The etoolbox package

An e-TeX toolbox for class and package authors

Philipp Lehman
plehman@gmx.net

Version 1.9
April 10, 2010

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 Requirements

This package requires e-TeX. TeX distributions have been shipping e-TeX binaries for quite some time, most distributions even use them by default these days. This package checks if it is running under e-TeX. If you get an error message, try compiling the document with `elatex` instead of `latex` or `pdfelatex` instead of `pdflatex`, respectively.

1.3 License

1.4 Feedback

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

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

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.

1.5 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. The behavior of 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 does not require an additional macro to implement the 'robustness'.

`\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.

`\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 the newly defined ⟨*command*⟩ will be robust. Note that this command will provide a robust definition of the ⟨*command*⟩ only if it is undefined. It will not make an already defined ⟨*command*⟩ robust.

2.2 Patching

`\robustify{`⟨*command*⟩`}`

Modifies a ⟨*command*⟩ defined with `\newcommand` 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.

2.3 Protection

`\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.

2.4 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*⟩}{⟨*glue 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 ⟨*glue expression*⟩ which will be processed with `\glueexpr`. The ⟨*glue 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`.

2.5 Predefined all-purpose hooks

LaTeX provides two hooks which defer the execution of code either to the beginning or to the end of the document body. Any `\AtBeginDocument` code is executed towards the beginning of the document body, *after* the main `aux` file has been read for the first time. Any `\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. Parameter characters in the ⟨*code*⟩ argument need not be doubled.

`\AfterPreamble`{⟨*code*⟩}

This hook is a variant of `\AtBeginDocument` which may be used in both the preamble 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` would issue an error in this case. This hook is useful to defer code which needs to write to the main `aux` file.

`\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 on the previous LaTeX pass) is read and prior to any `\AtBeginDocument` code. Note that it is not possible to write to the `aux` file at this point.

`\AfterEndPreamble{`⟨*code*⟩`}`

This hook differs from `\AtBeginDocument` in that the ⟨*code*⟩ is executed at the very end of `\begin{document}`, after any `\AtBeginDocument` code. Note that commands whose scope has been restricted to the preamble with `\@onlypreamble` are no longer available when this hook is executed.

`\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 on the current LaTeX pass) has been read and after any `\AtEndDocument` code.

In a way, `\AtBeginDocument` code is part neither of the preamble nor the document body but located in-between them since it gets executed in the middle of the initialization sequence performed prior to typesetting. It is sometimes desirable to move code to the end of the preamble because all requested packages have been loaded at this point. `\AtBeginDocument` code, however, is executed too late if it is required in the `aux` file. In contrast to that, `\AtEndPreamble` code is part of the preamble; `\AfterEndPreamble` code is part of the document body and may contain printable text to be typeset at the very beginning of the document. To sum that up, LaTeX will perform the following tasks 'inside' `\begin{document}`:

- Execute any `\AtEndPreamble` code
- Start initialization for document body (page layout, default fonts, etc.)
- Load the main `aux` file written on the previous LaTeX pass
- Open the main `aux` file for writing on the current pass
- Continue initialization for document body
- Execute any `\AtBeginDocument` code
- Complete initialization for document body
- Disable all `\@onlypreamble` commands
- Execute any `\AfterEndPreamble` code

Inside `\end{document}`, LaTeX will perform the following tasks:

- Execute any `\AtEndDocument` code
- Perform a final `\clearpage` operation
- Close the main `aux` file for writing
- Load the main `aux` file written on the current LaTeX pass
- Perform final tests and issue warnings, if applicable
- Execute any `\AfterEndDocument` code

Any `\AtEndDocument` code may be considered as being part of the document body insofar as it is still possible to perform typesetting tasks and write to the main `aux` file when it gets executed. `\AfterEndDocument` code is not part of the document body. This hook is useful to evaluate the data in the `aux` file at the very end of a LaTeX pass.

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 se-

quence name. If ⟨*command*⟩ is undefined, ⟨*csname*⟩ will be undefined as well after the assignment. 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. If ⟨*csname*⟩ is undefined, the ⟨*command*⟩ will be undefined as well after the assignment. 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. If the second ⟨*csname*⟩ is undefined, the first ⟨*csname*⟩ will be undefined as well after the assignment. 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 in 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 cal-

culation. 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 a ⟨*command*⟩ once and prevents further expansion of the replacement text. This command is expandable.

