
\directlua 554
\discretionary 235
\displayindent 200
\displaylimits 210
\displaystyle 188
\displaywidowpenalties 438
\displaywidowpenalty 199
\displaywidth 201
\divide 78
\documentclass 5788
\doublehyphendemerits 268
\dp 379
\ds@ 713
\dump 362

E

\E 1654, 2463
\edef 22, 66
\efcode 463
\else 12, 119
\else: *8*, 746, 760, 763, 867, 954, 959, 964, 974, 979, 984, 1014, 1032, 1051, 1054, 1062, 1068, 1083, 1089, 1154, 1155, 1157, 1174, 1281, 1286, 1290, 1295, 1299, 1308, 1342, 1442, 1522, 1735, 1739, 1743, 1747, 1897, 1926, 1933, 1936, 1949, 1994, 2191, 2199, 2208, 2216, 2226, 2231, 2238, 2248, 2384, 2388, 2392, 2396, 2400, 2404, 2408, 2412, 2416, 2420, 2424, 2428, 2432, 2436, 2443, 2447, 2451, 2484, 2495, 2506, 2517, 2583, 2586, 2590, 2593, 2600, 2602, 2604, 2606, 2608, 2610, 2666, 2673, 2689, 3108, 3163, 3173, 3177, 3181, 3185, 3189, 3193, 3197, 3202, 3216, 3226, 3238, 3243, 3246, 3249, 3267, 3271, 3479, 3546, 3608, 3620, 3724, 3731, 3739, 3753, 3834, 3848, 3855, 3862, 3869, 3876, 3883, 3890, 3897, 3904, 3911, 3918, 3925, 3932, 3939, 4172, 4177, 4182, 4195, 4405, 4575, 4586, 5178, 5612, 5615, 5626, 6590, 6592, 6594, 6615–6620, 6622, 6636, 6639, 6642, 6652, 6655, 6658, 6750, 6797, 6803, 6809, 6810, 6815
\emergencystretch 283
\end 157, 5835
\endcsname 11, 159

\endgroup 13, 17, 92
\endinput 18, 131
\endL 446
\endlinechar 29, 37, 173
\endR 448
\eqno 193
\errhelp 14, 139
\errmessage 15, 133
\erroneous 1534
\ERROR 1792, 2125, 2126, 4370, 4371, 4494, 4500, 5952, 6014, 6015
\errorcontextlines 140
\errorstopmode 154
\escapechar 172
\etex_beginL:D 445
\etex_beginR:D 447
\etex_botmarks:D 394
\etex_clubpenalties:D 436
\etex_currentgrouplevel:D 410
\etex_currentgrouptype:D 411, 1753
\etex_currentifbranch:D 407
\etex_currentiflevel:D 406
\etex_currentiftype:D 408
\etex_detokenize:D 398, 1670, 3945
\etex_dimexpr:D 425, 3475, 6678, 6687, 6704, 6713, 6761, 6773
\etex_displaywidowpenalties:D 438
\etex_endL:D 446
\etex_endR:D 448
\etex_eTeXrevision:D 390
\etex_eTeXversion:D 389
\etex_everyeof:D 450, 4028, 4044, 4054, 4062
\etex_firstmarks:D 393
\etex_fontchardp:D 418
\etex_fontcharht:D 417
\etex_fontcharic:D 420
\etex_fontcharwd:D 419
\etex_glueexpr:D 426, 3391, 6679, 6688, 6705, 6714, 6762, 6774
\etex_glueshrink:D 429, 3424
\etex_glueshrinkorder:D 431, 3412
\etex_gluestretch:D 428, 3423
\etex_gluestretchorder:D 430, 3411
\etex_gluetomu:D 432
\etex_ifcsname:D 387, 777
\etex_ifdefined:D 386, 555, 556, 776
\etex_iffontchar:D 416
\etex_interactionmode:D 414
\etex_interlinepenalties:D 435
\etex_lastlinefit:D 434
\etex_lastnodetype:D 415, 1755, 1757
\etex_marks:D 391
\etex_middle:D 439
\etex_muexpr:D . 427, 3509, 3511, 3513, 6680, 6689, 6706, 6715, 6763, 6775
\etex_mutoglue:D 433, 6751, 6752
\etex_numexpr:D 424, 1322, 1356, 3149, 6677, 6686, 6703, 6712, 6760, 6772
\etex_pagediscards:D 442
\etex_parshapedimen:D 423
\etex_parshapeindent:D 421
\etex_parshapelength:D 422
\etex_predisplaydirection:D 449
\etex_protected:D 451, 560, 563, 586, 590, 594, 599, 618, 627, 634, 642, 652, 656, 660, 680, 730, 797
\etex_readline:D 401
\etex_savinghyphcodes:D 440
\etex_savingvdiscards:D 441
\etex_scantokens:D 399, 4031, 4047, 4057, 4065
\etex_showgroups:D 412
\etex_showifs:D 413
\etex_showtokens:D 400, 3553
\etex_splitbotmarks:D 396
\etex_splitdiscards:D 443
\etex_splitfirstmarks:D 395
\etex_TeXXETstate:D 444
\etex_topmarks:D 392
\etex_tracingassigns:D 402
\etex_tracinggroups:D 409
\etex_tracingifs:D 405
\etex_tracingnesting:D 404
\etex_tracingscantokens:D 403
\etex_unexpanded:D 397, 564, 665, 669, 672, 780, 1469
\etex_unless:D 388, 765
\etex_widowpenalties:D 437
\etexmissingerror 5, 15, 16
\etexmissinghelp 6, 14, 16
\eTeXrevision 390
\eTeXversion 389
\everycr 101
\everydisplay 202
\everyeof 450
\everyhbox 341
\everyjob 370
\everymath 226
\everypar 289
\everyvbox 342

\ExecuteOptions 53
\exhyphenpenalty 265
\exp_after:cc 5758
\exp_after:wN *15*, *182*, 778, 778, 879–881, 947, 954, 959, 964, 969, 974, 979, 984, 989, 997, 1013, 1015, 1017, 1026, 1031, 1033, 1061, 1063, 1066, 1078, 1082, 1084, 1159–1164, 1170, 1307, 1309, 1312, 1365, 1404, 1466, 1467, 1486, 1489, 1492, 1493, 1499, 1502, 1503, 1510, 1511, 1518, 1523, 1525, 1528, 1536–1543, 1545–1548, 1550, 1551, 1599, 1601, 1604, 1607, 1609, 1619, 1645, 1661, 1669, 1671, 1673, 1686, 1692, 1693, 1696, 1699, 1700, 1703, 1704, 1715, 1721, 1764, 1932, 1935, 1937, 1943, 1946, 1954, 1956, 1958, 2146, 2176, 2178, 2183, 2188, 2190, 2192, 2198, 2200, 2399, 2439, 2450, 2468, 2475, 2485, 2496, 2507, 2518, 2526, 2533, 2540, 2562, 2568, 2574, 2665, 2667, 2672, 2674, 2684, 2688, 2690, 2902, 2906, 3100–3108, 3158, 3162, 3547, 3738, 3750, 3778, 3781, 3802, 3808, 3814, 3820, 3826, 3946, 3954, 3955, 4031, 4036–4039, 4065, 4073, 4109, 4138, 4171, 4175, 4180, 4193, 4233, 4236, 4238, 4240, 4289, 4300, 4304, 4315, 4316, 4321, 4322, 4331, 4332, 4407, 4414, 4422, 4447, 4486, 4587, 4593, 4599, 4647, 4648, 4809, 4884, 4885, 4889, 5752, 5759, 5760, 5905, 5917, 5931, 5938, 5959, 5960, 5977, 5978, 5988, 5993, 6209, 6502, 6504, 6589
\exp_arg_last_unbraced:nn 1690, 1690, 1692, 1696, 1699, 1703
\exp_arg_next:nnn 1472, 1472, 1489, 1492, 1499, 1502, 1510
\exp_arg_next_nobrace:nnn 1472, 1475, 1486
\exp_args:cc *30*, 1541, 1542
\exp_args:Nc *30*, 787, 789, 790, 879, 879, 953, 958, 963, 968, 1095–1098, 1137, 1147, 1199, 1246, 1253, 1258, 1264, 1319, 1351, 1444–1447, 1541, 1541, 1665, 2887, 3966, 4681, 6164, 6894, 6904
\exp_args:Ncc *31*, 1248, 1255, 1259, 1266, 1452–1455, 1541, 1545
\exp_args:Nccc *32*, 1541, 1547
\exp_args:Ncco *32*, 1578, 1592, 1593
\exp_args:Nccx *31*, 1578, 1594
\exp_args:Ncf . . *31*, 1556, 1564, 1832, 1833
\exp_args:NcNc *32*, 1578, 1590, 5752
\exp_args:NcNo *32*, 1578, 1591
\exp_args:Ncnx *31*, 1578, 1595
\exp_args:Nco *31*, 1550, 1550, 1556
\exp_args:Ncx *30*, 1556, 1561
\exp_args:Nf *30*, 1552, 1552, 1805, 1806, 1813, 1822, 1856, 1857, 1864, 1872, 2009, 2916, 2919, 2921, 2992, 3113, 3122
\exp_args:Nff *31*, 1556, 1563, 2994
\exp_args:Nfo *31*, 1556, 1562
\exp_args:NNc *31*, 1000, 1002, 1247, 1254, 1257, 1265, 1448–1451, 1541, 1543
\exp_args:NNf . . *31*, 1556, 1556, 1842, 1850
\exp_args:Nnf *31*, 907, 913, 920, 927, 1556, 1565
\exp_args:Nnnc *32*, 1578, 1585
\exp_args:NNNo *32*, 1536, 1539
\exp_args:NNno *32*, 1578, 1580
\exp_args:Nnno *32*, 1578, 1586
\exp_args:NNNV . *32*, 1578, 1578, 4032, 4048
\exp_args:NNnx *31*, 1578, 1581
\exp_args:Nnnx *31*, 1578, 1587
\exp_args:NNo *31*, 1536, 1537, 2222, 2244, 5916
\exp_args:Nno *31*, 1556, 1566
\exp_args:NNoo *32*, 1578, 1582
\exp_args:NNox *31*, 1578, 1583
\exp_args:Nnox *31*, 1578, 1588
\exp_args:NNV *31*, 1556, 1558
\exp_args:NNv *31*, 1556, 1557
\exp_args:NnV *31*, 1556, 1567
\exp_args:NNx *30*, 1556, 1559, 5776
\exp_args:Nnx *30*, 1556, 1568
\exp_args:No *29*, 1536, 1536, 2019, 3984, 4693, 6733
\exp_args:Noc *31*, 1556, 1570
\exp_args:Noo *31*, 1556, 1571
\exp_args:Nooo *32*, 1578, 1598
\exp_args:Noox *31*, 1578, 1597
\exp_args:Nox *30*, 1556, 1572
\exp_args:NV . . . *30*, 1552, 1554, 4066, 6190
\exp_args:Nv *30*, 1552, 1553
\exp_args:NVV *31*, 1556, 1574
\exp_args:Nx *30*, 1552, 1555
\exp_args:Nxo *30*, 1556, 1576
\exp_args:Nxx *30*, 1556, 1577

\exp_args:wN 842
\exp_eval_error_msg:w *183*, 1517, 1520, 1533
\exp_eval_register:c *182*, 1513, 1517, 1527, 1604, 1705, 4236
\exp_eval_register:N *182*, 1505, 1517, 1517, 1528, 1607, 1700, 4233, 4317
\exp_last_unbraced:NcV *33*, 1708, 1711, 3975, 4669
\exp_last_unbraced:Nf *33*, 1708, 1708, 2133
\exp_last_unbraced:NNNo . *33*, 1708, 1720
\exp_last_unbraced:NNo *33*, 1708, 1714, 4577, 5911, 6208
\exp_last_unbraced:NNV *33*, 1708, 1717, 4277, 5929
\exp_last_unbraced:NV . . . *33*, 1708, 1709
\exp_last_unbraced:Nv . . . *33*, 1708, 1710
\exp_not:c *32*, 1609, 1609, 1630, 1684, 2051, 2055, 2060, 2064, 2067, 2069–2071, 2076, 2078–2080, 2085, 2087–2089, 2094, 2096–2098, 2103, 2105, 2107, 2109, 2111, 2113, 2115, 2117, 2119–2121, 2704, 3830, 3833, 5350, 5353, 5356, 5359, 5494, 5499, 5504, 5508, 5738, 5744, 6060, 6311
\exp_not:f *32*, 1599, 1600
\exp_not:N . . . *15*, *32*, 778, 779, 1364–1366, 1403–1405, 1466, 1468, 1518, 1609, 2056, 2059, 2063, 2067, 2076, 2085, 2094, 2214, 2236, 2380, 2383, 2387, 2391, 2395, 2399, 2403, 2407, 2411, 2415, 2419, 2423, 2431, 2435, 2450, 2671, 2687, 3830, 3832, 4044, 4054, 4175, 4176, 4180, 4193, 4194, 5743, 5745, 5894, 5902, 5912, 6061, 6064, 6066, 6213, 6311, 6847, 6850, 6853, 6856, 6859, 6863, 6867, 6871
\exp_not:n *15*, *32*, 778, 780, 1466, 1469, 1599, 1601, 1604, 1607, 2052, 2215, 2237, 2702, 2705, 3555, 3561, 3627, 3642, 3666, 3681, 3790, 3793, 3796, 3799, 3802, 3805, 3808, 3814, 3820, 3826, 3834, 3845, 3846, 3852, 3859, 3866, 3874, 3881, 3916, 4440, 4618, 4797, 4799, 4818, 5144, 5170, 5742, 6108, 6109, 6325, 6839, 6842, 6847, 6850, 6853, 6856, 6859, 6863, 6867, 6871, 6875
\exp_not:o *32*, 1599, 1599, 3639, 3651, 3678, 3690, 3860, 3880, 3894, 3895, 3908, 3930, 5975
\exp_not:V *32*, 1599, 1606, 3627, 3630, 3633, 3636, 3639, 3642, 3645, 3648, 3651, 3654, 3666, 3669, 3672, 3675, 3678, 3681, 3684, 3687, 3690, 3693, 3799, 3805, 3811, 3817, 3823, 3853, 3873, 3887, 3888, 3901, 3923, 4391, 4704, 4706, 6111
\exp_not:v *32*, 1599, 1603, 3633, 3648, 3672, 3687
\exp_stop_f: *32*, 1491, 1496, 1525, 2134, 4001
\expandafter 11, 12, 14, 15, 89
\ExplSyntaxNamesOff *6*, 586, 590
\ExplSyntaxNamesOn *6*, 586, 586, 5794
\ExplSyntaxOff *4*, *6*, 22, 560, 563, 667, 674, 685, 702, 710, 5812
\ExplSyntaxOn *4*, *6*, 560, 560, 654, 658, 662, 683, 724, 5790
\ExplSyntaxPopStack . . . *136*, 676, 680, 680
\ExplSyntaxStack *136*, 666, 673, 676, 680, 681, 688
\ExplSyntaxStatus 23, 24, *136*, 560, 561, 565, 566, 576, 585, 666

F

\F 2458
\fam 81
\fi 12, 32, 120
\fi: *8*, 748, 760, 764, 866–869, 948, 954, 959, 964, 969, 974, 979, 984, 989, 998, 1016, 1034, 1056, 1057, 1064, 1070, 1085, 1091, 1156, 1157, 1170, 1171, 1174, 1283, 1286, 1292, 1295, 1299, 1310, 1347, 1442, 1521, 1524, 1534, 1620, 1645, 1687, 1735, 1739, 1743, 1747, 1750, 1751, 1897, 1926, 1938, 1939, 1951, 1994, 2179, 2184, 2193, 2201, 2208, 2217, 2227, 2231, 2239, 2249, 2384, 2388, 2392, 2396, 2400, 2404, 2408, 2412, 2416, 2420, 2424, 2428, 2432, 2436, 2443, 2447, 2451, 2487, 2498, 2509, 2520, 2585, 2592, 2595, 2596, 2612–2617, 2668, 2675, 2691, 2930, 2939, 2971, 2984, 3109, 3163, 3168, 3173, 3177, 3181, 3185, 3189, 3193, 3197, 3202, 3208, 3218, 3228, 3245, 3251, 3252, 3254, 3267, 3271, 3479, 3548, 3610, 3622, 3724, 3731, 3739, 3753, 3834, 3849, 3856, 3863, 3870, 3877, 3884, 3891, 3898,

3905, 3912, 3919, 3926, 3933, 3940, 4172, 4177, 4184, 4195, 4400, 4405, 4566, 4575, 4588, 4890, 5178, 5612, 5615, 5626, 6596–6598, 6625–6631, 6644–6646, 6660–6662, 6680, 6689, 6706, 6715, 6753, 6763, 6775, 6797, 6803, 6809, 6815, 6816, 6908, 6926
\file_if_exist:n 6981
\file_if_exist:nF 7028
\file_if_exist:nT 7027, 7031
\file_if_exist:nTF *135*, 6980, 7029
\file_if_exist:VTF *135*, 6980
\file_if_exist_aux:n 6980
\file_if_exist_path_aux:n . . . 6984, 6991
\file_input:n *136*, 7030, 7030, 7044
\file_input:V *136*, 7030
\file_list: *136*, 7052, 7052
\file_list_aux:n 7052, 7055, 7058
\file_path_include:n . . . *136*, 7045, 7045
\file_path_remove:n *136*, 7045, 7049
\fileauthor *6*, 594, 608, 623
\filedate *6*, 594, 609, 636, 640, 645, 648, 755, 1462, 1729, 2163, 2266, 2784, 3142, 3301, 3519, 4202, 4366, 4538, 4767, 4912, 5190, 5585, 5727, 5839, 6041, 6474, 6950
\filedescription *6*, 594, 605, 622, 640, 648, 755, 1462, 1729, 2163, 2266, 2784, 3142, 3301, 3519, 4202, 4366, 4538, 4767, 4912, 5190, 5585, 5727, 5839, 6041, 6474, 6950
\filename *6*, 594, 606, 620, 640, 648, 755, 1462, 1729, 2163, 2266, 2784, 3142, 3301, 3519, 4202, 4366, 4538, 4767, 4912, 5190, 5585, 5727, 5839, 6041, 6474, 6950
\filenameext *6*, 594, 610, 637, 644
\filetimestamp *6*, 594, 611, 638, 646
\fileversion *6*, 594, 607, 621, 640, 648, 755, 1462, 1729, 2163, 2266, 2784, 3142, 3301, 3519, 4202, 4366, 4538, 4767, 4912, 5190, 5585, 5727, 5839, 6041, 6474, 6950
\finalhyphendemerits 269
\firstchoice@true 6889
\firstmark 167
\firstmarks 393
\firstoftwo 3, 12
\floatingpenalty 314
\font 80
\fontchardp 418
\fontcharht 417
\fontcharic 420
\fontcharwd 419
\fontdimen 347
\fontname 171
\frozen@everydisplay 693
\frozen@everymath 692
\futurelet 76

