

The etextools¹ macros

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

Florent CHERVET

Version 3.1

florent.chervet@free.fr

30 September 2009

Contents

Introduction	2	9 ► Lists management	21
1 · Motivation	2	9.1 · The natural loop	21
2 · Purely Expandable macros	2	9.2 · Lists of single tokens / characters	21
3 · The example file	2	9.3 · The General Command-List-Parser	23
4 · Requirements	2	9.4 · Loops into lists	25
5 · Acknowledgements – Thank You !	2	9.5 · Adding elements to csv lists	26
6 · A note for package writers	2	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
List of commands		L^AT_EX code	
1 ► General Helper Macros	4	I ► Implementation	30
2 ► Expansion control	4	I.1 · Package identification	30
3 ► Meaning of control sequences	7	I.2 · Requirements	30
4 ► Single tokens/single characters	9	I.3 · Some “helper” macros	31
4.1 · The \ifx test and the character test	9	I.4 · Expansion control	33
4.2 · Basic test macros	9	I.5 · Meaning of control sequences	34
5 ► Characters and Strings	11	I.6 · Single tokens / single characters	35
6 ► Fully expandable macros with options and modifiers	15	I.7 · Character and Strings	37
7 ► Define control sequences through groups	17	I.8 · Purely expandable macros with options	40
8 ► Vectorized \futurelet	18	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 L^AT_EX 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/ directory².

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...

²This 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 L^AT_EX.

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`.

8 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.

18 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	<code>\expandnext{\def\result}{\FEmacro{arguments}}</code>
2	<code>\expandnext\expandnext{\def\result}{\FEmacro{arguments}}</code>
more	<code>\ExpandNext{\def\result}{\FEmacro{arguments}}</code> ³

pdfT_EX 8

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 documenta-
tion). 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 -T_EXing ⚡

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`

List of Commands Provided

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



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 \longrightarrow **#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 \longrightarrow **blank space**

\expandafter \@swap \expandafter{*codeA*}{*codeB*}

will expand *codeA* once and then 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 \longrightarrow **#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 \longrightarrow **#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 \longrightarrow **{#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 instead.)

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\F0o{\Foo}     →      \def\F00{\F0o}   ←
\def\fool{\F00}
```

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** \longrightarrow **\expandnext** {**\expandnext**{**#1** } **#2** } } **#3**

$\underbrace{\hspace{1.5cm}}$ $\underbrace{\hspace{1.5cm}}$
expanded once after 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 \longrightarrow swap {21}345 \longrightarrow swap {321}45 \longrightarrow 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}}  $\longrightarrow$  macro:->54321
\ExpandNext{\def\RESULT }{\naturalloop{4}{12345}}  $\longrightarrow$  :->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** \longrightarrow **\ExpandNext** {**\ExpandNext**{**#1** } **#2** } } **#3**

$\underbrace{\hspace{1.5cm}}$ $\underbrace{\hspace{1.5cm}}$
totally expanded after 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:
 \longrightarrow **\edef**\abc{\noexpandcs{abc@ \@gobblescape\controlword}}
 \longrightarrow 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 **\ifdefmacro** 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 \langle *single token* \rangle given as argument. The test is always false if this \langle *single token* \rangle is an undefined control sequence.

\avoidvoid[\langle replacement code \rangle]{ \langle cs-token / string \rangle }

\avoidvoid*[\langle replacement code \rangle]{ \langle cs-token / string \rangle }



\avoidvoid will test the \langle cs-token \rangle with `\ifdefvoid` (from **etoolbox**). In case \langle cs-token \rangle 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 \langle replacement code \rangle (optional parameter whose default is an empty string).

Otherwise, \langle cs-token \rangle 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 \langle cs-token \rangle :

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

\avoidvoid is based on `\ifblank` test, either onto \langle string \rangle or, if \langle string \rangle is in fact a control word (tested with `\ifiscs`) on the replacement text of this control word⁷. If for your special purpose, you prefer to test if the \langle string \rangle (or the replacement text of \langle cs-token \rangle) 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 \langle string \rangle 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[\langle replacement code \rangle]{ \langle csname \rangle }

\avoidvoidcs*[\langle replacement code \rangle]{ \langle csname \rangle }



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

\avoidvoidcs { <code>@undefined</code> }	will expand to an empty string
\avoidvoidcs [<code>\deblank</code>]{ <code>zap@space</code> }	will expand to <code>\zap@space</code>
<code>\def\test{This is a test}</code>	
\avoidvoidcs [<code>\errmessage{void macro}</code>]{ <code>\test</code> }	will expand <code>\test</code>
\avoidvoidcs [<code>\errmessage{void macro}</code>]{ <code>\test</code> }	will expand <code>\errmessage{void macro}</code>

this is because `\csname This is a test\endcsname` is not defined !

Finally, clever !

<code>\protected\def\test{<code>_</code>}</code>	
\avoidvoidcs [<code>other</code>]{ <code>\test</code> }	will expand <code>other</code> : <code>\test</code> is void
\avoidvoidcs* [<code>other</code>]{ <code>\test</code> }	will expand <code>\test</code> : <code>\test</code> is not <code>\@empty</code>
\avoidvoidcs [<code>other</code>] <code>\test</code>	will expand <code>\</code> : control space, which is not void
\avoidvoidcs* [<code>other</code>] <code>\test</code>	will expand <code>\</code> : control space, which is not void

⁷if 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 character of category 0 followed by characters 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:
`\ifx<tokenA><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`):
`\ifx<tokenA><tokenB>` if: `\ifx<tokenA><tokenB>` returns **true**
 - ii) otherwise:
`\ifx<tokenA><tokenB>` if: `\if\noexpand<tokenB>\string<tokenA>` returns **true**

The **character test** is implemented in `\ettl@ifchar` and its behaviour may be tested with `\ifsingletoken`.

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{< 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 <i>true</i>
\ifiscs {x}	will expand <i>false</i> — even if x is active
\ifiscs {\ifx AB C\else D\fi}	will expand <i>false</i>
\ifiscs {\else}	will expand <i>false</i>
\ifiscs {\else}	will expand <i>true</i>
\ifiscs {\}	will expand <i>false</i>
\ifiscs {\@sptoken}	will expand <i>true</i>
\ifiscs {}	will expand <i>false</i>
\let \ALPHA=A	
\ifiscs {\ALPHA}	will expand <i>true</i>

\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 pdf_{TE}X is in use, the macro is based on the `\pdfstrcmp` primitive.



\ifnotempty{ $\langle string \rangle$ }{ $\langle true \rangle$ }{ $\langle false \rangle$ }

  **\ifnotempty** reverses the test of **\ifempty**.

\xifblank{ $\langle string \rangle$ }{ $\langle true \rangle$ }{ $\langle false \rangle$ }

 **\xifblank**
is similar to **\ifblank** except that the $\langle string \rangle$ is first expanded with **\protected@edef**.

\ifnotblank{ $\langle string \rangle$ }{ $\langle true \rangle$ }{ $\langle false \rangle$ }

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

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

\deblank{ $\langle string \rangle$ }

  **\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**⁸⁹ package.

\ifstrcmp{ $\langle string1 \rangle$ }{ $\langle string2 \rangle$ }{ $\langle true \rangle$ }{ $\langle false \rangle$ }



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

Neither $\langle string1 \rangle$ nor $\langle string2 \rangle$ is expanded during comparison. The comparison is *catcode agnostic* (use of **\detokenize**).

\xifstrequal{ $\langle string1 \rangle$ }{ $\langle string2 \rangle$ }{ $\langle true \rangle$ }{ $\langle false \rangle$ }

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



\xifstrcmp{ $\langle string1 \rangle$ }{ $\langle string2 \rangle$ }{ $\langle true \rangle$ }{ $\langle false \rangle$ }

  **\xifstrcmp** is the L^AT_EX form of **\pdfstrcmp** primitive. If this primitive is not available, **\xifstrcmp** is **\let** to **\xifstrequal**.

$\langle string1 \rangle$ and $\langle string2 \rangle$ are expanded during comparison.

\ifcharupper{ $\langle single char \rangle$ }{ $\langle true \rangle$ }{ $\langle false \rangle$ }

\ifcharlower{ $\langle single char \rangle$ }{ $\langle true \rangle$ }{ $\langle false \rangle$ }

  **\ifcharupper** compares with **\ifnum** the character code of $\langle single char \rangle$ with its **\uccode**.

\ifcharlower compares with **\ifnum** the character code of $\langle single char \rangle$ with its **\lccode**.

\ifuppercase{ $\langle string \rangle$ }{ $\langle true \rangle$ }{ $\langle false \rangle$ }

\iflowercase{ $\langle string \rangle$ }{ $\langle true \rangle$ }{ $\langle false \rangle$ }

 **\ifuppercase** compares the $\langle string \rangle$ with **\uppercase**{ $\langle string \rangle$ }.


\iflowercase compares the $\langle string \rangle$ with **\lowercase**{ $\langle string \rangle$ }.

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{ $\langle pattern \rangle$ }{ $\langle string \rangle$ }{ $\langle true \rangle$ }{ $\langle false \rangle$ }

 **\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{[*]$}{ $\langle string \rangle$ }`

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

\ifstrdigit{ $\langle string \rangle$ }{ $\langle true \rangle$ }{ $\langle false \rangle$ }

  **\ifstrdigit** expands to $\langle true \rangle$ if $\langle string \rangle$ 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{ $\langle string \rangle$ }{ $\langle true \rangle$ }{ $\langle false \rangle$ }

  **\ifstrnum** expands to $\langle true \rangle$ if $\langle string \rangle$ is a **number in the sense of ε -TeX**, that means:

`\number $\langle string \rangle$` will be the same as: `\deblank{ $\langle string \rangle$ }`

under the standard catcode regime, if $\langle string \rangle$ is a positive integer.

in other words:

```
\edef\resultA{\number $\langle string \rangle$ }
\edef\resultB{\deblank{ $\langle string \rangle$ }}
\ifx\resultA\resultB    will be true
```

$\langle string \rangle$ must be of the form: `blue***`


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 $\langle true \rangle$.

To tell all the truth, **\ifstrnum** expands $\langle true \rangle$ 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 $\langle string \rangle$.**

\DeclareStringFilter[$\langle global \rangle$]{ $\langle command-name \rangle$ }{ $\langle stringA \rangle$ }

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

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

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

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

Equality is \catcode dependent.

You may also check that $\langle stringA \rangle$ 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" \longrightarrow $\langle stringA \rangle$

`\CompareYES` $\{\langle string \rangle\}\{\langle true \rangle\}\{\langle false \rangle\}$ expands $\langle true \rangle$ if $\langle string \rangle$ is "**`_YES_`**"

`\CompareYES=` $\{\langle string \rangle\}\{\langle true \rangle\}\{\langle false \rangle\}$ is the same

`\CompareYES=.` $\{\langle string \rangle\}\{\langle true \rangle\}\{\langle false \rangle\}$ is also the same

`\CompareYES==` $\{\langle string \rangle\}\{\langle true \rangle\}\{\langle false \rangle\}$ expands $\langle true \rangle$ if $\langle string \rangle$ is "**`YES`**"

`\CompareYES<` $\{\langle string \rangle\}\{\langle true \rangle\}\{\langle false \rangle\}$ expands $\langle true \rangle$ if $\langle string \rangle$ **begins with** "**`_YES_`**"

`\CompareYES<=` $\{\langle string \rangle\}\{\langle true \rangle\}\{\langle false \rangle\}$ expands $\langle true \rangle$ if $\langle string \rangle$ **begins with** "**`YES`**"

`\CompareYES>` $\{\langle string \rangle\}\{\langle true \rangle\}\{\langle false \rangle\}$ expands $\langle true \rangle$ if $\langle string \rangle$ **ends with** "**`YES_`**"

`\CompareYES>=` $\{\langle string \rangle\}\{\langle true \rangle\}\{\langle false \rangle\}$ expands $\langle true \rangle$ if $\langle string \rangle$ **ends with** "**`YES`**"

`\CompareYES?` $\{\langle string \rangle\}\{\langle true \rangle\}\{\langle false \rangle\}$ expands $\langle true \rangle$ if $\langle string \rangle$ **contains** "**`YES`**"

`\CompareYES?[n]` $\{\langle string \rangle\}\{\langle true \rangle\}\{\langle false \rangle\}$ expands $\langle true \rangle$ if $\langle string \rangle$
contains "**`YES`**" more than n times

`\CompareYES-` $\{\langle string \rangle\}$ **removes all occurrences** of "**`YES`**" in $\langle string \rangle$

`\CompareYES-[n]` $\{\langle string \rangle\}$ **removes at most n occurrences** of "**`YES`**"

`\CompareYES+` $\{\langle string \rangle\}\{\langle stringB \rangle\}$ **replaces all occurrences**
of "**`YES`**" by $\langle stringB \rangle$ in $\langle string \rangle$

`\CompareYES+[n]` $\{\langle string \rangle\}\{\langle stringB \rangle\}$ **replaces at most n occurrences**
of "**`YES`**" by $\langle stringB \rangle$ in $\langle string \rangle$

And finally:

`\CompareYES!` $\{\langle string \rangle\}$ expands to **the number of times** "**`YES`**" can be found in $\langle string \rangle$

`\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 $\langle string \rangle$ 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 $\langle string \rangle$ is '`[`'.

`\CompareYES` and each of its form are purely expandable thank to [`\FE@modifiers`](#).

You should not test a $\langle string \rangle$ which contains the following sequence:

`/_8E_11n_11d_11§_7S_11t_11r_11i_11n_11g_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 $\{\langle\#1\rangle\}\{\langle\textit{commands}\rangle\}\{\langle\textit{default option}\rangle\}$



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

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

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

`\MacroWithOptions{[...]` 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 $\{\langle\#1\rangle\}\{\langle\textit{star-commands}\rangle\}\{\langle\textit{non-star commands}\rangle\}$



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 $\{\langle\textit{Variant Character}\rangle\}\{\langle\#1\rangle\}\{\langle\textit{special-commands}\rangle\}\{\langle\textit{normal-commands}\rangle\}$



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 `\langle\textit{Variant Character}\rangle` (the `character-test` is used).

Really, `\FE@ifchar` is based on `\ifsinglechar` therefore the "character" 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 interesting example of use of **\FE@modifiers** is given in the implementation of \ettl@lst@modif.

\ettl@supergobble[*code*]{*<n>*}{*<N>*}{*<tok₁>*}...{*<tok_n>*}{*<TOK_{n+1}>*}{*<tok_{n+2}>*}...{*<tok_N>*}



\ettl@supergobble{*<n>*}{*<N>*} will:

- i) gobble the first *<n>* tokens (or groups of tokens) it founds just after
- ii) keep the *<n + 1>* token
- iii) gobble the last tokens *<n + 2>* to *<N>*
- iv) then and after all, expand to *<TOK_{n+1}>*

In other words, the list contains *<N>* tokens, **\ettl@supergobble** expands the *<n + 1>* and discards the rest.

Now if *<n>*=*<N>*, **\ettl@supergobble** gobbles the *<N>* tokens (including the last).

And if *<n>*>*<N>* or if *<n>*< 0, **\ettl@supergobble** expands to *<TOK_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 *<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 were 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 L^AT_EX's **\newcommand**, with robust commands from **etoolbox**-**\newrobustcmd** and with L^AT_EX'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 L^AT_EX's **\@ifstar** but for any character (or *modifier*). Whereas **\@ifstar**-test is sensitive to the category code of the star (the character ‘ \star_{12} ’ – that means that the category code of \star must be 12 as defined in L^AT_EX'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 finds 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 **\futuredef¹¹**.

¹¹this 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 category 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*}}}

\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 i.e. **\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 equality 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:

<code>\begin{group}</code>	<code>\catcode'\!=13 \catcode'\.=8 \catcode'\: 3</code>	
<code>\global\def\mylist{:\!relax=.0}</code>		<code>\ifintokslist</code>
<code>\end{group}</code>		
<code>\expandnext{\ifincharlist!}\mylist{true}{false}</code>	true	false
<code>\expandnext{\ifincharlist0}\mylist{true}{false}</code>	true	true
<code>\expandnext{\ifincharlist:}\mylist{true}{false}</code>	true	false
<code>\expandnext{\ifincharlist\relax}\mylist{true}{false}</code>	true	true

`\ifincharlist` is used in the definition of `\futuredef`.

`\gettokslistindex` $\{\langle item \rangle\}\{\langle list\ of\ single\ tokens \rangle\}$



`\gettokslistindex` expands to the index of $\langle item \rangle$ 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` $\{\langle item \rangle\}\{\langle list\ of\ single\ tokens \rangle\}$



`\getcharlistindex` expands to the index of $\langle item \rangle$ in the list of single tokens (the index is 0 for the first item, -1 if $\langle item \rangle$ 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` $\{\langle list\ of\ single\ tokens \rangle\}$

`\gettokslisttoken` $\{\langle item \rangle\}\{\langle list\ of\ single\ tokens \rangle\}$



`\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 $\langle item \rangle$ 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 $\langle item \rangle$ 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 beginning: 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`, `\getcsvglistindex` 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 $\langle separator \rangle$. This $\langle separator \rangle$ may be any sequence of tokens, but the special sequence:

`/_8E_11n_11d_11§_7L_11i_11s_11t_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{,}`

¹⁴Unfortunately, `\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, specifying 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 itemn
or: \MyParser [n]{item<sep>item<sep>item} expands to itemn
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_11n_11d_11§_7L_11i_11s_11t_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 L^AT_EX'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** \rightarrow { 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** \rightarrow { 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\}` \longrightarrow 6

Spaces are not counted, however...

