The etextools package

An e-TeX package providing useful tools for LaTeX Users and package Writers

Florent CHERVET florent.chervet@free.fr

Version 2 ε

14 July 2009

Contents

~ Abstract ~

The etextools package is based on the etex and etoolbox packages and defines more tools 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. Both of them are available on CTAN under the /latex/contrib/ directory [1].

The main contributions of this package are :

- the \expandnext macro;
- the ability to define fully expandable macros with parameter and/or star version (with a small restriction) – see \FE@testopt, \FE@ifstar;
- a Command-List Parser constructor that fully uses this new feature : command-list parser are fully expandable – see \csvloop and \listloop;
- three macros that are lacking from the etoolbox package for removing elements from lists : \listdel and \listgdel, \listxdel.

[1] 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

1 Introduction

1.1 Identification

```
⟨*package⟩
\NeedsTeXFormat{LaTeX2e}[1996/12/01]
\ProvidesPackage{etextools}
   [2009/07/14 v2e e-TeX more useful tools for LaTeX package writers]
\csname ettl@onlyonce\endcsname\let\ettl@onlyonce\endinput
```

1.2 Requirements

This package requires both the etex package by David Carlisle and the etoolbox package by Philipp Lehman.

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

Furthermore, the space token must have its natural catcode (10) all along this package.

```
\edef\ettl@restore@space@catcode{\catcode`\ =\the\catcode`\ }
\AtEndOfPackage{\ettl@restore@space@catcode
   \let\ettl@restore@space@catcode\ettl@undefined}
\catcode`\ =10
```

1.3 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.

Thank to Heiko Oberdiek who has done so much for the TEX and LATEX community, and allows me to produce a package in one only .dtx file.

1.4 Hint for the reader

Every command provided in this etextools package are fully expandable unless explicitly specified. As it appears to be the philosophy of etoolbox to make the most of its commands fully expandable (and this is very useful for package writing and/or document-style programming) the fully expandable commands have a special sign in the margin (displayed here for information).

Remain what a control sequence, a control word and a control character are...

User commands

2 A few (7) “helper” macros

\ettl@afterelse

\ettl@afterfi

Those two commands are the copies of \bbl@afterelse and \bbl@afterfi from the babel package. They allow to get out of \if...\fi conditionals.

```
\long\def\ettl@afterelse#1\else#2\fi{\fi#1}
\long\def\ettl@afterfi#1\fi{\fi#1}
```

Those commands are used in \iffirstchar and \expandnext

\@gobblescape

This sequence of command 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{\expandafter\@gobble\string}
```

\str@gobblescape{⟨*string*⟩}

With ε-TEX \detokenize it is now possible to invoke a \string on an arbitrary sequence of tokens. Unfortunately, \detokenize adds a space tokens at the very end of the sequence. \str@gobblescape acts exactly the same way as @gobblescape i. e., it removes the first character from a string, but removes the trailing spaces as well:

```
\newcommand\str@gobblescape[1]{\expandafter\deblank\expandafter{%
        \expandafter\@gobble\detokenize{#1}}}
```

The \deblank command will be desbribed later.

3 Expansion control

NOTA BENE: Pay attention that the expansion control macros are not *primitives*! Therefore:

\expandafter\controlsequence\noexpandcs{⟨*csname*⟩}
\expandafter\controlsequence\expandonce{⟨*command*⟩}

etc. etc. may not lead to the result you expected. The \expandonce macro from etoolbox worth the same warning.

For expansion control, it's better to use primitives, unless you clearly knows what you do and what will happen... **In particular, you'd very better avoid to use expansion control macros at the very beginning of the definition of a command, for if this command is preceded (in the user's code) by \expandafter, you will be lead to undesirable result and most probably to an error !**

\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}.

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

\noexpandcs is used in the definition of \DeclareListCmdParser.

\noexpandafter

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

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

This command is used in the definition of \DeclareListCmdParser.

\expandnext{⟨*cstoken*⟩}{⟨*first parameter of cstoken*⟩}

We often want a control sequence to be expanded after its first argument. For example, if we want to test if a command \foo has a blank replacement text we will type :

