Das Paket `etoolbox`

e-TeX-Werkzeuge für Klassen- und Paketautoren

Philipp Lehman[1] **Version 2.1**
plehman@gmx.net **3. Januar 2011**

Inhaltsverzeichnis

1 Einführung

1.1 Über `etoolbox`

Das Paket `etoolbox` ist eine Sammlung von Programmierwerkzeugen, die vorrangig für Autoren von LaTeX-Klassen und Paketen gedacht ist. Es liefert LaTeX-Schnittstellen für einige neue Grundfunktionen von e-TeX, sowie einige allgemeine Werkzeuge, die nicht e-TeX-spezifisch sind, aber zum Profil dieses Paketes passen.

1.2 Lizenz

[1] Übersetzung von Tim Enderling (`t.enderling@gmx.de`)
[2] `http://www.ctan.org/tex-archive/macros/latex/base/lppl.txt`

2 Befehle für Anwender

Die Befehle in diesem Abschnitt richten sich an Anwender von LaTeX, sowie an Klassen- und Paketautoren.

2.1 Schutz durch Neudefinition

`\newrobustcmd`{⟨*Befehl*⟩}[⟨*Parameter*⟩][⟨*Standardwert*⟩]{⟨*Ersetzungstext*⟩}
`\newrobustcmd*`{⟨*Befehl*⟩}[⟨*Parameter*⟩][⟨*Standardwert*⟩]{⟨*Ersetzungstext*⟩}

Dieser Befehl verhält sich wie `\newcommand`, macht den neu definierten ⟨*Befehl*⟩ aber zusätzlich robust (*robust command*[3]). Der Befehl unterscheidet sich vom LaTeX-Befehl `\DeclareRobustCommand` darin, dass er eine Fehlermeldung und nicht nur eine Warnung ausgibt, falls der ⟨*Befehl*⟩ bereits definiert wurde. Da er den systemnahen Mechanismus von e-TeX statt der höher angesiedelten LaTeX-Befehle verwendet, wird kein zusätzliches Makro benötigt um den ⟨*Befehl*⟩ robust zu machen.

`\renewrobustcmd`{⟨*Befehl*⟩}[⟨*Parameter*⟩][⟨*Standardwert*⟩]{⟨*Ersetzungstext*⟩}
`\renewrobustcmd*`{⟨*Befehl*⟩}[⟨*Parameter*⟩][⟨*Standardwert*⟩]{⟨*Ersetzungstext*⟩}

Dieser Befehl verhält sich wie `\renewcommand`, macht den umdefinierten ⟨*Befehl*⟩ aber zusätzlich robust.

`\providerobustcmd`{⟨*Befehl*⟩}[⟨*Parameter*⟩][⟨*Standardwert*⟩]{⟨*Ersetzungstext*⟩}
`\providerobustcmd*`{⟨*Befehl*⟩}[⟨*Parameter*⟩][⟨*Standardwert*⟩]{⟨*Ersetzungstext*⟩}

Dieser Befehl verhält sich wie `\providecommand`, macht den neu definierten ⟨*Befehl*⟩ aber zusätzlich robust. Allerdings stellt der Befehl nur dann eine robuste Variante des ⟨*Befehls*⟩ zur Verfügung, wenn der ⟨*Befehl*⟩ vorher undefiniert war. Bereits definierte Befehle können damit nicht robust gemacht werden.

2.2 Schutz existierender Befehle

`\robustify`{⟨*Befehl*⟩}

Definiert einen ⟨*Befehl*⟩, der mit `\newcommand` definiert wurde, als robusten Befehl neu. Seine Parameter, seine Präfixe und sein Ersetzungstext bleiben dabei erhalten. Falls der ⟨*Befehl*⟩ ursprünglich mit `\DeclareRobustCommand` definiert wurde, wird dies automatisch erkannt und der höher angesiedelter Schutzmechanismus von LaTeX durch den systemnahen Mechanismus von e-TeX ersetzt.

[3] http://www.matthiaspospiech.de/blog/2008/04/16/definition-von-makros-und-umgebungen/#toc-protect-fragile-und-robust oder
http://de.wikibooks.org/wiki/LaTeX-W%C3%B6rterbuch:_protect

2.3 Schutz beliebigen Codes

\protecting{⟨*Code*⟩}

Dieser Befehl wendet den LaTeX-eigenen Schutzmechanismus auf einen Block beliebigen ⟨*Codes*⟩ an, was normalerweise die Angabe von \protect vor jedem fragilen Befehl (*fragile command*) verlangt. Sein Verhalten hängt vom aktuellen Zustand von \protect ab. Wichtig ist, dass der ⟨*Code*⟩ in Klammern gesetzt werden muss, auch wenn es sich um einen einzelnen Befehl handelt.

2.4 Definition von Längen und Zählern

Die Werkzeuge in diesem Abschnitt können anstelle der Befehle \setcounter und \setlength verwendet werden und unterstützten arithmetische Ausdrücke.

\defcounter{⟨*Zähler*⟩}{⟨*ganzzahliger Ausdruck*⟩}

Weist einem zuvor mit \newcounter initialisierten LaTeX-⟨*Zähler*⟩ einen Wert zu. Dieser Befehl verhält sich wie \setcounter mit zwei wichtigen Unterschieden:
1.) Der zweite Parameter kann ein ⟨*ganzzahliger Ausdruck*⟩ sein, der mit \numexpr ausgewertet wird. Der ⟨*ganzzahlige Ausdruck*⟩ kann beliebiger, in diesem Kontext gültiger Code sein. Dem ⟨*Zähler*⟩ wird das Ergebnis der Berechnung als Wert zugewiesen. 2.) Im Gegensatz zu \setcounter ist die Zuweisung standardmäßig lokal. Es ist aber möglich, \defcounter ein \global voranzustellen. Die funktionale Entsprechung zu \setcounter ist also \global\defcounter.

\deflength{⟨*Längenvariable*⟩}{⟨*Abstandsausdruck*⟩}

Weist einer zuvor mit \newlength initialisierten Längenvariable einen Wert zu. Dieser Befehl verhält sich wie \setlength. Allerdings kann der zweite Parameter ein ⟨*Abstandsausdruck*⟩ (Ausdruck vom e-TeX-Typ *glue*) sein, der mit \glueexpr ausgewertet wird. Der ⟨*Abstandsausdruck*⟩ kann beliebiger, in diesem Kontext gültiger Code sein. Der ⟨*Längenvariable*⟩ wird das Ergebnis der Berechnung als Wert zugewiesen. Die Zuweisung ist standardmäßig lokal, es ist aber möglich \deflength ein \global voranzustellen und den resultierenden Befehl anstelle von \setlength zu verwenden.

2.5 Zusätzliche Hooks für Dokumente

LaTeX sieht zwei Hooks (*Einsprungpunkte*) vor, die den Ausführungszeitpunkt von Code auf den Anfang oder das Ende des Textteils verschieben. Jeder mit \AtBeginDocument markierte Code wird zu Beginn des Textteils ausgeführt, nachdem die aux-Hauptdatei zum ersten Mal gelesen wurde. Jeder mit \AtEndDocument markierte Code wird am Ende des Textteils ausgeführt, bevor die aux-Hauptdatei

zum zweiten Mal gelesen wird. Die hier vorgestellten Hooks verfolgen dieselbe Grundidee, verschieben den Ausführungszeitpunkt des ⟨*Codes*⟩ aber an geringfügig abweichende Stellen im Dokument. Der Parameter ⟨*Code*⟩ kann beliebiger TeX-Code sein. Parameterzeichen im ⟨*Code*⟩ sind zulässig und müssen nicht verdoppelt werden.

`\AfterPreamble{`⟨*Code*⟩`}`

Dieser Hook ist eine Variante von `\AtBeginDocument`, die sowohl in der Präambel als auch im Textteil verwendet werden kann. Wird er in der Präambel verwendet, verhält er sich genau wie `\AtBeginDocument`. Wird er im Textteil verwendet, führt er den ⟨*Code*⟩ sofort aus. `\AtBeginDocument` würde in diesem Fall einen Fehler melden. Dieser Hook ist nützlich um den Ausführungszeitpunkt von Code zu verschieben, der in die `aux`-Hauptdatei schreibt.

`\AtEndPreamble{`⟨*Code*⟩`}`

Dieser Hook unterscheidet sich von `\AtBeginDocument` dadurch, dass der ⟨*Code*⟩ am Ende der Präambel ausgeführt wird, bevor die `aux`-Hauptdatei (wie vom vorergehenden LaTeX-Lauf ausgegeben) gelesen wird und vor jedem mit `\AtBeginDokument` makierten Code. Zu beachten ist, dass der ⟨*Code*⟩ zu diesem Zeitpunkt nicht in die `aux`-Datei schreiben kann.

`\AfterEndPreamble{`⟨*Code*⟩`}`

Dieser Hook unterscheidet sich von `\AtBeginDocument` dadurch, dass der ⟨*Code*⟩ als Allerletztes im Befehl `\begin{document}` ausgeführt wird, nach jedem mit `\AtBeginDocument` markierten Code. Befehle, deren Gültigkeitsbereich mit Hilfe von `\@onlypreamble` auf die Präambel beschränkt wurde, stehen zur Ausführungszeit des ⟨*Codes*⟩ nicht mehr zur Verfügung.

`\AfterEndDocument{`⟨*Code*⟩`}`

Dieser Hook unterscheidet sich von `\AtEndDocument` dadurch, dass der ⟨*Code*⟩ als Allerletztes im Dokument ausgeführt wird, nachdem die `aux`-Hauptdatei (wie vom aktuellen LaTeX-Lauf ausgegeben) gelesen wurde und außerdem nach jedem mit `\AtEndDocument` markierten Code.

