The etoolbox package

An e-TeX toolbox for class and package authors

Philipp Lehman
plehman@gmx.net

Version 1.5
April 26, 2008

Contents

1 Introduction

1.1 About

The etoolbox package is a toolbox of programming facilities geared primarily towards LaTeX class and package authors. It provides LaTeX frontends to some of the new primitives provided by e-TeX as well as some generic tools which are not related to e-TeX but match the profile of this package. The package is work in progress. Note that the initial versions of this package were released under the name elatex.

1.2 License

1.3 Feedback and contributions

I started to work on this package when I found myself implementing the same tools and shorthands I had employed in previous LaTeX packages for yet another package. For the most part, the facilities provided by etoolbox address my needs as a package author and future development is likely to be guided by these needs as well. Please note that I will not be able to address any feature requests. However, I am open to contributions by other class and package authors, provided that the contributed code is sufficiently generic, matches the profile of this package, and may be added to the package without requiring significant adaption.

1 http://www.ctan.org/tex-archive/macros/latex/base/lppl.txt

1.4 Acknowledgments

The `\ifblank` test of this package is based on code by Donald Arseneau.

2 User commands

The facilities in this section are geared towards regular users as well as class and package authors.

2.1 Definitions

`\newrobustcmd{`⟨*command*⟩`}[`⟨*arguments*⟩`][`⟨*optarg default*⟩`]{`⟨*replacement text*⟩`}`
`\newrobustcmd*{`⟨*command*⟩`}[`⟨*arguments*⟩`][`⟨*optarg default*⟩`]{`⟨*replacement text*⟩`}`

The syntax and behavior of this command is similar to `\newcommand` except that the newly defined ⟨*command*⟩ will be robust. This command differs from the `\DeclareRobustCommand` command from the LaTeX kernel in that it issues an error rather than just an informational message if the ⟨*command*⟩ is already defined. Since it uses e-TeX's low-level protection mechanism rather than the corresponding higher-level LaTeX facilities, it also does not require an additional macro to implement the 'robustness'. This command itself is also robust.

`\renewrobustcmd{`⟨*command*⟩`}[`⟨*arguments*⟩`][`⟨*optarg default*⟩`]{`⟨*replacement text*⟩`}`
`\renewrobustcmd*{`⟨*command*⟩`}[`⟨*arguments*⟩`][`⟨*optarg default*⟩`]{`⟨*replacement text*⟩`}`

The syntax and behavior of this command is similar to `\renewcommand` except that the redefined ⟨*command*⟩ will be robust. This command itself is also robust.

`\providerobustcmd{`⟨*command*⟩`}[`⟨*arguments*⟩`][`⟨*optarg default*⟩`]{`⟨*replacement text*⟩`}`
`\providerobustcmd*{`⟨*command*⟩`}[`⟨*arguments*⟩`][`⟨*optarg default*⟩`]{`⟨*replacement text*⟩`}`

The syntax and behavior of this command is similar to `\providecommand` except that a newly defined ⟨*command*⟩ will be robust. Note that this command only provides a robust definition if the ⟨*command*⟩ is undefined. It will not make an already defined ⟨*command*⟩ robust. This command itself is robust.

2.2 Patching

`\robustify{`⟨*command*⟩`}`

Modifies an already defined ⟨*command*⟩ such that it is robust without altering its parameters, its prefixes, or its replacement text. If the ⟨*command*⟩ has been defined with `\DeclareRobustCommand`, this will be detected automatically. LaTeX's high-level protection mechanism will be replaced by the corresponding low-level e-TeX facility in this case. This command is robust and may only be used in the document preamble.

2.3 Length and counter assignments

The facilities in this section are replacements for `\setcounter` and `\setlength` which support arithmetic expressions.

`\defcounter{`⟨*counter*⟩`}{`⟨*integer expression*⟩`}`

Assigns a value to a LaTeX ⟨*counter*⟩ previously initialized with `\newcounter`. This command is similar in concept and syntax to `\setcounter` except for two major differences. 1) The second argument may be an ⟨*integer expression*⟩ which will be processed with `\numexpr`. The ⟨*integer expression*⟩ may be any arbitrary code which is valid in this context. The value assigned to the ⟨*counter*⟩ will be the result of that calculation. 2) In contrast to `\setcounter`, the assignment is local by default but `\defcounter` may be prefixed with `\global`. The functional equivalent of `\setcounter` would be `\global\defcounter`.

`\deflength{`⟨*length*⟩`}{`⟨*dimen expression*⟩`}`

Assigns a value to a ⟨*length*⟩ register previously initialized with `\newlength`. This command is similar in concept and syntax to `\setlength` except that the second argument may be a ⟨*dimen expression*⟩ which will be processed with `\dimexpr`. The ⟨*dimen expression*⟩ may be any arbitrary code which is valid in this context. The value assigned to the ⟨*length*⟩ register will be the result of that calculation. The assignment is local by default but `\deflength` may be prefixed with `\global`. This command may be used as a drop-in replacement for `\setlength`.