```
\expandafter\ifblank\expandafter{\foo}
```

Now suppose you wish to define a macro \detokenizecs{⟨*csname*⟩} that expands to the detokenized content of \csname ⟨*csname*⟩\endcsname. Without \expandnext you will have to say:

```
\expandafter\expandafter\expandafter\detokenize
\expandafter\expandafter\expandafter{\csname ⟨csname⟩\endcsname}
```

six \expandafter(s). With \expandnext you will just have to say:

```
\expandnext\expandnext\detokenize{\csname #1\endcsname}
```

Now observe the following game :

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

And **guess how many** \expandafter **you will need in order to test "**\ifblank{foo}**" directly from** \FOOL **???**

\expandnext solves this problem : \FOOL has 5 degrees of expansion until it expands to "foo", so one require exactly 5 \expandnext : the solution is :

```
\expandnext\expandnext\expandnext\expandnext\expandnext\ifblank{\FOOL}
```

And now : if you define, say : \csdef{bloody fool}{\FOO} you just require 2 more \expandnext in order to test "foo" from "\csname bloody fool\endcsname" ! *just test it!*

```
\newcommand\expandnext[2]{%
   \ifx#1\expandnext
      \ettl@afterelse\expandafter\expandafter\expandafter
            \expandafter\@expandnext{#2}{\expandafter\expandafter\expandafter}
   \else\ettl@afterfi\expandafter#1\expandafter{#2}
   \fi}
\long\def\@expandnext#1#2#3{%
   \ifx#1\expandnext
      \expandafter\ettl@afterelse\expandafter\expandafter\expandafter
            \expandafter\@expandnext{#3}{\expandafter#2#2}
   \else
      \expandafter\ettl@afterfi#2#1#2{#3}
   \fi}
```

This code is not very tricky but difficult to explain. The better is to watch at the log if the reader is eager to understand the job of each \expandafter.

Note that the first argument of \expandnext must be a single ⟨*cstoken*⟩ (for \expandnext acts only on the first following token).

4 String manipulation

4.1 String trimming

\deblank{⟨*string*⟩}

The macro \deblank will remove the extra blank space inserted by \detokenize at the very end of the ⟨*string*⟩. Actually, it removes all trailing spaces (charcode 32, catcode 10) from its argument:

```
\newcommand\deblank{}
\begingroup\catcode`\&=3% a & as a math shift
\long\gdef\deblank#1{\@deblank#1 &}
\long\gdef\@deblank#1 #2&{\ifblank{#2}{#1}{#1\@deblank#2 &}}
\endgroup
```

Note, by the way, that the leading spaces are also removed by TEX's mastication mechanism.

4.2 Expanded string comparison

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

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

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

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

In the same way, \xifstrequal is similar to \ifstrequal but both ⟨*string*⟩ arguments are expanded with \protected@edef before comparison:

```
\newrobustcmd\xifstrequal[2]{%
   \begingroup\protected@edef\@tempa{#1}\protected@edef\@tempb{#2}%
      \ifx\@tempa\@tempb   \aftergroup\@firstoftwo
      \else                \aftergroup\@secondoftwo
      \fi\endgroup}
```

4.3 Testing characters

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

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

\iffirstchar will compare the character codes of the **first** characters of each ⟨*string*⟩. The comparison is *catcode agnostic* and the macro is fully expandable. Neither ⟨*string1*⟩ not ⟨*string2*⟩ is expanded before comparison.

There is a "special" use of this command when the user want to test if a ⟨*string*⟩ begins with an *escape* character (\) one may say:

\iffirstchar \@backslashchar{⟨*string*⟩} or even easier:
\iffirstchar {}{⟨*string*⟩}

Please note that the tests:

\iffirstchar {⟨*whatever string1 is*⟩}{}
and: \iffirstchar {}{}

are **always expanded into** ⟨*false*⟩ (for consistency with the shortcut-test for \@backslashchar):

```
\newcommand\iffirstchar[2]{%
   \if \expandafter\@car\string#2\relax\@nil\expandafter\@car#1\string\\\@nil
         \ettl@afterelse\ifblank{#2}\@secondoftwo\@firstoftwo
   \else \expandafter\@secondoftwo
   \fi}
```

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

\ifsinglechar will perform the test \iffirstchar but will also check that ⟨*string2*⟩ has only one character.