```

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

```



Those macros are just like `\csvloop`, `\listloop` and `\toksloop` but the syntax is quite the same as L^AT_EX'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 be 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: `\forname-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 \}
\csvlistteadd{\ csvListmacro }\{\ item \}
\csvlistxadd{\ csvListmacro }\{\ item \}

```



`\csvlistadd` adds an item to a csvlist. `\csvlistteadd` 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 characters, it is always possible to convert lists (i. e., changing their separator) using them. For example, 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
\edef\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}

```

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



is the same as:

if you want `\myList` to be global, use the second form with `\xdef` instead of `\edef`.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

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



is the same as:

```
\meaning\myList: macro:->\alpha |2 \beta |2 \gamma |2 \ifeof |2 + |2 * |2 $ |2
```

if you want `\myList` to be global, use the second form with `\xdef` instead of `\edef`.

```
listtocsv*({ target: csvlistmacro }){{ source: Listmacro or expanded List }}
```



```
\edef\csvList{\listtocsv\etbList}
```

if you want `\csvList` to be global, use the second form with `\xdef` instead of `\edef`.

```
csvtolistadd*{ { target: Listmacro } } { { source: item, item, item } }
```

`\csvtolistadd` acts similarly but both arguments are mandatory:

```
\csvtolistadd\myList{three.four.five}
```

```
\meaning\myList: macro:->one|_three|_five|_
```

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

\tokstolistadd acts similarly but both arguments are mandatory.

0

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`, `\listadd` 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 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}
\mycounter = \listdel!\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).
etextools ©  Florent Chervet

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`:

<code>\interval{0}{3,5,12,20}</code>	will expand to	0
<code>\interval{3}{3,5,12,20}</code>	will expand to	1
<code>\interval{4}{3,5,12,20}</code>	will expand to	1
<code>\interval{5}{3,5,12,20}</code>	will expand to	2
<code>\interval{19}{3,5,12,20}</code>	will expand to	3
<code>\interval{20}{3,5,12,20}</code>	will expand to	4
<code>\interval{21}{3,5,12,20}</code>	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{\mathit{X}}{\mathit{X}_i}{\mathit{Y}_i}`

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

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.



L^AT_EX code



Implementation

I-1 ↗ Package identification

```

1 \<package>
2 \NeedsTeXFormat{LaTeX2e}[1996/12/01]
3 \ProvidesPackage{etextools}
4   [2009/09/30 v3.1 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 undesirable side effects if read inside `\halign` or `\valign`. Finally, we could have chosen **7** but then a sequence like: ‘/7/7’ is read by T_EX like ‘^7^7’ 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}% f ignore
31   \TMP@EnsureCode{128}{14}% \texteuro comment €

```

```

32 }\TMP@EnsureCode{163}{14}% f 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@expandaftthree{\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@firstspace\detokenize{#1} \\\{#3}{#2}/}
56 \long\def\ettl@firstspace#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@onlypdfTeX{#1}{#2}}{}}
63 \def\ettl@onlypdfTeX#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 ‘`/s/s`’) is **not blank**.



Usage: `\ettl@nbk <string>\chcat/8\chcat/8<true>\chcat/8\chcat/8`

if $\langle string \rangle$ is blank: #1=' / ', #2=∅, #3= $\langle true \rangle$, #4= $\langle false \rangle$
otherwise: #3=' / ', #4= $\langle true \rangle$ (and #5= $\langle false \rangle$)

```
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 $\langle true \rangle$ if $\langle string \rangle$ is not blank AND if its first token has the same category code of $\langle tokenA \rangle$:

Usage: **\ettl@nbkcat** $\langle tokenA \rangle \langle string \rangle // \langle same\ catcodes \rangle \langle different\ catcodes \rangle //$

```
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 $\langle true \rangle$ if $\langle string \rangle$ is not blank AND if its first token is equal to $\langle tokenA \rangle$ in the sense of **\ifx**:

USAGE: **\ettl@nbk@ifx** $\langle tokenA \rangle \langle string \rangle // \langle true \rangle \langle false \rangle //$

```
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 $\langle true \rangle$ if $\langle string \rangle$ is not blank AND if its first token is equal to $\langle tokenA \rangle$ in the sense of **\if**:

USAGE: **\ettl@nbk@if** $\langle tokenA \rangle \langle string \rangle // \langle true \rangle \langle false \rangle //$

```
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\etl@thefontname\expandafter\strip@meaning\the\font}
117 \font\etl@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}{\etl@meaningcs\endcsname{#1}}
121   {meaning\endcsname}\undefined}}
122 \def\etl@meaningcs#1{\expandafter\meaning\csname#1\endcsname}% here we don't need \z@
123                                     % because \meaning is never defined
```

`\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}{\etl@strip@meaning}{\etl@meaning}\endcsname#1}{z@\endcsname}}
126 \providecommand*\strip@meaningcs[1]{\romannumeral\csname\ifcsdef{#1}%
127   {\ifcsmacro{#1}{\etl@strip@meaning}{\etl@meaning}%
128     \expandafter\endcsname\csname#1\endcsname}
129   {z@\endcsname}}
130 \def\etl@strip@meaning{\expandafter\expandafter\expandafter\z@% for \romannumeral in c
131   \expandafter\strip@prefix\meaning}
132 \def\etl@meaning{\expandafter\z@\meaning}
```

`\parameters@meaning`

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

`\ifdefcount`

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

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

`\ifdefcount`

```
144 \def\etl@ifdef[#1]{\expandafter\etl@ifd@f\expandafter{#1}}
145 \def\etl@ifd@f#1#2{%
146   \csdef{etl@ifdef#2}##1#1##2/End$Meaning/{\etl@nbk##2//{\rmn@firstoftwo}{\rmn@secondoftwo}}
147   \csdef{ifdef#2}##1{\noexpand\romannumeral\noexpandafter%
148     \noexpandcs{etl@ifdef#2}\noexpand\meaning##1#1/End$Meaning/}%//{##2}{##3}}/}
149 }
150 \etl@ifdef[\string\count]{count} % defines \def\ifdefcount
151 \etl@ifdef[\string\toks]{toks} % \def\ifdefcount
152 \etl@ifdef[\string\dimen]{dimen} % \def\ifdefcount
153 \etl@ifdef[\string\skip]{skip} % \def\ifdefcount
154 \etl@ifdef[\string\muskip]{muskip} % \def\ifdefcount
155 \etl@ifdef[\string\char]{char} % \def\ifdefcount
156 \etl@ifdef[\string\mathchar]{mathchar} % \def\ifdefcount
157 \etl@ifdef[\detokenize{blank space}]{blank space}% \def\ifdefcount
158 \etl@ifdef[\detokenize{the character}]{thechar}% \def\ifdefcount
159 \etl@ifdef[\detokenize{the letter}]{theletter} % \def\ifdefcount
160 \undef\etl@ifdef\undef\etl@ifd@f
```


\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   {\etl@voidvoid{\etl@ifdefempty\ifempty}}
163   {\etl@voidvoid{\etl@ifdefvoid\ifblank}}}
164 \long\def\etl@voidvoid#1#2{\FE@testopt{#2}{\etl@voidv{id#1}}{}}
165 \long\def\etl@voidv{id#1#2[#3]#4{\ifiscs{#4}{#1{#4}}{#2{#4}}{\z@#3}{\z@#4}}
```

and the helper macros:

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

\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   {\etl@avoidvoidcs{\etl@ifdefempty}}
173   {\etl@avoidvoidcs{\etl@ifdefvoid}}}
174 \long\def\etl@avoidvoidcs#1#2{\FE@testopt{#2}{\etl@voidvoidcs#1}{}}
175 \long\def\etl@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

\etl@ifx **\etl@ifx** is the *equality-test macro* for character-test`\ifx` test. It is designed to be used inside `\csname...\endcsname` like:

`\etl@ifx<tokenA><tokenB>firstsecond`:

```
178 \long\def\etl@ifx#1#2{\csname etl@\ifx#1#2\else2\fi oftwo\endcsname}
```

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

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

\ifsingletoken **\ifsingletoken** is a safe `\ifx`-test:

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

`\ifsinglechar` Test if #2 is a single character equal to #1:



28

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

`\ifOneChar` `\ifOneChar<string><true><false>` detokenizes <string> first (see also `\ifOneToken`):



28

```
201 \etl@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@\etl@firstspace{#1}
207   {\etl@nbk#1//{second}}{\if @\detokenize{#1}@second\else first\fi}//}
208   {\if @\expandafter\etl@cdr\detokenize{#1}@nil @%
209     first\else second\fi}oftwo\endcsname}
210 \fi%\pdfmatch
```

`\ifOneCharWithBlanks`



28

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



28

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



28

