The `toolbox` package*

Martin Väth†‡

2013/11/26

Abstract

The TEX programmer's toolbox; enhanced version. This package provides some macros which are convenient for writing indices, glossaries, or other macros. It contains macros which support

- implicit macros – a useful supplement to `\index` or `varindex` for writing glossaries or indices.
- fancy optional arguments
- loops over tokenlists and itemlists
- searching, splitting, and replacing
- controlled expansion
- redefinition of macros
- concatenated macro names

Contents

*This package has version number 5.1, last revised 2004/04/29. The package may be distributed and/or modified under the conditions of the LaTeX Project Public License, either version 1.3c of this license or (at your option) any later version. The latest version of this license is in `http://www.latex-project.org/lppl.txt`, and version 1.3c or later is part of all distributions of LaTeX version 2005/12/01 or later.

†`martin@mvath.de`

‡The author thanks David Kastrup `<David.Kastrup@neuroinformatik.ruhr-uni-bochum.de>`.

1 Changes

v5.1 (*2013/11/26*) Update email, add license. Date/version remains the same since only commments and documentation are modified.

v5.1 (*2004/04/29*) Added \toolboxReplace, \toolboxReplaceSplit. Some corrections in the documentation. Added some forgotten \long's in the code..

v4.4 (*2003/10/07*) Added \toolboxIfEmpty, \toolboxIfx, \toolboxIfX, \toolboxIfElse. Some corrections in the documentation.

v4.3 (*2002/09/27*) Major enhancements in documentation: Added lots of examples and explanation about typical applications. Bugfix in \toolboxMakeDef: \⟨*Prefix*⟩Provide⟨*Name*⟩ now behaves like \providecommand as documented. The old meaning is now called \⟨*Prefix*⟩Def⟨*Name*⟩. Introduced \toolboxGobbleArg.

v4.2 (*2001/09/26*) Treating # properly now also in \toolboxMakeDef.

v4.1 (*2001/09/24*) Took more care of treating # properly when this symbol occurs in arguments of certain macros (the treatment is not completely downward compatible; for this reason the major release number has been changed). For this reason, also the new macro \toolboxTokDef is provided. I thank David Kastrup <David.Kastrup@neuroinformatik.ruhr-uni-bochum.de> for pointing out this problem (and a solution) to me.

v3.3 (*2001/08/19*) Eliminated a bug in \toolboxMakeSplit*.

v3.2 (*2001/05/08*) Eliminated a serious bug in \toolboxIf. Due to this bugfix, the usage of \toolboxIf had to be slightly restricted.

v3.1 (*2001/05/06*) Major advantage: \toolboxMakeDef implemented.

Reimplemented \newif (apparently by mistake, this had been declared as \outer in TEX and LATEX2.09 which caused an error during loading of toolbox.sty). For this reason, also \toolboxNewifTrue and \toolboxNewifFalse were introduced.

v2.1 (*2001/04/30*) Introduced `\toolboxIf`, `\toolboxAppend`, and `\toolboxSurround`.

v2.0 (*2001/04/08*) Many major enhancements and new tools: Added section (and corresponding macros) for fancy optional arguments. Made `\toolboxLoop` semi-reentrant by introducing `\toolboxLoopName`. Added `\toolboxTokenLoop` and friends. Added `\toolboxSpaceToken`. Made many macros `\long` and added `\long` versions of `\toolboxSplitAt` and `\toolboxMakeSplit`.

v1.0 (*2001/03/29*) First release.

2 Installation

This package was tested with TEX, LATEX2.09, and LATEX 2_ε, and it should actually run with all other TEX formats.
To use `toolbox`, you have to put the file `toolbox.sty` in a path where TEX looks for its input files. The TEX documents using `toolbox` need the following modifications in their header:

- If you use LATEX 2_ε, put in the preamble the command

  ```
  \usepackage{toolbox}
  ```

- If you use LATEX2.09, use `toolbox` as a style option, e.g.

  ```
  \documentstyle[toolbox]{article}
  ```

  or

  ```
  \documentstyle[toolbox,12pt]{article}
  ```

- If you use some other (non-LATEX) format, you will probably have to insert a line like

  ```
  \catcode`\@=11\input toolbox.sty\catcode`\@=12\relax
  ```

The only LATEX-specific commands used in `toolbox.sty` are:

- `\newcommand` (used only in the form `\newcommand{`⟨*command*⟩`}{}` to ensure that ⟨*command*⟩ was not defined before)
- `\ProvidesPackage`
- `\typeout`

The above commands are used only if they are defined.

3 Description of the macros

General remark: Many macros could appear in several sections. For example, `\toolboxMakeDef` and `\toolboxSourround` might be considered as macros which support redefinitions of macros. However, we put them in different sections which perhaps explain better their nature.

