The etextools[1] macros

An e-TeX package providing useful (purely expandable) tools for LaTeX Users and package Writers

Florent CHERVET
florent.chervet@free.fr

Version 3.14
04 October 2009

Contents

Introduction 2
1 · Motivation 2
2 · Purely Expandable macros 2
3 · The example file 2
4 · Requirements 2
5 · Acknowledgements – Thank You ! 2
6 · A note for package writers 2

List of commands

1 ▸ General Helper Macros 4
2 ▸ Expansion control 4
3 ▸ Meaning of control sequences 7
4 ▸ Single tokens/single characters 9
4.1 · The \ifx test and the character test 9
4.2 · Basic test macros 9
5 ▸ Characters and Strings 11
6 ▸ Fully expandable macros with options and modifiers 15
7 ▸ Define control sequences through groups 17
8 ▸ Vectorized \futurelet 18
9 ▸ Lists management 21
9.1 · The natural loop 21
9.2 · Lists of single tokens / characters 21
9.3 · The General Command-List-Parser 23
9.4 · Loops into lists 25
9.5 · Adding elements to csv lists 26
9.6 · Converting lists 26
9.7 · Test if an element is in a list 28
9.8 · Removing elements from lists 28
9.9 · Index of an element in a list 29
9.10 · Arithmetic: lists of numbers 29

LaTeX code

I ▸ Implementation 30
I.1 · Package identification 30
I.2 · Requirements 30
I.3 · Some “helper” macros 31
I.4 · Expansion control 33
I.5 · Meaning of control sequences 34
I.6 · Single tokens / single characters 35
I.7 · Character and Strings 37
I.8 · Purely expandable macros with options 40
I.9 · Define control sequence through groups 42
I.10 · \futuredef 42
I.11 · Loops and Lists Management 46

Revision history 52

Abstract

The etextools package is based on the etex and etoolbox packages and defines a lot of macros for LaTeX Users or package Writers. Before using this package, it is highly recommended to read the documentation (of this package and...) of the etoolbox package.

This package requires the etex package from David Carlisle and the etoolbox package from Philipp Lehman. They are available on CTAN under the /latex/contrib/ directory [2].

The main contributions of etextools are : ⟶ see the complete list

- \expandnext: a vectorized form of **\expandafter** and \ExpandNext that works like \expandnext but expands infinitely (with \expandaftercmds and \ExpandAftercmds)
- a String-Filter constructor to compare strings in a purely expandable way and many other macros on strings among them \ifstrnum
- \futuredef: a macro (and vectorized) version of **\futurelet**.
- the ability to define fully expandable macros with optional parameters or star form (with a small restriction) – \FE@testopt, \FE@ifstar, \FE@ifchar and \FE@modifiers
- a Command-List Parser constructor that uses those new features: command-list parsers are fully expandable: \csvloop, \listloop, \toksloop, \naturalloop and more...

[2] This documentation is produced with the ltxdockit classe and package by Philipp Lehman using the DocStrip utility.
⟶ To get the documentation, run (twice): pdflatex etextools.dtx
⟶ To get the package, run: etex etextools.dtx
The .dtx file is embeded in this pdf thank to embedfile by H. Oberdiek.

Introduction

1 Motivation

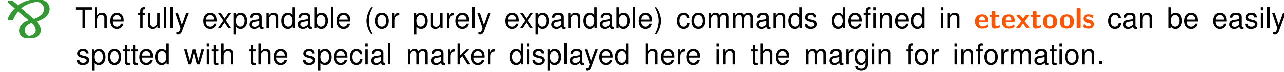
The first motivation for this package was to define a powerful list-parser macro that enhance the one provided by **etoolbox**. Loops are a basic in programming, and the need for them comes sooner or later when using LATEX.

As a result, a lot of "derived" macro have been build, their definition and name carefully chosen... For exemple, removing an element in a list is the same as removing a substring in a string, and then quite the same as testing if two strings are equal...

Finally, **etextools** provides a lot a tools to make definitions of new commands more flexible (modifiers...) maintain list for special purpose (like the lists of purely expandable macros in this very pdf document), to get rid of catcode considerations when dealing with characters (the *character-test*): the list of (nearly all) commands defined by **etextools** lies on next page...

2 Purely Expandable macros

A **purely expandable command** is a command whose expected result can be obtained in an `\edef`. They can also be placed inside `\csname...\endcsname`, and are totally expanded after `\if`, `\ifnum`, `\ifcase`, `\ifcat`, `\number`, `\romannumeral`.



The fully expandable (or purely expandable) commands defined in **etextools** can be easily spotted with the special marker displayed here in the margin for information.

A purely expandable macro may require one, two or many more **levels of expansion** in order to reach its goal. Such macros that expands to the expected result at once are marked with the special sign displayed here in the marginpar. And such macros that requires only two levels of expansions are marked with the special sign displayed here in the marginpar.

levels	sequence to get the result
1	`\expandnext{\def\result}{\FEmacro{⟨arguments⟩}}`
2	`\expandnext\expandnext{\def\result}{\FEmacro{⟨arguments⟩}}`
more	`\ExpandNext{\def\result}{\FEmacro{⟨arguments⟩}}` [3]

A few macros are only expandable if the `\pdfstrcmp` (or `\strcmp`) primitives are available Those macros are marked with the special marker displayed here in the margin for information.

3 The example file

The example file provided with **etextools** illustrates the macros defined here.

4 Requirements

This package requires the packages **etex**[4] by David Carlisle and **etoolbox**[5] by Philipp Lehman. The `\aftergroup@def` macro uses the feature provided by **letltxmacro**[6] by Heiko Oberdiek.

5 Acknowledgements – Thank You !

Thanks to Philipp Lehman for the **etoolbox** package (and also for this nice class of documentation). Much of my work on lists are based on his work and package.

6 A note for package writers

If you are interested in writing your own purely expandable macros (using the features of **etextools**...) it's important to know well the basics: you must understand the job of `\ettl@nbk` and `\romannumeral`, and take a lot of care of malicious spaces.

Happy ε-TEXing

[3] `\ExpandNext` is not alway enough: `\csvloop` for exemple requires `\edef` (or `\csname...`) to be completely expanded.

[4] **etex**: CTAN:macros/latex/contrib/etex-pkg

[5] **etoolbox**: CTAN:macros/latex/contrib/etoolbox

[6] **letltxmacro**: CTAN:macros/latex/contrib/oberdiek/letltxmacro

etextools

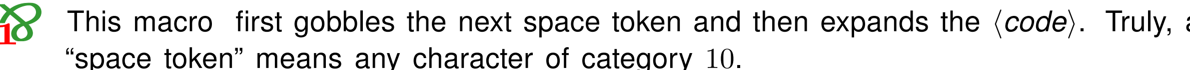
List of Commands Provided

1 ▸ General Helper Macros 4
1 \@gobblespace 4
2 \@gobblescape 4
3 \@swap \@swaparg \@swaplast 4
4 \@swaptwo 4
2 ▸ Expansion control 4
5 \expandaftercmds 5
6 \expandnext 5
7 \expandnexttwo 5
8 \ExpandAftercmds 6
9 \ExpandNext 6
10 \ExpandNextTwo 6
11 \noexpandcs 6
12 \noexpandafter 6
3 ▸ Meaning of control sequences 7
13 \thefontname 7
14 \showcs 7
15 \meaningcs 7
16 \strip@meaning 7
17 \strip@meaningcs 7
18 \parameters@meaning 7
19 \parameters@meaningcs 7
20 \ifdefcount 7
21 \ifdeftoks 7
22 \ifdefdimen \ifdefskip \ifdefmuskip 7
23 \ifdefchar \ifdefmathchar 7
24 \avoidvoid \avoidvoidcs 8
4 ▸ Single tokens/single characters 9
25 \ifsingletoken \ifOneToken 9
26 \ifsinglechar \ifOneChar 10
27 \iffirstchar 10
28 \ifiscs 11
29 \detokenizeChars 11
30 \protectspace 11
5 ▸ Characters and Strings 11
31 \ifempty 11
32 \xifempty 11
33 \ifnotempty 11
34 \xifblank 12
35 \ifnotblank 12
36 \deblank 12
37 \ifstrcmp 12
38 \xifstrequal 12
39 \xifstrcmp 12
40 \ifcharupper \ifcharlower 12
41 \ifuppercase 12
42 \ifstrmatch 12
43 \ifstrdigit 13
44 \ifstrnum 13
45 \DeclareStringFilter 13
6 ▸ Fully expandable macros with options and modifiers 15
46 \FE@testopt 15
47 \FE@ifstar 15
48 \FE@ifchar 15
49 \FE@modifiers 16
50 \ettl@supergobble 16
7 ▸ Define control sequences through groups 17
51 \AfterGroup \AfterGroup* 17
52 \AfterAssignment 17
53 \aftergroup@def 17
8 ▸ Vectorized \futurelet 18
54 \@ifchar 18
55 \ettl@ifnextchar 18
56 \futuredef 18
57 \futuredef= 19
9 ▸ Lists management 21
58 \naturalloop 21
59 \ifintokslist \ifincharlist 21
60 \gettokslistindex 22
61 \getcharlistindex 22
62 \gettokslistcount/token 22
63 \getcharlistcount/token 23
64 \DeclareCmdListParser 23
65 \csvloop \csvloop+ \csvloop! 25
66 \listloop \listloop+ \listloop! 25
67 \toksloop \toksloop+ \toksloop! 25
68 \forcsvloop \forcsvloop+ 26
69 \forlistloop \forlistloop+ 26
70 \fortoksloop \fortoksloop+ 26
71 \csvlistadd/gadd/eadd/xadd 26
72 \csvtolist \tokstolist \listtocsv 27
73 \csvtolistadd \tokstolistadd 27
74 \ifincsvlist \xifincsvlist 28
75 \listdel/gdel/edel/xdel 28
76 \csvdel/gdel/edel/xdel 28
77 \toksdel/gdel/edel/xdel 28
78 \getlistindex 29
79 \getcsvlistindex 29
80 \interval 29
81 \locinterplin 29

All User Commands

1 ▶ General Helper Macros

\@gobblespace{⟨ *code* ⟩}



This macro first gobbles the next space token and then expands the ⟨*code*⟩. Truly, a "space token" means any character of category 10.

\@gobblescape

Just gobble the first character on the result of `\string` (escape character).

`\@gobblescape` is used in the definition of `\DeclareStringFilter`, `\DeclareCmdListParser` and for the general constructor to remove elements from lists (`\listdel` etc.): `\ettl@RemoveInList`.

\@swap{⟨ *token1* ⟩}{⟨ *token2* ⟩}

Just reverse the order of the two tokens: `\@swap#1#2` ⟶ `#2#1`.

`\@swap` does not add any curly braces (be aware that it does not remove them, however).

`\@swap` is so simple that it requires a special attention: `\@swap` is powerful...

```
\@swap{ }\meaning          ⟶  blank space
\expandafter\@swap\expandafter{⟨codeA⟩}{⟨codeB⟩}
```
will expand ⟨*codeA*⟩ once and the put ⟨*codeB*⟩ just before

`\@swap` is used in the definitions of `\expandaftercmds` and `\protectspace`.

\@swaparg{⟨ *code* ⟩}{⟨ *command* ⟩}

Just make ⟨*code*⟩ the first argument of ⟨*command*⟩: `\@swaparg#1#2` ⟶ `#2{#1}`.

`\@swaparg` is used in the definition of `\expandnext`.

\@swaplast{⟨ *token1* ⟩}{⟨ *token2* ⟩}{⟨ *token3* ⟩}

`\@swaplast` swaps ⟨*token2*⟩ and ⟨*token3*⟩ but ⟨*token1*⟩ remains in first position:
`\@swaplast#1#2#3` ⟶ `#1#3#2`

`\@swaplast` is used in the definition of the command-list-parser defined with `\DeclareCmdListParser`.

\@swaptwo{⟨ *token1* ⟩}{⟨ *token2* ⟩}

Just reverse the order of the **arguments:** `\@swaptwo#1#2` ⟶ `{#2}{#1}`.

`\@swaptwo` keeps the curly braces around its arguments (be aware that it does not add them, however).

`\@swaptwo` is used in the definition of `\gettokslistindex` and `\getcharlistindex`.

2 ▶ Expansion control

We often want a control sequence to be expanded after its first argument. It is normally the job of `\expandafter`. With many `\expandafter`s it is always possible to expand once, twice, thrice or more, the **very first token that occurs after the begin-group character** delimiting the argument.

`\expandnext` simplifies the syntax (without making the execution process too heavy).

Now it is also possible to expand the *very first* token **infinitely**: this is the aim of `\ExpandNext`.

\expandaftercmds{⟨ *code* ⟩}{⟨ *control sequences* ⟩}

\expandafter is sometimes limited because it affects only the very next token. **\expandaftercmds** works just like the **\expandafter** primitive but may be followed by arbitrary ⟨*code*⟩, not only a single token.

A typical example is the following code, which detokenizes the character ‘ # ‘:

```
\expandaftercmds{\expandafter\@gobble\string}{\csname #\endcsname}
```

without duplication (\detokenize{#} leads to ‘ ## ‘ if catcode of # is 6)

\expandaftercmds is used in the definition of \ettl@Remove and then in \listdel, and the string-comparators declared with \DeclareStringFilter.

\expandnext{⟨ *code* ⟩}{⟨ *control sequences* ⟩}

\expandnext is quite the same as **\expandaftercmds** except that the ⟨*control sequences*⟩ are the **argument of** ⟨***code***⟩, i. e., they are enclosed with curly braces after expansion.

Suppose you want to test if the replacement text of a macro is blank (only spaces). You will say:

```
\expandafter\ifblank\expandafter {\foo}{⟨true part⟩}{⟨false part⟩}
With \expandnext you'll just have to say:
\expandnext\ifblank{\foo}{⟨true part⟩}{⟨false part⟩}
```

⟨*code*⟩ may be arbitrarily TEX code, unlike \expandafter, you may say:

```
\expandnext{\def\test}{\csname name\endcsname}          and it is exactly:
\edef\test{\expandafter\noexpand\csname name\endcsname}
 and also exactly:
 \expandafter\def\expandafter\test\expandafter{\csname name\endcsname}
 Genauer gesagt: \meaning\test = macro:->\name
```

\expandnext can be used for macros with optional arguments:

```
expandnext{\Macro[option]}{⟨argument⟩}
```

\expandnext can be used to test if a purely expandable macro is expandable at once. (If it is not, the \ExpandNext macro can be used intead.)

Now **\expandnext** behaves like \expandafter and is cumulative: if you need two levels of expansions you may say:

```
\expandnext\expandnext{\def\test}{\csname name\endcsname}
 and it is exactly:
\edef\test{\expandafter\expandafter\expandafter\noexpand\csname name\endcsname}
 and also exactly:
\expandafter\expandafter\expandafter\def\expandafter\expandafter\expandafter\test
            \expandafter\expandafter\expandafter{\csname name\endcsname}
Genauer gesagt: \meaning\test = macro:-> ⟨the meaning of \name⟩
```

\expandnext is an \expandafter saver !

Now observe the following game :

```
\def\foo{foo}      ⟶    \def\Foo{\foo}   ⟵
\def\\FOo{\Foo}    ⟶    \def\FOO{\FOo}   ⟵
\def\fool{\FOO}
```

Guess how many \expandafter are needed to test “\ifblank{foo}” directly from \fool ???

\expandnext solves this problem : \fool has 5 degrees of expansion until it expands to “foo”, therefore exactly 5 \expandnext are required. The solution is:

```
\expandnext\expandnext\expandnext\expandnext\expandnext\ifblank{\fool}
```

\expandnexttwo{⟨ *code* ⟩}{⟨ *control sequences* ⟩}{⟨ *control sequences* ⟩}

\expandnexttwo will act as `\expandnext` on two arguments:

`\expandnexttwo: #1#2#3` ⟶ `\expandnext {\expandnext{#1} {#2} } {#3}`
(`{#2}`: expanded **once after**; `{#3}`: expanded **once first**)

You may easily define `\expandnextthree` the same way, if you need it...

`\expandnexttwo` is used in `\iffirstchar`.

\ExpandAftercmds{⟨ *code* ⟩}{⟨ *control sequences* ⟩}

\ExpandAftercmds acts like the primitive `\expandafter` but:

– the *very first* token in ⟨***control sequences***⟩ is **totally expanded**

– ⟨***code***⟩ may be arbitrarily code (not necessarily a single token)

\ExpandNext{⟨ *code* ⟩}{⟨ *control sequences* ⟩}

More on expansion! Suppose you have a string say "12345" and you wish to reverse the order of the letters (here, the *figures*). To do that we need a macro that swaps two elements, and then group them in order to swap with the next in a loop: the idea is to do: `12345` ⟶ swap `{21}345` ⟶ swap `{321}45` ⟶ swap `{4321}5`.

etextools provides a tool to loop against natural integers from 1 to n. `\naturalloop` is purely expandable and we get the result with:

```
\def\swap#1#2{{#2#1}}
\def\do[#1]#2#3{\swap #3}
\edef\result{\naturalloop[\do]{4}{12345}}   ⟶  macro:->54321
\ExpandNext{\def\RESULT }{\naturalloop{4}{12345}} ⟶ :->54321
```

\ExpandNext has expanded the second argument totally without the use of \edef!

In fact, it is possible because **\naturalloop** is defined in terms of **\ExpandNext**.

`\ExpandNext` is used in the definition of `\naturalloop` and `\DeclareStringFilter`.

\ExpandNextTwo{⟨ *code* ⟩}{⟨ *arg1* ⟩}{⟨ *arg2* ⟩}

`\ExpandNextTwo` will act like `\ExpandNext` on two arguments:

`\ExpandNextTwo: #1#2#3` ⟶ `\ExpandNext {\ExpandNext{#1} {#2} } {#3}`
(`{#2}`: totally expanded **after**; `{#3}`: totally expanded **first**)

You may easily define `\ExpandNextThree` the same way, if you need it...

`\ExpandNextTwo` is used in the final step of `\gettokslistindex` and `\getcharlistindex`.

\noexpandcs{⟨ *csname* ⟩}

In an expansion context (`\edef`) we often want a control sequence whose name results from the expansion of some macros and/or other tokens to be created, but not expanded at that point. Roughly:

`\edef{\noexpandcs{<balanced text to be expanded as a cs-name>}}`

will expand to: `\"cs-name"` but this (new) control sequence itself will not be expanded. A typical use is shown in the following code:

⟶ `\edef\abc{\noexpandcs{abc@\@gobblescape\controlword}}`

⟶ if equivalent to: `\def\abc{\abc@controlword}`.

hint★ `\noexpandcs` may be abbreviated f.ex. in `\"#1"` in `\edef` that take place in a group.

\noexpandafter

`\noexpandafter` only means `\noexpand\expandafter` and is shorter to type.

This command is used in the definition of `\DeclareCmdListParser`.

3 ▶ Meaning of control sequences – determining their type.

\thefontname

\thefontname will display (in Computer Modern font at 10 points) the name of the current font selected. Something like:

```
select font musix11 at 10.0pt
```

\showcs{⟨ *csname* ⟩}

\showcs does \show on the named control sequence.

\meaningcs{⟨ *csname* ⟩}

\meaningcs gives the \meaning of the named control sequence. However, if the control sequence is not defined, \meaningcs expands to \meaning\@undefined (i. e., the word 'undefined') rather than the expected \relax.

\strip@meaning{⟨ *cs-token* ⟩}

\strip@meaningcs{⟨ *csname* ⟩}

\strip@meaning gives the \meaning of the ⟨*cs-token*⟩:

i) without the prefix 'macro:#1#2...->)' if ⟨*cs-token*⟩ is a macro
ii) integrally if ⟨*cs-token*⟩ is defined and is not a macro
iii) expands to an empty string if ⟨*cs-token*⟩ is undefined.

\strip@meaningcs does the same for named control sequences.

\parameters@meaning{⟨ *cs-token* ⟩}

\parameters@meaningcs{⟨ *csname* ⟩}

\parameters@meaning expands to the part of the \meaning which corresponds to the ***parameter string***. If a macro has no parameter, then it expands to an empty string. If the ⟨*cs-token*⟩ or the ⟨*csname*⟩ given is not a macro, it also expands to an empty string.

to summarize

	macro	**not macro**	**undefined**
\meaning	the meaning e. g., macro:[#1]#2->#1#2	the meaning e. g., \count21	undefined
\meaningcs	the meaning e. g., macro:[#1]#2->#1#2	the meaning e. g., \count21	undefined
\strip@meaning	the replacement text e. g.,#1#2	the meaning e. g., \count21	an empty string
\strip@meaningcs	the replacement text e. g.,#1#2	the meaning e. g., \count21	an empty string
\parameters@meaning	the parameter string e. g., [#1]#2	an empty string	an empty string
\parameters@meaningcs	the parameter string e. g., [#1]#2	an empty string	an empty string

\ifdefcount{⟨ *single token* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ifdeftoks{⟨ *single token* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ifdefdimen{⟨*cs-token* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ifdefskip{⟨ *single token* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ifdefmuskip{⟨ *single token* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ifdefchar{⟨ *single token* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ifdefmathchar{⟨ *single token* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

etoolbox provides \ifdefmaco to test if a given control sequence is defined as a macro. etextools provides tests for other types of tokens.

Test is made by a filter on the meaning of the ⟨*single token*⟩ given as argument. The test is always false if this ⟨*single token*⟩ is an undefined control sequence.

`\avoidvoid`[⟨ *replacement code* ⟩]{⟨ *cs-token / string* ⟩}

`\avoidvoid*`[⟨ *replacement code* ⟩]{⟨ *cs-token / string* ⟩}

`\avoidvoid` will test the ⟨*cs-token*⟩ with `\ifdefvoid` (from **etoolbox**). In case ⟨*cs-token*⟩ is void (that means: it is either undefined or has been `\let` to `\relax` or it is a parameterless macro with blank – i. e., empty or space – replacement string), then `\avoidvoid` expands ⟨*replacement code*⟩ (optional parameter whose default is an empty string).

Otherwise, ⟨*cs-token*⟩ is not void (that means: it is defined, its meaning is not `\relax` AND it is either a macro with parameters or a parameterless macro with a replacement string which is NOT blank) then `\avoidvoid` expands ⟨*cs-token*⟩:

```
\avoidvoid {\@undefined}                    will expand to an empty string
\avoidvoid [\macro]\relax                   will expand \macro
\avoidvoid [string is blank]{␣}             will expand string is blank
\avoidvoid*[string is empty]{␣}             will expand ␣
\avoidvoid [\errmessage{string must not be empty}]{some text}
                                            will expand some text
\avoidvoid [\errmessage{macro is void}]\macro
                                            will expand \errmessage{...} if \macro is void
\protected\def\test{␣}
\edef\result{\avoidvoid*\test}
\meaning\result                             macro:->\test  1-expansion of \test not empty
\edef\result{\avoidvoid[other]\test}
\meaning\result                             macro:->other  1-expansion of \test is blank
```

`\avoidvoid` is based on `\ifblank` test, either onto ⟨*string*⟩ or, if ⟨*string*⟩ is in fact a control word (tested with `\ifiscs`) on the replacement text of this control word[7]. If for your special purpose, you prefer to test if the ⟨*string*⟩ (or the replacement text of ⟨*cs-token*⟩) is **really empty and not only blank**, the * star-form of `\avoidvoid` is made for you!

`\avoidvoid` is purely expandable and uses `\FE@ifstar` and `\FE@testopt`: if the mandatory argument is a ⟨*string*⟩ equal to '$*_{12}$' or '$[_{12}$' there will be a problem (and most probably an error). Therefore, **when using `\avoidvoid` you are encourage to specify always an option, even if it is empty.**

`\avoidvoidcs`[⟨ *replacement code* ⟩]{⟨ *csname* ⟩}

`\avoidvoidcs*`[⟨ *replacement code* ⟩]{⟨ *csname* ⟩}

`\avoidvoidcs` will do the same as the former (`\avoidvoid`) but the mandatory argument ⟨*csname*⟩ is interpreted as a control sequence name. Therefore, **you cannot test a string with `\avoidvoidcs`!**

```
\avoidvoidcs{@undefined}                 will expand to an empty string
\avoidvoidcs[\deblank]{zap@space}        will expand to \zap@space
\def\test{This is a test}
\avoidvoidcs[\errmessage{void macro}]{test}
                                         will expand \test
\avoidvoidcs[\errmessage{void macro}]{\test}
                                         will expand \errmessage{void macro}
      this is because \csname This is a test\endcsname is not defined !
Finally, clever !
\protected\def\test{␣}
\avoidvoidcs [other]{test}               will expand other : \test is void
\avoidvoidcs*[other]{test}               will expand \test : \test is not \@empty
\avoidvoidcs [other]\test                will expand \  : control space, which is not void
\avoidvoidcs*[other]\test                will expand \  : control space, which is not void
```

---

[7] if it is defined as a macro. Well: the test occurs on the result of `\strip@meaning` onto the control-sequence

4 ▶ Single tokens/single characters

A single token is either a control word (that means a caracter of category 0 followed by caracters of category 11) or a single character with a valid category code (i. e., $\neq 15$ and $\neq 9$).