In bestimmter Hinsicht gehört mit `\AtBeginDocument` markierter Code weder zur Präambel noch zum Textteils, sondern liegt dazwischen, weil er mitten in der Initialisierungssequenz ausgeführt wird, die dem Textsatz vorausgeht. Manchmal ist es wünschenswert, Code am Ende der Präambel auszuführen, weil dann alle benötigten Pakete geladen sind. Mit `\AtBeginDocument` markierter Code wird allerdings zu spät ausgeführt, um in die `aux`-Datei einzugehen. Im Gegensatz dazu gehört mit `\AtEndPreamble` markierter Code zur Präambel; mit `\AfterEndPreamble`

markierter Code gehört zum Textteil und kann darstellbaren Text enthalten, der im Dokument zuallererst gesetzt wird. Zusammengefasst werden von LaTeX folgende Schritte ‚innerhalb' von `\begin{document}` ausgeführt:

- Ausführen sämtlichen mit `\AtEndPreamble` markierten Codes
- Initialisieren des Textteils (Seitenlayout, Standardschriften, usw.)
- Laden der `aux`-Hauptdatei, wie sie vom vorangegangen LaTeX-Lauf ausgegeben wurde
- Öffnen der `aux`-Hauptdatei zum Schreiben durch den aktuellen Lauf
- Initialisierung des Textteils fortsetzen
- Ausführen sämtlichen mit `\AtBeginDocument` markierten Codes
- Initialisierung des Textteils abschließen
- Deaktivieren aller mit `\@onlypreamble` markierten Befehle
- Ausführen sämtlichen mit `\AfterEndPreamble` markierten Codes

Innerhalb von `\end{document}` werden von LaTeX folgende Schritte ausgeführt:

- Ausführen sämtlichen mit `\AtEndDocument` markierten Codes
- abschließendes Ausführen von `\clearpage`
- Schließen der `aux`-Hauptdatei für den schreibenden Zugriff
- Laden der `aux`-Hauptdatei, wie sie vom aktuellen LaTeX-Lauf ausgegeben wurde
- abschließendes Testen und eventuell Ausgabe von Fehlermeldungen
- Ausführen sämtlichen mit `\AfterEndDocument` markierten Codes

Mit `\AtEndDocument` markierter Code gehört insofern noch zum Textteil, als dass er immer noch Text setzen und in die `aux`-Hauptdatei schreiben kann. Mit `\AfterEndDocument` markierter Code gehört nicht mehr zum Textteil. Dieser Hook ist nützlich, um die Daten in der `aux`-Datei am Ende eines LaTeX-Laufs auszuwerten.

2.6 Hooks für Umgebungen

Die Werkzeuge in diesem Abschnitt liefern Hooks für beliebige Umgebungen. Sie verändern dabei nicht die Definition der ⟨*Umgebung*⟩, sondern verschieben den Ausführungszeitpunkt des Codes in `\begin` und `\end`. Neudefinieren der ⟨*Umgebung*⟩ führt deshalb nicht zum Löschen bestehender Hooks. Der Parameter ⟨*Code*⟩ kann beliebiger TeX-Code sein. Parameterzeichen im ⟨*Code*⟩ sind zulässig und müssen nicht verdoppelt werden.

`\AtBeginEnvironment{`⟨*Umgebung*⟩`}{`⟨*Code*⟩`}`

Hängt beliebigen ⟨*Code*⟩ an einen Hook an, der vom `\begin`-Befehl am Anfang einer gegebenen ⟨*Umgebung*⟩ genau vor `\`⟨*Umgebung*⟩ ausgeführt wird, innerhalb der von `\begin` mit `{` geöffneten Gruppierung.

`\AtEndEnvironment{`⟨*Umgebung*⟩`}{`⟨*Code*⟩`}`

Hängt beliebigen ⟨*Code*⟩ an einen Hook an, der vom `\end`-Befehl am Ende einer gegebenen ⟨*Umgebung*⟩ genau vor `\end`⟨*environment*⟩ ausgeführt wird, innerhalb der von `\begin` geöffneten Gruppierung.

`\BeforeBeginEnvironment{`⟨*Umgebung*⟩`}{`⟨*Code*⟩`}`

Hängt beliebigen ⟨*Code*⟩ an einen Hook an, der vom `\begin`-Befehl noch vor Öffnen der Gruppierung ausgeführt wird.

`\AfterEndEnvironment{`⟨*Umgebung*⟩`}{`⟨*Code*⟩`}`

Hängt beliebigen ⟨*Code*⟩ an einen Hook an, der vom `\end`-Befehl nach Schließen der Gruppierung ausgeführt wird.

3 Befehle für Autoren

Die Werkzeuge in diesem Abschnitt richten sich an Klassen- und Paketautoren.

3.1 Definitionen

3.1.1 Definieren von Makros

Die Werkzeuge in diesem Abschnitt sind einfache, aber oft gebrauchte Kurzschreibweisen, die den Anwendungsbereich der Makros `\@namedef` und `\@nameuse` vergrößern.

`\csdef{`⟨*Name*⟩`}`⟨*Parameter*⟩`{`⟨*Ersetzungstext*⟩`}`

Wie die TeX-Primitive `\def`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist. Dieser Befehl ist robust und entspricht `\@namedef`.

`\csgdef{`⟨*Name*⟩`}`⟨*Parameter*⟩`{`⟨*Ersetzungstext*⟩`}`

Wie die TeX-Primitive `\gdef`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist. Dieser Befehl ist robust.

`\csedef`{⟨*Name*⟩}⟨*Parameter*⟩{⟨*Ersetzungstext*⟩}

Wie die TeX-Primitive `\edef`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist. Dieser Befehl ist robust.

`\csxdef`{⟨*Name*⟩}⟨*Parameter*⟩{⟨*Ersetzungstext*⟩}

Wie die TeX-Primitive `\xdef`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist. Dieser Befehl ist robust.

`\protected@csedef`{⟨*Name*⟩}⟨*Parameter*⟩{⟨*Ersetzungstext*⟩}

Wie `\csedef`, außer dass vorübergehend der LaTeX-eigene Schutzmechanismus aktiviert wird. Mit anderen Worten: Dieser Befehl verhält sich wie `\protected@edef` aus dem LaTeX-Kern, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist. Dieser Befehl ist robust.

`\protected@csxdef`{⟨*Name*⟩}⟨*Parameter*⟩{⟨*Ersetzungstext*⟩}

Wie `\csxdef`, außer dass vorübergehend der LaTeX-eigene Schutzmechanismus aktiviert wird. Mit anderen Worten: Dieser Befehl verhält sich wie `\protected@xdef` aus dem LaTeX-Kern, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist. Dieser Befehl ist robust.

`\cslet`{⟨*Name*⟩}{⟨*Befehl*⟩}

Wie die TeX-Primitive `\let`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist. Ist der übergebene ⟨*Befehl*⟩ undefiniert, so ist nach der Zuweisung auch die Kontrollsequenz undefiniert. Dieser Befehl ist robust und kann mit `\global` versehen werden.

`\letcs`{⟨*Befehl*⟩}{⟨*Name*⟩}

Wie die TeX-Primitive `\let`, außer dass der zweite Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist. Ist die Kontrollsequenz undefiniert, so ist nach der Zuweisung auch der übergebene ⟨*Befehl*⟩ undefiniert. Dieser Befehl ist robust und kann mit `\global` versehen werden.

`\csletcs`{⟨*Name*⟩}{⟨*Name*⟩}

Wie die TeX-Primitive `\let`, außer dass beide Parameter ⟨*Namen*⟩ von Kontrollsequenzen sind. Ist die zweite Kontrollsequenz undefiniert, so ist nach der Zuweisung auch die erste Kontrollsequenz undefiniert. Dieser Befehl ist robust und kann mit `\global` versehen werden.

`\csuse{`⟨*Name*⟩`}`

Erhält den ⟨*Namen*⟩ einer Kontrollsequenz als Parameter und erzeugt den zugehörigen Befehlstoken. Dieser Befehl unterscheidet sich vom Makro `\@nameuse` aus dem LaTeX-Kern darin, dass er eine leere Zeichenfolge liefert, falls die Kontrollsequenz undefiniert ist.

`\undef`⟨*Befehl*⟩

Löscht einen ⟨*Befehl*⟩, so dass die Tests `\ifdefined` und `\ifcsname` ihn als undefiniert behandeln. Dieser Befehl ist robust und kann mit `\global` versehen werden.

`\csundef{`⟨*Name*⟩`}`

Wie `\undef`, außer dass der Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist. Dieser Befehl ist robust und kann mit `\global` versehen werden.

`\csshow{`⟨*Name*⟩`}`

Wie die TeX-Primitive `\show`, außer dass der Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist. Ist die Kontrollsequenz undefiniert, weist der Befehl der Kontrollsequenz nicht automatisch die Bedeutung von `\relax` zu. Dieser Befehl ist robust.

3.1.2 Definieren arithmetischer Makros

Die Werkzeuge in diesem Abschnitt erlauben anstelle von Längenvariablen und Zählern Makros für Berechnungen zu verwenden.

`\numdef`⟨*Befehl*⟩`{`⟨*ganzzahliger Ausdruck*⟩`}`

Wie `\edef`, außer dass der ⟨*ganzzahlige Ausdruck*⟩ mit `\numexpr` ausgewertet wird. Der ⟨*ganzzahlige Ausdruck*⟩ kann beliebiger, in diesem Kontext gültiger Code sein. Der Ersetzungstext des ⟨*Befehls*⟩ ist das Ergebnis der Berechnung. Ist der ⟨*Befehl*⟩ undefiniert, wird er auf `0` initialisiert, bevor der ⟨*ganzzahlige Ausdruck*⟩ ausgewertet wird.

`\numgdef`⟨*Befehl*⟩`{`⟨*ganzzahliger Ausdruck*⟩`}`

Wie `\numdef`, außer dass die Zuweisung global ist.

`\csnumdef{`⟨*Name*⟩`}{`⟨*ganzzahliger Ausdruck*⟩`}`

Wie `\numdef`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

`\csnumgdef{`⟨*Name*⟩`}{`⟨*ganzzahliger Ausdruck*⟩`}`

Wie `\numgdef`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\dimdef⟨*Befehl*⟩{⟨*Längenausdruck*⟩}

Wie \edef, außer dass der ⟨*Längenausdruck*⟩ (Ausdruck vom e-TeX-Typ *dimen*) mit \dimexpr ausgewertet wird. Der ⟨*Längenausdruck*⟩ kann beliebiger, in diesem Kontext gültiger Code sein. Der Ersetzungstext des ⟨*Befehls*⟩ ist das Ergebnis der Berechnung. Ist der ⟨*Befehl*⟩ undefiniert, wird er auf 0 initialisiert, bevor der ⟨*Längenausdruck*⟩ ausgewertet wird.

\dimgdef⟨*Befehl*⟩{⟨*Längenausdruck*⟩}

Wie \dimdef, außer dass die Zuweisung global ist.

\csdimdef{⟨*Name*⟩}{⟨*Längenausdruck*⟩}

Wie \dimdef, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\csdimgdef{⟨*Name*⟩}{⟨*Längenausdruck*⟩}

Wie \dimgdef, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\gluedef⟨*Befehl*⟩{⟨*Abstandsausdruck*⟩}

Wie \edef, außer dass der ⟨*Abstandsausdruck*⟩ (Ausdruck vom e-TeX-Typ *glue*) mit \glueexpr ausgewertet wird. Der ⟨*Abstandsausdruck*⟩ kann beliebiger, in diesem Kontext gültiger Code sein. Der Ersetzungstext des ⟨*Befehls*⟩ ist das Ergebnis der Berechnung. Ist der ⟨*Befehl*⟩ undefiniert, wird er auf 0pt plus 0pt minus 0pt initialisiert, bevor der ⟨*Abstandsausdruck*⟩ ausgewertet wird.

