The etextools[1] macros

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

FC
florent.chervet@free.fr

Version 3.1415926
07 December 2010

Contents

LaTeX code

Abstract

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

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

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

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

---

[1] **etextools**: `CTAN:macros/latex/contrib/etextools`

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

Introduction

1 Motivation

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

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

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

2 Purely Expandable macros

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

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

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

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

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

3 The example file

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

4 Requirements

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

5 Acknowledgements – Thank You !

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

6 A note for package writers

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

Happy ε-TEXing

---

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

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

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

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

etextools

List of Commands Provided

All User Commands

1 ▶ General Helper Macros

`\@gobblespace{⟨ code ⟩}`

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

`\@gobblescape`

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

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

`\@swap{⟨ token1 ⟩}{⟨ token2 ⟩}`

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

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

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

```
\@swap{ }\meaning          ⟶  blank space
\expandafter\@swap\expandafter{⟨codeA⟩}{⟨codeB⟩}
```

will expand ⟨*codeA*⟩ once and the put ⟨*codeB*⟩ just before

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

`\@swaparg{⟨ code ⟩}{⟨ command ⟩}`

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

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

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

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

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

`\@swaptwo{⟨ token1 ⟩}{⟨ token2 ⟩}`

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

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

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

2 ▶ Expansion control

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

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

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

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

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

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

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

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

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

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

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

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

```
\expandafter\ifblank\expandafter {\foo}{⟨true part⟩}{⟨false part⟩}
```

With `\expandnext` you'll just have to say:

```
\expandnext\ifblank{\foo}{⟨true part⟩}{⟨false part⟩}
```

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

```
\expandnext{\def\test}{\csname name\endcsname}
```

and it is exactly:

```
\edef\test{\expandafter\noexpand\csname name\endcsname}
```

and also exactly:

```
\expandafter\def\expandafter\test\expandafter{\csname name\endcsname}
```

Genauer gesagt: `\meaning\test = macro:->\name`

`\expandnext` can be used for macros with optional arguments:

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

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

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

```
\expandnext\expandnext{\def\test}{\csname name\endcsname}
```

and it is exactly:

```
\edef\test{\expandafter\expandafter\expandafter\noexpand\csname name\endcsname}
```

and also exactly:

```
\expandafter\expandafter\expandafter\def\expandafter\expandafter\expandafter\test
              \expandafter\expandafter\expandafter{\csname name\endcsname}
```

Genauer gesagt: `\meaning\test = macro:->` ⟨***the meaning of*** **`\name`**⟩

`\expandnext` is an `\expandafter` saver !

Now observe the following game :

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

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

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

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

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

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

$$\texttt{\textbackslash expandnexttwo: \#1\#2\#3} \longrightarrow \texttt{\textbackslash expandnext \{\textbackslash expandnext\{\#1\}} \underbrace{\texttt{\{\#2\}}}_{\substack{\text{expanded}\\ \textbf{once after}}} \texttt{\}} \underbrace{\texttt{\{\#3\}}}_{\substack{\text{expanded}\\ \textbf{once first}}}$$

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

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

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

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

- the *very first* token in ⟨***control sequences***⟩ is **totally expanded**
- ⟨***code***⟩ may be arbitrarily code (not necessarily a single token)

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

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

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

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

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

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

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

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

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

$$\texttt{\textbackslash ExpandNextTwo: \#1\#2\#3} \longrightarrow \texttt{\textbackslash ExpandNext \{\textbackslash ExpandNext\{\#1\}} \underbrace{\texttt{\{\#2\}}}_{\substack{\text{totally}\\ \text{expanded}\\ \textbf{after}}} \texttt{\}} \underbrace{\texttt{\{\#3\}}}_{\substack{\text{totally}\\ \text{expanded}\\ \textbf{first}}}$$

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

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

\noexpandcs{⟨ *csname* ⟩}

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

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

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

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

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

\noexpandafter

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

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

3 ▶ Meaning of control sequences – determining their type.

`\thefontname`

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

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

`\showcs`{⟨ *csname* ⟩}

`\showcs` does `\show` on the named control sequence.

`\showthecs`{⟨ *csname* ⟩}

`\showthecs` does `\showthe` on the named control sequence.

`\meaningcs`{⟨ *csname* ⟩}

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

`\strip@meaning`{⟨ *cs-token* ⟩}

`\strip@meaningcs`{⟨ *csname* ⟩}

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

i) without the prefix ‘`macro:#1#2...->)`‘ if ⟨*cs-token*⟩ is a macro

ii) integrally if ⟨*cs-token*⟩ is defined and is not a macro

iii) expands to an empty string if ⟨*cs-token*⟩ is undefined.

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

`\parameters@meaning`{⟨ *cs-token* ⟩}

`\parameters@meaningcs`{⟨ *csname* ⟩}

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

to summarize

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`\avoidvoid` is purely expandable and uses `\FE@ifstar` and `\FE@testopt`: if the mandatory argument is a ⟨*string*⟩ equal to ‘ $\star_{12}$ ‘ 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.**

---

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

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

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

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

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

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

Finally, clever !

```
\protected\def\test{␣}
```

`\avoidvoidcs [other]{test}`	will expand `other` : `\test` is void
`\avoidvoidcs*[other]{test}`	will expand `\test` : `\test` is not `\@empty`
`\avoidvoidcs [other]\test`	will expand `\` : control space, which is not void
`\avoidvoidcs*[other]\test`	will expand `\` : control space, which is not void

---

4 ▶ Single tokens/single characters

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

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

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

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

1) The **`\ifx`** test: is the standard test for tokens:
   ⟨*tokenA*⟩ is equal to ⟨*tokenB*⟩ if: `\ifx`⟨*tokenA*⟩⟨*tokenB*⟩ returns **true**

   The **`\ifx`** test is implemented in `\ettl@ifx`.

2) The **character test** is a bit more sophisticated and works as follow:
   i) if ⟨*tokenA*⟩ and ⟨*tokenB*⟩ have the same category code (tested with an unexpandable `\ifcat`):
      ⟨*tokenA*⟩ is equal to ⟨*tokenB*⟩ if: `\ifx`⟨*tokenA*⟩⟨*tokenB*⟩ returns **true**
   ii) otherwise:
      ⟨*tokenA*⟩ is equal to ⟨*tokenB*⟩ if: `\if\noexpand`⟨*tokenB*⟩`\string`⟨*tokenA*⟩ returns **true**

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

4•2 Basic test macros

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

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

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

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

now clever !

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

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

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

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

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

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

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

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

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

`\ifsinglechar{A}{A}`	will expand ⟨***true***⟩
`\ifsinglechar{A}{␣A}`	will expand ⟨***false***⟩
`\ifsinglechar{␣}{␣}`	will expand ⟨***true***⟩ no matter the number of spaces
`\ifsinglechar{\ifx}{\ifx\test\relax YES\else NO\fi}`	will expand ⟨***false***⟩
`\ifsinglechar{}{`⟨ *whatever* ⟩`}`	will expand ⟨***true***⟩ only if ⟨***whatever***⟩ is empty
`\ifsinglechar{\scantokens}{\scantokens}`	will expand ⟨***true***⟩

```
\begingroup\catcode`\: 13\global\def\test{:}\endgroup  \catcode`\: 12
```

`\expandnext\ifsinglechar{\test}{:}`	will expand ⟨***true***⟩

now clever!

```
\catcode`\: \active  \let:=\fi
\def\test{\ifsinglechar:}
\let:=\else
```

`\test:{`⟨*true*⟩`}{`⟨*false*⟩`}`	will expand ⟨***true***⟩
`\test\fi{`⟨*true*⟩`}{`⟨*false*⟩`}`	will expand ⟨***false***⟩
`\test\else{`⟨*true*⟩`}{`⟨*false*⟩`}`	will expand ⟨***false***⟩

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

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

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

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

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

`\ifOneChar` is used in `\detokenizeChars`

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

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

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

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

`\iffirsttoken`{⟨ *string1* ⟩}{⟨ *string2* ⟩}{⟨ *true* ⟩}{⟨ *false* ⟩}

`\iffirsttoken` compares the very first tokens of each ⟨*string*⟩. The comparison is done using **`\ifx`** and the macro is fully expandable. Neither ⟨*string1*⟩ nor ⟨*string2*⟩ is expanded before comparison. Example:

```
\iffirsttoken \relax{\relax\textbf{hello world}}{begins with \relax}{begins wit
```

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

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

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

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

Alternatively, you may use the `\ifstrmatch` test.

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

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

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

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

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

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

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

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

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

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

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

`\protectspace`{⟨ *code* ⟩}

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

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

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

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

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

5 ▶ Characters and Strings

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

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

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

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

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

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

pdfTEX

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

```
\def\x{\@empty}\def\y{}
```

`\xifempty{\x\y}` ⟨*true*⟩⟨*false*⟩ will expand ⟨*true*⟩

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

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

\ifnotempty reverses the test of `\ifempty`.

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

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

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

\ifnotblank reverses the test of `\ifblank`.

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

\deblank{⟨ *string* ⟩}

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

An application is for the normalisation of comma separated lists:

```
\csvloop*[\deblank]{   item1 , item2    ,   item3
      , item4   , item5     ,item6   ,
                  item7 ,  item8}%
```

will normalize the list:

```
{item1,item2,item3,item4,item5,item6,item7,item8}
```

This construction is purely expandable:

`\edef\result{\csvloop [\deblank]{...}}`

will normalize the list and assign the result to the replacement text of `\result`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The commands are robust.

---

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

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

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

pdfTEX **`\ifstrmatch`** is based on the `\pdfmatch` primitive that implements POSIX-regex. You can test the last character of a string in a purely expandable way by:

`\ifstrmatch{[*]$}{⟨string⟩}`

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

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

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

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

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

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

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

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

in other words:

```
\edef\resultA{\number⟨string⟩}
\edef\resultB{\deblank{⟨string⟩}}
\ifx\resultA\resultB      is true if {⟨string⟩} is a positive integer
```

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

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

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

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

To tell all the truth, `\ifstrnum` expands ⟨*true*⟩ even if digits have a category code$\neq 12$ whereas `\number` throws an error or stops. But if numbers and minus signs are of category 12 (more than recommended after all...) **`\ifstrnum` is a purely expandable test to check if it is possible to expand `\number`** (or `\romannumeral`) **onto ⟨*string*⟩** (but `\ifstrnum` does not expand ⟨*string*⟩.)

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

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

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

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

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

Equality is `\catcode` dependent.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6 ▶ Fully expandable macros with options and modifiers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For example, to define a ‘ **+** ‘ variant:

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

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

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

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

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

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

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

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

`\FE@modifiers` works as follow:

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

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

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

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

`\ettl@supergobble`[⟨*code*⟩]{$\langle\mathbf{n}\rangle$}{$\langle\mathbf{N}\rangle$}{⟨*tok*$_1$⟩}...{⟨*tok*$_n$⟩}{⟨***TOK***$_{\mathbf{n+1}}$⟩}{⟨*tok*$_{n+2}$⟩}...{⟨*tok*$_{\mathbf{N}}$⟩}

`\ettl@supergobble`{$\langle\mathbf{n}\rangle$}{$\langle\mathbf{N}\rangle$} will:

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

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

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

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

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

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

If you're interested in what `\ettl@supergobble` does when $\langle\mathbf{N}\rangle\leq 0$: it does nothing!