3.1 Implicit macro definitions

What we mean by implicit macro definitions is probably best explained by the following examples which show the intended usage:

(In the following examples, we always refer to the `\index` command. Note that it may be more convenient to use for indices the `varindex` package in addition – the documentation of `varindex` (release 2.3 or newer) gives additional hints and examples how these two (essentially independent) approaches can be combined in practice).

Assume that you want to write an index for a book which has rather long and complicated `\index` entries. The first idea that one might have in this connection is to put the various `\index` commands at the beginning of the document into several macros (one for each `\index` entry), and to use just these macros in the main text. For example, one might want to write near the beginning of the document commands like

```
\newcommand{\Start}{\index{finish or end}}
\newcommand{\End}{\index{finish or end}}
```

and then to use in the main text `\Start` and `\End` whenever a reference in the corresponding index to the current place is desired. However, this has two major disadvantages:

1. It is easy to forget that `\End` writes an index entry. So the macro `\End` in the main text might be very confusing.
2. You cannot choose short and intuitive macro names for common phrases, because they are usually already reserved by TEX, LATEX, or some packages. For example, `\end` could not be used.

To avoid these problems, one may be very disciplinary and call the involved macros systematically e.g. `\GlossaryStart` `\GlossaryEnd` etc. However, this produces terrible long and unreadable macro names in the main text.

The implicit macro definitions of `toolbox` provide a more convenient solution. The idea is that you do not use the corresponding macros directly but only implicitly by a call of other macros where your "macro name" is just an argument.
Moreover, `toolbox` assists you in writing the corresponding definitions. For example, if you know that you want a set of macros which all expand into something

of the form `\index{...}`, you can give a "mask" which contains this form, and you only have to fill in the changing content (similarly as for usual TEX macros with arguments, but the level of abstraction is one step higher). For the above task, you might use the command:

```
\toolboxMakeDef{Glossary}{\index{#1}}
```

The argument `Glossary` serves to distinguish independent definitions (this will become clear later). Its effect visible now is that it determines the name of the following macros which you can use after the above call:

```
\NewGlossary{start-1}{start}
\NewGlossary{start-2}{start or beginning}
\NewGlossary{end}{finish or end}
```

These command are now similar to the `\newcommand` definitions explained above. However, there is no name collision with the TEX-internal command `\end`. Of course, this means that you cannot just write `\end` in the main text to get the desired index entry. Instead, you have to write the more intuitive commands

```
\Glossary{start-1}
\Glossary{start-2}
\Glossary{end}
```

(again, the name `\Glossary` stems from our first call of `\toolboxMakeDef`). Note that e.g. `\Glossary{start-1}` expands not only to `start` but actually to `\index{start}` (because of our first call of `\toolboxMakeDef`).
Note also that you can use symbols like "-" or numbers which are usually not allowed in TEX macro names.
Of course, similarly as for `\newcommand`, you can also do other things with the macros. For example,

```
\LetGlossary\tempname{end}
\NewGlossary*{finish}\tempname
```

will first define `\tempname` to expand to the same text as `\Glossary{end}`, and then defines a new entry `\Glossary{finish}` to expand to the same text as `\tempname`. Hence, the above two lines make the calls `\Glossary{end}` and `\Glossary{finish}` equivalent.
At the end of your list of `\NewGlossary` commands, you might want to put

```
\toolboxFreeDef*{Glossary}
```

The purpose of this command is that `\NewGlossary` cannot be used anymore (unless, of course, you define it again). So you cannot unintentionally add new entries to your glossary list (but you still can use `\Glossary{...}` to reference to the already produced entries). Moreover, the above command frees some memory which was needed for `\NewGlossary` to work.
If you additionally want to free the memory used by `\Glossary`, you can use

```
\toolboxFreeDef{Glossary}
```

(without the *). This may be necessary, if you want to call again e.g.

```
\toolboxMakeDef{Glossary}{\emph{#1}\index{#1}}
```

(if you have not freed the memory for \Glossary before this repeated call, TEX will complain that \Glossary is already defined).

Of course, it is possible to call \toolboxMakeDef with several different names, for example, for \Glossary, \SymbolList etc. Another application might be to use a different command to mark e.g. the main occurrence of some index entry or to output additionally the entry into the running text. We do this in the following example which simultaneously demonstrates that the names can also be constructed in another way:

```
\toolboxMakeDef[Ind]{}{\index{#1}}
\toolboxMakeDef[Ind]{Main}{\index{#1|textbf}}
\toolboxMakeDef{OutInd}{#1}

\IndNew{A}{A is a letter}
\IndNewMain{A}{A is a letter}
\NewOutInd{A}{\textbf{The letter A}\Ind{A}}
```