`\csexpandonce{`⟨*csname*⟩`}`

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

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 `\outer` commands may not be patched. Also note that the commands in this section will not automatically issue any error messages if patching fails. Instead, they take a ⟨*failure*⟩ argument which should provide suitable fallback code or an error message. Issuing `\tracingpatches` in the preamble will cause the commands to write debugging information to the transcript file.

`\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. 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 the @ sign 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. If an optional ⟨*prefix*⟩ is specified, it replaces the prefixes of the ⟨*command*⟩. An empty ⟨*prefix*⟩ argument strips all prefixes from the ⟨*command*⟩. The assignment is local. This command implicitly performs the equivalent of an `\ifpatchable` test prior to patching. If this test succeeds, the command applies the patch and executes ⟨*success*⟩. If the test fails, it executes ⟨*failure*⟩ without modifying the original ⟨*command*⟩. This command is robust.

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

This command executes ⟨*true*⟩ if the ⟨*command*⟩ may be patched with `\patchcmd` and if the ⟨*search*⟩ pattern is found in its replacement text, and ⟨*false*⟩ otherwise. This command is robust.

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

Similar to `\ifpatchable` except that the starred variant does not require a search pattern. Use this version to check if a command may be patched with `\apptocmd` and `\pretocmd`.

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

This command appends ⟨*code*⟩ to the replacement text of a ⟨*command*⟩. If the ⟨*command*⟩ is a parameterless macro, it behaves like `\appto` from section 3.3.1. In contrast to `\appto`, `\apptocmd` may also be used to patch commands with parameters. In this case, it will detokenize the replacement text of the ⟨*command*⟩, apply the patch, and retokenize it under the current category code regime. The category code of the @ sign is temporarily set to 11. The ⟨*code*⟩ may refer to the parameters of the ⟨*command*⟩. The assignment is local. If patching succeeds, this command executes ⟨*success*⟩. If patching fails, it executes ⟨*failure*⟩ without modifying the original ⟨*command*⟩. This command is robust.

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

This command is similar to `\apptocmd` except that the ⟨*code*⟩ is inserted at the beginning of the replacement text of the ⟨*command*⟩. If the ⟨*command*⟩ is a parameterless macro, it behaves like `\preto` from section 3.3.1. In contrast to `\preto`, `\pretocmd` may also be used to patch commands with parameters. In this case, it will detokenize the replacement text of the ⟨*command*⟩, apply the patch, and reto-

kenize it under the current category code regime. The category code of the @ sign is temporarily set to 11. The ⟨*code*⟩ may refer to the parameters of the ⟨*command*⟩. The assignment is local. If patching succeeds, this command executes ⟨*success*⟩. If patching fails, it executes ⟨*failure*⟩ without modifying the original ⟨*command*⟩. This command is robust.

`\tracingpatches` Enables tracing for all patching commands, including `\ifpatchable`. The debugging information will be written to the transcript file. This is useful if the reason why a patch is not applied or `\ifpatchable` yields ⟨*false*⟩ is not obvious. This command must be issued in the preamble.

3.5 Boolean flags

This package provides two interfaces to boolean flags which are completely independent of each other. The facilities in section 3.5.1 are a LaTeX frontend to `\newif`. Those in section 3.5.2 use a different mechanism.

3.5.1 TeX flags

Since the facilities in this section are based on `\newif` internally, they may be used to test and alter the state of flags previously defined with `\newif`. They are also compatible with the boolean tests of the `ifthen` package and may serve as a LaTeX interface for querying TeX primitives such as `\ifmmode`. The `\newif` approach requires a total of three macros per flag.

`\newbool{`⟨*name*⟩`}`

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

`\providebool{`⟨*name*⟩`}`

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

`\booltrue{`⟨*name*⟩`}`

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

`\boolfalse{`⟨*name*⟩`}`

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

`\setbool{`⟨*name*⟩`}{`⟨*value*⟩`}`

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

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

Expands to ⟨*true*⟩ if the state of the boolean flag ⟨*name*⟩ is `true`, and to ⟨*false*⟩ otherwise. If the flag is undefined, this command issues an error. This command may be used to perform any boolean test based on plain TeX syntax, i. e., any test normally employed like this:

```
\iftest true\else false\fi
```

This includes all flags defined with `\newif` as well as TeX primitives such as `\ifmmode`. The `\if` prefix is omitted when using the flag or the primitive in the expression. For example:

```
\ifmytest true\else false\fi
\ifmmode true\else false\fi
```

becomes

```
\ifbool{mytest}{true}{false}
\ifbool{mmode}{true}{false}
```

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

Similar to `\ifbool` but negates the test.

