

# Makroprogrammierung in L<sup>A</sup>T<sub>E</sub>X

J.-E. Thiede

Seminar L<sup>A</sup>T<sub>E</sub>X and Friends, SS 2005

Seminarleiter: Prof. Dr. Renz, Prof. Dr. Iglar

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>2</b>
<b>2</b>	<b>Grundlegende Dateien</b>	<b>2</b>
<b>3</b>	<b>Robuste und fragile Befehle</b>	<b>2</b>
3.1	Überschrift <sup>1</sup> mit ...	
	Zeilenumbruch . . . . .	3
<b>4</b>	<b>Arbeiten mit Zählern</b>	<b>4</b>
4.1	Anzeigen eines Zählerstandes . . . . .	4
4.2	Befehle zum Ändern der Zählerstände . . . . .	5
4.3	Definition eigener Zähler . . . . .	6
<b>5</b>	<b>Erstellen von eigenen Makros</b>	<b>7</b>
5.1	Einfache Befehle . . . . .	7
5.2	Befehle mit Parametern . . . . .	9
5.3	Befehle mit optionalem Parameter . . . . .	11
5.4	Erstellen von eigenen Umgebungen . . . . .	12
5.5	Redefinieren vorhandener Makros . . . . .	17
<b>6</b>	<b>Listen</b>	<b>18</b>
6.1	Verändern von Listen . . . . .	18
	6.1.1 Itemize-Liste . . . . .	18
	6.1.2 Enumerate-Liste . . . . .	19
	6.1.3 Description-Liste . . . . .	20
6.2	Neue Listen . . . . .	20
<b>7</b>	<b>Kommunikation mit L<sup>A</sup>T<sub>E</sub>X, Ein- und Ausgabe</b>	<b>23</b>
<b>8</b>	<b>Kontrollstrukturen in L<sup>A</sup>T<sub>E</sub>X bzw. T<sub>E</sub>X</b>	<b>24</b>
8.1	Variablen . . . . .	24
8.2	Operationen . . . . .	25
8.3	Kontrollstrukturen . . . . .	27
	8.3.1 Bedingte Anweisungen . . . . .	27
	8.3.2 Schleifen . . . . .	28
<b>9</b>	<b>Fazit</b>	<b>30</b>
<b>10</b>	<b>Quellen</b>	<b>31</b>

---

<sup>1</sup>Fußnote

# 1 Einführung

$\LaTeX$  ist ein Softwarepaket, das die Benutzung des Textsatzprogramms  $\TeX$  mit Hilfe von Makros vereinfacht.  $\LaTeX$  wurde 1984 von Leslie Lamport entwickelt. Der Name bedeutet soviel wie *Lamports  $\TeX$*  und existiert derzeit in der Version 2 $\epsilon$ . Das zugrunde liegende Prinzip ist dabei „Konzentration auf den Inhalt – nicht das Design“. Der Autor versieht seinen Text mit *logischen* Strukturinformationen die von  $\LaTeX$  entsprechend in Textsatzanweisungen für  $\TeX$  umgesetzt werden. Um komplexere Arbeiten mit  $\LaTeX$  erfolgreich abzuschließen, ist das Standardverhalten, so wie von Leslie Lamport programmiert, jedoch nicht immer ausreichend. Manchmal ist es notwendig, dass unser Dokument eben eigene Regeln befolgt, die von Lamport so nicht vorgesehen wurden. In diesem Fall müssen wir genauer verstehen, wie  $\LaTeX$  einige Dinge handhabt und wie wir eigene Kommandos definieren können. Letztendlich ist  $\LaTeX$  selbst ja eine Sammlung von Makros für  $\TeX$  die es uns erlaubt uns auf den Inhalt und die Strukturierung eines Dokumentes zu konzentrieren. Bekannte Klassen wie Beamer sowie Pakete wie PGF, Graphicx usw. stellen ebenso Erweiterungen und Modifikationen des Standardverhaltens dar.

Viele der Informationen, die hier vorgestellt werden, stammen aus dem Dokument „ $\LaTeX$ – Fortgeschrittene Anwendungen“ von Manuela Jürgens von der Fernuniversität Hagen. Das Dokument ist frei verfügbar uns für alle Interessierten sehr zu empfehlen.

Im Folgenden wird kein Unterschied zwischen den Begriffen Kommando, Makro, Anweisung und Befehl gemacht, die Begriffe werden synonym verwendet.