After the above commands, you can use \Ind{A}, \IndMain{A}, and \OutInd{A} to produce the corresponding \index entry, the "main" \index entry (with a fat page number), and the text **The letter A** with an additional entry into the index, respectively. Of course, it might usually be more convenient to define the \Ind and \IndMain entries simultaneously, e.g. by the commands

```
\toolboxMakeDef[Ind]{}{#1}
\toolboxMakeDef[Ind]{Main}{#1}
\newcommand{\NewStandardInd}[2]{%
   \IndNew{#1}{\index{#2}}%
   \IndNewMain{#1}{\index{#2|textbf}}}

\NewStandardInd{A}{A is a letter}
```

This approach has the additional advantage that you can define exceptional cases "by hand" (e.g. if you want that for certain "main" index entries the page number is printed with \textsl instead of \textbf).

Since the motivation for implicit definitions now is hopefully clear, let us now describe in detail which commands are provided by toolbox for this purpose. As explained in the example, the main generic macro provided to this purpose is

\toolboxMakeDef

\toolboxMakeDef. Its call syntax is as follows:

\toolboxMakeDef[⟨*Prefix*⟩]{⟨*Name*⟩}{⟨*Replacement mask*⟩}

(the argument [⟨*Prefix*⟩] is optional and by default empty). The above command generates new macros

\⟨*Prefix*⟩`New`⟨*Name*⟩
\⟨*Prefix*⟩`Renew`⟨*Name*⟩
\⟨*Prefix*⟩`Provide`⟨*Name*⟩
\⟨*Prefix*⟩`Def`⟨*Name*⟩
\⟨*Prefix*⟩`Let`⟨*Name*⟩
\⟨*Prefix*⟩⟨*Name*⟩

which in turn can be called as follows:

\⟨*Prefix*⟩`New`⟨*Name*⟩{⟨*something*⟩}{⟨*text to remember*⟩}
\⟨*Prefix*⟩`Renew`⟨*Name*⟩{⟨*something*⟩}{⟨*text to remember*⟩}
\⟨*Prefix*⟩`Provide`⟨*Name*⟩{⟨*something*⟩}{⟨*text to remember*⟩}
\⟨*Prefix*⟩`Def`⟨*Name*⟩{⟨*something*⟩}{⟨*text to remember*⟩}
\⟨*Prefix*⟩`New`⟨*Name*⟩`*`{⟨*something*⟩}\⟨*SomeMacro*⟩
\⟨*Prefix*⟩`Renew`⟨*Name*⟩`*`{⟨*something*⟩}\⟨*SomeMacro*⟩
\⟨*Prefix*⟩`Provide`⟨*Name*⟩`*`{⟨*something*⟩}\⟨*SomeMacro*⟩
\⟨*Prefix*⟩`Def`⟨*Name*⟩`*`{⟨*something*⟩}\⟨*SomeMacro*⟩
\⟨*Prefix*⟩`Let`⟨*Name*⟩\⟨*SomeMacro*⟩{⟨*something*⟩}
\⟨*Prefix*⟩⟨*Name*⟩{⟨*something*⟩}
\⟨*Prefix*⟩⟨*Name*⟩`*`{⟨*something*⟩}

These calls are in a sense similar to the respective commands

`\newcommand{`\⟨*something*⟩`}`{⟨*text to remember*⟩}
`\renewcommand{`\⟨*something*⟩`}`{⟨*text to remember*⟩}
`\providecommand{`\⟨*something*⟩`}`{⟨*text to remember*⟩}
`\def`\⟨*something*⟩{⟨*text to remember*⟩}
`\newcommand{`\⟨*something*⟩`}{}\let`\⟨*something*⟩\⟨*SomeMacro*⟩
`\renewcommand{`\⟨*something*⟩`}{}\let`\⟨*something*⟩\⟨*SomeMacro*⟩
`\@ifundefined{`⟨*something*⟩`}{\let`\⟨*something*⟩\⟨*SomeMacro*⟩`}{}`
`\let`\⟨*something*⟩\⟨*SomeMacro*⟩
`\let`\⟨*SomeMacro*⟩\⟨*something*⟩
\⟨*something*⟩
\⟨*something*⟩ (but without error if \⟨*something*⟩ is undefined)

with the differences already pointed out before:

1. The macro name actually used is not \⟨*something*⟩. Instead, it is a name which does not conflict with any existing macro (except one generated previously by another \⟨*Prefix*⟩New⟨*Name*⟩, but in this case a descriptive error is reported).

   For this reason, it is not possible to use this macro directly but only indirectly by the call \⟨*Prefix*⟩⟨*Name*⟩{⟨*something*⟩} (or with \⟨*Prefix*⟩Let⟨*Name*⟩).