```
232 \newcommand\ifiscs[1]{\romannumeral\csname rmn@\etl@nbk#1//%
233   {\if @\expandafter\etl@cdr\detokenize{#1}@nil @secondoftwo\etl@csname\fi
234   \if @\detokenize\expandafter{\etl@cdr#1@nil}%
235     \expandafter\etl@firstspace
236     \else secondoftwo\etl@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:



28

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

`\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 example of a loop which is 2-purely expandable:

✓ Character and Strings

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

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

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

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

```

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

```

```
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}
```


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

```

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

```

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


```

\ifupercase Compares the \uppercase transformation of a string with itself: 

```

275 \newrobustcmd\ifupercase[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. 

```

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

```

\ifstrdigit \ifstrdigit expands *<true>* if *<string>* is a single digit (without spaces): 

```

280 \etl@ifdefined\pdfmatch
281 \newcommand\ifstrdigit[1]{\romannumeral\csname rmn@\ifnum\pdfmatch{\detokenize{^[[[:digit:]]{1}}}{\detokenize{#1}}}=1 first\else second\fi oftwo\endcsname}
282   {\detokenize{#1}}=1 first\else second\fi oftwo\endcsname}
283 \else
284 \def\do#1{\cslet\etl@number#1=#1%
285 }\docsvlist{0,1,2,3,4,5,6,7,8,9}
286 \newcommand\ifstrdigit[1]{\romannumeral\csname rmn@%
287   \ifcsname etl@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 \etl@ifdefined\pdfmatch
290 \newcommand\ifstrnum[1]{\romannumeral\csname rmn@\ifnum\pdfmatch
291   {\detokenize{^([[:space:]]*~?)*+[[[:digit:]]+[[[:space:]]*$}}{\detokenize{#1}}}=1 %
292   first\else second\fi oftwo\endcsname}
293 \else
294 \newcommand\ifstrnum[1]{\romannumeral\csname rmn@\etl@nbk#1//%
295   {\expandafter\etl@numberminus\detokenize{#1}-/End$String/}{second}}//oftwo\endcsname}
296 \long\def\etl@numberminus#1-#2/End$String/{\etl@nbk#2//%
297   {\etl@nbk#1//{second}}{\etl@numberminus#2/End$String/}}//%
298   {\expandafter\expandafter\expandafter\etl@numberspace\deblank{#1} /End$String,
299 \long\def\etl@numberspace#1 #2/End$String/{\etl@nbk#2//{second}}{\etl@ifstrnum#1/End$String,
300 \long\def\etl@ifstrnum#1#2/End$String/{%
301   \ifcsname etl@number#1\endcsname% #1 detokenized before, ok
302     \etl@nbk#2//{\etl@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\etl@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 \etl@else\etl@strfilt\etl@strfilt@replace{#1}{#7}{#3}[#5]{#6}
361 \else\expandafter\@firstoftwo%
362 \fi}

```

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

```

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


```

\etl@Remove@loop  **\etl@Remove@loop** is entitled to break the loop:

```

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

```


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

```

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

```

402 \long\def\etl@BasicFilter#1#2#3/End$String/{\expandafter\etl@B@sicFilter #1#3//#2/End$
403 \long\def\etl@B@sicFilter#1/#2//#3/End$String/{\etl@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]{\etl@FE@testopt#1[//%
407     {#2#1}%
408     {#2[{#3}]{#1}}}%
409 \long\def\etl@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]{\etl@FE@ifstar#1/*/%
412     {#2}%
413     {#3{#1}}}%
414 \long\def\etl@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]{\etl@FE@charequal#1/=/%
417     {#2}%
418     {#3{#1}}}%
419 \long\def\etl@FE@charequal#1=#2/#3#{\csname @\if @\detokenize{#1#2}%
420     first\else second\fi oftwo\endcsname}
```

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



```
421 \newcommand\etl@ifchardot[1]{\etl@FE@chardot#1/. /%
422 \long\def\etl@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\etl@FE@modifiers%
428         {\ExpandAftercmds{\etl@setresult 12of3><}
429         {\etl@getsinglelist{\etl@ifchar{#2}}{#1}}{#2}}
430     {\ExpandNextTwo{\etl@supergobble[{{#2}}]{-1}{\getcharlistcount{#1}+1}}%
431 \long\def\etl@FE@modifiers#1#2#3{\expandafter\etl@supergobble%
432     \expandafter[\romannumeral-\q\ifnum#2<0 {#3}\fi]{#2}{#1+1}}
```

`\etl@supergobble` `\etl@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 `\etl@supergobbleheight` etc.) to avoid too long loops.



```
433 \newcommand\etl@supergobble[1]{\FE@testopt{#1}\etl@supergobble{}}
434 \long\def\etl@supergobble[#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             \etl@supergobble@loop{#3+2}0{\etl@supergobble@end{}}}%
441         \else
442             \expandafter\etl@supergobble@loop\expandafter{%
```



```

443      \number\numexpr\ifnum\numexpr2*(#2-(#3))>0 #3+1\else#2+2\fi}{#3+2}%
444      {\etl@supergobble@next}{#1}}%
445  \fi\fi}
446 \long\def\etl@supergobble@loop#1#2#3{%
447   \ifcsname etl@supergobble\number\numexpr#1\endcsname
448     \csname etl@supergobble\number\numexpr#1\endcsname
449     {#3{#2-(#1)-1}}}%
450   \else\etl@supergobbleheight{\etl@supergobble@loop{#1-8}{#2-8}{#3}}%
451   \fi}
452 \long\def\etl@supergobble@end#1#2#3{\fi\fi\fi#1#2}
453 \long\csdef{etl@supergobbleheight}{#1\fi#2#3#4#5#6#7#8#9{\fi#1}
454 \long\csdef{etl@supergobble7}{#1#2\fi#3#4#5#6#7#8#9{#1}
455 \long\csdef{etl@supergobble6}{#1#2\fi#3#4#5#6#7#8{#1}
456 \long\csdef{etl@supergobble5}{#1#2\fi#3#4#5#6#7{#1}
457 \long\csdef{etl@supergobble4}{#1#2\fi#3#4#5#6{#1}
458 \long\csdef{etl@supergobble3}{#1#2\fi#3#4#5{#1}
459 \long\csdef{etl@supergobble2}{#1#2\fi#3#4{#1}
460 \long\csdef{etl@supergobble1}{#1#2\fi#3{#1}
461 \long\csdef{etl@supergobble0}{#1#2\fi{#1}
462 \long\def\etl@supergobble@next#1#2#3#4{\fi
463   \etl@supergobble@loop{#3}0{\etl@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 **\etl@ifnextchar** to allow macro definitions (with arguments) inside the argument of **\AfterGroup**:

```

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

```

\AfterAssignment **\AfterAssignment** can be given arbitray code:



```

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

```

\aftergroup@def The macro is based on **letltxmacro** package. Therefore, **\aftergroup@def** works with commands with optional arguments and with the ones defined using L^AT_EX'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 etl@ftergroup@def\number\numexpr\the\etl@fter+1\endcsname
479   \global\advance\etl@fter@ne
480   \etex@let@primitive\let=\etex@let@primitive
481   \aftergroup\LetLtxMacro \aftergroup#1%
482   \expandafter\aftergroup\csname etl@ftergroup@def\the\etl@fter\endcsname
483   \aftergroup\global \aftergroup\undef
484   \expandafter\aftergroup\csname etl@ftergroup@def\the\etl@fter\endcsname}
485 \let\etl@aftergroup@def\aftergroup@def

```

I-10 \futuredef

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



```

486 \long\def\ifchar#1#2{\etl@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` conditionals (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@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@xifnch%
501     \fi\fi\fi/{#1}/{}}%
502   \fi\futurelet\@let@token\ettl@xifnch}
```

`\ettl@xifnch` 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. If the test fails, the parameters is appended again to the input):

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

`\ettl@xifntk` `\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@xifntk%
513     \fi\fi\fi/{#1}/{}}%
514   \fi\futurelet\@let@token\ettl@xifntk}
```

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

`\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` This is the scanner.