3.5.2 LaTeX flags

In contrast to the flags from section 3.5.1, the facilities in this section require only one macro per flag. They also use a separate namespace to avoid name clashes with regular macros.

`\newtoggle{`⟨*name*⟩`}`

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

`\providetoggle{`⟨*name*⟩`}`

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

`\toggletrue{`⟨*name*⟩`}`

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

`\togglefalse{`⟨*name*⟩`}`

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

`\settoggle{`⟨*name*⟩`}{`⟨*value*⟩`}`

Sets the boolean flag ⟨*name*⟩ to ⟨*value*⟩ which may be either `true` or `false`. This

command is robust and may be prefixed with `\global`. It will issue an error if the flag is undefined.

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

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

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

Similar to `\iftoggle` but negates the test.

3.6 Generic tests

3.6.1 Macro tests

`\ifdef{`⟨*control sequence*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Expands to ⟨*true*⟩ if the ⟨*control sequence*⟩ is defined, and to ⟨*false*⟩ otherwise. Note that control sequences will be considered as defined even if their meaning is `\relax`. This command is a LaTeX wrapper for the e-TeX primitive `\ifdefined`.

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

Similar to `\ifdef` except that it takes a control sequence name as its first argument. This command is a LaTeX wrapper for the e-TeX primitive `\ifcsname`.

`\ifundef{`⟨*control sequence*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

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

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

Similar to `\ifundef` except that it takes a control sequence name as its first argument. This command may be used as a drop-in replacement for the `\@ifundefined` test in the LaTeX kernel.

`\ifdefmacro{`⟨*control sequence*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Expands to ⟨*true*⟩ if the ⟨*control sequence*⟩ is defined and is a macro, and to ⟨*false*⟩ otherwise.

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

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

`\ifdefparam{`⟨*control sequence*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Expands to ⟨*true*⟩ if the ⟨*control sequence*⟩ is defined and is a macro with one or more parameters, and to ⟨*false*⟩ otherwise.

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

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

`\ifdefprefix`{⟨*control sequence*⟩}{⟨*true*⟩}{⟨*false*⟩}

Expands to ⟨*true*⟩ if the ⟨*control sequence*⟩ is defined and is a macro prefixed with `\long` and/or `\protected`, and to ⟨*false*⟩ otherwise. Note that `\outer` macros may not be tested.

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

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

`\ifdefprotected`{⟨*control sequence*⟩}{⟨*true*⟩}{⟨*false*⟩}

Expands to ⟨*true*⟩ if the ⟨*control sequence*⟩ is defined and is a macro prefixed with `\protected`, and to ⟨*false*⟩ otherwise.

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

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

`\ifdefltxprotect`{⟨*control sequence*⟩}{⟨*true*⟩}{⟨*false*⟩}

Executes ⟨*true*⟩ if the ⟨*control sequence*⟩ is defined and is a LaTeX protection shell, and ⟨*false*⟩ otherwise. This command is robust. It will detect commands which have been defined with `\DeclareRobustCommand` or by way of a similar technique.

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

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

`\ifdefempty`{⟨*control sequence*⟩}{⟨*true*⟩}{⟨*false*⟩}

Expands to ⟨*true*⟩ if the ⟨*control sequence*⟩ is defined and is a parameterless macro whose replacement text is empty, and to ⟨*false*⟩ otherwise. In contrast to `\ifx`, this test ignores the prefixes of the ⟨*command*⟩.

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

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

`\ifdefvoid`{⟨*control sequence*⟩}{⟨*true*⟩}{⟨*false*⟩}

Expands to ⟨*true*⟩ if the ⟨*control sequence*⟩ is undefined, is a macro whose meaning is `\relax`, or is a parameterless macro whose 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`{⟨*control sequence*⟩}{⟨*control sequence*⟩}{⟨*true*⟩}{⟨*false*⟩}

Compares two control sequences and expands to ⟨*true*⟩ if they are equal in the sense of `\ifx`, and to ⟨*false*⟩ otherwise. In contrast to `\ifx`, this test will also yield ⟨*false*⟩ if both control sequences 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.