G

\G 2463
\g 2128
\g_box_allocation_seq 5591
\g_calc_A_dim *353*, 6482, 6485, 6528, 6531, 6534, 6537
\g_calc_A_int *353*, 6482, 6482, 6510, 6513, 6516, 6519, 6522, 6525, 6540, 6543, 6695, 6707, 6716
\g_calc_A_muskip . . . *353*, 6482, 6491, 6564, 6568, 6572, 6576, 6580, 6584
\g_calc_A_register *353*, 6478, 6479, 6495, 6505, 6605, 6608, 6611, 6612, 6667, 6681, 6690, 6695, 6734, 6795, 6801, 6807, 6813
\g_calc_A_skip *353*, 6482, 6488, 6546, 6549, 6552, 6555, 6558, 6561
\g_clist_inline_level_int 4663, 4666, 4667, 4670, 4672, 4678, 4679, 4682, 4684
\g_file_current_name_tl *135*, 6954, 6954, 6959, 6963, 6970, 7038, 7039, 7041
\g_file_record_seq 6966, 6966, 6970, 7053, 7055, 7063
\g_file_stack_seq 6954, 6955, 7038, 7041
\g_file_test_stream 6980, 6982, 6983, 6986, 7006, 7007, 7022
\g_ior_streams_prop *105*, 4940, 4941, 4947, 4999, 5090, 5116, 5126
\g_ior_tmp_stream *105*, 5018, 5067, 5068, 5071
\g_iow_streams_prop *105*, 4940, 4940, 4944–4946, 4986, 5005, 5012, 5050, 5104, 5123
\g_iow_tmp_stream *105*, 5018, 5027, 5028, 5031
\g_peek_token *52*, 2619, 2620, 2624
\g_prg_inline_level_int *194*, 1827, 1827, 1829, 1830, 1834, 1836

\g_prop_inline_level_int *101*, 4895, 4895, 4897, 4898, 4901, 4902
\g_tl_inline_level_int *77*, 3135, 3962, 3963, 3964, 3967, 3969, 3972, 3973, 3976, 3978
\g_tmpa_bool *193*, 1894, 1895
\g_tmpa_dim *68*, 3467, 3471
\g_tmpa_int *58*, 2986, 2989
\g_tmpa_skip *65*, 3392, 3397
\g_tmpa_tl *77*, 3714, 3714
\g_tmpa_toks *84*, 4351, 4355
\g_tmpb_dim *68*, 3467, 3472
\g_tmpb_int *58*, 2986, 2990
\g_tmpb_skip *65*, 3392, 3398
\g_tmpb_tl *77*, 3714, 3715
\g_tmpb_toks *84*, 4351, 4356
\g_tmpc_toks *84*, 4351, 4357
\g_toks_allocation_seq 4352
\g_trace_statistics_status 741
\g_xref_all_curr_deferred_fields_prop *327*, 5731, 5732, 5779
\g_xref_all_curr_immediate_fields_prop *327*, 5731, 5731, 5775
\gaddtolength *133*, 6883, 6886
\gdef 67
.generate_choices:n *125*
\GetIdInfo *6*, 594, 594
\GetIdInfoAuxCVS:w 594, 629, 634
\GetIdInfoAuxi:w 594, 614, 618
\GetIdInfoAuxii:w 594, 624, 627
\GetIdInfoAuxSVN:w 594, 631, 642
\GetIdInfoMaybeMissing:w 597, 599
\getname 5802, 5830, 5833
\getpage 5805, 5831, 5834
\getvaluepage 5808, 5831, 5834
\global 82
\globaldefs 86
\glueexpr 426
\glueshrink 429
\glueshrinkorder 431
\gluestretch 428
\gluestretchorder 430
\gluetomu 432
\group_align_safe_begin: *40*, 1749, 1749, 1920, 2635, 2650
\group_align_safe_end: *40*, 1749, 1751, 1966, 1967, 2633, 2634, 2649, 2660
\group_begin: *25*, 791, 792, 840, 1020, 1530, 1647, 2127, 2140, 2367, 2456, 2548, 4017, 4027, 4043, 4053, 4061, 5165, 5228, 5280, 5288, 5314, 5321, 5764, 5793, 5878, 5891, 5924, 5935, 6498, 6601, 6602, 6668, 6694, 6728, 6746, 6765, 6780, 6785, 6791, 6821, 6826, 6920
\group_end: *25*, 791, 793, 843, 1024, 1535, 1658, 2131, 2144, 2381, 2466, 2556, 4023, 4033, 4049, 4058, 4066, 5165, 5231, 5286, 5303, 5318, 5325, 5772, 5796, 5883, 5921, 5931, 5938, 6503, 6506, 6611, 6612, 6615, 6666, 6667, 6693, 6742, 6746, 6757, 6769, 6787, 6817, 6820, 6825, 6922
\group_execute_after:N *26*, 794, 794, 6601, 6602, 6668, 6669, 6698, 6823
\gsetlength *133*, 6883, 6884

H

\H 2463
\halign 93
\hangafter 271
\hangindent 272
\hbadness 333
\hbox 328
\hbox:n *116*, 5692, 5692
\hbox:w 749
\hbox_gset:cn *116*, 5693
\hbox_gset:Nn *116*, 5693, 5695, 5696
\hbox_gset_inline_begin:c *116*, 5703
\hbox_gset_inline_begin:N *116*, 5703, 5708, 5711
\hbox_gset_inline_end: . *116*, 5703, 5712
\hbox_gset_to_wd:cnn *116*, 5697
\hbox_gset_to_wd:Nnn *116*, 5697, 5701, 5702
\hbox_overlap_left:n . . . *116*, 5715, 5715
\hbox_overlap_right:n . . *116*, 5715, 5716
\hbox_set:cn *116*, 5693
\hbox_set:Nn . . *116*, 5693, 5693–5695, 6723
\hbox_set_inline_begin:c *116*, 5703
\hbox_set_inline_begin:N *116*, 5703, 5703, 5706, 5709
\hbox_set_inline_end: . . *116*, 5703, 5707
\hbox_set_to_wd:cnn *116*, 5697
\hbox_set_to_wd:Nnn *116*, 5697, 5697, 5700, 5701
\hbox_to_wd:nn *116*, 5713, 5713
\hbox_to_zero:n . . . *116*, 5713, 5714–5716
\hbox_unpack:c *117*, 5717
\hbox_unpack:N *117*, 5717, 5717, 5718
\hbox_unpack_clear:c *117*, 5717

\hbox_unpack_clear:N *117*, 5717, 5719, 5720
\heightof *133*, 6877, 6879
\hfil 236
\hfill 238
\hfilneg 237
\hfuzz 335
\hoffset 310
\holdinginserts 313
\hrule 249
\hsize 274
\hskip 239
\hss 240
\ht 378
\hyphenation 364
\hyphenchar 348
\hyphenpenalty 266

I

\I 2463
\if 102
\if:w . *9*, 760, 766, 946, 996, 1013, 1167, 1170, 1171, 1618, 1645, 3163, 4175
\if_bool:N *9*, 760, 767, 1285, 1294, 1298, 1897
\if_box_empty:N *113*, 5608, 5610
\if_case:w *61*, 2924, 2933, 2961, 2974, 3099, 3146, 3147, 6676, 6685, 6702, 6711, 6759, 6771, 6792
\if_catcode:w . *9*, 760, 770, 2383, 2387, 2391, 2395, 2399, 2403, 2407, 2411, 2415, 2419, 2423, 2431, 2671, 4193
\if_charcode:w *9*, 760, 769, 2435, 2687, 4180
\if_cs_exist:N *9*, 776, 776, 1052, 1279, 1288
\if_cs_exist:w *9*, 776, 777, 1060
\if_dim:w *67*, 3476, 3476, 3478, 6802, 6808, 6814
\if_eof:w *105*, 5175, 5175
\if_false: *8*, 760, 761, 881, 1750
\if_hbox:N *113*, 5608, 5608
\if_horizontal_mode: 743
\if_inexpr_compare:w *61*
\if_intexpr_case:w *61*, 3148, 3153
\if_intexpr_compare:w 3148, 3151, 3172, 3176, 3180, 3184, 3188, 3192, 3196, 3200, 3206, 3213, 3223, 3236, 3240, 3241, 3247
\if_intexpr_odd:w *62*, 3148, 3152, 3266, 3270
\if_meaning:w *9*, 771, 771, 1030, 1049, 1066, 1153–1155, 1441, 1518, 1519, 1685, 1926, 1931, 1934, 1947, 2176, 2182, 2188, 2197, 2207, 2231, 2427, 2450, 2482, 2493, 2504, 2515, 2664, 3168, 3543, 3723, 3731, 3738, 3753, 3847, 3854, 3861, 3868, 3875, 3882, 3889, 3896, 3903, 3910, 3917, 3924, 3931, 3938, 4171, 4397, 4404, 4563, 4574, 4583, 4888, 6588, 6591, 6593, 6615–6621, 6634, 6637, 6640, 6650, 6653, 6656
\if_mode_horizontal: . . *10*, 772, 773, 1738
\if_mode_inner: *10*, 772, 775, 1742
\if_mode_math: *10*, 772, 772, 1746
\if_mode_vertical: *10*, 772, 774, 1734
\if_num:w *61*, 1750, 1751, 2213, 2235, 3146, 3146, 6747, 6796
\if_predicate:w . . . *9*, 760, 768, 1306, 1993, 2223, 2245, 2442, 2446, 2580, 2581, 2587, 2588, 2599, 2601, 2603, 2605, 2607, 2609, 3606, 3618, 4187
\if_true: *8*, 760, 760, 880
\if_vbox:N *113*, 5608, 5609
\ifcase 103
\ifcat 104
\ifcsname 387
\ifdefined 386
\ifdim 107
\ifeof 108
\iffalse 113
\iffirstchoice@ 6889, 6900, 6915
\iffontchar 416
\ifhbox 109
\ifhmode 115
\ifinner 118
\ifmmode 116
\ifnum 105
\ifodd 23, 106
\iftrue 114
\ifvbox 110
\ifvmode 117
\ifvoid 111
\ifx 11, 112
\ignorespaces 160
\immediate 122
\indent 256
\input 130
\input@path 6995, 6999, 7015
\InputIfFileExists 5795
\inputlineno 132
\insert 312
\insertpenalties 315
\int_add:cn *55*, 2856

\int_add:Nn *55*, 2844, 2845, 2856, 2856, 2873, 2880, 6516
\int_advance:w *231*, 2788, 2790, 2822, 2827, 2857, 2863
\int_Alph_default_conversion_rule:n *57*, 2923, 2932, 2947
\int_alph_default_conversion_rule:n *57*, 2923, 2923, 2943
\int_compare:nNnTF 2801, 3315, 3437, 3503, 4216, 5600
\int_const:Nn *57*, 2996, 2996, 3016–3031, 3033–3044
\int_convert_from_base_ten:nn *59*, 3071, 3071
\int_convert_from_base_ten_aux:ffn 3087
\int_convert_from_base_ten_aux:fon 3071
\int_convert_from_base_ten_aux:nfn 3074, 3078
\int_convert_from_base_ten_aux:nnn 3071, 3083, 3096, 3097
\int_convert_from_base_ten_aux:non 3071
\int_convert_letter_to_number:N 3123, 3127
\int_convert_number_to_letter:n *231*, 3085, 3089, 3098, 3098
\int_convert_number_with_rule:nnN *57*, 2913, 2913, 2916, 2942, 2946, 2952, 2956
\int_convert_to_base_ten:nn *59*, 3110, 3110
\int_convert_to_base_ten_aux:nn 3113, 3116
\int_convert_to_base_ten_auxi:nnN 3117, 3119, 3122
\int_decr:c *55*, 2822
\int_decr:N *55*, 2822, 2827, 2843, 2845, 2849, 5872
\int_gadd:cn *55*, 2856
\int_gadd:Nn *55*, 2846, 2847, 2856, 2868, 2881, 6519
\int_gdecr:c *55*, 2822
\int_gdecr:N *55*, 1836, 2822, 2838, 2847, 2851, 3969, 3978, 4672, 4684, 4902
\int_get_digits:n . *232*, 3045, 3051, 3113
\int_get_sign:n . . . *232*, 3045, 3048, 3112
\int_get_sign_and_digits:n *232*, 3045, 3045
\int_get_sign_and_digits_aux:nNNN 3045, 3046, 3049, 3052, 3054, 3070
\int_get_sign_and_digits_aux:oNNN 3045, 3058, 3059, 3063
\int_gincr:c *54*, 2822, 6919
\int_gincr:N *54*, 1829, 2822, 2832, 2846, 2850, 3963, 3972, 4666, 4678, 4897
.int_gset:c *126*
\int_gset:cn *55*, 2809
.int_gset:N *126*
\int_gset:Nn *55*, 2809, 2814, 2821, 3002, 3011, 6513
\int_gsub:cn *55*, 2856
\int_gsub:Nn . . . *55*, 2856, 2874, 2883, 6525
\int_gzero:c *55*, 2852
\int_gzero:N *55*, 2852, 2854, 2855
\int_incr:c *54*, 2822
\int_incr:N *54*, 2822, 2822, 2837, 2844, 2848, 5044, 5084, 5870, 6113
\int_new:c *54*, 2791
\int_new:N *54*, 1827, 2791, 2795, 2807, 2986–2990, 3001, 3010, 3135, 4663, 4895, 4950, 5849, 6049, 6481–6484
\int_new_local:c *54*, 2791
\int_new_local:N . . . *54*, 2791, 2799, 2808
\int_pre_eval_one_arg:Nn *231*, 2991, 2991
\int_pre_eval_two_args:Nnn *231*, 2991, 2993
\int_roman_lcuc_mapping:Nnn *57*, 2889, 2889, 2893–2900
.int_set:c *126*
\int_set:cn *55*, 2809
.int_set:N *126*
\int_set:Nn *55*, 2809, 2809, 2819, 2820, 4980, 4993, 5007, 5014, 5028, 5068, 5194, 6109, 6497, 6510
\int_show:c *56*, 2886, 2887
\int_show:N *56*, 2886, 2886, 2887
\int_sub:cn *55*, 2856
\int_sub:Nn *55*, 2856, 2862, 2879, 2882, 6522
\int_symbol_math_conversion_rule:n *57*, 2953, 2960, 2960
\int_symbol_text_conversion_rule:n *57*, 2957, 2960, 2973
\int_to_Alph:n *56*, 2941, 2945
\int_to_alph:n *56*, 2941, 2941
\int_to_arabic:n *56*, 2888, 2888
\int_to_number:w *57*, 2788, 2789
\int_to_Roman:n *56*, 2901, 2905
\int_to_roman:n *56*, 2901, 2901

\int_to_roman:w *56*, 1959, 2788, 2788, 2903, 2907, 4233, 4236, 4240, 4317, 4332
\int_to_roman_lcuc:NN *57*, 2901, 2902, 2906, 2909, 2911
\int_to_symbol:n *56*, 2949, 2949
\int_use:c *55*, 2884, 2885
\int_use:N *55*, 1830, 1834, 2274, 2324, 2336, 2346, 2356, 2884, 2884, 2885, 3964, 3967, 3973, 3976, 4667, 4670, 4679, 4682, 4898, 4901, 5019, 5020, 5030, 5035, 5038, 5048, 5059, 5060, 5070, 5075, 5078, 5088, 5863, 5866, 5874, 5876, 6110, 6755
\int_zero:c *55*, 2852
\int_zero:N *55*, 2852, 2852, 2853, 6093, 6697, 6822
\interactionmode 414
\interlinepenalties 435
\interlinepenalty 294
\intexpr_abs:n *60*, 3204, 3204
\intexpr_compare:n 3157
\intexpr_compare:nF 3277, 3283, 5101, 5113
\intexpr_compare:nNn 3199
\intexpr_compare:nNnF 1755, 1757, 1810, 1819, 1839, 1847, 1861, 1869, 3289, 3295
\intexpr_compare:nNnT . . 1753, 3286, 3292
\intexpr_compare:nNnTF *60*, 1804, 1831, 1855, 1989, 2014, 2914, 3072, 3084, 3128, 3131, 3199
\intexpr_compare:nT 3274, 3280, 5025, 5045, 5065, 5085
\intexpr_compare:nTF *59*, 2997, 2999, 3157, 4982, 4995, 6163
\intexpr_compare_auxi:w 3158, 3161
\intexpr_compare_auxii:w 3162, 3165
\intexpr_compare_p:n *59*, 3157
\intexpr_compare_p:nNn *60*, 3199, 3411, 3412
\intexpr_div_round:nn . . . *61*, 3231, 3258
\intexpr_div_truncate:nn *61*, 2917, 3092, 3231, 3234, 3262
\intexpr_div_truncate_raw:nn 3231
\intexpr_do_until:nn *62*, 3273, 3282, 3283
\intexpr_do_until:nNnn *62*, 3285, 3294, 3295
\intexpr_do_while:nn *62*, 3273, 3279
\intexpr_do_while:nNnn *62*, 3280, 3285, 3291, 3292
\intexpr_eval:n *59*, 1765, 1807, 1814, 1823, 1835, 1843, 1851, 1858, 1865, 1873, 2009, 2888, 2903, 2907, 2919, 2921, 2992, 2994, 3006, 3075, 3079, 3111, 3123, 3130, 3148, 3154, 3232, 3258, 4004, 4010
\intexpr_eval:w *61*, 2272, 2276, 2282, 2318, 2322, 2326, 2330, 2334, 2338, 2341, 2344, 2348, 2351, 2354, 2358, 2362, 2809, 2857, 2863, 3099, 3148, 3149, 3155, 3159, 3172, 3176, 3180, 3184, 3188, 3192, 3196, 3200, 3206, 3209, 3212, 3214, 3222, 3224, 3235, 3236, 3240, 3241, 3247, 3261, 3266, 3270
\intexpr_eval_end: *61*, 2272, 2276, 2282, 2318, 2322, 2326, 2330, 2334, 2338, 2341, 2344, 2348, 2351, 2354, 2358, 2362, 2809, 2857, 2863, 3099, 3148, 3150, 3155, 3172, 3176, 3180, 3184, 3188, 3192, 3196, 3201, 3209, 3214, 3219, 3224, 3229, 3256, 3263, 3266, 3270
\intexpr_if_even:n 3269
\intexpr_if_even:nTF *60*, 3265
\intexpr_if_even_p:n *60*, 1943, 3265
\intexpr_if_odd:n 3265
\intexpr_if_odd:nTF *60*, 3265
\intexpr_if_odd_p:n *60*, 3265
\intexpr_max:nn *60*, 3204, 3211
\intexpr_min:nn *60*, 3204, 3221
\intexpr_mod:nn *61*, 2919, 3089, 3231, 3259
\intexpr_until_do:nn *62*, 3273, 3276, 3277
\intexpr_until_do:nNnn *62*, 3285, 3288, 3289
\intexpr_value:w *61*, 3148, 3148, 3155, 3158, 3205, 3212, 3222, 3235, 3260
\intexpr_while_do:nn *62*, 3273, 3273, 3274
\intexpr_while_do:nNnn *62*, 3285, 3285, 3286
\ior_alloc_read:n 4994, 5004, 5011
\ior_close:c *102*
\ior_close:N *102*, 4992, 5110, 5121, 6986, 7022
\ior_gto:NN *104*, 5181, 5184
\ior_if_eof:N 5176
\ior_if_eof:NF 7007
\ior_if_eof:NTF *104*, 5176, 6983
\ior_if_eof_p:N *104*, 5176
\ior_new:c *102*, 4970
\ior_new:N *102*, 4970, 4974, 4977
\ior_open:cn *102*, 4978
\ior_open:Nn *102*, 4978, 4991, 5003, 6982, 7006