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

```

520      {\ettl@futdef\ettl@futures@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 ett1@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}}

```

`\futuredef` (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\xt/{#2}{##1}}%
540   \long \def \ettl@futuredef@end##1#2/Ne\xt/##3##4{##2\endgroup\def#3{##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\xt/#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@futuredef
547   {\ettl@futuredef@end{#5}/Ne\xt/{#4}}/Ne\xt/}

```

`\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\xt/#3#4 {%
549   \ettl@futur@def@collect#1#2/Ne\xt/{#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\xt/#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\xt/{##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}}

```

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

```

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

```

```

563 \ifcat\noexpand\etl@x\etl@sptoken\etl@futures@f@space#1\else
564 \etl@futures@def@collect#1\fi\fi\fi/Next/{##1}{##2}}}%
565 \long \def \etl@futures@f@end##1/Next/##2##3##4{##1\endgroup\def#3{##2}#4##4}%
566 \futurelet \etl@x \etl@futures@f@loop}
567 \long\def\etl@futures@f@space#1#2/Next/#3#4#5 {%
568 \etl@futures@def@collect#1#2/Next/{#3}{#4}{#5}{ }}

```

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

```

569 \long\def\etl@futures@f@collect#1#2/Next/#3#4#5#6{##2%
570 \ifcat\noexpand\etl@x\relax \etl@futures@f@filt\unexpanded
571 \else \etl@futures@f@filt#1
572 \fi{#6}{#4}
573 {\let \etl@y \@undefined \etl@futures@f@append/Next/{#3}}{#6}}%
574 {\etl@futures@f@try@expand{#3}\etl@futures@f@end{#6}}/Next/}

```

`\etl@futures@f@collectall` is used when `\futures@f*` is called with an empty optional argument:

```

575 \long\def\etl@futures@f@collectall#1#2/Next/#3#4#5#6{##2%
576 \etl@futures@f@try@expand{#3}\etl@futures@f@append{#6}}

```

`\etl@futures@f@space` is used in case the token is a space token:

```

577 \long\def\etl@futures@f@space#1#2/Next/#3#4#5 {%
578 \etl@futures@def@collect#1#2/Next/{#3}{#4}{#5}{ }}

```

`\etl@futures@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\etl@futures@f@try@expand#1#2#3{%
580 \expandafter\ifx\noexpand\etl@x\etl@x
581 \let\etl@y=#2%
582 \else\etl@futures@f@CheckSpecials{#3}%
583 {\let \etl@y=#2}%
584 {\ifx\etl@x\etl@y \let \etl@y \etl@futures@f@end\else
585 \let \etl@y \etl@futures@f@expand\fi}%
586 \fi\etl@y/Next/{#1}{#3}}

```

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

```

587 \long\def\etl@futures@f@expand/Next/#1#2#3{\let\etl@y\etl@x
588 \expandafter\futurelet\expandafter\etl@x\expandafter\etl@futures@f@loop#3}

```

`\etl@futures@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 T_EX):

```

589 \long\def\etl@futures@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, `\etl@futures@f@append` appends the token to the result and goes back to the loop:


```

594 \def\etl@futures@f@append/Next/#1#2#3{%
595 \def\etl@futures@f@loop{\etl@futures@f@test{#1#3}}%
596 \futurelet\etl@x\etl@futures@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}\etl@naturalloop{\do}}
598 \def\etl@naturalloop[#1]#2#3{%
599   \ifnum\numexpr#2>0 \expandafter\@swaparg\expandafter{\romannumeral-'\q#1[0]{#3}{#3}}%
600   {\etl@naturalloop@p[#1]}{#2-1}{0}{#3}}
601   \ExpandNext{\etl@naturalloop@p[#1]}{#2-1}{1}{#3}}{#1[1]{#3}{#3}}%
602   \else\@swap{\unexpanded{#3}}}%
603   \fi}
604 \def\etl@naturalloop@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   {\etl@naturalloop@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 `\etl@nbk...` of course: 

`\ifincharlist` `\ifincharlist<character or token><list of single characters or tokens>` is the same, with a different test macro: `\etl@ifchar` is used instead of `\etl@ifx`: 

```

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


```

`\etl@dosinglist` 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\etl@dosinglist#1#2{\etl@nbk#2//%
619   {\etl@dosinglist@loop{#1}#2//{\etl@dosinglist@loop{#1}}{\etl@breakloop{}}}
620   {\etl@breakloop{}}//End$List/}
621 \long\def\etl@dosinglist@loop#1#2#3#4/#5#6#7/End$List/{%
622   #1{#2}#6{#3}#4/{#6}{#7}/End$List/}

```

`\gettokslistindex` `\gettokslistindex<item><tokenlist-macro>` 

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

`\gettokslistcount` `\getcharlistcount` `\gettokslisttoken` `\getcharlisttoken` The following three macros are the *entry points*. `\ExpandAftercmds` is applied to `\etl@getsinglist` which initiates the loop: we ask for total expansion. After expansion, `\etl@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 `\etl@XofY` macros:

```

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

```



```

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

```

`\etl@getsinglelist` initiates the loop (we test if the list or the $\langle item \rangle$ is empty first):

```

640 \cmdlabel{impl:etl@getsinglelist}
641 \long\def\etl@getsinglelist#1#2{\etl@singlelist@loop{-1}{-1}}{#2}/%
642   {\etl@expandafthree\etl@singlelist@loop#1}%
643   {\expandafter\etl@singlelist@result\@thirdofthree}/End$List/}

```

`\etl@singlelist@loop` tests each token and update registers:

```

644 \long\def\etl@singlelist@loop#1#2#3#4#5/#6#7#8/End$List/{%
645   #7{#4}
646   {{#1+1}{#2+1+0*(0)}{#4}}
647   {{#1+1}{#2+1}{#3}}#5//{#7}{#8}/End$List/}
648 %   \csname @#1#5{first}{second}oftwo\endcsname
649 %   {#8{#1}{#2+1}{#3+1+0*(0)}{#5}{#6}}{#8#9}
650 %   {#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 $\langle item \rangle$ (if found): it is incremented by 1 but at the time $\langle item \rangle$ 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 $\langle item \rangle$ is found, **#4** becomes this $\langle item \rangle$ (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 `\etl@nbk`).

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

```

651 \def\etl@singlelist@result#1#2#3#4/End$List/{\ExpandNextTwo\@swaptwo%
652   {\number\numexpr\ifempty{#3}{-1}{#2}}{\number\numexpr#1}{#3}}

```

Then `\etl@setresult` finishes the job:

```

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

```

I-11.3 – General Lists and Loops Constructor

`\DeclareCmdListParser` acts in the same way as **etoolbox**-`\DeclareListParser` and the command-list-parser are sensitive to the category code of the $\langle separator \rangle$

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

```

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

```

`\ettl@defcmdparser` does 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.

```

667 \protected\long\def\ettl@defcmdparser#1#2#3#4#5#6#7{%#1=global,#2=command,#3=sep,#4=star
668 #1\long\def#4##1##2[##3]##4{% ##1=case, ##2=expandafter???, ##3=do, ##4=list
669 ##2{##4}% ifiscs or @thirdofthree
670 {\expandafter\@swaparg\expandafter{##4}{#4{##1}\@thirdofthree[##3]}}
671 {\ettl@nbk##4//}%
672 {\ifcase##1 \ettl@or\@swaptast{\number\numexpr#60{\ettl@lst@count}}#6%
673 \or \ettl@or\@swaptast{#60{\ettl@lst@getitem{##3}}#6%
674 \or \ettl@or\@swaptast{#5{##3}}#5%
675 \or \ettl@fi\@swaptast{#60{##3}}#6%
676 \fi{##4#3//}{\ettl@breakloop{}}}%
677 {\ettl@breakloop{}}//End$List/{}%
678 #1\long\def#5##1##2#3##3##4/#5##6##7/End$List/{%
679 \if @\detokenize{##2}\@expandafter\@gobbletwo\fi\@firstofone{##1{##2}}%
680 ##6{##1}##3##4//{##6}{##7}/End$List/{%
681 #1\long\def#6##1##2#3##3##4##5/#6##7##8/End$List/{%
682 \if @\detokenize{##3}\@expandafter\@gobbletwo\fi\@firstofone{##2{##1}{##3}}%
683 \expandafter##7\expandafter{\number\numexpr##1+1}{##2}##4##5//{##7}{##8}/End$List,
684 #1\protected\def#7{\ifchar*%
685 {\@ifchar+\ettl@forloop{\expandafter#2\expandafter*\expandafter+}{####1}####2}}
686 {\ettl@forloop{\expandafter#2\expandafter*\expandafter+}{####1}}
687 {\@ifchar+\@ifchar*%
688 {\ettl@forloop{\expandafter#2\expandafter*\expandafter+}{####1}##
689 {\ettl@forloop{\expandafter#2\expandafter+}{####1}####2}}
690 {\ettl@forloop{\expandafter#2}{####1}}
691 #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@lst@getitem` is the helper macro in case we ask for an item (cf. `\csvloop[4]\mylist`) and `\ettl@lst@count` is as basic as it can be!

```

692 \long\def\ettl@lst@getitem#1[#2]#3{%
693 \ifnum\numexpr#1<0 \@swap{\breakloop{}}\fi
694 \ifnum\numexpr#1=#2 \@swap{\breakloop{##3}}\fi
695 \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+*`):

```

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

```

`\breakloop` `\breakloop` gobbles anything until the ‘/EndList/’ delimiter:

```

704 \long\def\ettl@breakloop#1#2/End$List/{#1}
705 \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.

```

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

```

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

`\forcsvloop`

```

716 \DeclareCmdListParser\csvloop{,}

```

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

`\forlistloop`

```

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

```

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

`\fortoksloop`

```

720 \DeclareCmdListParser\toksloop{}

```

`\csvlistadd`

`\csvlistgadd`

`\csvlistead`

`\csvlistxadd`

```

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

```

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



```

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