3.6.2 String tests

`\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.

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

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

`\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.

`\notblank`{⟨*string*⟩}{⟨*not true*⟩}{⟨*not false*⟩}

Similar to `\ifblank` but negates the test.

3.6.3 Arithmetic tests

`\ifnumcomp`{⟨*integer expression*⟩}{⟨*relation*⟩}{⟨*integer expression*⟩}{⟨*true*⟩}{⟨*false*⟩}

Compares two integer expressions according to ⟨*relation*⟩ and expands to ⟨*true*⟩ or ⟨*false*⟩ depending on the result. The ⟨*relation*⟩ may be <, >, or =. Both integer expressions will be processed with `\numexpr`. An ⟨*integer expression*⟩ may be any arbitrary code which is valid in this context. All arithmetic expressions may contain spaces. Here are some examples:

```
\ifnumcomp{3}{>}{6}{true}{false}
\ifnumcomp{(7 + 5) / 2}{=}{6}{true}{false}
\ifnumcomp{(7+5) / 4}{>}{3*(12-10)}{true}{false}
\newcounter{countA}
\setcounter{countA}{6}
\newcounter{countB}
\setcounter{countB}{5}
\ifnumcomp{\value{countA} * \value{countB}/2}{=}{15}{true}{false}
\ifnumcomp{6/2}{=}{5/2}{true}{false}
```

Technically, this command is a LaTeX wrapper for the TeX primitive `\ifnum`, incorporating `\numexpr`. Note that `\numexpr` will round the result of all integer expressions, i. e., both expressions will be processed and rounded prior to being compared. In the last line of the above examples, the result of the second expression is 2.5, which is rounded to 3, hence `\ifnumcomp` will expand to ⟨*true*⟩.

`\ifnumequal`{⟨*integer expression*⟩}{⟨*integer expression*⟩}{⟨*true*⟩}{⟨*false*⟩}

Alternative syntax for `\ifnumcomp{...}{=}{...}{...}{...}`.

`\ifnumgreater`{⟨*integer expression*⟩}{⟨*integer expression*⟩}{⟨*true*⟩}{⟨*false*⟩}

Alternative syntax for `\ifnumcomp{...}{>}{...}{...}{...}`.

`\ifnumless`{⟨*integer expression*⟩}{⟨*integer expression*⟩}{⟨*true*⟩}{⟨*false*⟩}

Alternative syntax for `\ifnumcomp{...}{<}{...}{...}{...}`.

`\ifnumodd`{⟨*integer expression*⟩}{⟨*true*⟩}{⟨*false*⟩}

Evaluates an integer expression and expands to ⟨*true*⟩ if the result is an odd number, and to ⟨*false*⟩ otherwise. Technically, this command is a LaTeX wrapper for the TeX primitive `\ifodd`, incorporating `\numexpr`.

`\ifdimcomp`{⟨*dimen expression*⟩}{⟨*relation*⟩}{⟨*dimen expression*⟩}{⟨*true*⟩}{⟨*false*⟩}

Compares two dimen expressions according to ⟨*relation*⟩ and expands to ⟨*true*⟩ or ⟨*false*⟩ depending on the result. The ⟨*relation*⟩ may be <, >, or =. Both dimen expressions will be processed with `\dimexpr`. A ⟨*dimen expression*⟩ may be any arbitrary code which is valid in this context. All arithmetic expressions may contain spaces. Here are some examples:

```
\ifdimcomp{1cm}{=}{28.45274pt}{true}{false}
\ifdimcomp{(7pt + 5pt) / 2}{<}{2pt}{true}{false}
```

```
\ifdimcomp{(3.725pt + 0.025pt) * 2}{<}{7pt}{true}{false}
\newlength{\lengthA}
\setlength{\lengthA}{7.25pt}
\newlength{\lengthB}
\setlength{\lengthB}{4.75pt}
\ifdimcomp{(\lengthA + \lengthB) / 2}{>}{2.75pt * 2}{true}{false}
\ifdimcomp{(\lengthA + \lengthB) / 2}{>}{25pt / 6}{true}{false}
```

Technically, this command is a LaTeX wrapper for the TeX primitive `\ifdim`, incorporating `\dimexpr`. Since both `\ifdimcomp` and `\ifnumcomp` are expandable, they may also be nested:

```
\ifnumcomp{\ifdimcomp{5pt+5pt}{=}{10pt}{1}{0}}{>}{0}{true}{false}
```

`\ifdimequal{`⟨*dimen expression*⟩`}{`⟨*dimen expression*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Alternative syntax for `\ifdimcomp{...}{=}{...}{...}{...}`.