2.4 Predefined all-purpose hooks

LaTeX provides two hooks which defer the execution of code either to the beginning or the end of the document body. The `\AtBeginDocument` code is executed at the beginning of the document body, *after* the main `aux` file has been read for the first time. The `\AtEndDocument` code is executed at the end of the document body, *before* the main `aux` file is read for the second time. The hooks introduced here are similar in concept but defer the execution of their ⟨*code*⟩ argument to slightly different locations. The ⟨*code*⟩ may be arbitrary TeX code. Macro parameter characters in the ⟨*code*⟩ argument need not be doubled.

`\AtEndPreamble{`⟨*code*⟩`}`

This hook differs from `\AtBeginDocument` in that the ⟨*code*⟩ is executed right at the end of the preamble, before the main `aux` file (as written during the previous LaTeX pass) is read and prior to any `\AtBeginDocument` code. It is sometimes desirable to defer code to the end of the preamble because all requested packages have been loaded at this point. Code deferred with `\AtBeginDocument`, however, is executed too late if it is required in the `aux` file.

`\AfterEndDocument{`⟨*code*⟩`}`

This hook differs from `\AtEndDocument` in that the ⟨*code*⟩ is executed at the very end of the document, after the main `aux` file (as written during the current LaTeX pass) has been read and after any `\AtEndDocument` code. This hook is useful if a package needs to evaluate any data in the `aux` file at the end of a LaTeX run.

`\AfterPreamble{`⟨*code*⟩`}`

This hook is a variant of `\AtBeginDocument` which may be used in both the pream-

ble and the document body. When used in the preamble, it behaves exactely like `\AtBeginDocument`. When used in the document body, it immediately executes its ⟨*code*⟩ argument (`\AtBeginDocument` issues an error in this case).

3 Author commands

The facilities in this section are geared towards class and package authors.

3.1 Definitions

3.1.1 Macro definitions

The facilities in this section are simple but frequently required shorthands which extend the scope of the `\@namedef` and `\@nameuse` macros from the LaTeX kernel.

`\csdef`{⟨*csname*⟩}⟨*arguments*⟩{⟨*replacement text*⟩}

Similar to the TeX primitive `\def` except that it takes a control sequence name as its first argument. This command is robust and corresponds to `\@namedef`.

`\csgdef`{⟨*csname*⟩}⟨*arguments*⟩{⟨*replacement text*⟩}

Similar to the TeX primitive `\gdef` except that it takes a control sequence name as its first argument. This command is robust.

`\csedef`{⟨*csname*⟩}⟨*arguments*⟩{⟨*replacement text*⟩}

Similar to the TeX primitive `\edef` except that it takes a control sequence name as its first argument. This command is robust.

`\csxdef`{⟨*csname*⟩}⟨*arguments*⟩{⟨*replacement text*⟩}

Similar to the TeX primitive `\xdef` except that it takes a control sequence name as its first argument. This command is robust.

`\protected@csedef`{⟨*csname*⟩}⟨*arguments*⟩{⟨*replacement text*⟩}

Similar to `\csedef` except that LaTeX's protection mechanism is temporarily enabled. To put it in other words: this command is similar to the LaTeX kernel command `\protected@edef` except that it takes a control sequence name as its first argument. This command is robust.

`\protected@csxdef`{⟨*csname*⟩}⟨*arguments*⟩{⟨*replacement text*⟩}

Similar to `\csxdef` except that LaTeX's protection mechanism is temporarily enabled. To put it in other words: this command is similar to the LaTeX kernel command `\protected@xdef` except that it takes a control sequence name as its first argument. This command is robust.

`\cslet`{⟨*csname*⟩}{⟨*command*⟩}

Similar to the TeX primitive `\let` except that the first argument is a control sequence name. This command is robust and may be prefixed with `\global`.

`\letcs{`⟨*command*⟩`}{`⟨*csname*⟩`}`

Similar to the TeX primitive `\let` except that the second argument is a control sequence name. This command is robust and may be prefixed with `\global`.

`\csletcs{`⟨*csname*⟩`}{`⟨*csname*⟩`}`

Similar to the TeX primitive `\let` except that both arguments are control sequence names. This command is robust and may be prefixed with `\global`.

`\csuse{`⟨*csname*⟩`}`

Takes a control sequence name as its argument and forms a control sequence token. This command differs from the `\@nameuse` macro from the LaTeX kernel in that it expands to an empty string if the control sequence is undefined.

`\undef`⟨*command*⟩

Clears a ⟨*command*⟩ such that e-TeX's `\ifdefined` and `\ifcsname` tests will consider it as undefined. This command is robust and may be prefixed with `\global`.

`\csundef{`⟨*csname*⟩`}`

Similar to `\undef` except that it takes a control sequence name as its argument. This command is robust and may be prefixed with `\global`.