```

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

```

738 \newcommand\listtocsv[1]{\FE@ifstar{#1}{\etl@convertlist{\listloop*}\etl@do@listtocsv.
739   {\etl@convertlist{\listloop\etl@do@listtocsv}}}}
740 \long\def\etl@do@listtocsv#1{\unexpanded{#1,}}

```

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

```

741 \newcommand\tokstolist[1]{\FE@ifstar{#1}{\etl@convertlist{\toksloop*}\etl@do@tokstolist.
742   {\etl@convertlist{\toksloop\etl@do@tokstolist}}}}
743 \begingroup\catcode'\|=3% etb catcode
744 \long\gdef\etl@do@tokstolist#1{\unexpanded{#1}|}
745 \endgroup

```

`\csvtolistadd` `\csvtolistadd` is not purely expandable:



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

`\tokstolistadd` `\tokstolistadd` is not purely expandable:



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

`\etl@RemoveInList` This is the general constructor for deletion into lists with any separator:

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

`\listdel` `\listdel` removes an *<item>* from a list, `\listedel` expands the *<item>* (with `\protected@edef`) first, `\listgdel` make the assignment to the (shorter-)list global and `\listxdel` both expands the *<item>* and makes the assignment global:



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

`\csvdel` `\csvdel` removes an *<item>* from a list, `\csvedel` expands the *<item>* (with `\protected@edef`) first, `\csvgdel` make the assignment to the (shorter-)list global and `\csvxdel` both expands the *<item>* and makes the assignment global:



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

`\toksdel` `\toksdel` removes an *<item>* from a list, `\toksedel` expands the *<item>* (with `\protected@edef`) first, `\toksgdel` make the assignment to the (shorter-)list global and `\toksxdel` both expands the *<item>* and makes the assignment global:



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

```

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

```

`\getcsvglistindex` The command is robust, not purely expandable:

```

798 \newrobustcmd\getcsvglistindex[3][\@ifstar%
799   {\etl@getlistindex}\csvgloop{,\#1}{\#2}}
800   {\ifiscs{\#1}{\etl@getlistindex\expandafter\csvgloop,\#1}{\#2}}
801     {\etl@getlistindex}\csvgloop,\#1}{\#2}}}}

```

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

```

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

```

`\ifincsvglist` A robust command with a star form.

`\xifincsvglist` The same with `\protected@edef`.

```

812 \etl@if@inlist\ifincsvglist{,\#
813 \etl@xif@inlist\xifincsvglist{,\#
814 \undef\etl@if@inlist
815 \undef\etl@xif@inlist

```

`\interval` `\interval` will expand to the number of the interval of *(number)* into the *(sorted comma separated)*

```

816 \newcommand\interval[2]{\romannumeral-'\q%
817   \ExpandNext{\avoidvoid[\csvgloop!{\#2}]{\csvgloop+[\etl@do@interval{\#1}]{\#2}}}
818 \def\etl@do@interval#1[#2]#3{\ifdim#1p@<#3p@ \@swap{\breakloop{\#2}}\fi}

```

`\locinterplin`

```

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

```



```

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

```

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

*Undocumented option **etoolbox**.*

```

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

```

Revision history

3.1 2009-09-30

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

3.0 2009-09-09

Definition of \DeclareStringFilter, \FE@modifiers and \etl@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 \etl@ifnotblank)
- \ifsinglechar for optimisation

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

Test on pdf ε TeX and Xe ε TeX.

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`, `\ifintokslst` and `\xifincsvlist`.

Addition of `\forcsvloop`, `\forlistloop` and `\fortokslloop`.

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

Optimization of `\getlistindex` and `\getcsvglistindex`

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 L^AT_EX 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

<code>\#</code>	272, 274
<code>\/</code>	826
<code>\@car</code>	49
<code>\@cdr</code>	48
<code>\@expandnext</code>	103, 105, 107
<code>\@firstofone</code>	43, 357, 465, 471, 679, 682
<code>\@firstoftwo</code>	44, 361, 486
<code>\@gobblescape</code>	87, 87, 258, 308, 310, 525, 531, 666, 756
<code>\@gobblespace</code>	86, <u>86</u>
<code>\@gobbletwo</code>	679, 682
<code>\@ifchar</code>	486, <u>486</u> , 521, 684, 685, 687
<code>\@ifdefinable</code>	306, 657
<code>\@intmax</code>	60
<code>\@let@token</code>	495, 497– 499, 502, 506, 507, 509–511, 514, 517
<code>\@secondoftwo</code>	45
<code>\@sptoken</code>	495, 499, 511
<code>\@swap</code>	88, <u>88</u> , 89, 96, 100, 109, 243, 602, 606, 608, 693, 694, 818
<code>\@swaparg</code>	90, 90, 104, 108, 112, 599, 605, 670, 778
<code>\@swaptlast</code>	91, <u>91</u> , 672–675
<code>\@swaptwo</code>	92, <u>92</u> , 651
<code>\@thirdofthree</code>	643, 670, 697
<code>\@undefined</code>	121, 558, 573, 590
<code>\@xifstrequal</code>	269, 270

Numbers

<code>\1</code>	488, 495, 504, 506, 507, 516
<code>\2</code>	489, 493, 495, 497, 498, 504, 506, 507, 509, 510, 516

A

<code>\AfterAssignment</code>	471
<code>\AfterGroup</code>	464
<code>\aftergroup</code>	470, 481–484
<code>\AfterGroup*</code>	464
<code>\aftergroup@def</code>	476, <u>476</u> , 485, 772–775, 796
<code>\appto</code>	721, 725
<code>\avoidvoid</code>	<u>161</u> , 817
<code>\avoidvoid*</code>	161
<code>\avoidvoidcs</code>	<u>171</u>
<code>\avoidvoidcs*</code>	<u>171</u>

B

<code>\basic_string_filter</code>	402
<code>\bgroup</code>	497, 506, 509, 536, 561
<code>\breakloop</code>	693, 694, <u>704</u> , 818

C

<code>\catcode</code>	11, 13, 717, 735, 743, 767, 776, 791, 797, 826
<code>\char</code>	155
<code>\cmdlabel</code>	640
<code>\csvdel</code>	<u>779</u>
<code>\csvedel</code>	<u>779</u>
<code>\csvgdel</code>	<u>779</u>

<code>\csvlistadd</code>	<u>721</u>
<code>\csvlistead</code>	<u>721</u>
<code>\csvlistgadd</code>	<u>721</u>
<code>\csvlistxadd</code>	<u>721</u>
<code>\csvloop</code>	716, 729, 730, 799– 801, 817, 820, 825, 828–831, 833–836
<code>\csvtolist</code>	<u>729</u> , 747
<code>\csvtolistadd</code>	746
<code>\csvxdel</code>	<u>779</u>
<code>\csxdef</code>	467

D

<code>\deblank</code>	255, 298
<code>\DeclareCmdListParser</code>	<u>657</u> , 716, 718, 720
<code>\DeclareOption</code>	839
<code>\DeclareStringFilter</code>	<u>306</u>
<code>\detokenize</code>	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, 679, 682, 849, 851, 854
<code>\detokenizeChars</code>	<u>238</u> , 530
<code>\dimen</code>	152
<code>\dimexpr</code>	832
<code>\docsvlist</code>	285

E

<code>\eappto</code>	747, 749
<code>\egroup</code>	498, 507, 510, 537, 562
<code>\etex@let@primitive</code>	477, 480
<code>\ettl@voidvoidcs</code>	174, 175
<code>\ettl@AfterAssignment</code>	471, 472
<code>\ettl@AfterGroup</code>	465, 466
<code>\ettl@aftergroup@def</code>	485
<code>\ettl@AtEnd</code>	7, 9, 10, 35
<code>\ettl@avoidvoidcs</code>	172–174
<code>\ettl@B@sicFilter</code>	402, 403
<code>\ettl@BasicFilter</code>	402
<code>\ettl@breakloop</code>	613, 617, 619, 620, 676, 677, 704, 705
<code>\ettl@car</code>	49, 224
<code>\ettl@carcar</code>	54, 186, 198
<code>\ettl@cdr</code>	48, 58, 184, 192, 196, 208, 218, 233, 234, 491
<code>\ettl@char</code>	<u>58</u> , 503, 515
<code>\ettl@convert@list</code>	731, 732
<code>\ettl@convertlist</code>	729–731, 738, 739, 741, 742
<code>\ettl@csname</code>	57, 179, 180, 225, 228, 233, 236
<code>\ettl@csvtolistadd</code>	746, 747
<code>\ettl@deblank</code>	255, 256, 259
<code>\ettl@deblank@i</code>	256, 257
<code>\ettl@declarestrfilter</code>	307, 309
<code>\ettl@defcmdparser</code>	658, 659, 665–667
<code>\ettl@do@csvtolist</code>	729, 730, 736
<code>\ettl@do@detokenChars</code>	239, 240