7 ▶ More options with \futuredef : vectorized \futurelet:

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

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

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

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

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

\@char@testopt{⟨ *code* ⟩}{⟨ *single token* ⟩}{⟨ *default option* ⟩}{⟨ *single token* ⟩}

\@char@testopt is a generalization of LATEX's\@testopt that may be used as follow:

```
\newcommand\SpecialMacro{\@char@testopt\GeneralMacro({default})}
 with\GeneralMacro a macro accepting optional parenthesis:
    \def\GeneralMacro(#1){...}
 or:
\newcommand\SpecialMacro{\@char@testopt\GeneralMacro<{default}>}
 with\GeneralMacro a macro accepting optional brackets:
    \def\GeneralMacro<#1>{...}
```

NB: \@char@testopt uses \ettl@ifnextchar and therefore, the category code of single **characters** is not taken into account.

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

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

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

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

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

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

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

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

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

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

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

A token is in the ⟨*list of allowed tokens*⟩ if it can be found in this list using the `character-test`. This means that if `\relax` is in the ⟨***list of allowed tokens***⟩, then it will be appended to `\macro` (if encountered) and if ‘ $\$_3$ ‘ is in the ⟨***list of allowed tokens***⟩, any ‘ **$** ‘ character will be appended to `\macro` (if encountered) no matter of its category code. If you really absolutely need the `\ifx`-test, you shall use `\futuredef=`[10].

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**[11] 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
```

---

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

[11] **keycommand**: `CTAN:macros/latex/contrib/keycommand`

`\switch` is purely expandable. See **boolexpr**[12] for more information on `\switch`.

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

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

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

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=*`.

---

[12] **boolexpr**: `CTAN:macros/latex/contrib/boolexpr`

8 ▶ Define control sequences through groups

`\AfterGroup{⟨ code ⟩}`

`\AfterGroup⋆{⟨ code ⟩}`

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

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

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

`\AfterAssignment{⟨ code ⟩}`

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

`\aftergroup@def{⟨ command ⟩}`

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

The idea of `\aftergroup@def` is to keep a control sequence though `\endgroup` or '}'. This is done by redefining it after the group. `\aftergroup@def` is based on **letltxmacro**[13] and on `\AfterGroup` just defined. Therefore, `\aftergroup@def` works with commands with optional arguments declared with LATEX's `\newcommand`, with robust commands from **etoolbox**-`\newrobustcmd` and with LATEX's robust commands (`\DeclareRobustCommand`).

```
{ \newcommand\test[2][default]{ #1 and #2 }
    \aftergroup@def\test
}
\test[option]{mandatory}  is defined outside the group - but NOT globally
```

---

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

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:

$$\underbrace{\texttt{\textbackslash MyCommand}\ \{\underbrace{\texttt{\textbackslash MyCommand}\ \{\underbrace{\texttt{\textbackslash MyCommand}\ \{\langle\textit{argument}\rangle\}}_{\text{expanded first}}\}}_{\text{expanded second}}\}}_{\text{expanded last}}$$

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

`\MyCommand:macro [#1]#2#3 ->` **Something to do with `#1` `#2` and `#3`**

Where:

- `#1:` is the current index of the loop 1, 2, 3 until to n
- `#2:` is the original ⟨*argument*⟩
- `#3:` is the result of the recursion :ie `\do{\do{\do{\do{`⟨*argument*⟩`}}}}` f.ex. in loop of index 4.

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

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

Another example is given in the `\ExpandNext` section.

9•2 Lists of single tokens / characters

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

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

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

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

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

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

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

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

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

Therefore, `\ifincharlist` behaves as follow:

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

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

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

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

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

It is possible to say:

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

Please, refer to the examples...

This feature is extensively used in `\FE@modifiers`.

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

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

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

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

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

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

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

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

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

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

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

The parameters of the *test-macro* include:

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

Definition of `\ettl@getsinglelist` worth a close look!

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

Unlike `\getlistindex`, `\getcsvlistindex` etc., `\gettokslistindex`, `\gettokslistcount` and `\gettokslisttoken` have no star form nor optional parameter. This is because we might be able to test:

`\gettokslistindex{*}{`⟨*list of single tokens*⟩`}` or `\gettokslistindex{[}{`⟨*list of single tokens*⟩`}`

and `\FE@ifstar` or `\FE@testopt` don't allow this.

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

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

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

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

9•3 The General Command-List Parser Constructor

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

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

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

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

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

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

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

`\breakloop``{`⟨ *code* ⟩`}`

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

$$/_8E_{11}n_{11}d_{11}\S_7L_{11}i_{11}s_{11}t_{11}/_8$$

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

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

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

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

---

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

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

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

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

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

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

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

You may then use the following syntaxes:

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

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

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

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

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

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

iii) otherwise, no expansion of the list occurs

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

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

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

9•4 Loops into lists

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

All of them are defined using `\DeclareCmdListParser`.

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

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

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

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

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

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

Examples:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`\toksloop!{abcdef}` ⟶ 6

Spaces are not counted, however...

`\forcsvloop`{⟨ *csvlistmacro or* `item,item,item`⟩}`\do`{⟨...**#1**...⟩}

`\forlistloop`{⟨ *Listmacro or expanded List* ⟩}`\do`{⟨...**#1**...⟩}

`\fortoksloop`{⟨ *tokenslistmacro or list of single tokens* ⟩}`\do`{⟨...**#1**...⟩}

`\forcsvloop+`{⟨ *csvlistmacro or* `item,item,item`⟩}`\do`{⟨...**#1**=*index*...**#2**=*element*...⟩}

`\forlistloop+`{⟨ *Listmacro or expanded List* ⟩}`\do`{⟨...**#1**=*index*...**#2**=*element*...⟩}

`\fortoksloop+`{⟨ *tokenslistmacro or list of single tokens* ⟩}`\do`{⟨...**#1**=*index*...**#2**=*element*...⟩}

`\forcsvloop* (+)` {⟨`item,item,item`⟩}`\do`{⟨...**#1**...⟩}

`\forlistloop* (+)` {⟨ *expanded List* ⟩}`\do`{⟨...**#1**...⟩}

`\fortoksloop* (+)` {⟨ *list of single tokens* ⟩}`\do`{⟨...**#1**...⟩}

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

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

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

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

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

9•5 Adding elements to csv lists

etextools provides a facility to add items to a csvlist.

`\csvlistadd`{⟨ *csvListmacro* ⟩}{⟨ *item* ⟩}

`\csvlistgadd`{⟨ *csvListmacro* ⟩}{⟨ *item* ⟩}

`\csvlisteadd`{⟨ *csvListmacro* ⟩}{⟨ *item* ⟩}

`\csvlistxadd`{⟨ *csvListmacro* ⟩}{⟨ *item* ⟩}

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

These macros are robust.

9•6 Converting lists

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

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

But there is another way, may be easier:

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

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

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

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

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

```
    \csvtolist[\myList]{one,two,three}
is the same as:
    \edef\myList{\csvtolist{one,two,three}}
```

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

N.B.: the items are not expanded.

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

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

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

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

```
    \tokstolist[\myList]{\alpha\beta\gamma\ifeof+*$}
is the same as:
    \edef\myList{\tokstolist{\alpha\beta\gamma\ifeof+*$}}
    \meaning\myList:   macro:->\alpha|3\beta|3\gamma|3\ifeof|3+|3*|3$|3
```

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

N.B.: the items are not expanded.

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

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

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

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

```
\listtocsv[\csvList]\etbList                    is the same as:
\edef\csvList{\listtocsv\etbList}
```

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

N.B.: the items are not expanded.

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

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

\csvtolistadd acts similarly but both arguments are mandatory:

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

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

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

\tokstolistadd acts similarly but both arguments are mandatory.

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

9•7 Test if an element is in a list

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

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

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

`\ifincsvlist*`{⟨ *item* ⟩}{⟨`item,item,item`⟩}{⟨ *true* ⟩}{⟨ *false*⟩}

`\xifincsvlist*`{⟨ *item* ⟩}{⟨`item,item,item`⟩}{⟨ *true* ⟩}{⟨ *false*⟩}

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

9•8 Removing elements from lists

9.8.1 etoolbox lists

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

`\listdel`[⟨ *deleted* n *times* ⟩]{⟨ *Listmacro* ⟩}{⟨ *item* ⟩}

`\listgdel`[⟨ *deleted* n *times* ⟩]{⟨ *Listmacro* ⟩}{⟨ *item* ⟩}

`\listedel`[⟨ *deleted* n *times* ⟩]{⟨ *Listmacro* ⟩}{⟨ *item* ⟩}

`\listxdel`[⟨ *deleted* n *times* ⟩]{⟨ *Listmacro* ⟩}{⟨ *item* ⟩}

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

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

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

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

9.8.2 csv-lists

`\csvdel`[⟨ *deleted* n *times* ⟩]{⟨ *csvlistmacro* ⟩}{⟨ *item* ⟩}

`\csvgdel`[⟨ *deleted* n *times* ⟩]{⟨ *csvlistmacro* ⟩}{⟨ *item* ⟩}

`\csvedel`[⟨ *deleted* n *times* ⟩]{⟨ *csvlistmacro* ⟩}{⟨ *item* ⟩}

`\csvxdel`[⟨ *deleted* n *times* ⟩]{⟨ *csvlistmacro* ⟩}{⟨ *item* ⟩}

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

9.8.3 Lists of single tokens

`\toksdel`[⟨ *deleted* n *times* ⟩]{⟨ *tokslistmacro* ⟩}{⟨ *item* ⟩}

`\toksgdel`[⟨ *deleted* n *times* ⟩]{⟨ *tokslistmacro* ⟩}{⟨ *item* ⟩}

`\toksedel`[⟨ *deleted* n *times* ⟩]{⟨ *tokslistmacro* ⟩}{⟨ *item* ⟩}

`\toksxdel`[⟨ *deleted* n *times* ⟩]{⟨ *tokslistmacro* ⟩}{⟨ *item* ⟩}

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

9•9 Index of an element in a list

9.9.1 etoolbox-lists

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

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

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

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

9•10 Arithmetic: lists of numbers

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

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

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

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

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

\locinterplin{⟨X⟩}{⟨X_i⟩}{⟨Y_i⟩}

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

and expands to the local linear interpolation Y:

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

X_i and Y_i must have the same number of elements.

LATEX code

Implementation

I•1 Package identification

```
⟨*package⟩
\NeedsTeXFormat{LaTeX2e}[1996/12/01]
\ProvidesPackage{etextools}
  [2010/12/07 v3.1415926 e-TeX more useful tools for LaTeX package writers]
\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`):

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

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

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

Therefore, the choice might not be bad...