\gluegdef⟨*Befehl*⟩{⟨*Abstandsausdruck*⟩}

Wie \gluedef, außer dass die Zuweisung global ist.

\csgluedef{⟨*Name*⟩}{⟨*Abstandsausdruck*⟩}

Wie \gluedef, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\csgluegdef{⟨*Name*⟩}{⟨*Abstandsausdruck*⟩}

Wie \gluegdef, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\mudef⟨*Befehl*⟩{⟨*Abstandsausdruck in mu*⟩}

Wie \edef, außer dass der ⟨*Abstandsausdruck in mu*⟩ (Ausdruck vom e-TeX-Typ *muglue*; Abstand in *math units*) mit \muexpr ausgewertet wird. Der übergebene ⟨*Abstandsausdruck*⟩ kann beliebiger, in diesem Kontext gültiger Code sein. Der Ersetzungstext des ⟨*Befehls*⟩ ist das Ergebnis der Berechnung. Ist der ⟨*Befehl*⟩ undefiniert, wird er auf 0mu initialisiert, bevor der ⟨*Abstandsausdruck*⟩ ausgewertet wird.

`\mugdef`⟨*Befehl*⟩`{`⟨*Abstandsausdruck in mu*⟩`}`

Wie `\mudef`, außer dass die Zuweisung global ist.

`\csmudef{`⟨*Name*⟩`}{`⟨*Abstandsausdruck in mu*⟩`}`

Wie `\mudef`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

`\csmugdef{`⟨*Name*⟩`}{`⟨*Abstandsausdruck in mu*⟩`}`

Wie `\mugdef`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

3.2 Expansionssteuerung

Die Werkzeuge in diesem Abschnitt sind nützlich, um die Expansion von Definitionen mit `\edef` oder ähnlichen Befehlen zu steuern.

`\expandonce`⟨*Befehl*⟩

Dieser Befehl expandiert einen ⟨*Befehl*⟩ einmalig und verhindert jede weitere Expansion des Ersetzungstextes. Dieser Befehl ist selbst expandierbar.

`\csexpandonce{`⟨*Name*⟩`}`

Wie `\expandonce`, außer dass der Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

3.3 Hook-Verwaltung

Die Werkzeuge in diesem Abschnitt dienen der Verwaltung von Hooks. Ein ⟨*Hook*⟩ ist in diesem Zusammenhang ein einfaches Makro ohne Parameter und Präfixe, dessen Zweck es ist, Code zur späteren Ausführung zusammenzufassen. Diese Werkzeuge können auch verwendet werden, um einfache Makros durch Anhängen von Code an ihren Ersetzungstext anzupassen. Für komplexe Anpassungen, siehe Abschnitt 3.4. Alle Befehle in diesem Abschnitt initialisieren den ⟨*Hook*⟩, wenn er undefiniert ist.

3.3.1 Code an einen Hook anhängen

Die Werkzeuge in diesem Abschnitt hängen beliebigen Code an einen Hook an.

`\appto`⟨*Hook*⟩`{`⟨*Code*⟩`}`

Dieser Befehl hängt beliebigen ⟨*Code*⟩ an einen ⟨*Hook*⟩ an. Falls der ⟨*Code*⟩ Parameterzeichen enthält, müssen sie nicht verdoppelt werden. Dieser Befehl ist robust.

\gappto⟨*Hook*⟩{⟨*Code*⟩}

Wie \appto, außer dass die Zuweisung global ist. Dieser Befehl kann anstelle des Makros \g@addto@macro aus dem LaTeX-Kern verwendet werden.

\eappto⟨*Hook*⟩{⟨*Code*⟩}

Dieser Befehl hängt beliebigen ⟨*Code*⟩ an einen ⟨*Hook*⟩ an. Der ⟨*Code*⟩ wird bei der Definition expandiert. Nur der neue ⟨*Code*⟩ wird expandiert, nicht der bereits bestehende Ersetzungtext des ⟨*Hooks*⟩. Dieser Befehl ist robust.

\xappto⟨*Hook*⟩{⟨*Code*⟩}

Wie \eappto, außer dass die Zuweisung global ist.

\protected@eappto⟨*Hook*⟩{⟨*Code*⟩}

Wie \eappto, außer dass vorübergehend der LaTeX-eigene Schutzmechanismus aktiviert wird.

\protected@xappto⟨*Hook*⟩{⟨*Code*⟩}

Wie \xappto, außer dass vorübergehend der LaTeX-eigene Schutzmechanismus aktiviert wird.

\csappto{⟨*Name*⟩}{⟨*Code*⟩}

Wie \appto, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\csgappto{⟨*Name*⟩}{⟨*Code*⟩}

Wie \gappto, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\cseappto{⟨*Name*⟩}{⟨*Code*⟩}

Wie \eappto, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\csxappto{⟨*Name*⟩}{⟨*Code*⟩}

Wie \xappto, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\protected@cseappto{⟨*Name*⟩}{⟨*Code*⟩}

Wie \protected@eappto, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

`\protected@csxappto{`⟨*Name*⟩`}{`⟨*Code*⟩`}`

Wie `\protected@xappto`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

3.3.2 Einem Hook Code voranstellen

Die Werkzeuge in diesem Abschnitt stellen einem Hook beliebigen Code voran, d. h. der Code wird am Anfang des Hooks eingefügt, statt ans Ende angehängt zu werden.

`\preto`⟨*Hook*⟩`{`⟨*Code*⟩`}`

Wie `\appto`, außer dass der ⟨*Code*⟩ vorangestellt wird.

`\gpreto`⟨*Hook*⟩`{`⟨*Code*⟩`}`

Wie `\preto`, außer dass die Zuweisung global ist.

`\epreto`⟨*Hook*⟩`{`⟨*Code*⟩`}`

Wie `\eappto`, außer dass der ⟨*Code*⟩ vorangestellt wird.

`\xpreto`⟨*Hook*⟩`{`⟨*Code*⟩`}`

Wie `\epreto`, außer dass die Zuweisung global ist.

`\protected@epreto`⟨*Hook*⟩`{`⟨*Code*⟩`}`

Wie `\epreto`, außer dass vorübergehend der LaTeX-eigene Schutzmechismus aktiviert wird.

`\protected@xpreto`⟨*Hook*⟩`{`⟨*Code*⟩`}`

Wie `\xpreto`, außer dass vorübergehend der LaTeX-eigene Schutzmechismus aktiviert wird.

`\cspreto{`⟨*Name*⟩`}{`⟨*Code*⟩`}`

Wie `\preto`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

`\csgpreto{`⟨*Name*⟩`}{`⟨*Code*⟩`}`

Wie `\gpreto`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

`\csepreto{`⟨*Name*⟩`}{`⟨*Code*⟩`}`

Wie `\epreto`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\csxpreto{⟨*Name*⟩}{⟨*Code*⟩}

Wie \xpreto, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\protected@csepreto{⟨*Name*⟩}{⟨*Code*⟩}

Wie \protected@epreto, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\protected@csxpreto{⟨*Name*⟩}{⟨*Code*⟩}

Wie \protected@xpreto, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

3.4 Anpassung existierender Befehle

Die Werkzeuge in diesem Abschnitt sind nützlich, um bestehenden Code anzupassen oder ihn mit Hooks zu versehen. Bei allen hier vorgestellten Befehlen bleiben die Parameter und Präfixe des angepassten ⟨*Befehls*⟩ erhalten. Befehle, die mit \outer markiert wurden, können nicht angepasst werden. Die hier vorgestellten Befehle melden nicht automatisch einen Fehler, wenn die Anpassung fehlschlägt. Statt dessen erwarten sie einen Parameter ⟨*Fehlerroutine*⟩, der eine angebrachte Ausweichlösung oder Fehlermeldung liefert. Die Verwendung von \tracingpatches in der Präambel führt zur Aufnahme von Debug-Informationen in die Log-Datei.

\patchcmd[⟨*Präfix*⟩]{⟨*Befehl*⟩}{⟨*Suchmuster*⟩}{⟨*Ersatztext*⟩}{⟨*Erfolgsroutine*⟩}{⟨*Fehlerroutine*⟩}

Dieser Befehl ermittelt den Ersetzungstext des ⟨*Befehls*⟩, ersetzt das ⟨*Suchmuster*⟩ durch den ⟨*Ersatztext*⟩, und fügt den ⟨*Befehl*⟩ wieder zusammen. Die Mustererkennung ignoriert Kategoriecodes und liefert das erste Vorkommen des ⟨*Suchmusters*⟩ im Ersetzungstext des anzupassenden ⟨*Befehls*⟩. Dabei wird für den Anpassungsprozess der Ersetzungstext des ⟨*Befehls*⟩ aus seinen Token zusammengesetzt und nach der Anpassung anhand der aktuellen Regeln für Kategoriecodes wieder in Token zerlegt. Der Kategoriecode des @-Zeichens wird zwischenzeitlich auf 11 gesetzt. Sollte der Ersetzungstext des ⟨*Befehls*⟩ Token mit benutzerdefinierten Kategoriecodes enthalten, müssen die entsprechenden Kategoriecodes vor der Anpassung eingestellt werden. Falls der ⟨*Ersatztext*⟩ sich auf die Parameter des anzupassenden ⟨*Befehls*⟩ bezieht, müssen Parameterzeichen nicht verdoppelt werden. Wird das optionale ⟨*Präfix*⟩ angegeben, so ersetzt es alle bestehenden Präfixe des ⟨*Befehls*⟩. Ein leeres ⟨*Präfix*⟩ entfernt alle bestehenden Präfixe des ⟨*Befehls*⟩. Die Zuweisung ist lokal. Der Befehl führt vor der Anpassung automatisch einen zu \ifpatchable äquivalenten Test durch. Ist der Test erfolgreich, wird der ⟨*Befehl*⟩ angepasst und die ⟨*Erfolgsroutine*⟩ ausgeführt. Schlägt der Test fehl, wird die ⟨*Fehlerroutine*⟩ durchgeführt, ohne den ⟨*Befehl*⟩ zu verändern. Dieser Befehl ist robust.

`\ifpatchable`{⟨*Befehl*⟩}{⟨*Suchmuster*⟩}{⟨*Erfolgsroutine*⟩}{⟨*Fehlerroutine*⟩}

Dieser Befehl führt die ⟨*Erfolgsroutine*⟩ aus, falls der ⟨*Befehl*⟩ mit `\patchcmd` angepasst werden kann und das ⟨*Suchmuster*⟩ in seinem Ersetzungstext vorkommt. Sonst wird die ⟨*Fehlerroutine*⟩ ausgeführt. Dieser Befehl ist robust.