<code>\ettl@do@interval</code>	817, 818	<code>\ettl@ifincharlist</code>	615, 617
<code>\ettl@do@listtocsv</code>	738–740	<code>\ettl@ifinlist</code>	<u>802</u>
<code>\ettl@do@tokstolist</code>	741, 742, 744	<code>\ettl@ifintokslist</code>	611, 613
<code>\ettl@dosinglelist</code> ..	238, 611, 615, <u>618</u>	<code>\ettl@ifnch</code>	<u>494</u>
<code>\ettl@dosinglelist@loop</code>	619, 621	<code>\ettl@ifnextchar</code>	486, 487, <u>487</u>
<code>\ettl@else</code> ..	38, 79, 81, 83, 85, 356, 360	<code>\ettl@ifnspc</code>	499, 511, <u>517</u>
<code>\ettl@expandafthree</code> ..	41, 138, 258, 642	<code>\ettl@ifntk</code>	<u>505</u>
<code>\ettl@expandaftwo</code>	40, 856	<code>\ettl@ifstrnum</code>	299, 300, 302
<code>\ettl@FE@chardot</code>	421, 422	<code>\ettl@ifx</code>	178, <u>178</u> , 624, 630, 635
<code>\ettl@FE@charequal</code>	416, 419	<code>\ettl@ifxsingle</code>	185, 189
<code>\ettl@FE@ifstar</code>	411, 414	<code>\ettl@intmax</code>	
<code>\ettl@FE@modifiers</code>	427, 431		<u>60</u> , 314, 318, 340, 341, 343, 344, 394
<code>\ettl@FE@testopt</code>	406, 409	<code>\ettl@listedel</code>	
<code>\ettl@fi</code> .	37, 79, 81, 83, 85, 342, 380, 675		... 770, 771, 777, 781, 782, 785, 786
<code>\ettl@firstsp@ce</code>	55, 56	<code>\ettl@locinterplin</code>	825, 827
<code>\ettl@firstspace</code>		<code>\ettl@lst@count</code>	672, 695
	55, 182, 190, 194, 206, 226, 229, 235	<code>\ettl@lst@getitem</code>	673, 692
<code>\ettl@for@nested</code>	706, 708–715	<code>\ettl@lst@modif</code>	691, <u>696</u>
<code>\ettl@forloop</code> ...	685, 686, 688–690, 707	<code>\ettl@lst@opt</code>	700–702
<code>\ettl@fter</code>	464,	<code>\ettl@meaning</code>	132
	467–470, 473–475, 478, 479, 482, 484	<code>\ettl@meaningcs</code>	122
<code>\ettl@futdef</code>	518–521	<code>\ettl@naturall@p</code> ...	600, 601, 604, 607
<code>\ettl@futdef@filt</code>	553, 554	<code>\ettl@naturalloop</code>	597, 598
<code>\ettl@futur@def</code>	522, 523	<code>\ettl@nbk</code> ...	56, 73, <u>73</u> , 78, 80, 83, 84,
<code>\ettl@futur@def@collect</code>			146, 167, 183, 189, 191, 195, 207,
	... 525, 531, 539, 549, 564, 568, 578		216, 217, 222, 223, 227, 232, 250,
<code>\ettl@futur@def@filt</code>	<u>552</u>		294, 296, 297, 299, 302, 383, 385,
<code>\ettl@futures@f</code>	520, 557		387, 391, 393, 394, 398, 401, 403,
<code>\ettl@futures@f@append</code> ...	573, 576, 594		611, 615, 618, 623, 626, 634, 637,
<code>\ettl@futures@f@CheckSpecials</code>	582, 589		653, 671, 695, 721, 722, 724, 727,
<code>\ettl@futures@f@collect</code>	569		732, 761, 763, 789, 840, 851, 856, 858
<code>\ettl@futures@f@collectall</code>	575	<code>\ettl@nbk@cat</code>	<u>78</u>
<code>\ettl@futures@f@end</code>		<code>\ettl@nbk@else</code>	74, 377
	... 561, 562, 565, 574, 584	<code>\ettl@nbk@IF</code>	<u>84</u>
<code>\ettl@futures@f@expand</code>	585, 587	<code>\ettl@nbk@if</code>	<u>82</u>
<code>\ettl@futures@f@loop</code>		<code>\ettl@nbk@ifx</code>	<u>80</u>
	... 559, 566, 588, 595, 596	<code>\ettl@ney</code>	<u>75</u>
<code>\ettl@futures@f@space</code>	563, 567, 577	<code>\ettl@numberminus</code>	295–297
<code>\ettl@futures@f@test</code>	559, 560, 595	<code>\ettl@numberspace</code>	298, 299
<code>\ettl@futures@f@try@expand</code>	574, 576, 579	<code>\ettl@only@pdfTeX</code>	62, 63
<code>\ettl@futuredef</code>	518, 533	<code>\ettl@onlypdfTeX</code> .	<u>62</u> , 249, 263, 267, 279
<code>\ettl@futuredef@collect</code>	542	<code>\ettl@or</code>	
<code>\ettl@futuredef@collectall</code>	550		39, 321, 326, 331, 336, 339, 672–674
<code>\ettl@futuredef@end</code> .	536, 537, 540, 547	<code>\ettl@params@meaning</code>	
<code>\ettl@futuredef@filt</code> 135, 139, 141–143, 856
	... 543, 544, 555, 570, 571	<code>\ettl@protectspace</code>	241–243
<code>\ettl@futuredef@loop</code> .	534, 541, 546, 551	<code>\ettl@Rem@veInList</code>	754, 757
<code>\ettl@futuredef@space</code>	538, 548	<code>\ettl@Remove</code>	352, <u>363</u> , 378
<code>\ettl@futuredef@test</code> .	534, 535, 546, 551	<code>\ettl@Remove@loop</code>	369, <u>370</u>
<code>\ettl@getlistindex</code>		<code>\ettl@RemoveInList</code>	
	... 787, 788, 790, 793–795, 799–801		... <u>750</u> , 768, 769, 778–780, 783, 784
<code>\ettl@getsingletlist</code>		<code>\ettl@setresult</code>	428, 624,
	... 429, 624, 627, 630, 635, 638, 641		627, 630, 635, 638, 653, 761, 763, 789
<code>\ettl@gobble@relax</code>	766	<code>\ettl@singlelist@loop</code>	641, 642, 644
<code>\ettl@if@inlist</code>	802, 812, 814	<code>\ettl@singlelist@result</code>	643, 651
<code>\ettl@ifchar</code>		<code>\ettl@sptoken</code>	89, 538, 563
	179, 179, 197, 224, 429, 617, 627, 638	<code>\ettl@strfilt</code>	318, 319,
<code>\ettl@ifchardot</code>			322, 324, 325, 327, 329, 330, 332,
	... 321, 326, 331, 336, 339, 342, <u>421</u>		334, 335, 337, 338, <u>346</u> , 354, 356, 360
<code>\ettl@ifd@f</code>	144, 145, 160	<code>\ettl@strfilt@count</code>	318, 394
<code>\ettl@ifdef</code>	144, 150–160	<code>\ettl@strfilt@endby</code>	332, 335, 390
<code>\ettl@ifdefempty</code>	162, 169, 172	<code>\ettl@strfilt@endbyeq</code>	334, 392
<code>\ettl@ifdefined</code> ..	36, 201, 211, 280, 289	<code>\ettl@strfilt@equal</code> .	319, 322, 325, 382
<code>\ettl@ifdefmacro</code>	849–852, 854	<code>\ettl@strfilt@equaleq</code>	324, 384
<code>\ettl@ifdefvoid</code>	163, 166, 173	<code>\ettl@strfilt@instr</code>	337, 338, 395