2. The replacement text is not ⟨*text to remember*⟩ but determined by ⟨*Replacement Mask*⟩ where every occurrence of `#1` in ⟨*Replacement Mask*⟩ is replaced by ⟨*text to remember*⟩ (recall the examples). If you want to have the plain ⟨*text to remember*⟩, use `{#1}` as ⟨*Replacement Mask*⟩.

Since `toolbox 4.2` there is another slight difference: The symbol `#` is treated as usual and not as in a macro definition.
`\toolboxMakeDef` gives an error message if the commands \⟨*Prefix*⟩... are already defined. If you intentionally want to change a previous definition, you have to call the command

`\toolboxFreeDef` `\toolboxFreeDef[`⟨*Prefix*⟩`]{`⟨*Name*⟩`}`

before. The latter not only lets all of the macros \⟨*Prefix*⟩... be `\undefined`, but also frees all other memory internally used by the corresponding call of `\toolboxMakeDef` (note, however, that the above command does not free the memory allocated before by calls of \⟨*Prefix*⟩`New...` – to free the latter, you have to call subsequently e.g.

\⟨*Prefix*⟩`New`⟨*Name*⟩`*{...}{\undefined}`

before). There is also the command

`\toolboxFreeDef*[`⟨*Prefix*⟩`]{`⟨*Name*⟩`}`

which acts similarly as `\toolboxFreeDef` but which does not undefine the two macros \⟨*Prefix*⟩⟨*Name*⟩ and \⟨*Prefix*⟩`Let`⟨*Name*⟩.

3.2 Fancy optional argument parsing

This section contains macros which are convenient if you e.g. write a package that contains macros which contain a lot of optional arguments and flags (like `*`). Typically, to read such an optional argument or flag, you save the next token with `\futurelet` and then call a macro which decides what to do with the token read. Thus, a typical use of `\futurelet` looks like

```
\def\MacroWithOptionalFlag{\futurelet\tokread\myscan}
```

which will define `\tokread` to be the token *following* the macro `\MacroWithOptionalFlag` in the token stream and then execute `\myscan`. In this context, it is not very convenient that you are *forced* to define a macro `\myscan`: It could be more convenient if you could just write the *content* of `\tokread` (in braces) into the above definition. You can indeed do this if you
`\toolboxFuturelet` replace `\futurelet` by `\toolboxFuturelet`:

`\toolboxFuturelet\token{`⟨*argument*⟩`}`

The call `\toolboxFuturelet\token{\command}` has precisely the same effect as `\futurelet\token\command`. The advantage of `\toolboxFuturelet` is that instead of a single `\command` one may use also a sequence of commands.
Let us consider `\MacroWithOptionalFlag` as above. Assume that the user has called this macro in the form `\MacroWithOptionalFlag*` where the `*` is a flag which should cause your macro to do something slightly different. On some place in your macro definition you will have recognized (e.g. with `\futurelet`

or \toolboxFuturelet) that a * is following in the calling sequence. So you now want to execute your action (whatever \MacroWithOptionalFlag is supposed to do). However, if you do not take special care, after this action, TEX will print a *, because this is the next token on the token stream: \futurelet does not delete any tokens. So you have to "gobble" this token away. A rude way to do this is by using the macro \gobblenext as the last token in you macro which can be defined by

```
\def\gobblenext#1{}
```

However, this has two major drawbacks:

1. This works for *, but not for { or space tokens. For space tokens the situation is even worse, since TEX eats spaces around arguments, so sometimes space tokens might unexpectedly disappear.

2. It is not possible in this way to *read* another argument following the *: Recall that \gobblenext must be the *last* token in your macro expansion, i.e. you have "lost control" after this call.

\toolboxGobbleNext

The solution to these problems is instead of calling \gobblenext to use \toolboxGobbleNext as the last command in your call: You can pass it an argument which describes the action that you want to do *after* gobbling the next token (* in the above example) from the token stream. Thus

\toolboxGobbleNext{⟨*cmd*⟩}

erases the token following that command from the token stream and then executes ⟨*cmd*⟩. This is similar to

```
\def\toolboxGobbleNext#1#2{#1}
```

with the difference that #2 is considered as a token and that no spaces are eaten. The effect is that e.g. the call

```
\toolboxGobbleNext{\foo}{{arg}
```

is the same as \foo{arg} (the brace { is eaten in this example).
As described earlier, the commands \futurelet or \toolboxFuturelet can be used to check for optional flags. Frequently you will only want to test for one particular flag and decide the next action on this flag. Of course, you can test the token found with \ifx...\fi but this has the disadvantage that some tokens (e.g. \fi) follow your action, which might be bad (recall that e.g. \toolboxGobbleNext must be the last command of your action, i.e. it would in the above examples not gobble the * but the \fi which is probably not what you want). The simplest solution is to use the command \toolboxIfNextToken which already has the test included. For example, to test for an optional [, you can simply write