\ior_open_streams: 5122, 5125
\ior_raw_new:c 105, 5075
\ior_raw_new:N 105, 4952, 4959, 4965, 4969, 5067
\ior_stream_alloc:N 4998, 5018, 5058
\ior_stream_alloc_aux: 5018, 5064, 5083, 5091, 5093
\ior_to:NN 104, 5181, 5181, 5185
\iow_alloc_write:n 4981, 5004, 5004
\iow_char:N 104, 4440, 4618, 4797, 4799, 5174, 5174
\iow_close:c 102, 5098
\iow_close:N . . 102, 4979, 5098, 5098, 5109
\iow_log:n 103, 5146, 5147, 7054, 7056, 7058
\iow_log:x 15, 103, 1108, 1108, 1132, 1635, 5146, 5146, 5317
\iow_new:c 102, 4970
\iow_new:N 102, 4970, 4970, 4973
\iow_newline: 104, 5173, 5173
\iow_now:Nn 103, 5143, 5143, 5147, 5149, 5151, 5162
\iow_now:Nx 103, 5142, 5142, 5144, 5146, 5148, 5155, 5159
\iow_now_buffer_safe:Nn 103, 5158, 5158
\iow_now_buffer_safe:Nx 103, 5158, 5161
\iow_now_buffer_safe_aux:w 5159, 5162, 5164
\iow_now_buffer_safe_expanded_aux:w 5158
\iow_now_when_avail:cn 103, 5150
\iow_now_when_avail:cx 103, 5150
\iow_now_when_avail:Nn 103, 5150, 5150, 5153
\iow_now_when_avail:Nx 103, 5150, 5154, 5157
\iow_open:cn 102, 4978
\iow_open:Nn . . 102, 4978, 4978, 4990, 5791
\iow_open_streams: 102, 5122, 5122
\iow_raw_new:c 105, 5035
\iow_raw_new:N 105, 4952, 4955, 4964, 4968, 5027
\iow_shipout:Nn . . . 103, 5169, 5169, 5172
\iow_shipout:Nx 103, 5169
\iow_shipout_x:Nn 15, 104, 781, 781, 1109, 1112, 5142, 5167, 5167, 5168, 5170, 5776
\iow_shipout_x:Nx 104, 5167
\iow_stream_alloc:N 4985, 5018, 5018
\iow_stream_alloc_aux: 5018, 5024, 5043, 5051, 5053
\iow_term:n 103, 5146, 5149
\iow_term:x 15, 103, 1108, 1111, 1115, 1131, 4432, 4610, 4789, 5146, 5148, 5324

J

\jobname 369, 5791, 5795

K

\K 2463
\kern 247
\keys_bool_set:NN 129, 6058, 6058, 6316, 6319
\keys_choice_code_store:x 129, 6074, 6074, 6325, 6328
\keys_choice_find:n 130, 6077, 6077, 6084
\keys_choice_make: 130, 6068, 6082, 6082, 6092, 6322
\keys_choices_generate:n 130, 6091, 6091, 6355
\keys_choices_generate_aux:n 6091, 6104, 6106
\keys_cmd_set:nn 130, 6083, 6086, 6115, 6115, 6208, 6331
\keys_cmd_set:nx . . . 130, 6059, 6063, 6107, 6115, 6120, 6212, 6310, 6334
\keys_cmd_set_aux:n 6115, 6116, 6121, 6125
\keys_default_set:n 130, 6072, 6131, 6131, 6136, 6337
\keys_default_set:V 130, 6131, 6340
\keys_define:nn . . . 124, 6137, 6137, 6434
\keys_define_elt:n . 130, 6140, 6142, 6142
\keys_define_elt:nn 130, 6140, 6142, 6146
\keys_define_elt_aux:nn 6144, 6148, 6150, 6150
\keys_define_key:n . 130, 6155, 6161, 6161
\keys_execute: 130, 6176, 6176, 6279, 6285
\keys_execute_aux:nn 6078, 6079, 6176, 6177, 6182, 6187
\keys_execute_unknown: 130, 6176, 6178, 6181
\keys_if_exist:nn 6195
\keys_if_exist:nnTF 129, 6195
\keys_if_value_requirement:nTF 130, 6202, 6202, 6265, 6277
\keys_meta_make:n . 130, 6207, 6207, 6370
\keys_meta_make:x . 130, 6207, 6211, 6373
\keys_property_find:n 131, 6151, 6216, 6216
\keys_property_find_aux:n 6216, 6219, 6224

\keys_property_find_aux:w 6216, 6225, 6227, 6232
\keys_property_new:nn 131, 6238, 6238, 6321, 6411, 6414
\keys_property_new_arg:nn 131, 6238, 6241, 6315, 6318, 6324, 6327, 6330, 6333, 6336, 6339, 6342, 6345, 6348, 6351, 6354, 6357, 6360, 6363, 6366, 6369, 6372, 6375, 6378, 6381, 6384, 6387, 6390, 6393, 6396, 6399, 6402, 6405, 6408
\keys_property_undefine:n 131, 6126, 6244, 6244
\keys_set:nn 128, 6209, 6213, 6247, 6247, 6252
\keys_set:nV 128, 6247
\keys_set:nv 128, 6247
\keys_set_elt:n . . . 131, 6250, 6253, 6253
\keys_set_elt:nn . . 131, 6250, 6253, 6257
\keys_set_elt_aux: 6261, 6270, 6273, 6276
\keys_set_elt_aux:nn 6255, 6259, 6261, 6261
\keys_show:nn 129, 6288, 6288
\keys_tmp:w 131, 6152, 6154, 6167, 6173, 6188–6190, 6291, 6291
\keys_value_or_default:n 131, 6264, 6292, 6292
\keys_value_requirement:n 131, 6302, 6302, 6412, 6415
\keys_variable_set:cnNN 131, 6306, 6346, 6352, 6361, 6367, 6379, 6385, 6391, 6397, 6403, 6409
\keys_variable_set:NnNN . . 131, 6306, 6306, 6314, 6343, 6349, 6358, 6364, 6376, 6382, 6388, 6394, 6400, 6406
\KV_add_element_aux:w . . 5917, 5922, 5922
\KV_add_value_element:w 332, 5891, 5900, 5997
\KV_key_no_value_elt:n . . . 121, 5864, 5868, 5873, 5959, 6004, 6014, 6014
\KV_key_value_elt:nn 121, 5867, 5869, 5875, 5977, 6009, 6014, 6015
\KV_parse_elt:w 5929, 5937, 5941, 5941, 5943, 5948
\KV_parse_no_sanitize_aux:n 5934, 5934, 6018, 6026
\KV_parse_no_space_removal_no_sanitize:n 121, 5859, 6016, 6016
\KV_parse_sanitize_aux:n 5923, 5923, 6022
\KV_parse_space_removal_no_sanitize:n 121, 5856, 6020, 6024
\KV_parse_space_removal_sanitize:n 121, 5853, 6020, 6020
\KV_process_aux:NNNn 5852, 5853, 5856, 5859, 5861
\KV_process_no_space_removal_no_sanitize:NNn 121, 5852, 5858, 6139
\KV_process_space_removal_no_sanitize:NNn 120, 5852, 5855
\KV_process_space_removal_sanitize:NNn 120, 5852, 5852, 6249
\KV_remove_surrounding_spaces:nw 332, 5891, 5893
\KV_remove_surrounding_spaces_auxi:w . . . 332, 5891, 5894, 5896, 5902, 5911
\KV_remove_surrounding_spaces_auxii:w 5891, 5897, 5899
\KV_sanitize_outerlevel_active_commas:N 332, 5878, 5887, 5928
\KV_sanitize_outerlevel_active_equals:N 332, 5878, 5884, 5927
\KV_set_key_element:w 332, 5891, 5908, 5954
\KV_split_key_value_current:w 332, 5947, 5952, 5952, 6017, 6021, 6025
\KV_split_key_value_no_space_removal:w 332, 6000, 6000, 6017
\KV_split_key_value_space_removal:w 332, 5953, 5953, 6021, 6025
\KV_split_key_value_space_removal_aux:w 5953, 5965, 5974
\KV_split_key_value_space_removal_detect_error:wTF 332, 5953, 5964, 5970
\KV_val_preserve_braces:NnN . 5973, 5993

L

\L 2463
\l_calc_B_dim . . 353, 6482, 6486, 6528, 6531, 6534, 6537, 6800, 6802, 6803
\l_calc_B_int 353, 6482, 6483, 6510, 6513, 6516, 6519, 6522, 6525, 6540, 6543, 6696, 6748, 6751, 6755, 6794, 6796, 6797
\l_calc_B_muskip 353, 6482, 6492, 6564, 6568, 6572, 6576, 6580, 6584, 6812, 6814, 6815
\l_calc_B_register 353, 6478, 6480, 6496, 6505, 6507, 6605, 6611, 6612, 6667, 6675, 6681, 6684, 6690, 6696, 6701, 6707, 6710, 6716, 6758, 6764, 6770, 6776

\l_calc_B_skip *353*, 6482, 6489, 6546, 6549, 6552, 6555, 6558, 6561, 6806, 6808, 6809
\l_calc_C_dim *353*, 6482, 6487, 6799, 6802, 6803
\l_calc_C_int *353*, 6482, 6484, 6749, 6752, 6755, 6793, 6796, 6797
\l_calc_C_muskip *353*, 6482, 6493, 6811, 6814, 6815
\l_calc_C_skip *353*, 6482, 6490, 6805, 6808, 6809
\l_calc_current_type_int . . *353*, 6478, 6481, 6497, 6676, 6685, 6697, 6702, 6711, 6747, 6759, 6771, 6792, 6822
\l_calc_expression_tl *353*, 6478, 6478, 6501, 6504
\l_clist_remove_clist 4503, 4713, 4713, 4715, 4717, 4720, 4721, 4737, 4740, 4744
\l_exp_tl *182*, 1471, 1471, 1498, 1499
\l_file_name_tl 6980, 6980, 6987, 6992, 7009, 7023, 7040
\l_file_search_path_saved_seq 6975, 6977, 6997, 7018
\l_file_search_path_seq 6974, 6974, 6998, 7000, 7004, 7017, 7046, 7047, 7050
\l_ior_stream_int *105*, 4950, 4951, 4993, 4995, 4999, 5014, 5059, 5060, 5065, 5068, 5075, 5078, 5084, 5086, 5088, 5090
\l_iow_stream_int *105*, 4950, 4950, 4951, 4980, 4982, 4986, 5007, 5019, 5020, 5025, 5028, 5030, 5035, 5038, 5044, 5046, 5048, 5050, 5070
\l_kernel_testa_tl . . . *77*, 3716, 3716, 3845, 3847, 3852, 3854, 3859, 3861, 3866, 3868, 3873, 3875, 3880, 3882, 3887, 3889, 3894, 3896, 3901, 3903, 3908, 3910, 3915, 3917, 3922, 3924, 3929, 3931, 3936, 3938, 4398, 4564
\l_kernel_testb_tl *77*, 3716, 3717, 3846, 3847, 3853, 3854, 3860, 3861, 3867, 3868, 3874, 3875, 3881, 3882, 3888, 3889, 3895, 3896, 3902, 3903, 3909, 3910, 3916, 3917, 3923, 3924, 3930, 3931, 3937, 3938
\l_kernel_tmpa_tl *77*, 600, 602, 603, 614, 616, 3720, 3720, 3829, 3837, 4047, 4050
\l_kernel_tmpb_tl *77*, 601, 602, 3720, 3721
\l_keys_choice_code_tl *132*, 6051, 6051, 6097, 6111
\l_keys_choice_int *128*, 6049, 6049, 6093, 6109, 6110, 6113
\l_keys_choice_tl . *128*, 6049, 6050, 6108
\l_keys_key_tl *129*, 6052, 6052, 6262, 6263
\l_keys_module_tl *132*, 6055, 6055, 6138, 6182, 6184, 6209, 6213, 6217, 6248, 6263
\l_keys_no_value_bool *132*, 6056, 6056, 6143, 6147, 6162, 6254, 6258, 6266, 6278, 6294
\l_keys_path_tl *132*, 6052, 6053, 6059, 6063, 6075, 6078, 6079, 6083, 6086, 6088, 6095, 6098, 6102, 6107, 6132–6134, 6158, 6170, 6177, 6183, 6204, 6208, 6212, 6217, 6229, 6230, 6234, 6263, 6268, 6282, 6295, 6297, 6303, 6310, 6331, 6334
\l_keys_property_tl *132*, 6052, 6054, 6153, 6158, 6165, 6170, 6235
\l_keys_value_tl *132*, 6057, 6057, 6190, 6282, 6293, 6296
\l_KV_currkey_tl *122*, 5846, 5847, 5909, 5916, 5917, 5960, 5978
\l_KV_currval_tl *122*, 5846, 5848, 5901, 5905
\l_KV_level_int 5849, 5849, 5863, 5866, 5870, 5872, 5874, 5876
\l_KV_parse_tl *122*, 5846, 5846, 5904, 5925, 5932, 5936, 5939, 5958, 5976, 5981, 5987, 5992, 6004, 6009
\l_KV_remove_one_level_of_braces_bool . . . *121*, 5850, 5850, 5851, 5914, 5985
\l_KV_tmpa_tl *122*, 5843, 5843, 5926–5928, 5930, 5955, 5956, 5975, 5980, 5983, 6001, 6002, 6007
\l_KV_tmpb_tl *122*, 5843, 5844
\l_last_box *115*, 5636, 5636, 5637
\l_msg_class_tl *112*, 5239, 5239, 5408, 5409, 5412
\l_msg_coding_error_text_tl 6426, 6448, 6455
\l_msg_current_class_tl *112*, 5239, 5240, 5388, 5409
\l_msg_current_module_tl *112*, 5239, 5241, 5389
\l_msg_names_clist *112*, 5242, 5242, 5276, 5277
\l_msg_redirect_classes_clist *112*, 5245, 5245, 5364, 5368, 5371

\l_msg_redirect_classes_prop *112*, 5243, 5243
\l_msg_redirect_names_prop *112*, 5243, 5244, 5390, 5477
\l_msg_tmp_tl 5246, 5246, 5293, 5308, 5310
\l_peek_false_tl . . . *214*, 2626, 2627, 2634, 2649, 2667, 2674, 2683, 2690
\l_peek_search_tl *214*, 2629, 2629, 2632, 2646, 2684
\l_peek_search_token *52*, 2619, 2621, 2631, 2645, 2664, 2671
\l_peek_token *52*, 2619, 2619, 2622, 2664, 2671, 2679–2681, 2772
\l_peek_true_aux_tl *214*, 2647, 2659, 2659, 2661
\l_peek_true_tl *214*, 2626, 2626, 2633, 2648, 2665, 2672, 2688
\l_seq_remove_seq 4503, 4505, 4507, 4510, 4511
\l_tl_replace_tl *77*
\l_tl_replace_toks *84*, 4359, 4359
\l_tmpa_bool *193*, 1894, 1894
\l_tmpa_box *115*, 5661, 5662, 5664, 6723, 6726
\l_tmpa_dim *68*, 3467, 3467
\l_tmpa_int *58*, 2986, 2986
\l_tmpa_skip *65*, 3392, 3392–3394
\l_tmpa_tl *5*, *77*, 3718, 3718, 4034, 4040, 4067, 4701, 4704, 4705
\l_tmpa_toks *84*, 4351, 4351, 4434, 4436, 4437, 4439, 4443, 4612, 4614, 4615, 4617, 4621, 4791, 4793, 4794, 4796, 4802, 5741, 5747
\l_tmpb_box *115*, 5661, 5665
\l_tmpb_dim *68*, 3467, 3468
\l_tmpb_int *58*, 2986, 2987
\l_tmpb_skip *65*, 3392, 3395
\l_tmpb_tl *77*, 3718, 3719, 4702, 4705, 4706
\l_tmpb_toks *84*, 4351, 4353
\l_tmpc_dim *68*, 3467, 3469
\l_tmpc_int *58*, 2986, 2988
\l_tmpc_skip *65*, 3392, 3396
\l_tmpc_toks *84*, 4351, 4354
\l_tmpd_dim *68*, 3467, 3470
\l_xref_curr_name_tl 5801
\language 164
\lastbox 321
\lastkern 254
\lastlinefit 434
\lastnodetype 415
\lastpenalty 360
\lastskip 255
\lccode 383
\leaders 251
\left 219
\lefthyphenmin 275
\leftmarginkern 497
\leftskip 277
\leqno 194
\let 40, 64
\limits 211
\linepenalty 267
\lineskip 261
\lineskiplimit 262
\locbox 5602
\loccount 2803
\locdimen 3439
\locmuskip 3505
\locskip 3317
\loctoks 4218
\long 43, 49, 52, 57, 83
\looseness 279
\lower 316
\lowercase 355
\lpcode 464
\luatex_directlua:D 554, 1288
\luatex_if_engine: 1297
\luatex_if_engine:TF *26*, 1279

M

\M 2463, 2550
\m@ne 1100
\mag 163
\mark 165
\marks 391
\mathaccent 176
\mathbin 206
\mathchar 177
\mathchardef 74
\mathchoice 174
\mathclose 207
\mathcode 385
\mathinner 208
\mathop 209
\mathopen 213
\mathord 214
\mathparagraph 2966
\mathpunct 215
\mathrel 216
\mathsection 2965
\mathsurround 227