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

```
  \TMP@EnsureCode{128}{9}% \texteuro ignore
}
\AtEndOfPackage{\ettl@AtEnd\undef\ettl@AtEnd}
```

I•3 Some “helper” macros

helper macros

```
\let\ettl@ifdefined\ifdefined%\ifdefined% turn to \iffalse to test other implementation o
\long\def\ettl@fi#1\fi{\fi#1}
\long\def\ettl@else#1\else#2\fi{\fi#1}
\long\def\ettl@or#1\or#2\fi{\fi#1}
\def\ettl@expandaftwo{\expandafter\expandafter\expandafter}
\def\ettl@expandafthree{\expandafter\expandafter\expandafter%
                        \expandafter\expandafter\expandafter\expandafter}
\cslet{ettl@1of1}\@firstofone    %% for internal use only
\cslet{ettl@1of2}\@firstoftwo    %% for internal use only
\cslet{ettl@2of2}\@secondoftwo   %% for internal use only
\long\def\rmn@firstoftwo#1#2{\z@#1} %% for romannumeral
\long\def\rmn@secondoftwo#1#2{\z@#2}%% for romannumeral
\long\def\ettl@cdr#1#2\@nil{#2}  %% \@cdr should be a LONG macro
\long\def\ettl@car#1#2\@nil{#1}  %% \@car should be a LONG macro
\long\csdef{ettl@1of3}#1#2#3{#1}
\long\csdef{ettl@2of3}#1#2#3{#2}
\long\csdef{ettl@3of3}#1#2#3{#3}
\long\csdef{ettl@12of3}#1#2#3{{#1}{#2}}
\long\def\ettl@carcar#1#2#3#4{#4}
\long\def\ettl@firstspace#1#2#3{\expandafter\ettl@firstsp@ce\detokenize{#1} \\{#3}{#2}//}
\long\def\ettl@firstsp@ce#1 #2\\{\ettl@nbk#1//}
\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`.

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

`\ettl@intmax`

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

```
\providecommand*\@intmax{2147483647}
\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`).

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

`\ettl@nbk`

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

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

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

```
\long\def\ettl@nbk #1#2/#3#4#5//{#4}
\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`:

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

`\ettl@nbk@cat`

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

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

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

`\ettl@nbk@ifx`

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

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

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

`\ettl@nbk@if`

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

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

```
\long\def\ettl@nbk@if#1#2#3/#4#5#6//%
      {\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`:

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

`\@gobblespace`

```
\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`:

```
\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:

```
\newcommand\@swap[2]{#2#1}
\@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 (**!**):

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

`\@swaplast`

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

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

`\@swaptwo`

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

```
\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 `\expandnext`s. At exit, swap again.

```
\newcommand\expandaftercmds[2]{%
   \ifsingletoken\expandaftercmds{#1}
      {\expandafter@cmds{#2}{\expandafter\expandafter\expandafter}}
      {\expandafter\@swap\expandafter{#2}{#1}}}
\long\def\expandafter@cmds#1#2#3{%
   \ifsingletoken\expandaftercmds{#1}
      {\expandafter@cmds{#3}{\expandafter#2#2}}
      {#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.

```
\newcommand\expandnext[2]{%
   \ifsingletoken\expandnext{#1}
      {\@expandnext{#2}{\expandafter\expandafter\expandafter}}
      {\expandafter\@swaparg\expandafter{#2}{#1}}}
\long\def\@expandnext#1#2#3{%
   \ifsingletoken\expandnext{#1}
      {\@expandnext{#3}{\expandafter#2#2}}
      {#2\@swaparg#2{#3}{#1}}}
```

`\expandnexttwo`

`\ExpandAftercmds`

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

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

`\ExpandNext`

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

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

`\ExpandNextTwo`

```
\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.

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

`\noexpandafter`

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

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

I•5 Meaning of control sequences

`\thefontname`

```
\newcommand\thefontname{\nfss@text{\expandafter\expandafter\expandafter\expandafter\ettl@thefontname
   \expandafter\expandafter\expandafter\meaning
      \expandafter\the\expandafter\font
      \expandafter\string\expandafter(%
      \expandafter\string\the\font\string)}}
\ifcsname T1/cmr/m/n/10\endcsname
   \letcs\ettl@thefontname{T1/cmr/m/n/10}%
\else
   \font\ettl@thefontname=ecrm1000
\fi
```

`\showcs`

`\showcs` shows the meaning of a named control sequence:

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

`\showthecs`

`\showthecs` shows the value of the named register:

```
\providecommand*\showthecs[1]{\expandafter\showthe\csname#1\endcsname}
```

`\meaningcs`

`\meaningcs` expands in one level:

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

`\strip@meaning`

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

`\strip@meaningcs`

The same but for named control sequences:

```
\newcommand*\strip@meaning[1]{\romannumeral\csname\ifdef{#1}%
      {\ifdefmacro{#1}{ettl@strip@meaning}{ettl@meaning}\endcsname#1}{z@\endcsname}}
\providecommand*\strip@meaningcs[1]{\romannumeral\csname\ifcsdef{#1}%
      {\ifcsmacro{#1}{ettl@strip@meaning}{ettl@meaning}%
               \expandafter\endcsname\csname#1\endcsname}
      {z@\endcsname}}
\def\ettl@strip@meaning{\expandafter\expandafter\expandafter\z@% for \romannumeral in cas
   \expandafter\strip@prefix\meaning}
\def\ettl@meaning{\expandafter\z@\meaning}
```

`\parameters@meaning`
`\parameters@meaningcs`

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

```
\providecommand*\parameters@meaning[1]{}
\edef\parameters@meaning#1{\unexpanded{\romannumeral\expandafter
   \expandafter\expandafter\z@\expandafter\ettl@params@meaning%
         \meaning}#1\detokenize{macro:->}/}
\providecommand*\parameters@meaningcs[1]{}
\edef\parameters@meaningcs#1{\unexpanded{\romannumeral\ettl@expandafthree\z@
   \expandafter\expandafter\expandafter\ettl@params@meaning%
         \expandafter\meaning\csname}#1\endcsname\detokenize{macro:->}/}
\edef\ettl@params@meaning{%
   \def\noexpand\ettl@params@meaning\detokenize{macro:}##1\detokenize{->}##2/{##1}%
}\ettl@params@meaning
```

`\ifdefcount`
`\ifdeftoks`
`\ifdefdimen`
`\ifdefskip`
`\ifdefmuskip`
`\ifdefchar`
`\ifdefmathchar`

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

```
\def\ettl@ifdef[#1]{\expandafter\ettl@ifd@f\expandafter{#1}}
\def\ettl@ifd@f#1#2{%
   \csdef{ettl@ifdef#2}##1#1##2/End§Meaning/{\ettl@nbk##2//\rmn@firstoftwo\rmn@secondoftw
   \csedef{ifdef#2}##1{\noexpand\romannumeral\noexpandafter%
```

```
      \noexpandcs{ettl@ifdef#2}\noexpand\meaning##1#1/End§Meaning/}%//{##2}{##3}//}
}
\ettl@ifdef[\string\count]{count}               % defines   \def\ifdefcount
\ettl@ifdef[\string\toks]{toks}                 %           \def\ifdeftoks
\ettl@ifdef[\string\dimen]{dimen}               %           \def\ifdefdimen
\ettl@ifdef[\string\skip]{skip}                 %           \def\ifdefskip
\ettl@ifdef[\string\muskip]{muskip}             %           \def\ifdefmuskip
\ettl@ifdef[\string\char]{char}                 %           \def\ifdefchar
\ettl@ifdef[\string\mathchar]{mathchar}         %           \def\ifdefmathchar
\ettl@ifdef[\detokenize{blank space}]{blankspace}%          \def\ifdefblankspace
\ettl@ifdef[\detokenize{the character}]{thechar}%           \def\ifdefthechar
\ettl@ifdef[\detokenize{the letter}]{theletter} %           \def\ifdeftheletter
\undef\ettl@ifdef\undef\ettl@ifd@f
```

`\avoidvoid`
`\avoidvoid*`

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

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

and the helper macros:

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

`\avoidvoidcs`
`\avoidvoidcs*`

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

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

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

I•6 Single tokens / single characters

`\ettl@ifx`

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

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

`\ettl@ifchar`

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

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

`\ifsingletoken`

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

```
\newcommand\ifsingletoken[2]{\romannumeral\csname rmn@\ettl@firstspace{#2}
```

```
   {\ettl@nbk#1#2//{second}{\ifcat $\detokenize{#1#2}$first\else\ifx#1#2first\else second
   {\ifcat $\detokenize\expandafter{\ettl@cdr#2\@nil}$%
         \expandafter\ettl@ifxsingle
   \else\expandafter\ettl@carcar
   \fi{#1}{#2}{first}{second}}%
   oftwo\endcsname}
\def\ettl@ifxsingle#1#2#3#4{\ettl@nbk#1//{\ifx#1#2#3\else#4\fi}{#4}//}
```

`\iffirsttoken`

`\iffirsttoken` tests if **`#1`** and **`#2`** begins with the same token (the `\ifx`-test is used):

```
\providecommand\iffirsttoken[2]{\romannumeral\csname rmn@%
   \ettl@nbk#2//%
      {\ettl@nbk#1//%
         {\expandnexttwo\ettl@ifx{\ettl@car#2\@nil}{\ettl@car#1\@nil}{first}{second}}
         {\ifcat $\detokenize{#1}$secondoftwo\ettl@csname\fi
         \ettl@firstspace{#2}{first}{second}}//}%
      {\ettl@nbk#1//%
         {\ifcat $\detokenize{#2}$secondoftwo\ettl@csname\fi
         \ettl@firstspace{#1}{first}{second}}
         {\ifcat $\detokenize{#1#2}$first\else second\fi}}//%
   oftwo\endcsname}
```

`\ifOneToken`

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

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

`\ifsinglechar`

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

```
\long\def\ifsinglechar#1#2{\romannumeral\csname rmn@\ettl@firstspace{#2}
   {\ettl@nbk#2//{second}{\ifcat $\detokenize{#1#2}$first\else\ifx#1#2first\else second\f
   {\ifcat $\detokenize\expandafter{\ettl@cdr#2\@nil}$%
         \expandafter\ettl@ifchar
      \else\expandafter\ettl@carcar
      \fi{#1}{#2}{first}{second}}%
   oftwo\endcsname}
```

`\ifOneChar`

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

```
\ettl@ifdefined\pdfmatch
\newcommand\ifOneChar[1]{\romannumeral\csname rmn@%
      \ifnum\pdfmatch{\detokenize{^.$}}{\detokenize{#1}}=1 first\else second\fi
      oftwo\endcsname}
\else
\newcommand\ifOneChar[1]{\romannumeral\csname rmn@\ettl@firstspace{#1}
   {\ettl@nbk#1//{second}{\ifcat $\detokenize{#1}$second\else first\fi}//}
   {\ifcat $\expandafter\ettl@cdr\detokenize{#1}\@nil$%
    first\else second\fi}oftwo\endcsname}
\fi%\pdfmatch
```

`\ifOneCharWithBlanks`

```
\ettl@ifdefined\pdfmatch
\newcommand\ifOneCharWithBlanks[1]{\romannumeral\csname rmn@%
      \ifnum\pdfmatch{\detokenize{^[[:space:]]*[^[:space:]][[:space:]]*$}}{\detokenize{#1
      first\else second\fi oftwo\endcsname}
\else
\newcommand\ifOneCharWithBlanks[1]{\romannumeral\csname rmn@\ettl@nbk#1//%
      {\expandafter\expandafter\expandafter\ettl@nbk
            \expandafter\ettl@cdr\detokenize{#1}\@nil//{second}{first}//}%
      {second}//oftwo\endcsname}
\fi
```

`\iffirstchar`

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

```
\newcommand\iffirstchar[2]{\romannumeral\csname rmn@%
   \ettl@nbk#2//%
      {\ettl@nbk#1//%
         {\expandnexttwo\ettl@ifchar{\ettl@car#2\@nil}{\ettl@car#1\@nil}{first}{second}}
         {\ifcat $\detokenize{#1}$secondoftwo\ettl@csname\fi
         \ettl@firstspace{#2}{first}{second}}//}%
      {\ettl@nbk#1//%
         {\ifcat $\detokenize{#2}$secondoftwo\ettl@csname\fi
         \ettl@firstspace{#1}{first}{second}}
         {\ifcat $\detokenize{#1#2}$first\else second\fi}}//%
   oftwo\endcsname}
```