\toolboxIfNextToken

```
\def\MacroWithOptionalBrace{\toolboxIfNextToken[{\yes}{\no}}
```

and then the call `\MacroWithOptionalBrace[...]` will expand to `\yes[...]` while `\MacroWithOptionalBrace x` will expand to `\no x` (note that the brace is not gobbled—if you want the latter, use `\toolboxIfNextGobbling` described below).
More precisely, the calling syntax of `\toolboxIfNextToken` is

`\toolboxIfNextToken` `\toolboxIfNextToken{`⟨*token*⟩`}{`⟨*if*⟩`}{`⟨*else*⟩`}`

The semantic is the following: If the token following this command is ⟨*token*⟩, then ⟨*if*⟩ is executed, otherwise ⟨*else*⟩. It is explicitly admissible that ⟨*token*⟩ is a space. To support further tests, `\toolboxToken` is `\let` to the token which follows the command. `\toolboxToken` is only a temporary token, i.e. it may also be modified by other commands of this package; in particular, you may also freely modify `\toolboxToken`.

`\toolboxToken`

The token `\toolboxSpaceToken` which is described later may be handy in connection with this command.
In contrast to similar LaTeX 2ε macros much care has been taken that spaces are not eaten. This solves the following problem:
Assume that you want to write a macro which should have the calling syntax `\mymacro{arg1}` or `\mymacro{arg1}[arg2]`. You will probably implement `\mymacro` to read the first argument and then to look whether the next token is a `[`. If you use the LaTeX 2ε macro to test for `[`, then all spaces until the next non-space token would be gobbled which means that if you would use the LaTeX 2ε macros for the test, then the call `\mymacro{arg1}␣Text` would behave like `\mymacro{arg1}Text`, i.e. the space is "mysteriously" lost. With the `toolbox` macros this does not happen. The "disadvantage" is that `\mymacro{arg1}␣[arg2]` is not the same as `\mymacro{arg1}[arg2]` either (which is reasonable IMHO).
Example:

```
\def\mycmd#1{\toolboxIfNextToken[{\ParseOpt{#1}}{\NoOpt{#1}}}
   \def\ParseOpt#1[#2]{\OptAtEnd{#1}{#2}}
```

After the above definition, `\mycmd{arg}` executes `\NoOpt{arg}` while `\mycmd{arg}[optional]` executes `\OptAtEnd{arg}{optional}`. We point out once more that in the first call a space following `\mycmd{arg}` does not vanish (as would be the case if the LaTeX 2ε macros would have been used).
If `\toolboxIfNextToken` has found the required token, it does *not* gobble that token from the token stream. Of course, you can do this by yourself using the earlier described macro `\toolboxGobbleNext`. However, it is simpler to use

`\toolboxIfNextGobbling` `\toolboxIfNextGobbling{`⟨*token*⟩`}{`⟨*if*⟩`}{`⟨*else*⟩`}`

This command is analogous to `\toolboxIfNextToken` with the difference that in the case that the next token is ⟨*token*⟩, it is gobbled before ⟨*if*⟩ is executed.
Example:

```
\def\my{\toolboxIfNextGobbling*\toolboxTokenLoop\toolboxLoop}
```

This makes `\my*...` behave like `\toolboxTokenLoop...` and `\my...` (without `*`) behave like `\toolboxLoop...`.
The following macro is one which you may want to use in connection with LaTeX2e optional arguments:

`\toolboxIfEmpty` `\toolboxIfEmpty{`⟨*arg*⟩`}{`⟨*if*⟩`}{`⟨*else*⟩`}`

⟨*arg*⟩ is not expand; it is only used to decide whether ⟨*if*⟩ or ⟨*else*⟩ will be expanded.
For further tests there are more involved macros:

`\toolboxIfx` `\toolboxIfx{`⟨*arg*⟩`}\macro{`⟨*if*⟩`}{`⟨*else*⟩`}`

This tests via `\ifx` whether `\def\Macro{`⟨*arg*⟩`}` would give the definition of `\macro`.

`\toolboxIfX` `\toolboxIfX{`⟨*arg 1*⟩`}{`⟨*arg 2*⟩`}{`⟨*if*⟩`}{`⟨*else*⟩`}`

This tests whether ⟨*arg 1*⟩ and ⟨*arg 2*⟩ are the same token sequences.
If you want to avoid the `\else` and `\fi` commands to avoid certain side effects, you can use instead:

`\toolboxIfElse` `\toolboxIfElse{`⟨*ifcmd*⟩`}{`⟨*if*⟩`}{`⟨*else*⟩`}`

This is rather analogous to ⟨*ifcmd*⟩⟨*if*⟩`\else`⟨*else*⟩`\fi` but has everything in this line already eliminated from the tokenlist when ⟨*if*⟩ resp. ⟨*else*⟩ are expanded.