\maxdeadcycles 297
\maxdepth 298
\maxdimen 3407
\maxof *133*, 6838, 6844
\meaning 357
\medmuskip 228
\message 134
\MessageBreak 733
.meta:n *126*
.meta:x *126*
\middle 439
\minof *133*, 6838, 6845
\mkern 181
\mode_if_horizontal: 1737
\mode_if_horizontal:TF *40*, 1737
\mode_if_horizontal_p: *40*, 1737
\mode_if_inner: 1741
\mode_if_inner:TF *40*, 1741
\mode_if_inner_p: *40*, 1741
\mode_if_math: 1745
\mode_if_math:TF *40*, 1745, 2950
\mode_if_math_p: *40*, 1745
\mode_if_vertical: 1733
\mode_if_vertical:TF *39*, 1733
\mode_if_vertical_p: *39*, 1733
\month 367
\moveleft 317
\moveright 318
\msg_class_new:nn *107*, 5339, 5339, 5415, 5426, 5443, 5450, 5457, 5463, 5469
\msg_class_set:nn . *107*, 5339, 5342, 5344
\msg_direct_interrupt:n 5280
\msg_direct_interrupt:xxxx *110*, 5280, 5287, 5416, 5427, 5512, 5525, 5576
\msg_direct_interrupt_aux:n . 5291, 5306
\msg_direct_log:xx *110*, 5313, 5313, 5451, 5458, 5464
\msg_direct_term:xx *110*, 5313, 5320, 5444
\msg_error:nn *107*, 5426
\msg_error:nnx *107*, 5426
\msg_error:nnxx *107*, 5426
\msg_error:nnxxx *107*, 5426
\msg_error:nnxxxx *107*, 5426
\msg_fatal:nn *107*, 5415
\msg_fatal:nnx *107*, 5415
\msg_fatal:nnxx *107*, 5415
\msg_fatal:nnxxx *107*, 5415
\msg_fatal:nnxxxx *107*, 5415
\msg_generic_new:nn *110*, 5257, 5261, 5331
\msg_generic_new:nnn *110*, 5257, 5257, 5328
\msg_generic_set:nn *110*, 5263, 5265, 5270, 5337
\msg_generic_set:nnn *110*, 5259, 5265, 5265, 5334
\msg_generic_set_clist:n 5265, 5266, 5271, 5275
\msg_info:nn *108*, 5450
\msg_info:nnx *108*, 5450
\msg_info:nnxx *108*, 5450
\msg_info:nnxxx *108*, 5450
\msg_info:nnxxxx *108*, 5450, 5549
\msg_kernel_bug:x *15*, *111*, 1114, 1114, 1124, 1142, 1354, 3544, 4399, 4565, 4584, 5575, 5575
\msg_kernel_classes_new:n 5491, 5491, 5523, 5543, 5547, 5551
\msg_kernel_error:nn *111*, 4983, 4996, 5369, 5524, 5966, 6011
\msg_kernel_error:nnx *111*, 5375, 5524, 6101, 6221, 6267
\msg_kernel_error:nnxx . . . *111*, 5382, 5524, 6087, 6157, 6169, 6183, 6281
\msg_kernel_error:nnxxx *111*, 5524
\msg_kernel_error:nnxxxx *111*, 5524, 5524
\msg_kernel_fatal:nn *111*, 5511
\msg_kernel_fatal:nnx *111*, 5511
\msg_kernel_fatal:nnxx *111*, 5511
\msg_kernel_fatal:nnxxx *111*, 5511
\msg_kernel_fatal:nnxxxx *111*, 5511, 5511
\msg_kernel_info:nn *111*, 5544
\msg_kernel_info:nnx *111*, 5544
\msg_kernel_info:nnxx *111*, 5544
\msg_kernel_info:nnxxx *111*, 5544
\msg_kernel_info:nnxxxx *111*, 5544, 5548
\msg_kernel_new:nnn *110*, 5479, 5482
\msg_kernel_new:nnnn *110*, 5128, 5135, 5479, 5479, 5552, 5560, 5568, 6028, 6417, 6423, 6430, 6439, 6445, 6452, 6459, 6465
\msg_kernel_set:nnn *110*, 5479, 5488
\msg_kernel_set:nnnn . . . *110*, 5479, 5485
\msg_kernel_warning:nn *111*, 5544
\msg_kernel_warning:nnx *111*, 5544
\msg_kernel_warning:nnxx *111*, 5544
\msg_kernel_warning:nnxxx *111*, 5544
\msg_kernel_warning:nnxxxx *111*, 5544, 5544
\msg_line_context: . *109*, 5247, 5250, 6029
\msg_line_number: . *109*, 5247, 5247, 5253
\msg_log:nn *108*, 5457

\msg_log:nnx *108*, 5457
\msg_log:nnxx *108*, 5457
\msg_log:nnxxx *108*, 5457
\msg_log:nnxxxx *108*, 5457
\msg_new:nnn *106*, 5327, 5330, 5483
\msg_new:nnnn *106*, 5327, 5327, 5480
\msg_newline: *110*, 5214, 5225, 5255, 5255, 5289, 5292, 5296, 5315, 5317, 5322, 5324, 5428, 5526
\msg_none:nn *108*, 5469
\msg_none:nnx *108*, 5469
\msg_none:nnxx *108*, 5469
\msg_none:nnxxx *108*, 5469
\msg_none:nnxxxx *108*, 5469
\msg_redirect_class:nn . *108*, 5470, 5470
\msg_redirect_module:nnn *109*, 5473, 5473
\msg_redirect_name:nnn . *109*, 5476, 5476
\msg_set:nnn *106*, 5327, 5336, 5489
\msg_set:nnnn *106*, 5327, 5333, 5486
\msg_trace:nn *108*, 5463
\msg_trace:nnx *108*, 5463
\msg_trace:nnxx *108*, 5463
\msg_trace:nnxxx *108*, 5463
\msg_trace:nnxxxx *108*, 5463
\msg_two_newlines: *110*, 5201, 5223, 5255, 5256, 5297, 5417, 5513
\msg_use:nnnnxxxx 5347, 5362, 5362
\msg_use_aux:nn 5393, 5396, 5396
\msg_use_aux:nnn 5380, 5387, 5387
\msg_use_code: 5363, 5385, 5385, 5403, 5410
\msg_use_loop: 5385
\msg_use_loop:n 5367, 5386, 5412
\msg_use_loop_check:nn 5391, 5398, 5401, 5407, 5407
\msg_warning:nn *107*, 5443
\msg_warning:nnx *107*, 5443
\msg_warning:nnxx *107*, 5443
\msg_warning:nnxxx *107*, 5443
\msg_warning:nnxxxx *107*, 5443, 5545
\mskip 178
\muexpr 427
\multiply 79
\muskip 375
\muskip_add:Nn *69*, 3509, 3511, 3512, 6573, 6581
\muskip_gadd:Nn *69*, 3509, 3512, 6577, 6585
\muskip_gset:Nn *68*, 3509, 3510, 6569
\muskip_gsub:Nn *69*, 3509, 3514
\muskip_new:N . . *68*, 3493, 3497, 6491–6493
\muskip_new_local:N *68*, 3493, 3501
\muskip_set:Nn . *68*, 3509, 3509, 3510, 6565
\muskip_sub:Nn *69*, 3509, 3513, 3514
\muskip_use:N *69*, 3515, 3515
\muskipdef 73
\mutoglue 433

N

\n 2459
\name_pop_stack:w 698, 709, 728
\name_primitive:NN 57, 61–451, 453–473, 475–481, 483–486, 488–509, 511–522, 524–554, 557
\name_tmp: 728
\name_undefine:N 43, 49, 52, 59, 697
\NeedsTeXFormat 720
\newbox 5596, 5602
\newcount 2797, 2803
\newdimen 3433, 3439
\newif 6889
\newlinechar 129
\newmuskip 3499, 3505
\newread 4965
\newskip 3311, 3317
\newtoks 4212, 4218
\newwrite 4964
\noalign 97
\noboundary 232
\noexpand 32, 90
\noindent 258
\nolimits 212
\nonscript 192
\nonstopmode 155
\nulldelimiterspace 225
\nullfont 343
\number 352, 5807
\numexpr 424

O

\O 1652, 2463, 2554
\omit 98
\openin 124
\openout 125
\or 121
\or: *8*, *61*, 760, 762, 868, 1324, 1326, 1328, 1330, 1332, 1334, 1336, 1338, 1340, 2925–2929, 2934–2938, 2962–2970, 2975–2983, 3100–3108, 6677–6679, 6686–6688, 6703–6705, 6712–6714, 6760–6762, 6772–6774, 6798, 6804

\outer 84
\output 299
\outputpenalty 309
\over 186
\overfullrule 337
\overline 217
\overwithdelims 187

P

\P 1650, 2463
\p 2146
\package_check_loaded_expl: 730, 756, 1463, 1730, 2164, 2267, 2785, 3143, 3302, 3520, 4203, 4367, 4539, 4768, 4913, 5191, 5586, 5728, 5840, 6042, 6475, 6951
\package_provides:w 703, 704
\PackageError 16, 732, 6834
\pagedepth 301
\pagediscards 442
\pagefilllstretch 305
\pagefillstretch 304
\pagefilstretch 303
\pagegoal 307
\pagenumbering 5814, 5822
\pageshrink 306
\pagestretch 302
\pagetotal 308
\par 257, 696
\parfillskip 288
\parindent 281
\parshape 273
\parshapedimen 423
\parshapeindent 421
\parshapelength 422
\parskip 280
\patterns 363
\pausing 150
\pdf_adjustspacing:D 461
\pdf_annot:D 530
\pdf_catalog:D 540
\pdf_compresslevel:D 455
\pdf_creationdate:D 490
\pdf_decimaldigits:D 456
\pdf_dest:D 534
\pdf_destmargin:D 480
\pdf_efcode:D 463
\pdf_elapsedtime:D 520
\pdf_endlink:D 532
\pdf_endthread:D 537
\pdf_escapehex:D 501
\pdf_escapename:D 500
\pdf_escapestring:D 499
\pdf_filedump:D 509
\pdf_filemoddate:D 507
\pdf_filesize:D 508
\pdf_fontattr:D 544
\pdf_fontexpand:D 546
\pdf_fontname:D 493
\pdf_fontobjnum:D 494
\pdf_fontsize:D 495
\pdf_forcepagebox:D 466
\pdf_gamma:D 472
\pdf_horigin:D 475
\pdf_imageapplygamma:D 471
\pdf_imagegamma:D 473
\pdf_imagehicolor:D 470
\pdf_imageresolution:D 457
\pdf_includechars:D 496
\pdf_inclusionerrorlevel:D 468
\pdf_info:D 539
\pdf_lastannot:D 516
\pdf_lastdemerits:D 519
\pdf_lastobj:D 512
\pdf_lastxform:D 513
\pdf_lastximage:D 514
\pdf_lastximagepages:D 515
\pdf_lastxpos:D 517
\pdf_lastypos:D 518
\pdf_leftmarginkern:D 497
\pdf_linkmargin:D 479
\pdf_literal:D 548
\pdf_lpcode:D 464
\pdf_mapfile:D 542
\pdf_mapline:D 543
\pdf_mdfivesum:D 506
\pdf_minorversion:D 454
\pdf_names:D 541
\pdf_noligatures:D 552
\pdf_normaldeviate:D 505
\pdf_obj:D 524
\pdf_optionalwaysusepdfpagebox:D . . 467
\pdf_optionpdfinclusionerrorlevel:D 469
\pdf_outline:D 533
\pdf_output:D 453
\pdf_pageattr:D 484
\pdf_pageheight:D 478
\pdf_pageref:D 491
\pdf_pageresources:D 485
\pdf_pagesattr:D 483

\pdf_pagewidth:D 477
\pdf_pkmode:D 486
\pdf_pkresolution:D 458
\pdf_protrudechars:D 462
\pdf_randomseed:D 521
\pdf_refobj:D 525
\pdf_refxform:D 527
\pdf_refximage:D 529
\pdf_resettimer:D 550
\pdf_rightmarginkern:D 498
\pdf_rpcode:D 465
\pdf_savepos:D 538
\pdf_setrandomseed:D 551
\pdf_shellescape:D 522
\pdf_startlink:D 531
\pdf_startthread:D 536
\pdf_strcmp:D . . 503, 557, 2213, 2235, 3788
\pdf_texbanner:D 489
\pdf_texrevision:D 488
\pdf_texversion:D 511
\pdf_thread:D 535
\pdf_threadmargin:D 481
\pdf_tracingfonts:D 459
\pdf_trailer:D 545
\pdf_unescapehex:D 502
\pdf_uniformdeviate:D 504
\pdf_uniqueresname:D 460
\pdf_vorigin:D 476
\pdf_xform:D 526
\pdf_xformname:D 492
\pdf_ximage:D 528
\pdfadjustspacing 461
\pdfannot 530
\pdfcatalog 540
\pdfcompresslevel 455
\pdfcreationdate 490
\pdfdecimaldigits 456
\pdfdest 534
\pdfdestmargin 480
\pdfelapsedtime 520
\pdfendlink 532
\pdfendthread 537
\pdfescapehex 501
\pdfescapename 500
\pdfescapestring 499
\pdffiledump 509
\pdffilemoddate 507
\pdffilesize 508
\pdffontattr 544
\pdffontexpand 546
\pdffontname 493
\pdffontobjnum 494
\pdffontsize 495
\pdfforcepagebox 466
\pdfgamma 472
\pdfhorigin 475
\pdfimageapplygamma 471
\pdfimagegamma 473
\pdfimagehicolor 470
\pdfimageresolution 457
\pdfincludechars 496
\pdfinclusionerrorlevel 468
\pdfinfo 539
\pdflastannot 516
\pdflastdemerits 519
\pdflastobj 512
\pdflastxform 513
\pdflastximage 514
\pdflastximagepages 515
\pdflastxpos 517
\pdflastypos 518
\pdflinkmargin 479
\pdfliteral 548
\pdfmapfile 542
\pdfmapline 543
\pdfmdfivesum 506
\pdfminorversion 454
\pdfnames 541
\pdfnoligatures 552
\pdfnormaldeviate 505
\pdfobj 524
\pdfoptionalwaysusepdfpagebox 467
\pdfoptionpdfinclusionerrorlevel . . 469
\pdfoutline 533
\pdfoutput 453
\pdfpageattr 484
\pdfpageheight 478
\pdfpageref 491
\pdfpageresources 485
\pdfpagesattr 483
\pdfpagewidth 477
\pdfpkmode 486
\pdfpkresolution 458
\pdfprotrudechars 462
\pdfrandomseed 521
\pdfrefobj 525
\pdfrefxform 527
\pdfrefximage 529
\pdfresettimer 550
\pdfsavepos 538

\pdfsetrandomseed 551
\pdfshellescape 522
\pdfstartlink 531
\pdfstartthread 536
\pdfstrcmp 503
\pdftexbanner 489
\pdftexrevision 488
\pdftexversion 511
\pdfthread 535
\pdfthreadmargin 481
\pdftracingfonts 459
\pdftrailer 545
\pdfunescapehex 502
\pdfuniformdeviate 504
\pdfuniqueresname 460
\pdfvorigin 476
\pdfxform 526
\pdfxformname 492
\pdfximage 528
\peek_after:NN *52*, 2622, 2622, 2636, 2651, 2769
\peek_catcode:NTF *53*, 2728
\peek_catcode_ignore_spaces:NTF *53*, 2733
\peek_catcode_remove:NTF *53*, 2738
\peek_catcode_remove_ignore_spaces:NTF *53*, 2743
\peek_charcode:NTF *53*, 2748
\peek_charcode_ignore_spaces:NTF *53*, 2753
\peek_charcode_remove:NTF *53*, 2758
\peek_charcode_remove_ignore_spaces:NTF *53*, 2763
\peek_def_aux:nnnn 2694, 2694, 2708, 2713, 2718, 2723, 2728, 2733, 2738, 2743, 2748, 2753, 2758, 2763
\peek_def_aux_ii:nnnnn 2694, 2695–2697, 2699
\peek_execute_branches: 2702, 2776
\peek_execute_branches_catcode: *54*, 2663, 2670, 2732, 2735, 2742, 2745
\peek_execute_branches_charcode: . . . *54*, 2663, 2677, 2752, 2755, 2762, 2765
\peek_execute_branches_charcode_aux:NN 2663, 2684, 2686
\peek_execute_branches_meaning: *54*, 2663, 2663, 2712, 2715, 2722, 2725
\peek_gafter:NN *52*, 2622, 2623
\peek_ignore_spaces_aux: *214*, 2768, 2768, 2773
\peek_ignore_spaces_execute_branches: *214*, 2717, 2727, 2737, 2747, 2757, 2767, 2768, 2769, 2771
\peek_meaning:NTF *53*, 2708
\peek_meaning_ignore_spaces:NTF *53*, 2713
\peek_meaning_remove:NTF *53*, 2718
\peek_meaning_remove_ignore_spaces:NTF *53*, 2723
\peek_tmp:w . . . *214*, 2628, 2628, 2661, 2774
\peek_token_generic:NNF 2641
\peek_token_generic:NNT 2638
\peek_token_generic:NNTF *53*, 2630, 2630, 2639, 2642
\peek_token_remove_generic:NNF . . 2656
\peek_token_remove_generic:NNT . . 2653
\peek_token_remove_generic:NNTF *53*, 2644, 2644, 2654, 2657
\penalty 358
\postdisplaypenalty 205
\predisplaydirection 449
\predisplaypenalty 204
\predisplaysize 203
\pref_global:D *24*, 795, 795, 1256, 1262, 2320, 2624, 2818, 2836, 2842, 2854, 2872, 2878, 3333, 3348, 3370, 3378, 3447, 3451, 3457, 3461, 3510, 3512, 3514, 3707, 4244, 4245, 4256, 4261, 4271, 4281, 4295, 4309, 4462, 4471, 5185, 5634, 5639, 5647, 5669, 5675, 5681, 5695, 5701, 5709, 6505, 6608, 6611, 6612, 6667, 6795, 6801, 6807, 6813
\pref_global_chk: 2816, 2834, 2840, 2870, 2876, 3331, 3346, 3368, 3376, 3705, 4270, 4280, 4294, 4308, 4459, 4468
\pref_long:D *24*, 795, 796, 801, 804, 813, 816, 821, 824, 833, 836
\pref_protected:D *24*, 795, 797, 800, 803, 806, 807, 809, 810, 813, 816, 827, 830, 833, 836
\pretolerance 284
\prevdepth 331
\prevgraf 290
\prg_case_dim:nnn *38*, 2018, 2018
\prg_case_dim_aux:nnn 2018, 2019, 2022, 2026
\prg_case_int:nnn *38*, 2008, 2008
\prg_case_int_aux:nnn 2008, 2009, 2012, 2016