4•1 The `\ifx` test and the character test

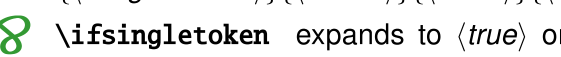
When dealing with single tokens, we need an *equality-test* macro that expands to `\@firstoftwo` in case of equality and `\@secondoftwo` in case of inequality.

etextools implements two such ***equality-test macros***:

1) The `\ifx` test: is the standard test for tokens:
   ⟨*tokenA*⟩ is equal to ⟨*tokenB*⟩ if: `\ifx`⟨*tokenA*⟩⟨*tokenB*⟩ returns **true**
   The `\ifx` test is implemented in `\ettl@ifx`.
2) The **character test** is a bit more sophisticated and works as follow:
   i) if ⟨*tokenA*⟩ and ⟨*tokenB*⟩ have the same category code (tested with an unexpandable `\ifcat`):
   ⟨*tokenA*⟩ is equal to ⟨*tokenB*⟩ if: `\ifx`⟨*tokenA*⟩⟨*tokenB*⟩ returns **true**
   ii) otherwise:
   ⟨*tokenA*⟩ is equal to ⟨*tokenB*⟩ if: `\if\noexpand`⟨*tokenB*⟩`\string`⟨*tokenA*⟩ returns **true**
   The **character test** is implemented in `\ettl@ifchar` and its behaviour may be tested with `\ifsinglechar`.

4•2 Basic test macros

`\ifsingletoken`{⟨ *single token* ⟩}{⟨ *code* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}



`\ifsingletoken` expands to ⟨*true*⟩ only if ⟨***code***⟩ **is a single token and is equal to** ⟨***single token***⟩ **in the sense of `\ifx`.**

`\ifsingletoken` is a **safe `\ifx` test**: ⟨*code*⟩ may be anything (including `\if` conditionals, even not properly closed):

```
\ifsingletoken{A}{A}                will expand ⟨true⟩
\ifsingletoken{\else}{␣\else}       will expand ⟨false⟩
\ifsingletoken{␣}{␣}                will expand ⟨true⟩
\ifsingletoken{\ifx}{\else D\fi}    will expand ⟨false⟩
\ifsingletoken{}{⟨ whatever ⟩}      will expand ⟨true⟩ only if ⟨whatever⟩ is empty !!
\begingroup\catcode`\: 13\global\def\test{:}\endgroup   \catcode`\: 12
\expandnext\ifsingletoken{\test}{:}   will expand ⟨false⟩
```

now clever !

```
\begingroup\catcode`\: 13 \global\let:=\fi \gdef\test{\ifsingletoken :}
\endgroup
\test\fi{⟨true⟩}{⟨false⟩}          will expand ⟨true⟩
```

Be aware that ⟨*single token*⟩ (the first parameter) must be a single token (or empty, but then the test is always false unless ⟨*code*⟩ is empty).

`\ifOneToken`{⟨ *code* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

`\ifOneToken` expands to ⟨*true*⟩ if ⟨***code***⟩ **is a single token**. ⟨*code*⟩ may be anything (including `\if` conditionals, even not properly closed):

```
\ifOneToken{\relax}{␣\relax}       will expand ⟨false⟩
\ifOneToken{\relax}{\relax␣}       will expand ⟨true⟩
\ifOneToken{A}{A␣}                 will expand ⟨false⟩
\ifOneToken{\ifx AB C\else D\fi}   will expand ⟨false⟩
\ifOneToken{C\else D\fi}           will expand ⟨false⟩
```

`\ifOneToken` is used in the definition of `\FE@modifiers`.

\ifsinglechar{⟨ single token ⟩}{⟨ string ⟩}{⟨ true ⟩}{⟨ false ⟩}

\ifsinglechar expands to ⟨*true*⟩ only if **⟨*string*⟩ is a single token and is equal to ⟨*single token*⟩ in the sense of the character-test.**

\ifsinglechar is a **safe *character-test***: ⟨*string*⟩ may be anything (including \if conditionals, even not properly closed):

```
\ifsinglechar{A}{A}              will expand ⟨true⟩
\ifsinglechar{A}{␣A}             will expand ⟨false⟩
\ifsinglechar{␣}{␣}              will expand ⟨true⟩ no matter the number of spaces
\ifsinglechar{\ifx}{\ifx\test\relax YES\else NO\fi}   will expand ⟨false⟩
\ifsinglechar{}{⟨ whatever ⟩} will expand ⟨true⟩ only if ⟨whatever⟩ is empty
\ifsinglechar{\scantokens}{\scantokens}  will expand ⟨true⟩
\begingroup\catcode‘\: 13\global\def\test{:}\endgroup  \catcode‘\: 12
\expandnext\ifsinglechar{\test}{:}  will expand ⟨true⟩
now clever!
\catcode‘\: \active  \let:=\fi
\def\test{\ifsinglechar:}
\let:=\else
\test:{⟨true⟩}{⟨false⟩}          will expand ⟨true⟩
\test\fi{⟨true⟩}{⟨false⟩}        will expand ⟨false⟩
\test\else{⟨true⟩}{⟨false⟩}      will expand ⟨false⟩
```

\ifsinglechar is used in the definition of \FE@ifchar.

\ifOneChar{⟨ string ⟩}{⟨ true ⟩}{⟨ false ⟩}

\ifOneChar expands to ⟨*true*⟩ if **⟨*string*⟩ is a single character.**

⟨*string*⟩ is detokenized before the test (therefore, \relax for example does not contain a single *character*):

```
\ifOneChar{A}          will expand ⟨true⟩
\ifOneChar{␣A}         will expand ⟨false⟩
\ifOneChar{A␣}         will expand ⟨false⟩
\ifOneChar{␣}          will expand ⟨true⟩ (even if there are many spaces !)
\ifOneChar{}           will expand ⟨false⟩
\ifOneChar{\relax}     will expand ⟨false⟩  (\relax is detokenized)
\let\ZERO=0
\ifOneChar{\ZERO}      will expand ⟨false⟩  (\ZERO is detokenized)
```

\ifOneChar is used in \detokenizeChars

\ifOneCharWithBlanks{⟨ string ⟩}{⟨ true ⟩}{⟨ false ⟩}

\ifOneCharWithBlanks switches to ⟨*true*⟩ if and only if ⟨*string*⟩ contains a single **character** possibly with blank spaces before and/or after. It's an optimisation of:

```
\ExpandNext\ifOneChar{\expandnext\deblank{\detokenize{⟨string⟩}}}
```

If ⟨*string*⟩ contains **only spaces**, \ifOneCharWithBlanks expands ⟨***false***⟩.

\iffirstchar{⟨ string1 ⟩}{⟨ string2 ⟩}{⟨ true ⟩}{⟨ false ⟩}

\iffirstchar compares the character codes of the **first** characters of each ⟨*string*⟩. The comparison is *catcode agnostic* and the macro is fully expandable. Neither ⟨*string1*⟩ nor ⟨*string2*⟩ is expanded before comparison. Example:

```
\iffirstchar *{*hello*}{begins with a star}{begins with something else}
```

Alternatively, you may use the \ifstrmatch test.

```
\iffirstchar{⟨⟩}{⟨whatever⟩}   expands⟨true⟩ only if ⟨whatever⟩ is empty.
```

`\ifiscs`{⟨ *string* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

`\ifiscs` will expand ⟨***true***⟩ only **if ⟨*string*⟩ is a single control word.** ⟨*string*⟩ may be anything, including `\if`-conditional, even not properly closed:

`\ifiscs{\MyMacro}`	will expand ⟨***true***⟩
`\ifiscs{x}`	will expand ⟨***false***⟩ — even if x is active
`\ifiscs{\ifx AB C\else D\fi}`	will expand ⟨***false***⟩
`\ifiscs{␣\else}`	will expand ⟨***false***⟩
`\ifiscs{\else␣}`	will expand ⟨***true***⟩
`\ifiscs{␣}`	will expand ⟨***false***⟩
`\ifiscs{\@sptoken}`	will expand ⟨***true***⟩
`\ifiscs{}`	will expand ⟨***false***⟩
`\let\ALPHA=A`	
`\ifiscs{\ALPHA}`	will expand ⟨***true***⟩

`\ifiscs` is an optimized form of: "`\ifOneToken` AND NOT `\ifOneChar`".

`\ifiscs` is used in the definition of the `command-list parsers`.

`\detokenizeChars`{⟨ *list of single tokens* ⟩}

`\detokenizeChars` will selectively detokenize the tokens in ⟨*list of single tokens*⟩. That means: single characters (tested with `\ifOneChar`) are detokenized while control sequences are not detokenized:

```
\edef\result{\detokenizeChars{*+=␣$@\relax\else;}}
\result:      *12+12=12␣10$12@12\relax\else;12
```

`\detokenizeChars` is used in the normal form of `\futuredef`.

`\protectspace`{⟨ *code* ⟩}

`\protectspace` will protect the spaces in ⟨*code*⟩, replacing spaces by a space surrounded by braces:

```
\def\test{abc␣␣def\else\relax\fi ghi␣␣j␣}
\edef\result{\unexpanded\expandafter\expandafter\expandafter{%
                                        \protectspace{\test}}}
\meaning\result:     macro:->abc{␣}def\else \relax \fi ghi{␣}j{␣}
```

N.B.: there is no space after `\fi` in the definition of `\test`...

`\protectspace` is used in `\detokenizeChars`.

`\protectspace` is an example of a **recursive macro which is 2-purely expandable.**

5 ▸ Characters and Strings

`\ifempty`{⟨ *string* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

`\ifempty` is similar to `\ifblank` but it test if a string is really empty (it shall not contain any character nor spaces). To test if the replacement text of a macro is empty, one may use `\ifempty` in conjunction with `\expandnext`:

`\expandnext\ifempty{\macro}` ⟨*true*⟩⟨*false*⟩

`\ifempty` is based on `\detokenize` and accept anything in its argument.

This is NOT: `\expandafter\ifx\expandafter\relax\detokenize{\#1}\relax` !

`\xifempty`{⟨ *string or cs-token* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

`\xifempty` is similar to `\ifempty` but the argument is expanded during comparison.

```
\def\x{\@empty}\def\y{}
\xifempty{\x\y} ⟨true⟩⟨false⟩       will expand ⟨true⟩
```

If pdfTEX is in use, the macro is based on the `\pdfstrcmp` primitive.

\ifnotempty{⟨ *string* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ifnotempty reverses the test of \ifempty.

\xifblank{⟨ *string* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\xifblank
is similar to \ifblank except that the ⟨*string*⟩ is first expanded with \protected@edef.

\ifnotblank{⟨ *string* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ifnotblank reverses the test of \ifblank.

\ifnotblank is a foundamental of purely expandability. It is extensively used in **etextools** but in an optimized form: \ettl@nbk.

\deblank{⟨ *string* ⟩}

\deblank removes all leading and trailing blank spaces from its argument.

An application is for the normalisation of comma separated lists:

```
\csvloop*[\deblank]{   item1 , item2    ,    item3
           , item4   , item5      ,item6   ,
                        item7 ,  item8}%
```
will normalize the list:
```
{item1,item2,item3,item4,item5,item6,item7,item8}
```

This construction is purely expandable:
```
\edef\result{\csvloop [\deblank]{...}}
```
will normalize the list and assign the result to the replacement text of \result.

For more on normalisation, refer to the **kvsetkeys**[8][9] package.

\ifstrcmp{⟨ *string1* ⟩}{⟨ *string2* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

pdfTEX **\ifstrcmp** is based on the \pdfstrcmp primitive (or the XeTeX-\strcmp) if available. Otherwise, \ifstrcmp is \let to **etoolbox**-\ifstrequal.

Neither ⟨*string1*⟩ nor ⟨*string2*⟩ is expanded during comparison. The comparison is *catcode agnostic* (use of \detokenize).

\xifstrequal{⟨ *string1* ⟩}{⟨ *string2* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\xifstrequal is the same as **etoolbox**-\ifstrequal apart that each parameter string is expanded (with \protected@edef) before comparison.

\xifstrcmp{⟨ *string1* ⟩}{⟨ *string2* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

pdfTEX \xifstrcmp is the LATEX form of \pdfstrcmp primitive. If this primitive is not available, \xifstrcmp is \let to \xifstrequal.

⟨*string1*⟩ and ⟨*string2*⟩ are expanded during comparison.

\ifcharupper{⟨ *single char* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ifcharlower{⟨ *single char* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ifcharupper compares with \ifnum the character code of ⟨*single char*⟩ with its \uccode.

\ifcharlower compares with \ifnum the character code of ⟨*single char*⟩ with its \lccode.

\ifuppercase{⟨ *string* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\iflowercase{⟨*string* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ifuppercase compares the ⟨*string*⟩ with \uppercase{⟨*string*⟩}.

\iflowercase compares the ⟨*string*⟩ with \lowercase{⟨*string*⟩}.

The commands are robust.

[8] **kvsetkeys**: CTAN:macros/latex/contrib/kvsetkeys

[9] **kvsetkeys**-normalisation also include a replacement of ' , ' and '=' to ensure that their category code are 12.

`\ifstrmatch`{⟨ *pattern* ⟩}{⟨ *string* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

pdfTEX **\ifstrmatch** is based on the `\pdfmatch` primitive that implements POSIX-regex.

You can test the last character of a string in a purely expandable way by:

`\ifstrmatch{[*]$}{⟨string⟩}`

for example to test ' * ' at the end of a string.

`\ifstrdigit`{⟨ *string* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ifstrdigit expands to ⟨*true*⟩ if ⟨*string*⟩ is a single digit.

A *single digit* is $0, 1, 2, 3, 4, 5, 6, 7, 8$ or 9 without spaces around, no matter of the category code.

`\ifstrnum`{⟨ *string* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ifstrnum expands to ⟨*true*⟩ if ⟨*string*⟩ is a **number in the sense of ε-TEX**, that means:

`\number⟨string⟩` will be the same as: `\deblank{⟨string⟩}`

under the standard catcode regime, if ⟨*string*⟩ is a positive integer.

in other words:

```
\edef\resultA{\number⟨string⟩}
\edef\resultB{\deblank{⟨string⟩}}
\ifx\resultA\resultB        will be true
```

⟨*string*⟩ must be of the form: ␣–␣–␣⋆⋆⋆␣

where blue is optional (one ore more spaces and/or minus signs)

⋆⋆⋆ denotes 1 or more digit(s) without spaces around

for `\ifstrnum` to expand to ⟨*true*⟩.

To tell all the truth, `\ifstrnum` expands ⟨*true*⟩ even if digits have a category code$\neq 12$ whereas `\number` throws an error or stops. But if numbers and minus signs are of category 12 (more than recommended after all...) **\ifstrnum may be a test to check if it is possible to expand \number (or \romannumeral) onto ⟨*string*⟩.**

`\DeclareStringFilter`[⟨\global⟩]{⟨ *command-name* ⟩}{⟨ *stringA* ⟩}

With **\DeclareStringFilter** , you will define **a purely expandable command** designed to test if a string:

- **=** is is **equal** to a *given* string ⟨*stringA*⟩ (with possibly spaces before and after)
- **==** is **strictly equal** to a *given* string ⟨*stringA*⟩ (no spaces allowed)
- **<** **begins with** ⟨*stringA*⟩ (possibly with leading spaces)
- **<=** **strictly begins with** ⟨*stringA*⟩ (no leading spaces allowed)
- **>** **ends with** ⟨*stringA*⟩ (possibly with trailing spaces)
- **>=** **strictly ends with** ⟨*stringA*⟩ (no trailing spaces allowed)
- **?** **contains** ⟨*stringA*⟩, and optionally how many times

and also your ***string-filter*** will be able to

- **–** **remove** ⟨*stringA*⟩ from any string 0, 1 or more times (maximum = `\ettl@intmax`= $2^{13} - 1$ =2 147 483 647)
- **+** **replace** ⟨*stringA*⟩ by any other string 0, 1 or more times
- **!** **count** the number of occurences of ⟨*stringA*⟩ in any string

Equality is \catcode dependent.

You may also check that ⟨*stringA*⟩ may be a blank space (but as for now, you cannot replace blank spaces at the end of the string...).

Let's see how this works (␣ is zero or more spaces):

\DeclareStringFilter\CompareYES{YES} defines **\CompareYES**
\CompareYES is the ***string-filter*** for the string "**YES**" ⟶ ⟨*stringA*⟩

\CompareYES {⟨*string*⟩}{⟨*true*⟩}{⟨*false*⟩} expands ⟨*true*⟩ if ⟨*string*⟩ **is "␣YES␣"**
\CompareYES= {⟨*string*⟩}{⟨*true*⟩}{⟨*false*⟩} is the same
\CompareYES=.{⟨*string*⟩}{⟨*true*⟩}{⟨*false*⟩} is also the same
\CompareYES=={⟨*string*⟩}{⟨*true*⟩}{⟨*false*⟩} expands ⟨*true*⟩ if ⟨*string*⟩ **is "YES"**

\CompareYES< {⟨*string*⟩}{⟨*true*⟩}{⟨*false*⟩} expands ⟨*true*⟩ if ⟨*string*⟩ **begins with "␣YES"**
\CompareYES<={⟨*string*⟩}{⟨*true*⟩}{⟨*false*⟩} expands ⟨*true*⟩ if ⟨*string*⟩ **begins with "YES"**

\CompareYES> {⟨*string*⟩}{⟨*true*⟩}{⟨*false*⟩} expands ⟨*true*⟩ if ⟨*string*⟩ **ends with "YES␣"**
\CompareYES>={⟨*string*⟩}{⟨*true*⟩}{⟨*false*⟩} expands ⟨*true*⟩ if ⟨*string*⟩ **ends with "YES"**

\CompareYES? {⟨*string*⟩}{⟨*true*⟩}{⟨*false*⟩} expands ⟨*true*⟩ if ⟨*string*⟩ **contains "YES"**
\CompareYES?[*n*]{⟨*string*⟩}{⟨*true*⟩}{⟨*false*⟩} expands ⟨*true*⟩ if ⟨*string*⟩
contains "YES" more than *n* times

\CompareYES- {⟨*string*⟩} **removes all occurences** of "**YES**" in ⟨*string*⟩
\CompareYES-[*n*]{⟨*string*⟩} **removes at most *n* occurences** of "**YES**"

\CompareYES+ {⟨*string*⟩}{⟨*stringB*⟩} **replaces all occurences**
of "YES" by ⟨*stringB*⟩ **in** ⟨*string*⟩
\CompareYES+[*n*]{⟨*string*⟩}{⟨*stringB*⟩} **replaces at most *n* occurences**
of "YES" by ⟨*stringB*⟩ **in** ⟨*string*⟩

And finally:
\CompareYES!{⟨*string*⟩} expands to **the number of times "YES"** can be found in ⟨*string*⟩

```
\edef\result{\CompareYES+[2]{She never says YES but he says YES to everything. YES...}{NO}}
\meaning\result:   macro:->She never says NO but he says NO to everything. YES...
```

A problem may arise if the ⟨*string*⟩ to compare is the string '=', because purely expandable tests for modifiers don't make difference between '=' and '{=}'. To avoid this problem, you may say =. or >. or >. instead of =, > and <.

All the same, you may say ?., +. and -. to avoid problems if the ⟨*string*⟩ is '['.

\CompareYES and each of its form are purely expandable thank to \FE@modifiers.

You should not test a ⟨*string*⟩ which contains the following sequence:

$/_8E_{11}n_{11}d_{11}\S_7S_{11}t_{11}r_{11}i_{11}n_{11}g_{11}/_8$

nor a string which contains '$/_8$' because $/_8$ has a special meaning for **etextools**-\ettl@nbk.

6 ▶ Fully expandable macros with options and modifiers

With \ifblank and \ifempty which are purely expandable macros, it becomes possible to write fully expandable macros with an option, **provided that this macro has at least one non-optional argument**, as far as we don't use \futurelet nor any assignment.

\FE@testopt{⟨#1⟩}{⟨ *commands* ⟩}{⟨ *default option* ⟩}

\FE@testopt mimics the behaviour of \@testopt but is Fully Expandable (FE) and can be used as follow:

```
\def\MacroWithOption#1{\FE@testopt{#1}\MacroHasOption{default}}
```

Limitation: \FE@testopt will look for an option if **#1** is '$[_{12}$' (without spaces around). Therefore:

\MacroWithOption{[}{...} will most probably lead to an error... because \FE@testopt is looking for an option. This is the price, for purely expandability (all the same for \FE@ifstar, \FE@ifchar and \FE@modifiers).

Just like \@testopt, \FE@testopt is sensitive to the category code of '$*_{12}$' which must be other.

\FE@testopt is used in the definition of \DeclareStringFilter, \avoidvoid, \ettl@supergobble and \csvtolist.

\FE@ifstar{⟨#1⟩}{⟨ *star-commands* ⟩}{⟨ *non-star commands* ⟩}

Similarly, it becomes possible to mimic the behaviour of \@ifstar but in a fully expandable(FE) way. **\FE@ifstar** can be used as follow:

```
\def\StarOrNotCommand#1{\FE@ifstar{#1}
                          {\StarredCommand}
                          {\NotStarredCommand}}
```

Just like \@ifstar, \FE@ifstar is sensitive to the category code of * which must be other.

\FE@ifstar is used in the definitions of \csvtolist, \listtocsv and \tokstolist.

\FE@ifchar{⟨ *Variant Character* ⟩}{⟨#1⟩}{⟨ *special-commands* ⟩}{⟨ *normal-commands* ⟩}

As a generalisation of \FE@ifstar **etextools** provides **\FE@ifchar** for use with other variants than the *-form.

For example, to define a '+' variant:

```
\def\SpecialFormMacro#1{\FE@ifchar+{#1}
                          {\SpecialFormMacro}
                          {\NormalFormMacro}}
```

Like \@ifchar but ***unlike*** \@ifstar and \FE@ifstar, \@testopt and \FE@testopt **\FE@ifchar** is NOT sensitive to the category code of the ⟨*Variant Character*⟩ (the character-test is used).

Really, **\FE@ifchar** *is based on* \ifsinglechar *therefore the "caracter" to test may be any token, and you may define a purely expandable macro with a '*\relax*' form, a '*\ignorespaces*' form and a '*\afterassignment*' form.* But may be this is useless...

\FE@modifiers{⟨ *Allowed Modifiers* ⟩}{⟨**#1**⟩}{⟨ *1st case* ⟩}{⟨ *2nd case* ⟩}{⟨...⟩}{⟨ *Normal case* ⟩}

\FE@modifiers is a generalization of \FE@ifchar to allow different modifiers for a single macro. The first argument is the ⟨*Allowed Modifiers*⟩ for this macro.

For example, if you want to define a **purely expandable** macro with a *** star** form, a **+ plus** form and a **– minus** form you may say:

```
\def\MySuperMacro #1{\FE@modifiers{ * + - }{#1}
                        {\MySuperStarredMacro}         % first position
                        {\MySuperPlusMacro}            % second position
                        {\MySuperMinusMacro}           % third position
                        {\MySuperMacroWithoutModifier}} % next to last position
```

Then when called by the user, \MySuperMacro will switch to the sub-macro corresponding to the modifier specified (purely expandable macro with different modalities).

\FE@modifiers works as follow:

1) it checks if **#1** is a single character (\ifOneToken does the job)
2) then it tries to find it in the list of ⟨*Allowed Modifiers*⟩ (this is a list of single tokens)
3) if found, the index of the modifier in the list is known, as well as the length of the list. Then, \ettl@supergobble expands the chosen one.

\FE@modifiers uses the character-test. Therefore, single ***character tokens*** are found in the list of ⟨*Allowed Modifiers*⟩ even if their category code don't match.

\FE@modifiers is used in the definition of the string-filters defined with \DeclareStringFilter.

An intesting example of use of \FE@modifiers is given in the implementation of \ettl@lst@modif.

\ettl@supergobble[⟨*code*⟩]{⟨$\mathbf{n}$⟩}{⟨$\mathbf{N}$⟩}{⟨tok_1⟩}...{⟨tok_n⟩}{⟨$\mathbf{TOK}_{\mathbf{n+1}}$⟩}{⟨$tok_{n+2}$⟩}...{⟨$tok_{\mathbf{N}}$⟩}

\ettl@supergobble{⟨$\mathbf{n}$⟩}{⟨$\mathbf{N}$⟩} will:

i) gobble the first ⟨$\mathbf{n}$⟩ tokens (or groups of tokens) it founds just after
ii) keep the ⟨$\mathbf{n+1}$⟩ token
iii) gobble the last tokens ⟨$\mathbf{n+2}$⟩ to ⟨$\mathbf{N}$⟩
iv) then and after all, expand to ⟨$\mathbf{TOK}_{\mathbf{n+1}}$⟩

In other words, the list contains ⟨$\mathbf{N}$⟩ tokens, **\ettl@supergobble** expands the ⟨$\mathbf{n+1}$⟩ and discards the rest.