3.3 Loops over tokenlists and itemlists

`\toolboxLoop` `\toolboxLoop{`⟨*items*⟩`}{`⟨*action*⟩`}`

This calls iteratively ⟨*action*⟩`{#1}`, where `#1` runs over each item in ⟨*items*⟩. Here, an item is either a token or a group braced by `{...}`. In the latter case, the braces are lost. Spaces in ⟨*items*⟩ are ignored (unless they are braced). It is admissible that ⟨*action*⟩ is not a single macro but instead a sequence of tokens.
Examples follow below.
The counterintuitive order of arguments is explained by the fact that the typical usage is

`\expandafter\toolboxLoop\expandafter{\ExpandingMacro}{`⟨*action*⟩`}`

which for swapped order of arguments could hardly be written.
`\toolboxLoop` is not reentrant, i.e. {⟨*action*⟩} may not expand to something which contains a call to `\toolboxLoop`. To enable such calls anyway, the command

`\toolboxLoopName` `\toolboxLoopName{`⟨*name*⟩`}{`⟨*items*⟩`}{`⟨*action*⟩`}`

is provided which is analogous to \toolboxLoopName. This is also not reentrant, but in contrast to \toolboxLoop, calls with different ⟨*name*⟩ can be used independently of each other, i.e. in the {⟨*action*⟩} part of a \toolboxLoop (or \toolboxLoopName) can be a call to \toolboxLoopName with a *different* ⟨*name*⟩ argument. In particular, using a counter in ⟨*name*⟩, one could easily implement even recursive calls. In this connection, it should be noted that ⟨*name*⟩ is expanded via \csname ... \endcsname, and so you may use constructs like \the\namecounter there.

\toolboxTokenLoop `\toolboxTokenLoop{`⟨*tokens*⟩`}{`⟨*action*⟩`}`

This is similar to \toolboxLoop: The command ⟨*action*⟩\toolboxToken is executed iteratively where \toolboxToken runs over each token in ⟨*tokens*⟩. The important difference is that \toolboxToken is a token (instead of an item). In particular, \toolboxToken runs through every single token including spaces and braces.
The token \toolboxSpaceToken which is described later may be handy in connection with this command.
Example:

```
\toolboxTokenLoop{Some text}{\kern0.1em}
```

is the similar to \kern0.1em S\kern0.1em o\kern0.1em m..., i.e. you get wider spacing between the letters of Some text (I do not claim that this is typographically a good idea).
Note that you do not have to take special care about the space. With \toolboxLoop, you would have to mask the space e.g. with

```
\toolboxLoop{Some{ }text}{\kern0.1em}
```

or

```
\toolboxLoop{Some\toolboxSpace text}{\kern0.1em}
```

In contrast, \toolboxTokenLoop would behave differently here:

```
\toolboxTokenLoop{Some{ }text}{\kern0.1em}
```

would produce \kern0.1em S...\kern0.1em{\kern0.1em␣\kern0.1em}... because the braces are simply considered as tokens.
\toolboxTokenLoop is not reentrant. Analogously to \toolboxLoopName, independent versions can be generated by

\toolboxTokenName `\toolboxTokenName{`⟨*name*⟩`}{`⟨*tokens*⟩`}{`⟨*action*⟩`}`

3.4 Controlled expansion

There are some occasions when you want more control over the expansion. E.g. you might want to concatenate the contents of two macros to a further macro or you want to expand a macro by one level but no full expansion. Usually you can get this effects with `\expandafter`, but if you expand several concatenated tokens in this way you either have to write a lot of `\expandafter`s or you have to define subsidiary macros that help you to `\expandafter` certain parts of macros. The macros in this section allow you to do this in the most generic way that I could implement.

`\toolboxDef` `\toolboxDef\macrotodefine{`⟨*argumentlist*⟩`}`

This call is similar to

`\def\macrotodefine{`⟨*argumentlist*⟩`}`

with two important differences:
For `\toolboxDef`, ⟨*argumentlist*⟩ is expanded precisely by one level. ⟨*argumentlist*⟩ may not contain macros with parameters, and spaces in the highest level are ignored. If you want to force a space on a particular place, use the macro `\toolboxSpace` at this place (which is described later). Contrary to the usual `\def`, the symbol `#` is treated as a usual symbol.
Example of usage:

```
\toolboxDef\chain{\chain\toolboxSpace\after}
```

This modifies the macro `\chain` such that a space and the content of the macro `\after` is appended at the end.