\prg_case_str:nnn *39*, 2028, 2028
\prg_case_str_aux:nnn 2028, 2029, 2032, 2036
\prg_case_tl:Nnn *39*, 2038, 2038
\prg_case_tl_aux:NNn 2038, 2039, 2042, 2046
\prg_conditional_form_F:nnn 1009
\prg_conditional_form_p:nnn 1006
\prg_conditional_form_T:nnn 1008
\prg_conditional_form_TF:nnn 1007
\prg_define_quicksort:nnn 2049, 2049, 2124
\prg_do_nothing: *15*, 1301, 1301, 4103, 4110, 4133, 4139, 4231, 4260, 6623
\prg_end_case:nw 2015, 2025, 2035, 2045, 2048, 2048
\prg_generate_conditional_aux:nnNNnnnn 884, 890, 896, 902, 908, 914, 921, 928, 940
\prg_generate_conditional_aux:nnw 941, 945, 950
\prg_generate_conditional_parm_aux:nnNNnnnn 940
\prg_generate_conditional_parm_aux:nw 940
\prg_generate_F_form_count:Nnnnn 972, 987
\prg_generate_F_form_parm:Nnnnn 952, 967
\prg_generate_p_form_count:Nnnnn 972, 972
\prg_generate_p_form_parm:Nnnnn 952, 952
\prg_generate_T_form_count:Nnnnn 972, 982
\prg_generate_T_form_parm:Nnnnn 952, 962
\prg_generate_TF_form_count:Nnnnn 972, 977
\prg_generate_TF_form_parm:Nnnnn 952, 957
\prg_get_count_aux:nn 907, 913, 920, 927, 938, 938
\prg_get_parm_aux:nw 883, 889, 895, 901, 938, 939
\prg_new_conditional:Nnn *34*, 906, 912, 1733, 2206, 2212, 2221, 2230, 2234, 2243, 2382, 2386, 2390, 2394, 2398, 2402, 2406, 2410, 2414, 2418, 2422, 2426, 2430, 2434, 2438, 2445, 2448, 2467, 2474, 2481, 2492, 2503, 2514, 2525, 2532, 2539, 2579, 3409, 3477, 3763, 3774, 4336, 4343, 5176, 5611, 5614, 5625
\prg_new_conditional:Npnn *34*, 882, 888, 1284, 1733, 3730, 3737, 3749, 3830, 4170, 4174, 4192
\prg_new_eq_conditional:NNn *34*, 932, 935, 1733, 4394, 4395, 4560, 4561, 4568–4571, 4868–4873
\prg_new_protected_conditional:Nnn *34*, 906, 926, 1733, 4402, 4572, 4874, 6981
\prg_new_protected_conditional:Npnn *34*, 882, 900, 1733, 4069, 4078
\prg_quicksort:n *41*, 2124
\prg_quicksort_compare:nnTF *41*, 2125, 2126
\prg_quicksort_function:n *41*, 2125, 2125
\prg_replicate:nn *41*, 1762, 1762
\prg_replicate_ 1774
\prg_replicate_aux:N 1762, 1768, 1769, 1772
\prg_replicate_first_aux:N 1762, 1764, 1771
\prg_return_false: *34*, 880, 881, 1050, 1055, 1067, 1072, 1090, 1093, 1155, 1157, 1174, 1286, 1295, 1299, 1442, 1733, 1735, 1739, 1743, 1747, 1897, 1994, 2208, 2216, 2226, 2231, 2238, 2248, 2384, 2388, 2392, 2396, 2400, 2404, 2408, 2412, 2416, 2420, 2424, 2428, 2432, 2436, 2443, 2447, 2451, 2453, 2472, 2479, 2483, 2490, 2494, 2501, 2512, 2516, 2523, 2530, 2537, 2545, 2582, 2589, 2594, 2601, 2603, 2605, 2607, 2609, 3173, 3177, 3181, 3185, 3189, 3193, 3197, 3202, 3267, 3271, 3413, 3479, 3724, 3731, 3739, 3753, 3765, 3776, 3786, 3834, 3848, 3855, 3862, 3869, 3876, 3883, 3890, 3897, 3904, 3911, 3918, 3925, 3932, 3939, 4071, 4080, 4172, 4177, 4195, 4337, 4345, 4405, 4575, 4878, 5178, 5612, 5615, 5626, 6199, 7024
\prg_return_true: *34*, 880, 880, 1053, 1069, 1088, 1154, 1174, 1286, 1294, 1298, 1442, 1733, 1735, 1739, 1743, 1747, 1897, 1994, 2208, 2216, 2226, 2231, 2238, 2248, 2384, 2388, 2392, 2396, 2400, 2404, 2408, 2412, 2416, 2420, 2424, 2428, 2432, 2436, 2443, 2447, 2451, 2472, 2479, 2490, 2501, 2512, 2523, 2530, 2537, 2545, 2611, 3173, 3177, 3181, 3185,

3189, 3193, 3197, 3202, 3267, 3271, 3413, 3479, 3724, 3731, 3739, 3753, 3768, 3779, 3786, 3834, 3848, 3855, 3862, 3869, 3876, 3883, 3890, 3897, 3904, 3911, 3918, 3925, 3932, 3939, 4071, 4080, 4172, 4177, 4195, 4337, 4345, 4405, 4575, 4878, 5178, 5179, 5612, 5615, 5626, 6197, 6988, 7025
\prg_set_conditional:Nnn 34, 906, 906, 1733, 6195
\prg_set_conditional:Npnn . . . 34, 882, 882, 1048, 1059, 1080, 1149, 1158, 1166, 1293, 1297, 1440, 1733, 1733, 1737, 1741, 1745, 1896, 1992, 3157, 3199, 3265, 3269, 3722
\prg_set_eq_conditional:NNn 34, 932, 932, 1733
\prg_set_eq_conditional_aux:NNNn . . . 933, 936, 992, 992
\prg_set_eq_conditional_aux:NNNw . . . 992, 993, 995, 1004
\prg_set_protected_conditional:Nnn . . . 34, 906, 919, 1733
\prg_set_protected_conditional:Npnn . . 34, 882, 894, 1733, 3844, 3851, 3858, 3865, 3872, 3879, 3886, 3893, 3900, 3907, 3914, 3921, 3928, 3935
\prg_stepwise_function:nnnN 41, 1803, 1803
\prg_stepwise_function_decr:nnnN . . . 1803, 1805, 1818, 1822
\prg_stepwise_function_incr:nnnN . . . 1803, 1806, 1809, 1813
\prg_stepwise_inline:nnnn 41, 1827, 1828
\prg_stepwise_inline_decr:Nnnn 1832, 1846, 1850
\prg_stepwise_inline_decr:nnnn . . 1827
\prg_stepwise_inline_incr:Nnnn 1833, 1838, 1842
\prg_stepwise_inline_incr:nnnn . . 1827
\prg_stepwise_variable:nnnNn 41, 1854, 1854
\prg_stepwise_variable_decr:nnnNn . . 1854, 1856, 1868, 1872
\prg_stepwise_variable_incr:nnnNn . . 1854, 1857, 1860, 1864
\prg_variable_get_scope:N 42, 2127, 2132
\prg_variable_get_scope_aux:w 2127, 2133, 2136
\prg_variable_get_type:N . 42, 2127, 2145
\prg_variable_get_type:w 2127
\prg_variable_get_type_aux:w 2149, 2153
\ProcessOptions 54
\prop_clear:c 97, 4774, 4775, 5345
\prop_clear:N 97, 4774, 4774, 4830
\prop_del:Nn 99, 4850, 4850, 4861
\prop_del:NV 99, 4850
\prop_del_aux:w 100, 4850, 4851, 4853, 4854
\prop_display:c 99, 4788
\prop_display:N 99, 4788, 4788, 4804, 5123, 5126
\prop_gclear:c 97, 4774, 4777
\prop_gclear:N 97, 4774, 4776, 4836
\prop_gdel:Nn 99, 4850, 4852, 4862
\prop_gdel:NV 99, 4850, 5104, 5116
\prop_get:cnN 98, 4811, 5408
\prop_get:cVN 98, 4811
\prop_get:NnN 98, 4811, 4811, 4815
\prop_get:NVN 98, 4811
\prop_get_aux:w . . . 100, 4811, 4812, 4814
\prop_get_del_aux:w 100, 4820, 4821, 4822
\prop_get_gdel:NnN 99, 4820, 4820
\prop_gget:cnN 98
\prop_gget:cVN 98
\prop_gget:NnN 98, 4816, 4816, 4819
\prop_gget:NVN 98, 4816
\prop_gget_aux:w . . 100, 4816, 4817, 4818
\prop_gput:ccx 98, 4828, 5736
\prop_gput:cnn 98, 4828
\prop_gput:Nnn 98, 4828, 4834, 4849
\prop_gput:Nno 98, 4828
\prop_gput:NnV 98, 4828
\prop_gput:Nnx 98, 4828
\prop_gput:NVn 98, 4828, 4986, 4999
\prop_gput_if_new:Nnn . . . 98, 4864, 4864
\prop_gset_eq:cc 98, 4778, 4785
\prop_gset_eq:cN 98, 4778, 4784
\prop_gset_eq:Nc 98, 4778, 4783
\prop_gset_eq:NN 98, 4778, 4782
\prop_if_empty:c 4869
\prop_if_empty:cTF 100, 4868
\prop_if_empty:N 4868
\prop_if_empty:NTF 100, 4868
\prop_if_empty_p:c 100, 4868
\prop_if_empty_p:N 100, 4868
\prop_if_eq:cc 4873
\prop_if_eq:ccTF 100, 4870
\prop_if_eq:cN 4871
\prop_if_eq:cNTF 100, 4870
\prop_if_eq:Nc 4872

\prop_if_eq:NcTF *100*, 4870
\prop_if_eq:NN 4870
\prop_if_eq:NNTF *100*, 4870
\prop_if_eq_p:cc *100*, 4870
\prop_if_eq_p:cN *100*, 4870
\prop_if_eq_p:Nc *100*, 4870
\prop_if_eq_p:NN *100*, 4870
\prop_if_in:ccTF *100*, 4874
\prop_if_in:cnTF . . *100*, 4874, 5397, 5400
\prop_if_in:Nn 4874
\prop_if_in:NnF 4882, 5005, 5012
\prop_if_in:NnT 4881
\prop_if_in:NnTF . . *100*, 4874, 4880, 5390
\prop_if_in:NoTF *100*, 4874
\prop_if_in:NVT 5050, 5090
\prop_if_in:NVTF *100*, 4874
\prop_if_in_aux:w . *101*, 4874, 4875, 4877
\prop_map_break: . . . *99*, 4889, 4905, 4905
\prop_map_function:cc *99*, 4883
\prop_map_function:cN *99*, 4883
\prop_map_function:Nc . . . *99*, 4883, 4900
\prop_map_function:NN *99*, 4883, 4883, 4894
\prop_map_function_aux:w *101*, 4883, 4884, 4887, 4892
\prop_map_inline:cn *99*, 4895
\prop_map_inline:Nn *99*, 4792, 4895, 4896, 4904
\prop_new:c *97*, 4772, 4773, 5341, 5770
\prop_new:N *97*, 4772, 4772, 4940, 4941, 5243, 5244, 5731, 5732
\prop_put:cnn *98*, 4828, 5471, 5474
\prop_put:Nnn *98*, 4828, 4828, 4848, 4944–4947, 5477
\prop_put:NnV *98*, 4828
\prop_put:NVn *98*, 4828
\prop_put:NVV *98*, 4828
\prop_put_aux:w *100*, 4828, 4831, 4837, 4840
\prop_put_if_new_aux:w *100*, 4864, 4865, 4866
\prop_set_eq:cc *98*, 4778, 4781
\prop_set_eq:cN *98*, 4778, 4780
\prop_set_eq:Nc *98*, 4778, 4779
\prop_set_eq:NN *98*, 4778, 4778
\prop_show:c *99*, 4786, 4787
\prop_show:N *99*, 4786, 4786
\prop_split_aux:Nnn *101*, 4805, 4805, 4812, 4817, 4821, 4829, 4835, 4851, 4853, 4865, 4875
\prop_tmp:w 4806, 4809, 4825, 4826, 4844, 4845, 4855–4858
\protect 734
\protected 22, 451
\ProvidesClass 657
\ProvidesExplClass *6*, 651, 656
\ProvidesExplFile *6*, 651, 660
\ProvidesExplPackage *6*, 651, 652, 754, 1461, 1728, 2162, 2265, 2783, 3141, 3300, 3518, 4201, 4365, 4537, 4766, 4911, 5189, 5584, 5726, 5838, 6040, 6473, 6949
\ProvidesFile 661
\ProvidesPackage 653, 700

Q

\Q 5892
\q 1027, 1765, 1959
\q_error *45*, 2171, 2171
\q_mark *45*, 2171, 2172, 4289
\q_nil *45*, 676, 680, 860, 863, 1013, 1019, 1027, 1029, 1036, 1038, 1313, 1317, 1662, 1667, 1670, 1971, 1973, 2052, 2056, 2134, 2136, 2147, 2149, 2153, 2168, 2170, 2177, 2189, 2205, 2231, 2236, 2246, 2439, 2441, 2469, 2471, 2476, 2478, 2486, 2489, 2497, 2500, 2508, 2511, 2519, 2522, 2527, 2529, 2534, 2536, 2541, 2544, 2557, 2563, 2569, 2575, 3117, 3163, 3171, 3175, 3179, 3183, 3187, 3191, 3195, 3738, 3750, 3752, 3770, 3781, 3785, 4158, 4160, 4162, 4166, 4167, 4171, 4176, 4180, 4194, 4486, 4599, 4606, 4885, 4888, 5747, 5754, 5893, 5900, 5908, 5917, 5922, 5930, 5937, 5947, 5953, 5954, 5964, 5970, 5975, 5988, 5993, 5997, 6000
\q_no_value *44*, 1971, 2168, 2169, 2207, 2214, 2224, 4073, 4082, 4100, 4112, 4141, 4404, 4407, 4574, 4577, 4809, 4825, 4844, 4857, 4885, 5929, 5937, 5943, 5948, 5964, 5971
\q_prop *100*, 4771, 4771, 4806, 4809, 4825, 4841, 4844, 4857, 4867, 4885, 4887, 5743, 5745, 5762
\q_recursion_stop *44*, 862, 865, 943, 993, 1615, 2010, 2020, 2030, 2040, 2048, 2173, 2174, 3951, 3955, 3968, 3977, 3982, 3993, 4648, 4654, 4671, 4683, 4690

\q_recursion_tail *44*, 2010, 2020, 2030, 2040, 2173, 2173, 2177, 2182, 2189, 2197, 2205, 3951, 3955, 3968, 3977, 3982, 3993, 4648, 4654, 4671, 4683, 4690
\q_stop . *44*, 861, 864, 2052, 2056, 2118, 2168, 2168, 3159, 3161, 3163, 3165, 4070, 4073, 4079, 4082, 4095, 4112, 4127, 4133, 4141, 4290, 4299, 4403, 4407, 4414, 4416, 4422, 4425, 4447, 4449, 4486, 4493, 4499, 4573, 4577, 4593, 4595, 4599, 4601, 4806, 4809, 4885, 5900, 5965, 6225, 6227, 6232
\quark_if_nil:N 2230
\quark_if_nil:n 2234, 2243
\quark_if_nil:nF 2255, 2259
\quark_if_nil:NT 4480
\quark_if_nil:nT . 2059, 2063, 2254, 2258
\quark_if_nil:NTF *43*, 2230, 3120, 4604, 5983
\quark_if_nil:nTF *43*, 2067, 2076, 2085, 2094, 2233, 2253, 2257
\quark_if_nil:oF 5945
\quark_if_nil:oTF *43*, 2233
\quark_if_nil:VTF *43*, 2233
\quark_if_nil_p:N *43*, 2230
\quark_if_nil_p:n . . *43*, 2233, 2252, 2256
\quark_if_nil_p:o *43*, 2233
\quark_if_nil_p:V *43*, 2233
\quark_if_no_value:N 2206
\quark_if_no_value:n 2212, 2221
\quark_if_no_value:NF 4824, 4856
\quark_if_no_value:nF 4097
\quark_if_no_value:NTF . . *43*, 1974, 2206
\quark_if_no_value:nTF *43*, 2206, 4071, 4080, 4129, 4878
\quark_if_no_value_p:N *43*, 2206
\quark_if_no_value_p:n *43*, 2206
\quark_if_recursion_tail_aux:w 2177, 2189, 2205, 2205
\quark_if_recursion_tail_stop:N *44*, 2175, 2181, 3988, 4697
\quark_if_recursion_tail_stop:n *44*, 2175, 2175, 2186, 3958, 4658
\quark_if_recursion_tail_stop:o *44*, 2175
\quark_if_recursion_tail_stop_do:Nn *44*, 2043, 2187, 2196
\quark_if_recursion_tail_stop_do:nn *44*, 2013, 2023, 2033, 2187, 2187, 2204, 3996
\quark_if_recursion_tail_stop_do:on *44*, 2187
\quark_new:N . . . *43*, 2167, 2167–2174, 4771

R

\R 1651, 2463, 2553
\radical 179
\raise 319
\ratio *133*, 6500, 6877, 6881
\read 126
\readline 401
\real *133*, 6499, 6877, 6882
\relax 11, 23, 26–31, 34–39, 161, 709
\relpenalty 222
\RequirePackage 721
\reverse_if:N *8*, 760, 765
\right 220
\righthyphenmin 276
\rightmarginkern 498
\rightskip 278
\romannumeral 353
\rpcode 465

S

\S 2463
\savinghyphcodes 440
\savingvdiscards 441
\scan_align_safe_stop: *40*, 1746, 1752, 1752
\scan_stop: *25*, 741, 791, 791, 996, 1021, 1150, 1154, 1155, 1163, 1165, 1167, 1170, 1174, 1648–1656, 1758, 2446, 2564, 2570, 2576, 3014, 3015, 3032, 3297, 3339, 3391, 3423, 3424, 3454, 3459, 3475, 3509, 3511, 3513, 4018–4021, 4351, 4583, 4987, 5000, 5282, 5666, 5692, 6621, 6681, 6690, 6707, 6716, 6764, 6779
\scantokens 399
\scriptfont 345
\scriptscriptfont 346
\scriptscriptstyle 191
\scriptspace 231
\scriptstyle 190
\scrollmode 156
\secondoftwo 4, 12
\seq_clear:c *85*, 4374, 4375
\seq_clear:N *85*, 4374, 4374, 4505
\seq_clear_new:c *85*, 4378, 4379
\seq_clear_new:N *85*, 4378, 4378
\seq_display:c *88*, 4431