`\ifiscs`

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

```
\newcommand\ifiscs[1]{\romannumeral\csname rmn@\ettl@nbk#1//%
   {\ifcat $\expandafter\ettl@cdr\detokenize{#1}\@nil$secondoftwo\ettl@csname\fi
    \ifcat $\detokenize\expandafter{\ettl@cdr#1\@nil}$%
      \expandafter\ettl@firstspace
    \else secondoftwo\ettl@csname\fi{#1}{second}{first}}
   {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:

```
\newcommand\detokenizeChars[1]{\expandafter\ettl@dosinglelist
   \expandafter\ettl@do@detokenChars\expandafter{\romannumeral\protectspace{\z@#1}}}
\long\def\ettl@do@detokenChars#1{\ifOneChar{#1}\detokenize\unexpanded{#1}}
```

`\protectspace`

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

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

I•7 Character and Strings

`\ifempty`

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

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

`\ifnotempty`

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

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

`\xifempty`

pdfTEX

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

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

`\ifnotblank`

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

`ifnotblank`**`ifnotblank`**

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

`\xifblank`

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

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

`\deblank`

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

```
\newcommand\deblank[1]{\romannumeral\ettl@deblank#1/ /}
\long\def\ettl@deblank#1 /{\ettl@deblank@i#1/}
\long\def\ettl@deblank@i#1/#2{\z@#1}
```

`\ettl@stringify`

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

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

`\ifstrcmp`
pdfTEX

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

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

`\xifstrcmp`
pdfTEX

The macro is based on the `\pdfstrcmp` primitive.

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

`\xifstrequal`

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

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

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

`\ifuppercase`

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

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

`\iflowercase`

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

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

`\ifstrmatch`
pdfTEX

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

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

`\ifstrdigit`

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

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

`\ifstrnum`

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

```
\ettl@ifdefined\pdfmatch
\newcommand\ifstrnum[1]{\romannumeral\csname rmn@\ifnum\pdfmatch
  {\detokenize{^([[:space:]]*-?)*+[[:digit:]]+[[:space:]]*$}}{\detokenize{#1}}=1 %
  first\else second\fi oftwo\endcsname}
\else
\newcommand\ifstrnum[1]{\romannumeral\csname rmn@\ettl@nbk#1//%
      {\expandafter\ettl@numberminus\detokenize{#1}-/End§String/}{second}//oftwo\endcs
\long\def\ettl@numberminus#1-#2/End§String/{\ettl@nbk#2//%
      {\ettl@nbk#1//{second}{\ettl@numberminus#2/End§String/}//}%
      {\expandafter\expandafter\expandafter\ettl@numberspace\deblank{#1} /End§String/}
\long\def\ettl@numberspace#1 #2/End§String/{\ettl@nbk#2//{second}{\ettl@ifstrnum#1/End§St
\long\def\ettl@ifstrnum#1#2/End§String/{%
  \ifcsname ettl@number#1\endcsname% #1 detokenized before, ok
    \ettl@nbk#2//{\ettl@ifstrnum#2/End§String/}{first}//%
  \else second%
  \fi}
\fi%\pdfmatch
```

`\DeclareStringFilter`

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

```
\newrobustcmd\DeclareStringFilter[3][\global]{\@ifdefinable#2%
  {\expandnext\ettl@declarestrfilter%
        {\csname\@gobblescape#2\detokenize{->"#3"}\endcsname}{#1}{#2}{#3}}}
\newcommand\ettl@declarestrfilter[4]{%
  #2\csdef{\@gobblescape#1}##1#4##2/End§String/{##1/##2}% This the FILTER
  #2\long\def#3##1{\FE@modifiers{=<>?-+!}{##1}
    {\ettl@strfilt@mod 0{{#4}{}{#1}[1]}}%=
    {\ettl@strfilt@mod 1{{#4}{}{#1}[1]}}%<
    {\ettl@strfilt@mod 2{{#4}{}{#1}[\ettl@intmax]}}%>
    {\ettl@strfilt@mod 3{{#4}{}{#1}}}%?
    {\ettl@strfilt@mod 4{{#4}{}{#1}}}%-
    {\ettl@strfilt@mod 5{{#4}{}{#1}}}%+
    {\ettl@strfilt\ettl@strfilt@count{#4}{}{#1}[\ettl@intmax]}%!
    {\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:

```
\def\ettl@strfilt@mod #1#2#3{%
  \ifcase#1 \ettl@or\ettl@ifchardot{#3}%
          {\ettl@strfilt\ettl@strfilt@equal#2}
          {\FE@ifcharequal{#3}%
            {\ettl@strfilt\ettl@strfilt@equaleq#2}%
            {\ettl@strfilt\ettl@strfilt@equal#2}}%
  \or\ettl@or\ettl@ifchardot{#3}%
          {\ettl@strfilt\ettl@strfilt@start#2}%
          {\FE@ifcharequal{#3}
            {\ettl@strfilt\ettl@strfilt@starteq#2}%
            {\ettl@strfilt\ettl@strfilt@start#2}}%
  \or\ettl@or\ettl@ifchardot{#3}%
          {\ettl@strfilt\ettl@strfilt@endby#2}%
          {\FE@ifcharequal{#3}
            {\ettl@strfilt\ettl@strfilt@endbyeq#2}%
            {\ettl@strfilt\ettl@strfilt@endby#2}}%
  \or\ettl@or\ettl@ifchardot{#3}%
```

```
               {\ettl@strfilt\ettl@strfilt@instr#2[1]}
               {\FE@testopt{#3}{\ettl@strfilt\ettl@strfilt@instr#2}{1}}%
   \or\ettl@or\ettl@ifchardot{#3}%
               {\ettl@strfilt@REMOVE{#2}[\ettl@intmax]}%
               {\FE@testopt{#3}{\ettl@strfilt@REMOVE{#2}}{\ettl@intmax}}%
   \or\ettl@fi\ettl@ifchardot{#3}%
               {\ettl@strfilt@REPLACE#2[\ettl@intmax]}%
               {\FE@testopt{#3}{\ettl@strfilt@REPLACE#2}{\ettl@intmax}}%
   \fi}
```

`\ettl@strfilt`

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

```
\long\def\ettl@strfilt#1#2#3#4[#5]#6{% % #1 = test macro
% #2 = substr
% #3 = replacement
% #4 = filter macro
% #5 = number of times
% #6 = user-given string
     \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`:

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

`ettl@strfilt@REPLACE`

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

```
\long\def\ettl@strfilt@REPLACE #1#2#3#4[#5]#6#7{%
   \ifnum\numexpr#5>0 \ettl@else\ettl@strfilt\ettl@strfilt@replace{#1}{#7}{#3}[{#5}]{#6}%
   \else\expandafter\@firstoftwo%
   \fi}
```

`\ettl@Remove`

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

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

`\ettl@Remove@loop`

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

```
\long\def\ettl@Remove@loop#1/#2//#3/End§String/#4[#5]#6{%
% #1 = str before filter
% #2 = str after filter
% #3 = substr to remove
% #4 = REPLACEMENT
% #5 = iterindex
% #6 = filter macro
   \ifnum\numexpr#5>0 \ettl@nbk@else#2//%
            {\ettl@Remove #1#4#2/End§String/{#3}{#4}[{#5}]{#6}}
            {{#1}{#4#2}{#3}{#5}}//%
   \else\ettl@fi{#1}{#4#2}{#3}{#5}%
   \fi}
```

`est and result macros`

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

```
\long\def\ettl@strfilt@equal    #1#2#3#4{\csname @%
      \ettl@nbk#3//{\ettl@nbk#1#2//{second}{first}//}{second}//oftwo\endcsname}
```

```
\long\def\ettl@strfilt@equaleq #1#2#3#4{\csname @%
      \ettl@nbk#3//{\ifnotempty{#1#2}{second}{first}}{second}//oftwo\endcsname}
\long\def\ettl@strfilt@start   #1#2#3#4{\csname @%
      \ettl@nbk#1//{second}{first}//oftwo\endcsname}
\long\def\ettl@strfilt@starteq #1#2#3#4{\csname @%
      \ifnotempty{#1}{second}{first}oftwo\endcsname}
\long\def\ettl@strfilt@endby   #1#2#3#4{\csname @%
      \ettl@nbk#3//{first}{second}//oftwo\endcsname}
\long\def\ettl@strfilt@endbyeq #1#2#3#4{\csname @%
      \ettl@nbk#3//{\ifempty{#2}{first}{second}}{second}//oftwo\endcsname}
\long\def\ettl@strfilt@count   #1#2#3#4{\number\numexpr\ettl@intmax-(#4)-\ettl@nbk#3//01/
\long\def\ettl@strfilt@instr   #1#2#3#4{\csname @%
      \ifnum\numexpr#4>0 second%
      \else\ifnum\numexpr#4<0 first%
      \else\ettl@nbk#3//{first}{second}//%
      \fi\fi oftwo\endcsname}
\long\def\ettl@strfilt@remove  #1#2#3#4{#1#2}
\long\def\ettl@strfilt@replace #1#2#3#4{#1\ettl@nbk#3//{#2}{}//}
```

I•8 Purely expandable macros with options

basic string filter

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

```
\long\def\ettl@BasicFilter#1#2#3/End§String/{\expandafter\ettl@B@sicFilter #1#3//#2/End§S
\long\def\ettl@B@sicFilter#1/#2//#3/End§String/{@\ettl@nbk#3//%
      {\ifcat $\detokenize{#1#2}$first\else second\fi}
      {second}//oftwo}
```

\FE@testopt

Purely expandable \@testopt-like test:

```
\newcommand\FE@testopt[3]{\ettl@FE@testopt#1/[/%
               {#2#1}%
               {#2[{#3}]{#1}}}%]
\long\def\ettl@FE@testopt#1[#2/#3#{\csname @\ifcat $\detokenize{#1#2}$%
   first\else second\fi oftwo\endcsname}
```

\FE@ifstar

Purely expandable \@ifstar-like test:

```
\newcommand\FE@ifstar[3]{\ettl@FE@ifstar#1/*/%
               {#2}%
               {#3{#1}}}
\long\def\ettl@FE@ifstar#1*#2/#3#{\csname @\ifcat $\detokenize{#1#2}$%
   first\else second\fi oftwo\endcsname}
```

\FE@ifcharequal

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

```
\newcommand\FE@ifcharequal[3]{\ettl@FE@charequal#1/=/%
               {#2}%
               {#3{#1}}}
\long\def\ettl@FE@charequal#1=#2/#3#{\csname @\ifcat $\detokenize{#1#2}$%
   first\else second\fi oftwo\endcsname}
```

\ettl@ifchardot

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

```
\newcommand\ettl@ifchardot[1]{\ettl@FE@chardot#1/./}
\long\def\ettl@FE@chardot#1.#2/#3#{\csname @\ifcat $\detokenize{#1#2}$%
   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}:

```
\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 Characte*: USAGE:
`\FE@modifiers{`⟨*Allowed Characters*⟩`}{#1}{\MacroA}...{\MacroZ}{\NormalMacro}`:

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

`\ettl@supergobble`

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