`\toolboxAppend` `\toolboxAppend\macrotodefine{`⟨*argumentlist*⟩`}`

This is equivalent to

`\toolboxDef\macrotodefine{\macrotodefine`⟨*argumentlist*⟩`}`

The macro

`\toolboxSurround` `\toolboxSurround{`⟨*content before*⟩`}{`⟨*content after*⟩`}\macro`

redefines `\macro` such that ⟨*content before*⟩ is put at the beginning and ⟨*content after*⟩ after the definition of `\macro`. So this is equivalent to

`\def\macro{`⟨*content before*⟩⟨*old content of* `\macro`⟩⟨*content after*⟩`}`

It is required that `\macro` is a usual macro without any arguments. If you want to patch more complicated macros, use the `patch.doc` package instead.
The order of the arguments may appear strange, but it is convenient if ⟨*content before*⟩ or ⟨*content after*⟩ are macros which should be expanded with `\expandafter`.
There is some subsidiary macro used in the implementation of the above macros which might be useful also in some other situations:

\toolboxTokDef `\toolboxTokDef{⟨argumentlist⟩}\macrotodefine`

This call is similar to

`\def\macrotodefine{⟨argumentlist⟩}`

with the difference that the symbol `#` is stored as such. The order of the arguments has been swapped in order to simplify the application of `\expandafter` to ⟨*argumentlist*⟩.

3.5 Searching, splitting, and replacing

\toolboxSplitAt `\toolboxSplitAt{⟨argument⟩}{⟨search⟩}{\beforestring}{\afterstring}`

Here, `\beforestring` and `\afterstring` are arbitrary macro names, and ⟨*search*⟩ and ⟨*argument*⟩ are any sequences of tokens (which are in the following considered as 'strings').
This call scans ⟨*argument*⟩ for the first occurrence of ⟨*search*⟩. The macros `\beforestring` and `\afterstring` are defined correspondingly such that `\beforestring` expands to the part before the first occurrence, and `\afterstring` to the part following the first occurrence. If ⟨*search*⟩ does not occur in ⟨*argument*⟩, `\beforestring` is defined to ⟨*argument*⟩, and `\afterstring` is `\let \undefined`.
If `\beforestring` or `\afterstring` had already been defined before the call, the previous definition is tacitly overridden. It is explicitly allowed that `\beforestring` and `\afterstring` are the same names. In this case, the result has the meaning of `\afterstring`.
It is guaranteed that braces `{...}` are *not* lost in ⟨*argument*⟩. However, ⟨*search*⟩ may not contain any braces, and ⟨*argument*⟩ may contain only matching pairs of braces. Moreover, occurrences of ⟨*search*⟩ within a pair of braces in ⟨*argument*⟩ are not recognized.
(The order of the arguments has been chosen in order to simplify the use of `\expandafter`).
There are some restrictions for the strings in search. For example, the symbol `#` is not allowed.
In the above call, the arguments may not run over several paragraphs. If you want the latter, you have to use the alternative call

`\toolboxSplitAt*{⟨argument⟩}{⟨search⟩}{\beforestring}{\afterstring}`

Example of usage:

```
\def\examplemacro#1{\toolboxSplitAt{#1}{@}\testme\testme
  \ifx\testme\undefined
     ... (do this when #1 contains no '@' token)
  \fi}
```

If `\toolboxSplitAt` should be used several times with the same ⟨*search*⟩ string, it is much more efficient to use the following command:

\toolboxMakeSplit `\toolboxMakeSplit{⟨search⟩}{command}`

This call defines a new macro `\command` (the name is determined by the second argument of `\toolboxMakeSplit`) which can be called in the form

`\command{⟨argument⟩}{\beforestring}{\afterstring}`

and which has the analogous meaning as `\SplitAt` (the argument {⟨*search*⟩} is implicitly fixed and taken from the call of `\toolboxMakeSplit`). It is explicitly admissible that the above macro `\toolboxMakeSplit` is used with an already existing command name. In this case, the previous definition of `\command` is tacitly overridden.
The command created by `\toolboxMakeSplit` does not accept arguments which run over several paragraphs. If you want the latter, you have to create this command by the alternative call

`\toolboxMakeSplit*{⟨search⟩}{command}`

The command

`\toolboxFreeSplit{⟨command⟩}`

frees the memory used by a previous `\toolboxMakeSplit` (and lets `\command` again be undefined).
The command

\toolboxReplace `\toolboxReplace{⟨search⟩}{⟨replace⟩}\macro`

replaces in `\macro` all occurences of ⟨*search*⟩ by ⟨*replace*⟩. The same matches are found as in `\toolboxSplitAt`. If you need to search for the same text several times, it is faster to use the command

\toolboxReplaceSplit `\toolboxReplaceSplit{⟨replace⟩}\SplitCmd\macro`

where `\SplitCmd` is a command previously generated with `\toolboxMakeSplit*` according to your ⟨*search*⟩ string. (You could also use `\toolboxMakeSplit` to generate `\SplitCmd`, but then `\macro` should not contain any `\par`s).