Now if ⟨$\mathbf{n}$⟩=⟨$\mathbf{N}$⟩, **\ettl@supergobble** gobbles the ⟨$\mathbf{N}$⟩ tokens (including the last).

And if ⟨$\mathbf{n}$⟩>⟨$\mathbf{N}$⟩ or if ⟨$\mathbf{n}$⟩< 0, **\ettl@supergobble** expands to ⟨$\mathbf{TOK}_{\mathbf{N}}$⟩ (the last).

Finally, if the optional parameter [⟨*code*⟩] is specified, it will be appended to the list after ⟨tok_N⟩ (but not in the special case where n=N...).

\ettl@supergobble has been designed for and is used in \FE@modifiers.

If you're interested in what \ettl@supergobble does when ⟨$\mathbf{N}$⟩≤ 0: it does nothing!

7 ▶ Define control sequences through groups

\AfterGroup{⟨ *code* ⟩}

\AfterGroup*{⟨ *code* ⟩}

The \aftergroup primitive does not allow arbitrary code: only a single token may be placed after \aftergroup. **\AfterGroup** allows arbitrary ⟨*code*⟩ to be expanded after \endgroup or an end-group character.

The * star form of \AfterGroup does the same, but expands its argument with \edef:

```
\newcommand\macro[1]{\textbf{Just to see...#1}}
\begingroup
   \newcommand\othermacro[1]{\textbf{will we see...#1}}
   \AfterGroup{\macro{if it works}}
   \AfterGroup*{\expandonce{\othermacro{if it works}}}
\endgroup
```

and here \macro{if it works} will be executed
and here \textbf{will we see...if it works} will be executed

\AfterAssignment{⟨ *code* ⟩}

In the same order of idea, **\AfterAssignment** allows arbitrary ⟨*code*⟩ to be expanded \afterassignment.

\aftergroup@def{⟨ *command* ⟩}

When leaving a group with the end-group character '}' or the execution of **\endgroup** the meaning of the control sequences that where locally defined inside the group are restored to what they were before.

The idea of \aftergroup@def is to keep a control sequence though **\endgroup** or '}'. This is done by redefining it after the group. \aftergroup@def is based on **letltxmacro**[10] and on \AfterGroup just defined. Therefore, \aftergroup@def works with commands with optional arguments declared with LATEX's \newcommand, with robust commands from **etoolbox**-\newrobustcmd and with LATEX's robust commands (\DeclareRobustCommand).

```
{ \newcommand\test[2][default]{ #1 and #2 }
    \aftergroup@def\test
}
```

\test[option]{mandatory} is defined outside the group - but NOT globally

[10] **letltxmacro**: CTAN:macros/latex/contrib/oberdiek/letltxmacro

8 ▶ Vectorized \futurelet: \futuredef

\@ifchar{⟨ *single token* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\@ifchar does the same as LATEX'\@ifstar but for any character (or *modifier*). Whereas \@ifstar-test is sensitive to the category code of the star (the *character* '$*_{12}$' – that means that the category code of * must be 12 as defined in LATEX's kernel), **\@ifchar** is based on the character-test and does not check the equality of category code for single **characters**.

\@ifchar is NOT purely expandable. It relies on \futurelet and on the character-test. The syntax is the same as for \@ifstar with the specification of the (character) token to test:

```
\newcommand\SpecialMacro{\@ifchar+%
                         {\let\modifier=+\GeneralMacro}
                         {\let\modifier=\relax\GeneralMacro}}
```

Unless \@ifstar, \@ifchar is a \long macro...

\ettl@ifnextchar{⟨ *single token* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

\ettl@ifnextchar is the engine for **\@ifchar**. It is based on \futurelet and on the character-test:

```
\begingroup   \catcode`\! \active   \let!=\else
    \gdef \test {\ettl@ifnextchar !{true}{false\@gobble}}
\endgroup
\catcode`\!\active  \let!=\ifodd
\test!              will expand ⟨true⟩
\test\ifodd         will expand ⟨false⟩
\test\else          will expand ⟨false⟩
```

etextools defines a vectorized version of \futurelet. The idea is to say:

\futuredef[⟨*list of allowed tokens*⟩]\macro{⟨*commands to execute next*⟩}

Then \futuredef is a kind of simple scanner for tokens. It can be used to define an *undelimited macro* i. e., a macro that has no delimiter but whose content of arguments is restricted.

\futuredef[⟨ *list of allowed tokens* ⟩]{⟨ \macro ⟩}{⟨ *commands to expand after* ⟩}

\futuredef*[⟨ *list of allowed tokens* ⟩]{⟨ \macro ⟩}{⟨ *commands to expand after* ⟩}

\futuredef will read the following token with **\futurelet**. If that token is in the ⟨*list of allowed tokens*⟩, then it will append it to \macro and continue, scanning the tokens one after another.

Until it founds a token which is not in the ⟨*list of allowed tokens*⟩. Then it stops reading and executes the ⟨*commands to expand after*⟩. Those commands may use the \macro just defined for analyse or whatever the user want.

The space token must be **explicitly specified** in the ⟨*list of allowed tokens*⟩: otherwise \futuredef stops at a space (and executes the ⟨*commands to expand after*⟩).

A token is in the ⟨*list of allowed tokens*⟩ if it can be found in this list using the character-test. This means that if \relax is in the ⟨***list of allowed tokens***⟩, then it will be appended to \macro (if encountered) and if '$\$_3$' is in the ⟨***list of allowed tokens***⟩, any '$' character will be appended to \macro (if encountered) no matter of its category code. If you really absolutely need the \ifx-test, you shall use \futuredef=[11].

[11]this may be the case if, for some reason, you have detokenized the ⟨*list of allowed tokens*⟩ before, and want to skip the expansion of \detokenizeChars which occurs at the beginning of the normal form of \futuredef...

If the ⟨*list of allowed tokens*⟩ is not specified, \futuredef will read all tokens until the next *begin-group* or *end-group* token.

\futuredef may be used instead of \FE@modifiers for (non purely expandable) macros with multiple modifiers. (The modifiers of the \newkeycommand macro in the **keycommand**[12] package are scanned with this feature.) As far as it is based on \futurelet, the limitation of \FE@modifiers (i. e., {*} is the same as * without the braces) is not applicable to \futuredef.

Limitation: as far as \macro has to be correctly defined (it's replacement text must be balanced in begin-group/end-group delimiters) **it is not allowed to have a character of category code 1 or 2** (or a token having been \let to such a character) **in the** ⟨***list of allowed tokens***⟩: \futuredef will stop scanning the next tokens if it encounters a begin-group or an end-group character.

The **star-form** of \futuredef is more dangerous: **\futuredef*** captures the tokens as \futuredef does, storing them into \macro as long as they are in the ⟨*list of allowed tokens*⟩. But if the next token is not in the list, \futuredef* does not stop at first stage but expands this very token and starts again.

Example:

```
\def\test{TeX\relax{*}}
\futuredef[TeX\relax]\macro{"\meaning\macro"}eTeX\test.
                          "macro:->eTeX"     each token is allowed until \test
\futuredef*[TeX\relax]\macro{"\meaning\macro"}eTeX\test.
                          "macro:->eTeXTeX\relax "    \test is expanded and
                                        futuredef stops at begin-group character
```

As an application, it can be used to define an easy interface for \hdashline (the dashed lines in tabulars and arrays provided by the **arydshln** package): modifying \hline in order to give sense to the following:

\hline.. \hline-- \hline== \hline.- \hline.-. etc.

After having collected the allowed tokens with:
\futuredef[.-=]\nexttokens{⟨*commands next*⟩} it is possible to test the pattern given using \pdfstrcmp or \ifstrequal (or even a \string-filter) and, for example, the **\switch** construction of the **boolexpr** package:

```
\switch[\pdfstrcmp{\nexttokens}]%
\case{{..}}\hdashline[parameters]%
\case{{--}}\hdashline[parameters]%
\case{{==}}\hdashline[parameters]%
\case{{.-.}}\hdashline[parameters]%
\otherwise \original@hline%
\endswitch
```

\switch is purely expandable. See **boolexpr**[13] for more information on \switch.

\futuredef=[⟨ *list of allowed tokens* ⟩]{⟨ \macro ⟩}{⟨ *commands to expand after* ⟩}

\futuredef*=[⟨ *list of allowed tokens* ⟩]{⟨ \macro ⟩}{⟨ *commands to expand after* ⟩}

The '**=**' form of \futuredef is the same as \futuredef but the checking of single characters is sensitive to their category code. If a control sequence is in the ⟨*list of allowed tokens*⟩ it is appended to \macro (if encountered) just like the normal \futuredef does. But if it is a single character token, then it is appended to \macro only if the same character with the same ccategory code is found in the ⟨*list of allowed tokens*⟩: otherwise, \futuredef stops reading and executes the ⟨*commands to expand after*⟩.

[12] **keycommand**: CTAN:macros/latex/contrib/keycommand
[13] **boolexpr**: CTAN:macros/latex/contrib/boolexpr

In general, we are not willing this behaviour and the **=** form of `\futuredef` would probably never be used, unless you know that the ⟨*list of allowed tokens*⟩ is already detokenized... Anyway, it was not difficult at all to `implement`.

You may use indifferently `\futuredef*=` or `\futuredef=*`.

9 ▶ Lists management

9•1 The natural loop

\naturalloop[⟨auxiliary commands⟩]{⟨ number of times ⟩}{⟨ argument ⟩}

The \naturalloop macro applies the ⟨*auxiliary commands*⟩ exactly n times onto the ⟨*argument*⟩, i. e.,:

```
\naturalloop [\MyCommand]{3}{⟨argument⟩}
will expand to:
\MyCommand {\MyCommand {\MyCommand {⟨argument⟩}}}
```

(innermost: expanded first; middle: expanded second; outermost: expanded last)

\MyCommand should be purely expandable. In fact, it's a bit more sophisticated: \MyCommand should be defined as:

\MyCommand:macro [#1]#2#3 -> Something to do with #1 #2 and #3
Where:
- **#1:** is the current index of the loop 1, 2, 3 until to n
- **#2:** is the original ⟨*argument*⟩
- **#3:** is the result of the recursion :ie **\do{\do{\do{\do{**⟨*argument*⟩**}}}}** f.ex. in loop of index 4.

If you want a list of integers from 17 to 24 separated by semi-colon:

```
\def\do[#1]#2#3{#3 ; \number\numexpr#2+#1}
\naturalloop{7}{17}
```
⟶ 17 ; 18 ; 19 ; 20 ; 21 ; 22 ; 23 ; 24

Another example is given in the \ExpandNext section.

9•2 Lists of single tokens / characters

Lists of single tokens are a special case of lists: they have no separator. The test for equalty of tokens is made by \ifx and therefore, finding a token in a list of single tokens is always a purely expandable operation.

A ***list of single tokens*** is a list of ***single*** tokens: that means you can't group them with braces (the list may contain the \bgroup and \egroup tokens however).

Lists of single tokens may also be tested with a special test which is \ifx in case of control sequences and a detokenized-\if in case of single characters.

Lists of single characters are used for testing *modifiers* in a purely expandable way. ***modifiers*** are a vectorialisation of \FE@ifstar (and \FE@ifchar).

\ifintokslist{⟨ single token ⟩}{⟨ list of single tokens ⟩}{⟨ true ⟩}{⟨ false ⟩}
\ifincharlist{⟨ single token ⟩}{⟨ list of single tokens ⟩}{⟨ true ⟩}{⟨ false ⟩}

\ifintokslist will switch to ⟨*true*⟩ if the ⟨*single token*⟩ is found in the ⟨*list of single tokens*⟩ while testing against each token of the list using **\ifx**.

\ifintokslist could be tested with \ifnum\getokslistindex{⟨*token*⟩}{⟨*list of tokens*⟩} but \ifintokslist optimises the loop in case the token is in the list.

\ifincharlist will expands ⟨*true*⟩ if the ⟨*single token*⟩ is found in the ⟨*list of single tokens*⟩ but the test for equality of tokens is the character-test.

Therefore, \ifincharlist behaves as follow:

```
\begingroup    \catcode`\!=13 \catcode`\.=8 \catcode`\: 3
   \global\def\mylist{:!\relax=.0}                         | \ifintokslist
\endgroup                                                  |
\expandnext{\ifincharlist!}\mylist{true}{false}       true |    false
\expandnext{\ifincharlist0}\mylist{true}{false}       true |    true
\expandnext{\ifincharlist:}\mylist{true}{false}       true |    false
\expandnext{\ifincharlist\relax}\mylist{true}{false}  true |    true
```

\ifincharlist is used in the definition of \futuredef.

\gettokslistindex{⟨*item*⟩}{⟨*list of single tokens*⟩}

\gettokslistindex expands to the index of ⟨***item***⟩ in the list of single tokens given as a second argument.

Note that the index is $0-$based for consistency with \ifcase (and also with \ettl@supergobble).

It is possible to say:

```
\newcount\result
\result = \gettokslistindex{d}{abcdef}      ⟶  \result= 3
\ifcase \gettokslistindex{d}{abcef}
        what to do if  a
\or     what to do if  b
\or     what to do if  c
\or     etc. etc. etc.
\else   what to do if  d  is not in the list:   ⟶  result= -1
\fi
```

Please, refer to the examples...

This feature is extensively used in \FE@modifiers.

\gettokslistindex is kind of masterpiece of purely expandable programming with ε-TEX

\getcharlistindex{⟨*item*⟩}{⟨*list of single tokens*⟩}

\getcharlistindex expands to the index of ⟨***item***⟩ in the list of single tokens (the index is 0 for the first item, -1 if ⟨*item*⟩ is not in the list). The character-test is used instead of \ifx (see \ifincharlist).

\getcharlistindex is used - indirectly - in the definition of \FE@modifiers.

\gettokslistcount{⟨*list of single tokens*⟩}

\gettokslisttoken{⟨*item*⟩}{⟨*list of single tokens*⟩}

\gettokslistcount, **\gettokslisttoken** and **\gettokslistindex** work all three with the same engine, and this is also the case for \getcharlistcount, \getcharlisttoken and \getcharlistindex. All are fully expandable.

\gettokslistcount gives the number of tokens in the list, while \gettokslisttoken should be seldom used (but it was natural to define it as well).

```
 if you say: \let\plus = +
   \gettokslisttokens{\plus}{ABCD+EFG}  will expand to: +
 and:
   \gettokslisttokens{+}{ABCD\plus EFG}  will expand to: \plus
```

The idea is to loop into the list, testing each token of the list against ⟨*item*⟩ with \ifx. The *test-macro* (together with its own parameters) is a parameter of the *loop-macro*, and therefore, it can be changed without redefining it. As a result, the loop is purely expandable.

Finally, when the loop is finished, the test macro becomes the *give-result-macro* (without \let) and its own parameters are *extracted using projections* (like \@firstoftwo).

The parameters of the *test-macro* include:

- the current index in the list
- the index of the ⟨*item*⟩ found if \ifx returned true
- the name of the *test-macro* to use at the next iteration. Usually it is the *test-macro* itself, but for the last token in the list, this parameter is the *give-result-macro*.

Definition of \ettl@getsinglelist worth a close look!

Back to the begining: lists of single tokens are also lists without separator. Therefore, the other standard macros \toksloop is provided by the general constructor \DeclareCmdListParser invoked with an empty separator.

Unlike \getlistindex, \getcsvlistindex etc., \gettokslistindex, \gettokslistcount and \gettokslisttoken have no star form nor optional parameter. This is because we might be able to test:
\gettokslistindex{*}{⟨*list of single tokens*⟩} or \gettokslistindex{[]}{⟨*list of single tokens*⟩}
and \FE@ifstar or \FE@testopt don't allow this.

\getcharlistcount{⟨*list of single tokens*⟩}

\getcharlisttoken{⟨*item*⟩}{⟨*list of single tokens*⟩}

They work the same way as the -tokslist versions but with the \character test.

\getcharlistcount is exactly the same as \gettokslistcount and is 2-expandable.

9•3 The General Command-List Parser Constructor

The **etoolbox** package provides a way to define list parsers as fully expandable macros: the list parser is able to expand the auxiliary command \do on each item of a list.

Here we provide a \DeclareCmdListParser macro that is compatible and slightly different, because **the auxiliary command is not necessarily \do.** Such a command-list-parser is fully expandable.

The idea is that if \csvloop has been defined as a command-list-parser then, thank to the fully expandable macro \FE@testopt we can call for expansion:

\csvloop{item,item,item} as a shortcut for \csvloop[\do]{item,item,item}
or: \csvloop[\listadd\mylist]{item,item,item}

for example to convert the csv-list into internal **etoolbox** list.

The star-form of \csvloop will be explained below.

\DeclareCmdListParser[⟨\global⟩]{⟨ *command* ⟩}{⟨ *separator* ⟩}

\breakloop{⟨ *code* ⟩}

\DeclareCmdListParser acts in the same way as **etoolbox**-\DeclareListParser and the command-list-parsers defined are sensitive to the category codes of the ⟨*separator*⟩. This ⟨*separator*⟩ may be any sequence of tokens, but the special sequence:

$/_8 E_{11} n_{11} d_{11} §_7 L_{11} i_{11} s_{11} t_{11} /_8$

which is used as the end-of-list-delimiter for any list.

As long as \ettl@nbk is used to check the end of the list, '$/_8$' is not allowed in the list as well. Therefore, you may not try to define lists with '$/_8$' as separator: they are *useless*[14].

To declare a new command-list-parser with ',' (with the current catcode) as a separator you say:

```
\DeclareCmdListParser\myParser{,}
```

[14]Unfortunately, \ettl@nbk requires a single character as a delimiter... The choice for '$/_8$' is explained in the implementation part.

The Command-List-Parser declared: (here `\MyParser`)

- ⟶ is a **purely expandable macro** with three modifiers (`*`, `+` and `!`) an optional parameter (the ***auxiliary macro*** whose default is **\do**) and a mandatory argument (the expanded List or the List-macro)
- ⟶ iterates into the list, giving each element to the ***auxiliary macro***
- ⟶ the ***auxiliary macro*** must be of one of the following form:

`\MyParser`	**macro:#1->** { something to do with **#1**}	**#1** is an element of the list
`\MyParser+`	**macro:[#1]#2->**{ " " " **#1** and **#2**}	**#1** is the index and **#2** the element
`\MyParser!`	expands to the number of elements in the list	

The default is to define command-list-parsers **globally**, in order to make easier the modifications of category code inside a group: if you wish '$+_8$' to be the separator of your list, you will say:

```
\begingroup\catcode`\+=8
\DeclareCmdListParser\MyParser{+}
\endgroup
```

If you rather like a locally-defined command-list-parser, it is always possible, specifiying an empty option: `\DeclareCmdListParser[]\MyLocalParser{+}`. The default option is `\global`, command-list-parsers are always `\long` macros.

You may then use the following syntaxes:

	`\MyParser`	`\myList`	
or:	`\MyParser`	`[\UserCommands]\myList`	
or:	`\MyParser+`	`\myList`	
or:	`\MyParser+`	`[\UserCommands]\myList`	
or:	`\MyParser`	`{item`⟨*sep*⟩`item`⟨*sep*⟩`item}`	
or:	`\MyParser`	`[\UserCommands]{item`⟨*sep*⟩`item`⟨*sep*⟩`item}`	
or:	`\MyParser+`	`{item`⟨*sep*⟩`item`⟨*sep*⟩`item}`	
or:	`\MyParser+`	`[\UserCommands]{item`⟨*sep*⟩`item`⟨*sep*⟩`item}`	
or:	`\MyParser`	`[n]\myList`	expands to item_n
or:	`\MyParser`	`[n]{item`⟨*sep*⟩`item`⟨*sep*⟩`item}`	expands to item_n
or:	`\MyParser!`	`\myList`	expands to the number of elements
or:	`\MyParser!`	`{item`⟨*sep*⟩`item`⟨*sep*⟩`item}`	expands to the number of items
or:	`\MyParser*`	`{item`⟨*sep*⟩`item`⟨*sep*⟩`item}`	
or:	`\MyParser*`	`[\UserCommands]{item`⟨*sep*⟩`item`⟨*sep*⟩`item}`	
or:	`\MyParser+*`	`{item`⟨*sep*⟩`item`⟨*sep*⟩`item}`	
or:	`\MyParser+*`	`[\UserCommands]{item`⟨*sep*⟩`item`⟨*sep*⟩`item}`	
or:	`\MyParser*!`	`[\UserCommands]{item`⟨*sep*⟩`item`⟨*sep*⟩`item}`	

It's possible to break the loop by saying **`\breakloop`** in your `\UserCommands`. **`\breakloop`** will gobble anything until the end-of-list delimiter ($/_8E_{11}n_{11}d_{11}\S_7L_{11}i_{11}s_{11}t_{11}/_8$) and will append the **mandatory** parameter ⟨***code***⟩ after.

'`+*`' and '`*+`' are identical, as well as '`!*`' and '`*!`'.

The **star-form** of `\MyParser` **is seldom used:** `\MyParser` abide by the following rules:

i) it checks if the list parameter (here `\mylist` or `{item`⟨*sep*⟩`item`⟨*sep*⟩`item}` is a single control word (`\ifiscs` does the job)

ii) if this is a single control word, then it is expanded once

iii) otherwise, no expansion of the list occurs

*Therefore, the need for the ⋆ form is only in the special case where the **expanded List** contains a single control-word, not followed by a separator.*

The reader interested in macros with multiple modifiers which may be used in any order can have a look at the definition of \ettl@lst@modif.

Moreover, \DeclareCmdListParser defines a macro named **\forMyParser** to do loops with a syntax very close to LATEX's \@for: see \forcsvloop for more explanation.

9•4 Loops into lists

The following macros are purely expandable loops into comma-separated lists (\csvloop), **etoolbox** list (\listloop) and token lists (lists of tokens without a separator).

All of them are defined using \DeclareCmdListParser.

\csvloop[⟨ *auxiliary commands* ⟩]{⟨ *csvlist-macro or* item, item, item⟩}

\csvloop+[⟨ *auxiliary commands* ⟩]{⟨ *csvlist-macro or* item, item, item⟩}

\csvloop![⟨ *auxiliary commands*⟩]{⟨ *csvlist-macro or* item, item, item⟩}

\csvloop⋆[⟨ *auxiliary commands* ⟩]{⟨ item, item, item ⟩}

\csvloop⋆+[⟨ *auxiliary commands*⟩]{⟨ item, item, item ⟩}

\csvloop⋆![⟨ *auxiliary commands*⟩]{⟨ item, item, item ⟩}

Examples:

\csvloop\mylist is the same as: \csvloop[\do]\mylist
and applies \do sequentially to each element of the comma-separated list.
\do is a user command of the form:
macro: #1 -> { something to do with **#1 = item }**
The star form \csvloop⋆ ***may be*** used when \mylist is already expanded.
The plus form \csvloop+ ***is*** used when \do is of the form:
macro: [#1]#2 -> { something to do with **#1=index** and **#2=item }**
If \do is in fact a number:
\csvloop[4]\mylist will expand to the ***fifth*** element of \mylist
\csvloop!\mylist will expand to the number of elements in \mylist

---

Be aware that indexes in lists are 0-based: they begin with 0.

Remember that the ⋆ form is seldom used: you probably will forget it!

\listloop[⟨ *auxiliary commands* ⟩]{⟨ *Listmacro or expanded List* ⟩}

\listloop+[⟨ *auxiliary commands* ⟩]{⟨ *Listmacro or expanded List* ⟩}

\listloop![⟨ *auxiliary commands* ⟩]{⟨ *expanded List* ⟩}

\listloop⋆(+)(!)[⟨ *auxiliary commands* ⟩]{⟨ *expanded List* ⟩}

\listloop is designed to work with **etoolbox** lists (lists with '$|_3$' as separator). \listloop enhances **etoolbox**-\dolistloop with an optional argument to change the default auxiliary command \do to apply to each item of the list, a + form a ! form and a ⋆ form. It behaves exactly as \csvloop does.

\toksloop[⟨ *auxiliary commands* ⟩]{⟨ *tokenslistmacro or list of single tokens* ⟩}

\toksloop+[⟨ *auxiliary commands* ⟩]{⟨ *tokenslistmacro or list of single tokens* ⟩}

\toksloop![⟨ *auxiliary commands* ⟩]{⟨ *tokenslistmacro or list of single tokens* ⟩}

\toksloop⋆(+)(!)[⟨ *auxiliary commands* ⟩]{⟨ *list of single tokens* ⟩}

\toksloop is a list parser for lists without separator (list of single tokens).

With \toksloop you are able to count the number of characters in a string:

\toksloop!{abcdef} ⟶ 6

Spaces are not counted, however...