3.1.2 Arithmetic definitions

The facilities in this section permit calculations using macros rather than length registers and counters.

`\numdef`⟨*command*⟩`{`⟨*integer expression*⟩`}`

Similar to `\edef` except that the ⟨*integer expression*⟩ is processed with `\numexpr`. The ⟨*integer expression*⟩ may be any arbitrary code which is valid in this context. The replacement text assigned to the ⟨*command*⟩ will be the result of that calculation. If the ⟨*command*⟩ is undefined, it will be initialized to '0' before the ⟨*integer expression*⟩ is processed.

`\numgdef`⟨*command*⟩`{`⟨*integer expression*⟩`}`

Similar to `\numdef` except that the assignment is global.

`\csnumdef{`⟨*csname*⟩`}{`⟨*integer expression*⟩`}`

Similar to `\numdef` except that it takes a control sequence name as its first argument.

`\csnumgdef{`⟨*csname*⟩`}{`⟨*integer expression*⟩`}`

Similar to `\numgdef` except that it takes a control sequence name as its first argument.

`\dimdef`⟨*command*⟩`{`⟨*dimen expression*⟩`}`

Similar to `\edef` except that the ⟨*dimen expression*⟩ is processed with `\dimexpr`.

The ⟨*dimen expression*⟩ may be any arbitrary code which is valid in this context. The replacement text assigned to the ⟨*command*⟩ will be the result of that calculation. If the ⟨*command*⟩ is undefined, it will be initialized to '`0pt`' before the ⟨*dimen expression*⟩ is processed.

`\dimgdef`⟨*command*⟩`{`⟨*dimen expression*⟩`}`

Similar to `\dimdef` except that the assignment is global.

`\csdimdef{`⟨*csname*⟩`}{`⟨*dimen expression*⟩`}`

Similar to `\dimdef` except that it takes a control sequence name as its first argument.

`\csdimgdef{`⟨*csname*⟩`}{`⟨*dimen expression*⟩`}`

Similar to `\dimgdef` except that it takes a control sequence name as its first argument.

`\gluedef`⟨*command*⟩`{`⟨*glue expression*⟩`}`

Similar to `\edef` except that the ⟨*glue expression*⟩ is processed with `\glueexpr`. The ⟨*glue expression*⟩ may be any arbitrary code which is valid in this context. The replacement text assigned to the ⟨*command*⟩ will be the result of that calculation. If the ⟨*command*⟩ is undefined, it will be initialized to '`0pt plus 0pt minus 0pt`' before the ⟨*glue expression*⟩ is processed.

`\gluegdef`⟨*command*⟩`{`⟨*glue expression*⟩`}`

Similar to `\gluedef` except that the assignment is global.

`\csgluedef{`⟨*csname*⟩`}{`⟨*glue expression*⟩`}`

Similar to `\gluedef` except that it takes a control sequence name as its first argument.

`\csgluegdef{`⟨*csname*⟩`}{`⟨*glue expression*⟩`}`

Similar to `\gluegdef` except that it takes a control sequence name as its first argument.

`\mudef`⟨*command*⟩`{`⟨*muglue expression*⟩`}`

Similar to `\edef` except that the ⟨*muglue expression*⟩ is processed with `\muexpr`. The ⟨*muglue expression*⟩ may be any arbitrary code which is valid in this context. The replacement text assigned to the ⟨*command*⟩ will be the result of that calculation. If the ⟨*command*⟩ is undefined, it will be initialized to '`0mu`' before the ⟨*muglue expression*⟩ is processed.

`\mugdef`⟨*command*⟩`{`⟨*muglue expression*⟩`}`

Similar to `\mudef` except that the assignment is global.

`\csmudef{`⟨*csname*⟩`}{`⟨*muglue expression*⟩`}`

Similar to `\mudef` except that it takes a control sequence name as its first argument.

`\csmugdef{`⟨*csname*⟩`}{`⟨*muglue expression*⟩`}`

Similar to `\mugdef` except that it takes a control sequence name as its first argument.

3.2 Expansion control

The facilities in this section are useful to control expansion in an `\edef` or a similar context.

`\expandonce`⟨*command*⟩

This command expands ⟨*command*⟩ once and prevents further expansion of the replacement text.

`\csexpandonce{`⟨*csname*⟩`}`

Similar to `\expandonce` except that it takes a control sequence name as its argument.

`\protecting{`⟨*code*⟩`}`

This command applies LaTeX's protection mechanism, which normally requires prefixing each fragile command with `\protect`, to an entire chunk of arbitrary ⟨*code*⟩. Its behavior depends on the current state of `\protect`. Note that the braces around the ⟨*code*⟩ are mandatory even if it is a single token.

3.3 Hook management

The facilities in this section are intended for hook management. A ⟨*hook*⟩ in this context is a plain macro without any parameters and prefixes which is used to collect code to be executed later. These facilities may also be useful to patch simple macros by appending code to their replacement text. For more complex patching operations, see section 3.4. All commands in this section will initialize the ⟨*hook*⟩ if it is undefined.