3.6 Redefinition of macros

\toolboxMakeHarmless `\toolboxMakeHarmless{\macro}`

The above call redefines `\macro` such that it expands to an ASCII text containing the previous definition of `\macro` (i.e. the catcodes of `\macro` are changed).
The call

\toolboxDropBrace `\toolboxDropBrace{\macro}`

drops possible outer braces of `\macro`. More precisely, if `\macro` expands to {⟨*content*⟩}, then `\macro` is redefined to ⟨*content*⟩ (without braces). Otherwise, nothing happens.
The command

`\toolboxIf` `\toolboxIf`⟨*comparison*⟩{⟨*definition commands*⟩}\⟨*macro*⟩...

allows conditional definitions. Here, {⟨*definition command*⟩} is either `\def`, `{\long\def}`, `\let`, or some similar command like e.g. the LaTeX `\newcommand`. If the test `\ifx`⟨*comparison*⟩\⟨*macro*⟩ evaluates positive, then \⟨*macro*⟩ is defined correspondingly. Otherwise, \⟨*macro*⟩ is not changed.

Examples:

```
\toolboxIf\undefined\def\macro{....}
\toolboxIf\undefined\let\macro...
\toolboxIf\undefined{\long\def}\macro{....}
\toolboxIf\undefined\newcommand{\macro}{....}
```

are similar to `\def\macro{...}` resp. `\let\macro...` resp. `\long\def\macro{...}` resp. `\newcommand{\macro}` with the difference that `\macro` is not changed if it was already defined. In this sense, `\toolboxIf` is a more flexible variant of `\providecommand`.
The commands

`\toolboxNewiftrue` `\toolboxNewiftrue`{⟨*name*⟩}
`\toolboxNewiffalse` `\toolboxNewiffalse`{⟨*name*⟩}

test whether the command `\if`⟨*name*⟩ was already introduced with `\newif`; in this case nothing happens. Otherwise, `\if`⟨*name*⟩ is introduced similarly to `\newif\if`⟨*name*⟩ and set to `true` respectively `false`. In contrast to the corresponding command in TeX or LaTeX2.09, this macro is not `\outer`!

`\toolboxNewifTrue` `\toolboxNewifTrue`{⟨*name*⟩}
`\toolboxNewifFalse` `\toolboxNewifFalse`{⟨*name*⟩}

are similar to `\toolboxNewiftrue`{⟨*name*⟩} and `\toolboxNewiffalse`{⟨*name*⟩}, respectively, with the difference that `\if`⟨*name*⟩ is set unconditionally to `true` respectively `false`.

3.7 Concatenated macro names

`\toolboxLet` `\toolboxLet\variable`{⟨*macroname*⟩}

The above command is analogous to `\let\variable\macroname` with the difference that ⟨*macroname*⟩ can also contain other tokens like numbers (it is obtained via `\csname`). Some converse to this command is

`\toolboxWithNr`{⟨*number*⟩}`\command`{⟨*macro*⟩}

which translates into `\command\macronumber` (here, ⟨*macro*⟩ and ⟨*number*⟩ are just concatenated and evaluated via `\csname`). Examples:

```
\toolboxWithNr 1\let{name}\toolboxEmpty
```

This is the same as `\let\name1\toolboxEmpty` (but such that `\name1` is considered as a name, not as `\name 1`)

```
\toolboxWithNr {10}\def{name}{Foo}
```

This corresponds analogously to `\def\name10{Foo}`.

```
\toolboxLet\mymacro{name\the\mycount}
```

This is similar to `\let\mymacro\namexx` where `xx` is the content of the counter `\mycount`.

3.8 Various

The following macros have equivalents in most formats (like LaTeX 2_ε). However, we do not want to rely too much on these formats, so we provide our own definitions. The macro

`\toolboxEmpty` `\toolboxEmpty`

expands to nothing (usually, this is the same as `\empty`). Similarly, the macro

`\toolboxSpace` `\toolboxSpace`

expands to a space symbol (usually, this is the same as `\space`). The token

`\toolboxSpaceToken` `\toolboxSpaceToken`

is `\let` a space token (usually, this is the same as `\@sptoken`). This token is convenient in tests of tokens (because it is hard to get a space there which is not eaten by the TeX parser, although sometimes also constructions like

```
\expandafter\ifx\toolboxSpace\token
```

can be used). Also the macros

`\toolboxFirstOfTwo` `\toolboxFirstOfTwo`
`\toolboxSecondOfTwo` `\toolboxSecondOfTwo`

are provided which read two arguments and return only the first respectively second argument (usually, this is the same as `\@firstoftwo` respectively `\@secondoftwo`). Similarly,

`\toolboxGobbleArg` `\toolboxGobbleArg`{⟨*argument*⟩}

just reads its argument and expands to nothing.