`\ifdimgreater{`⟨*dimen expression*⟩`}{`⟨*dimen expression*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Alternative syntax for `\ifdimcomp{...}{>}{...}{...}{...}`.

`\ifdimless{`⟨*dimen expression*⟩`}{`⟨*dimen expression*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Alternative syntax for `\ifdimcomp{...}{<}{...}{...}{...}`.

3.6.4 Boolean expressions

The commands in this section are replacements for the `\ifthenelse` command provided by the `ifthen` package. They serve the same purpose but differ in syntax, concept, and implementation. In contrast to `\ifthenelse`, they do not provide any tests of their own but serve as a frontend to other tests. Any test which satisfies certain syntactical requirements may be used in a boolean expression.

`\ifboolexpr{`⟨*expression*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Evaluates the ⟨*expression*⟩ and executes ⟨*true*⟩ if it is true, and ⟨*false*⟩ otherwise. The ⟨*expression*⟩ is evaluated sequentially from left to right. The following elements, discussed in more detail below, are available in the ⟨*expression*⟩: the test operators `togl`, `bool`, `test`; the logical operators `not`, `and`, `or`; and the subexpression delimiter `(...)`. Spaces, tabs, and line endings may be used freely to arrange the ⟨*expression*⟩ visually. Blank lines are not permissible in the ⟨*expression*⟩. This command is robust.

`\ifboolexpe{`⟨*expression*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

An expandable version of `\ifboolexpr` which may be processed in an expansion-only context, e. g., in an `\edef` or in a `\write` operation. Note that all tests used in the ⟨*expression*⟩ must be expandable, even if `\ifboolexpe` is not located in an expansion-only context.

`\whileboolexpr{`⟨*expression*⟩`}{`⟨*code*⟩`}`

Evaluates the ⟨*expression*⟩ like `\ifboolexpr` and repeatedly executes the ⟨*code*⟩

while the expression is true. The ⟨*code*⟩ may be any valid TeX or LaTeX code. This command is robust.

`\unlessboolexpr{`⟨*expression*⟩`}{`⟨*code*⟩`}`

Similar to `\whileboolexpr` but negates the ⟨*expression*⟩, i. e., it keeps executing the ⟨*code*⟩ repeatedly unless the expression is true. This command is robust.

The following test operators are available in the ⟨*expression*⟩:

`togl` Use the `togl` operator to test the state of a flag defined with `\newtoggle`. For example:

```
\iftoggle{mytoggle}{true}{false}
```

becomes

```
\ifboolexpr{ togl {mytoggle} }{true}{false}
```

The `togl` operator may be used with both `\ifboolexpr` and `\ifboolexpe`.

`bool` Use the `bool` operator to perform a boolean test based on plain TeX syntax, i. e., any test normally employed like this:

```
\iftest true\else false\fi
```

This includes all flags defined with `\newif` as well as TeX primitives such as `\ifmmode`. The `\if` prefix is omitted when using the flag or the primitive in the expression. For example:

```
\ifmmode true\else false\fi
\ifmytest true\else false\fi
```

becomes

```
\ifboolexpr{ bool {mmode} }{true}{false}
\ifboolexpr{ bool {mytest} }{true}{false}
```

This also works with flags defined with `\newbool` (see § 3.5.1). In this case

```
\ifbool{mybool}{true}{false}
```

becomes

```
\ifboolexpr{ bool {mybool} }{true}{false}
```

The `bool` operator may be used with both `\ifboolexpr` and `\ifboolexpe`.