## 2 Grundlegende Dateien

Einige der Makros, die wir verwenden und eventuell auch verändern wollen, benötigen Hilfsdateien um Informationen für weitere  $\LaTeX$ -Läufe abzulegen. Folgende Dateien werden von  $\LaTeX$  automatisch generiert:

Protokoll des $\LaTeX$ -Laufs	.log
Hilfsdatei für Querverweise	.aux
Inhaltsverzeichnis	.toc
Abbildungsverzeichnis	.lof
Tabellenverzeichnis	.lot
Sachregister	.idx

Weitere wichtige Dateien sind die .sty und die .cls Dateien. Diese Dateien befinden sich nicht im Arbeitsverzeichnis (also dort wo die .tex-Datei liegt) sondern enthalten unter anderem die Makrodefinitionen die wir in  $\LaTeX$  verwenden.

## 3 Robuste und fragile Befehle

Für einige der folgenden Erläuterungen ist es nützlich, wenn man den Unterschied zwischen so genannten „fragilen“ (zerbrechlichen) und „robusten“  $\LaTeX$ -Befehlen kennt. Viele  $\LaTeX$ -Kommandos bieten die Möglichkeit weitere Argumente anzugeben in der Form:  $\backslash$ Kommando[OptionalesArgument]{Argument}.

Die Argumente sollen dabei manchmal wiederum weitere Kommandos enthalten, z.B. wenn innerhalb einer Überschrift ein Wort betont oder eine Fußnote verwendet werden soll:

BEISPIEL

## Dieses *Wort* ist mit `emph`<sup>2</sup> betont worden

Dabei kann es jedoch zu Problemen (und Fehlermeldungen) kommen, wenn „fragile“ Befehle in „beweglichen“ Argumenten verwendet werden.

**Bewegliches Argument** ist ein Parameter, der im Dokument nicht einfach nur an einer Stelle ausgegeben wird, sondern der von L<sup>A</sup>T<sub>E</sub>X auch an anderen Stellen (Beispielsweise im Inhaltsverzeichnis) verwendet wird.

Der Befehl `\section{Überschrift des Abschnitts}` bekommt als Argument die gewünschte Überschrift übergeben. Diese Überschrift wird aber nicht nur an der entsprechenden Stelle im Dokument angezeigt, sondern wird auch in der .toc-Datei vermerkt um im Inhaltsverzeichnis eingefügt zu werden. Der `\section{}`-Befehl besitzt also ein bewegliches Argument. Innerhalb eines solchen beweglichen Argumentes können nur „robuste“ Kommandos verwendet werden. Das `\emph{}`-Kommando ist beispielsweise robust.

Um dennoch fragile Kommandos innerhalb von beweglichen Argumenten verwenden zu können, kann das `\protect`-Kommando vorangestellt werden. Damit wird jeweils genau das nachfolgende Kommando „geschützt“. Um Fußnoten innerhalb von Überschriften zu verwenden, ist dies zum Beispiel nötig<sup>3</sup>.

Die Befehlsfolge für eine Überschrift mit Zeilenumbruch und Fußnote:

```
\subsection{
  Überschrift\protect\footnote{Fußnote} mit \dots
\protect\\ Zeilenumbruch
}
```

BEISPIEL

### 3.1 Überschrift<sup>4</sup> mit ... Zeilenumbruch

---

<sup>2</sup>Diese Fußnote befindet sich ebenfalls in der Überschrift, kann dort jedoch nur in Verbindung mit `protect` verwendet werden. Das `verb`-Kommando kann auch bei Verwendung von `protect` nicht in Überschriften verwendet werden.

<sup>3</sup>Tatsächlich wird in dem obigen Beispiel eine Fußnote mit `protect` verwendet, wird jedoch trotzdem nicht angezeigt! In diesem Fall handelt es sich um eine Besonderheit, weil der `section`-Befehl mit einem `*` versehen ist, also nicht im Inhaltsverzeichnis auftaucht und folglich eben auch kein bewegliches Argument besitzt. Lässt man den Stern weg taucht auch die Fußnote auf, allerdings natürlich unterhalb des Inhaltsverzeichnisses. Merkwürdig (im Sinne des Wortes).

<sup>4</sup>Fußnote

## 4 Arbeiten mit Zählern

Eine der positiven Eigenschaften von  $\text{\LaTeX}$  ist die automatische Nummerierung von Kapiteln, Abbildungen, Fußnoten, Listenelemente, Definitionen, usw. Dieses automatische Durchzählen wird durch Zählervariablen gesteuert, die wir verändern, auf die wir uns beziehen oder mit denen wir rechnen können. Abgesehen davon haben wir sogar die Möglichkeit eigene neue Zähler zu definieren.