3.3.1 Appending to a hook

The facilities in this section append arbitrary code to a hook.

`\appto`⟨*hook*⟩`{`⟨*code*⟩`}`

This command appends arbitrary ⟨*code*⟩ to a ⟨*hook*⟩. If the ⟨*code*⟩ contains any parameter characters, they need not be doubled. This command is robust.

`\gappto`⟨*hook*⟩`{`⟨*code*⟩`}`

Similar to `\appto` except that the assignment is global. This command may be used as a drop-in replacement for the `\g@addto@macro` command in the LaTeX kernel.

`\eappto`⟨*hook*⟩`{`⟨*code*⟩`}`

This command appends arbitrary ⟨*code*⟩ to a ⟨*hook*⟩. The ⟨*code*⟩ is expanded at definition-time. Only the new ⟨*code*⟩ is expanded, the current replacement text of the ⟨*hook*⟩ is not. This command is robust.

`\xappto`⟨*hook*⟩`{`⟨*code*⟩`}`

Similar to `\eappto` except that the assignment is global.

`\protected@eappto`⟨*hook*⟩`{`⟨*code*⟩`}`

Similar to `\eappto` except that LaTeX's protection mechanism is temporarily enabled.

`\protected@xappto`⟨*hook*⟩`{`⟨*code*⟩`}`

Similar to `\xappto` except that LaTeX's protection mechanism is temporarily enabled.

`\csappto{`⟨*csname*⟩`}{`⟨*code*⟩`}`

Similar to `\appto` except that it takes a control sequence name as its first argument.

`\csgappto{`⟨*csname*⟩`}{`⟨*code*⟩`}`

Similar to `\gappto` except that it takes a control sequence name as its first argument.

`\cseappto{`⟨*csname*⟩`}{`⟨*code*⟩`}`

Similar to `\eappto` except that it takes a control sequence name as its first argument.

`\csxappto{`⟨*csname*⟩`}{`⟨*code*⟩`}`

Similar to `\xappto` except that it takes a control sequence name as its first argument.

`\protected@cseappto`⟨*hook*⟩`{`⟨*code*⟩`}`

Similar to `\protected@eappto` except that it takes a control sequence name as its first argument.

`\protected@csxappto`⟨*hook*⟩`{`⟨*code*⟩`}`

Similar to `\protected@xappto` except that it takes a control sequence name as its first argument.

3.3.2 Prepending to a hook

The facilities in this section 'prepend' arbitrary code to a hook, i. e., the code is inserted at the beginning of the hook rather than being added at the end.

`\preto`⟨*hook*⟩`{`⟨*code*⟩`}`

Similar to `\appto` except that the ⟨*code*⟩ is prepended.

`\gpreto`⟨*hook*⟩`{`⟨*code*⟩`}`

Similar to `\preto` except that the assignment is global.

\epreto⟨*hook*⟩{⟨*code*⟩}

Similar to \eappto except that the ⟨*code*⟩ is prepended.

\xpreto⟨*hook*⟩{⟨*code*⟩}

Similar to \epreto except that the assignment is global.

\protected@epreto⟨*hook*⟩{⟨*code*⟩}

Similar to \epreto except that LaTeX's protection mechanism is temporarily enabled.

\protected@xpreto⟨*hook*⟩{⟨*code*⟩}

Similar to \xpreto except that LaTeX's protection mechanism is temporarily enabled.

\cspreto{⟨*csname*⟩}{⟨*code*⟩}

Similar to \preto except that it takes a control sequence name as its first argument.

\csgpreto{⟨*csname*⟩}{⟨*code*⟩}

Similar to \gpreto except that it takes a control sequence name as its first argument.

\csepreto{⟨*csname*⟩}{⟨*code*⟩}

Similar to \epreto except that it takes a control sequence name as its first argument.

\csxpreto{⟨*csname*⟩}{⟨*code*⟩}

Similar to \xpreto except that it takes a control sequence name as its first argument.

\protected@csepreto⟨*hook*⟩{⟨*code*⟩}

Similar to \protected@epreto except that it takes a control sequence name as its first argument.

\protected@csxpreto⟨*hook*⟩{⟨*code*⟩}

Similar to \protected@xpreto except that it takes a control sequence name as its first argument.

3.4 Patching

The facilities in this section are useful to hook into or modify existing code. All commands presented here preserve the parameters and the prefixes of the patched ⟨*command*⟩. Note that the patching process involves detokenizing the replacement text of the ⟨*command*⟩ and retokenizing it under the current category code regime after patching. The category code of '@' is temporarily set to 11. If the replacement text of the ⟨*command*⟩ includes any tokens with non-standard category codes, the respective category codes must be adjusted prior to patching. If the code to be

replaced or inserted refers to the parameters of the ⟨*command*⟩ to be patched, the parameter characters need not be doubled when invoking one of the commands below. Note that `\outer` commands may not be patched.