`test` Use the `test` operator to perform a test based on LaTeX syntax, i. e., any test normally employed like this:

```
\iftest{true}{false}
```

This applies to all macros based on LaTeX syntax, i. e., the macro must take a ⟨*true*⟩ and a ⟨*false*⟩ argument and these must be the final arguments. For example:

```
\ifdef{\somemacro}{true}{false}
\ifdimless{\textwidth}{365pt}{true}{false}
\ifnumcomp{\value{somecounter}}{>}{3}{true}{false}
```

When using such tests in the ⟨*expression*⟩, their ⟨*true*⟩ and ⟨*false*⟩ arguments are omitted. For example:

```
\ifcsdef{mymacro}{true}{false}
```

becomes

```
\ifboolexpr{ test {\ifcsdef{mymacro}} }{true}{false}
```

and

```
\ifnumcomp{\value{mycounter}}{>}{3}{true}{false}
```

becomes

```
\ifboolexpr{
  test {\ifnumcomp{\value{mycounter}}{>}{3}}
}
{true}
{false}
```

The `test` operator may be used with `\ifboolexpr` without any restrictions. It may also be used with `\ifboolexpe`, provided that the test is expandable. Some of the generic tests in § 3.6 are robust and may not be used with `\ifboolexpe`, even if `\ifboolexpe` is not located in an expansion-only context. Use `\ifboolexpr` instead if the test is not expandable.

Since `\ifboolexpr` and `\ifboolexpe` imply processing overhead, there is generally no benefit in employing them for a single test. The stand-alone tests in § 3.6 are more efficient than `test`, `\ifbool` from § 3.5.1 is more efficient than `bool`, and `\iftoggle` from § 3.5.2 is more efficient than `togl`. The point of `\ifboolexpr` and `\ifboolexpe` is that they support logical operators and subexpressions. The following logical operators are available in the ⟨*expression*⟩:

`not` The `not` operator negates the truth value of the immediately following element. You may prefix `togl`, `bool`, `test`, and subexpressions with `not`. For example:

```
\ifboolexpr{
  not bool {mybool}
}
{true}
{false}
```

will yield ⟨*true*⟩ if `mybool` is false and ⟨*false*⟩ if `mybool` is true, and

```
\ifboolexpr{
  not ( bool {boolA} and bool {boolB} )
}
{true}
{false}
```

will yield ⟨*true*⟩ if both `boolA` and `boolB` are false.

`and` The `and` operator expresses a conjunction (both *a* and *b*). The ⟨*expression*⟩ is true if all elements joined with `and` are true. For example:

```
\ifboolexpr{
  bool {boolA} and bool {boolB}
}
{true}
{false}
```

will yield ⟨*true*⟩ if both `bool` tests are true. The `nand` operator (negated `and`, i. e., not both) is not provided as such but may be expressed by using `and` in a negated subexpression. For example:

```
bool {boolA} nand bool {boolB}
```

may be written as

```
not ( bool {boolA} and bool {boolB} )
```

`or` The `or` operator expresses a non-exclusive disjunction (either *a* or *b* or both). The ⟨*expression*⟩ is true if at least one of the elements joined with `or` is true. For example:

```
\ifboolexpr{
  togl {toglA} or togl {toglB}
}
{true}
{false}
```

will yield ⟨*true*⟩ if either `togl` test or both tests are true. The `nor` operator (negated non-exclusive disjunction, i. e., neither *a* nor *b* nor both) is not provided as such but may be expressed by using `or` in a negated subexpression. For example:

```
bool {boolA} nor bool {boolB}
```

may be written as

```
not ( bool {boolA} or bool {boolB} )
```

`(...)` The parentheses delimit a subexpression in the ⟨*expression*⟩. The subexpression is evaluated and the result of this evaluation is treated as a single truth value in the enclosing expression. Subexpressions may be nested. For example, the expression:

```
( bool {boolA} or bool {boolB} )
and
( bool {boolC} or bool {boolD} )
```

is true if both subexpressions are true, i. e., if at least one of `boolA`/`boolB` and at least one of `boolC`/`boolD` is true. Note that subexpressions are generally not required if all elements are joined with `and` or with `or`. For example, the expressions

```
bool {boolA} and bool {boolB} and {boolC} and bool {boolD}
bool {boolA} or bool {boolB} or {boolC} or bool {boolD}
```

will yield the expected results: the first one is true if all elements are true; the second one is true if at least one element is true. However, when combining `and` and `or`, it is advisable to always group the elements in subexpressions in order

to avoid potential misconceptions which may arise from differences between the semantics of formal boolean expressions and the semantics of natural languages. For example, the following expression

```
bool {bagel} and bool {ham} or bool {cheese}
```

is always true if `cheese` is true since the `or` operator will take the result of the `and` evaluation as input. In contrast to the meaning of this expression when pronounced in English, it is not processed like this

```
bool {bagel} and ( bool {ham} or bool {cheese} )
```

but evaluated strictly from left to right:

```
( bool {bagel} and bool {ham} ) or bool {cheese}
```

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 may be preferable as they allow testing if an element is in a list.