`\ifpatchable*`{⟨*Befehl*⟩}{⟨*Erfolgsroutine*⟩}{⟨*Fehlerroutine*⟩}

Wie `\ifpatchable`, außer dass die Variante mit Stern kein Suchmuster als Parameter benötigt. Dieser Version ermittelt, ob der ⟨*Befehl*⟩ mit `\apptocmd` und `\pretocmd` angepasst werden kann.

`\apptocmd`{⟨*Befehl*⟩}{⟨*Code*⟩}{⟨*Erfolgsroutine*⟩}{⟨*Fehlerroutine*⟩}

Dieser Befehl hängt den angegebenen ⟨*Code*⟩ an den Ersetzungstext eines ⟨*Befehls*⟩ an. Wenn der ⟨*Befehl*⟩ keine Parameter erwartet, verhält sich `\apptocmd` wie `\appto` im Abschnitt 3.3.1. Im Gegensatz zu `\appto` kann `\apptocmd` auch für Befehle mit Parametern verwendet werden. In diesem Fall wird der Ersetzungstext des ⟨*Befehls*⟩ aus seinen Token zusammengesetzt, angepasst und anhand der aktuellen Regeln für Kategoriecodes wieder in Token zerlegt. Der Kategoriecode des @-Zeichens wird zwischenzeitlich auf 11 gesetzt. Der anzuhängende ⟨*Code*⟩ kann sich auf die Parameter des ⟨*Befehls*⟩ beziehen. Die Zuweisung ist lokal. Ist die Anpassung erfolgreich, wird die ⟨*Erfolgsroutine*⟩ ausgeführt. Schlägt die Anpassung fehl, wird die ⟨*Fehlerroutine*⟩ ausgeführt, on den ⟨*Befehl*⟩ zu verändern. Dieser Befehl ist robust.

`\pretocmd`{⟨*Befehl*⟩}{⟨*Code*⟩}{⟨*Erfolgsroutine*⟩}{⟨*Fehlerroutine*⟩}

Dieser Befehl verhält sich wie `\apptocmd`, außer dass der ⟨*Code*⟩ dem Ersetzungstext des ⟨*Befehls*⟩ vorangestellt wird. Wenn der ⟨*Befehl*⟩ keine Parameter erwartet, verhält er sich wie `\preto` im Abschnitt 3.3.1. Im Gegensatz zu `\preto` kann `\pretocmd` auch für Befehle mit Parametern verwendet werden. In diesem Fall wird der Ersetzungstext des ⟨*Befehls*⟩ aus seinen Token zusammengesetzt, angepasst und anhand der aktuellen Regeln für Kategoriecodes wieder in Token zerlegt. Der Kategoriecode des @-Zeichens wird zwischenzeitlich auf 11 gesetzt. Der voranzustellende ⟨*Code*⟩ kann sich auf die Parameter des ⟨*Befehls*⟩ beziehen. Die Zuweisung ist lokal. Ist die Anpassung erfolgreich, wird die ⟨*Erfolgroutine*⟩ ausgeführt. Schlägt die Anpassung fehl, wird die ⟨*Fehlerroutine*⟩ ausgeführt, ohne den ⟨*Befehl*⟩ zu verändern. Dieser Befehl ist robust.

`\tracingpatches` Aktiviert die Ablaufverfolgung für alle Anpassungsbefehle inklusive `\ifpatchable`. Die Debug-Informationen werden in die Log-Datei geschrieben. Dies ist nützlich, wenn der Grund für das Fehlschlagen der Anpassung oder die Ausführung der Fehlerroutine von `\ifpatchable` unklar ist. Dieser Befehl muss in der Präambel stehen.

3.5 Flags

Dieses Paket beinhaltet zwei Schnittstellen für Flags (*boolesche Variablen*), die völlig unabhängig voneinander arbeiten. Die Werkzeuge im Abschnitt 3.5.1 bilden eine LaTeX-Schnittstelle zu `\newif`, während die Werkzeuge im Abschnitt 3.5.2 auf einem anderen Mechanismus basieren.

3.5.1 TeX-Flags

Da die Werkzeuge in diesem Abschnitt intern auf `\newif` zurückgreifen, können sie zum Abfragen und Verändern von Flags verwendet werden, die zuvor mit `\newif` definiert wurden. Des Weiteren sind sie kompatibel mit den booleschen Tests des Paketes `ifthen` und können als LaTeX-Schnittstelle zum Abfragen von TeX-Primitiven wie `\ifmmode` dienen. Der Ansatz von `\newif` erfordert ingesamt drei Makros pro Flag.

`\newbool{`⟨*Name*⟩`}`

Definiert ein neues boolesches Flag mit dem angegebenen ⟨*Namen*⟩. Ist das Flag bereits definiert, meldet der Befehl einen Fehler. Der Ausgangszustand von neu definierten Flags ist `false`. Dieser Befehl ist robust.

`\providebool{`⟨*Name*⟩`}`

Definiert ein neues boolesches Flag mit dem angegebenen ⟨*Namen*⟩, falls es nicht bereits definiert wurde. Dieser Befehl ist robust.

`\booltrue{`⟨*Name*⟩`}`

Setzt das Flag mit dem angegebenen ⟨*Namen*⟩ auf `true`. Dieser Befehl ist robust und kann mit `\global` versehen werden. Er meldet einen Fehler, falls das Flag undefiniert ist.

`\boolfalse{`⟨*Name*⟩`}`

Setzt das Flag mit dem angegebenen ⟨*Namen*⟩ auf `false`. Dieser Befehl ist robust und kann mit `\global` versehen werden. Er meldet einen Fehler, falls das Flag undefiniert ist.

`\setbool{`⟨*Name*⟩`}{`⟨*Wert*⟩`}`

Setzt das Flag mit dem angegebenen ⟨*Namen*⟩ auf einen ⟨*Wert*⟩, der entweder `true` oder `false` sein kann. Dieser Befehl ist robust und kann mit `\global` versehen werden. Er meldet einen Fehler, falls das Flag undefiniert ist.

`\ifbool{`⟨*Name*⟩`}{`⟨*Wahrausgabe*⟩`}{`⟨*Falschausgabe*⟩`}`

Liefert die ⟨*Wahrausgabe*⟩, wenn das Flags mit dem angegebenen ⟨*Namen*⟩ den Zustand `true` hat, sonst die ⟨*Falschausgabe*⟩. Ist das Flag undefiniert, meldet der Befehl einen Fehler. Dieser Befehl kann jeden booleschen Test durchführen, der auf Basis der einfachen TeX-Syntax funktioniert, d. h. jeder Test, der normalerweise wie folgt geschrieben wird:

```
\iftest Wahrausgabe\else Falschausgabe\fi
```

Das schließt alle Flags ein, die mit `\newif` definiert wurden, sowie alle TeX-Primitiven wie `\ifmmode`. Das `\if`-Präfix der Primitiven oder des Flags wird dabei im ⟨*Namen*⟩ weggelassen. Beispiel:

```
\ifmytest Wahrausgabe\else Falschausgabe\fi
\ifmmode Wahrausgabe\else Falschausgabe\fi
```

wird zu

```
\ifbool{mytest}{Wahrausgabe}{Falschausgabe}
\ifbool{mmode}{Wahrausgabe}{Falschausgabe}
```

`\notbool{`⟨*Name*⟩`}{`⟨*Falschausgabe*⟩`}{`⟨*Wahrausgabe*⟩`}`

Wie `\ifbool`, kehrt aber den Test um.

3.5.2 LaTeX-Flags

Im Gegensatz zu den Flags im Abschnitt 3.5.1 erfordern die Werkzeuge in diesem Abschnitt nur ein Makro pro Flag. Außerdem haben sie einen eigenen Namensraum, um Namenskonflikte mit gewöhnlichen Makros zu vermeiden.

`\newtoggle{`⟨*Name*⟩`}`

Definiert ein neues boolesches Flag mit dem angegebenen ⟨*Namen*⟩. Ist das Flag bereits definiert, meldet der Befehl einen Fehler. Der Ausgangszustand von neu definierten Flags ist `false`. Dieser Befehl ist robust.

`\providetoggle{`⟨*Name*⟩`}`

Definiert ein neues boolesches Flag mit dem angegebenen ⟨*Namen*⟩, falls es nicht bereits definiert wurde. Dieser Befehl ist robust.

`\toggletrue{`⟨*Name*⟩`}`

Setzt das Flag mit dem angegebenen ⟨*Namen*⟩ auf `true`. Dieser Befehl ist robust und kann mit `\global` versehen werden. Er meldet einen Fehler, falls das Flag undefiniert ist.

`\togglefalse{`⟨*Name*⟩`}`

Setzt das Flag mit dem angegebenen ⟨*Namen*⟩ auf `false`. Dieser Befehl ist robust und kann mit `\global` versehen werden. Er meldet einen Fehler, falls das Flag undefiniert ist.

`\settoggle{`⟨*Name*⟩`}{`⟨*Wert*⟩`}`

Setzt das Flag mit dem angegebenen ⟨*Namen*⟩ auf einen `Wert`, der entweder `true` oder `false` sein kann. Dieser Befehl ist robust und kann mit `\global` versehen werden. Er meldet einen Fehler, falls das Flag undefiniert ist.

`\iftoggle{`⟨*Name*⟩`}{`⟨*Wahrausgabe*⟩`}{`⟨*Falschausgabe*⟩`}`

Liefert die ⟨*Wahrausgabe*⟩, wenn das Flag mit dem angegebenen ⟨*Namen*⟩ den Zustand `true` hat, sonst die ⟨*Falschausgabe*⟩. Ist das Flag undefiniert, meldet der Befehl einen Fehler.

`\nottoggle{`⟨*Name*⟩`}{`⟨*Falschausgabe*⟩`}{`⟨*Wahrausgabe*⟩`}`

Wie `\iftoggle`, kehrt aber den Test um.

3.6 Tests

3.6.1 Makrotests

`\ifdef{`⟨*Kontrollsequenz*⟩`}{`⟨*Wahrausgabe*⟩`}{`⟨*Falschausgabe*⟩`}`

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Kontrollsequenz*⟩ definiert ist, im anderen Fall die ⟨*Falschausgabe*⟩. Die Kontrollsequenz wird auch dann als definiert angesehen, wenn ihre Bedeutung `\relax` lautet. Dieser Befehl ist eine LaTeX-Schnittstelle für die e-TeX-Primitive `\ifdefined`.