`\patchcmd[`⟨*prefix*⟩`]{`⟨*command*⟩`}{`⟨*search*⟩`}{`⟨*replace*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`

This command extracts the replacement text of a ⟨*command*⟩, replaces ⟨*search*⟩ with ⟨*replace*⟩, and reassembles the ⟨*command*⟩. The pattern match is category code agnostic and matches the first occurence of the ⟨*search*⟩ pattern in the replacement text of the ⟨*command*⟩ to be patched. If an optional ⟨*prefix*⟩ is specified, it replaces the prefixes of the ⟨*command*⟩. An empty ⟨*prefix*⟩ strips all prefixes from the ⟨*command*⟩. This command executes ⟨*success*⟩ if patching succeeds, and ⟨*failure*⟩ otherwise. It is robust and may only be used in the document preamble. The assignment is local.

`\ifpatchable{`⟨*command*⟩`}{`⟨*search*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

This command executes ⟨*true*⟩ if the ⟨*command*⟩ is defined and the ⟨*search*⟩ pattern is found in its replacement text, and ⟨*false*⟩ otherwise. This command is robust and may only be used in the document preamble.

`\apptocmd{`⟨*command*⟩`}{`⟨*code*⟩`}`

This command appends ⟨*code*⟩ to the replacement text of a ⟨*command*⟩. In contrast to the `\appto` command from section 3.3.1, this one may be used to patch commands with an arbitrary number of parameters. The ⟨*code*⟩ may refer to the parameters of the ⟨*command*⟩ in this case. This command is robust and may only be used in the document preamble. The assignment is local.

`\pretocmd{`⟨*command*⟩`}{`⟨*code*⟩`}`

This command is similar to `\apptocmd` except that the ⟨*code*⟩ is 'prepended', i. e., inserted at the beginning of the replacement text of the ⟨*command*⟩. In contrast to the `\preto` command from section 3.3.1, this one may be used to patch commands with an arbitrary number of parameters. The ⟨*code*⟩ may refer to the parameters of the ⟨*command*⟩ in this case. This command is robust and may only be used in the document preamble. The assignment is local.

3.5 Generic tests

`\ifdef{`⟨*command*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

A LaTeX wrapper for the e-TeX primitive `\ifdefined`. This command expands to ⟨*true*⟩ if the ⟨*command*⟩ is defined, and to ⟨*false*⟩ otherwise. Note that the ⟨*command*⟩ is considered as defined even if its meaning is `\relax`.

`\ifundef{`⟨*command*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Expands to ⟨*true*⟩ if the ⟨*command*⟩ is undefined, and to ⟨*false*⟩ otherwise. Apart from reversing the logic of the test, this command also differs from `\ifdef` in that the ⟨*command*⟩ is considered as undefined if its meaning is `\relax`.

`\ifcsdef{`⟨*csname*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

A LaTeX wrapper for the e-TeX primitive `\ifcsname`. This command expands to ⟨*true*⟩ if ⟨*csname*⟩ is defined, and to ⟨*false*⟩ otherwise. Note that ⟨*csname*⟩ is considered as defined even if its meaning is `\relax`.

`\ifcsundef{`⟨*csname*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Expands to ⟨*true*⟩ if ⟨*csname*⟩ is undefined, and to ⟨*false*⟩ otherwise. Apart from reversing the logic of the test, this command also differs from `\ifcsdef` in that the ⟨*csname*⟩ is considered as undefined if its meaning is `\relax`. This command may be used as a drop-in replacement for the `\@ifundefined` test in the LaTeX kernel.

`\ifdefvoid{`⟨*command*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Expands to ⟨*true*⟩ if the ⟨*command*⟩ is undefined, its meaning is `\relax`, or its replacement text is empty, and to ⟨*false*⟩ otherwise.

`\ifcsvoid{`⟨*csname*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Similar to `\ifdefvoid` except that it takes a control sequence name as its first argument.

`\ifdefequal{`⟨*command*⟩`}{`⟨*command*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Compares two commands and expands to ⟨*true*⟩ if they are equal in the sense of the TeX primitive `\ifx`, and to ⟨*false*⟩ otherwise. In contrast to `\ifx`, this test will also yield ⟨*false*⟩ if both commands are undefined or have a meaning of `\relax`.

`\ifcsequal{`⟨*csname*⟩`}{`⟨*csname*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Similar to `\ifdefequal` except that it takes control sequence names as arguments.

`\ifdefstring{`⟨*command*⟩`}{`⟨*string*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Compares the replacement text of a ⟨*command*⟩ to a ⟨*string*⟩ and executes ⟨*true*⟩ if they are equal, and ⟨*false*⟩ otherwise. Neither the ⟨*command*⟩ nor the ⟨*string*⟩ is expanded in the test and the comparison is category code agnostic. Control sequence tokens in the ⟨*string*⟩ argument will be detokenized and treated as strings. This command is robust. Note that it will only consider the replacement text of the ⟨*command*⟩. For example, this code

```
\long\def\@gobbletwo#1#2{}
\ifdefstring{\@gobbletwo}{}{true}{false}
```

would yield ⟨*true*⟩. The prefix and the parameters of `\@gobbletwo` are ignored.