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

This command defines a list parser similar to the `\docsvlist` command below, which is defined like this:

```
\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 in 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*⟩. A blank ⟨*item*⟩ is not added to the list.

`\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 ⟨*listmacro*⟩ is not. If the expanded ⟨*item*⟩ is blank, it is not added to the list.

`\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 is safer 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 arithmetic tests:

```
\rmntonum{mcmxcv}
\setcounter{counter}{\rmntonum{CXVI}}
\ifnumless{\rmntonum{mcmxcviii}}{2000}{true}{false}
```

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` would yield `5`, `IC` would yield `99`, etc.

`\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, but it will not check if they really are a valid Roman numeral.

4 Revision history

This revision history is a list of changes relevant to users of this package. Changes of a more technical nature which do not affect the user interface or the behavior of the package are not included in the list. If an entry in the revision history states that a feature has been *improved* or *extended*, this indicates a syntactically backwards compatible modification, such as the addition of an optional argument to an existing command. Entries stating that a feature has been *modified* demand attention. They indicate a modification which may require changes to existing documents in some, hopefully rare, cases. The numbers on the right indicate the relevant section of this manual.

1.9 2010-04-10

1.8 2009-08-06

Added \ifnumequal 3.6.3
Added \ifnumgreater 3.6.3
Added \ifnumless 3.6.3
Added \ifdimcomp 3.6.3
Added \ifdimequal 3.6.3
Added \ifdimgreater 3.6.3
Added \ifdimless 3.6.3

1.7 2008-06-28

Renamed \AfterBeginDocument to \AfterEndPreamble (name clash) . 2.5
Resolved conflict with hyperref
Rearranged manual slightly

1.6 2008-06-22

Improved \robustify 2.2
Improved \patchcmd and \ifpatchable 3.4
Modified and improved \apptocmd 3.4
Modified and improved \pretocmd 3.4
Added \ifpatchable* 3.4
Added \tracingpatches 3.4
Added \AfterBeginDocument 2.5
Added \ifdefmacro 3.6.1
Added \ifcsmacro 3.6.1
Added \ifdefprefix 3.6.1
Added \ifcsprefix 3.6.1
Added \ifdefparam 3.6.1
Added \ifcsparam 3.6.1
Added \ifdefprotected 3.6.1
Added \ifcsprotected 3.6.1
Added \ifdefltxprotect 3.6.1
Added \ifcsltxprotect 3.6.1
Added \ifdefempty 3.6.1
Added \ifcsempty 3.6.1
Improved \ifdefvoid 3.6.1
Improved \ifcsvoid 3.6.1
Added \ifstrempty 3.6.2
Added \setbool 3.5.1
Added \settoggle 3.5.2

1.5 2008-04-26

Added \defcounter 2.4
Added \deflength 2.4
Added \ifdefstring 3.6.1
Added \ifcsstring 3.6.1
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) 3.5.2
Renamed `\provideswitch` to `\providetoggle` (consistency) 3.5.2
Renamed `\switchtrue` to `\toggletrue` (consistency) 3.5.2
Renamed `\switchfalse` to `\togglefalse` (consistency) 3.5.2
Renamed `\ifswitch` to `\iftoggle` (consistency) 3.5.2
Renamed `\notswitch` to `\nottoggle` (consistency) 3.5.2
Added `\AtEndPreamble` 2.5
Added `\AfterEndDocument` 2.5
Added `\AfterPreamble` 2.5
Added `\undef` 3.1.1
Added `\csundef` 3.1.1
Added `\ifdefvoid` 3.6.1
Added `\ifcsvoid` 3.6.1
Added `\ifdefequal` 3.6.1
Added `\ifcsequal` 3.6.1
Added `\ifstrequal` 3.6.2
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` 3.5.1
Added `\providebool` 3.5.1
Added `\booltrue` 3.5.1

Added \boolfalse 3.5.1
Added \ifbool 3.5.1
Added \notbool 3.5.1
Added \newswitch 3.5.2
Added \provideswitch 3.5.2
Added \switchtrue 3.5.2
Added \switchfalse 3.5.2
Added \ifswitch 3.5.2
Added \notswitch 3.5.2
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