`\ifcsdef{`⟨*Name*⟩`}{`⟨*Wahrausgabe*⟩`}{`⟨*Falschausgabe*⟩`}`

Wie `\ifdef`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist. Dieser Befehl ist eine LaTeX-Schnittstelle für die e-TeX-Primitive `\ifcsname`.

\ifundef{⟨*Kontrollsequenz*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Kontrollsequenz*⟩ undefiniert ist, sonst die ⟨*Falschausgabe*⟩. Abgesehen vom Umkehren der Logik unterscheidet sich dieser Befehl außerdem dadurch von `\ifdef`, dass die ⟨*Kontrollsequenz*⟩ als undefiniert angesehen wird, falls ihre Bedeutung `\relax` lautet.

\ifcsundef{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifundef`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist. Dieser Befehl kann anstelle des Tests `\@ifundefined` aus dem LaTeX-Kern verwendet werden.

\ifdefmacro{⟨*Kontrollsequenz*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Kontrollsequenz*⟩ definiert ist und ein Makro darstellt, sonst die ⟨*Falschausgabe*⟩.

\ifcsmacro{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifdefmacro`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\ifdefparam{⟨*Kontrollsequenz*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Kontrollsequenz*⟩ definiert ist und ein Makro mit einem oder mehr Parametern darstellt, sonst die ⟨*Falschausgabe*⟩.

\ifcsparam{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifdefparam`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\ifdefprefix{⟨*Kontrollsequenz*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Kontrollsequenz*⟩ definiert ist und ein Makro darstellt, das mit `\long` und/oder `\protected` versehen wurde, sonst die ⟨*Falschausgabe*⟩. Auf das Präfix `\outer` kann nicht getestet werden.

\ifcsprefix{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifdefprefix`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\ifdefprotected{⟨*Kontrollsequenz*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Kontrollsequenz*⟩ definiert ist und ein Makro darstellt, das mit `\protected` versehen wurde, sonst die ⟨*Falschausgabe*⟩.

`\ifcsprotected`{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifdefprotected`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

`\ifdefltxprotect`{⟨*Kontrollsequenz*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Kontrollsequenz*⟩ definiert ist und einen Hüllenbefehl im Rahmen des LaTeX-eigenen Schutzmechanismus darstellt, sonst die ⟨*Falschausgabe*⟩. Damit lassen sich Befehle erkennen, die mit dem Schutzbefehl `\DeclareRobustCommand` oder einem ähnlichen Mechanismus definiert wurden.

`\ifcsltxprotect`{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifdefltxprotect`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

`\ifdefempty`{⟨*Kontrollsequenz*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Kontrollsequenz*⟩ definiert ist und ein Makro ohne Parameter darstellt, dessen Ersetzungstext leer ist, sonst die ⟨*Falschausgabe*⟩. Im Gegensatz zu `\ifx` ignoriert dieser Test die Präfixe des Makros.

`\ifcsempty`{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifdefempty`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

`\ifdefvoid`{⟨*Kontrollsequenz*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Kontrollsequenz*⟩ undefiniert ist, ihre Bedeutung `\relax` lautet oder sie ein Makro ohne Parameter darstellt, dessen Ersetzungstext leer ist, sonst die ⟨*Falschausgabe*⟩.

`\ifcsvoid`{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifdefvoid`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

`\ifdefequal`{⟨*Kontrollsequenz*⟩}{⟨*Kontrollsequenz*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Vergleicht zwei Kontrollsequenzen und liefert die ⟨*Wahrausgabe*⟩, wenn sie im Sinne von `\ifx` gleich sind, sonst die ⟨*Falschausgabe*⟩. Im Gegensatz zu `\ifx`, liefert dieser Test auch dann die ⟨*Falschausgabe*⟩, wenn beide Kontrollsequenzen undefiniert sind oder ihre Bedeutung `\relax` lautet.

`\ifcsequal`{⟨*Name*⟩}{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifdefequal`, außer dass die ersten beiden Parameter ⟨*Namen*⟩ von Kontrollsequenzen sind.

`\ifdefstring`{⟨*Befehl*⟩}{⟨*Zeichenfolge*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Vergleicht den Ersetzungtext eines ⟨*Befehls*⟩ mit einer ⟨*Zeichenfolge*⟩ und liefert die ⟨*Wahrausgabe*⟩, wenn sie gleich sind, sonst die ⟨*Falschausgabe*⟩. Weder der ⟨*Befehl*⟩ noch die ⟨*Zeichenfolge*⟩ werden für den Test expandiert, Kategoriecodes werden ignoriert. Kontrollsequenzen in der ⟨*Zeichenfolge*⟩ werden aus ihren Token zusammengesetzt und als Zeichenfolgen behandelt. Dieser Befehl ist robust. Dieser Test berücksichtigt nur den Ersetzungstext des ⟨*Befehls*⟩. Zum Beispiel liefert der Test

```
\long\edef\MeinMakro#1#2{\string&}
\ifdefstring{\MeinMakro}{&}{Wahrausgabe}{Falschausgabe}
```

die ⟨*Wahrausgabe*⟩. Das Präfix und die Parameter von `\MeinMakro` werden ignoriert, ebenso die Kategoriecodes im Ersetzungstext.

`\ifcsstring`{⟨*Name*⟩}{⟨*Zeichenfolge*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifdefstring`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

`\ifdefstrequal`{⟨*Befehl*⟩}{⟨*Befehl*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Führt einen Vergleich der Ersetzungstexte zweier ⟨*Befehle*⟩ durch und ignoriert dabei die Kategoriecodes. Dieser Befehl funktioniert wie `\ifdefstring`, außer dass beide zu vergleichenden Parameter Makros sind. Dieser Befehl ist robust.

`\ifcsstrequal`{⟨*Name*⟩}{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifdefstrequal`, außer dass die ersten beiden Parameter ⟨*Namen*⟩ von Kontrollsequenzen sind.

3.6.2 Tests für Zähler- und Längenvariablen

`\ifdefcounter`{⟨*Kontrollsequenz*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Kontrollsequenz*⟩ ein TeX-Register vom Typ `\count` ist, das mit `\newcount` allokiert wurde, sonst die ⟨*Falschausgabe*⟩.

`\ifcscounter`{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifdefcounter`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\ifltxcounter{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn der erste Parameter der ⟨*Name*⟩ eines LaTeX-Zählers ist, der mit `\newcounter` allokiert wurde, sonst die ⟨*Falschausgabe*⟩.

\ifdeflength{⟨*Kontrollsequenz*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Kontrollsequenz*⟩ ein TeX-Register vom Typ `\skip` ist, das mit `\newskip` oder `\newlength` allokiert wurde, im anderen Fall die ⟨*Falschausgabe*⟩.

\ifcslength{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifdeflength`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\ifdefdimen{⟨*Kontrollsequenz*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Kontrollsequenz*⟩ ein TeX-Register vom Typ `\dimen` ist, das mit `\newdimen` allokiert wurde, sonst die ⟨*Falschausgabe*⟩.

\ifcsdimen{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie `\ifdefdimen`, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

3.6.3 Tests für Zeichenfolgen

\ifstrequal{⟨*Zeichenfolge*⟩}{⟨*Zeichenfolge*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Vergleicht zwei ⟨*Zeichenfolgen*⟩ und liefert die ⟨*Wahrausgabe*⟩, falls sie gleich sind, sonst die ⟨*Falschausgabe*⟩. Die Zeichenfolgen werden für den Test nicht expandiert, Kategoriecodes werden ignoriert. Kontrollsequenzen in den ⟨*Zeichenfolgen*⟩ werden aus ihren Token zusammengesetzt und als Zeichenfolgen behandelt. Dieser Befehl ist robust.

\ifstrempty{⟨*Zeichenfolge*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Zeichenfolge*⟩ leer ist, sonst die ⟨*Falschausgabe*⟩. Die Zeichenfolge wird für den Test nicht expandiert.

\ifblank{⟨*Zeichenfolge*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Zeichenfolge*⟩ leer ist oder aus Leerzeichen besteht, sonst die ⟨*Falschausgabe*⟩. Die Zeichenfolge wird für den Test nicht expandiert.[4]

[4] Dieses Makro basiert auf Code von Donald Arseneau.

`\notblank`{⟨*Zeichenfolge*⟩}{⟨*Falschausgabe*⟩}{⟨*Wahrausgabe*⟩}

Wie `\ifblank`, kehrt aber den Test um.

3.6.4 Arithmetische Tests

`\ifnumcomp`{⟨*Ausdruck*⟩}{⟨*Operator*⟩}{⟨*Ausdruck*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Vergleicht zwei ganzzahlige ⟨*Ausdrücke*⟩ anhand des gegebenen ⟨*Operators*⟩ und liefert die ⟨*Wahrausgabe*⟩ oder die ⟨*Falschausgabe*⟩, abhängig vom Ergebnis des Vergleichs. Als ⟨*Operatoren*⟩ stehen <, > und = zur Verfügung. Beide ganzzahligen ⟨*Ausdrücke*⟩ werden mit `\numexpr` ausgewertet und können beliebiger, in diesem Kontext gültiger Code sein. Alle arithmetischen Ausdrücke können Leerzeichen enthalten. Hier einige Beispiele:

```
\ifnumcomp{3}{>}{6}{wahr}{falsch}
\ifnumcomp{(7 + 5) / 2}{=}{6}{wahr}{falsch}
\ifnumcomp{(7+5) / 4}{>}{3*(12-10)}{wahr}{falsch}
\newcounter{zaehlerA}
\setcounter{zaehlerA}{6}
\newcounter{zaehlerB}
\setcounter{zaehlerB}{5}
\ifnumcomp{\value{zaehlerA} * \value{zaehlerB}/2}{=}{15}{wahr}{falsch}
\ifnumcomp{6/2}{=}{5/2}{wahr}{falsch}
```

Technisch gesehen ist dieser Befehl eine LaTeX-Schnittstelle zur TeX-Primitiven `\ifnum` unter Einbeziehung von `\numexpr`. Wichtig ist, dass `\numexpr` die Ergebnisse aller Ausdrücke rundet, d. h. beide Ausdrücke werden vor dem Vergleich ausgewertet und gerundet. In der letzten Zeile des Beispiels lautet das Ergebnis des zweiten Ausdruck 2,5, das vor dem Vergleich auf 3 gerundet wird, `\ifnumcomp` liefert also ⟨*wahr*⟩.

`\ifnumequal`{⟨*Ausdruck*⟩}{⟨*Ausdruck*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Alternative Syntax für `\ifnumcomp{...}{=}{...}{...}{...}`.

`\ifnumgreater`{⟨*Ausdruck*⟩}{⟨*Ausdruck*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Alternative Syntax für `\ifnumcomp{...}{>}{...}{...}{...}`.