`\ifcsstring{`⟨*csname*⟩`}{`⟨*string*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Similar to `\ifdefstring` except that it takes a control sequence name as its first argument.

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

Compares two strings and executes ⟨*true*⟩ if they are equal, and ⟨*false*⟩ otherwise. The strings are not expanded in the test and the comparison is category code agnostic. Control sequence tokens in any of the ⟨*string*⟩ arguments will be detokenized and treated as strings. This command is robust.

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

Expands to ⟨*true*⟩ if the ⟨*string*⟩ is blank (empty or spaces), and to ⟨*false*⟩ otherwise. The ⟨*string*⟩ is not expanded in the test.

3.6 Boolean switches

This package provides two interfaces to boolean switches which are completely independent of each other. The facilities in section 3.6.1 are a LaTeX frontend to `\newif`. Those in section 3.6.2 use a different mechanism.

3.6.1 TeX switches

Since the facilities in this section are based on `\newif` internally, they may be used to test and alter the state of switches previously defined with `\newif`. They are also compatible with the boolean tests of the `ifthen` package. The `\newif` approach requires a total of three macros per switch.

`\newbool{`⟨*name*⟩`}`

Defines a new boolean switch called ⟨*name*⟩. If the switch has already been defined, this command issues an error. The initial state of newly defined switches is `false`. This command is robust.

`\providebool{`⟨*name*⟩`}`

Defines a new boolean switch called ⟨*name*⟩ unless it has already been defined. This command is robust.

`\booltrue{`⟨*name*⟩`}`

Sets the boolean switch ⟨*name*⟩ to `true`. This command is robust and may be prefixed with `\global`. It will issue an error if the switch is undefined.

`\boolfalse{`⟨*name*⟩`}`

Sets the boolean switch ⟨*name*⟩ to `false`. This command is robust and may be prefixed with `\global`. It will issue an error if the switch is undefined.

`\ifbool{`⟨*name*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Expands to ⟨*true*⟩ if the state of the boolean switch ⟨*name*⟩ is `true`, and to ⟨*false*⟩ otherwise. If the switch is undefined, this command issues an error.

`\notbool{`⟨*name*⟩`}{`⟨*not true*⟩`}{`⟨*not false*⟩`}`

Similar to `\ifbool` but reverses the logic of the test.

3.6.2 LaTeX switches

In contrast to the switches from section 3.6.1, the facilities in this section require only one macro per switch. They also use a separate namespace to avoid name clashes with regular macros.

`\newtoggle{`⟨*name*⟩`}`

Defines a new boolean switch called ⟨*name*⟩. If the switch has already been defined, this command issues an error. The initial state of newly defined switches is `false`. This command is robust.

`\providetoggle{`⟨*name*⟩`}`

Defines a new boolean switch called ⟨*name*⟩ unless it has already been defined. This command is robust.

`\toggletrue{`⟨*name*⟩`}`

Sets the boolean switch ⟨*name*⟩ to `true`. This command is robust and may be prefixed with `\global`. It will issue an error if the switch is undefined.

`\togglefalse{`⟨*name*⟩`}`

Sets the boolean switch ⟨*name*⟩ to `false`. This command is robust and may be prefixed with `\global`. It will issue an error if the switch is undefined.

`\iftoggle{`⟨*name*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Expands to ⟨*true*⟩ if the state of the boolean switch ⟨*name*⟩ is `true`, and to ⟨*false*⟩ otherwise. If the switch is undefined, this command issues an error.

`\nottoggle{`⟨*name*⟩`}{`⟨*not true*⟩`}{`⟨*not false*⟩`}`

Similar to `\iftoggle` but reverses the logic of the test.

3.7 List processing

3.7.1 User input

The facilities in this section are primarily designed to handle user input. When building lists for internal use by a package, using the facilities in section 3.7.2 is preferable.

`\DeclareListParser{`⟨*command*⟩`}{`⟨*separator*⟩`}`

This command defines a list parser similar to the `\docsvlist` command below, which is defined with '`\DeclareListParser{\docsvlist}{,}`'. Note that the list parsers are sensitive to the category code of the ⟨*separator*⟩.

`\docsvlist{`⟨*item, item, ...*⟩`}`

This command loops over a comma-separated list and executes the auxiliary command `\do` for every item in the list, passing the item as an argument. In contrast to the `\@for` loop in the LaTeX kernel, `\docsvlist` is expandable. With a suitable definition of `\do`, lists may be processed inside an `\edef` or a comparable context.