\seq_display:N 88, 4431, 4431, 4445
\seq_elt:w 89, 4369, 4370, 4403, 4407, 4416, 4425, 4451, 4455, 4478, 4486, 4492, 4494, 4498, 4500
\seq_elt_end: 89, 4369, 4371, 4403, 4407, 4416, 4425, 4451, 4455, 4478, 4486, 4492, 4498
\seq_gclear:c 85, 4374, 4377
\seq_gclear:N 85, 4374, 4376
\seq_gclear_new:c 85, 4378, 4381
\seq_gclear_new:N 85, 4378, 4380
\seq_gconcat:ccc 86, 4390
\seq_gconcat:NNN . . . 86, 4390, 4390, 4393
\seq_get:cN 87, 4412, 4531
\seq_get:NN 87, 4412, 4412, 4419, 4530
\seq_get_aux:w 89, 4412, 4414, 4416
\seq_gpop:cN 90, 4526
\seq_gpop:NN . . . 90, 4526, 4527, 4529, 7041
\seq_gpush:cn 89, 4526
\seq_gpush:Nn 89, 4526, 4526, 4528
\seq_gpush:No 89, 4526
\seq_gpush:NV 89, 4526, 7038
\seq_gpush:Nv 89, 4526
\seq_gput_left:cn 86, 4457
\seq_gput_left:co 86, 4457
\seq_gput_left:cV 86, 4457
\seq_gput_left:Nn 86, 4457, 4457, 4475, 4526
\seq_gput_left:No 86, 4457
\seq_gput_left:NV 86, 4457
\seq_gput_left:Nx 86, 4457
\seq_gput_right:cn 86, 4457
\seq_gput_right:co 86, 4457
\seq_gput_right:cV 86, 4457
\seq_gput_right:Nc 87, 4477
\seq_gput_right:Nn 86, 4457, 4466, 4476, 4477
\seq_gput_right:No 86, 4457
\seq_gput_right:NV 86, 4457, 6970
\seq_gput_right:Nx 86, 4457
\seq_gremove_duplicates:N 88, 4504, 4517
\seq_gset_eq:cc 85, 4386, 4389
\seq_gset_eq:cN 85, 4386, 4387
\seq_gset_eq:Nc 85, 4386, 4388
\seq_gset_eq:NN 85, 4386, 4386, 4518
\seq_if_empty:c 4395
\seq_if_empty:cTF 88, 4394
\seq_if_empty:N 4394
\seq_if_empty:NTF 88, 4394
\seq_if_empty_err:N 89, 4396, 4396, 4413, 4421
\seq_if_empty_p:c 88, 4394
\seq_if_empty_p:N 88, 4394
\seq_if_in:cnTF 88, 4402
\seq_if_in:coTF 88, 4402
\seq_if_in:cVTF 88, 4402
\seq_if_in:cxTF 88, 4402
\seq_if_in:Nn 4402
\seq_if_in:NnF 4411, 4510
\seq_if_in:NnT 4410
\seq_if_in:NnTF 88, 4402, 4409
\seq_if_in:NVTF 88, 4402
\seq_map_break: 88, 4480, 4489, 4489, 7010
\seq_map_break:n 88, 4489, 4490
\seq_map_function:cN 87, 4491
\seq_map_function:NN 87, 4491, 4491, 4496, 4506, 7055
\seq_map_inline:cn 87, 4497
\seq_map_inline:Nn 87, 4435, 4497, 4497, 4502, 7004
\seq_map_variable:cNn 87, 4478
\seq_map_variable:NNn 87, 4478, 4484, 4488
\seq_map_variable_aux:Nnw 4478, 4478, 4482, 4485
\seq_new:c 4, 85, 4372, 4373
\seq_new:N 4, 85, 4372, 4372, 4503, 6955, 6966, 6974, 6977
\seq_pop:cN 90, 4520
\seq_pop:NN 90, 4520, 4524, 4525
\seq_pop_aux:nnNN 89, 4420, 4420, 4524, 4527
\seq_pop_aux:w 89, 4420, 4422, 4424
\seq_push:cn 89, 4520, 4523
\seq_push:Nn 89, 4520, 4520
\seq_push:No 89, 4520, 4522
\seq_push:NV 89, 4520, 4521
\seq_put_aux:Nnn 89, 4446, 4446, 4451, 4455
\seq_put_aux:w 89, 4446, 4447, 4449
\seq_put_left:cn 86, 4450, 4523
\seq_put_left:co 86, 4450
\seq_put_left:cV 86, 4450
\seq_put_left:Nn 86, 4450, 4450, 4453, 4464, 4520
\seq_put_left:No 86, 4450, 4522
\seq_put_left:NV 86, 4450, 4521
\seq_put_left:Nx 86, 4450
\seq_put_right:cn 86, 4450
\seq_put_right:co 86, 4450
\seq_put_right:cV 86, 4450
\seq_put_right:Nn 86, 4352, 4450, 4454, 4456, 4473, 4511, 5591, 7000, 7046, 7063

\seq_put_right:No 86, 4450
\seq_put_right:NV 86, 4450
\seq_put_right:Nx 86, 4450
\seq_remove_duplicates:N 88, 4504, 4514, 7047, 7053
\seq_remove_duplicates_aux:n 4504, 4506, 4509
\seq_remove_duplicates_aux:NN 4504, 4504, 4515, 4518
\seq_remove_element:Nn 7050
\seq_set_eq:cc 85, 4382, 4385
\seq_set_eq:cN 85, 4382, 4383
\seq_set_eq:Nc 85, 4382, 4384
\seq_set_eq:NN 85, 4382, 4382, 4515, 6997, 7017
\seq_show:c 88, 4429, 4430
\seq_show:N 88, 4429, 4429
\seq_tmp:w 4403, 4407
\seq_top:cN 90, 4530, 4531
\seq_top:NN 90, 4530, 4530
\setbox 327
\setcounter 133, 6887, 6933, 6939
\setlabel 5810, 5816, 5817, 5819, 5820, 5824, 5825, 5827, 5828
\setlanguage 85
\setlength 133, 6883, 6883
\setname 5801, 5816, 5817, 5819, 5820, 5824, 5827
\sfcode 382
\shipout 292
\show 135
\showbox 137
\showboxbreadth 151
\showboxdepth 152
\showgroups 412
\showifs 413
\showlists 138
\showMemUsage 742, 751, 1458, 1725, 2159, 2262, 2780, 3138, 4198, 4362, 4534, 4763, 4908, 5723, 5785, 6037, 6946
\showthe 136
\showtokens 400
\skewchar 349
\skip 373
\skip@ 3393
\skip_add:cn 63, 3353
\skip_add:Nn 63, 3353, 3353, 3359, 3371, 6552
\skip_eval:n 65, 3324, 3354, 3361, 3383, 3386, 3391, 3391
\skip_gadd:cn 63, 3353
\skip_gadd:Nn . . 63, 3353, 3366, 3373, 6555
.skip_gset:c 126
\skip_gset:cn 63, 3323
.skip_gset:N 126
\skip_gset:Nn . . 63, 3323, 3329, 3337, 6549
\skip_gsub:Nn 64, 3353, 3374, 6561
\skip_gzero:c 63, 3338
\skip_gzero:N 63, 3338, 3344, 3352
\skip_horizontal:c 64, 3381
\skip_horizontal:N . . 64, 3381, 3381–3383
\skip_horizontal:n 64, 3381, 3383
\skip_if_infinite_glue:n 3409
\skip_if_infinite_glue:nTF 64, 3409, 3416
\skip_if_infinite_glue_p:n 64, 3409
\skip_new:c 63, 3305
\skip_new:N 63, 3305, 3309, 3321, 3394–3398, 3400, 3402, 6488–6490
\skip_new_local:c 63, 3305
\skip_new_local:N . . 63, 3305, 3313, 3322
.skip_set:c 126
\skip_set:cn 63, 3323
.skip_set:N 126
\skip_set:Nn 63, 3323, 3323, 3334, 3336, 3401, 3403, 6546
\skip_show:c 64, 3389, 3390
\skip_show:N 64, 3389, 3389, 3390
\skip_split_finite_else_action:nnNN 65, 3415, 3415
\skip_sub:Nn . . . 64, 3353, 3360, 3379, 6558
\skip_use:c 64, 3387
\skip_use:N 64, 3387, 3387, 3388
\skip_vertical:c 64, 3381
\skip_vertical:N . . . 64, 3381, 3384–3386
\skip_vertical:n 64, 3381, 3386
\skip_zero:c 63, 3338
\skip_zero:N . . . 63, 3338, 3338, 3349, 3351
\skipdef 72
\spacefactor 291
\spaceskip 286
\span 99
\special 361, 549
\splitbotmark 170
\splitbotmarks 396
\splitdiscards 443
\splitfirstmark 169
\splitfirstmarks 395
\splitmaxdepth 339
\splittopskip 340
\startrecording 5791, 5798
\stepcounter 133, 6887, 6935, 6941

\str_if_eq:nn 1149
\str_if_eq_p:nn *11*, 1078, 1149
\str_if_eq_p_aux:w 1149, 1150, 1152, 1156, 1160
\str_if_eq_var:nf 1166
\str_if_eq_var_p:nf . *11*, 1166, 2223, 2245
\str_if_eq_var_start:nnN 1166, 1167, 1169, 1171, 1173
\str_if_eq_var_stop:w . . 1166, 1170, 1173
\strcmp 555, 557
\string 354

T

\T 1653, 2457
\t 2460
\tabskip 100
\tex_above:D 182
\tex_abovedisplayshortskip:D 195
\tex_abovedisplayskip:D 196
\tex_abovewithdelims:D 183
\tex_accent:D 233
\tex_adjdemerits:D 270
\tex_advance:D 77, 2790, 3354, 3361, 3454, 3459, 3511, 3513
\tex_afterassignment:D 87, 2661, 2773, 6607
\tex_aftergroup:D 88, 794
\tex_atop:D 184
\tex_atopwithdelims:D 185
\tex_badness:D 332
\tex_baselineskip:D 260
\tex_batchmode:D 153
\tex_begingroup:D 91, 595, 701, 792
\tex_belowdisplayshortskip:D 197
\tex_belowdisplayskip:D 198
\tex_binoppenalty:D 221
\tex_botmark:D 168
\tex_box:D 376, 5632, 5655
\tex_boxmaxdepth:D 338
\tex_brokenpenalty:D 295
\tex_catcode:D 380, 568–570, 572, 573, 577–579, 581, 582, 587, 588, 591, 592, 596, 718, 1022, 1531, 1532, 1650–1656, 2129, 2142, 2270, 2274, 2279, 4020, 4021
\tex_char:D 234
\tex_chardef:D . . 69, 1010, 1011, 1106, 1107, 1981, 1983, 3032, 4956, 4960
\tex_cleaders:D 252
\tex_closein:D 128
\tex_closeout:D 123, 5103, 5115
\tex_clubpenalty:D 263
\tex_copy:D 320, 5657
\tex_count:D 371
\tex_countdef:D 70, 1103, 2482, 2792, 3014
\tex_cr:D 95
\tex_crcr:D 96
\tex_csname:D 158, 708, 723, 784
\tex_currentgrouplevel:D 2802, 3316, 3438, 3504, 4217, 5601
\tex_day:D 366
\tex_deadcycles:D 300
\tex_def:D 65, 560, 566, 576, 585, 586, 590, 594, 599–601, 603, 605–611, 614, 618, 620–623, 627, 634, 636–638, 642, 644–646, 652, 656, 660, 680, 681, 688, 698, 700, 704, 709, 715, 720, 721, 730, 787, 789, 790, 798
\tex_defaulthyphenchar:D 350
\tex_defaultskewchar:D 351
\tex_delcode:D 381
\tex_delimiter:D 175
\tex_delimiterfactor:D 224
\tex_delimitershortfall:D 223
\tex_dimen:D 372
\tex_dimendef:D 71, 2504, 3428
\tex_discretionary:D 235
\tex_displayindent:D 200
\tex_displaylimits:D 210
\tex_displaystyle:D 188
\tex_displaywidowpenalty:D 199
\tex_displaywidth:D 201
\tex_divide:D 78
\tex_doublehyphendemerits:D 268
\tex_dp:D 379, 5650
\tex_dump:D 362
\tex_edef:D . . . 66, 563, 664, 666, 671, 799
\tex_else:D 119, 562, 613, 630, 675, 684, 699, 763, 1672
\tex_emergencystretch:D 283
\tex_end:D 157, 691, 5424, 5521
\tex_endcsname:D 159, 708, 723, 785
\tex_endgroup:D . . . 92, 604, 619, 705, 793
\tex_endinput:D 131
\tex_endlinechar:D 173, 571, 580, 4029, 4045, 4055, 4063
\tex_eqno:D 193
\tex_errhelp:D 139, 5293
\tex_errmessage:D . . 133, 744, 1116, 5295
\tex_errorcontextlines:D 140, 5194
\tex_errorstopmode:D 154

\tex_escapechar:D 172
\tex_everycr:D 101
\tex_everydisplay:D 202, 693
\tex_everyhbox:D 341
\tex_everyjob:D 370, 6958, 6969
\tex_everymath:D 226, 692
\tex_everypar:D 289
\tex_everyvbox:D 342
\tex_exhyphenpenalty:D 265
\tex_expandafter:D . 89, 629, 631, 669, 672, 676, 707, 711, 722, 778, 1467, 7040
\tex_fam:D 81
\tex_fi:D 120, 558, 559, 574, 583, 615, 632, 677, 686, 726, 727, 764, 1674
\tex_finalhyphendemerits:D 269
\tex_firstmark:D 167
\tex_floatingpenalty:D 314
\tex_font:D 80
\tex_fontdimen:D 347
\tex_fontname:D 171
\tex_futurelet:D 76, 728, 2622, 2624
\tex_gdef:D 67, 716, 717, 719, 818
\tex_global:D 82, 713, 714, 795, 1983, 3006
\tex_globaldefs:D 86
\tex_halign:D 93
\tex_hangafter:D 271
\tex_hangindent:D 272
\tex_hbadness:D 333
\tex_hbox:D 328, 5692, 5693, 5698, 5704, 5713, 5714
\tex_hfil:D 236
\tex_hfill:D 238
\tex_hfilneg:D 237
\tex_hfuzz:D 335
\tex_hoffset:D 310
\tex_holdinginserts:D 313
\tex_hrule:D 249
\tex_hsize:D 274
\tex_hskip:D 239, 3381
\tex_hss:D 240, 5715, 5716
\tex_ht:D 378, 5649
\tex_hyphen:D 63, 695
\tex_hyphenation:D 364
\tex_hyphenchar:D 348
\tex_hyphenpenalty:D 266
\tex_if:D 102, 673, 766, 769
\tex_ifcase:D 103, 1322, 3147, 3153
\tex_ifcat:D 104, 770
\tex_ifdim:D 107, 3476
\tex_ifeof:D 108, 5175, 5178
\tex_iffalse:D 113, 761
\tex_ifhbox:D 109, 5608, 5612
\tex_ifhmode:D 115, 773
\tex_ifinner:D 118, 775
\tex_ifmmode:D 116, 772
\tex_ifnum:D 105, 1081, 1087, 3146, 3151, 3832
\tex_ifodd:D 106, 561, 565, 682, 767, 768, 3152
\tex_iftrue:D 114, 760
\tex_ifvbox:D 110, 5609, 5615
\tex_ifvmode:D 117, 774
\tex_ifvoid:D 111, 5610, 5626
\tex_ifx:D 112, 602, 628, 697, 722, 771, 1669
\tex_ignorespaces:D 160
\tex_immediate:D 122, 639, 647, 706, 1109, 1112, 4987, 5103, 5142
\tex_indent:D 256
\tex_input:D 130, 689, 725, 7040
\tex_inputlineno:D . 132, 1133, 1639, 5248
\tex_insert:D 312
\tex_insertpenalties:D 315
\tex_interlinepenalty:D 294
\tex_italiccor:D 62, 694
\tex_jobname:D 369, 3711, 6959
\tex_kern:D 247
\tex_language:D 164
\tex_lastbox:D 321, 5636
\tex_lastkern:D 254
\tex_lastpenalty:D 360
\tex_lastskip:D 255
\tex_lccode:D 383, 1021, 1648, 1649, 2128, 2141, 2332, 2336, 2339, 4018, 4019
\tex_leaders:D 251
\tex_left:D 219
\tex_lefthyphenmin:D 275
\tex_leftskip:D 277
\tex_leqno:D 194
\tex_let:D 40, 44, 50, 58, 64, 689–696, 712–714, 759
\tex_limits:D 211
\tex_linepenalty:D 267
\tex_lineskip:D 261
\tex_lineskiplimit:D 262
\tex_long:D 83, 796
\tex_looseness:D 279
\tex_lower:D 316, 5644
\tex_lowercase:D . . 355, 1023, 1657, 3943
\tex_mag:D 163

\tex_mark:D 165
\tex_mathaccent:D 176
\tex_mathbin:D 206
\tex_mathchar:D 177
\tex_mathchardef:D . . 74, 3006, 3297, 5590
\tex_mathchoice:D 174
\tex_mathclose:D 207
\tex_mathcode:D 385, 2316, 2320, 2324, 2328
\tex_mathinner:D 208
\tex_mathop:D 209
\tex_mathopen:D 213
\tex_mathord:D 214
\tex_mathpunct:D 215
\tex_mathrel:D 216
\tex_mathsurround:D 227
\tex_maxdeadcycles:D 297
\tex_maxdepth:D 298
\tex_meaning:D 357, 782, 786, 1662
\tex_medmuskip:D 228
\tex_message:D 134, 747
\tex_mkern:D 181
\tex_month:D 367
\tex_moveleft:D 317, 5641
\tex_moveright:D 318, 5642
\tex_mskip:D 178
\tex_multiply:D 79
\tex_muskip:D 375
\tex_muskipdef:D 73, 3494
\tex_newlinechar:D 129, 5165
\tex_noalign:D 97
\tex_noboundary:D 232
\tex_noexpand:D 90, 574, 779, 1468
\tex_noindent:D 258
\tex_nolimits:D 212
\tex_nonscript:D 192
\tex_nonstopmode:D 155
\tex_nulldelimiterspace:D 225
\tex_nullfont:D 343
\tex_number:D 352, 1167, 1943, 1954, 1956, 2789, 3148, 6777, 6778
\tex_omit:D 98
\tex_openin:D 124, 5000
\tex_openout:D 125, 4987
\tex_or:D 121, 762
\tex_outer:D 84
\tex_output:D 299
\tex_outputpenalty:D 309
\tex_over:D 186
\tex_overfullrule:D 337
\tex_overline:D 217
\tex_overwithdelims:D 187
\tex_pagedepth:D 301
\tex_pagefilllstretch:D 305
\tex_pagefillstretch:D 304
\tex_pagefilstretch:D 303
\tex_pagegoal:D 307
\tex_pageshrink:D 306
\tex_pagestretch:D 302
\tex_pagetotal:D 308
\tex_par:D 257, 696
\tex_parfillskip:D 288
\tex_parindent:D 281
\tex_parshape:D 273
\tex_parskip:D 280
\tex_patterns:D 363
\tex_pausing:D 150
\tex_penalty:D 358
\tex_postdisplaypenalty:D 205
\tex_predisplaypenalty:D 204
\tex_predisplaysize:D 203
\tex_pretolerance:D 284
\tex_prevdepth:D 331
\tex_prevgraf:D 290
\tex_radical:D 179
\tex_raise:D 319, 5643
\tex_read:D 126, 5182
\tex_relax:D 161, 561, 565, 568–573, 577–582, 587, 588, 591, 592, 596, 624, 625, 627, 634, 635, 642, 643, 682, 723, 725, 791, 1049, 1066, 1322, 1356, 1519, 1531, 1532, 2128, 2129, 2141, 2142, 3150, 3543
\tex_relpenalty:D 222
\tex_right:D 220
\tex_righthyphenmin:D 276
\tex_rightskip:D 278
\tex_romannumeral:D 353, 880, 881, 1027, 1493, 1504, 1512, 1601, 1604, 1607, 1693, 1700, 1705, 1765, 2788, 3162
\tex_scriptfont:D 345
\tex_scriptscriptfont:D 346
\tex_scriptscriptstyle:D 191
\tex_scriptspace:D 231
\tex_scriptstyle:D 190
\tex_scrollmode:D 156
\tex_setbox:D . . 327, 5632, 5637, 5667, 5672, 5678, 5686, 5693, 5698, 5704
\tex_setlanguage:D 85
\tex_sfcode:D 382, 2352, 2356, 2360
\tex_shipout:D 292, 749