\forcsvloop{⟨ csvlistmacro or item,item,item⟩}\do{⟨...#1...⟩}
\forlistloop{⟨ Listmacro or expanded List ⟩}\do{⟨...#1...⟩}
\fortoksloop{⟨ tokenslistmacro or list of single tokens ⟩}\do{⟨...#1...⟩}
\forcsvloop+{⟨ csvlistmacro or item,item,item⟩}\do{⟨...#1=index...#2=element...⟩}
\forlistloop+{⟨ Listmacro or expanded List ⟩}\do{⟨...#1=index...#2=element...⟩}
\fortoksloop+{⟨ tokenslistmacro or list of single tokens ⟩}\do{⟨...#1=index...#2=element...⟩}
\forcsvloop*(+){⟨item,item,item⟩}\do{⟨...#1...⟩}
\forlistloop*(+){⟨ expanded List ⟩}\do{⟨...#1...⟩}
\fortoksloop*(+){⟨ list of single tokens ⟩}\do{⟨...#1...⟩}

Those macros are just like \csvloop, \listloop and \toksloop but the syntax is quite the same as LATEX's **\@for**, but instead of giving a name to the current item being parsed, it is **#1**! (or **#2** with the **+** form).

forloop construct may by nested. Here is an example (merely silly):

```
\forcsvloop*{\relax\meaning\csname,%
             \afterassignment\global\count,%
             \endgroup\topskip}\do{%
                        \fortoksloop*{#1}\do{\meaning##1}}
```

Of course, those macros are NOT purely expandable... They are automatically defined by \DeclareCmdListParser with the name: **\for**name-of-parser.

The **+** form of \forcsvloop et al. are relative to the **+** form of \csvloop et al.: **#1** is the index and **#2** the element. There is no **!** form.

9•5 Adding elements to csv lists

etextools provides a facility to add items to a csvlist.

\csvlistadd{⟨ csvListmacro ⟩}{⟨ item ⟩}
\csvlistgadd{⟨ csvListmacro ⟩}{⟨ item ⟩}
\csvlisteadd{⟨ csvListmacro ⟩}{⟨ item ⟩}
\csvlistxadd{⟨ csvListmacro ⟩}{⟨ item ⟩}

\csvlistadd adds an item to a csvlist. \csvlisteadd expands the ⟨*item*⟩ (with \protected@edef) **before** appending it to ⟨*csvListmacro*⟩, whilst with \csvlistgadd the final assignment to ⟨*csvListmacro*⟩ is global. Finally, \csvlistxadd both expands the ⟨*item*⟩ and makes the assignment global.

These macros are robust.

9•6 Converting lists

Since string filters are sensitive to the category code of the caracters, it is always possible to convert lists (i. e., changing their separator) using them. For exemple, if one wish to convert a comma separated list into a list with '$\&_4$' as separator one may say:

```
\def\mycsvlist{one,two,three,four,five}
\DeclareStringFilter\CompareComma{,}
\begingroup \catcode`\& = 4      this is its standard catcode anyway
\xdef\myNewList{\expandnext{\CompareComma+}\mycsvlist{&}}
\endgroup
```

But there is another way, may be easier:

```
\begingroup \catcode`\& = 4      this is its standard catcode anyway
  \global\def\do#1{\unexpanded{#1&}}
\endgroup
\edef\myNewList{\csvloop[\do]\mycsvlist}
```

Nevertheless, some conversions could be used very often and **etextools** provides a few macros to convert lists easily:

`\csvtolist[⟨ target: Listmacro ⟩]{⟨ source: csvlistmacro or item,item,item⟩}`

`\csvtolist*[⟨ target: Listmacro ⟩]{⟨ source: item,item,item ⟩}`

\csvtolist converts a comma separated list into an internal **etoolbox** list. It is useful to insert more than one item at a time in a list. The ⟨*Listmacro*⟩ (target parameter) is optional and the user may prefer obtain the result in an `\edef`:

```
\csvtolist[\myList]{one,two,three}
```
is the same as:
```
\edef\myList{\csvtolist{one,two,three}}
```
if you want `\myList` to be global, use the second form with `\xdef` instead of `\edef`.

N.B.: the items are not expanded.

The ⋆ star form is seldom used: it is there to inhibits the expansion of ⟨*source:* `item,item,item`⟩. But expansion occurs only if this parameter is a single control word...

`\tokstolist[⟨ target: Listmacro ⟩]{⟨ source: tokenslistmacro or list of single tokens ⟩}`

`\tokstolist*[⟨ target: Listmacro ⟩]{⟨ source: list of single tokens ⟩}`

\tokstolist converts a list of tokens (no separator) into an internal **etoolbox** list:

```
\tokstolist[\myList]{\alpha\beta\gamma\ifeof+*$}
```
is the same as:
```
\edef\myList{\tokstolist{\alpha\beta\gamma\ifeof+*$}}
\meaning\myList:    macro:->\alpha|3\beta|3\gamma|3\ifeof|3+|3*|3$|3
```
if you want `\myList` to be global, use the second form with `\xdef` instead of `\edef`.

N.B.: the items are not expanded.

This is also the first application of the `\toksloop` macro just defined.

`\listtocsv[⟨ target: csvlistmacro ⟩]{⟨ source: Listmacro or expanded List ⟩}`

`\listtocsv*[⟨ target: csvlistmacro ⟩]{⟨ source: Listmacro or expanded List ⟩}`

\listtocsv converts an **etoolbox**-List into a comma separated list. Be aware that the items in the list does not contain commas (`\listtocsv` does not check this point!):

```
\listtocsv[\csvList]\etbList
```
is the same as:
```
\edef\csvList{\listtocsv\etbList}
```
if you want `\csvList` to be global, use the second form with `\xdef` instead of `\edef`.

N.B.: the items are not expanded.

`\csvtolistadd{⟨ target: Listmacro ⟩}{⟨ source: csvlistmacro or item,item,item⟩}`

`\csvtolistadd*{⟨ target: Listmacro ⟩}{⟨ source: item,item,item ⟩}`

\csvtolistadd acts similarly but both arguments are mandatory:

```
\listadd\myList{one}    \listadd\myList{two}
\csvtolistadd\myList{three,four,five}
\meaning\myList:    macro:->one|3two|3three|3four|3five|3
```

`\tokstolistadd{⟨ target: Listmacro ⟩}{⟨ source: tokenslistmacro or list of single tokens⟩}`

`\tokstolistadd*{⟨ target: Listmacro ⟩}{⟨ source: list of single tokens ⟩}`

\tokstolistadd acts similarly but both arguments are mandatory.

The ⋆ star-form inhibits the expansion of ⟨*source*⟩ (which otherwise occurs only if ⟨*source*⟩ is a single control word).

9•7 Test if an element is in a list

etoolbox provides \ifinlist and \xifinlist. Similarly, **etextools** provides:

\ifincsvlist{⟨ *item* ⟩}{⟨ *csvlistmacro or* item,item,item⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}
\xifincsvlist{⟨ *item* ⟩}{⟨ *csvlistmacro or* item,item,item⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}
\ifincsvlist*{⟨ *item* ⟩}{⟨ item,item,item⟩}{⟨ *true* ⟩}{⟨ *false*⟩}
\xifincsvlist*{⟨ *item* ⟩}{⟨ item,item,item⟩}{⟨ *true* ⟩}{⟨ *false*⟩}

These macros are not purely expandable. The search is sensitive to the category code of the characters in ⟨*item*⟩.

9•8 Removing elements from lists

9.8.1 etoolbox lists

The **etoolbox** package provides \listadd, \listgadd, \listeadd and \listxadd commands to add items to a list. **etextools** provides \listdel, \listgdel, \listedel and \listxdel to remove elements from a list.

\listdel[⟨ *deleted* n *times* ⟩]{⟨ *Listmacro* ⟩}{⟨ *item* ⟩}
\listgdel[⟨ *deleted* n *times* ⟩]{⟨ *Listmacro* ⟩}{⟨ *item* ⟩}
\listedel[⟨ *deleted* n *times* ⟩]{⟨ *Listmacro* ⟩}{⟨ *item* ⟩}
\listxdel[⟨ *deleted* n *times* ⟩]{⟨ *Listmacro* ⟩}{⟨ *item* ⟩}

The **\listdel** command removes the element ⟨*item*⟩ from the list ⟨*Listmacro*⟩. Note that the ⟨*Listmacro*⟩ is redefined after deletion. If the list contains more than one element equal to ⟨*item*⟩ each is removed.

\listedel expands the ⟨*item*⟩ (with \protected@edef) **before** deletion, whilst with **\listgdel** the final assignment to (the *shortened*) ⟨*Listmacro*⟩ is global. Finally, **\listxdel** both expands the ⟨*item*⟩ and makes the assignment global.

If the optional parameter ⟨*deleted* n *times*⟩ is specified as a control sequence, the macro does the same but but assigns to this control sequence the number of times ⟨*item*⟩ has been found in the list. If this parameter is not a counter, it is (possibly *re*-)defined as a macro:

```
\newcount\mycounter
\def\myList{one,two,three,two,three,four,five,three}
\listdel[\mycounter]\myList{three}
\the\mycounter      will be  3
```

9.8.2 csv-lists

\csvdel[⟨ *deleted* n *times* ⟩]{⟨ *csvlistmacro* ⟩}{⟨ *item* ⟩}
\csvgdel[⟨ *deleted* n *times* ⟩]{⟨ *csvlistmacro* ⟩}{⟨ *item* ⟩}
\csvedel[⟨ *deleted* n *times* ⟩]{⟨ *csvlistmacro* ⟩}{⟨ *item* ⟩}
\csvxdel[⟨ *deleted* n *times* ⟩]{⟨ *csvlistmacro* ⟩}{⟨ *item* ⟩}

Are similar for comma-separated lists. Those macros are NOT purely expandable.

9.8.3 Lists of single tokens

\toksdel[⟨ *deleted* n *times* ⟩]{⟨ *tokslistmacro* ⟩}{⟨ *item* ⟩}
\toksgdel[⟨ *deleted* n *times* ⟩]{⟨ *tokslistmacro* ⟩}{⟨ *item* ⟩}
\toksedel[⟨ *deleted* n *times* ⟩]{⟨ *tokslistmacro* ⟩}{⟨ *item* ⟩}
\toksxdel[⟨ *deleted* n *times* ⟩]{⟨ *tokslistmacro* ⟩}{⟨ *item* ⟩}

Are similar for lists of single tokens (lists without separator).

9•9 Index of an element in a list

9.9.1 etoolbox-lists

\getlistindex[⟨*result-index(counter or macro)*⟩]{⟨*item*⟩}{⟨*Listmacro*⟩}
\xgetlistindex[⟨*result-index(counter or macro)*⟩]{⟨*item*⟩}{⟨*Listmacro*⟩}
\getlistindex*[⟨*result-index(counter or macro)*⟩]{⟨*item*⟩}{⟨*list*⟩}
\xgetlistindex*[⟨*result-index(counter or macro)*⟩]{⟨*item*⟩}{⟨*list*⟩}

Sometimes it is interesting to know at which offset in a list lies a given item. `\getlistindex` answers to this question. `\xgetlistindex` does the same thing but expands the ⟨*item*⟩ while looking for it in the list.

As for the command-list-parser, the star versions are designed in case the list (in the second argument) is already expanded.

- If ⟨*item*⟩ is not found in the list, `\getlistindex` expands to 0
- If ⟨*item*⟩ is found in first position then `\getlistindex` expands to 1 and so on.

Those macros are not purely expandable.

N.B. If ⟨*result-index*⟩ is not a counter it is (possibly *re*-)defined as macro.

9.9.2 Comma-separated lists

\getcsvlistindex[⟨*result-index(counter or macro)*⟩]{⟨*item*⟩}{⟨*csvlistmacro*⟩}
\xgetcsvlistindex[⟨*result-index(counter or macro)*⟩]{⟨*item*⟩}{⟨*csvlistmacro*⟩}
\getcsvlistindex*[⟨*result-index(counter or macro)*⟩]{⟨*item*⟩}{⟨*item,item,item,...*⟩}
\xgetcsvlistindex*[⟨*result-index(counter or macro)*⟩]{⟨*item*⟩}{⟨*item,item,item,...*⟩}

This is the same as `\getlistindex` but for comma-separated lists.

As for the command-list-parser, the star versions are designed in case the list (in the second argument) is already expanded.

If ⟨*result-index*⟩ is not a counter it is (possibly *re*-)defined as macro.

9•10 Arithmetic: lists of numbers

\interval{⟨ *number* ⟩}{⟨ *sorted comma separated list of numbers* ⟩}

\interval will expand to the interval of ⟨*number*⟩ into the ⟨*sorted csv list of numbers*⟩:

`\interval{0}{3,5,12,20}`	will expand to	0
`\interval{3}{3,5,12,20}`	will expand to	1
`\interval{4}{3,5,12,20}`	will expand to	1
`\interval{5}{3,5,12,20}`	will expand to	2
`\interval{19}{3,5,12,20}`	will expand to	3
`\interval{20}{3,5,12,20}`	will expand to	4
`\interval{21}{3,5,12,20}`	will expand to	4

\locinterplin{⟨ *number* ⟩}{⟨ *sorted csv list of numbers* ⟩}{⟨ *csv list of numbers* ⟩}

\locinterplin will locally and linearly interpolate the series Y_i in ⟨*csv list of numbers*⟩:

\locinterplin{⟨X⟩}{⟨X_i⟩}{⟨Y_i⟩}

finds i such that: $X_i \leqslant X \leqslant X_{i+1}$

and expands to the local linear interpolation Y:

$$Y = Y_i + \frac{X - X_i}{X_{i+1} - X_i}(Y_{i+1} - Y_i)$$

X_i and Y_i must have the same number of elements.

LATEX code

Implementation

I•1 Package identification

```
 1 ⟨*package⟩
 2 \NeedsTeXFormat{LaTeX2e}[1996/12/01]
 3 \ProvidesPackage{etextools}
 4   [2009/10/04 v3.14 e-TeX more useful tools for LaTeX package writers]
 5 \csname ettl@onlyonce\endcsname\let\ettl@onlyonce\endinput
```

I•2 Requirements

This package requires the packages **etex** package by David Carlisle **etoolbox** by Philipp Lehman and **letltxmacro** by Heiko Oberdiek (for `\aftergroup@def`):

```
 6 \RequirePackage{etex,etoolbox,letltxmacro}
```

The divide sign ' / ' (or slash) is given a catcode of 8. **It is used as a delimiter.** This choice is driven by three reasons:

1) ' / ' cannot be used in `\numexpr` expressions if its catcode is different of 12, making unlikely that someone changes its catcode in his document. However, the same is true for ' < ', ' > ', ' = ', ' + ', ' − ' and ' . ' (for dimensions) but:
2) ' / ' is not used in etextools but as a delimiter (whereas ' + ', ' - ', ' < ', ' > ', ' = ' and ' . ' are used with their normal meaning).
3) but why 8 ? if someone changes the catcode of ' / ' it is unlikely that she will choose 8 (the *math subscript* which has nothing to do with /...) whereas it is not so unlikely that someone needs ' / ' as a *tab alignment character* (catcode 4) or a *math shift* (catcode 3) or another special need (catcode 13)... Moreover, catcode 4 may have indesirable side effects if read inside `\halign` or `\valign`. Finally, we could have chosen 7 but then a sequence like: ' $/_7/_7$ ' is read by TEX like ' $^\wedge{}_7{}^\wedge{}_7$ ' with a very special meaning...

Therefore, the choice might not be bad...

```
 7 \let\ettl@AtEnd\@empty
 8 \def\TMP@EnsureCode#1#2{%
 9   \edef\ettl@AtEnd{%
10     \ettl@AtEnd
11     \catcode#1 \the\catcode#1\relax
12   }%
13   \catcode#1 #2\relax
14 }
15 \TMP@EnsureCode{32}{10}% space... just in case
16 \TMP@EnsureCode{47}{8}%  /
17 \TMP@EnsureCode{167}{7}% §
18 \TMP@EnsureCode{164}{7}% ¤
19 \TMP@EnsureCode{95}{11}% _
20 \TMP@EnsureCode{42}{12}% *
21 \TMP@EnsureCode{43}{12}% +
22 \TMP@EnsureCode{45}{12}% -
23 \TMP@EnsureCode{46}{12}% .
24 \TMP@EnsureCode{60}{12}% <
25 \TMP@EnsureCode{61}{12}% =
26 \TMP@EnsureCode{62}{12}% >
27 \TMP@EnsureCode{33}{12}% !
28 \TMP@EnsureCode{152}{13}% ~ for the character test
29 \ifundef\pdfstrcmp{%
30   \TMP@EnsureCode{163}{9}% £ ignore
31   \TMP@EnsureCode{128}{14}% \texteuro comment €
```

```
32 }{\TMP@EnsureCode{163}{14}% £ comment
33   \TMP@EnsureCode{128}{9}% \texteuro ignore
34 }
35 \AtEndOfPackage{\ettl@AtEnd\undef\ettl@AtEnd}
```

I•3 Some “helper” macros

helper macros

```
36 \let\ettl@ifdefined\ifdefined%\ifdefined% turn to \iffalse to test other implementation
37 \long\def\ettl@fi#1\fi{\fi#1}
38 \long\def\ettl@else#1\else#2\fi{\fi#1}
39 \long\def\ettl@or#1\or#2\fi{\fi#1}
40 \def\ettl@expandaftwo{\expandafter\expandafter\expandafter}
41 \def\ettl@expandafthree{\expandafter\expandafter\expandafter%
42                         \expandafter\expandafter\expandafter\expandafter}
43 \cslet{ettl@1of1}\@firstofone    %% for internal use only
44 \cslet{ettl@1of2}\@firstoftwo    %% for internal use only
45 \cslet{ettl@2of2}\@secondoftwo   %% for internal use only
46 \long\def\rmn@firstoftwo#1#2{\z@#1} %% for romannumeral
47 \long\def\rmn@secondoftwo#1#2{\z@#2}%% for romannumeral
48 \long\def\ettl@cdr#1#2\@nil{#2}  %% \@cdr should be a LONG macro
49 \long\def\ettl@car#1#2\@nil{#1}  %% \@car should be a LONG macro
50 \long\csdef{ettl@1of3}#1#2#3{#1}
51 \long\csdef{ettl@2of3}#1#2#3{#2}
52 \long\csdef{ettl@3of3}#1#2#3{#3}
53 \long\csdef{ettl@12of3}#1#2#3{{#1}{#2}}
54 \long\def\ettl@carcar#1#2#3#4{#4}
55 \long\def\ettl@firstspace#1#2#3{\expandafter\ettl@firstsp@ce\detokenize{#1} \\{#3}{#2}/
56 \long\def\ettl@firstsp@ce#1 #2\\{\ettl@nbk#1//}
57 \long\def\ettl@csname#1\endcsname{\fi\endcsname}%% useful to get out of \if
```

\ettl@char

\ettl@char expands to ⟨*true*⟩ if its argument is a single character token. It is used in \ettl@ifnextchar.

```
58 \long\def\ettl@char#1{\csname ettl@\if @\expandafter\ettl@cdr\detokenize{#1}\@nil @%
59    1\else2\fi of2\endcsname}
```

\ettl@intmax

This is the maximum integer allowed by eTEX for \numexpr (2^31-1) and all arithmetic operations:

```
60 \providecommand*\@intmax{2147483647}
61 \def\ettl@intmax{2147483647}
```

\ettl@onlypdfTeX

This is an *internal macro* used by the package: if the ⟨*primitive*⟩ in **#1** is available (e. g., \pdfstrcmp) then the ⟨*command*⟩ in **#2** can be defined, otherwise, the ⟨*command*⟩ is \let to the optional argument **#3**. If there is no such optional argument, the ⟨*command*⟩ throws an error (e. g., \ifstrmatch).

```
62 \def\ettl@onlypdfTeX#1#2{\@testopt{\ettl@only@pdfTeX{#1}{#2}}{}}
63 \def\ettl@only@pdfTeX#1#2[#3]{\ifundef{#1}
64   {\ifblank{#3}
65      {\def#2{\PackageError{etextools}{\string#1\space primitive not found\MessageBreak
66      pdfTeX seems not to be running}
67      {\string#2\space works only if used with pdfTeX (requires \string#1)}}}
68      {\AtEndOfPackage{\let#2=#3}%
69      \PackageWarning{etextools}{\string#1\space primitive not found\MessageBreak
70      Macro \string#2\space has been replaced by \string#3\space\MessageBreak
71      It is not purely expandable}}
72   }\relax}
```

\ettl@nbk

\ettl@nbk is an optimized form of \ifblank. TEX switches to the ⟨*true*⟩ part if the expanded argument (delimited by ‘$/_8/_8$’) is **not blank**.

Usage: \ettl@nbk ⟨*string*⟩$/_8/_8$⟨*true*⟩⟨*false*⟩$/_8/_8$

if ⟨*string*⟩ is blank: **#1**='/', '**#2**=ø, **#3**=⟨*true*⟩, **#4**=⟨*false*⟩
otherwise: **#3**='/', **#4**=⟨*true*⟩ (and **#5**=⟨*false*⟩)

```
73 \long\def\ettl@nbk #1#2/#3#4#5//{#4}
74 \long\def\ettl@nbk@else#1#2/#3#4#5//#6\else#7\fi{\fi#4}
```

\ettl@ney

\ettl@ney is exactly \ifnotempty but with the syntax of \ettl@nbk: it may be used in place of \ettl@nbk:

```
75 \long\def\ettl@ney#1//#2#3//{\romannumeral 0\csname @%
76   \if @\detokenize{#1}@first\else second\fi oftwo\endcsname
77      { #2}{ #3}}
```

\ettl@nbk@cat

\ettl@nbk@cat switches to ⟨*true*⟩ if ⟨*string*⟩ is not blank AND if its first token has the same category code of ⟨*tokenA*⟩:

Usage: \ettl@nbkcat ⟨*tokenA*⟩⟨*string*⟩//⟨*same catcodes*⟩⟨*different catcodes*⟩//

```
78 \long\def\ettl@nbk@cat#1#2#3/#4#5#6//{\ettl@nbk#6//%
79          {\ifcat#1#2\ettl@else#5\else\ettl@fi#6\fi}{#5}//}
```

\ettl@nbk@ifx

\ettl@nbk@ifx switches to ⟨*true*⟩ if ⟨*string*⟩ is not blank AND if its first token is equal to ⟨*tokenA*⟩ in the sense of \ifx:

USAGE: \ettl@nbk@ifx ⟨*tokenA*⟩⟨*string*⟩//⟨*true*⟩⟨*false*⟩//

```
80 \long\def\ettl@nbk@ifx#1#2#3/#4#5#6//{\ettl@nbk#6//%
81          {\ifx#1#2\ettl@else#5\else\ettl@fi#6\fi}{#5}//}
```

\ettl@nbk@if

\ettl@nbk@if switches to ⟨*true*⟩ if ⟨*string*⟩ is not blank AND if its first token is equal to ⟨*tokenA*⟩ in the sense of \if:

USAGE: \ettl@nbk@if ⟨*tokenA*⟩⟨*string*⟩//⟨*true*⟩⟨*false*⟩//

```
82 \long\def\ettl@nbk@if#1#2#3/#4#5#6//%
83      {\ettl@nbk#6//{\if#1#2\ettl@else#5\else\ettl@fi#6\fi}{#5}//}
```

\ettl@nbk@IF

More generally: \ettl@nbk@IF[cat]=\ettl@nbk@ifcat \ettl@nbk@IF[x]=\ettl@nbk@ifx \ettl@nbk@IF[]=\ettl@nbk@if:

```
84 \long\def\ettl@nbk@IF[#1]#2#3#4/#5#6#7//{\ettl@nbk#7//%
85      {\csname if#1\endcsname\ettl@else#6\else\ettl@fi#7\fi}{#6}//}
```

\@gobblespace

```
86 \long\def\@gobblespace#1 {#1}
```

\@gobblescape

This sequence of commands is very often used (even in latex.ltx). So it appears to be better to put it in a macro. It's aim is to reverse the mechanism of \csname...\endcsname:

```
87 \newcommand*\@gobblescape{\romannumeral-`\q\expandafter\@gobble\string}
```

May be we could do better, testing first if the next token is a control sequence...

\@swap

\@swap reverses the order and does not add any curly braces:

```
88 \newcommand\@swap[2]{#2#1}
89 \@swap{ }{\let\ettl@sptoken= }% This makes \ettl@sptoken a space token
```

\@swaparg

\@swaparg reverses the order: the first argument (that will become the second), is considered to be the first argument of the second (!):

```
90 \newcommand\@swaparg[2]{#2{#1}}
```

\@swaplast

\@swaplast reverse the order of two tokens, but keeps the first in first position:

```
91 \newcommand\@swaplast[3]{#1#3#2}
```

\@swaptwo

\@swaptwo reserves the order but keeps the curly braces:

```
92 \newcommand\@swaptwo[2]{{#2}{#1}}
```

this macro is used in \gettokslistindex

I•4 Expansion control

\expandaftercmds

\expandaftercmds generalizes **\expandafter**: arbitrarily ⟨*code*⟩ might be put as a first argument.