```
\newcommand\ifsinglechar[2]{%
```

Remember that neither ⟨*string1*⟩ nor ⟨*string2*⟩ is expanded.
First: test if #2 is a single character:

```
   \expandnext\expandnext\ifblank{\expandafter\@cdr\string#2 \@nil}
```

```
52 %  \expandafter\expandafter\expandafter\ifblank
53 %     \expandafter\expandafter\expandafter{%
54 %        \expandafter\@cdr\string#2 \@nil}
```

The test returned ⟨*true*⟩: therefore test further the character codes of `#2` and `#1`, and switch to ⟨*true*⟩ only in case of equality:

```
55   {\iffirstchar{#1}{#2}}
```

Otherwise, switch to ⟨*false*⟩

```
56    \@secondoftwo}
```

Now with the macro `\ifsinglechar` 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 assignation. The "trick" is the following:

```
\def\MacroWithOption#1{\ifsinglechar[{#1}
                                {\MacroHasOption[}
                                {\MacroNoOption{#1}}}}
\def\MacroHasOption[#1]#2...    definition
\def\MacroNoOption#1...         definition
```

Moreover (in the style of `\@testopt`):

```
\def\MacroWithOption#1{\ifsinglechar[{#1}
                                {\MacroHasOption#1}
                                {\MacroHasOption[default]{#1}}}
```

Therefore, the following macro is defined:

4.4 Fully expandable macros with options

`\FE@testopt`{⟨*argument to be tested*⟩}{⟨*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}}
```

Remember that `\MacroWithOption` **must have at least one mandatory argument.**

```
57 \newcommand\FE@testopt[3]{\ifsinglechar [{#1}{#2#1}{#2[{#3}]{#1}}}
```

Limitation : if you call such an option command without option and with `{[}` as mandatory argument, it will be considered that the command has an option, whose end will be at the next '`]`' found in the input string.

`\FE@testopt` is used in the definition of `\DeclareListCmdParser`.

`\FE@ifstar`{⟨*argument to be tested*⟩}{⟨*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}}
```

Remember that `\StarredCommand` and `\NotStarredCommand` **must have at least one mandatory argument.**

```
58 \newcommand\FE@ifstar[3]{\ifsinglechar *{#1}{#2}{#3{#1}}}
```

Limitation : if you call such a command with `{*}` as mandatory argument, it will be considered that the command is starred, and the mandatory argument will be found next inside the input string.

This "limitation" is in fact a benediction: thank to it we can have *Fully-Expandable-starred-macros-with-option*!

\FE@ifstar is used in the definitions of \DeclareListCmdParser, \csvloop, \listloop and \csvtolist.

5 List management

5.1 The Command-List Parser

The etoolbox package provides a way to define list parsers a 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-parser then, thank to the fully expandable macro \FE@testopt we can call for expansion:

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

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

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

\DeclareCmdListParser{⟨*command*⟩}{⟨*separator*⟩}

\DeclareCmdListParser acts in the same way as etoolbox-\DeclareListParser and the command-list-parser are sensitive to the category code of the ⟨*separator*⟩ *(which-is-not-necessarily-a-single-character-and-shall-not-be-a-&-with-a-catcode-of-3)*.

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

```
\newrobustcmd*\DeclareCmdListParser[2]{%
  \@ifdefinable#1
    {\expandafter\etextools@defcmdparser\expandafter{#1}{#2}}}
```

Then the job is done by \etextools@defcmdparser: we need the ' & ' to have a catcode of 3 and globally define the macro inside a purposeful group:

```
\begingroup\catcode`\&=3
\gdef\etextools@defcmdparser#1#2{%
   \begingroup
```

Now the parser definition is made inside an protected-\edef in order to expand control sequences names:

```
   \protected@edef\defineparser{\endgroup
```

Here we define the entry-point: we first test if the command was starred. This is possible because the list-parser has always a mandatory argument (the ⟨*list*⟩ itself or the ⟨*listmacro*⟩) :

```
   \long\def#1####1{\noexpand\FE@ifstar{####1}
      {\noexpandcs{ettl@lst@star\string#1}}
      {\noexpandcs{ettl@lst@nost\string#1}}}%
```

Both starred and not-starred versions have an optional argument which is the auxiliary command, whose name is \do if not specified. It is possible to test the option for the same reason:

```
   \long\csdef{ettl@lst@star\string#1}####1{\noexpand\FE@testopt{####1}
      {\noexpandcs{ettl@lst@star@pt\string#1}}{\noexpand\do}}%
   \long\csdef{ettl@lst@nost\string#1}####1{\noexpand\FE@testopt{####1}
      {\noexpandcs{ettl@lst@nost@pt\string#1}}{\noexpand\do}}%
```

Definition of the parser with its arguments : `[optional command]{list or listmacro}`. If the starred version was used, then we do not have to expand the (mandatory) ⟨*list*⟩:

```
\long\csdef{ettl@lst@star@pt\string#1}[####1]####2{%
   \noexpandcs{ettl@lst\string#1}{####2}{####1}}%
```

On the other hand, if the parser was not starred, the (mandatory) ⟨*listmacro*⟩ is expanded once:

```
\long\csdef{ettl@lst@nost@pt\string#1}[####1]####2{%
   \noexpandafter\noexpandcs{ettl@lst\string#1}\noexpandafter{%
                                        ####2}{####1}}%
```

ListParser is defined and leads to `\"ettl@lst\string\ListParser"` in all cases; here the ⟨*list*⟩ is in first position, the auxiliary commands come after, so we reverse the order and add a ⟨*separator*⟩ in case the ⟨*list*⟩ is empty:

```
\long\csdef{ettl@lst\string#1}####1####2{%
      \noexpandcs{ettl@lst@\string#1}{####2}####1\noexpand#2&}%
```

In the following macro, we extract the first item from the list (before the ⟨*separator*⟩ #2), for treatment:

```
\long\csdef{ettl@lst@\string#1}####1####2\noexpand#2####3&{%
```

Proceed with the first item, if not empty:

```
   \noexpand\ifblank{####2}
         {}
         {\noexpand\ettl@lst@doitem{####1}{####2}}%
```

If the remainder of the list is empty then break loop, otherwise restart extracting the next, first item for treatment:

```
   \noexpand\ifblank{####3}
         {\noexpand\ettl@listbreak}
         {\noexpandcs{ettl@lst@\string#1}{####1}####3}&}%
```

Now the definitions are ready, execute them:

```
}\defineparser}%
```

`\ettl@lst@doitem` apply the auxiliary command(s) (in `#1`) to the item (`#2`): `\ettl@listbreak` breaks the loop, removing the extra `&`:

```
\long\gdef\ettl@lst@doitem#1#2{#1{#2}}%
\long\gdef\ettl@listbreak#1&{}%
```

Leave “catcode-group”:

```
\endgroup
```

5.2 Loops into lists

Now we have to declare two command-list-parsers : `\listloop` for `etoolbox` lists and `\csvloop` for comma-separated lists.

`\csvloop`[⟨*auxiliary commands*⟩]{⟨*csvlistmacro*⟩}

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

`\listloop`[⟨*auxiliary commands*⟩]{⟨*listmacro*⟩}

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

`\listloop` acts exactly as `etoolbox`-`\dolistloop` with an optional argument to change the default auxiliary command `\do` to apply to each item of the list :

```
\listloop\mylist is \listloop[\do]\mylist and is also \dolistloop\mylist
\csvloop\csvlist is \csvloop[\do]\csvlist and is also: ↩
                    \expandafter\docsvlist\expandafter{\csvlist}
```

Definition of \csvloop :

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

Definition of \listloop (with a '|' of catcode 3 – cf.etoolbox):

```
\begingroup\catcode`\|=3
   \gdef\do{\DeclareCmdListParser\listloop{|}\undef\do}\aftergroup\do
\endgroup
```

5.3 Converting csv lists to etoolbox-lists

\csvtolist{⟨*listmacro*⟩}{⟨*csvlistmacro*⟩}

\csvtolist*{⟨*listmacro*⟩}{⟨*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. It's also the first application of the \csvloop macro just defined :