`\ifnumless`{⟨*Ausdruck*⟩}{⟨*Ausdruck*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Alternative Syntax für `\ifnumcomp{...}{<}{...}{...}{...}`.

`\ifnumodd`{⟨*Ausdruck*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wertet einen ganzzahligen ⟨ *Ausdruck*⟩ aus und liefert die ⟨*Wahrausgabe*⟩, wenn das Ergebnis eine ungerade Zahl ist, sonst die ⟨*Falschausgabe*⟩. Technisch gesehen ist dieser Befehl eine LaTeX-Schnittstelle zur TeX-Primitiven `\ifodd` unter Einbeziehung von `\numexpr`.

`\ifdimcomp`{⟨*Längenausdruck*⟩}{⟨*Operator*⟩}{⟨*Längenausdruck*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Vergleicht zwei ⟨*Längenausdrücke*⟩ (Ausdrücke vom TeX-Typ *dimen*) anhand des gegebenen ⟨*Operators*⟩ und liefert die ⟨*Wahrausgabe*⟩ oder die ⟨*Falschausgabe*⟩, abhängig vom Ergebnis des Vergleichs. Als ⟨*Operatoren*⟩ stehen <, > und = zur Verfügung. Beide ⟨*Längenausdrücke*⟩ werden mit `\dimenexpr` ausgewertet und können beliebiger, in diesem Kontext gültiger Code sein. Alle arithmetischen Ausdrücke können Leerzeichen enthalten. Hier einige Beispiele:

```
\ifdimcomp{1cm}{=}{28.45274pt}{wahr}{falsch}
\ifdimcomp{(7pt + 5pt) / 2}{<}{2pt}{wahr}{falsch}
\ifdimcomp{(3.725pt + 0.025pt) * 2}{<}{7pt}{wahr}{falsch}
\newlength{\laengeA}
\setlength{\laengeA}{7.25pt}
\newlength{\laengeB}
\setlength{\laengeB}{4.75pt}
\ifdimcomp{(\laengeA + \laengeB) / 2}{>}{2.75pt * 2}{wahr}{falsch}
\ifdimcomp{(\laengeA + \laengeB) / 2}{>}{25pt / 6}{wahr}{falsch}
```

Technisch gesehen ist dieser Befehl eine LaTeX-Schnittstelle zur TeX-Primitiven `\ifdim` unter Einbeziehung von `\dimenexpr`. Da `\ifdimcomp` und `\ifnumcomp` expandierbar sind, können sie auch verschachtelt werden.

```
\ifnumcomp{\ifdimcomp{5pt+5pt}{=}{10pt}{1}{0}}{>}{0}{wahr}{falsch}
```

`\ifdimequal`{⟨*Längenausdruck*⟩}{⟨*Längenausdruck*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Alternative Syntax für `\ifdimcomp{...}{=}{...}{...}{...}`.

`\ifdimgreater`{⟨*Längenausdruck*⟩}{⟨*Längenausdruck*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Alternative Syntax für `\ifdimcomp{...}{>}{...}{...}{...}`.

`\ifdimless`{⟨*Längenausdruck*⟩}{⟨*Längenausdruck*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Alternative Syntax für `\ifdimcomp{...}{<}{...}{...}{...}`.

3.6.5 Boolesche Ausdrücke

Die Befehle in diesem Abschnitt können anstelle des Befehls `\ifthenelse` aus dem Paket `ifthen` verwendet werden. Sie dienen dem gleichen Zweck, unterscheiden sich aber in Syntax, Konzeption und Implementierung. Im Gegensatz zu `\ifthenelse` liefern sie keine eigenen Tests, sondern bilden eine Schnittstelle zu anderen Tests. Jeder Test, der bestimmte syntaktische Voraussetzungen erfüllt, kann in booleschen Ausdrücken verwendet werden.

`\ifboolexpr`{⟨*Ausdruck*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wertet den gegebenen ⟨*Ausdruck*⟩ aus und liefert die ⟨*Wahrausgabe*⟩, wenn der Ausdruck wahr ist, sonst die ⟨*Falschausgabe*⟩. Der ⟨*Ausdruck*⟩ wird von links nach rechts ausgewertet. Die folgenden Bausteine können im ⟨*Ausdruck*⟩ verwendet werden (Details siehe unten): die Testoperatoren `togl`, `bool` und `test`, die logischen Operatoren `not`, `and` und `or`, sowie die Klammern `(...)` zur Gruppierung von Teilausdrücken. Leerzeichen, Tabulatoren und Zeilenumbrüche können beliebig zur optischen Gestaltung des ⟨*Ausdrucks*⟩ eingesetzt werden. Leerzeilen im ⟨*Ausdruck*⟩ sind nicht zulässig. Dieser Befehl ist robust.

`\ifboolexpe`{⟨*Ausdruck*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Eine expandierbare Fassung von `\ifboolexpr`, die in einem reinen Expansionskontext verwendet werden kann, z. B. in einer Definition mit `\edef` oder einer `\write`-Operation. Wichtig ist, dass alle im ⟨*Ausdruck*⟩ vorkommenden Tests expandierbar sein müssen, auch wenn `\ifboolexpe` nicht in einem reinen Expansionskontext verwendet wird.

`\whileboolexpr`{⟨*Ausdruck*⟩}{⟨*Code*⟩}

Wertet den gegebenen ⟨*Ausdruck*⟩ wie `\ifboolexpr` aus und wiederholt die Ausführung des ⟨*Codes*⟩, solange der Ausdruck wahr ist. Der ⟨*Code*⟩ kann beliebiger gültiger TeX- oder LaTeX-Code sein. Dieser Befehl ist robust.

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

Wie `\whileboolexpr`, aber mit negiertem ⟨*Ausdruck*⟩, d. h. der ⟨*Code*⟩ wird wiederholt, bis der ⟨*Ausdruck*⟩ wahr ist. Dieser Befehl ist robust.

Folgende Testoperatoren stehen in booleschen ⟨*Ausdrücken*⟩ zur Verfügung:

`togl` Der Operator `togl` testet den Zustand eines Flags, das mit `\newtoggle` definiert wurde. Beispiel:

```
\iftoggle{BoolescheVariable}{Wahrausgabe}{Falschausgabe}
```

wird zu

```
\ifboolexpr{ togl {BoolescheVariable} }{Wahrausgabe}{Falschausgabe}
```

Der Operator `togl` kann sowohl mit `\ifboolexpr`, als auch mit `\ifboolexpe` verwendet werden.

`bool` Der Operator `bool` testet auf Basis der einfachen TeX-Syntax, d. h. jeder Test, der normalerweise wie folgt geschrieben wird:

```
\iftest Wahrausgabe\else Falschausgabe\fi
```

Das schließt alle Flags ein, die mit `\newif` definiert wurden, sowie alle TeX-Primitiven wie `\ifmmode`. Das `\if`-Präfix der Primitiven oder des Flags wird dabei im ⟨*Namen*⟩ weggelassen. Beispiele:

```
\ifmmode Wahrausgabe\else Falschausgabe\fi
\ifmytest Wahrausgabe\else Falschausgabe\fi
```

wird zu

```
\ifboolexpr{ bool {mmode} }{Wahrausgabe}{Falschausgabe}
\ifboolexpr{ bool {mytest} }{Wahrausgabe}{Falschausgabe}
```

Das funktioniert auch mit Flags, die mit `\newbool` definiert wurden (siehe § 3.5.1). In diesem Fall wird

```
\ifbool{BoolescheVariable}{Wahrausgabe}{Falschausgabe}
```

zu

```
\ifboolexpr{ bool {BoolescheVariable} }{Wahrausgabe}{Falschausgabe}
```

Der Operator `bool` kann sowohl mit `\ifboolexpr`, als auch mit `\ifboolexpe` verwendet werden.

`test` Der Operator `test` testet auf Basis der LaTeX-Syntax, d. h. jeder Test, der normalerweise wie folgt geschrieben wird:

```
\iftest{Wahrausgabe}{Falschausgabe}
```

Dies schließt alle Makros ein, die die LaTeX-Syntax verwenden, d. h. das Makro muss die beiden Parameter ⟨*Wahrausgabe*⟩ und ⟨*Falschausgabe*⟩ erwarten und zwar am Ende der Parameterliste. Beispiele:

```
\ifdef{\EinMakro}{Wahrausgabe}{Falschausgabe}
\ifdimless{\textwidth}{365pt}{Wahrausgabe}{Falschausgabe}
\ifnumcomp{\value{EinZaehler}}{>}{3}{Wahrausgabe}{Falschausgabe}
```

Bei der Verwendung solcher Tests in booleschen ⟨*Ausdrücken*⟩, werden ihre Parameter ⟨*Wahrausgabe*⟩ und ⟨*Falschausgabe*⟩ weggelassen. Beispiel:

```
\ifcsdef{EinMakro}{Wahrausgabe}{Falschausgabe}
```

wird zu

```
\ifboolexpr{ test {\ifcsdef{EinMakro}} }{Wahrausgabe}{Falschausgabe}
```

und

```
\ifnumcomp{\value{EinZaehler}}{>}{3}{Wahrausgabe}{Falschausgabe}
```

wird zu

```
\ifboolexpr{
  test {\ifnumcomp{\value{EinZaehler}}{>}{3}}
}
{Wahrausgabe}
{Falschausgabe}
```

Der Operator `test` kann mit `\ifboolexpr` ohne Einschränkungen verwendet werden. Er kann auch mit `\ifboolexpe` verwendet werden, wenn der Test expandierbar ist. Einige Tests in § 3.6 sind robust, können aber nicht mit `\ifboolexpe` verwendet werden, auch wenn `\ifboolexpe` nicht in einem reinen Expansionskontext steht. Statt dessen kann `\ifboolexpr` verwendet werden, wenn der Test nicht expandierbar ist.

Da `\ifboolexpr` und `\ifboolexpe` mit einem gewissen Berechnungsaufwand verbunden sind, ist es wenig sinnvoll sie für einzelne Tests zu verwenden. Die Tests in § 3.6 sind effizienter als `test`, `\ifbool` in § 3.5.1 ist effizienter als `bool` und `\iftoggle` in § 3.5.2 ist effizienter als `togl`. Der Sinn von `\ifboolexpr` und `\ifboolexpe` ist, dass sie logische Operatoren und Teilausdrücke unterstützen. Die folgenden Operatoren stehen in booleschen ⟨*Ausdrücken*⟩ zur Verfügung:

`not` Der Operator `not` negiert den Wahrheitswert des unmittelbar darauffolgenden Bausteins. Es ist möglich ihn vor `togl`, `bool`, `test` und Teilausdrücken zu verwenden. Beispiele:

```
\ifboolexpr{
  not bool {BoolescheVariable}
}
{Wahrausgabe}
{Falschausgabe}
```

liefert ⟨*Wahrausgabe*⟩, wenn die `BoolescheVariable` falsch ist und ⟨*Falschausgabe*⟩, wenn die Variable wahr ist. Des Weiteren liefert

```
\ifboolexpr{
  not ( bool {boolA} or bool {boolB} )
}
{Wahrausgabe}
{Falschausgabe}
```

⟨*Wahrausgabe*⟩, wenn `boolA` und `boolB` falsch sind.