\tex_show:D 135, 788
\tex_showbox:D 137, 5659
\tex_showboxbreadth:D 151
\tex_showboxdepth:D 152
\tex_showlists:D 138
\tex_showthe:D . 136, 2279, 2328, 2339, 2349, 2360, 2886, 3389, 3465, 4286
\tex_skewchar:D 349
\tex_skip:D 373
\tex_skipdef:D 72, 2493, 3306, 3393
\tex_space:D 61
\tex_spacefactor:D 291
\tex_spaceskip:D 286
\tex_span:D 99
\tex_special:D 361
\tex_splitbotmark:D 170
\tex_splitfirstmark:D 169
\tex_splitmaxdepth:D 339
\tex_splittopskip:D 340
\tex_string:D 354, 783
\tex_tabskip:D 100
\tex_textfont:D 344
\tex_textstyle:D 189
\tex_the:D 162, 568–573, 1133, 1356, 1525, 1533, 1639, 2884, 3387, 3463, 3515, 4224
\tex_thickmuskip:D 230
\tex_thinmuskip:D 229
\tex_time:D 365
\tex_toks:D 374
\tex_toksdef:D 75, 2515, 4207, 4351
\tex_tolerance:D 285
\tex_topmark:D 166
\tex_topskip:D 296
\tex_tracingcommands:D 141
\tex_tracinglostchars:D 142
\tex_tracingmacros:D 143
\tex_tracingonline:D 144
\tex_tracingoutput:D 145
\tex_tracingpages:D 146
\tex_tracingparagraphs:D 147
\tex_tracingrestores:D 148
\tex_tracingstats:D 149
\tex_uccode:D 384, 2342, 2346, 2349
\tex_uchyph:D 282
\tex_underline:D 218, 690
\tex_unhbox:D 323, 5719
\tex_unhcopy:D 324, 5717
\tex_unkern:D 248
\tex_unpenalty:D 359
\tex_unskip:D 246
\tex_unvbox:D 325, 5690
\tex_unvcopy:D 326, 5688
\tex_uppercase:D 356, 3944
\tex_vadjust:D 259
\tex_valign:D 94
\tex_vbadness:D 334
\tex_vbox:D 329, 5666, 5667, 5672, 5678, 5683, 5684
\tex_vcenter:D 180
\tex_vfil:D 241
\tex_vfill:D 243
\tex_vfilneg:D 242
\tex_vfuzz:D 336
\tex_voffset:D 311
\tex_vrule:D 250
\tex_vsize:D 293
\tex_vskip:D 244, 3384
\tex_vsplit:D 322, 5686
\tex_vss:D 245
\tex_vtop:D 330
\tex_wd:D 377, 5651
\tex_widowpenalty:D 264
\tex_write:D . 127, 639, 647, 706, 781, 5167
\tex_xdef:D 68, 707, 819
\tex_xleaders:D 253
\tex_xspaceskip:D 287
\tex_year:D 368
\textasteriskcentered 2975, 2981
\textbardbl 2980
\textdagger 2976, 2982
\textdaggerdbl 2977, 2983
\textfont 344
\textparagraph 2979
\textsection 2978
\textstyle 189
\TeXXETstate 444
\the 26–31, 162
\thepage 5804
\thickmuskip 230
\thinmuskip 229
\time 365
\tl_clear:c *71*, 3599, 4375, 6129
\tl_clear:N *71*, 3599, 3599, 3600, 3607, 3613, 4126, 4374, 4398, 4544, 4564, 5925, 5936, 6992
\tl_clear_new:c *71*, 3603, 4379
\tl_clear_new:N *71*, 3603, 3604, 3613, 3614, 4378, 4548
\tl_compare:nn 3789

\tl_compare:no 3801
\tl_compare:nV 3798
\tl_compare:nx 3792
\tl_compare:on 3807
\tl_compare:oo 3813
\tl_compare:ox 3825
\tl_compare:Vn 3804
\tl_compare:VV 3810
\tl_compare:Vx 3822
\tl_compare:xn 3795
\tl_compare:xo 3819
\tl_compare:xV 3816
\tl_compare:xx 3788, 3790, 3793, 3796, 3799, 3802, 3805, 3808, 3811, 3814, 3817, 3820, 3823, 3826
\tl_const:Nn *70*, 2167, 2660, 3538, 3538, 3712, 3713, 4024, 4920, 5195–5199, 5203, 5206, 5209, 5212, 5218, 5222, 5232, 5236–5238, 5845, 6045–6048
\tl_elt_count:N *76*, 4003, 4009
\tl_elt_count:n *76*, 4003, 4003, 4008
\tl_elt_count:o *76*, 4003
\tl_elt_count:V *76*, 4003
\tl_elt_count_aux:n 4005, 4011, 4014
\tl_gclear:c *71*, 3599, 4377
\tl_gclear:N *71*, 3599, 3601, 3602, 3619, 3624, 4376, 4546
\tl_gclear_new:c *71*, 3615, 4381
\tl_gclear_new:N *71*, 3615, 3616, 3624, 3625, 4380, 4550
\tl_gput_left:cn *71*, 3665
\tl_gput_left:co *71*
\tl_gput_left:cV *71*, 3665
\tl_gput_left:cv 3665
\tl_gput_left:cx 3665
\tl_gput_left:Nn *71*, 3665, 3680, 3699, 4634
\tl_gput_left:No *71*, 3665, 3689
\tl_gput_left:NV . . . *71*, 3665, 3683, 3700
\tl_gput_left:Nv 3665, 3686, 3701
\tl_gput_left:Nx . . . *71*, 3665, 3692, 3702
\tl_gput_right:cn *72*, 3626
\tl_gput_right:co *72*, 3626
\tl_gput_right:cV *72*, 3626
\tl_gput_right:cv 3626
\tl_gput_right:cx 3626
\tl_gput_right:Nn *72*, 3626, 3641, 3660, 4642
\tl_gput_right:No . . *72*, 3626, 3650, 3663
\tl_gput_right:NV . . *72*, 3626, 3644, 3661
\tl_gput_right:Nv 3626, 3647, 3662
\tl_gput_right:Nx . . *72*, 3626, 3653, 3664
\tl_gremove_all_in:cn *78*, 4150
\tl_gremove_all_in:Nn *78*, 4150, 4153, 4157
\tl_gremove_in:cn *78*, 4146
\tl_gremove_in:Nn . . *78*, 4146, 4147, 4149
\tl_greplace_all_in:cnn *78*, 4119
\tl_greplace_all_in:Nnn *78*, 4122, 4145, 4154
\tl_greplace_in:cnn *78*, 4090
\tl_greplace_in:Nnn *78*, 4090, 4114, 4118, 4147
.tl_gset:c *126*
\tl_gset:cn *70*, 3554
\tl_gset:cV 3554
\tl_gset:cv 3554
\tl_gset:cx *70*, 3554
.tl_gset:N *126*
\tl_gset:Nc *70*, 3703, 3703
\tl_gset:Nf 3554
\tl_gset:Nn *70*, 3540, 3554, 3560, 3575–3581, 4016, 4527, 4634, 4642, 4757, 7039
\tl_gset:No *70*, 3554
\tl_gset:NV *70*, 3554
\tl_gset:Nv *70*, 3554
\tl_gset:Nx *70*, 3554, 3563, 3582, 4057, 4391, 4710, 4818, 6959
\tl_gset_eq:cc *72*, 3583, 4559
\tl_gset_eq:cN *72*, 3583, 4557
\tl_gset_eq:Nc *72*, 3583, 4558
\tl_gset_eq:NN . . *72*, 3583, 3588, 3595, 3598, 3601, 4115, 4123, 4556, 6963
\tl_gset_rescan:Nnn *73*, 4015, 4016
\tl_gset_rescan:Nnx *73*, 4042, 4052
.tl_gset_x:c *126*
.tl_gset_x:N *126*
\tl_head:n *79*, 4158, 4158, 4159, 4168
\tl_head:V *79*, 4158
\tl_head:w *79*, 4158, 4158, 4164, 4165, 4171, 4176, 4180, 4194
\tl_head_i:n *79*, 4158, 4159
\tl_head_i:w *79*, 4158, 4165
\tl_head_iii:f *79*, 4158
\tl_head_iii:n *79*, 4158, 4162, 4163
\tl_head_iii:w *79*, 4158, 4162, 4167
\tl_if_blank:n 3749
\tl_if_blank:nF 3758, 3762, 4653
\tl_if_blank:nT 3757, 3761
\tl_if_blank:nTF *75*, 3749, 3756, 3760, 3767, 3778
\tl_if_blank:oTF *75*, 3749, 5942

\tl_if_blank:VTF 75, 3749
\tl_if_blank_p:n . . . 75, 3749, 3755, 3759
\tl_if_blank_p:o 75, 3749
\tl_if_blank_p:V 75, 3749
\tl_if_blank_p_aux:w . . . 3749, 3750, 3752
\tl_if_empty:c 4395, 4561
\tl_if_empty:cTF 73, 3722
\tl_if_empty:N 3722, 4394, 4560
\tl_if_empty:n 3737
\tl_if_empty:NF 3729, 4705
\tl_if_empty:nF 2701, 3744, 3748, 4626, 4677, 4688, 4842
\tl_if_empty:NT 3728
\tl_if_empty:nT 3743, 3747, 4867
\tl_if_empty:NTF 73, 3722, 3727, 3775, 5980, 6002, 7023
\tl_if_empty:nTF 74, 2472, 2479, 2490, 2501, 2512, 2523, 2530, 2537, 2545, 3737, 3742, 3746, 3764, 3786, 5307
\tl_if_empty:oTF 74, 3737
\tl_if_empty:VTF 74, 3737, 4337
\tl_if_empty_p:c 73, 3722
\tl_if_empty_p:N 73, 3722, 3726
\tl_if_empty_p:n 74, 2442, 3737, 3741, 3745
\tl_if_empty_p:o 74, 3737
\tl_if_empty_p:V 74, 3737
\tl_if_eq:cc 4571
\tl_if_eq:ccTF 73, 3730, 6203
\tl_if_eq:cN 4569
\tl_if_eq:cNTF 73, 3730
\tl_if_eq:Nc 4570
\tl_if_eq:NcTF 73, 3730
\tl_if_eq:NN 3730, 4568
\tl_if_eq:nn 3844
\tl_if_eq:NNF 3736
\tl_if_eq:nnF 4739
\tl_if_eq:NNT 3735
\tl_if_eq:NNTF 73, 2044, 3730, 3734, 5409, 5956, 6007
\tl_if_eq:nnTF 74, 3788
\tl_if_eq:no 3858
\tl_if_eq:noTF 74, 3788
\tl_if_eq:nV 3851
\tl_if_eq:nVTF 74, 3788
\tl_if_eq:nx 3865
\tl_if_eq:nxTF 74, 3788
\tl_if_eq:on 3879
\tl_if_eq:onTF 74, 3788
\tl_if_eq:oo 3893
\tl_if_eq:ooTF 74, 3788
\tl_if_eq:ox 3907
\tl_if_eq:oxTF 74, 3788
\tl_if_eq:Vn 3872
\tl_if_eq:VnTF 74, 3788
\tl_if_eq:VV 3886
\tl_if_eq:VVTF 74, 3788
\tl_if_eq:Vx 3900
\tl_if_eq:VxTF 74, 3788
\tl_if_eq:xn 3914
\tl_if_eq:xnTF 74, 3788
\tl_if_eq:xo 3928
\tl_if_eq:xoTF 74, 3788
\tl_if_eq:xV 3921
\tl_if_eq:xVTF 74, 3788
\tl_if_eq:xx 3935
\tl_if_eq:xxTF 74, 2034, 3788, 4344
\tl_if_eq_p:cc 73, 3730
\tl_if_eq_p:cN 73, 3730
\tl_if_eq_p:Nc 73, 3730
\tl_if_eq_p:NN 73, 3730, 3733
\tl_if_eq_p:nn 74, 3788
\tl_if_eq_p:no 74, 3788
\tl_if_eq_p:nV 74, 3788
\tl_if_eq_p:nx 74, 3788
\tl_if_eq_p:on 74, 3788
\tl_if_eq_p:oo 74, 3788
\tl_if_eq_p:ox 74, 3788
\tl_if_eq_p:Vn 74, 3788
\tl_if_eq_p:VV 74, 3788
\tl_if_eq_p:Vx 74, 3788
\tl_if_eq_p:xn 74, 3788
\tl_if_eq_p:xo 74, 3788
\tl_if_eq_p:xV 74, 3788
\tl_if_eq_p:xx 74, 3788
\tl_if_head_eq_catcode:nN 4192
\tl_if_head_eq_catcode:nNTF . . . 79, 4170
\tl_if_head_eq_catcode_p:nN . . . 79, 4170
\tl_if_head_eq_charcode:fNTF 79, 3055, 3062, 4170
\tl_if_head_eq_charcode:nN 4174
\tl_if_head_eq_charcode:nNF 4191
\tl_if_head_eq_charcode:nNT 4190
\tl_if_head_eq_charcode:nNTF 79, 4170, 4189
\tl_if_head_eq_charcode_p:fN 79, 4170, 4179, 4187
\tl_if_head_eq_charcode_p:nN 79, 4170, 4188
\tl_if_head_eq_meaning:nN 4170

\tl_if_head_eq_meaning:nNTF 79, 4170, 5971
\tl_if_head_eq_meaning_p:nN . . . 79, 4170
\tl_if_in:cnTF 77, 4069
\tl_if_in:Nn 4069
\tl_if_in:nn 4078
\tl_if_in:NnF 4077
\tl_if_in:nnF 4086, 4089
\tl_if_in:NnT 4076
\tl_if_in:nnT 4085, 4088
\tl_if_in:NnTF 77, 4069, 4075
\tl_if_in:nnTF 77, 4078, 4084, 4087, 6218, 6228
\tl_if_in:onTF 77, 4078
\tl_if_in:VnTF 77, 4078
\tl_if_single:N 3774
\tl_if_single:n 3763
\tl_if_single:NTF 75
\tl_if_single:nTF 75, 3763
\tl_if_single_p:N 75
\tl_if_single_p:n 75, 3763
\tl_map_break: . 76, 3991, 3991, 5008, 5015
\tl_map_function:cN 75, 3950
\tl_map_function:NN 75, 3950, 3953, 3961, 4011, 4981, 4994
\tl_map_function:nN 75, 3950, 3950, 4005, 6724
\tl_map_function_aux:NN 3950
\tl_map_function_aux:Nn 3951, 3954, 3957, 3959, 3966, 3975
\tl_map_inline:cn 76, 3962
\tl_map_inline:Nn . . 76, 3962, 3971, 3980
\tl_map_inline:nn . . 76, 2463, 3962, 3962
\tl_map_inline_aux:n 3962
\tl_map_variable:cNn 76, 3981
\tl_map_variable:NNn 76, 3981, 3984, 3985
\tl_map_variable:nNn 76, 3981, 3981, 3984
\tl_map_variable_aux:NnN 3986
\tl_map_variable_aux:Nnn 3982, 3986, 3989
\tl_new:c 69, 3523, 3537, 4373, 6128, 6133
\tl_new:cn 69, 3523, 3537, 5740
\tl_new:N . 69, 2626, 2627, 2629, 2659, 3523, 3536, 3539, 3609, 3621, 3710, 3714–3721, 4090, 4372, 4542, 5239–5241, 5246, 5843, 5844, 5846–5848, 6050–6055, 6057, 6478, 6954, 6980
\tl_new:Nn 69, 3523, 3523, 3530, 3536
\tl_new:Nx 69, 3523, 3531
\tl_num_elt_count_aux:n 4014
\tl_put_left:cn 71, 3665
\tl_put_left:co 71
\tl_put_left:cV 71, 3665
\tl_put_left:cv 3665
\tl_put_left:cx 3665
\tl_put_left:Nn 71, 3665, 3665, 3695, 4630
\tl_put_left:No 71, 3665, 3677
\tl_put_left:NV 71, 3665, 3668, 3696
\tl_put_left:Nv 3665, 3671, 3697
\tl_put_left:Nx 71, 3665, 3674, 3698
\tl_put_right:cn 71, 3626
\tl_put_right:co 71
\tl_put_right:cV 71, 3626
\tl_put_right:cv 3626
\tl_put_right:cx 3626
\tl_put_right:Nn 71, 3626, 3626, 3656, 4638, 5981, 6004, 6009
\tl_put_right:No 71, 3626, 3638, 4101, 4130, 4132, 5904, 5958, 5976, 5987, 5992
\tl_put_right:NV . . . 71, 3626, 3629, 3657
\tl_put_right:Nv 3626, 3632, 3658
\tl_put_right:Nx . . . 71, 3626, 3635, 3659
\tl_remove_all_in:cn 78, 4150
\tl_remove_all_in:Nn 78, 4150, 4150, 4156
\tl_remove_in:cn 78, 4146
\tl_remove_in:Nn . . . 78, 4146, 4146, 4148
\tl_replace_all_in:cnn 78
\tl_replace_all_in:Nnn 78, 4119, 4119, 4144, 4151, 5885, 5888
\tl_replace_in:cnn 78, 4090
\tl_replace_in:Nnn 78, 4090, 4091, 4117, 4146
\tl_rescan:nn 73, 4060, 4060
\tl_rescan_aux:w . 4031, 4036, 4037, 4065
\tl_reverse:N 76, 4000, 4000
\tl_reverse:n 76, 3992, 3992, 3999
\tl_reverse:o 76, 3992, 4001
\tl_reverse:V 76, 3992
\tl_reverse_aux:nN 3992, 3993, 3995, 3997
.tl_set:c 126
\tl_set:cn 70, 3554, 6134
\tl_set:co 70, 3554
\tl_set:cV 70, 3554
\tl_set:cv 3554
\tl_set:cx 70, 3554, 6075
.tl_set:N 126
\tl_set:Nc 70, 3703, 3708, 3709
\tl_set:Nf 70, 3554, 4001, 5901, 5909
\tl_set:Nn 70, 2632–2634, 2646, 2647, 2649, 3554, 3554, 3566–3573,