You may use \listbreak at the end of the replacement text of \do to stop processing and discard the remaining items in the list. Whitespace after list separators is ignored. If an item contains a comma or starts with a space, it must be wrapped in curly braces. The braces will be removed as the list is processed. Here is a usage example which prints a comma-separated list as an itemize environment:

```
\begin{itemize}
  \renewcommand*{\do}[1]{\item #1}
  \docsvlist{item1, item2, {item3a, item3b}, item4}
\end{itemize}
```

Here is another example:

```
\renewcommand*{\do}[1]{* #1\MessageBreak}
\PackageInfo{mypackage}{%
  Example list:\MessageBreak
  \docsvlist{item1, item2, {item3a, item3b}, item4}}
```

In this example, the list is written to the log file as part of an informational message. The list processing takes place during the \write operation.

3.7.2 Internal lists

The facilities in this section handle internal lists of data. An 'internal list' in this context is a plain macro without any parameters and prefixes which is employed to collect data. These lists use a special character as internal list separator. When processing user input in a list format, see the facilities in section 3.7.1.

\listadd{⟨*listmacro*⟩}{⟨*item*⟩}

This command appends an ⟨*item*⟩ to a ⟨*listmacro*⟩.

\listgadd{⟨*listmacro*⟩}{⟨*item*⟩}

Similar to \listadd except that the assignment is global.

\listeadd{⟨*listmacro*⟩}{⟨*item*⟩}

Similar to \listadd except that the ⟨*item*⟩ is expanded at definition-time. Only the new ⟨*item*⟩ is expanded, the current list is not.

\listxadd{⟨*listmacro*⟩}{⟨*item*⟩}

Similar to \listeadd except that the assignment is global.

\listcsadd{⟨*listcsname*⟩}{⟨*item*⟩}

Similar to \listadd except that it takes a control sequence name as its first argument.

\listcsgadd{⟨*listcsname*⟩}{⟨*item*⟩}

Similar to \listcsadd except that the assignment is global.

`\listcseadd{`⟨*listcsname*⟩`}{`⟨*item*⟩`}`

Similar to `\listeadd` except that it takes a control sequence name as its first argument.

`\listcsxadd{`⟨*listcsname*⟩`}{`⟨*item*⟩`}`

Similar to `\listcseadd` except that the assignment is global.

`\dolistloop{`⟨*listmacro*⟩`}`

This command loops over all items in a ⟨*listmacro*⟩ and executes the auxiliary command `\do` for every item in the list, passing the item as an argument. The list loop itself is expandable. You may use `\listbreak` at the end of the replacement text of `\do` to stop processing and discard the remaining items in the list. Here is a usage example which prints an internal list called `\mylist` as an `itemize` environment:

```
\begin{itemize}
  \renewcommand*{\do}[1]{\item #1}
  \dolistloop{\mylist}
\end{itemize}
```

`\dolistcsloop{`⟨*listcsname*⟩`}`

Similar to `\dolistloop` except that it takes a control sequence name as its argument.

`\ifinlist{`⟨*item*⟩`}{`⟨*listmacro*⟩`}`

This command checks if an ⟨*item*⟩ is included in a ⟨*listmacro*⟩. Note that this test uses pattern matching based on TeX's argument scanner to check if the search string is included in the list. This means that it is usually faster than looping over all items in the list, but it also implies that the items must not include curly braces which would effectively hide them from the scanner. In other words, this macro is most useful when dealing with lists of plain strings rather than printable data. When dealing with printable text, it may be preferable to use `\dolistloop` to check if an item is in the list as follows:

```
\renewcommand*{\do}[1]{%
  \ifstrequal{#1}{item}
    {item found!\listbreak}
    {}}
\dolistloop{\mylist}
```

`\xifinlist{`⟨*item*⟩`}{`⟨*listmacro*⟩`}`

Similar to `\ifinlist` except that the ⟨*item*⟩ is expanded prior to the test.

`\ifinlistcs{`⟨*item*⟩`}{`⟨*listcsname*⟩`}`

Similar to `\ifinlist` except that it takes a control sequence name as its second argument.

`\xifinlistcs{`⟨*item*⟩`}{`⟨*listcsname*⟩`}`

Similar to `\xifinlist` except that it takes a control sequence name as its second argument.

3.8 Miscellaneous tools

`\rmntonum{`⟨*numeral*⟩`}`

The TeX primitive `\romannumeral` converts an integer to a Roman numeral but TeX or LaTeX provide no command which goes the opposite way. `\rmntonum` fills this gap. It takes a Roman numeral as its argument and converts it to the corresponding integer. Since it is expandable, it may also be used in counter assignments or `\ifnum` comparisons:

```
\rmntonum{mcmxcv}
\setcounter{counter}{\rmntonum{CXVI}}
\ifnum\rmntonum{mcmxcviii}<2000 ...
```

The ⟨*numeral*⟩ argument must be a literal string. It will be detokenized prior to parsing. The parsing of the numeral is case-insensitive and whitespace in the argument is ignored. If there is an invalid token in the argument, `\rmntonum` will expand to `-1`; an empty argument will yield an empty string. Note that `\rmntonum` will not check the numeral for formal validity. For example, both `V` and `VX` will yield 5.