```
\newcommand\ettl@supergobble[1]{\FE@testopt{#1}\ettl@superg@bble{}}
\long\def\ettl@superg@bble[#1]#2#3{%
% #1 = commands to put after the list (optional)
% #2 = number to gobble first
% #3 = total number of items
   \ifnum\numexpr#3>0
      \ifnum\numexpr#3-(#2)=0
         \ettl@supergobble@loop{#3+2}0{\ettl@supergobble@end{}{}}%
      \else
         \expandafter\ettl@supergobble@loop\expandafter{%
            \number\numexpr\ifnum\numexpr#2*(#2-(#3))>0 #3+1\else#2+2\fi}{#3+2}%
                           {\ettl@supergobble@next{}{#1}}%
   \fi\fi}
\long\def\ettl@supergobble@loop#1#2#3{%
   \ifcsname ettl@supergobble\number\numexpr#1\endcsname
      \csname ettl@supergobble\number\numexpr#1\endcsname
         {#3{#2-(#1)-1}}%
   \else\ettl@supergobbleeight{\ettl@supergobble@loop{#1-8}{#2-8}{#3}}%
   \fi}
\long\def\ettl@supergobble@end#1#2#3{\fi\fi\fi#1#2}
\long\csdef{ettl@supergobbleeight}#1\fi#2#3#4#5#6#7#8#9{\fi#1}
\long\csdef{ettl@supergobble7}#1#2\fi#3#4#5#6#7#8#9{#1}
\long\csdef{ettl@supergobble6}#1#2\fi#3#4#5#6#7#8{#1}
\long\csdef{ettl@supergobble5}#1#2\fi#3#4#5#6#7{#1}
\long\csdef{ettl@supergobble4}#1#2\fi#3#4#5#6{#1}
\long\csdef{ettl@supergobble3}#1#2\fi#3#4#5{#1}
\long\csdef{ettl@supergobble2}#1#2\fi#3#4{#1}
\long\csdef{ettl@supergobble1}#1#2\fi#3{#1}
\long\csdef{ettl@supergobble0}#1#2\fi{#1}
\long\def\ettl@supergobble@next#1#2#3#4{\fi
   \ettl@supergobble@loop{#3}0{\ettl@supergobble@end{#4}{#2}}}
```

I•9 Define control sequence through groups

`\AfterGroup`
`\AfterGroup*`

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

```
\newcount\ettl@fter
\newrobustcmd\AfterGroup{\@ifstar{\ettl@AfterGroup\@firstofone}{\ettl@AfterGroup\unexpand
\newrobustcmd\ettl@AfterGroup[2]{%
   \csxdef{ettl@fterGroup\number\numexpr\the\ettl@fter+1}%
      {\global\csundef{ettl@fterGroup\number\numexpr\the\ettl@fter+1}#1{#2}}%
   \global\advance\ettl@fter\@ne
   \expandafter\aftergroup\csname ettl@fterGroup\the\ettl@fter\endcsname}
```

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

```
\newrobustcmd\AfterAssignment{\@ifstar{\ettl@AfterAssignment\@firstofone}{\ettl@AfterAssi
\newrobustcmd\ettl@AfterAssignment[2]{%
   \csedef{ettl@afterassignment@hook\number\numexpr\the\ettl@fter}{#1{#2}}%
   \global\advance\ettl@fter\@ne
   \expandafter\afterassignment\csname ettl@afterassignment@hook\the\ettl@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 LATEX's `\DeclareRobustCommand`.

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

```
\newrobustcmd*\aftergroup@def{\@ifstar\aftergroup@defstar\aftergroup@d@f}
\newrobustcmd*\@ftergroup@def[3]{%
   \aftergroup#2\aftergroup#3\aftergroup#1%
   \aftergroup\global \aftergroup\undef \aftergroup#1}
\newrobustcmd*\aftergroup@defstar[1]{%
   \global\expandafter\let\csname ettl@ftergroup@def\the\numexpr\ettl@fter+1\endcsname#1%
   \global\advance\ettl@fter\@ne
   \expandafter\@ftergroup@def\csname ettl@ftergroup@def\the\ettl@fter\endcsname\let#1}
\newrobustcmd*\aftergroup@d@f[1]{%
   \let\etex@let@primitive\let    \def\let{\global\etex@let@primitive}%
   \expandafter\LetLtxMacro\csname ettl@ftergroup@def\the\numexpr\ettl@fter+1\endcsname#1
   \global\advance\ettl@fter\@ne
   \etex@let@primitive\let=\etex@let@primitive
   \expandafter\@ftergroup@def\csname ettl@ftergroup@def\the\ettl@fter\endcsname\LetLtxMa
\let\ettl@aftergroup@def\aftergroup@def
```

I•10 \futuredef

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

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

`\@char@testopt` `\@char@testopt` is a generalisation of `\@testopt` for other pairs of character than [and] to pass optional parameters to macros. It is based on `\ettl@ifnextchar` and therefore on the `character-test`.

```
\newcommand\@char@testopt[4]{\ettl@ifnextchar#2{#1}{#1#2#3#4}}
```

`\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 exit from `\if`...`\fi` conditionnals (even in the case where the macro parameter is `\else`, `\if` or `\fi`...):

```
\newrobustcmd\ettl@ifnextchar[3]{\begingroup
   \long\edef\1##1/##2/##3{##1\endgroup\unexpanded{#2}##3}%
   \long\edef\2##1/##2/##3{##1\endgroup\unexpanded{#3}##3}%
   \ifOneToken{#1}
      {\csname ettl@\ifcat $\expandafter\ettl@cdr\detokenize{#1}\@nil$% OneChar
         xifnch\else xifntk\fi\endcsname{#1}}
      {\2//{}}}
```

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

```
\long\def\ettl@xifnch#1{%
   \ifx#1\@sptoken \def\ettl@xifnch{\ifx\@let@token\@sptoken\1\else\2\fi//{}}%
   \else \def\ettl@xifnch{%
      \ifx\@let@token\bgroup      \2
      \else\ifx\@let@token\egroup \2
      \else\ifx\@let@token\@sptoken \ettl@ifnspace\ettl@xifnch
     \else\ettl@ifnch
      \fi\fi\fi/{#1}/{}}%
   \fi\futurelet\@let@token\ettl@xifnch}
```

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

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

`\ettl@xifntk`
`\ettl@ifntk`

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

```
\long\def\ettl@xifntk#1{%
   \ifx#1\bgroup\def\ettl@xifntk{\ifx\@let@token\bgroup\1\else\2\fi//{}}%
   \else\ifx#1\egroup\def\ettl@xifntk{\ifx\@let@token\egroup\1\else\2\fi//{}}%
   \else\def\ettl@xifntk{%
      \ifx\@let@token\bgroup        \2
      \else\ifx\@let@token\egroup   \2
      \else\ifx\@let@token\@sptoken \ettl@ifnspace\ettl@xifntk%
      \else\ettl@ifntk%
      \fi\fi\fi/{#1}/{}}%
   \fi\futurelet\@let@token\ettl@xifntk}
```

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

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

`\ettl@ifnspace`

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

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

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

This is the scanner.

```
\newrobustcmd*\futuredef{\begingroup\ettl@futdef\ettl@futuredef\detokenize}
\protected\def\ettl@futdef#1#2{\@ifstar%
      {\ettl@futdef\ettl@futured@f#2}
      {\@ifchar={\ettl@futdef#1\unexpanded}
                {\@testopt{\ettl@futur@def#1#2}{}}}}
\long\def\ettl@futur@def#1#2[#3]{%
   \csname ettl@\ifcat $\detokenize{#3}$1\else2\fi of2\endcsname
   {\let \ettl@x \@empty \letcs \ettl@futur@def@collect{\@gobblescape#1@collectall}}%
   {\def \ettl@x {#3}\edef \ettl@y {#2{#3}}}%
   \ifx\ettl@x\ettl@y      \let\ettl@y\@gobble
   \else \ifx#2\unexpanded \let\ettl@y\@gobble
   \else             \def\ettl@y{\edef\ettl@x}%
   \fi\fi\ettl@y{\detokenizeChars{#3}}%
   \letcs\ettl@futur@def@collect{\@gobblescape#1@collect}}%
   \expandafter#1\expandafter#2\expandafter{\ettl@x}}
```

`turedef (not starred)`

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

```
\long\def\ettl@futuredef#1#2#3#4{% #1=detokenize #2=list, #3=macro result, #4=code-next
   \def \ettl@futuredef@loop{\ettl@futuredef@test{}}%
   \long \def \ettl@futuredef@test##1{%
      \ifcat\noexpand\ettl@x\bgroup\ettl@futuredef@end{}\else
      \ifcat\noexpand\ettl@x\egroup\ettl@futuredef@end{}\else
      \ifcat\noexpand\ettl@x\ettl@sptoken\ettl@futuredef@space#1\else
      \ettl@futur@def@collect#1\fi\fi\fi/Ne¤t/{#2}{##1}}%
   \long \def \ettl@futuredef@end##1##2/Ne¤t/##3##4{##2\endgroup\def#3{##4}#4##1}%
   \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:

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

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

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

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

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

`\ettl@futur@def@filt`

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

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

`futured@f (starred)`

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

```
\long\def\ettl@futured@f#1#2#3#4{% #1=detokenize #2=list, #3=macro result, #4=code-next
   \let \ettl@y \@undefined
   \def \ettl@futured@f@loop{\ettl@futured@f@test{}}%
   \long \def \ettl@futured@f@test##1{%
      \ifcat\noexpand\ettl@x\bgroup\ettl@futured@f@end\else
      \ifcat\noexpand\ettl@x\egroup\ettl@futured@f@end\else
      \ifcat\noexpand\ettl@x\ettl@sptoken\ettl@futured@f@space#1\else
      \ettl@futur@def@collect#1\fi\fi\fi/Ne¤t/{##1}{#2}{}}%
   \long \def \ettl@futured@f@end##1/Ne¤t/##2##3##4{##1\endgroup\def#3{##2}#4##4}%
   \futurelet \ettl@x \ettl@futured@f@loop}
\long\def\ettl@futured@f@space#1#2/Ne¤t/#3#4#5 {%
      \ettl@futur@def@collect#1#2/Ne¤t/{#3}{#4}{#5}{ }}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

I•11 Loops and Lists Management

I•11•1 – naturalloop

`\naturalloop`

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

```
\newcommand\naturalloop[1]{\FE@testopt{#1}\ettl@naturalloop{\do}}
\def\ettl@naturalloop[#1]#2#3{%
   \ifnum\numexpr#2>0 \expandafter\@swaparg\expandafter{\romannumeral-`\q#1[0]{#3}{#3}}%
         {\ettl@naturall@@p[{#1}]{#2-1}{0}{#3}}
%             \ExpandNext{\ettl@naturall@@p[{#1}]{#2-1}{1}{#3}}{#1[1]{#3}{#3}}%
   \else\@swap{\unexpanded{#3}}%
   \fi}
\def\ettl@naturall@@p[#1]#2#3#4#5#6\fi{\fi%
   \ifnum\numexpr#2>0 \expandafter\@swaparg\expandafter{\romannumeral-`\q%
      \expandafter\@swap\expandafter{\expandafter[\number\numexpr#3+1]}{#1}{#4}{#5}}%
      {\ettl@naturall@@p[{#1}]{#2-1}{#3+1}{#4}}%
   \else\@swap{\unexpanded{#5}}%
   \fi}
```