3987, 4015, 4040, 4050, 4417, 4449, 4479, 4485, 4524, 4527, 4595, 4630, 4638, 4751, 4757, 4814, 4823, 5388, 5389, 5801, 5926, 5955, 6001, 6108, 6138, 6235, 6248, 6293, 6987, 7009
\tl_set:No *70*, 3554, 3709, 4099, 4701, 4702, 5916
\tl_set:NV *70*, 3554
\tl_set:Nv *70*, 3554, 6097, 6296
\tl_set:Nx *70*, 3554, 3557, 3574, 3711, 3829, 3845, 3846, 3852, 3853, 3859, 3860, 3866, 3867, 3873, 3874, 3880, 3881, 3887, 3888, 3894, 3895, 3901, 3902, 3908, 3909, 3915, 3916, 3922, 3923, 3929, 3930, 3936, 3937, 4047, 4709, 5308, 5310, 5975, 6217, 6229, 6234, 6262, 6263, 6501
\tl_set_eq:cc *72*, 3583, 4555, 6303
\tl_set_eq:cN *72*, 3583, 4553
\tl_set_eq:Nc *72*, 3583, 4554
\tl_set_eq:NN . . . *72*, 2648, 3583, 3584, 3594, 3597, 3599, 4092, 4120, 4552
\tl_set_rescan:Nnn *73*, 4015, 4015
\tl_set_rescan:Nnx *73*, 4042, 4042
\tl_set_rescan_aux:NNnn 4015, 4016, 4017, 4026
.tl_set_x:c *126*
.tl_set_x:N *126*
\tl_show:c *70*, 3551, 4430, 4608
\tl_show:N *70*, 3551, 3551, 3552, 4429, 4607
\tl_show:n *70*, 3551, 3553
\tl_tail:f *79*, 4158
\tl_tail:n *79*, 4158, 4160, 4161, 4169
\tl_tail:V *79*, 4158
\tl_tail:w *79*, 4158, 4160, 4166
\tl_tmp:w 3828, 3839–3842, 4070, 4073, 4079, 4082
\tl_to_lowercase:n *75*, 2130, 2143, 2465, 2555, 3943, 3943, 4022, 5230, 5285, 5883
\tl_to_str:c *72*, 3946
\tl_to_str:N *72*, 3946, 3946, 3949
\tl_to_str:n *72*, 2225, 2247, 3738, 3750, 3945, 3945, 6230, 6234, 6262, 6289
\tl_to_str_aux:w 3946, 3946, 3948
\tl_to_uppercase:n *75*, 3943, 3944
\tl_use:c *70*, 3542
\tl_use:N *70*, 3542, 3542, 3550, 6282
\token_get_arg_spec:N . . . *52*, 2548, 2566
\token_get_prefix_arg_replacement_aux:w 2548, 2557, 2562, 2568, 2574
\token_get_prefix_spec:N . *52*, 2548, 2560
\token_get_replacement_spec:N *52*, 2548, 2572
\token_if_active_char:N 2422
\token_if_active_char:NTF *50*, 2422
\token_if_active_char_p:N *50*, 2422, 2587
\token_if_alignment_tab:N 2394
\token_if_alignment_tab:NTF . . . *49*, 2394
\token_if_alignment_tab_p:N . . . *49*, 2394
\token_if_chardef:N 2467
\token_if_chardef:NTF *51*, 2456
\token_if_chardef_aux:w 2468, 2471
\token_if_chardef_p:N . . . *51*, 2456, 2599
\token_if_chardef_p_aux:w 2456
\token_if_cs:N 2445
\token_if_cs:NTF *50*, 2445
\token_if_cs_p:N *50*, 2445, 2580
\token_if_dim_register:N 2503
\token_if_dim_register:NTF *51*, 2456
\token_if_dim_register_aux:w 2507, 2511
\token_if_dim_register_p:N *51*, 2456, 2607
\token_if_dim_register_p_aux:w . . 2456
\token_if_eq_catcode:NN 2430
\token_if_eq_catcode:NNTF *50*, 2430
\token_if_eq_catcode_p:NN *50*, 2430, 2446, 2679, 2680
\token_if_eq_charcode:NN 2434
\token_if_eq_charcode:NNTF *50*, 2434
\token_if_eq_charcode_p:NN *50*, 2434
\token_if_eq_meaning:NN 2426
\token_if_eq_meaning:NNT 2137
\token_if_eq_meaning:NNTF *50*, 2150, 2426, 2772
\token_if_eq_meaning_p:NN *50*, 2426, 2681
\token_if_expandable:N 2448
\token_if_expandable:NTF *50*, 2448
\token_if_expandable_p:N *50*, 2448
\token_if_group_begin:N 2382
\token_if_group_begin:NTF *48*, 2382
\token_if_group_begin_p:N *48*, 2382
\token_if_group_end:N 2386
\token_if_group_end:NTF *48*, 2386
\token_if_group_end_p:N *48*, 2386
\token_if_int_register:N 2481
\token_if_int_register:NTF *51*, 2456
\token_if_int_register_aux:w 2485, 2489
\token_if_int_register_p:N *51*, 2456, 2603
\token_if_int_register_p_aux:w . . 2456

\token_if_letter:N 2414
\token_if_letter:NTF *49*, 2414
\token_if_letter_p:N *49*, 2414
\token_if_long_macro:N 2532
\token_if_long_macro:NTF *51*, 2456
\token_if_long_macro_aux:w . . 2533, 2536
\token_if_long_macro_p:N *51*, 2456
\token_if_long_macro_p_aux:w 2456
\token_if_macro:N 2438
\token_if_macro:NTF *50*, 2438, 2561, 2567, 2573
\token_if_macro_p:N . *50*, 2438, 2581, 2588
\token_if_macro_p_aux:w 2438, 2439, 2441
\token_if_math_shift:N 2390
\token_if_math_shift:NTF *49*, 2390
\token_if_math_shift_p:N *49*, 2390
\token_if_math_subscript:N 2406
\token_if_math_subscript:NTF . . *49*, 2406
\token_if_math_subscript_p:N . . *49*, 2406
\token_if_math_superscript:N 2402
\token_if_math_superscript:NTF *49*, 2402
\token_if_math_superscript_p:N *49*, 2402
\token_if_mathchardef:N 2474
\token_if_mathchardef:NTF *51*, 2456
\token_if_mathchardef_aux:w . 2475, 2478
\token_if_mathchardef_p:N *51*, 2456, 2601
\token_if_mathchardef_p_aux:w . . . 2456
\token_if_other_char:N 2418
\token_if_other_char:NTF *49*, 2418
\token_if_other_char_p:N *49*, 2418
\token_if_parameter:N 2398
\token_if_parameter:NTF *49*, 2398
\token_if_parameter_p:N *49*, 2398
\token_if_primitive:N 2579
\token_if_primitive:NTF *52*, 2579
\token_if_primitive_p:N *52*, 2579
\token_if_primitive_p_aux:N 2579, 2584, 2591, 2598
\token_if_protected_long_macro:N . 2539
\token_if_protected_long_macro:NTF *51*, 2456
\token_if_protected_long_macro_aux:w 2540, 2543
\token_if_protected_long_macro_p:N *51*, 2456
\token_if_protected_long_macro_p_aux:w 2456
\token_if_protected_macro:N 2525
\token_if_protected_macro:NTF . *51*, 2456
\token_if_protected_macro_aux:w 2526, 2529
\token_if_protected_macro_p:N . *51*, 2456
\token_if_protected_macro_p_aux:w 2456
\token_if_skip_register:N 2492
\token_if_skip_register:NTF . . . *52*, 2456
\token_if_skip_register_aux:w 2496, 2500
\token_if_skip_register_p:N *52*, 2456, 2605
\token_if_skip_register_p_aux:w . . 2456
\token_if_space:N 2410
\token_if_space:NTF *49*, 2410
\token_if_space_p:N *49*, 2410
\token_if_toks_register:N 2514
\token_if_toks_register:NTF . . . *52*, 2456
\token_if_toks_register_aux:w 2518, 2522
\token_if_toks_register_p:N *52*, 2456, 2609
\token_if_toks_register_p_aux:w . . 2456
\token_new:Nn . . . *48*, 2364, 2364, 2369, 2371–2373, 2375–2378, 2619–2621
\token_to_meaning:N *25*, 781, 782, 1126, 2439, 2469, 2476, 2486, 2497, 2508, 2519, 2527, 2534, 2541, 2563, 2569, 2575, 3947
\token_to_str:c *24*, 781, 790, 1636
\token_to_str:N *5*, *24*, 781, 783, 790, 1013, 1017, 1078, 1124, 1131, 1132, 1142, 1161, 1165, 1355, 2147, 2224, 2246, 3544, 4399, 4432, 4565, 4585, 4610, 4789, 6434, 6835
\toks 374
\toks_clear:c *82*, 4265, 4775
\toks_clear:N . . . *82*, 4265, 4265, 4272, 4274, 4277, 4434, 4612, 4774, 4791
\toks_gclear:c *82*, 4265, 4777
\toks_gclear:N . *82*, 4265, 4269, 4275, 4776
\toks_gput_left:cn *83*, 4288
\toks_gput_left:co *83*, 4288
\toks_gput_left:cV *83*, 4288
\toks_gput_left:Nn . . *83*, 4288, 4293, 4298
\toks_gput_left:No *83*, 4288
\toks_gput_left:NV *83*, 4288
\toks_gput_left:Nx *83*, 4288
\toks_gput_right:cn *83*, 4303
\toks_gput_right:co *83*, 4303
\toks_gput_right:cV *83*, 4303
\toks_gput_right:Nn *83*, 4303, 4307, 4327, 4837
\toks_gput_right:No *83*, 4303

\toks_gput_right:NV 83, 4303
\toks_gput_right:Nx 83, 4303
\toks_gset:cn 81, 4244, 5771
\toks_gset:co 81, 4244
\toks_gset:cV 81, 4244
\toks_gset:cx 81, 4244
\toks_gset:Nn 81, 4244, 4244–4246, 4821, 4853
\toks_gset:No 81, 4244
\toks_gset:NV 81, 4244
\toks_gset:Nx 81, 4244
\toks_gset_eq:cc 82, 4247, 4785
\toks_gset_eq:cN 82, 4247, 4784
\toks_gset_eq:Nc 82, 4247, 4783
\toks_gset_eq:NN 82, 4247, 4253, 4261, 4264, 4782
\toks_if_empty:c 4869
\toks_if_empty:cTF 83, 4336
\toks_if_empty:N 4336, 4868
\toks_if_empty:NF 4342, 4436, 4614, 4793
\toks_if_empty:NT 4341
\toks_if_empty:NTF 83, 4336, 4340
\toks_if_empty_p:c 83, 4336
\toks_if_empty_p:N 83, 4336, 4339
\toks_if_eq:cc 4873
\toks_if_eq:ccTF 84, 4343
\toks_if_eq:cN 4871
\toks_if_eq:cNTF 84, 4343
\toks_if_eq:Nc 4872
\toks_if_eq:NcTF 84, 4343
\toks_if_eq:NN 4343, 4870
\toks_if_eq:NNF 4350
\toks_if_eq:NNT 4349
\toks_if_eq:NNTF 84, 4343, 4348
\toks_if_eq_p:cc 84, 4343
\toks_if_eq_p:cN 84, 4343
\toks_if_eq_p:Nc 84, 4343
\toks_if_eq_p:NN 84, 4343, 4347
\toks_new:c 80, 4206, 4773
\toks_new:N 80, 4206, 4210, 4222, 4353–4359, 4772
\toks_new_local:c 80, 4206
\toks_new_local:N . . 80, 4206, 4214, 4223
\toks_put_left:cn 82, 4288
\toks_put_left:co 82, 4288
\toks_put_left:cV 82, 4288
\toks_put_left:Nn 82, 4288, 4288, 4292, 4296
\toks_put_left:No 82, 4288
\toks_put_left:NV 82, 4288
\toks_put_left:Nx 82, 4288
\toks_put_left_aux:w . . . 4288, 4289, 4299
\toks_put_right:cn 83, 4303
\toks_put_right:co 83, 4303
\toks_put_right:cV 83, 4303
\toks_put_right:Nf 83, 4328, 4330
\toks_put_right:Nn 83, 4303, 4303, 4310, 4312, 4326, 4328, 4831, 6958, 6969
\toks_put_right:No 83, 4303, 4320
\toks_put_right:NV 83, 4303, 4314
\toks_put_right:Nx 83, 4303, 4437, 4439, 4615, 4617, 4794, 4796
\toks_set:cf 81, 4226
\toks_set:cn 81, 4226
\toks_set:co 81, 4226
\toks_set:cV 81, 4226
\toks_set:cv 81, 4226
\toks_set:cx 81, 4226
\toks_set:Nf 81, 4226, 4239
\toks_set:Nn 81, 4226, 4227, 4228, 4231, 4243, 4851
\toks_set:No 81, 4226, 4238
\toks_set:NV . . . 81, 4028, 4062, 4226, 4232
\toks_set:Nv 81, 4226, 4235
\toks_set:Nx 81, 4226, 5741
\toks_set_eq:cc 81, 4247, 4781
\toks_set_eq:cN 81, 4247, 4780
\toks_set_eq:Nc 81, 4247, 4779
\toks_set_eq:NN 81, 4247, 4248, 4260, 4263, 4778
\toks_show:c 82, 4286, 4787
\toks_show:N 82, 4286, 4286, 4287, 4443, 4621, 4786, 4802
\toks_use:c 80, 4224
\toks_use:N 80, 4224, 4224, 4225, 4290, 4304, 4316, 4322, 4332, 4344, 4809, 4885, 5248, 5747, 5753, 5775, 5779
\toks_use_clear:c 82, 4276
\toks_use_clear:N 82, 4276, 4276, 4282, 4284
\toks_use_gclear:c 82, 4276
\toks_use_gclear:N . . 82, 4276, 4279, 4285
\toksdef 75
\tolerance 285
\topmark 166
\topmarks 392
\topskip 296
\totalheightof 133, 6877, 6880
\tracingassigns 402
\tracingcommands 141
\tracinggroups 409
\tracingifs 405

\tracinglostchars 142
\tracingmacros 143
\tracingnesting 404
\tracingonline 144
\tracingoutput 145
\tracingpages 146
\tracingparagraphs 147
\tracingrestores 148
\tracingscantokens 403
\tracingstats 149

U

\U 2463
\uccode 384
\uchyph 282
\underline 218
\unexpanded 23, 397
\unhbox 323
\unhcopy 324
\unkern 248
\unless 388
\unpenalty 359
\unskip 246
\unvbox 325
\unvcopy 326
\uppercase 356
\use:c *12*, 838, 838, 949, 1622, 1627, 1637, 1924, 1961, 2910, 3166, 5373, 5420, 5431, 5437, 5446, 5453, 5459, 5465, 5516, 5529, 5537, 6064, 6308, 6921
\use:n *12*, 846, 846, 964, 969, 984, 989, 1356, 2013, 2023, 2033, 2043, 4107, 4136
\use:nn *12*, 846, 847
\use:nnn *12*, 846, 848
\use:nnnn *12*, 846, 849
\use:x *12*, 839, 839
\use_0_parameter: 1233
\use_1_parameter: 1233
\use_2_parameter: 1233
\use_3_parameter: 1233
\use_4_parameter: 1233
\use_5_parameter: 1233
\use_6_parameter: 1233
\use_7_parameter: 1233
\use_8_parameter: 1233
\use_9_parameter: 1233
\use_i:nn *13*, 850, 850, 959, 979, 1031, 1061, 1082, 1307, 4300
\use_i:nnn *13*, 852, 852, 1043, 2563
\use_i:nnnn *13*, 852, 855
\use_i_after_else:nw *14*, 866, 867
\use_i_after_fi:nw *14*, 866, 866, 1343, 3109
\use_i_after_or:nw *14*, 866, 868
\use_i_after_orelse:nw *14*, 866, 869, 1323, 1325, 1327, 1329, 1331, 1333, 1335, 1337, 1339, 1341
\use_i_delimit_by_q_nil:nw *14*, 863, 863, 4164
\use_i_delimit_by_q_recursion_stop:nw *14*, 863, 865, 2190, 2198
\use_i_delimit_by_q_stop:nw *14*, 863, 864, 4490
\use_i_ii:nnn *14*, 852, 859, 1523
\use_ii:nn *13*, 850, 851, 959, 979, 1015, 1033, 1063, 1084, 1309, 5945
\use_ii:nnn . *13*, 852, 853, 1046, 2569, 5988
\use_ii:nnnn *13*, 852, 856
\use_iii:nnn *13*, 852, 854, 2575
\use_iii:nnnn *13*, 852, 857
\use_iv:nnnn *13*, 852, 858
\use_none:n *13*, 870, 870, 1017, 1345, 1686, 1935, 2192, 2200, 3058, 3059, 3063, 4493, 4499, 5151, 5155, 5912, 5942
\use_none:nn *13*, 870, 871, 964, 969, 984, 989, 1645, 2900
\use_none:nnn *13*, 870, 872, 5975
\use_none:nnnn *13*, 870, 873
\use_none:nnnnn *13*, 870, 874
\use_none:nnnnnn *13*, 870, 875
\use_none:nnnnnnn *13*, 870, 876
\use_none:nnnnnnnn *13*, 870, 877
\use_none:nnnnnnnnn . . . *13*, 870, 878, 1313
\use_none_delimit_by_q_nil:w *14*, 860, 860
\use_none_delimit_by_q_recursion_stop:w *14*, 860, 862, 947, 997, 1619, 2178, 2183, 3991, 4662
\use_none_delimit_by_q_stop:w *14*, 860, 861, 2059, 2063, 4489, 4905
\usepackage 734, 5789

V

\vadjust 259, 547
\valign 94
\value 5807
.value_forbidden: *126*
.value_required: *126*
\vbadness 334
\vbox 329

\vbox:n *117*, 5666, 5666
\vbox_gset:cn *117*, 5667
\vbox_gset:Nn *117*, 5667, 5669, 5670
\vbox_gset_inline_begin:N *117*, 5677, 5680
\vbox_gset_inline_end: . *117*, 5677, 5682
\vbox_gset_to_ht:ccn *117*, 5671
\vbox_gset_to_ht:cnn *117*, 5671
\vbox_gset_to_ht:Nnn *117*, 5671, 5675, 5676
\vbox_set:cn *117*, 5667
\vbox_set:Nn *117*, 5667, 5667–5669
\vbox_set_inline_begin:N *117*, 5677, 5677, 5681
\vbox_set_inline_end: . . *117*, 5677, 5679
\vbox_set_split_to_ht:NNn *118*, 5685, 5685
\vbox_set_to_ht:cnn *117*, 5671
\vbox_set_to_ht:Nnn *117*, 5671, 5671, 5674, 5675
\vbox_to_ht:nn *118*, 5683, 5683
\vbox_to_zero:n *118*, 5683, 5684
\vbox_unpack:c *118*, 5688
\vbox_unpack:N *118*, 5688, 5688, 5689
\vbox_unpack_clear:c *118*, 5688
\vbox_unpack_clear:N *118*, 5688, 5690, 5691
\vcenter 180
\vfil 241
\vfill 243
\vfilneg 242
\vfuzz 336
\voffset 311
\voidb@x 5661
\vrule 250
\vsize 293
\vskip 244
\vsplit 322
\vss 245
\vtop 330

W

\WARNING 5770
\wd 377
\widowpenalties 437
\widowpenalty 264
\widthof *133*, 6877, 6878
\write 127

X

\X 2459, 2463
\xdef 68
\xetex_if_engine: 1284, 1293
\xetex_if_engine:TF *4*, *26*, 1279
\xetex_version:D 553, 556, 1279
\XeTeXversion 553
\xleaders 253
\xref_deferred_new:nn *118*, 5733, 5734, 5804, 5807
\xref_define_label:nn *327*, 5763, 5763, 5777
\xref_define_label_aux:nn 5763, 5766, 5768
\xref_get_value:nn *119*, 5749, 5749, 5802, 5805, 5808
\xref_new:nn *118*, 5733, 5733, 5800
\xref_new_aux:nnn 5733, 5733–5735
\xref_set_label:n . *118*, 5774, 5774, 5810
\xref_tmp:w 5775, 5778
\xref_write *327*, 5776, 5791
\xspaceskip 287

Y

\Y 2460, 2463
\year 368

Z

\Z 1648, 1656, 2461, 2463
\z@ 3406