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

Expands to ⟨*true*⟩ if ⟨*string*⟩ is a Roman numeral, and to ⟨*false*⟩ otherwise. The ⟨*string*⟩ will be detokenized prior to performing the test. The test is case-insensitive and ignores whitespace in the ⟨*string*⟩. Note that `\ifrmnum` will not check the numeral for formal validity. For example, both `V` and `VXV` will yield ⟨*true*⟩. Strictly speaking, what `\ifrmnum` does is parse the ⟨*string*⟩ in order to find out if it consists of characters which may form a valid Roman numeral.

4 Revision history

1.5 2008-04-26

Added `\defcounter` 2.3
Added `\deflength` 2.3
Added `\ifdefstring` 3.5
Added `\ifcsstring` 3.5
Improved `\rmntonum` 3.8
Added `\ifrmnum` 3.8
Added extended PDF bookmarks to this manual
Rearranged manual slightly

1.4 2008-01-24

Resolved conflict with `tex4ht`

1.3 2007-10-08

Renamed package from `elatex` to `etoolbox` 1
Renamed `\newswitch` to `\newtoggle` (name clash) 3.6.2
Renamed `\provideswitch` to `\providetoggle` (consistency) 3.6.2
Renamed `\switchtrue` to `\toggletrue` (consistency) 3.6.2
Renamed `\switchfalse` to `\togglefalse` (consistency) 3.6.2
Renamed `\ifswitch` to `\iftoggle` (consistency) 3.6.2
Renamed `\notswitch` to `\nottoggle` (consistency) 3.6.2
Added `\AtEndPreamble` 2.4
Added `\AfterEndDocument` 2.4
Added `\AfterPreamble` 2.4
Added `\undef` 3.1.1
Added `\csundef` 3.1.1
Added `\ifdefvoid` 3.5
Added `\ifcsvoid` 3.5
Added `\ifdefequal` 3.5
Added `\ifcsequal` 3.5
Added `\ifstrequal` 3.5
Added `\listadd` 3.7.2
Added `\listeadd` 3.7.2
Added `\listgadd` 3.7.2
Added `\listxadd` 3.7.2
Added `\listcsadd` 3.7.2
Added `\listcseadd` 3.7.2
Added `\listcsgadd` 3.7.2
Added `\listcsxadd` 3.7.2
Added `\ifinlist` 3.7.2
Added `\xifinlist` 3.7.2
Added `\ifinlistcs` 3.7.2
Added `\xifinlistcs` 3.7.2
Added `\dolistloop` 3.7.2
Added `\dolistcsloop` 3.7.2

1.2 2007-07-13

Renamed `\patchcommand` to `\patchcmd` (name clash) 3.4
Renamed `\apptocommand` to `\apptocmd` (consistency) 3.4
Renamed `\pretocommand` to `\pretocmd` (consistency) 3.4
Added `\newbool` 3.6.1
Added `\providebool` 3.6.1
Added `\booltrue` 3.6.1
Added `\boolfalse` 3.6.1
Added `\ifbool` 3.6.1
Added `\notbool` 3.6.1
Added `\newswitch` 3.6.2
Added `\provideswitch` 3.6.2

Added `\switchtrue` . . . 3.6.2
Added `\switchfalse` . . . 3.6.2
Added `\ifswitch` . . . 3.6.2
Added `\notswitch` . . . 3.6.2
Added `\DeclareListParser` . . . 3.7.1
Added `\docsvlist` . . . 3.7.1
Added `\rmntonum` . . . 3.8

1.1 2007-05-28

Added `\protected@csedef` . . . 3.1.1
Added `\protected@csxdef` . . . 3.1.1
Added `\gluedef` . . . 3.1.2
Added `\gluegdef` . . . 3.1.2
Added `\csgluedef` . . . 3.1.2
Added `\csgluegdef` . . . 3.1.2
Added `\mudef` . . . 3.1.2
Added `\mugdef` . . . 3.1.2
Added `\csmudef` . . . 3.1.2
Added `\csmugdef` . . . 3.1.2
Added `\protected@eappto` . . . 3.3.1
Added `\protected@xappto` . . . 3.3.1
Added `\protected@cseappto` . . . 3.3.1
Added `\protected@csxappto` . . . 3.3.1
Added `\protected@epreto` . . . 3.3.2
Added `\protected@xpreto` . . . 3.3.2
Added `\protected@csepreto` . . . 3.3.2
Added `\protected@csxpreto` . . . 3.3.2
Fixed bug in `\newrobustcmd` . . . 2.1
Fixed bug in `\renewrobustcmd` . . . 2.1
Fixed bug in `\providerobustcmd` . . . 2.1

1.0 2007-05-07

Initial public release