Die Bezeichnung der meisten Zählervariablen entspricht dem  $\text{\LaTeX}$ -Befehl, der diesen Zähler beeinflusst. Der `\Section`-Befehl beeinflusst beispielsweise den Section-Zähler. Die vordefinierten Standardzähler sind unter anderem:

Kategorie	Zähler
Textgliederung:	part, chapter, section, subsection, subsubsection, paragraph, subparagraph
Seiten:	page
Listen:	enumi, enumii, enumiii, enumiv itemi, itemii, itemiii, itemiv
Sonstige:	figure, table, footnote, equation

Alle diese Zähler sind zunächst einmal mit 0 initialisiert, und werden von dem Befehl, der die Ausgabe des Zählerstandes vornimmt vor der Ausgabe erhöht. Eine Ausnahme bildet dabei der page-Zähler, der bereits zu Beginn mit 1 initialisiert und jeweils erst nach der Ausgabe um 1 erhöht wird. Die verschiedenen Zähler stehen teilweise in Beziehung zueinander, so werden beim Hochzählen des chapter-Zählers alle anderen Zähler zur Textgliederung wieder auf 0 gesetzt.

### 4.1 Anzeigen eines Zählerstandes

Um den aktuellen Wert eines Zählers anzeigen zu lassen, kann der `\the...`-Befehl verwendet werden. Dabei werden die „...“ durch die Bezeichnung des jeweiligen Zählers ersetzt:

```
\thepage           Seitenzähler:      4
\thefootnote       Fußnotenzähler:    4
```

BEISPIEL

Die Darstellung eines Zählers kann in arabischen Ziffern (Standard) aber auch als römische Ziffern, Buchstaben des Alphabetes oder abstrakten Symbolen erfolgen. Die notwendigen Befehle dafür sind:

- `\arabic{zähler}` arabische Ziffern
- `\roman{zähler}` kleine römische Ziffern (iv)
- `\Roman{zähler}` große römische Ziffern (IV)
- `\alph{zähler}` kleine Buchstaben (d)
- `\Alph{zähler}` große Buchstaben (D)
- `\fnsymbol{zähler}` max. 9 spezielle Symbole (\$)

```

\arabic{page}      5
\roman{page}      v
\Roman{page}      V
\Alph{page}       E

```

Um zu erreichen, dass die Werte eines bestimmten Zählers immer in einer speziellen Darstellung ausgedruckt werden, muss das `\the...`-Kommando entsprechend redefiniert werden.<sup>v</sup> Die Redefinition von Makros erfolgt mit dem `\renewcommand`-Befehl, der im Abschnitt ??, Seite ?? näher erläutert wird.<sup>vi</sup>

Verwendet wird das `\renewcommand`-Makro folgendermaßen:

```
\renewcommand{\thefootnote}{\roman{footnote}}7
```

Die Wirkung des Kommandos ist an den Fußnoten dieses Abschnittes erkennbar, zunächst wurde das `\the...`-Makro auf römische Ziffern umgestellt, anschließend wieder auf arabische.

## 4.2 Befehle zum Ändern der Zählerstände

Um den aktuellen Wert eine beliebigen Zählers zu beeinflussen stehen folgende Kommandos zur Verfügung:

Befehl	robust/fragil	Beschreibung
<code>\setcounter{zähler}{wert}</code>	fragil	Mit diesem Befehl lässt sich ein Zähler auf einen beliebigen Wert setzen. Beispiel: <code>\setcounter{section}{42}</code>
<code>\addtocounter{zähler}{wert}</code>	fragil	Einem Zähler kann auch ein beliebiger Wert hinzu addiert oder (bei negativen Werten) abgezogen werden. Beispiel: <code>\addtocounter{section}{-12}</code>
<code>\stepcounter{zähler}</code>	fragil	Das <code>stepcounter</code> -Kommando setzt den angegeben Zähler um einen Schritt weiter. D.h. der Zähler wird um 1 erhöht, und alle verbundenen (untergeordneten) Zähler werden wieder auf 0 gesetzt. Beispiel: <code>\stepcounter{section}</code>
<code>\refstepcounter{zähler}</code>	fragil	Wirkt genauso wie <code>stepcounter</code> , auf den angegeben Zähler können jedoch Querverweise (Referenzen) weisen.

<sup>v</sup>Diese Fußnote ist nun römisch, genauso wie alle folgenden

<sup>vi</sup>Aber nun stellen wir die Ansicht wieder um.

<sup>7</sup>Sehen Sie?