<code>\ettl@strfilt@mod</code>	312–317, <u>320</u>	<code>\gettokslistcount</code>	<u>623</u>
<code>\ettl@strfilt@REMOVE</code>	340, 341, <u>353</u>	<code>\gettokslistindex</code>	<u>623</u>
<code>\ettl@strfilt@remove</code>	356, 400	<code>\gettokslisttoken</code>	<u>623</u>
<code>\ettl@strfilt@REPLACE</code>	343, 344, <u>359</u>	<code>\globcount</code>	706
<code>\ettl@strfilt@replace</code>	360, 401		
<code>\ettl@strfilt@start</code>	327, 330, 386	H	
<code>\ettl@strfilt@starteq</code>	329, 388	<code>\helper_macros</code>	<u>36</u>
<code>\ettl@stringify</code>	258, <u>258</u>		
<code>\ettl@strip@meaning</code>	130	I	
<code>\ettl@superg@bbble</code>	433, 434	<code>\ifblank</code>	64, 163, 253, 840
<code>\ettl@supergobbble</code> ...	430, 431, 433, <u>433</u>	<code>\ifcase</code>	321, 592, 672
<code>\ettl@supergobbble@end</code> ...	440, 452, 463	<code>\ifcat</code>	79,
<code>\ettl@supergobbble@loop</code>	440, 442, 446, 450, 463		180, 536–538, 543, 561–563, 570, 590
<code>\ettl@supergobbble@next</code>	444, 462	<code>\ifcharlower</code>	<u>271</u>
<code>\ettl@supergobbbleheight</code>	450	<code>\ifcharupper</code>	<u>271</u>
<code>\ettl@tempa</code>	807, 808, 810, 811	<code>\ifcsname</code> 175, 287, 301, 447, 592, 842, 846	
<code>\ettl@thefontname</code>	116, 117	<code>\ifdefblankspace</code>	<u>144</u>
<code>\ettl@tokstolistadd</code>	748, 749	<code>\ifdefchar</code>	<u>144</u>
<code>\ettl@voidv@id</code>	164, 165	<code>\ifdefcount</code>	<u>144</u> , 654
<code>\ettl@voidvoid</code>	162–164	<code>\ifdefdimen</code>	<u>144</u>
<code>\ettl@x</code>	525–527, 529,	<code>\ifdefmacro</code>	125, 848
	532, 536–538, 541, 546, 551, 561–	<code>\ifdefmathchar</code>	<u>144</u>
	563, 566, 570, 580, 584, 587, 588, 596	<code>\ifdefmuskip</code>	<u>144</u>
<code>\ettl@xif@inlist</code>	804, 813, 815	<code>\ifdefparam</code>	855
<code>\ettl@xifinlist</code>	805, 809	<code>\ifdefskip</code>	<u>144</u>
<code>\ettl@xifnch</code>	494	<code>\ifdefthechar</code>	<u>144</u>
<code>\ettl@xifntk</code>	505	<code>\ifdeftheletter</code>	<u>144</u>
<code>\ettl@y</code> .	526–530, 558, 573, 581, 583–587	<code>\ifdeftoks</code>	<u>144</u>
<code>\expandafter@cmds</code>	95, 97, 99	<code>\ifdim</code>	590, 818, 830
<code>\ExpandAftercmds</code>	<u>109</u> ,	<code>\ifempty</code> ...	162, 169, 242, <u>244</u> , 393, 652
	352, 427, 428, 624, 627, 630, 635, 638	<code>\ifeof</code>	592
<code>\expandaftercmds</code>	<u>93</u>	<code>\iffirstchar</code>	<u>221</u>
<code>\ExpandNext</code>	<u>110</u> , 113, 601, 817	<code>\iffontchar</code>	592
<code>\expandnext</code>	<u>101</u> , 110, 111, 307	<code>\ifhbox</code>	591
<code>\ExpandNextTwo</code>	<u>113</u> , 430, 651, 825	<code>\ifincharlist</code>	<u>610</u>
<code>\expandnexttwo</code>	<u>109</u> , 224	<code>\ifincsvlist</code>	<u>812</u>
		<code>\ifinner</code>	591
F		<code>\ifintokslist</code>	589, <u>610</u>
<code>\FE@ifchar</code>	424, <u>424</u>	<code>\ifiscs</code>	165, <u>232</u> , 691, 794, 800
<code>\FE@ifcharequal</code>	<u>416</u>	<code>\iflowercase</code>	<u>276</u>
<code>\FE@ifstar</code>	161, 171, 411, <u>411</u> , 729, 738, 741	<code>\ifnotblank</code>	250, 807
<code>\FE@modifiers</code>	311, 425, <u>425</u> , 696	<code>\ifnotempty</code>	246,
<code>\FE@testopt</code>	164, 174,		385, 389, 623, 626, 629, 634, 637, 759
	338, 341, 344, 406, <u>406</u> , 433, 597, 731	<code>\ifnum</code> ...	203, 213, 261, 265, 272, 274,
<code>\font</code>	116, 117		278, 281, 290, 356, 360, 377, 396,
<code>\forcsvloop</code>	<u>716</u>		397, 432, 438, 439, 443, 590, 599,
<code>\forlistloop</code>	<u>717</u>		605, 693, 694, 703, 820, 828, 829, 860
<code>\forloops</code>	<u>706</u>	<code>\ifodd</code>	590
<code>\fortoksloop</code>	<u>720</u>	<code>\ifOneChar</code>	<u>201</u> , 240
<code>\futured@f_L(starred)</code>	<u>557</u>	<code>\ifOneCharWithBlanks</code>	<u>211</u>
<code>\futuredef</code>	<u>518</u>	<code>\ifOneToken</code>	<u>190</u> , 426, 490
<code>\futuredef_L(not_Lstarred)</code>	<u>533</u>	<code>\ifsinglechar</code>	<u>194</u> , 424
<code>\futuredef*</code>	<u>518</u>	<code>\ifsingletoken</code> ...	94, 98, 102, 106, <u>182</u>
<code>\futurelet</code>	502,	<code>\ifstrcmp</code>	<u>260</u> , 275, 276
	514, 517, 541, 546, 551, 566, 588, 596	<code>\ifstrdigit</code>	<u>280</u>
G		<code>\ifstrequal</code>	263, 269
<code>\gappto</code>	722, 728	<code>\ifstrmatch</code>	<u>277</u>
<code>\getcharlistcount</code>	430, <u>623</u>	<code>\ifstrnum</code>	<u>289</u> , 703
<code>\getcharlistindex</code>	<u>623</u>	<code>\ifuppercase</code>	<u>275</u>
<code>\getcharlisttoken</code>	<u>623</u>	<code>\ifvbox</code>	591
<code>\getcsvglistindex</code>	<u>798</u>	<code>\ifvmode</code>	591
<code>\getlistindex</code>	<u>787</u>	<code>\ifvoid</code>	591
		<code>\interval</code>	<u>816</u> , 825

L		R	
<code>\lccode</code>	274	<code>\rmn@firstoftwo</code>	46, 146
<code>\letLtxMacro</code>	478, 481	<code>\rmn@secondoftwo</code>	47, 146
<code>\listdel</code>	767	<code>\romannumeral</code>	.. 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, 666, 756, 789, 816, 819
<code>\listedel</code>	767		
<code>\listgdel</code>	767		
<code>\listloop</code>	717, 738, 739, 793–795		
<code>\listtocsv</code>	738		
<code>\listxdel</code>	767		
<code>\locinterplin</code>	819		
<code>\lowercase</code>	276		
M		S	
<code>\mathchar</code>	156	<code>\show</code>	118
<code>\meaning</code>	122, 123, 130–132, 136, 140, 148, 849, 856	<code>\showcs</code>	118
<code>\meaningcs</code>	119, 854	<code>\skip</code>	153
<code>\muskip</code>	154	<code>\strip@meaning</code>	116, 124, 124, 167, 170
		<code>\strip@meaningcs</code>	124, 126
		<code>\strip@pt</code>	832
N		T	
<code>\naturalloop</code>	597	<code>\test_and_result_macros</code>	382
<code>\noexpandafter</code>	115, 147	<code>\texteuro</code>	31, 33
<code>\noexpandcs</code>	114, 148, 661–664, 754	<code>\thefontname</code>	116
<code>\number</code>	394, 443, 447, 448, 467, 468, 473, 478, 606, 623, 626, 629, 652, 672, 683, 703, 761, 763, 829–831, 833–836	<code>\toks</code>	151
<code>\numexpr</code>	356, 360, 377, 394, 396, 397, 438, 439, 443, 447, 448, 467, 468, 473, 478, 599, 605, 606, 652, 672, 683, 693, 694, 761, 763, 820, 829–831, 833–836, 860	<code>\toksdel</code>	783
		<code>\toksdel</code>	783
		<code>\toksgdel</code>	783
		<code>\toksloop</code>	720, 741, 742
		<code>\tokstolist</code>	741, 749
		<code>\tokstolistadd</code>	748
		<code>\toksxdel</code>	783
P		U	
<code>\p@</code>	818, 830, 833, 834	<code>\uccode</code>	272
<code>\PackageError</code>	65, 821	<code>\uppercase</code>	275
<code>\parameters@meaning</code>	133, 133, 134		
<code>\parameters@meaningcs</code>	133, 137, 138, 858		
<code>\pdfmatch</code>	201, 203, 210, 211, 213, 278–281, 288–290, 305		
<code>\pdfstrcmp</code>	29, 249, 261, 263, 265, 267		
<code>\protected@edef</code>	252, 269, 723, 726, 777, 810		
<code>\protectspace</code>	239, 241		
<code>\providerobustcmd</code>	721–723, 726		
X			
		<code>\xifblank</code>	251
		<code>\xifempty</code>	248
		<code>\xifincsvlist</code>	812
		<code>\xifstrcmp</code>	248, 264
		<code>\xifstreempty</code>	249
		<code>\xifstrequal</code>	267, 268