I•11•2 – Lists of single tokens

`\ifintokslist`

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

`\ifincharlist`

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

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

`\ettl@dosinglelist`

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

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

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

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

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

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

```
\newcommand\gettokslistindex[2]{\number\ifnotempty{#2}{\ettl@nbk#1//%
   {\ExpandAftercmds{\ettl@setresult 2of3><}{\ettl@getsinglelist{\ettl@ifx{#1}}{#2}}}
   {-1}//}{-1}}
\newcommand\getcharlistindex[2]{\number\ifnotempty{#2}{\ettl@nbk#1//%
   {\ExpandAftercmds{\ettl@setresult 2of3><}{\ettl@getsinglelist{\ettl@ifchar{#1}}{#2}}}
   {-1}//}{-1}}
\newcommand\gettokslistcount[1]{\number\ifnotempty{#1}%
   {\ExpandAftercmds{\ettl@setresult 1of3><}{\ettl@getsinglelist{\ettl@ifx{\\}}{#1}}}
   0}
\newcommand\getcharlistcount[1]{}%
\let\getcharlistcount=\gettokslistcount
\newcommand\gettokslisttoken[2]{\ifnotempty{#2}{\ettl@nbk#1//%
   {\ExpandAftercmds{\ettl@setresult 3of3><}{\ettl@getsinglelist{\ettl@ifx{#1}}{#2}}}
   {}//}{}}
\newcommand\getcharlisttoken[2]{\ifnotempty{#2}{\ettl@nbk#1//%
   {\ExpandAftercmds{\ettl@setresult 3of3><}{\ettl@getsinglelist{\ettl@ifchar{#1}}{#2}}}
   {}//}{}}
```

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

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

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

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

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

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

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

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

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

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

Then `\ettl@setresult` finishes the job:

```
\def\ettl@setresult#1of#2>#3<{\ettl@nbk #3//%
      {\ettl@set@result#1of#2>#3<}
      {\csname ettl@#1of#2\endcsname}//}
\def\ettl@set@result#1of#2>#3<#4{\ifdefcount{#3}
   {#3=\csname ettl@#1of#2\endcsname#4}
   {\expandafter\edef\noexpand#3{\csname ettl@#1of#2\endcsname{#4}}}%
}
```

I•11•3 – General Lists and Loops Constructor

`\DeclareCmdListParser`

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

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

```
\newrobustcmd\DeclareCmdListParser[3][\global]{\@ifdefinable{#2}{\begingroup
      \protected\def\ettl@defcmdparser##1{%
         \edef\ettl@defcmdparser{\endgroup\ettl@defcmdparser
            {#1}{\noexpand#2}{\unexpanded{#3}}
            {\noexpandcs{##1->start}}
            {\noexpandcs{##1->loop}}
            {\noexpandcs{##1->loop+}}
            {\noexpandcs{for##1}}%
         }\ettl@defcmdparser
      }\expandafter\ettl@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.

```
\protected\long\def\ettl@defcmdparser#1#2#3#4#5#6#7{%#1=global,#2=command,#3=sep,#4=start
   #1\long\def#4##1##2[##3]##4{% ##1=case, ##2=expandafter??? , ##3=do, ##4=list
      ##2{##4}% ifiscs or @thirdofthree
         {\expandafter\@swaparg\expandafter{##4}{#4{##1}\@thirdofthree[{##3}]}}
         {\ettl@nbk##4//%
            {\ifcase##1 \ettl@or\@swaplast{\number\numexpr#60{\ettl@lst@count}}#6%
               \or      \ettl@or\@swaplast{#60{\ettl@lst@getitem{##3}}}#6%
               \or      \ettl@or\@swaplast{#5{##3}}#5%
               \or      \ettl@fi\@swaplast{#60{##3}}#6%
               \fi{##4#3//}{\ettl@breakloop{\ifx##10\expandafter\relax\fi}}%
            }{\ettl@breakloop{}}///End§List/}}%
   #1\long\def#5##1##2#3##3##4/##5##6##7/End§List/{%
      \ifcat $\detokenize{##2}$\expandafter\@gobbletwo\fi\@firstofone{##1{##2}}%
      ##6{##1}##3##4//{##6}{##7}/End§List/}
   #1\long\def#6##1##2##3#3##4##5/##6##7##8/End§List/{%
```

```
      \ifcat $\detokenize{##3}$\expandafter\@gobbletwo\fi\@firstofone{##2[##1]{##3}}%
      \expandafter##7\expandafter{\number\numexpr##1+1}{##2}##4##5//{##7}{##8}/End§List/}
   #1\protected\def#7{\@ifchar*%
      {\@ifchar+{\ettl@forloop{\expandafter#2\expandafter*\expandafter+}{[####1]####2}}
                {\ettl@forloop{\expandafter#2\expandafter*}{####1}}}
      {\@ifchar+{\@ifchar*%
                     {\ettl@forloop{\expandafter#2\expandafter*\expandafter+}{[####1]####
                     {\ettl@forloop{\expandafter#2\expandafter+}{[####1]####2}}}
                {\ettl@forloop{\expandafter#2}{####1}}}}
   #1\def#2{\ettl@lst@modif#423\ifiscs}}
```

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

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

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

`\breakloop`

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

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

```
\globcount\ettl@for@nested
\long\def\ettl@forloop#1#2#3\do{%
   \global\advance\ettl@for@nested\@ne\relax
   \csdef{ettl@for@loop\the\ettl@for@nested}{%
      #1\expandafter[\csname ettl@for@do\the\ettl@for@nested\endcsname]{#3}%
      \csundef{ettl@for@do\the\ettl@for@nested}%
      \csundef{ettl@for@loop\the\ettl@for@nested}%
      \global\advance\ettl@for@nested\m@ne\relax}%
   \expandafter\afterassignment\csname ettl@for@loop\the\ettl@for@nested\endcsname
   \long\csdef{ettl@for@do\the\ettl@for@nested}#2%
}% \ettl@for@nested
```

`\csvloop`

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

`\forcsvloop`

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

\listloop
\forlistloop

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

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

\toksloop
\fortoksloop

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

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

\csvlistadd
\csvlistgadd
\csvlisteadd
\csvlistxadd

```
\providerobustcmd\csvlistadd[2]{\ettl@nbk#2//{\appto#1{#2,}}{}//}
\providerobustcmd\csvlistgadd[2]{\ettl@nbk#2//{\gappto#2{#2,}}{}//}
\providerobustcmd\csvlisteadd[2]{\begingroup \protected@edef#1{#2}%
   \expandafter\ettl@nbk#1//{\expandafter\endgroup
      \expandafter\appto\expandafter#1\expandafter{#1,}}\endgroup//}
\providerobustcmd\csvlistxadd[2]{\begingroup \protected@edef#1{#2}%
   \expandafter\ettl@nbk#1//{\expandafter\endgroup
      \expandafter\gappto\expandafter#1\expandafter{#1,}}\endgroup//}
```

\csvtolist

This is the first application of `\csvloop`:

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

\listtocsv

This is the first application of `\listloop`:

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

\tokstolist

This is the first application of `\toksloop`:

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

\csvtolistadd

`\csvtolistadd` is not purely expandable:

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

\tokstolistadd

`\tokstolistadd` is not purely expandable:

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

\ettl@RemoveInList

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

```
\newrobustcmd\ettl@RemoveInList[2]{\begingroup
% #1 = \global  #2 = macro name
   \def\ettl@RemoveInList##1{%
      \edef\ettl@RemoveInList####1####2{%
         \ettl@Rem@veInList{####1}####2\noexpandcs{##1->remove}\noexpandcs{##1->result}%
      }\ettl@RemoveInList{#1}#2%
   }\expandafter\ettl@RemoveInList\expandafter{\romannumeral-`\q\@gobblescape#2}}
\protected\long\def\ettl@Rem@veInList#1#2#3#4#5#6#7#8{%
   \long\def#3[##1]##2#5#8#5##3##4/##5##6##7/End§List/{##6[##1+1]##2#5##3##4//##6##7/End§
   \ifnotempty{#5}%% special case if no separator
      {\long\def#4[##1]#5##2#5#5##3//##4/End§List/{\unexpanded{#1\def#7{##2#5}}%
               \ettl@nbk#6//{\ettl@setresult 1of1>#6<{\number\numexpr##1-1\relax}}{}//}}%
      {\long\def#4[##1]##2//##3/End§List/{\unexpanded{#1\def#7{##2}}%
               \ettl@nbk#6//{\ettl@setresult 1of1>#6<{\number\numexpr##1-1\relax}}{}//}}%
   \long\def#2##1{#3[0]#5##1#5#5#8#5//#3#4/End§List/}%
   \edef#7{\endgroup\expandafter#2\expandafter{#7}}#7}
\def\ettl@gobble@relax#1\relax{}
```

`\listdel`
`\listedel`
`\listgdel`
`\listxdel`

\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:

```
\edef\ettl@restore@catcode{\catcode124 \the\catcode124}% |=124
\catcode`\|=3
\newrobustcmd\listdel[1][]{\ettl@RemoveInList{}\listdel|{#1}}
\newrobustcmd\listgdel[1][]{\ettl@RemoveInList\global\listdel|{#1}}
\newrobustcmd\listedel[1][]{\ettl@listedel{}\listdel|{#1}}
\newrobustcmd\listxdel[1][]{\ettl@listedel\global\listdel|{#1}}
\ettl@restore@catcode\undef\ettl@restore@catcode
\newrobustcmd\ettl@listedel[6]{\begingroup\protected@edef#5{#6}\expandafter\endgroup
   \expandafter\@swaparg\expandafter{#5}{\ettl@RemoveInList#1#2{#3}{#4}#5}}
```

`\csvdel`
`\csvedel`
`\csvgdel`
`\csvxdel`

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

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

`\toksdel`
`\toksedel`
`\toksgdel`
`\toksxdel`

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

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

`\getlistindex`

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

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

```
\newrobustcmd\ettl@getlistindex[6]{% #1=result, #2=\expandafter, #3=loop macro, #4=separa
   \begingroup\long\def\ettl@getlistindex##1#4#6#4##2/End§List/{\endgroup
   \ExpandAftercmds{\ettl@setresult 1of1>#1<}{\ettl@nbk##2//{#3*!{##1}}{-1}//}%
   }#2\ettl@getlistindex#5#4#6#4/End§List/}
\newrobustcmd\getlistindex{\@ifstar
   {\@testopt{\ettl@get@listindex\relax}{}}
   {\@testopt{\ettl@get@listindex\expandafter}{}}}
\begingroup\catcode`\|=3% etb catcode
\protected\long\gdef\ettl@get@listindex#1[#2]#3#4{%
   \ifx#1\relax
      \ettl@getlistindex{#2}{}\listloop|{#4}{#3}%
   \else \ifiscs{#4}
      {\ettl@getlistindex{#2}#1\listloop|{#4}{#3}}
      {\ettl@getlistindex{#2}\listloop|{#4}{#3}}%
   \fi}
\endgroup%\catcode group
```

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