`and` Der Operator `and` stellt die Konjunktion (sowohl *a* als auch *b*) dar. Der boolesche ⟨*Ausdruck*⟩ ist wahr, wenn alle mit `and` verbundenen Bausteine wahr sind. Beispiele:

```
\ifboolexpr{
  bool {boolA} and bool {boolB}
}
{Wahrausgabe}
{Falschausgabe}
```

liefert ⟨*Wahrausgabe*⟩, wenn beide booleschen Tests wahr sind. Der Operator `nand` (negiertes `and`, d. h. nicht beide) ist nicht definiert, kann aber aus `and` und einer Negation zusammengesetzt werden. Beispiel:

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

kann ausgedrückt werden als

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

`or` Der Operator `or` stellt die nicht-exklusive Disjunktion (entweder *a* oder *b* oder beide) dar. Der boolesche ⟨*Ausdruck*⟩ ist wahr, wenn mindestens einer der mit `or` verbundenen Bausteine wahr ist. Beispiel:

```
\ifboolexpr{
  togl {toglA} or togl {toglB}
}
{Wahrausgabe}
{Falschausgabe}
```

liefert ⟨*Wahrausgabe*⟩ wenn einer von beiden Tests `toglA`, `toglB` oder beide wahr sind. Der Operator `nor` (negiertes `or`, d. h. weder weder *a* noch *b*, noch beide) ist nicht definiert, kann aber aus `or` und einer Negation zusammengesetzt werden. Beispiel:

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

kann ausgedrückt werden als

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

`(...)` Klammern begrenzen einen Teilausdruck im booleschen ⟨*Ausdruck*⟩. Der Teilausdruck wird ausgewertet und im umgebenden Ausdruck als einzelner Wahrheitswert behandelt. Teilausdrücke können verschachtelt werden. So ist beispielsweise der Ausdruck:

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

wahr, wenn beide Teilausdrücke wahr sind, d. h. wenn mindestens eine der Variablen `boolA`/`boolB` und mindestens eine der Variablen `boolC`/`boolD` wahr ist. Die Bildung von Teilausdrücken ist im Allgemeinen nicht erforderlich, wenn alle Bausteine entweder nur mit `and` oder nur mit `or` verbunden werden. Beispielsweise verhalten sich die Ausdrücke

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

wie man es erwartet: Der erste ist wahr, wenn alle Bausteine wahr sind. Der zweite ist wahr, wenn mindestens ein Baustein wahr ist. Werden dagegen `and` und `or` kombiniert, ist es immer ratsam, die Bausteine in Teilausdrücken zu gruppieren um mögliche Irrtümer zu vermeiden, die aus den Unterschieden der Semantik von booleschen Ausdrücken und natürlicher Sprache erwachsen können. So ist beispielsweise der Ausdruck

```
bool {Kaffee} and bool {Milch} or bool {Zucker}
```

schon wahr, wenn nur `Zucker` wahr ist, weil ohne Angabe der Gruppierung in diesem Fall `and` zuerst ausgewertet wird und als Teilausdruck für `or` dient. Im Gegensatz zur Bedeutung des Ausdrucks, wenn er in natürlicher Sprache (d. h. Kaffee und Milch oder Zucker) erscheint, wird der boolesche Ausdruck nicht wie folgt ausgewertet:

```
bool {Kaffee} and ( bool {Milch} or bool {Zucker} )
```

sondern streng von links nach rechts:

```
( bool {Kaffee} and bool {Milch} ) or bool {Zucker}
```

was vermutlich nicht die gewünschte Bestellung ist.

3.7 Listenverarbeitung

3.7.1 Benutzereingaben

Die Werkzeuge in diesem Abschnitt dienen vorangig der Verarbeitung von Benutzereingaben. Sollen Listen für die interne Verwendung in einem Paket erstellt werden, sind die Werkzeuge im Abschnitt 3.7.2 vermutlich besser geeignet, weil sie testen können, ob ein Element in einer Liste enthalten ist.

`\DeclareListParser{`⟨*Befehl*⟩`}{`⟨*Trennzeichen*⟩`}`

Dieser Befehl definiert einen Listenparser analog zum Befehl `\docsvlist`, der wie folgt definiert ist:

```
\DeclareListParser{\docsvlist}{,}
```

Der Kategoriecode des ⟨*Trennzeichens*⟩ wird vom Listenparser beachtet.

`\DeclareListParser*{`⟨*Befehl*⟩`}{`⟨*Trennzeichen*⟩`}`

Eine mit Stern markierte Variante von `\DeclareListParser` definiert einen Listenparser analog zum Befehl `\forcsvlist`, der wie folgt definiert ist:

```
\DeclareListParser*{\forcsvlist}{,}
```

`\docsvlist{`⟨*Element, Element, ...*⟩`}`

Dieser Befehl führt den Hilfsbefehl `\do` in einer Schleife für jedes Element einer Komma-separierten Liste aus und übergibt das Element als Parameter. Im Gegensatz zur `\@for`-Schleif aus dem LaTeX-Kern ist `\docvslist` expandierbar. Mit einer passenden Definition für `\do` können Listen im Kontext von `\edef` oder vergleichbarer Befehle verarbeitet werden. Durch Anfügen von `\listbreak` am Ende des Ersetzungstextes von `\do` kann die Verarbeitung der Liste unter Auslassung der verbleibenden Elemente abgebrochen werden. Leerraum nach Trennzeichen wird ignoriert. Soll ein Listenelement ein Komma oder Leerzeichen enthalten, muss es in geschwungene Klammern eingeschlossen werden. Die Klammern werden bei der Verarbeitung der Liste entfernt. Ein Beispiel, das eine Komma-separierte Liste in einer `itemize`-Umgebung ausgibt:

```
\begin{itemize}
\renewcommand*{\do}[1]{\item #1}
\docsvlist{Element1, Element2, {Element3a, Element3b}, Element4}
\end{itemize}
```

Ein weiteres Beispiel:

```
\renewcommand*{\do}[1]{* #1\MessageBreak}
\PackageInfo{MeinPaket}{%
  Beispielliste:\MessageBreak
  \docsvlist{Element1, Element2, {Element3a, Element3b}, Element4}}
```

In diesem Beispiel wird die Liste als Teil der Informationsnachricht in die Log-Datei geschrieben. Die Listenverarbeitung findet hier während der Schreiboperation `\write` statt.

`\forcsvlist{`⟨*Elementroutine*⟩`}{`⟨*Element, Element, ...*⟩`}`

Dieser Befehl verhält sich wie `\docvslist`, außer dass anstelle von `\do` eine bei jedem Aufruf anzugebende ⟨*Elementroutine*⟩ verwendet wird. Die ⟨*Elementroutine*⟩ kann auch eine Folge von Befehlen sein, vorausgesetzt der letzte Befehl erwartet das Element als letzten Parameter. Beispielsweise wandelt folgender Code eine Komma-separierte Liste in eine interne List mit dem Namen `\MeineListe` um:

```
\forcsvlist{\listadd\MeineListe}{Element1, Element2, Element3}
```

3.7.2 Interne Listen

Die Werkzeuge in diesem Abschnitt verarbeiten interne Listen. Eine ‚interne Liste' ist in diesem Kontext ein einfaches Makro ohne Parameter oder Präfixe, das zur Datensammlung verwendet wird. Diese Listen verwenden ein spezielles Zeichen als internen Elementtrenner.[5] Zur Verarbeitung von Benutzereingaben in Listenform, siehe die Werkzeuge in Abschnitt 3.7.1.

`\listadd{`⟨*Listenmakro*⟩`}{`⟨*Element*⟩`}`

Dieser Befehl hängt ein ⟨*Element*⟩ an ein ⟨*Listenmakro*⟩ an. Ein leeres oder aus Leerraum bestehendes ⟨*Element*⟩ wird nicht zur Liste hinzugefügt.

`\listgadd{`⟨*Listenmakro*⟩`}{`⟨*Element*⟩`}`

Wie `\listadd`, außer dass die Zuweisung global ist.

[5] Das Zeichen | mit dem Kategoriecode 3. Eine Liste kann deshalb nicht mit dem nach ihr benannten Befehl `\Listenname` gesetzt werden. Stattdessen kann `\show` zur Analyse verwendet werden.

\listeadd{⟨*Listenmakro*⟩}{⟨*Element*⟩}

Wie \listadd, außer dass das ⟨*Element*⟩ bei der Definition expandiert wird. Nur das neue ⟨*Element*⟩ wird expandiert, nicht das ⟨*Listenmakro*⟩. Wenn das expandierte ⟨*Element*⟩ leer ist oder aus Leeraum besteht, wird es nicht zur Liste hinzugefügt.

\listxadd{⟨*Listenmakro*⟩}{⟨*Element*⟩}

Wie \listeadd, außer dass die Zuweisung global ist.

\listcsadd{⟨*Name*⟩}{⟨*Element*⟩}

Wie \listadd, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\listcsgadd{⟨*Name*⟩}{⟨*Element*⟩}

Wie \listcsadd, außer dass die Zuweisung global ist.

\listcseadd{⟨*Name*⟩}{⟨*Element*⟩}

Wie \listeadd, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\listcsxadd{⟨*Name*⟩}{⟨*Element*⟩}

Wie \listcseadd, außer dass die Zuweisung global ist.