The idea is to *swap* the arguments in order to expand the second (in first position after the swap) as many times as there are \expandnexts. At exit, swap again.

```
93 \newcommand\expandaftercmds[2]{%
94    \ifsingletoken\expandaftercmds{#1}
95       {\expandafter@cmds{#2}{\expandafter\expandafter\expandafter}}
96       {\expandafter\@swap\expandafter{#2}{#1}}}
97 \long\def\expandafter@cmds#1#2#3{%
98    \ifsingletoken\expandaftercmds{#1}
99       {\expandafter@cmds{#3}{\expandafter#2#2}}
100      {#2\@swap#2{#3}{#1}}}
```

\expandnext

This code is not properly tricky but if you're eager to understand the job of each \expandafter, it's best to go straight at the log.

```
101 \newcommand\expandnext[2]{%
102    \ifsingletoken\expandnext{#1}
103       {\@expandnext{#2}{\expandafter\expandafter\expandafter}}
104       {\expandafter\@swaparg\expandafter{#2}{#1}}}
105 \long\def\@expandnext#1#2#3{%
106    \ifsingletoken\expandnext{#1}
107       {\@expandnext{#3}{\expandafter#2#2}}
108       {#2\@swaparg#2{#3}{#1}}}
```

\expandnexttwo

\ExpandAftercmds

\ExpandAftercmds acts like the primitive \expandafter but expands totally the second **token**:

```
109 \newcommand\ExpandAftercmds[2]{\expandafter\@swap\expandafter{\romannumeral-`\q#2}{#1}}
```

\ExpandNext

\romannumeral forces the expansion of the second **argument.**

```
110    % I'm not sure it is interesting to use \expandnext here...
111    %\newcommand\ExpandNext[2]{\expandnext{#1}{\romannumeral-`\q#2}}
112 \newcommand\ExpandNext[2]{\expandafter\@swaparg\expandafter{\romannumeral-`\q#2}{#1}}
```

\ExpandNextTwo

```
113 \newcommand\ExpandNextTwo[3]{\ExpandNext{\ExpandNext{#1}{#2}}{#3}}
```

\noexpandcs

\noexpandcs may be abbreviated f.ex. in \`#1` or \"#1" in \edef that take place in a group.

```
114 \newcommand*\noexpandcs[1]{\expandafter\noexpand\csname #1\endcsname}
```

\noexpandafter

\noexpandafter only means \noexpand\expandafter and is shorter to type.

```
115 \newcommand*\noexpandafter{\noexpand\expandafter}
```

I•5 Meaning of control sequences

\thefontname

```
116 \newcommand\thefontname{{\expandafter\ettl@thefontname\expandafter\strip@meaning\the\fon
117 \font\ettl@thefontname=ecrm1000
```

\showcs

\showcs shows the meaning of a named control sequence:

```
118 \providecommand*\showcs[1]{\expandafter\show\csname#1\endcsname}
```

\meaningcs

\meaningcs expands in one level:

```
119 \providecommand\meaningcs[1]{\romannumeral-`\q
120    \csname\ifcsdef{#1}{ettl@meaningcs\endcsname{#1}}
121                       {meaning\endcsname\@undefined}}
122 \def\ettl@meaningcs#1{\expandafter\meaning\csname#1\endcsname}% here we don't need \z@ 
123                                                % because \meaning is neve
```

\strip@meaning

Just give the meaning without the prefix 'macro:'. \strip@prefix will expand to an empty string if its argument is undefined, and to the \meaning if it is not a macro.

\strip@meaningcs

The same but for named control sequences:

```
124 \newcommand*\strip@meaning[1]{\romannumeral\csname\ifdef{#1}%
125       {\ifdefmacro{#1}{ettl@strip@meaning}{ettl@meaning}\endcsname#1}{z@\endcsname}}
126 \providecommand*\strip@meaningcs[1]{\romannumeral\csname\ifcsdef{#1}%
127       {\ifcsmacro{#1}{ettl@strip@meaning}{ettl@meaning}%
128             \expandafter\endcsname\csname#1\endcsname}
129       {z@\endcsname}}
130 \def\ettl@strip@meaning{\expandafter\expandafter\expandafter\z@% for \romannumeral in ca
131    \expandafter\strip@prefix\meaning}
132 \def\ettl@meaning{\expandafter\z@\meaning}
```

\parameters@meaning
\parameters@meaningcs

Expands to the *parameter string* of a macro, or to an empty string if not a macro:

```
133 \providecommand*\parameters@meaning[1]{}
134 \edef\parameters@meaning#1{\unexpanded{\romannumeral\expandafter
135    \expandafter\expandafter\z@\expandafter\ettl@params@meaning%
136          \meaning}#1\detokenize{macro:->}/}
137 \providecommand*\parameters@meaningcs[1]{}
138 \edef\parameters@meaningcs#1{\unexpanded{\romannumeral\ettl@expandafthree\z@
139    \expandafter\expandafter\expandafter\ettl@params@meaning%
140          \expandafter\meaning\csname}#1\endcsname\detokenize{macro:->}/}
141 \edef\ettl@params@meaning{%
142    \def\noexpand\ettl@params@meaning\detokenize{macro:}##1\detokenize{->}##2/{##1}%
143 }\ettl@params@meaning
```

\ifdefcount
\ifdeftoks
\ifdefdimen
\ifdefskip
\ifdefmuskip
\ifdefchar
\ifdefmathchar
\ifdefblankspace
\ifdefthechar
\ifdeftheletter

\ettl@ifdef will defined those five macros (and be undefined itself at the end):

```
144 \def\ettl@ifdef[#1]{\expandafter\ettl@ifd@f\expandafter{#1}}
145 \def\ettl@ifd@f#1#2{%
146    \csdef{ettl@ifdef#2}##1#1##2/End§Meaning/{\ettl@nbk##2//{\rmn@firstoftwo}{\rmn@secon
147    \csedef{ifdef#2}##1{\noexpand\romannumeral\noexpandafter%
148       \noexpandcs{ettl@ifdef#2}\noexpand\meaning##1#1/End§Meaning/}%//{##2}{##3}//}
149 }
150 \ettl@ifdef[\string\count]{count}          % defines   \def\ifdefcount
151 \ettl@ifdef[\string\toks]{toks}            %           \def\ifdeftoks
152 \ettl@ifdef[\string\dimen]{dimen}          %           \def\ifdefdimen
153 \ettl@ifdef[\string\skip]{skip}            %           \def\ifdefskip
154 \ettl@ifdef[\string\muskip]{muskip}        %           \def\ifdefmuskip
155 \ettl@ifdef[\string\char]{char}            %           \def\ifdefchar
156 \ettl@ifdef[\string\mathchar]{mathchar}    %           \def\ifdefmathchar
157 \ettl@ifdef[\detokenize{blank space}]{blankspace}%     \def\ifdefblankspace
158 \ettl@ifdef[\detokenize{the character}]{thechar}%      \def\ifdefthechar
159 \ettl@ifdef[\detokenize{the letter}]{theletter} %      \def\ifdeftheletter
160 \undef\ettl@ifdef\undef\ettl@ifd@f
```

\avoidvoid
\avoidvoid*

\avoivoid[⟨*replacement code*⟩]⟨*cs-token*⟩ will expand the optional parameter (default: an empty string) if the mandatory argument is void (i. e., is either undefined, a token whose meaning is \relax, a parameterless macro whose replacement text is empty). Otherwise, it will expand its mandatory argument (⟨*cs-token*⟩):

```
161 \newcommand\avoidvoid[1]{\romannumeral\FE@ifstar{#1}
162       {\ettl@voidvoid{\ettl@ifdefempty\ifempty}}
163       {\ettl@voidvoid{\ettl@ifdefvoid\ifblank}}}
164 \long\def\ettl@voidvoid#1#2{\FE@testopt{#2}{\ettl@voidv@id#1}{}}
165 \long\def\ettl@voidv@id#1#2[#3]#4{\ifiscs{#4}{#1{#4}}{#2{#4}}{\z@#3}{\z@#4}}
```

and the helper macros:

```
166 \long\def\ettl@ifdefvoid#1{\csname @\ifx#1\relax first%
167    \else\expandafter\expandafter\expandafter\ettl@nbk\strip@meaning#1//{second}{first}/
168    \fi oftwo\endcsname}
169 \long\def\ettl@ifdefempty#1{\expandafter\expandafter\expandafter\ifempty%
170    \expandafter\expandafter\expandafter{\strip@meaning#1}}
```

\avoidvoidcs
\avoidvoidcs*

\avoidvoidcs does the same as \avoidvoid but the mandatory argument ⟨*cs-name*⟩ is interpreted as a control sequence name. Therefore, you cannot test a string with \avoidvoidcs.

\avoidcsvoid is an alias (for neu-neu...):

```
171 \newcommand\avoidvoidcs[1]{\romannumeral\FE@ifstar{#1}
172    {\ettl@avoidvoidcs{\ettl@ifdefempty}}
173    {\ettl@avoidvoidcs{\ettl@ifdefvoid}}}
174 \long\def\ettl@avoidvoidcs#1#2{\FE@testopt{#2}{\ettl@@voidvoidcs#1}{}}
175 \long\def\ettl@@voidvoidcs#1[#2]#3{\csname @\ifcsname#3\endcsname
176    \expandafter#1\csname#3\endcsname{first}{second}\else first\fi
177    oftwo\endcsname{\z@#2}{\z@\csname#3\endcsname}}
```

I•6 Single tokens / single characters

\ettl@ifx

\ettl@ifx is the *equality-test macro* for character-test\ifx test. In is designed to be used inside \csname...\endcsname like:
   \ettl@ifx⟨*tokenA*⟩⟨*tokenB*⟩firstsecond:

```
178 \long\def\ettl@ifx#1#2{\csname ettl@\ifx#1#21\else2\fi of2\endcsname}
```

\ettl@ifchar

\ettl@ifchar is the *equality-test macro* for character-test. It is designed to be in place of \ettl@ifx:

```
179 \long\def\ettl@ifchar#1#2{\csname ettl@\if\noexpand#2\string#11of2\ettl@csname\fi
180    \unless\ifcat\noexpand#1\noexpand#22of2\ettl@csname\fi
181    \ifx#1#21\else2\fi of2\endcsname}
```

\ifsingletoken

\ifsingletoken is a safe \ifx-test:

```
182 \newcommand\ifsingletoken[2]{\csname @\ettl@firstspace{#2}
183    {\ettl@nbk#1#2//{second}{\if @\detokenize{#1#2}@first\else\ifx#1#2first\else second\
184    {\if @\detokenize\expandafter{\ettl@cdr#2\@nil}@%
185          \expandafter\ettl@ifxsingle
186    \else\expandafter\ettl@carcar
187    \fi{#1}{#2}{first}{second}}%
188    oftwo\endcsname}
189 \def\ettl@ifxsingle#1#2#3#4{\ettl@nbk#1//{\ifx#1#2#3\else#4\fi}{#4}//}
```

\ifOneToken

\ifOneToken test if its argument contains only one token (possibly a space token):

```
190 \newcommand\ifOneToken[1]{\romannumeral\csname rmn@\ettl@firstspace{#1}
191    {\ettl@nbk#1//{second}{\if @\detokenize{#1}@second\else first\fi}//}
192    {\if @\detokenize\expandafter{\ettl@cdr#1\@nil}@%
193    first\else second\fi}oftwo\endcsname}
```

\ifsinglechar

Test if **#2** is a single character equal to **#1**:

```
194 \long\def\ifsinglechar#1#2{\romannumeral\csname rmn@\ettl@firstspace{#2}
195    {\ettl@nbk#2//{second}{\if @\detokenize{#1#2}@first\else\ifx#1#2first\else second\fi
196    {\if @\detokenize\expandafter{\ettl@cdr#2\@nil}@%
197          \expandafter\ettl@ifchar
198       \else\expandafter\ettl@carcar
199       \fi{#1}{#2}{first}{second}}%
200    oftwo\endcsname}
```

\ifOneChar

\ifOneChar⟨*string*⟩⟨*true*⟩⟨*false*⟩ detokenizes ⟨*string*⟩ first (see also \ifOneToken):

```
201 \ettl@ifdefined\pdfmatch
202 \newcommand\ifOneChar[1]{\romannumeral\csname rmn@%
203       \ifnum\pdfmatch{\detokenize{^.$}}{\detokenize{#1}}=1 first\else second\fi
204       oftwo\endcsname}
205 \else
206 \newcommand\ifOneChar[1]{\romannumeral\csname rmn@\ettl@firstspace{#1}
207    {\ettl@nbk#1//{second}{\if @\detokenize{#1}@second\else first\fi}//}
208    {\if @\expandafter\ettl@cdr\detokenize{#1}\@nil @%
209     first\else second\fi}oftwo\endcsname}
210 \fi%\pdfmatch
```

\ifOneCharWithBlanks

```
211 \ettl@ifdefined\pdfmatch
212 \newcommand\ifOneCharWithBlanks[1]{\romannumeral\csname rmn@%
213       \ifnum\pdfmatch{\detokenize{^[[:space:]]*[^[:space:]][[:space:]]*$}}{\detokenize{
214       first\else second\fi oftwo\endcsname}
215 \else
216 \newcommand\ifOneCharWithBlanks[1]{\romannumeral\csname rmn@\ettl@nbk#1//%
217       {\expandafter\expandafter\expandafter\ettl@nbk
218             \expandafter\ettl@cdr\detokenize{#1}\@nil//{second}{first}//}%
219       {second}//oftwo\endcsname}
220 \fi
```

\iffirstchar

\iffirstchar test if **#1** and **#2** begins with the same character or token (the character-test is used):

```
221 \newcommand\iffirstchar[2]{\romannumeral\csname rmn@%
222    \ettl@nbk#2//%
223       {\ettl@nbk#1//%
224          {\expandnexttwo\ettl@ifchar{\ettl@car#2\@nil}{\ettl@car#1\@nil}{first}{second}
225          {\if @\detokenize{#1}@secondoftwo\ettl@csname\fi
226          \ettl@firstspace{#2}{first}{second}}//}%
227       {\ettl@nbk#1//%
228          {\if @\detokenize{#2}@secondoftwo\ettl@csname\fi
229          \ettl@firstspace{#1}{first}{second}}
230          {\if @\detokenize{#1#2}@first\else second\fi}}//%
231    oftwo\endcsname}
```

\ifiscs

\ifiscs⟨*string*⟩ expands ⟨*true*⟩ only if ⟨*string*⟩ is a single control-word:

```
232 \newcommand\ifiscs[1]{\romannumeral\csname rmn@\ettl@nbk#1//%
233    {\if @\expandafter\ettl@cdr\detokenize{#1}\@nil @secondoftwo\ettl@csname\fi
234     \if @\detokenize\expandafter{\ettl@cdr#1\@nil}@%
235       \expandafter\ettl@firstspace
236     \else secondoftwo\ettl@csname\fi{#1}{second}{first}}
237    {second}//oftwo\endcsname}
```

\detokenizeChars

\detokenizeChars selectively detokenizes the tokens of the list of single tokens: single characters are detokenized while control sequences remain the same:

```
238 \newcommand\detokenizeChars[1]{\expandafter\ettl@dosinglelist
```

```
239     \expandafter\ettl@do@detokenChars\expandafter{\romannumeral\protectspace{\z@#1}}}
240 \long\def\ettl@do@detokenChars#1{\ifOneChar{#1}\detokenize\unexpanded{#1}}
```

\protectspace

\protectspace puts curly braces (group characters) around spaces in the string given as argument. This is useful for loops into lists (\listloop, \csvloop...). \protectspace is an exemple of a loop which is 2-purely expandable:

```
241 \newcommand\protectspace[1]{\romannumeral\ettl@protectspace#1 /End§String/}
242 \long\def\ettl@protectspace#1 #2/End§String/{\ifempty{#2}{\z@#1}
243     {\expandafter\@swap\expandafter{\romannumeral\ettl@protectspace#2/End§String/}{\z@#1
```

I•7 Character and Strings

\ifempty

\ifempty is based on \detokenize and can manage with any argument.

```
244 \newcommand\ifempty[1]{\romannumeral\csname rmn@\if @\detokenize{#1}@%
245     first\else second\fi oftwo\endcsname}
```

\ifnotempty

\ifnotempty is based on \detokenize and can manage with any argument.

```
246 \newcommand\ifnotempty[1]{\romannumeral\csname rmn@\if @\detokenize{#1}@%
247     second\else first\fi oftwo\endcsname}
```

\xifempty

\xifempty is based on pdf-TeX \pdfstrcmp and work with any argument.

```
248 \newcommand\xifempty[1]{\xifstrcmp{#1}{}}
249 \ettl@onlypdfTeX\pdfstrcmp\xifempty[\xifstrempty]
```

\ifnotblank

\ifnotblank \ifnotblank \ifnotblank **\ifnotblank** ifnotblank reverses the test of \ifblank.

ifnotblank**ifnotblank**

```
250 \long\def\ifnotblank#1#2#3{\ettl@nbk#1//{#2}{#3}//}
```

\xifblank

Just expands the parameter using \protected@edef before testing for \ifblank:

```
251 \newrobustcmd\xifblank[1]{\begingroup
252     \protected@edef\@xifblank{\endgroup
253           \noexpand\ifblank{#1}%
254     }\@xifblank}
```

\deblank

From a code in **environ.sty**.

```
255 \newcommand\deblank[1]{\romannumeral-`\q\ettl@deblank#1/ /}
256 \long\def\ettl@deblank#1 /{\ettl@deblank@i#1/}
257 \long\def\ettl@deblank@i#1/#2{ #1}
```

\ettl@stringify

\ettl@stringify is used in the definition of \ettl@safe@ifx:

```
258 \newcommand\ettl@stringify[1]{\romannumeral-`\q\ettl@expandafthree\@gobblescape%
259           \expandafter\ettl@deblank\detokenize{#1}/ /}
```

\ifstrcmp

The macro is based on the \pdfstrcmp primitive if it is available. Otherwise, \ifstrcmp is the same as **etoolbox**-\ifstrequal.

```
260 \newcommand\ifstrcmp[2]{\romannumeral\csname rmn@%
261     \ifnum\pdfstrcmp{\detokenize{#1}}{\detokenize{#2}}=0 first\else second\fi
262     oftwo\endcsname}
263 \ettl@onlypdfTeX\pdfstrcmp\ifstrcmp[\ifstrequal]
```

\xifstrcmp

The macro is based on the \pdfstrcmp primitive.

```
264 \newcommand\xifstrcmp[2]{\csname @%
265     \ifnum\pdfstrcmp{#1}{#2}=0 first\else second\fi
266     oftwo\endcsname}
267 \ettl@onlypdfTeX\pdfstrcmp\xifstrcmp[\xifstrequal]
```

`\xifstrequal` The macro is based on **etoolbox**-`\ifstrequal`.

```
268 \newrobustcmd\xifstrequal[2]{\begingroup
269   \protected@edef\@xifstrequal{\endgroup\noexpand\ifstrequal{#1}{#2}%
270   }\@xifstrequal}
```

`\ifcharupper` Test if the character code equals to its upper case code:

`\ifcharlower` Test if the character code equals to its lower case code:

```
271 \newcommand\ifcharupper[1]{\romannumeral\csname rmn@%
272   \ifnum`\#1=\uccode`\#1 first\else second\fi oftwo\endcsname}
273 \newcommand\ifcharlower[1]{\romannumeral\csname rmn@%
274   \ifnum`\#1=\lccode`\#1 first\else second\fi oftwo\endcsname}
```

`\ifuppercase` Compares the `\uppercase` transformation of a string with itself:

```
275 \newrobustcmd\ifuppercase[1]{\uppercase{\ifstrcmp{#1}}{#1}}
```

`\iflowercase` Compares the `\lowercase` transformation of a string with itself:

```
276 \newrobustcmd\iflowercase[1]{\lowercase{\ifstrcmp{#1}}{#1}}
```

`\ifstrmatch` The macro is base on the `\pdfmatch` primitive.

pdfTEX

```
277 \newcommand\ifstrmatch[2]{\romannumeral\csname rmn@%
278   \ifnum\pdfmatch{#1}{#2}=1 first\else second\fi oftwo\endcsname}
279 \ettl@onlypdfTeX\pdfmatch\ifstrmatch
```

`\ifstrdigit` `\ifstrdigit` expands ⟨*true*⟩ if ⟨*string*⟩ is a single digit (without spaces):

```
280 \ettl@ifdefined\pdfmatch
281 \newcommand\ifstrdigit[1]{\romannumeral\csname rmn@\ifnum\pdfmatch{\detokenize{^[[:digi
282        {\detokenize{#1}}=1 first\else second\fi oftwo\endcsname}
283 \else
284 \def\do#1{\cslet{ettl@number#1}=#1%
285 }\docsvlist{0,1,2,3,4,5,6,7,8,9}
286 \newcommand\ifstrdigit[1]{\romannumeral\csname rmn@%
287   \ifcsname ettl@number\detokenize{#1}\endcsname first\else second\fi oftwo\endcsname}
288 \fi%\pdfmatch
```

`\ifstrnum` `\ifstrnum` expands ⟨*true*⟩ if ⟨*string*⟩ is a number (integer) in the sense of ε-TEX:

```
289 \ettl@ifdefined\pdfmatch
290 \newcommand\ifstrnum[1]{\romannumeral\csname rmn@\ifnum\pdfmatch
291   {\detokenize{^([[:space:]]*-?)*+[[:digit:]]+[[:space:]]*$}}{\detokenize{#1}}=1 %
292   first\else second\fi oftwo\endcsname}
293 \else
294 \newcommand\ifstrnum[1]{\romannumeral\csname rmn@\ettl@nbk#1//%
295         {\expandafter\ettl@numberminus\detokenize{#1}-/End§String/}{second}//oftwo\end
296 \long\def\ettl@numberminus#1-#2/End§String/{\ettl@nbk#2//%
297         {\ettl@nbk#1//{second}{\ettl@numberminus#2/End§String/}//}%
298         {\expandafter\expandafter\expandafter\ettl@numberspace\deblank{#1} /End§String
299 \long\def\ettl@numberspace#1 #2/End§String/{\ettl@nbk#2//{second}{\ettl@ifstrnum#1/End§
300 \long\def\ettl@ifstrnum#1#2/End§String/{%
301   \ifcsname ettl@number#1\endcsname% #1 detokenized before, ok
302     \ettl@nbk#2//{\ettl@ifstrnum#2/End§String/}{first}//%
303   \else second%
304   \fi}
305 \fi%\pdfmatch
```

`\DeclareStringFilter` **`\DeclareStringFilter`** is the general contructor for purely expandable **string-filter** macros:

```
306 \newrobustcmd\DeclareStringFilter[3][\global]{\@ifdefinable#2%
307   {\expandnext\ettl@declarestrfilter%
```

```
308            {\csname\@gobblescape#2\detokenize{->"#3"}\endcsname}{#1}{#2}{#3}}}
309 \newcommand\ettl@declarestrfilter[4]{%
310    #2\csdef{\@gobblescape#1}##1#4##2/End§String/{##1/##2}% This the FILTER
311    #2\long\def#3##1{\FE@modifiers{=<>?-+!}{##1}
312       {\ettl@strfilt@mod 0{{#4}{}{#1}[1]}}%=
313       {\ettl@strfilt@mod 1{{#4}{}{#1}[1]}}%<
314       {\ettl@strfilt@mod 2{{#4}{}{#1}[\ettl@intmax]}}%>
315       {\ettl@strfilt@mod 3{{#4}{}{#1}}}%?
316       {\ettl@strfilt@mod 4{{#4}{}{#1}}}%-
317       {\ettl@strfilt@mod 5{{#4}{}{#1}}}%+
318       {\ettl@strfilt\ettl@strfilt@count{#4}{}{#1}[\ettl@intmax]}%!
319       {\ettl@strfilt\ettl@strfilt@equal{#4}{}{#1}[1]}}}% default
```

\ettl@strfilt@mod

`\ettl@strfilt@mod` test the possible second modifier and choose the right macro to expand with the right arguments:

```
320 \def\ettl@strfilt@mod #1#2#3{%
321    \ifcase#1 \ettl@or\ettl@ifchardot{#3}%
322                {\ettl@strfilt\ettl@strfilt@equal#2}
323                {\FE@ifcharequal{#3}%
324                   {\ettl@strfilt\ettl@strfilt@equaleq#2}%
325                   {\ettl@strfilt\ettl@strfilt@equal#2}}%
326    \or\ettl@or\ettl@ifchardot{#3}%
327                {\ettl@strfilt\ettl@strfilt@start#2}%
328                {\FE@ifcharequal{#3}
329                   {\ettl@strfilt\ettl@strfilt@starteq#2}%
330                   {\ettl@strfilt\ettl@strfilt@start#2}}%
331    \or\ettl@or\ettl@ifchardot{#3}%
332                {\ettl@strfilt\ettl@strfilt@endby#2}%
333                {\FE@ifcharequal{#3}
334                   {\ettl@strfilt\ettl@strfilt@endbyeq#2}%
335                   {\ettl@strfilt\ettl@strfilt@endby#2}}%
336    \or\ettl@or\ettl@ifchardot{#3}%
337                {\ettl@strfilt\ettl@strfilt@instr#2[1]}
338                {\FE@testopt{#3}{\ettl@strfilt\ettl@strfilt@instr#2}{1}}%
339    \or\ettl@or\ettl@ifchardot{#3}%
340                {\ettl@strfilt@REMOVE{#2}[\ettl@intmax]}%
341                {\FE@testopt{#3}{\ettl@strfilt@REMOVE{#2}}{\ettl@intmax}}%
342    \or\ettl@fi\ettl@ifchardot{#3}%
343                {\ettl@strfilt@REPLACE#2[\ettl@intmax]}%
344                {\FE@testopt{#3}{\ettl@strfilt@REPLACE#2}{\ettl@intmax}}%
345    \fi}
```