```
\newrobustcmd\getcsvlistindex{\@ifstar
   {\@testopt{\ettl@get@csvlistindex\relax}{}}
   {\@testopt{\ettl@get@csvlistindex\expandafter}{}}}
\protected\long\gdef\ettl@get@csvlistindex#1[#2]#3#4{%
   \ifx#1\relax
      \ettl@getlistindex{#2}{}\csvloop,{#4}{#3}%
   \else \ifiscs{#4}
      {\ettl@getlistindex{#2}#1\csvloop,{#4}{#3}}
      {\ettl@getlistindex{#2}\csvloop,{#4}{#3}}%
   \fi}
```

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

```
\def\ettl@if@inlist#1#2{%#1=macro,#2=separator
\newrobustcmd*#1{\@ifstar{\ettl@ifinlist{#2}{}}{\ettl@ifinlist{#2}\expandafter}}}
\def\ettl@xif@inlist#1#2{%
\newrobustcmd*#1{\@ifstar{\ettl@xifinlist{#2}{}}{\ettl@xifinlist{#2}\expandafter}}}
\protected\long\def\ettl@ifinlist#1#2#3#4{\begingroup
   \def\ettl@tempa##1#1##2#1/End§List/{\endgroup\ifnotblank{##2}%
   }#2\ettl@tempa#2#1#3#1#4#1/End§List/}
\protected\long\def\ettl@xifinlist#1#2#3#4{\begingroup
   \protected@edef\ettl@tempa{\endgroup\ettl@ifinlist{#1}{#2}{#3}{#4}%
   }\ettl@tempa}
```

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

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

```
\ettl@if@inlist\ifincsvlist{,}
\ettl@xif@inlist\xifincsvlist{,}
\undef\ettl@if@inlist
\undef\ettl@xif@inlist
```

`\interval` `\interval` will expand to the number of the interval of ⟨*number*⟩ into the ⟨*sorted comma sepa* *d*

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

`\locinterplin`

```
\newcommand\locinterplin[3]{\romannumeral-‘\q
   \unless\ifnum\numexpr(\csvloop!{#2})-(\csvloop!{#3})=0
      \PackageError{etextools}{Using \string\locinterplin\space the lists in argument 1 a
      must have the same number of elements}
      {You’re in trouble here and I cannot proceed...}
   \fi
   \ExpandNextTwo{\ettl@locinterplin{#1}{#3}{#2}}{\interval{#1}{#2}}{\csvloop!{#2}}}
\begingroup\catcode‘\/ 12%
\gdef\ettl@locinterplin#1#2#3#4#5{%
   \ifnum#4=0 \csvloop[#4]{#2}%
   \else\ifnum#4=#5 \expandafter\csvloop\expandafter[\number\numexpr#5-1]{#2}%
   \else\ifdim#1\p@=\expandafter\csvloop\expandafter[\number\numexpr#4-1]{#3}\p@
      \expandafter\csvloop\expandafter[\number\numexpr#4-1]{#2}%
   \else\strip@pt\dimexpr%
      \expandafter\csvloop\expandafter[\number\numexpr#4-1]{#2}\p@+%
         (#1\p@-\expandafter\csvloop\expandafter[\number\numexpr#4-1]{#3}\p@)*%
         (\expandafter\csvloop\expandafter[\number\numexpr#4-1]{#2}-\csvloop[#4]{#2})/%
         (\expandafter\csvloop\expandafter[\number\numexpr#4-1]{#3}-\csvloop[#4]{#3})\rel
   \fi\fi\fi}
\endgroup% catcode group
```

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

Undocumented option **etoolbox**.

```
\DeclareOption{etoolbox}{%
\renewcommand\ifblank[3]{\ettl@nbk #1//{#2}{#3}//}
\renewcommand\ifdef[1]{\csname @\ifdefined#1first\else second\fi oftwo\endcsname}
\renewcommand\ifcsdef[1]{\csname @\ifcsname#1\endcsname first\else second\fi oftwo\endcsn
\renewcommand\ifundef[1]{\csname @%
   \ifdefined#1\ifx#1\relax first\else second\fi\else first\fi oftwo\endcsname}
\renewcommand\ifcsundef[1]{\csname @%
   \ifcsname#1\endcsname\expandafter\ifx\csname#1\endcsname\relax
      first\else second\fi\else first\fi oftwo\endcsname}
\edef\ifdefmacro#1{\unexpanded{\csname @%
     \expandafter\ettl@ifdefmacro\meaning}#1\detokenize{macro:}/oftwo\endcsname}
\edef\ettl@ifdefmacro{%
   \def\noexpand\ettl@ifdefmacro##1\detokenize{macro:}##2/{\noexpand\ettl@nbk##2//{first}
}\ettl@ifdefmacro
\long\edef\ifcsmacro#1{\unexpanded{\csname @%
   \expandafter\expandafter\expandafter\ettl@ifdefmacro\meaningcs}{#1}\detokenize{macro:}
\renewcommand\ifdefparam[1]{\csname @%
   \ettl@expandaftwo\ettl@nbk\expandafter\ettl@params@meaning\meaning#1///{first}{second}
\renewcommand\ifcsparam[1]{\csname @%
   \expandafter\expandafter\expandafter\ettl@nbk\parameters@meaningcs{#1}//{first}{second
\renewcommand\ifnumcomp[3]{\csname @%
   \ifnum\numexpr#1#2\numexpr#3 first\else second\fi oftwo\endcsname}
}% etoolbox option - not to be used - experimental
\ProcessOptions*\relax

⟨/package⟩
```

Examples

This is the code of `etextools-example.tex` which comes with the package.

```
⟨*example⟩
\ProvidesFile{etextools-examples}
\documentclass[11pt,french,a4paper,oneside]{scrartcl}
\usepackage[latin1]{inputenc}
\usepackage[T1]{fontenc}
\usepackage[american]{babel}
\usepackage{geometry,doc,ltxdockit,txfonts,fancyhdr,stmaryrd,graphicx,enumitem}
\usepackage{etextools}
%
\usepackage{fancyvrb}
%
\makeatletter
\let\org@newif\newif
\def\newif#1{\ifx#1\ifnotempty\else\expandafter\org@newif\fi}
\usepackage{umrand}
\renewcommand\ifne[1]{\csname @\ifcat $\detokenize{#1}$first\else second\fi oftwo\endcsna
\let\newif\org@newif
\font\umranda=umranda \def\Ch{\char'}
%
\def\textvb#1{{\usefont{T1}{txtt}{m}{n}#1}}
\newrobustcmd\thispackage{\xpackage{\spot etextools}\xspace}
\newrobustcmd\xpackage[1]{{\usefont{T1}{lmss}{bx}{n}\db\mbox{#1}}}
\hypersetup{colorlinks,pdfstartview={FitH}}
\geometry{top=1.5cm,bottom=1.2cm,left=2.5cm,right=1cm}
\fancyhf{}
\fancyhead[L]{Examples for the \thispackage package}
\pagestyle{fancy}
\DefineShortVerb{\|}
\catcode`\^^a7 \active\def^^a7{\par\nobreak\vskip-\parskip}
```

```
\DefineVerbatimEnvironment{VerbLines}{Verbatim}
   {gobble=1,commandchars=!(),frame=lines,framesep=6pt,fontfamily=txtt,fontseries=m}
\apptocmd\@list@extra{\parsep\parskip\topsep\z@\itemsep\z@}{}{}
%
\def\smex{\leavevmode\hb@xt@2em{\hfil$\longrightarrow$\hfil}}
\def\FE{\setbox8\hbox{$\m@th\bindnasrepma$}%
         \textcolor{fecc}{\scalebox{2}{$\copy8\mkern-13.5mu\copy8\mkern-13.5mu\copy8$}}}
\def\pdfFE{\setbox8\hbox{$\m@th\bindnasrepma$}%
         \textcolor{fecc}{\llap{\textsf{pdf}\TeX{}\,}\scalebox{2}{$\copy8\mkern-13.5mu\co
\definecolor{fecc}{rgb}{.2,.6,.2}
\definecolor{dg}{rgb}{0.00,0.37,0.00} \newrobustcmd\dg{\color{dg}}   \newrobustcmd\dgbf{\
\definecolor{spot}{rgb}{1.00,0.33,0.00} \newrobustcmd\spot{\color{spot}}
\definecolor{db}{rgb}{0.00,0.00,0.25}   \newrobustcmd\db{\color{db}}
\newrobustcmd\blue{\color{blue}}
\newrobustcmd\nnn{\normalfont\mdseries\upshape}
%
\newrobustcmd\ClearPage{\@ifstar\clearpage{}}
\def\make@macro#1{\string\def\string#1\parameters@meaning#1\string{\strip@meaning#1\strin
\newcommand\preline{\@ifstar{\@preline}{\hrulefill\par\@preline}}
\newcommand\@preline[2][1.5ex]{\noindent\hskip6pt\textvb{\make@macro#2}\par\ifblank{#1}{}
%
\ifdef\pdfstrcmp{\let\ifpdfTeX\iffalse}{\let\ifpdfTeX\iftrue}
%
\newcommand*\test{\@ifstar{\let\fe\pdfFE\testi}{\let\fe\FE\testi}}
\newcommand\testi[1]{%
   \csname test#1\endcsname
   \edef\usercmd{\strip@meaningcs{test#1}}\edef\result{\meaningcs{#1Test}}\noindent
   \begin{tabular}{lp{15cm}}
   \multicolumn{2}{l}{\textcolor{blue}{\llap{\fe\,\smex}\tt \usercmd}} \\[1.5ex]
   \cmd{#1Test}= & \tt\bfseries\result
   \end{tabular}\par\nobreak\hrulefill\null\goodbreak}
%
\begin{document}
\title{\vskip-2cm\thispackage\ examples}
\subtitle{Examples for some macros provided by the \thispackage package}
\author{\small<FC -- December 12, 2010>}
\date{}
%
\newsavebox\helpbox \newsavebox\helpboxx
\newrobustcmd*\mydotleader[2][\z@]{\leavevmode\xleaders\hbox to\dimexpr1.7pt+#1{\hss\rais
\begingroup\let\clearpage\@empty
\setbox\helpbox\hbox to13cm{\hss\lower3cm\vbox to1.8cm{\maketitle\vss}\hss}
\fboxsep\z@
\newrobustcmd\corner[2][\spot\umranda]{\hbox{#1\rlap{\char'115}\char'#2}}
\null\vskip-1cm\hskip-1cm\null\hfil\RandBox {\fbox{\copy\helpbox}}
   font {\umranda} [0pt]
   (\corner{17}) ([\Ch111]) (\corner{14})
   ([\Ch112])          ([\Ch112])
   (\corner{21}) ([\Ch111]) (\corner{11})
\endgroup
%
\tableofcontents\hyperdef{ettlex}{toc}{}
%
\section{\cmd{expandnext} examples}
%
\subsection{Test if the replacement text of  macro is really empty}
%
\def\xx{   }
\def\testexpandnext{%
   \edef\expandnextTest{\string\xx\ is \expandnext\ifempty{\xx}{}{not} empty}
}
\preline\xx
\test{expandnext}
%
\def\xx{}
```