```
\newcommand\csvtolist[1]{\FE@ifstar{#1}\star@csvtolist\nost@csvtolist}
\def\star@csvtolist#1{\csvloop*[{\unexpanded{\listadd#1}}]}
\def\nost@csvtolist#1{\csvloop[{\unexpanded{\listadd#1}}]}
```

5.4 Removing elements from etoolbox-lists

The etoolbox package provides \listadd, \listgadd and \listxadd commands to add items to a list. However, no command is designed to remove an element from a list.

\listdel{⟨*listmacro*⟩}{⟨*item*⟩}

\listgdel{⟨*listmacro*⟩}{⟨*item*⟩}

\listxdel{⟨*listmacro*⟩}{⟨*item*⟩}

\listxdelThe \listdel command removes the element ⟨*item*⟩ from the list ⟨*listmacro*⟩. Note that the ⟨*listmacro*⟩ is redefined after deletion:

```
\newrobustcmd\listdel[2]{\@listdel\def{#1}{#2}}
```

Commands \listgdel and \listxdel are similar, except that the assignment (i. e., the redefinition of the list) is global; for the latter, the ⟨*item*⟩ is first expanded using \protected@edef:

```
\newrobustcmd\listgdel[2]{\@listdel\gdef{#2}{#2}}
\newrobustcmd\listxdel[2]{\begingroup
   \protected@edef\@listxdel{\endgroup
      \unexpanded{\@listdel\def#1}{#2}%
   }\@listxdel}
```

Now you noticed the job is delayed to a general macro \@listdel which is relatively tricky ! We need first to be placed in an environment where the '|' delimiter has a category code of 3 (cf etoolbox-lists definitions):

```
\begingroup\catcode`\|=3\catcode`\&=3
```

Inside this "catcode-group" the definition of \@listdel ought to be global:

```
\long\gdef\@listdel#1#2#3{%
```

#1=\def or \gdef, #2=⟨*listmacro*⟩, #3=⟨*item*⟩ to remove.

In order to preserve the hash-table from temporary definitions, a group is opened:

```
\begingroup
   \def\@tempa##1|#3|##2&{##1|##2\@tempb}%
```

`\@tempa` is a delimited macro whose aim is to remove the first ⟨*item*⟩ found in the list and it adds `\@tempb` after the result. If the ⟨*item*⟩ was not in the list, then `##2` will be empty. Note that `\@tempa` is the only macro whose definition depends on the ⟨*item*⟩ (and then leads `\@listdel` not to be fully-expandable).

The result of `\@tempa` expansion is then given to `\@tempb` whose purpose is to cancel out whatever is found between the two delimiters: |`\@tempb`...|`\@tempb`:

```
\def\@tempb|##1|\@tempb##2|\@tempb{%
   \ifblank{##2}{\unexpanded{##1}}
```

If the ⟨*item*⟩ was not is the list, then `##2` will be empty, and
`\@tempb`|...|tempb...|`\@tempb`
is replaced by the *original*-list (i. e., `##1` – that we shall not expand), then the loop is broken; otherwise the ⟨*item*⟩ was in the list and `##1` is the *shortened*-list without the ⟨*item*⟩. We have to loop to remove all the ⟨*item*s⟩ of the list, except in the case where the *shortened*-list is empty after having removing ⟨*item*⟩ (`##1` empty):

```
   {\ifblank{##1}{}{\@tempx##1&}}}%
```

Now we just have to (define and) expand `\@tempx` in an `\edef` which is going to redefine the ⟨*listmacro*⟩. `\@tempx` expands first `\@tempa` and then `\@tempb` on the result of the expansion of `\@tempa`. The macro `\@tempx` itself has an argument: it is (at first stage) the replacement text of ⟨*listmacro*⟩:

```
\def\@tempx##1&{\expandafter\@tempb\@tempa|##1|\@tempb|#3|&}%
\edef\@redef{\endgroup
   \unexpanded{#1#2}{% ie: \def or \gdef   \listmacro
```

`\@redef` redefines the list using (`\def` or `\gdef`), the replacement text is the result of the expansion of `\@tempx` on the ⟨*listmacro*⟩ which is expanded once (to see its items...):

```
   \expandafter\@tempx\unexpanded\expandafter{#2}&}%
```

Then just expand `\@redef` (fully expandable):

```
}\@redef}% end of \@listdel
```

And ends the “catcode-group”:

```
\endgroup
```

⟨/package⟩