\ettl@strfilt

`\ettl@strfilt` is the common start for the loop:

```
346 \long\def\ettl@strfilt#1#2#3#4[#5]#6{% % #1 = test macro
347 % #2 = substr
348 % #3 = replacement
349 % #4 = filter macro
350 % #5 = number of times
351 % #6 = user-given string
352     \ExpandAftercmds#1{\ettl@Remove #6/End§String/{#2}{#3}[{#5}]{#4}}}
```

\ettl@strfilt@REMOVE

`\ettl@strfilt@REMOVE` is a pre-stage just before the common `\ettl@strfilt`:

```
353 \long\def\ettl@strfilt@REMOVE #1[#2]{%
354 % #1 = arguments for \ettl@strfilt
355 % #2 = number of times
356    \ifnum\numexpr#2>0 \ettl@else\ettl@strfilt\ettl@strfilt@remove#1[#2]%
357    \else\expandafter\@firstofone%
358    \fi}
```

\ettl@strfilt@REPLACE

`\ettl@strfilt@REPLACE` is a pre-stage just before the common `\ettl@strfilt`:

```
359 \long\def\ettl@strfilt@REPLACE #1#2#3#4[#5]#6#7{%
```

```
360    \ifnum\numexpr#5>0 \ettl@else\ettl@strfilt\ettl@strfilt@replace{#1}{#7}{#3}[{#5}]{#6
361    \else\expandafter\@firstoftwo%
362    \fi}
```

\ettl@Remove

\ettl@Remove applies the filter (#5) and give the result to \ettl@Remove@loop:

```
363 \long\def\ettl@Remove#1/End§String/#2#3[#4]#5{%
364 % #1 = string or list
365 % #2 = substring or item to remove
366 % #3 = REPLACEMENT
367 % #4 = number of times to remove
368 % #5 = filter macro
369    \expandafter\ettl@Remove@loop #5#1//#2/End§String//End§String/{#3}[{#4-1}]{#5}}
```

\ettl@Remove@loop

\ettl@Remove@loop is entitled to break the loop:

```
370 \long\def\ettl@Remove@loop#1/#2//#3/End§String/#4[#5]#6{%
371 % #1 = str before filter
372 % #2 = str after filter
373 % #3 = substr to remove
374 % #4 = REPLACEMENT
375 % #5 = iterindex
376 % #6 = filter macro
377    \ifnum\numexpr#5>0 \ettl@nbk@else#2//%
378            {\ettl@Remove #1#4#2/End§String/{#3}{#4}[{#5}]{#6}}
379            {{#1}{#4#2}{#3}{#5}}//%
380    \else\ettl@fi{#1}{#4#2}{#3}{#5}%
381    \fi}
```

est and result macros

Those macros are expanded after the end of the loop: they give the final expected result from the four registers availblable at the end of the loop:

```
382 \long\def\ettl@strfilt@equal    #1#2#3#4{\csname @%
383       \ettl@nbk#3//{\ettl@nbk#1#2//{second}{first}//}{second}//oftwo\endcsname}
384 \long\def\ettl@strfilt@equaleq #1#2#3#4{\csname @%
385       \ettl@nbk#3//{\ifnotempty{#1#2}{second}{first}}{second}//oftwo\endcsname}
386 \long\def\ettl@strfilt@start    #1#2#3#4{\csname @%
387       \ettl@nbk#1//{second}{first}//oftwo\endcsname}
388 \long\def\ettl@strfilt@starteq #1#2#3#4{\csname @%
389       \ifnotempty{#1}{second}{first}oftwo\endcsname}
390 \long\def\ettl@strfilt@endby    #1#2#3#4{\csname @%
391       \ettl@nbk#3//{first}{second}//oftwo\endcsname}
392 \long\def\ettl@strfilt@endbyeq #1#2#3#4{\csname @%
393       \ettl@nbk#3//{\ifempty{#2}{first}{second}}{second}//oftwo\endcsname}
394 \long\def\ettl@strfilt@count    #1#2#3#4{\number\numexpr\ettl@intmax-(#4)-\ettl@nbk#3//0
395 \long\def\ettl@strfilt@instr    #1#2#3#4{\csname @%
396       \ifnum\numexpr#4>0 second%
397       \else\ifnum\numexpr#4<0 first%
398       \else\ettl@nbk#3//{first}{second}//%
399       \fi\fi oftwo\endcsname}
400 \long\def\ettl@strfilt@remove   #1#2#3#4{#1#2}
401 \long\def\ettl@strfilt@replace #1#2#3#4{#1\ettl@nbk#3//{#2}{}//}
```

I•8 Purely expandable macros with options

basic string filter

This basic string filter will be used for \FE@testopt and \FE@ifstar. As far as the later are used in the definition of \FE@modifiers we can't use the \general string filter contructor to do the job (infinite recursion).

```
402 \long\def\ettl@BasicFilter#1#2#3/End§String/{\expandafter\ettl@B@sicFilter #1#3//#2/End
403 \long\def\ettl@B@sicFilter#1/#2//#3/End§String/{@\ettl@nbk#3//%
404       {\if @\detokenize{#1#2}@first\else second\fi}
405       {second}//oftwo}
```

\FE@testopt Purely expandable \@testopt-like test:

```
406 \newcommand\FE@testopt[3]{\ettl@FE@testopt#1/[/%
407                  {#2#1}%
408                  {#2[{#3}]{#1}}}%]
409 \long\def\ettl@FE@testopt#1[#2/#3#{\csname @\if @\detokenize{#1#2}@%
410    first\else second\fi oftwo\endcsname}
```

\FE@ifstar Purely expandable \@ifstar-like test:

```
411 \newcommand\FE@ifstar[3]{\ettl@FE@ifstar#1/*/%
412                  {#2}%
413                  {#3{#1}}}
414 \long\def\ettl@FE@ifstar#1*#2/#3#{\csname @\if @\detokenize{#1#2}@%
415    first\else second\fi oftwo\endcsname}
```

\FE@ifcharequal This is the same as \FE@ifstar but for '=' character (used in \DeclareStringFilter):

```
416 \newcommand\FE@ifcharequal[3]{\ettl@FE@charequal#1/=/%
417                  {#2}%
418                  {#3{#1}}}
419 \long\def\ettl@FE@charequal#1=#2/#3#{\csname @\if @\detokenize{#1#2}@%
420    first\else second\fi oftwo\endcsname}
```

\ettl@ifchardot Used by \ettl@strfilt@mod to test if a character is a dot. It is used internally and is not the same as \FE@ifchar.

```
421 \newcommand\ettl@ifchardot[1]{\ettl@FE@chardot#1/./}
422 \long\def\ettl@FE@chardot#1.#2/#3#{\csname @\if @\detokenize{#1#2}@%
423    first\else second\fi oftwo\endcsname}
```

\FE@ifchar \FE@ifchar test if the character token following the macro is a single character equal to ⟨*Character*⟩:
USAGE: \FE@ifchar{⟨*Character*⟩}{**#1**}{\SpecialFormMacro}{\NormalMacro}:

```
424 \newcommand\FE@ifchar[4]{\ifsinglechar{#1}{#2}{#3}{#4{#2}}}
```

\FE@modifiers \FE@modifiers test if the character token following the macro is in the list of ⟨*Allowed Characters*⟩:
USAGE:
\FE@modifiers{⟨*Allowed Characters*⟩}{**#1**}{\MacroA}...{\MacroZ}{\NormalMacro}:

```
425 \newcommand\FE@modifiers[2]{%
426    \ifOneToken{#2}%
427       {\ExpandAftercmds\ettl@FE@modifiers%
428                  {\ExpandAftercmds{\ettl@setresult 12of3><}
429                     {\ettl@getsinglelist{\ettl@ifchar{#2}}{#1}}}{#2}}
430       {\ExpandNextTwo{\ettl@supergobble[{{#2}}]}{-1}{\getcharlistcount{#1}+1}}}
431 \long\def\ettl@FE@modifiers#1#2#3{\expandafter\ettl@supergobble%
432    \expandafter[\romannumeral-`\q\ifnum#2<0 {#3}\fi]{#2}{#1+1}}
```

\ettl@supergobble \ettl@supergobble gobbles the n first (groups of) tokens in the following list of N (groups of) tokens and expands the n+1. The macro is optimized (cf \ettl@supergobbleheight etc.) to avoid too long loops.

```
433 \newcommand\ettl@supergobble[1]{\FE@testopt{#1}\ettl@superg@bble{}}
434 \long\def\ettl@superg@bble[#1]#2#3{%
435 % #1 = commands to put after the list (optional)
436 % #2 = number to gobble first
437 % #3 = total number of items
438    \ifnum\numexpr#3>0
439       \ifnum\numexpr#3-(#2)=0
440          \ettl@supergobble@loop{#3+2}0{\ettl@supergobble@end{}{}}%
441       \else
442          \expandafter\ettl@supergobble@loop\expandafter{%
```

```
443            \number\numexpr\ifnum\numexpr#2*(#2-(#3))>0 #3+1\else#2+2\fi}{#3+2}%
444                            {\ettl@supergobble@next{}{#1}}%
445    \fi\fi}
446 \long\def\ettl@supergobble@loop#1#2#3{%
447    \ifcsname ettl@supergobble\number\numexpr#1\endcsname
448       \csname ettl@supergobble\number\numexpr#1\endcsname
449          {#3{#2-(#1)-1}}%
450    \else\ettl@supergobbleheight{\ettl@supergobble@loop{#1-8}{#2-8}{#3}}%
451    \fi}
452 \long\def\ettl@supergobble@end#1#2#3{\fi\fi\fi#1#2}
453 \long\csdef{ettl@supergobbleheight}#1\fi#2#3#4#5#6#7#8#9{\fi#1}
454 \long\csdef{ettl@supergobble7}#1#2\fi#3#4#5#6#7#8#9{#1}
455 \long\csdef{ettl@supergobble6}#1#2\fi#3#4#5#6#7#8{#1}
456 \long\csdef{ettl@supergobble5}#1#2\fi#3#4#5#6#7{#1}
457 \long\csdef{ettl@supergobble4}#1#2\fi#3#4#5#6{#1}
458 \long\csdef{ettl@supergobble3}#1#2\fi#3#4#5{#1}
459 \long\csdef{ettl@supergobble2}#1#2\fi#3#4{#1}
460 \long\csdef{ettl@supergobble1}#1#2\fi#3{#1}
461 \long\csdef{ettl@supergobble0}#1#2\fi{#1}
462 \long\def\ettl@supergobble@next#1#2#3#4{\fi
463    \ettl@supergobble@loop{#3}0{\ettl@supergobble@end{#4}{#2}}}
```

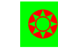
I•9 Define control sequence through groups

\AfterGroup
\AfterGroup*

\AfterGroup enhances the \aftergroup primitive: arbitrary code may be given to \AfterGroup. We use the \edef...\unexpanded trick already implemented in \ettl@ifnextchar to allow macro definitions (with arguments) inside the argument of \AfterGroup:

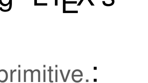
```
464 \newcount\ettl@fter
465 \newrobustcmd\AfterGroup{\@ifstar{\ettl@AfterGroup\@firstofone}{\ettl@AfterGroup\unexpa
466 \newrobustcmd\ettl@AfterGroup[2]{%
467    \csxdef{ettl@fterGroup\number\numexpr\the\ettl@fter+1}%
468       {\global\csundef{ettl@fterGroup\number\numexpr\the\ettl@fter+1}#1{#2}}%
469    \global\advance\ettl@fter\@ne
470    \expandafter\aftergroup\csname ettl@fterGroup\the\ettl@fter\endcsname}
```

\AfterAssignment

\AfterAssignment can be given arbitray code: 

```
471 \newrobustcmd\AfterAssignment{\@ifstar{\ettl@AfterAssignment\@firstofone}{\ettl@AfterAs
472 \newrobustcmd\ettl@AfterAssignment[2]{%
473    \csedef{ettl@afterassignment@hook\number\numexpr\the\ettl@fter}{#1{#2}}%
474    \global\advance\ettl@fter\@ne
475    \expandafter\afterassignment\csname ettl@afterassignment@hook\the\ettl@fter\endcsnam
```

\aftergroup@def

The macro is based on **letltxmacro** package. Therefore, \aftergroup@def works with commands with optional arguments and with the ones defined using LATEX's \DeclareRobustCommand. 

: we could have used the \AfterGroup macro but execution is lighter with 5 call to \aftergroup primitive.:

```
476 \newrobustcmd\aftergroup@def[1]{%
477    \let\etex@let@primitive\let    \def\let{\global\etex@let@primitive}%
478    \expandafter\LetLtxMacro\csname ettl@ftergroup@def\number\numexpr\the\ettl@fter+1\en
479    \global\advance\ettl@fter\@ne
480    \etex@let@primitive\let=\etex@let@primitive
481    \aftergroup\LetLtxMacro     \aftergroup#1%
482    \expandafter\aftergroup\csname ettl@ftergroup@def\the\ettl@fter\endcsname
483    \aftergroup\global          \aftergroup\undef
484    \expandafter\aftergroup\csname ettl@ftergroup@def\the\ettl@fter\endcsname}
485 \let\ettl@aftergroup@def\aftergroup@def
```

I•10 \futuredef

\@ifchar

\@ifchar works just like \@ifstar but uses the character-test.

```
486 \long\def\@ifchar#1#2{\ettl@ifnextchar #1{\@firstoftwo{#2}}}
```

\ettl@ifnextchar

\ettl@ifnextchar is based on the character-test rather than the \ifx-test. See the example for explanation on its behaviour.
\ettl@ifnextchar is used in the definition of \aftergroup@def and \@ifchar (of course...).
We take advantage of delimited definitions to exits from \if...\fi conditionnals (even in the case where the macro parameter may be \else, \if or \fi...):

```
487 \newrobustcmd\ettl@ifnextchar[3]{\begingroup
488    \edef\1##1/##2/##3{##1\endgroup\unexpanded{#2}##3}%
489    \edef\2##1/##2/##3{##1\endgroup\unexpanded{#3}##3}%
490    \ifOneToken{#1}
491       {\csname ettl@\if @\expandafter\ettl@cdr\detokenize{#1}\@nil @% OneChar
492          xifnch\else xifntk\fi\endcsname{#1}}
493       {\2//{}}}
```

\ettl@xifnch \ettl@ifnch

\ettl@xifnch is used in case the token to test (first parameter of \ettl@ifnextchar is a character token. It gobbles the possible spaces and exits at one if a begin-group or end-group character is found:

```
494 \long\def\ettl@xifnch#1{%
495    \ifx#1\@sptoken \def\ettl@xifnch{\ifx\@let@token\@sptoken\1\else\2\fi//{}}%
496    \else \def\ettl@xifnch{%
497       \ifx\@let@token\bgroup      \2
498       \else\ifx\@let@token\egroup \2
499       \else\ifx\@let@token\@sptoken \ettl@ifnspace\ettl@xifnch%
500       \else\ettl@ifnch%
501       \fi\fi\fi/{#1}/{}}%
502    \fi\futurelet\@let@token\ettl@xifnch}
```

\ettl@ifnch does the final comparison: the token is taken into the macro parameter to check if it is a single character (it was not possible to ensure this point for active characters that have been \let to something, unless by eating it in the parameter of a macro. It the test fails, the parameters is appended again to the input):

```
503 \long\def\ettl@ifnch#1/#2/#3{#1\long\def\ettl@ifnch##1{\ettl@char{##1}
504       {\if\string##1\string#2\1\else\2\fi}\2//{##1}}\ettl@ifnch}
```

\ettl@xifntk \ettl@ifntk

\ettl@xifntk is quite the same as \ettl@xifnch but for the case the token to test (i. e., the first parameter of \ettl@ifnextchar is a control sequence:

```
505 \long\def\ettl@xifntk#1{%
506    \ifx#1\bgroup\def\ettl@xifntk{\ifx\@let@token\bgroup\1\else\2\fi//{}}%
507    \else\ifx#1\egroup\def\ettl@xifntk{\ifx\@let@token\egroup\1\else\2\fi//{}}%
508    \else\def\ettl@xifntk{%
509       \ifx\@let@token\bgroup        \2
510       \else\ifx\@let@token\egroup   \2
511       \else\ifx\@let@token\@sptoken \ettl@ifnspace\ettl@xifntk%
512       \else\ettl@ifntk%
513       \fi\fi\fi/{#1}/{}}%
514    \fi\futurelet\@let@token\ettl@xifntk}
```

\ettl@ifntk finishes the job. We need to ensure that \@let@token is not an active character having been let to the token to test: there is no such thing as an active character for \ettl@ifnextchar!

```
515 \long\def\ettl@ifntk#1/#2/#3{#1\long\def\ettl@ifntk##1{\ettl@char{##1}
516    \2{\ifx##1#2\1\else\2\fi}//{##1}}\ettl@ifntk}
```

\ettl@ifnspace

\ettl@ifnspace is used to gobble a space and go back to the loop (this is very rare...):

```
517 \long\def\ettl@ifnspace#1#2/#3/#4 {#2\futurelet\@let@token#1}
```

\futuredef \futuredef* \futuredef= \futuredef*=

This is the scanner.

```
518 \newrobustcmd*\futuredef{\begingroup\ettl@futdef\ettl@futuredef\detokenize}
519 \protected\def\ettl@futdef#1#2{\@ifstar%
```

```
520      {\ettl@futdef\ettl@futured@f#2}
521      {\@ifchar={\ettl@futdef#1\unexpanded}
522                {\@testopt{\ettl@futur@def#1#2}{}}}}
523 \long\def\ettl@futur@def#1#2[#3]{%
524   \csname ettl@\if @\detokenize{#3}@1\else2\fi of2\endcsname
525   {\let \ettl@x \@empty \letcs \ettl@futur@def@collect{\@gobblescape#1@collectall}}%
526   {\def \ettl@x {#3}\edef \ettl@y {#2{#3}}%
527   \ifx\ettl@x\ettl@y      \let\ettl@y\@gobble
528   \else \ifx#2\unexpanded \let\ettl@y\@gobble
529   \else             \def\ettl@y{\edef\ettl@x}%
530   \fi\fi\ettl@y{\detokenizeChars{#3}}%
531   \letcs\ettl@futur@def@collect{\@gobblescape#1@collect}}%
532   \expandafter#1\expandafter#2\expandafter{\ettl@x}}
```

uredef (not starred) \ettl@futuredef defines the *test-macro* (which is entitled to break the loop) and the *loop-macro*:

```
533 \long\def\ettl@futuredef#1#2#3#4{% #1=detokenize #2=list, #3=macro result, #4=code-next
534   \def \ettl@futuredef@loop{\ettl@futuredef@test{}}%
535   \long \def \ettl@futuredef@test##1{%
536      \ifcat\noexpand\ettl@x\bgroup\ettl@futuredef@end{}\else
537      \ifcat\noexpand\ettl@x\egroup\ettl@futuredef@end{}\else
538      \ifcat\noexpand\ettl@x\ettl@sptoken\ettl@futuredef@space#1\else
539      \ettl@futur@def@collect#1\fi\fi\fi/Ne¤t/{#2}{##1}}%
540   \long \def \ettl@futuredef@end##1##2/Ne¤t/##3##4{##2\endgroup\def#3{##4}#4##1}%
541   \futurelet \ettl@x \ettl@futuredef@loop}
```

\ettl@futuredef@collect captures the next token (because it was found in the list) and selectively append it to the *result* (the argument of \ettl@futuredef@test). Then it loops:

```
542 \long\def\ettl@futuredef@collect#1#2/Ne¤t/#3#4#5{#2%
543   \ifcat\noexpand#5\relax \ettl@futuredef@filt\unexpanded
544   \else \ettl@futuredef@filt#1
545   \fi{#5}{#3}
546   {\def\ettl@futuredef@loop{\ettl@futuredef@test{#4#5}}\futurelet\ettl@x\ettl@futurede
547   {\ettl@futuredef@end{#5}/Ne¤t/{}{#4}}/Ne¤t/}
```

\ettl@futuredef@space gobbles the space token and append a space to the *result*. Then it loops:

```
548 \long\def\ettl@futuredef@space#1#2/Ne¤t/#3#4 {%
549   \ettl@futur@def@collect#1#2/Ne¤t/{#3}{#4}{ }}
```

\ettl@futuredef@collectall is used when no option (no ⟨*list of allowed tokens*⟩) has been given to \futuredef. In this case, \futuredef will stop only at the next begin-group or end-group token:

```
550 \long\def\ettl@futuredef@collectall#1#2/Ne¤t/#3#4#5{#2%
551   \def\ettl@futuredef@loop{\ettl@futuredef@test{#4#5}}\futurelet\ettl@x\ettl@futuredef
```

\ettl@futur@def@filt \ettl@futur@def@filt defines the *filter macro* to check if the token is in the ⟨*list of allowed tokens*⟩

```
552 \long\def\ettl@futur@def@filt#1#2{% #1=token to check, #2=allowed list
553   \long\def\ettl@futdef@filt##1#1##2##3/##4##5##6/Ne¤t/{##5}%
554   \ettl@futdef@filt#2#1//}
555 \long\def\ettl@futuredef@filt#1#2\fi#3#4{\fi % #1=detokenize/unexpanded, #2=discard, #3=
556   \expandafter\ettl@futur@def@filt\expandafter{#1{#3}}{#4}}
```

futured@f (starred) \ettl@futured@f defines the *test-macro* (which is entitled to break the loop) and the *loop-macro*:

```
557 \long\def\ettl@futured@f#1#2#3#4{% #1=detokenize #2=list, #3=macro result, #4=code-next
558   \let \ettl@y \@undefined
559   \def \ettl@futured@f@loop{\ettl@futured@f@test{}}%
560   \long \def \ettl@futured@f@test##1{%
561      \ifcat\noexpand\ettl@x\bgroup\ettl@futured@f@end\else
562      \ifcat\noexpand\ettl@x\egroup\ettl@futured@f@end\else
```

```
563       \ifcat\noexpand\ettl@x\ettl@sptoken\ettl@futured@f@space#1\else
564       \ettl@futur@def@collect#1\fi\fi\fi/Ne¤t/{##1}{#2}{}}%
565    \long \def \ettl@futured@f@end##1/Ne¤t/##2##3##4{##1\endgroup\def#3{##2}#4###4}%
566    \futurelet \ettl@x \ettl@futured@f@loop}
567 \long\def\ettl@futured@f@space#1#2/Ne¤t/#3#4#5 {%
568       \ettl@futur@def@collect#1#2/Ne¤t/{#3}{#4}{#5}{ }}
```

\ettl@futured@f@collect collects the next token which is appended to the argument of \ettl@futured@f@test (the *result*) if it is in the ⟨*list of allowed tokens*⟩, otherwise expansion is tried:

```
569 \long\def\ettl@futured@f@collect#1#2/Ne¤t/#3#4#5#6{#2%
570    \ifcat\noexpand\ettl@x\relax \ettl@futuredef@filt\unexpanded
571    \else \ettl@futuredef@filt#1
572    \fi{#6}{#4}
573    {\let \ettl@y \@undefined \ettl@futured@f@append/Ne¤t/{#3}{}{#6}}%
574    {\ettl@futured@f@try@expand{#3}\ettl@futured@f@end{#6}}/Ne¤t/}
```

\ettl@futured@f@collectall is used when \futuredef* is called with an empty optional argument:

```
575 \long\def\ettl@futured@f@collectall#1#2/Ne¤t/#3#4#5#6{#2%
576    \ettl@futured@f@try@expand{#3}\ettl@futured@f@append{#6}}
```

\ettl@futured@f@space is used in case the token is a space token:

```
577 \long\def\ettl@futured@f@space#1#2/Ne¤t/#3#4#5 {%
578       \ettl@futur@def@collect#1#2/Ne¤t/{#3}{#4}{#5}{ }}
```

\ettl@futured@f@try@expand checks if the token shall be expanded, or if the loop shall be broken (in case the ⟨*list of allowed tokens*⟩ is specified) or if this token shall be appended to the result (in case the ⟨*list of allowed token*⟩ is empty):

```
579 \long\def\ettl@futured@f@try@expand#1#2#3{%
580    \expandafter\ifx\noexpand\ettl@x\ettl@x
581       \let\ettl@y=#2%
582    \else\ettl@futured@f@CheckSpecials{#3}%
583       {\let \ettl@y=#2}%
584       {\ifx\ettl@x\ettl@y \let \ettl@y \ettl@futured@f@end\else
585        \let \ettl@y \ettl@futured@f@expand\fi}%
586    \fi\ettl@y/Ne¤t/{#1}{}{#3}}
```