```
\preline\xx
\test{expandnext}
%
\ClearPage*
\subsection{Test if the replacement text of a macro is blank (empty or spaces)}
%
\def\xx{something}
\def\testexpandnext{%
   \edef\expandnextTest{\string\xx\ is \expandnext\ifblank{\xx}{}{not} blank}
}
\preline\xx
\test{expandnext}
%
\def\xx{    }
\preline\xx
\test{expandnext}
%
\section{\cmd{ExpandNext} examples}
%
Example of the main documentation file to reverse the order of the characters in a string
%
\def\swap#1#2{{#2#1}}   \def\do[#1]#2{\swap #2}% \do{abcdef}   -> \swap ab + cdef   -> {b
                                                % \do{{ba}cdef} -> \swap {ba}c + def -> {
%
\edef\result{\naturalloop[\do]{4}{12345}}
\ExpandNext{\def\RESULT}{\naturalloop[\do]{4}{12345}}
%
\begin{VerbLines}
\def\swap#1#2{{#2#1}}   \def\do[#1]#2{\swap #2}
\edef\result{\naturalloop[\do]{4}{12345}}
\meaning\result = !bfseries!meaning!result
\ExpandNext{\def\RESULT}{\naturalloop[\do]{4}{12345}}
\meaning\RESULT = !bfseries!meaning!RESULT
\end{VerbLines}
%
\subsection{Test the parameter string of a macro}
%
The following commands create the filter for the string: "\textvb{[\#1]\#2"}":^^a7
\begin{VerbLines}[commandchars=!()]
\ExpandNext{(!blue\DeclareStringFilter\ParaFilt)}
      {\ExpandAftercmds\@gobblescape{\expandafter\string\csname(!blue[#1]#2)\endcsname}}
\end{VerbLines}
%
\ExpandNext{\DeclareStringFilter\ParaFilt}
      {\ExpandAftercmds\@gobblescape{\expandafter\string\csname[#1]#2\endcsname}}
%
\begin{enumerate}[label=\arabic*)~,noitemsep,nolistsep]
\item |\csname[#1]#2\endcsname| is expanded first
\item Immediately after: |\string|
\item At this stage: |\[#1]#2| (everything in category code other) is no more expandable
\item Then |\ExpandAftercmds| expands |\@gobblescape|
\item |[#1]#2| is no more expandable
\item Then |\ExpandNext| expands its first argument: |\DeclareStringFilter\ParaFilt{[#1]#
\end{enumerate}
%
Remark: |\detokenize| would have doubled the \# characters. Another possibility is to tem
category code of \# to 12 (other):
\begin{VerbLines}[commandchars=!()]
\begingroup\catcode‘\#=12
  (!blue\DeclareStringFilter\ParaFilt{[#1]#2})      !nnn global declaration
\endgroup
\end{VerbLines}
%
\def\macroA#1#2{Something to do with #1 and #2}
\def\macroB[#1]#2{Something to do with #1 and #2}
```

```
%
\preline[]\macroA
\preline*[]\macroB
\begin{Verbatim}[commandchars=!()]
 !blue\ExpandNext{\ParaFilt=.}{\parameters@meaning\macroA}{macro complies with [\#1]\#2}
                                                     !blue{macro does not comply }
\end{Verbatim}
%
\hfill\textvb{\dgbf\ExpandNext{\ParaFilt=.}{\parameters@meaning\macroA}{macro complies wi
                                                     {macro does not comply }}
\begin{Verbatim}[commandchars=!()]
 !blue\ExpandNext{\ParaFilt=.}{\parameters@meaning\macroB}{macro complies with [\#1]\#2}
                                                     !blue{macro does not comply }
\end{Verbatim}
\hfill\textvb{\dgbf\ExpandNext{\ParaFilt=.}{\parameters@meaning\macroB}{macro complies wi
                                                     {macro does not comply }}
\par\hrulefill\par
%
\ClearPage*
\section{Testing characters}
\subsection{\cmd{ifsinglechar} versus \cmd{iffirstchar}}
\def\testifsinglechar{%
   \edef\ifsinglecharTest{\ifsinglechar *{*hello*}{ single star }{ something else }}
}\hrulefill\par
\test{ifsinglechar}
%
\def\testifsinglechar{%
   \edef\ifsinglecharTest{\ifsinglechar *{  *}{ single star }{ something else }}
}\hrulefill\par
\test{ifsinglechar}
%
\def\testifsinglechar{%
   \edef\ifsinglecharTest{\ifsinglechar *{ *  }{ single star }{ something else }}
}\hrulefill\par
\test{ifsinglechar}
{\small Note the space \textbf{after} the star $\uparrow$.}
%
\def\testiffirstchar{%
   \edef\iffirstcharTest{\iffirstchar *{*hello*}{ first char is star }{ something else }}
}\hrulefill\par
\test{iffirstchar}
%
\subsection{Fully Expandable starred macros}
\def\starmacro#1{\FE@ifstar{#1}\starred\notstarred}
\def\starred#1{your "#1" will be processed by the  STAR form}
\def\notstarred#1{your "#1" will be processed by the  NORMAL form}
\def\testFE@ifstar{%
   \edef\FE@ifstarTest{\starmacro{sample text}}}
\preline\starmacro
\preline*\starred
\preline*\notstarred
\test{FE@ifstar}
%
\def\testFE@ifstar{%
   \edef\FE@ifstarTest{\starmacro*{sample text}}}
\hrulefill\par
\test{FE@ifstar}
%
\subsection{Fully Expandable macros with options}
\def\optmacro#1{\FE@testopt{#1}\OPTmacro{Mr.}}
\def\OPTmacro[#1]#2{#1 #2}
\def\testFE@testopt{%
   \edef\FE@testoptTest{\optmacro{Woody Allen}}}
\preline\optmacro
\preline*\OPTmacro
```

```
\test{FE@testopt}
%
\def\testFE@testopt{%
   \edef\FE@testoptTest{\optmacro[Ms.]{Vanessa Paradis}}}
\hrulefill\par
\test{FE@testopt}
%
\ClearPage*
\section{Lists management}
%
\subsection{\cmd{csvloop} and \cmd{csvloop*} examples}
%
\subsubsection{\cmd{makequotes}}
\def\makequotes#1{"#1"\space}
\def\testcsvloop{%
   \edef\csvloopTest{\csvloop*[\makequotes]{hello,world}}
}
\preline\makequotes
\test{csvloop}
%
\subsubsection{\cmd{detokenize}}
\def\testcsvloop{%
   \edef\csvloopTest{\csvloop*[\detokenize]{\un,\deux}}
}\hrulefill\par
\test{csvloop}
%
\subsubsection{\cmd{numexpr}}
\def\mylist{1,2,3,4,5}\def\BySeven#1{$#1\times 7 = \number\numexpr#1*7\relax$\par}
\def\testcsvloop{%
   \edef\csvloopTest{\csvloop[\BySeven]\mylist}}
\preline\mylist
\preline*\BySeven
\test{csvloop}
%
\subsubsection{protected \cmd{textbf}}
\def\testcsvloop{%
   \protected@edef\csvloopTest{\csvloop*[\textbf]{hello ,my ,friends}}
}\hrulefill\par
\test{csvloop}
%
\subsection{Index in lists and items by index}
%
\subsubsection{\cmd{listloop}: getting specific item}
\csvtolist*[\mylist]{one,two,three,four,five,alpha,beta,gamma}
\def\testgetlistitem{%
   \edef\getlistitemTest{\listloop[4]\mylist}
}\hrulefill\par
\noindent\hskip6pt|\csvtolist*[\mylist]{one,two,three,four,five,alpha,beta,gamma}|\par\vs
\test{getlistitem}
%
\ClearPage*
\subsubsection{\cmd{getlistindex}}
\ifpdfTeX
\leavevmode\vadjust{\textsl{Require the }\string\pdfstrcmp\ \textsl{primitive (pdf\TeX{})
\def\testgetlistindex{%
   \edef\getlistindexTest{\getlistindex{alpha}\mylist}
}\hrulefill\par
\noindent\hskip6pt|\csvtolist*[\mylist]{one,two,three,four,five,alpha,beta,gamma}|\par\vs
\test*{getlistindex}
%
\leavevmode\vadjust{\textsl{Require the }\string\pdfstrcmp\ \textsl{primitive (pdf\TeX{})
\def\testgetlistindex{%
   \edef\getlistindexTest{\getcsvlistindex*{alpha}{one,two,three,four,five,alpha,beta}}
}\hrulefill\par
\test*{getlistindex}
```

```
\fi
\hrulefill\par
%
\getlistindex[\myindex]{alpha}\mylist
{\color{blue}\noindent\hskip6pt\llap\smex|\getlistindex[\myindex]{alpha}\mylist|\par\vski
{\tt\string\myindex=\quad\textbf{\meaning\myindex}}
\par\hrulefill\par\vskip.5ex
%
\hrulefill\par
\newcount\myindex
\getcsvlistindex*[\myindex]{alpha}{one,two,three,four,five,alpha,beta}
\noindent\hskip6pt|\newcount\myindex|\par
{\color{blue}\noindent\hskip6pt\llap\smex|\getcsvlistindex*[\myindex]{alpha}{one,two,thre
{\tt\string\the\string\myindex=\qquad\textbf{\the\myindex}}
\par\hrulefill\par\vskip.5ex
%
\ClearPage
\subsubsection{\cmd{gettokslistindex} with \cmd{ifcase}}
%
\hrulefill\par\vskip2ex\noindent
\llap{\FE\,\smex}\par\vskip-2.5\baselineskip\strut
\leavevmode\vadjust{Always purely expandable (no need of |\pdfstrcmp|, comparison done by
\begin{Verbatim}
      \ifcase  \gettokslistindex{D}{LRDF\relax O}
            What do to if L
      \or   What do to if R
      \or   What do to if D
      \or   What do to if F
      \or   What do to if \relax
      \or   What do to if O
      \else Problem
      \fi
\end{Verbatim}
%
{\tt Result=\qquad\bfseries
\ifcase\gettokslistindex{D}{LRDF}
      What do to if L
\or   What do to if R
\or   What do to if D
\or   What do to if F
\else Problem
\fi}\par
\hrulefill\par\vskip.5ex
%
\end{document}\endinput
⟨/example⟩
```

Revision history

3.1415926 2010-12-07

Addition of `\@char@testopt`

Correction of a but in `\ifempty` and `\ifnotempty`.

Correction (again...) in `\thefontname`: now use of `\nfss@text` (better if in math mode).

Documentation revisited with **interfaces**[15].

3.14159 2010-04-20

`\ettl@thefontname` overwrote `\T1/cmr/m/n/10` on the log file when the font `ecmr1000` was used.
Fixed + modification of `\thefontname` in order to get both TEX and LATEX font names.

Correction of bugs into `\listdel`

Joining the example file `etextools-examples.tex` to this `.dtx` file.

---

[15] **interfaces**: CTAN:macros/latex/contrib/interfaces

3.1415 2009-10-14

Correction of a bug in `\locinterplin`.

3.141 2009-10-08

`\relax` added after `\listloop` **!** (`\csvloop` **!** etc.) in order to explicitly stop `\numexpr`.

3.14 2009-10-04

Stabilisation of some commands.

3.0 2009-09-09

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

2k 2009-09-04

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

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

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

Test on pdfLATEX and XeTEX.

2i 2009-08-31

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

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

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

Addition of `\ifincsvlist`, `\ifintokslist` and `\xifincsvlist`.

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

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

Optimization of `\getlistindex` and `\getcsvlistindex`

2t 2009-08-15

Addition of `\ifnotempty`, `\ifstrcmp`, `\ifstrmatch`

2h 2009-08-14

`\getlistindex` is now fully expandable

Addition of
`\toksloop`

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

2z 2009-08-12

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

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

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

Index added to this documentation paper.

2e 2009-07-14

First version (include an example file)

References

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

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

Index

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