\dolistloop{⟨*Listenmakro*⟩}

Dieser Befehl führt den Hilfsbefehl \do in einer Schleife für jedes Element einer Liste aus und übergibt das Element als Parameter. Die Schleife selbst ist expandierbar. Durch Anfügen von \listbreak am Ende des Ersetzungstextes von \do kann die Verarbeitung der Liste unter Auslassung der verbleibenden Elemente abgebrochen werden. Hier ein Anwendungsbespiel, das eine interne Liste mit dem Namen \MeineListe in einer itemize-Umgebung ausgibt:

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

\dolistcsloop{⟨*Name*⟩}

Wie \dolistloop, außer dass der erste Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\forlistloop{⟨*Elementroutine*⟩}{⟨*Listenmakro*⟩}

Dieser Befehl verhält sich wie \dolistloop, außer dass anstelle von \do eine ⟨*Elementroutine*⟩ verwendet wird, die bei jedem Aufruf angegeben wird. Die ⟨*Elementroutine*⟩ kann auch eine Folge von Befehlen sein, vorausgesetzt der letzte Befehl erwartet das Element als letzten Parameter. Das folgende Beispiel gibt alle Elemente der internen Liste \MeineListe in einer \itemize-Umgebung aus, zählt sie dabei und gibt am Ende die Anzahl aus:

```
\newcounter{Elementzaehler}
\begin{itemize}
\forlistloop{\stepcounter{Elementzaehler}\item}{\MeineListe}
\item Gesamt: \number\value{Elementzaehler} Elemente
\end{itemize}
```

\forlistcsloop{⟨*Elementroutine*⟩}{⟨*Name*⟩}

Wie \forlistloop, außer dass der zweite Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\ifinlist{⟨*Element*⟩}{⟨*Listenmakro*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Dieser Befehl liefert die ⟨*Wahrausgabe*⟩, falls ein angegebenes ⟨*Element*⟩ in einem ⟨*Listenmakro*⟩ enthalten ist, sonst die ⟨*Falschausgabe*⟩. Dieser Befehl verwendet eine Mustererkennung, die auf dem Parameterscanner von TeX basiert, um zu ermitteln, ob die gesuchte Zeichenfolge in der Liste enthalten ist. Das ist für gewöhnlich schneller als eine Schleife über alle Listenelemente, aber es führt dazu, dass die Elemente keine geschweiften Klammern enthalten dürfen, weil sie sonst vom Scanner überlesen werden. Mit anderen Worten: Dieser Befehl ist nützlich, wenn es um den Umgang mit einfachen Zeichenfolgen geht statt mit formatierten Daten. Soll eine Liste mit formatierten Daten durchsucht werden, ist es sicherer \dolistloop zu verwenden und wie folgt auf das Vorhandensein eines Elementes zu prüfen:

```
\renewcommand*{\do}[1]{%
  \ifstrequal{#1}{Element}
    {Element gefunden!\listbreak}
    {}}
\dolistloop{\MeineListe}
```

\xifinlist{⟨*Element*⟩}{⟨*Listenmakro*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie \ifinlist, außer dass das ⟨*Element*⟩ vor der Suche expandiert wird.

\ifinlistcs{⟨*Element*⟩}{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie \ifinlist, außer dass der zweite Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

\xifinlistcs{⟨*Element*⟩}{⟨*Name*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Wie \xifinlist, außer dass der zweite Parameter der ⟨*Name*⟩ einer Kontrollsequenz ist.

3.8 Verschiedenes

\rmntonum{⟨*römische Zahl*⟩}

Die TeX-Primitive \romannumeral wandelt eine Ganzzahl in eine römische Zahl um, aber TeX und LaTeX liefern keinen Befehl für die Rückrichtung. Der Befehl \rmntonum füllt diese Lücke. Er erwartet eine ⟨*römische Zahl*⟩ und wandelt sie in die entsprechende Ganzzahl um. Da der Befehl expandierbar ist, kann er auch in Zuweisungen zu Zählern oder arithmetischen Tests verwendet werden:

```
\rmntonum{mcmxcv}
\setcounter{Zaehler}{\rmntonum{CXVI}}
\ifnumless{\rmntonum{mcmxcviii}}{2000}{Wahrausgabe}{Falschausgabe}
```

Der Parameter ⟨*römische Zahl*⟩ muss eine einfache Zeichenfolge sein. Er wird vor der Berechnung aus seinen Token zusammengesetzt. Die Berechnung ignoriert Groß-/Kleinschreibung und Leerraum. Enthält die ⟨*römische Zahl*⟩ einen ungültigen Token, liefert \rmntonum den Wert -1 - ein leerer Parameter liefert dagegen eine leere Zeichenfolge. \rmntonum prüft die ⟨*römische Zahl*⟩ nicht auf formelle Richtigkeit. Z. B. liefern V und VX beide 5, IC ergibt 99.

\ifrmnum{⟨*Zeichenfolge*⟩}{⟨*Wahrausgabe*⟩}{⟨*Falschausgabe*⟩}

Liefert die ⟨*Wahrausgabe*⟩, wenn die ⟨*Zeichenfolge*⟩ eine römische Zahl ist, sonst die ⟨*Falschausgabe*⟩. Die ⟨*Zeichenfolge*⟩ wird vor dem Test aus ihren Token zusammengesetzt. Der Test ignoriert Groß-/Kleinschreibung und Leerraum. Der Test \ifrmnum prüft die römische Zahl nicht auf formelle Richtigkeit. Z. B. liefern V und VXV beide die ⟨*Wahrausgabe*⟩. Genau genommen untersucht \ifrmnum lediglich ob die ⟨*Zeichenfolge*⟩ nur aus Zeichen besteht, die in römischen Zahlen vorkommen dürfen, aber nicht ob sie in der Reihenfolge auch eine gültige römische Zahl bilden.

4 Versionsgeschichte

Die Versionsgeschichte ist eine Liste aller Änderungen, die für Benutzer dieses Paketes relevant sind. Änderungen technischer Art, die nicht die Benutzerschnittstelle betreffen, wurden nicht aufgenommen. Änderungen, bei denen etwas *verbessert* oder *hinzugefügt* wurde, sind syntaktisch rückwärtskompatibel - wie das Hinzufügen optionaler Parameter oder neuer Befehle. Änderungen, bei denen etwas *modifiziert* wurde, bedürfen der Aufmerksamkeit der Benutzers. Durch sie ist es möglicherweise in einigen, hoffentlich sehr wenigen Fällen nötig, existierende Dokumente anzupassen. Die Zahlen am rechten Rand bezeichnen den betreffenden Abschnitt dieses Handbuchs.

2.1 2011-01-03

2.0a 2010-09-12

2.0 2010-08-21

Testen von \par in Makrotests erlaubt 3.6.1

Einige Fehler behoben

1.9 2010-04-10

\letcs verbessert 3.1.1

\csletcs verbessert 3.1.1

\listeadd verbessert 3.7.2

\listxadd verbessert 3.7.2

\notblank hinzugefügt 3.6.3

\ifnumodd hinzugefügt 3.6.4

\ifboolexpr hinzugefügt 3.6.5

\ifboolexpe hinzugefügt 3.6.5

\whileboolexpr hinzugefügt 3.6.5

\unlessboolexpr hinzugefügt 3.6.5

1.8 2009-08-06

\deflength verbessert 2.4

\ifnumcomp hinzugefügt 3.6.4

\ifnumequal hinzugefügt 3.6.4

\ifnumgreater hinzugefügt 3.6.4

\ifnumless hinzugefügt 3.6.4

\ifdimcomp hinzugefügt 3.6.4

\ifdimequal hinzugefügt 3.6.4

\ifdimgreater hinzugefügt 3.6.4

\ifdimless hinzugefügt 3.6.4

1.7 2008-06-28

\AfterBeginDocument in \AfterEndPreamble umbenannt (Namenskonflikt) 2.5

Konflikte mit hyperref ausgeräumt

Handbuch geringfügig überarbeitet

1.6 2008-06-22

\robustify verbessert 2.2

\patchcmd und \ifpatchable verbessert 3.4

\apptocmd verbessert und modifiziert 3.4
\pretocmd verbessert und modifiziert 3.4
\ifpatchable* hinzugefügt 3.4
\tracingpatches hinzugefügt 3.4
\AfterBeginDocument hinzugefügt 2.5
\ifdefmacro hinzugefügt 3.6.1
\ifcsmacro hinzugefügt 3.6.1
\ifdefprefix hinzugefügt 3.6.1
\ifcsprefix hinzugefügt 3.6.1
\ifdefparam hinzugefügt 3.6.1
\ifcsparam hinzugefügt 3.6.1
\ifdefprotected hinzugefügt 3.6.1
\ifcsprotected hinzugefügt 3.6.1
\ifdefltxprotect hinzugefügt 3.6.1
\ifcsltxprotect hinzugefügt 3.6.1
\ifdefempty hinzugefügt 3.6.1
\ifcsempty hinzugefügt 3.6.1
\ifdefvoid verbessert 3.6.1
\ifcsvoid verbessert 3.6.1
\ifstrempty hinzugefügt 3.6.3
\setbool hinzugefügt 3.5.1
\settoggle hinzugefügt 3.5.2

1.5 2008-04-26

\defcounter hinzugefügt 2.4
\deflength hinzugefügt 2.4
\ifdefstring hinzugefügt 3.6.1
\ifcsstring hinzugefügt 3.6.1
\rmntonum verbessert 3.8
\ifrmnum hinzugefügt 3.8
Dem Handbuch erweiterte PDF-Lesezeichen hinzugefügt
Handbuch geringfügig überarbeitet

1.4 2008-01-24

Konflikt mit `tex4ht` ausgeräumt

1.3 2007-10-08

Paket von `elatex` in `etoolbox` umbenannt 1
`\newswitch` in `\newtoggle` umbenannt (name clash) 3.5.2
`\provideswitch` in `\providetoggle` umbenannt (consistency) 3.5.2
`\switchtrue` in `\toggletrue` umbenannt (consistency) 3.5.2
`\switchfalse` in `\togglefalse` umbenannt (consistency) 3.5.2
`\ifswitch` in `\iftoggle` umbenannt (consistency) 3.5.2
`\notswitch` in `\nottoggle` umbenannt (consistency) 3.5.2
`\AtEndPreamble` hinzugefügt 2.5
`\AfterEndDocument` hinzugefügt 2.5
`\AfterPreamble` hinzugefügt 2.5
`\undef` hinzugefügt 3.1.1
`\csundef` hinzugefügt 3.1.1
`\ifdefvoid` hinzugefügt 3.6.1
`\ifcsvoid` hinzugefügt 3.6.1
`\ifdefequal` hinzugefügt 3.6.1
`\ifcsequal` hinzugefügt 3.6.1
`\ifstrequal` hinzugefügt 3.6.3
`\listadd` hinzugefügt 3.7.2
`\listeadd` hinzugefügt 3.7.2
`\listgadd` hinzugefügt 3.7.2
`\listxadd` hinzugefügt 3.7.2
`\listcsadd` hinzugefügt 3.7.2
`\listcseadd` hinzugefügt 3.7.2
`\listcsgadd` hinzugefügt 3.7.2
`\listcsxadd` hinzugefügt 3.7.2
`\ifinlist` hinzugefügt 3.7.2
`\xifinlist` hinzugefügt 3.7.2
`\ifinlistcs` hinzugefügt 3.7.2

1.2 2007-07-13

1.1 2007-05-28

1.0 2007-05-07

Erste Veröffentlichung