\ettl@futured@f@expand expands the next token because it is not in the list and goes back to the loop:

```
587 \long\def\ettl@futured@f@expand/Ne¤t/#1#2#3{\let\ettl@y\ettl@x
588    \expandafter\futurelet\expandafter\ettl@x\expandafter\ettl@futured@f@loop#3}
```

\ettl@futured@f@CheckSpecials checks if the token is undefined or a \if... or \else etc. This is compulsory because we do not have to attempt expansion of such tokens (unless we want to get an error from TEX):

```
589 \long\def\ettl@futured@f@CheckSpecials#1{\ifintokslist{#1}{%
590    \@undefined\if\ifcat\ifnum\ifdim\ifodd%
591    \ifvmode\ifhmode\ifmmode\ifinner\ifvoid\ifhbox\ifvbox%
592    \ifx\ifeof\iftrue\iffalse\ifcase\ifdefined\ifcsname\iffontchar%
593    \else\fi\or}}
```

Finally, \ettl@futured@f@append appends the token to the result and goes back to the loop:

```
594 \def\ettl@futured@f@append/Ne¤t/#1#2#3{%
595    \def\ettl@futured@f@loop{\ettl@futured@f@test{#1#3}}%
596    \futurelet\ettl@x\ettl@futured@f@loop}%
```

I•11 Loops and Lists Management

I•11•1 – naturalloop

`\naturalloop`

This macro uses the capability of ε-TEX to build purely expandable loop using `\numexpr`:

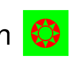
```
597 \newcommand\naturalloop[1]{\FE@testopt{#1}\ettl@naturalloop{\do}}
598 \def\ettl@naturalloop[#1]#2#3{%
599    \ifnum\numexpr#2>0 \expandafter\@swaparg\expandafter{\romannumeral-`\q#1[0]{#3}{#3}}%
600          {\ettl@naturall@@p[{#1}]{#2-1}{0}{#3}}
601 %            \ExpandNext{\ettl@naturall@@p[{#1}]{#2-1}{1}{#3}}{#1[1]{#3}{#3}}%
602    \else\@swap{\unexpanded{#3}}%
603    \fi}
604 \def\ettl@naturall@@p[#1]#2#3#4#5#6\fi{\fi%
605    \ifnum\numexpr#2>0 \expandafter\@swaparg\expandafter{\romannumeral-`\q%
606       \expandafter\@swap\expandafter{\expandafter[\number\numexpr#3+1]}{#1}{#4}{#5}}%
607       {\ettl@naturall@@p[{#1}]{#2-1}{#3+1}{#4}}%
608    \else\@swap{\unexpanded{#5}}%
609    \fi}
```

I•11•2 – Lists of single tokens

`\ifintokslist`

`\ifintokslist`⟨*token*⟩⟨*list of single tokens*⟩ breaks the loop at once when ⟨*token*⟩ is found in the list. The test for the end of the list is made by `\ettl@nbk`... of course:

`\ifincharlist`



`\ifincharlist`⟨*character or token*⟩⟨*list of single characters or tokens*⟩ is the same, with a different test macro: `\ettl@ifchar` is used instead of `\ettl@ifx`:

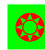
```
610 \newcommand\ifintokslist[2]{\romannumeral\csname rmn@%
611    \expandafter\ettl@nbk\romannumeral\ettl@dosinglelist{\ettl@ifintokslist{#1}}{#2}\z@/
612    {first}{second}//oftwo\endcsname}
613 \long\def\ettl@ifintokslist#1#2{\ifx#1#2\ettl@breakloop\z@\fi}
614 \newcommand\ifincharlist[2]{\romannumeral\csname rmn@%
615    \expandafter\ettl@nbk\romannumeral\ettl@dosinglelist{\ettl@ifincharlist{#1}}{#2}\z@/
616    {first}{second}//oftwo\endcsname}
617 \long\def\ettl@ifincharlist#1#2{\ettl@ifchar{#1}{#2}{\ettl@breakloop\z@}{}}
```

`\ettl@dosinglelist`

We define a very simple loop for single tokens (for internal use): it is the same as `\toksloop` but avoids overhead due to the parsing of modifiers:

```
618 \long\def\ettl@dosinglelist#1#2{\ettl@nbk#2//%
619       {\ettl@dosinglelist@loop{#1}#2//{\ettl@dosinglelist@loop{#1}}{\ettl@breakloop{}}}
620       {\ettl@breakloop{}}///End§List/}
621 \long\def\ettl@dosinglelist@loop#1#2#3#4/#5#6#7/End§List/{%
622       #1{#2}#6{#3}#4//{#6}{#7}/End§List/}
```

`\gettokslistindex`
`\getcharlistindex`
`\gettokslistcount`
`\getcharlistcount`
`\gettokslisttoken`
`\getcharlisttoken`



`\gettokslistindex`⟨*item*⟩⟨*tokenlist-macro*⟩

`\gettokslistindex` is always purely expandable (`\ifx` test).

The following three macros are the *entry points*. `\ExpandAftercmds` is applied to `\ettl@getsinglelist` which initiates the loop: we ask for total expansion. After expansion, `\ettl@setresult` will extract the wanted register by projection: The result comes from in the first register for `count`, the second for `index` and the third for `token`, therefore, we use the `\ettl@XofY` macros:

```
623 \newcommand\gettokslistindex[2]{\number\ifnotempty{#2}{\ettl@nbk#1//%
624    {\ExpandAftercmds{\ettl@setresult 2of3><}{\ettl@getsinglelist{\ettl@ifx{#1}}{#2}}}
625    {-1}//}{-1}}
626 \newcommand\getcharlistindex[2]{\number\ifnotempty{#2}{\ettl@nbk#1//%
627    {\ExpandAftercmds{\ettl@setresult 2of3><}{\ettl@getsinglelist{\ettl@ifchar{#1}}{#2}}
628    {-1}//}{-1}}
629 \newcommand\gettokslistcount[1]{\number\ifnotempty{#1}%
630    {\ExpandAftercmds{\ettl@setresult 1of3><}{\ettl@getsinglelist{\ettl@ifx{\\}}{#1}}}
631    0}
```

```
632 \newcommand\getcharlistcount[1]{}%
633 \let\getcharlistcount=\gettokslistcount
634 \newcommand\gettokslisttoken[2]{\ifnotempty{#2}{\ettl@nbk#1//%
635    {\ExpandAftercmds{\ettl@setresult 3of3><}{\ettl@getsinglelist{\ettl@ifx{#1}}{#2}}}
636    {}//}{}}
637 \newcommand\getcharlisttoken[2]{\ifnotempty{#2}{\ettl@nbk#1//%
638    {\ExpandAftercmds{\ettl@setresult 3of3><}{\ettl@getsinglelist{\ettl@ifchar{#1}}{#2}}}
639    {}//}{}}
```

\ettl@getsinglelist initiates the loop (we test if the list or the ⟨*item*⟩ is empty first):

```
640 \long\def\ettl@getsinglelist#1#2{\ettl@singlelist@loop{-1}{-1}{}#2//%
641          {\ettl@expandafthree\ettl@singlelist@loop#1}%
642          {\expandafter\ettl@singlelist@result\@thirdofthree}/End§List/}
```

\ettl@singlelist@loop tests each token and update registers:

```
643 \long\def\ettl@singlelist@loop#1#2#3#4#5/#6#7#8/End§List/{%
644       #7{#4}
645          {{#1+1}{#2+1+0*(0}{#4}}
646          {{#1+1}{#2+1}{#3}}#5//{#7}{#8}/End§List/}
647    %   \csname @#1#5{first}{second}oftwo\endcsname
648    %      {#8{#1}{#2+1}{#3+1+0*(0}{#5}#6//#8#9}
649    %      {#8{#1}{#2+1}{#3+1}{#4}#6//#8#9}/End§List/}
```

Well! **#1** is the *test-macro* to test against **#5**, the current token of the list.

#2 is the current index. It is incremented by 1 and will be equal to the length of the list, at the end. **#3** is the index of the ⟨*item*⟩ (if found): it is incremented by 1 but at the time ⟨*item*⟩ is found is the list, the next increments are canceled (multiplication by 0).

The fourth parameter remains the same (**#4**=**#4**=empty, set at the initiation of the loop) but at the time ⟨*item*⟩ is found, **#4** becomes this ⟨*item*⟩ (precisely the matching item found in the list: **#5**).

#6 is the remainder of the list. **#7**, **#8** and **#9** are the usual parameter for *blank-test* (see \ettl@nbk).

\ettl@tokslist@result extracts the count, the index and the token from the parameters of the *test-macro*:

```
650 \def\ettl@singlelist@result#1#2#3#4/End§List/{\ExpandNextTwo\@swaptwo%
651       {\number\numexpr\ifempty{#3}{-1}{#2)}}{\number\numexpr#1}{#3}}
```

Then \ettl@setresult finishes the job:

```
652 \def\ettl@setresult#1of#2>#3<{\ettl@nbk #3//%
653       {\ifdefcount{#3}{#3=\csname ettl@#1of#2\endcsname}
654                   {\edef#3{\csname ettl@#1of#2\endcsname}}}%
655       {\csname ettl@#1of#2\endcsname}//}
```

I•11•3 – General Lists and Loops Constructor

\DeclareCmdListParser

\DeclareCmdListParser acts in the same way as **etoolbox**-\DeclareListParser and the command-list-parser are sensitive to the category code of the ⟨*separator*⟩

The command-list-parser will be defined only if it is definable:

```
656 \newrobustcmd\DeclareCmdListParser[3][\global]{\@ifdefinable{#2}{\begingroup
657       \protected\def\ettl@defcmdparser##1{%
658          \edef\ettl@defcmdparser{\endgroup\ettl@defcmdparser
659             {#1}{\noexpand#2}{\unexpanded{#3}}
660             {\noexpandcs{##1->start}}
661             {\noexpandcs{##1->loop}}
662             {\noexpandcs{##1->loop+}}
663             {\noexpandcs{for##1}}%
664          }\ettl@defcmdparser
665       }\expandafter\ettl@defcmdparser\expandafter{\romannumeral-`\q\@gobblescape#2}}}
```

\ettl@defcmdparser doeas the definitions: \parser->start initiates the loop (and add a separator at the end of the list) and \parser->loop loops into the list, expanding the (optional, default \do) user code for each item.

In case the '+' form is used, the auxiliary macro \ettl@doitemidx overloads the user-code. Otherwise (simple form without index): \ettl@doitem overloads the user-code.

```
666 \protected\long\def\ettl@defcmdparser#1#2#3#4#5#6#7{%#1=global,#2=command,#3=sep,#4=sta
667    #1\long\def#4##1##2[##3]##4{% ##1=case, ##2=expandafter??? , ##3=do, ##4=list
668       ##2{##4}% ifiscs or @thirdofthree
669          {\expandafter\@swaparg\expandafter{##4}{#4{##1}\@thirdofthree[{##3}]}}
670          {\ettl@nbk##4//%
671             {\ifcase##1 \ettl@or\@swaplast{\number\numexpr#6{\ettl@lst@count}}#6%
672                \or      \ettl@or\@swaplast{#6{\ettl@lst@getitem{##3}}}#6%
673                \or      \ettl@or\@swaplast{#5{##3}}#5%
674                \or      \ettl@fi\@swaplast{#6{##3}}#6%
675                \fi{##4#3//}{\ettl@breakloop{}}%
676             }{\ettl@breakloop{}}///End§List/}}%
677    #1\long\def#5##1##2#3##3##4/##5##6##7/End§List/{%
678       \if @\detokenize{##2}@\expandafter\@gobbletwo\fi\@firstofone{##1{##2}}%
679       ##6{##1}##3##4//{##6}{##7}/End§List/}
680    #1\long\def#6##1##2##3#3##4##5/##6##7##8/End§List/{%
681       \if @\detokenize{##3}@\expandafter\@gobbletwo\fi\@firstofone{##2[##1]{##3}}%
682       \expandafter##7\expandafter{\number\numexpr##1+1}{##2}##4##5//{##7}{##8}/End§List
683    #1\protected\def#7{\@ifchar*%
684       {\@ifchar+{\ettl@forloop{\expandafter#2\expandafter*\expandafter+}{[####1]####2}}
685                 {\ettl@forloop{\expandafter#2\expandafter*\expandafter+}{####1}}}
686       {\@ifchar+{\@ifchar*%
687                    {\ettl@forloop{\expandafter#2\expandafter*\expandafter+}{[####1]##
688                    {\ettl@forloop{\expandafter#2\expandafter+}{[####1]####2}}}
689                 {\ettl@forloop{\expandafter#2}{####1}}}}
690    #1\def#2{\ettl@lst@modif#423\ifiscs}}
```

\ettl@lst@doitem gives the current item to the auxiliary macro, while \ettl@lst@doitemidx gives the index as well. \ettl@lts@getitem is the helper macro in case we ask for an item (cf. \csvloop[4]\mylist) and \etttl@lst@count is as basic as it can be!

```
691 \long\def\ettl@lst@getitem#1[#2]#3{%
692    \ifnum\numexpr#1<0 \@swap{\breakloop{}}\fi
693    \ifnum\numexpr#1=#2 \@swap{\breakloop{#3}}\fi}
694 \long\def\ettl@lst@count[#1]#2{+\ettl@nbk#2//10//}
```

\ettl@lst@modif

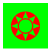
\ettl@lst@modif is used by any command-list-parser at the beginning to set the options. This macro is interesting because it is recursive: each allowed modifier is parsed one after the other in a purely expandable way, setting the registers (**#1** to **#4**) to the value corresponding to the modifier used (the registers are initialized to their default value).

Such a code is interesting because it may be used elsewhere: the aim is to parse modifiers without taking care of their order (\csvloop*+ is the same as \csvloop+*):

```
695 \long\def\ettl@lst@modif#1#2#3#4#5{\FE@modifiers{*+![}{#5}%
696    {\ettl@lst@modif{#1}#2#3\@thirdofthree}%     * case
697    {\ettl@lst@modif{#1}#3#2{#4}}%  + (case 3/default 2)
698    {\ettl@lst@modif{#1}00{#4}}% ! (case 0)
699    {\ettl@lst@opt{#1}{#2}{#4}#5}% [   (option)
700    {\ettl@lst@opt{#1}{#2}{#4}[\do]}}% (default option)
701 \long\def\ettl@lst@opt#1#2#3[#4]{%
702    \expandafter#1\expandafter{\number\ifnum#2=0 0\else\ifstrnum{#4}{1}{#2}\fi}{#3}[{#4}]
```

\breakloop

\breakloop gobbles anything until the '/EndList/' delimiter:



```
703 \long\def\ettl@breakloop#1#2/End§List/{#1}
704 \let\breakloop\ettl@breakloop
```

forloops

In order to define for \for...loop macros, and to handle the case they are nested, we need a counter.

```
705 \globcount\ettl@for@nested
706 \long\def\ettl@forloop#1#2#3\do{%
707    \global\advance\ettl@for@nested\@ne\relax
708    \csdef{ettl@for@loop\the\ettl@for@nested}{%
709       #1\expandafter[\csname ettl@for@do\the\ettl@for@nested\endcsname]{#3}%
710       \csundef{ettl@for@do\the\ettl@for@nested}%
711       \csundef{ettl@for@loop\the\ettl@for@nested}%
712       \global\advance\ettl@for@nested\m@ne\relax}
713    \expandafter\afterassignment\csname ettl@for@loop\the\ettl@for@nested\endcsname
714    \long\csdef{ettl@for@do\the\ettl@for@nested}#2}
```

`\csvloop` Definition of `\csvloop`: `\forcsvloop` is also defined by `\DeclareCmdListParser` but is not purely expandable:

`\forcsvloop`

```
715 \DeclareCmdListParser\csvloop{,}
```

`\listloop` `\forlistloop` Definition of `\listloop` (with a '|' of catcode 3 (math shift) – cf. **etoolbox**). `\forlistloop` is defined by `\DeclareCmdListParser` but is not purely expandable:

```
716 \begingroup\catcode`\|=3
717 \DeclareCmdListParser\listloop{|}% global declaration
718 \endgroup
```

`\toksloop` `\fortoksloop` Definition of `\toksloop` (with no delimiter). `\fortoksloop` is defiined by `\DeclareCmdListParser` but is not purely expandable:

```
719 \DeclareCmdListParser\toksloop{}
```

`\csvlistadd` `\csvlistgadd` `\csvlisteadd` `\csvlistxadd`

```
720 \providerobustcmd\csvlistadd[2]{\ettl@nbk#2//{\appto#1{#2,}}{}//}
721 \providerobustcmd\csvlistgadd[2]{\ettl@nbk#2//{\gappto#2{#2,}}{}//}
722 \providerobustcmd\csvlisteadd[2]{\begingroup \protected@edef#1{#2}%
723    \expandafter\ettl@nbk#1//{\expandafter\endgroup
724       \expandafter\appto\expandafter#1\expandafter{#1,}}\endgroup//}
725 \providerobustcmd\csvlistxadd[2]{\begingroup \protected@edef#1{#2}%
726    \expandafter\ettl@nbk#1//{\expandafter\endgroup
727       \expandafter\gappto\expandafter#1\expandafter{#1,}}\endgroup//}
```

`\csvtolist` This is the first application of `\csvloop`:

```
728 \newcommand\csvtolist[1]{\FE@ifstar{#1}{\ettl@convertlist{{\csvloop*}\ettl@do@csvtolist
729                                  {\ettl@convertlist{\csvloop\ettl@do@csvtolist}}
730 \long\def\ettl@convertlist#1#2{\FE@testopt{#2}{\ettl@convert@list#1}{}}
731 \long\def\ettl@convert@list#1#2[#3]#4{\ettl@nbk#3//%
732       {\edef#3{#1[#2]{#4}}}
733       {#1[#2]{#4}}//}
734 \begingroup\catcode`\|=3% etb catcode
735 \long\gdef\ettl@do@csvtolist#1{\unexpanded{#1}|}
736 \endgroup
```

`\listtocsv` This is the first application of `\listloop`:

```
737 \newcommand\listtocsv[1]{\FE@ifstar{#1}{\ettl@convertlist{{\listloop*}\ettl@do@listtocs
738                                  {\ettl@convertlist{\listloop\ettl@do@listtocsv}}
739 \long\def\ettl@do@listtocsv#1{\unexpanded{#1,}}
```

`\tokstolist` This is the first application of `\toksloop`:

```
740 \newcommand\tokstolist[1]{\FE@ifstar{#1}{\ettl@convertlist{{\toksloop*}\ettl@do@tokstol
741                                  {\ettl@convertlist{\toksloop\ettl@do@tokstolist
742 \begingroup\catcode`\|=3% etb catcode
743 \long\gdef\ettl@do@tokstolist#1{\unexpanded{#1}|}
744 \endgroup
```

\csvtolistadd

\csvtolistadd is not purely expandable:

```
745 \newrobustcmd*\csvtolistadd{\@ifstar{\ettl@csvtolistadd*}{\ettl@csvtolistadd{}}}
746 \long\def\ettl@csvtolistadd#1#2#3{\eappto#2{\csvtolist#1[]{#3}}}
```

\tokstolistadd

\tokstolistadd is not purely expandable:

```
747 \newrobustcmd*\tokstolistadd{\@ifstar{\ettl@tokstolistadd*}{\ettl@tokstolistadd{}}}
748 \long\def\ettl@tokstolistadd#1#2#3{\eappto#2{\tokstolist#1[]{#3}}}
```

\ettl@RemoveInList

This is the general constructor for deletion into lists with any separator:

```
749 \newrobustcmd\ettl@RemoveInList[2]{\begingroup
750 % #1 = \global  #2 = macro name
751    \def\ettl@RemoveInList##1{%
752       \edef\ettl@RemoveInList####1####2{%
753          \ettl@Rem@veInList{####1}####2\noexpandcs{##1->remove}\noexpandcs{##1->result}
754       }\ettl@RemoveInList{#1}#2%
755    }\expandafter\ettl@RemoveInList\expandafter{\romannumeral-`\q\@gobblescape#2}}
756 \protected\long\def\ettl@Rem@veInList#1#2#3#4#5#6#7#8{%
757    \long\def#3[##1]##2#5#8#5##3##4/##5##6##7/End§List/{##6[##1+1]##2#5##3##4//##6##7/En
758    \ifnotempty{#5}%% special case if no separator
759       {\long\def#4[##1]#5##2#5#5##3//##4/End§List/{\unexpanded{#1\def#7{##2#5}}%
760                \ettl@nbk#6//\ettl@setresult 1of1>#6<{\number\numexpr##1-1\relax}{}//}}%
761       {\long\def#4[##1]##2//##3/End§List/{\unexpanded{#1\def#7{##2}}%
762                \ettl@nbk#6//\ettl@setresult 1of1>#6<{\number\numexpr##1-1\relax}{}//}}%
763    \long\def#2##1{#3[0]#5##1#5#5#8#5//#3#4/End§List/}%
764    \edef#7{\endgroup\expandafter#2\expandafter{#7}}#7}
765 \def\ettl@gobble@relax#1\relax{}
```

\listdel
\listedel
\listgdel
\listxdel

\listdel removes an ⟨item⟩ from a list, \listedel expands the ⟨item⟩ (with \protected@e) first, \listgdel make the assignment to the (shorter-)list global and \listxdel both expands the ⟨item⟩ and makes the assignment global:

```
766 \begingroup\catcode`\|=3
767 \newrobustcmd\listdel[1][]{\ettl@RemoveInList{}\listdel|{#1}}
768 \newrobustcmd\listgdel[1][]{\ettl@RemoveInList\global\listdel|{#1}}
769 \newrobustcmd\listedel[1][]{\ettl@listedel{}\listdel|{#1}}
770 \newrobustcmd\listxdel[1][]{\ettl@listedel\global\listdel|{#1}}
771 \aftergroup@def\listdel
772 \aftergroup@def\listgdel
773 \aftergroup@def\listedel
774 \aftergroup@def\listxdel
775 \endgroup% \catcode group
776 \newrobustcmd\ettl@listedel[6]{\begingroup\protected@edef#5{#6}\expandafter\endgroup
777    \expandafter\@swaparg\expandafter{#5}{\ettl@RemoveInList#1#2{#3}{#4}#5}}
```

\csvdel
\csvedel
\csvgdel
\csvxdel

\csvdel removes an ⟨item⟩ from a list, \csvedel expands the ⟨item⟩ (with \protected@ede first, \csvgdel make the assignment to the (shorter-)list global and \csvxdel both expands the ⟨item⟩ and makes the assignment global:

```
778 \newrobustcmd\csvdel[1][]{\ettl@RemoveInList{}\csvdel,{#1}}
779 \newrobustcmd\csvgdel[1][]{\ettl@RemoveInList\global\csvdel,{#1}}
780 \newrobustcmd\csvedel[1][]{\ettl@listedel{}\csvdel,{#1}}
781 \newrobustcmd\csvxdel[1][]{\ettl@listedel\global\csvdel,{#1}}
```

\toksdel
\toksedel
\toksgdel
\toksxdel

\toksdel removes an ⟨item⟩ from a list, \toksedel expands the ⟨item⟩ (with \protected@e) first, \toksgdel make the assignment to the (shorter-)list global and \toksxdel both expands the ⟨item⟩ and makes the assignment global:

```
782 \newrobustcmd\toksdel[1][]{\ettl@RemoveInList{}\toksdel{}{#1}}
783 \newrobustcmd\toksgdel[1][]{\ettl@RemoveInList\global\toksdel{}{#1}}
784 \newrobustcmd\toksedel[1][]{\ettl@listedel{}\toksdel{}{#1}}
785 \newrobustcmd\toksxdel[1][]{\ettl@listedel\global\toksdel{}{#1}}
```

\getlistindex

\getlistindex may be defined, with its star form (no expansion of the list) and normal form (⟨*Listmacro*⟩ expanded once); The search-index is initialised at 1:

We first need to get into a group where delimiter ' | ' and '&' have catcode 3:

```
786 \newrobustcmd\ettl@getlistindex[6][]{% #1=result, #2=\expandafter, #3=loop macro, #4=sep
787    \begingroup\def\ettl@getlistindex##1#4#6#4##2/End§List/{\endgroup
788       \romannumeral-`\q\ettl@setresult 1of1>#1<{\ettl@nbk##2//{#3*!{##1}}{-1}//}%
789    }#2\ettl@getlistindex#2#5#4#6#4/End§List/}
790 \begingroup\catcode`\|=3% etb catcode
791 \newrobustcmd\getlistindex[3][]{\@ifstar%
792    {\ettl@getlistindex{}\listloop{|}{#1}{#2}{#3}}
793    {\ifiscs{#1}{\ettl@getlistindex\expandafter\listloop|{#1}{#2}{#3}}
794                {\ettl@getlistindex{}\listloop|{#1}{#2}{#3}}}}
795 \aftergroup@def\getlistindex
796 \endgroup%\catcode group
```

\getcsvlistindex

The command is robust, not purely expandable:

```
797 \newrobustcmd\getcsvlistindex[3][]{\@ifstar%
798    {\ettl@getlistindex{}\csvloop{,}{#1}{#2}}
799    {\ifiscs{#1}{\ettl@getlistindex\expandafter\csvloop,{#1}{#2}}
800                {\ettl@getlistindex{}\csvloop,{#1}{#2}}}}
```

\ettl@ifinlist

\ettl@ifinlist will build a \ifinlist macro for list with a given separator.

```
801 \def\ettl@if@inlist#1#2{%#1=macro,#2=separator
802 \newrobustcmd*#1{\@ifstar{\ettl@ifinlist{#2}{}}{\ettl@ifinlist{#2}\expandafter}}}
803 \def\ettl@xif@inlist#1#2{%
804 \newrobustcmd*#1{\@ifstar{\ettl@xifinlist{#2}{}}{\ettl@xifinlist{#2}\expandafter}}}
805 \protected\long\def\ettl@ifinlist#1#2#3#4{\begingroup
806    \def\ettl@tempa##1#1##2#1/End§List/{\endgroup\ifnotblank{##2}%
807    }#2\ettl@tempa#2#1#3#1#4#1/End§List/}
808 \protected\long\def\ettl@xifinlist#1#2#3#4{\begingroup
809    \protected@edef\ettl@tempa{\endgroup\ettl@ifinlist{#1}{#2}{#3}{#4}%
810    }\ettl@tempa}
```

\ifincsvlist

A robust command with a star form.

\xifincsvlist

The same with \protected@edef.

```
811 \ettl@if@inlist\ifincsvlist{,}
812 \ettl@xif@inlist\xifincsvlist{,}
813 \undef\ettl@if@inlist
814 \undef\ettl@xif@inlist
```

\interval

\interval will expand to the number of the interval of ⟨*number*⟩ into the ⟨*sorted comma separated*

```
815 \newcommand\interval[2]{\romannumeral-`\q%
816    \ExpandNext{\avoidvoid[\csvloop!{#2}]}{\csvloop+[\ettl@do@interval{#1}][#2}}}
817 \def\ettl@do@interval#1[#2]#3{\ifdim#1\p@<#3\p@ \@swap{\breakloop{#2}}\fi}
```

\locinterplin

```
818 \newcommand\locinterplin[3]{\romannumeral-`\q
819    \unless\ifnum\numexpr(\csvloop!{#2})-(\csvloop!{#3})=0
820       \PackageError{etextools}{Using \string\locinterplin\space the lists in argument 1
821       must have the same number of elements}
822       {You're in trouble here and I cannot proceed...}
823    \fi
824    \ExpandNextTwo{\ettl@locinterplin{#1}{#3}{#2}}{\interval{#1}{#2}}{\csvloop!{#2}}}
825 \begingroup\catcode`\/ 12%
826 \gdef\ettl@locinterplin#1#2#3#4#5{%
827    \ifnum#4=0 \csvloop[#4]{#2}%
828    \else\ifnum#4=#5 \expandafter\csvloop\expandafter[\number\numexpr#5-1]{#2}%
829    \else\ifdim#1\p@=\expandafter\csvloop\expandafter[\number\numexpr#4-1]{#3}\p@
```

```
830       \expandafter\csvloop\expandafter[\number\numexpr#4-1]{#2}%
831    \else\strip@pt\dimexpr%
832       \expandafter\csvloop\expandafter[\number\numexpr#4-1]{#2}\p@+%
833          (#1\p@-\expandafter\csvloop\expandafter[\number\numexpr#4-1]{#3}\p@)*%
834          (\expandafter\csvloop\expandafter[\number\numexpr#4-1]{#2}-\csvloop[#4]{#2})/%
835          (\expandafter\csvloop\expandafter[\number\numexpr#4-1]{#3}-\csvloop[#4]{#3})\re
836    \fi\fi\fi}
837 \endgroup% catcode group
```

etextools package options (undocumented - not tested, not to be used)

Undocumented option **etoolbox**.

```
838 \DeclareOption{etoolbox}{%
839 \renewcommand\ifblank[3]{\ettl@nbk #1//{#2}{#3}//}
840 \renewcommand\ifdef[1]{\csname @\ifdefined#1first\else second\fi oftwo\endcsname}
841 \renewcommand\ifcsdef[1]{\csname @\ifcsname#1\endcsname first\else second\fi oftwo\endcs
842 \renewcommand\ifundef[1]{\csname @%
843    \ifdefined#1\ifx#1\relax first\else second\fi\else first\fi oftwo\endcsname}
844 \renewcommand\ifcsundef[1]{\csname @%
845    \ifcsname#1\endcsname\expandafter\ifx\csname#1\endcsname\relax
846       first\else second\fi\else first\fi oftwo\endcsname}
847 \edef\ifdefmacro#1{\unexpanded{\csname @%
848      \expandafter\ettl@ifdefmacro\meaning}#1\detokenize{macro:}/oftwo\endcsname}
849 \edef\ettl@ifdefmacro{%
850    \def\noexpand\ettl@ifdefmacro##1\detokenize{macro:}##2/{\noexpand\ettl@nbk##2//{firs
851 }\ettl@ifdefmacro
852 \long\edef\ifcsmacro#1{\unexpanded{\csname @%
853    \expandafter\expandafter\expandafter\ettl@ifdefmacro\meaningcs}{#1}\detokenize{macro
854 \renewcommand\ifdefparam[1]{\csname @%
855    \ettl@expandaftwo\ettl@nbk\expandafter\ettl@params@meaning\meaning#1///{first}{secon
856 \renewcommand\ifcsparam[1]{\csname @%
857    \expandafter\expandafter\expandafter\ettl@nbk\parameters@meaningcs{#1}//{first}{secon
858 \renewcommand\ifnumcomp[3]{\csname @%
859    \ifnum\numexpr#1#2\numexpr#3 first\else second\fi oftwo\endcsname}
860 }% etoolbox option
861 \ProcessOptions

862 ⟨/package⟩
```

Revision history

3.14 2009-10-04

Stabilisation of some commands. the package sould now be OK.

3.0 2009-09-09

Definition of `\DeclareStringFilter`, `\FE@modifiers` and `\ettl@supergobble`

2k 2009-09-04

Addition of
- `\ExpandNext`
- `\naturalloop`
- the star form of `\futuredef`
- the `\global` option of `\DeclareCmdListParser`

Reimplementation of
- the lists macros for optimisation (cf `\ettl@ifnotblank`)
- `\ifsinglechar` for optimisation

Addition of examples to the `etextools-examples.tex`

Test on pdfLATEX and XeTEX.

2i 2009-08-31

Addition of `\futuredef` a macro (and vectorized) version of `\futurelet`.

Redesign of `\expandnext`: the first argument can now be arbitrary code (before, it was necessarily a single control sequence, as for `\expandafter`).

Redesign of \deblank, after a solution provided by **environ.sty**.

Addition of \ifincsvlist, \ifintokslist and \xifincsvlist.

Addition of \forcsvloop, \forlistloop and \fortoksloop.

Addition of \csvdel, \csvedel, \csvgdel and \csvxdel

Optimization of \getlistindex and \getcsvlistindex

2t 2009-08-15

Addition of \ifnotempty, \ifstrcmp, \ifstrmatch

2h 2009-08-14

\getlistindex is now fully expandable

Addition of
\toksloop

Addition of
\FE@ifchar as a generalization of \FE@ifstar.

2z 2009-08-12

Addition of
\ifempty, \toksloop, \tokstolist and \tokstolistadd

Modification of \ifsinglechar
\ifsinglechar now works with \ifempty so that:

\macro{ * } is no more considered as a starred form
because of the spaces following the *
however, the spaces **before** are skipped,
as does \@ifnextchar from the LATEX kernel.

Index added to this documentation paper.

2e 2009-07-14

First version (include an example file)

References

[1] David Carlisle and Peter Breitenlohner *The* **etex** *package*; 1998/03/26 v2.0; CTAN:macros/latex/contrib/etex-pkg/.

[2] Philipp Lehman *The* **etoolbox** *package*; 2008/06/28 v1.7; CTAN:macros/latex/contrib/etoolbox/.

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.

Symbols

\# . . . 272, 274
\/ . . . 825
\@car . . . 49
\@cdr . . . 48
\@expandnext . . . 103, 105, 107
\@firstofone . 43, 357, 465, 471, 678, 681
\@firstoftwo . . . 44, 361, 486
\@gobblescape . . . 87, 87, 258, 308, 310, 525, 531, 665, 755
\@gobblespace . . . 86, 86
\@gobbletwo . . . 678, 681
\@ifchar . . . 486, 486, 521, 683, 684, 686
\@ifdefinable . . . 306, 656
\@intmax . . . 60
\@let@token . . . 495, 497–499, 502, 506, 507, 509–511, 514, 517
\@secondoftwo . . . 45
\@sptoken . . . 495, 499, 511
\@swap . . . 88, 88, 89, 96, 100, 109, 243, 602, 606, 608, 692, 693, 817
\@swaparg . . . 90, 90, 104, 108, 112, 599, 605, 669, 777
\@swaplast . . . 91, 91, 671–674
\@swaptwo . . . 92, 92, 650
\@thirdofthree . . . 642, 669, 696
\@undefined . . . 121, 558, 573, 590
\@xifstrequal . . . 269, 270

Numbers

\1 . . . 488, 495, 504, 506, 507, 516
\2 . . . 489, 493, 495, 497, 498, 504, 506, 507, 509, 510, 516

A

\AfterAssignment . . . 471
\AfterGroup . . . 464
\aftergroup . . . 470, 481–484
\AfterGroup* . . . 464
\aftergroup@def . . . 476, 476, 485, 771–774, 795
\appto . . . 720, 724
\avoidvoid . . . 161, 816
\avoidvoid* . . . 161
\avoidvoidcs . . . 171
\avoidvoidcs* . . . 171

B

\basic␣string␣filter . . . 402
\bgroup . . . 497, 506, 509, 536, 561
\breakloop . . . 692, 693, 703, 817

C

\catcode . . . 11, 13, 716, 734, 742, 766, 775, 790, 796, 825
\char . . . 155
\csvdel . . . 778
\csvedel . . . 778
\csvgdel . . . 778
\csvlistadd . . . 720
\csvlisteadd . . . 720
\csvlistgadd . . . 720
\csvlistxadd . . . 720
\csvloop . . . 715, 728, 729, 798–800, 816, 819, 824, 827–830, 832–835
\csvtolist . . . 728, 746
\csvtolistadd . . . 745
\csvxdel . . . 778
\csxdef . . . 467

D

\deblank . . . 255, 298
\DeclareCmdListParser 656, 715, 717, 719
\DeclareOption . . . 838
\DeclareStringFilter . . . 306
\detokenize . . . 55, 58, 76, 136, 140, 142, 157–159, 183, 184, 191, 192, 195, 196, 203, 207, 208, 213, 218, 225, 228, 230, 233, 234, 240, 244, 246, 259, 261, 281, 282, 287, 291, 295, 308, 404, 409, 414, 419, 422, 491, 518, 524, 678, 681, 848, 850, 853
\detokenizeChars . . . 238, 530
\dimen . . . 152
\dimexpr . . . 831
\docsvlist . . . 285

E

\eappto . . . 746, 748
\egroup . . . 498, 507, 510, 537, 562
\etex@let@primitive . . . 477, 480
\ettl@@voidvoidcs . . . 174, 175
\ettl@AfterAssignment . . . 471, 472
\ettl@AfterGroup . . . 465, 466
\ettl@aftergroup@def . . . 485
\ettl@AtEnd . . . 7, 9, 10, 35
\ettl@avoidvoidcs . . . 172–174
\ettl@B@sicFilter . . . 402, 403
\ettl@BasicFilter . . . 402
\ettl@breakloop . . . 613, 617, 619, 620, 675, 676, 703, 704
\ettl@car . . . 49, 224
\ettl@carcar . . . 54, 186, 198
\ettl@cdr . . . 48, 58, 184, 192, 196, 208, 218, 233, 234, 491
\ettl@char . . . 58, 503, 515
\ettl@convert@list . . . 730, 731
\ettl@convertlist . . . 728–730, 737, 738, 740, 741
\ettl@csname . . . 57, 179, 180, 225, 228, 233, 236
\ettl@csvtolistadd . . . 745, 746
\ettl@deblank . . . 255, 256, 259
\ettl@deblank@i . . . 256, 257
\ettl@declarestrfilter . . . 307, 309
\ettl@defcmdparser . . 657, 658, 664–666
\ettl@do@csvtolist . . . 728, 729, 735
\ettl@do@detokenChars . . . 239, 240
\ettl@do@interval . . . 816, 817

\ettl@do@listtocsv 737–739
\ettl@do@tokstolist 740, 741, 743
\ettl@dosinglelist 238, 611, 615, 618
\ettl@dosinglelist@loop 619, 621
\ettl@else 38, 79, 81, 83, 85, 356, 360
\ettl@expandafthree 41, 138, 258, 641
\ettl@expandaftwo 40, 855
\ettl@FE@chardot 421, 422
\ettl@FE@charequal 416, 419
\ettl@FE@ifstar 411, 414
\ettl@FE@modifiers 427, 431
\ettl@FE@testopt 406, 409
\ettl@fi 37, 79, 81, 83, 85, 342, 380, 674
\ettl@firstsp@ce 55, 56
\ettl@firstspace 55, 182, 190, 194, 206, 226, 229, 235
\ettl@for@nested 705, 707–714
\ettl@forloop 684, 685, 687–689, 706
\ettl@fter 464, 467–470, 473–475, 478, 479, 482, 484
\ettl@futdef 518–521
\ettl@futdef@filt 553, 554
\ettl@futur@def 522, 523
\ettl@futur@def@collect 525, 531, 539, 549, 564, 568, 578
\ettl@futur@def@filt 552
\ettl@futured@f 520, 557
\ettl@futured@f@append 573, 576, 594
\ettl@futured@f@CheckSpecials 582, 589
\ettl@futured@f@collect 569
\ettl@futured@f@collectall 575
\ettl@futured@f@end 561, 562, 565, 574, 584
\ettl@futured@f@expand 585, 587
\ettl@futured@f@loop 559, 566, 588, 595, 596
\ettl@futured@f@space 563, 567, 577
\ettl@futured@f@test 559, 560, 595
\ettl@futured@f@try@expand 574, 576, 579
\ettl@futuredef 518, 533
\ettl@futuredef@collect 542
\ettl@futuredef@collectall 550
\ettl@futuredef@end 536, 537, 540, 547
\ettl@futuredef@filt 543, 544, 555, 570, 571
\ettl@futuredef@loop 534, 541, 546, 551
\ettl@futuredef@space 538, 548
\ettl@futuredef@test 534, 535, 546, 551
\ettl@getlistindex 786, 787, 789, 792–794, 798–800
\ettl@getsinglelist 429, 624, 627, 630, 635, 638, 640
\ettl@gobble@relax 765
\ettl@if@inlist 801, 811, 813
\ettl@ifchar 179, 179, 197, 224, 429, 617, 627, 638
\ettl@ifchardot 321, 326, 331, 336, 339, 342, 421
\ettl@ifd@f 144, 145, 160
\ettl@ifdef 144, 150–160
\ettl@ifdefempty 162, 169, 172
\ettl@ifdefined 36, 201, 211, 280, 289
\ettl@ifdefmacro 848–851, 853
\ettl@ifdefvoid 163, 166, 173
\ettl@ifincharlist 615, 617
\ettl@ifinlist 801
\ettl@ifintokslist 611, 613
\ettl@ifnch 494
\ettl@ifnextchar 486, 487, 487
\ettl@ifnspace 499, 511, 517
\ettl@ifntk 505
\ettl@ifstrnum 299, 300, 302
\ettl@ifx 178, 178, 624, 630, 635
\ettl@ifxsingle 185, 189
\ettl@intmax 60, 314, 318, 340, 341, 343, 344, 394
\ettl@listedel 769, 770, 776, 780, 781, 784, 785
\ettl@locinterplin 824, 826
\ettl@lst@count 671, 694
\ettl@lst@getitem 672, 691
\ettl@lst@modif 690, 695
\ettl@lst@opt 699–701
\ettl@meaning 132
\ettl@meaningcs 122
\ettl@naturall@@p 600, 601, 604, 607
\ettl@naturalloop 597, 598
\ettl@nbk 56, 73, 73, 78, 80, 83, 84, 146, 167, 183, 189, 191, 195, 207, 216, 217, 222, 223, 227, 232, 250, 294, 296, 297, 299, 302, 383, 385, 387, 391, 393, 394, 398, 401, 403, 611, 615, 618, 623, 626, 634, 637, 652, 670, 694, 720, 721, 723, 726, 731, 760, 762, 788, 839, 850, 855, 857
\ettl@nbk@cat 78
\ettl@nbk@else 74, 377
\ettl@nbk@IF 84
\ettl@nbk@if 82
\ettl@nbk@ifx 80
\ettl@ney 75
\ettl@numberminus 295–297
\ettl@numberspace 298, 299
\ettl@only@pdfTeX 62, 63
\ettl@onlypdfTeX 62, 249, 263, 267, 279
\ettl@or 39, 321, 326, 331, 336, 339, 671–673
\ettl@params@meaning 135, 139, 141–143, 855
\ettl@protectspace 241–243
\ettl@Rem@veInList 753, 756
\ettl@Remove 352, 363, 378
\ettl@Remove@loop 369, 370
\ettl@RemoveInList 749, 767, 768, 777–779, 782, 783
\ettl@setresult 428, 624, 627, 630, 635, 638, 652, 760, 762, 788
\ettl@singlelist@loop 640, 641, 643
\ettl@singlelist@result 642, 650
\ettl@sptoken 89, 538, 563
\ettl@strfilt 318, 319, 322, 324, 325, 327, 329, 330, 332, 334, 335, 337, 338, 346, 354, 356, 360
\ettl@strfilt@count 318, 394
\ettl@strfilt@endby 332, 335, 390
\ettl@strfilt@endbyeq 334, 392
\ettl@strfilt@equal 319, 322, 325, 382
\ettl@strfilt@equaleq 324, 384
\ettl@strfilt@instr 337, 338, 395
\ettl@strfilt@mod 312–317, 320

\ettl@strfilt@REMOVE 340, 341, 353
\ettl@strfilt@remove 356, 400
\ettl@strfilt@REPLACE 343, 344, 359
\ettl@strfilt@replace 360, 401
\ettl@strfilt@start 327, 330, 386
\ettl@strfilt@starteq 329, 388
\ettl@stringify 258, 258
\ettl@strip@meaning 130
\ettl@superg@bble 433, 434
\ettl@supergobble . . . 430, 431, 433, 433
\ettl@supergobble@end 440, 452, 463
\ettl@supergobble@loop 440, 442, 446, 450, 463
\ettl@supergobble@next 444, 462
\ettl@supergobbleheight 450
\ettl@tempa 806, 807, 809, 810
\ettl@thefontname 116, 117
\ettl@tokstolistadd 747, 748
\ettl@voidv@id 164, 165
\ettl@voidvoid 162–164
\ettl@x 525–527, 529, 532, 536–538, 541, 546, 551, 561–563, 566, 570, 580, 584, 587, 588, 596
\ettl@xif@inlist 803, 812, 814
\ettl@xifinlist 804, 808
\ettl@xifnch 494
\ettl@xifntk 505
\ettl@y . 526–530, 558, 573, 581, 583–587
\expandafter@cmds 95, 97, 99
\ExpandAftercmds 109, 352, 427, 428, 624, 627, 630, 635, 638
\expandaftercmds 93
\ExpandNext 110, 113, 601, 816
\expandnext 101, 110, 111, 307
\ExpandNextTwo 113, 430, 650, 824
\expandnexttwo 109, 224

F

\FE@ifchar 424, 424
\FE@ifcharequal 416
\FE@ifstar 161, 171, 411, 411, 728, 737, 740
\FE@modifiers 311, 425, 425, 695
\FE@testopt 164, 174, 338, 341, 344, 406, 406, 433, 597, 730
\font 116, 117
\forcsvloop 715
\forlistloop 716
\forloops 705
\fortoksloop 719
\futured@f␣(starred) 557
\futuredef 518
\futuredef␣(not␣starred) 533
\futuredef* 518
\futurelet 502, 514, 517, 541, 546, 551, 566, 588, 596

G

\gappto 721, 727
\getcharlistcount 430, 623
\getcharlistindex 623
\getcharlisttoken 623
\getcsvlistindex 797
\getlistindex 786
\gettokslistcount 623
\gettokslistindex 623
\gettokslisttoken 623
\globcount 705

H

\helper␣macros 36

I

\ifblank 64, 163, 253, 839
\ifcase 321, 592, 671
\ifcat 79, 180, 536–538, 543, 561–563, 570, 590
\ifcharlower 271
\ifcharupper 271
\ifcsname 175, 287, 301, 447, 592, 841, 845
\ifdefblankspace 144
\ifdefchar 144
\ifdefcount 144, 653
\ifdefdimen 144
\ifdefmacro 125, 847
\ifdefmathchar 144
\ifdefmuskip 144
\ifdefparam 854
\ifdefskip 144
\ifdefthechar 144
\ifdeftheletter 144
\ifdeftoks 144
\ifdim 590, 817, 829
\ifempty 162, 169, 242, 244, 393, 651
\ifeof 592
\iffirstchar 221
\iffontchar 592
\ifhbox 591
\ifincharlist 610
\ifincsvlist 811
\ifinner 591
\ifintokslist 589, 610
\ifiscs 165, 232, 690, 793, 799
\iflowercase 276
\ifnotblank 250, 806
\ifnotempty 246, 385, 389, 623, 626, 629, 634, 637, 758
\ifnum . . . 203, 213, 261, 265, 272, 274, 278, 281, 290, 356, 360, 377, 396, 397, 432, 438, 439, 443, 590, 599, 605, 692, 693, 702, 819, 827, 828, 859
\ifodd 590
\ifOneChar 201, 240
\ifOneCharWithBlanks 211
\ifOneToken 190, 426, 490
\ifsinglechar 194, 424
\ifsingletoken 94, 98, 102, 106, 182
\ifstrcmp 260, 275, 276
\ifstrdigit 280
\ifstrequal 263, 269
\ifstrmatch 277
\ifstrnum 289, 702
\ifuppercase 275
\ifvbox 591
\ifvmode 591
\ifvoid 591
\interval 815, 824

L

\lccode 274

\LetLtxMacro 478, 481
\listdel 766
\listedel 766
\listgdel 766
\listloop 716, 737, 738, 792–794
\listtocsv 737
\listxdel 766
\locinterplin 818
\lowercase 276

M

\mathchar 156
\meaning 122, 123, 130–132, 136, 140, 148, 848, 855
\meaningcs 119, 853
\muskip 154

N

\naturalloop 597
\noexpandafter 115, 147
\noexpandcs 114, 148, 660–663, 753
\number 394, 443, 447, 448, 467, 468, 473, 478, 606, 623, 626, 629, 651, 671, 682, 702, 760, 762, 828–830, 832–835
\numexpr 356, 360, 377, 394, 396, 397, 438, 439, 443, 447, 448, 467, 468, 473, 478, 599, 605, 606, 651, 671, 682, 692, 693, 760, 762, 819, 828–830, 832–835, 859

P

\p@ 817, 829, 832, 833
\PackageError 65, 820
\parameters@meaning 133, 133, 134
\parameters@meaningcs 133, 137, 138, 857
\pdfmatch 201, 203, 210, 211, 213, 278–281, 288–290, 305
\pdfstrcmp . . 29, 249, 261, 263, 265, 267
\protected@edef 252, 269, 722, 725, 776, 809
\protectspace 239, 241
\providerobustcmd 720–722, 725

R

\rmn@firstoftwo 46, 146
\rmn@secondoftwo 47, 146
\romannumeral 75, 87, 109, 111, 112, 119, 122, 124, 126, 130, 134, 138, 147, 161, 171, 190, 194, 202, 206, 212, 216, 221, 232, 239, 241, 243, 244, 246, 255, 258, 260, 271, 273, 277, 281, 286, 290, 294, 432, 599, 605, 610, 611, 614, 615, 665, 755, 788, 815, 818

S

\show 118
\showcs 118
\skip 153
\strip@meaning . . 116, 124, 124, 167, 170
\strip@meaningcs 124, 126
\strip@pt 831

T

\test␣and␣result␣macros 382
\texteuro 31, 33
\thefontname 116
\toks 151
\toksdel 782
\toksedel 782
\toksgdel 782
\toksloop 719, 740, 741
\tokstolist 740, 748
\tokstolistadd 747
\toksxdel 782

U

\uccode 272
\uppercase 275

X

\xifblank 251
\xifempty 248
\xifincsvlist 811
\xifstrcmp 248, 264
\xifstrempty 249
\xifstrequal 267, 268