<code>\value{zähler}</code>	robust	Gibt den aktuellen Wert des angegebenen Zählers zurück. Kann für Wertzuweisungen an andere Zähler benutzt werden.
<code>\theZÄHLER</code>	robust	Druckt den aktuellen Wert des angegebenen Zählers aus. Die Bezeichnung des Zählers wird nicht als Argument, sondern als Teil des Befehlsnamens angegeben. <code>\theSection</code> <code>\theFootnote</code>

### 4.3 Definition eigener Zähler

Mit dem Kommando `\newcounter{neuerZähler}[übergZähler]` kann ein neuer eigener Zähler definiert werden. Dieser eigene Zähler wird standardmäßig mit 0 initialisiert. Die Angabe des übergeordneten Zählers sorgt dafür, dass der neu definierte Zähler automatisch zurück auf 0 gesetzt wird, sobald der übergeordnete Zähler mit `\stepcounter{}` weitergezählt wird. Die Funktionsweise von über- bzw. untergeordneten Zählern kann gut anhand der Standardzähler verdeutlicht werden:

```
\newcounter{chapter}
\newcounter{section}[chapter]
\newcounter{subsection}[section]
```

BEISPIEL

Durch die Angabe der übergeordneten Zähler wird hier dafür gesorgt, dass beim Weiterzählen des chapter-Zählers der section-Zähler wieder auf 0 gesetzt wird. Die Veränderung des section-Zählers wirkt sich wiederum auf den subsection-zähler aus, usw.

Ein Beispiel für die Verwendung eines eigenen Zählers könnte die Nummerierung von „Merkeinheiten“ in einer Mathematik-Mitschrift bei Prof. Ecker sein (ca. 900) oder die Strukturierung von Definitionen und Beweisen in anderen Vorlesungen:

```
\newcounter{ME}[subsection]
```

BEISPIEL

```
Merkeinheit \theME: Satz von Pythagoras \stepcounter{ME} \\
Merkeinheit \theME: Satz von Euklid \stepcounter{ME} \\
Merkeinheit \theME: Satz von Thales \stepcounter{ME} \\
```

Merkeinheit 0: Satz von Pythagoras  
Merkeinheit 1: Satz von Euklid  
Merkeinheit 2: Satz von Thales

Das volle Potential von selbst definierten Zählern zeigt sich jedoch erst in Verbindung mit der Definition eigener Makros. Dann kann ein neuer  $\LaTeX$ -Befehl erstellt werden, der sowohl das Anzeigen des aktuellen Zählerstandes in der gewünschten Formatierung, als auch das Erhöhen des Zählers übernimmt.

## 5 Erstellen von eigenen Makros

### 5.1 Einfache Befehle

Wie bereits in der Einführung dargestellt besteht eines der wichtigsten Merkmale von  $\text{\LaTeX}$  darin, es dem Autor zu ermöglichen, sich auf Inhalt und Strukturierung des Dokumentes zu konzentrieren. Doch wenn das Ziel darin besteht, spezielle Gestaltungsmerkmale häufig zu verwenden, die nicht als eigene  $\text{\LaTeX}$ -Kommandos vorgesehen sind, leidet die Produktivität erheblich. Nehmen wir an, wir wollen alle Merkeinheiten oder z.B. Beweise in einer Mitschrift in einem grau hinterlegten Kasten, mit eigenem Zähler nummeriert, in spezieller Schriftart und -größe, sowie mit zusätzlichen Informationen versehen haben. Prinzipiell erlaubt uns  $\text{\LaTeX}$  die entsprechenden Formatierungen:

Beweis 1: Satz von Pythagoras

Autor:           Pythagoras  
 Jahr:            500 v. Chr.

Beweis mittels der Ähnlichkeit von Dreiecken:

$$a/p = c/a$$

$$b/q = c/b$$

$$a^2 = cp$$

$$b^2 = cq$$

$$a^2 + b^2 = c(p+q) = c^2$$

q.e.d

in dem wir folgende Makros verwenden:

BEISPIEL

```

\stepcounter{beweis}
\shadowbox{\fcolorbox{red}{yellow}{\parbox{\textwidth}
{
  \large{Beweis \thebeweis: Satz von Pythagoras} \\
  \begin{tabbing}
    xxxxxxxxxxxxxx\=xxxxxxxxxxx\kill
    Autor: \> Pythagoras \\
    Jahr:  \> 500 v. Chr. \\
  \end{tabbing}
}
}

```

```

\end{tabbing}

Beweis mittels der Ähnlichkeit von Dreiecken:
\includegraphics{Pythagoras}

\rightline{\Large\textbf{\textit{q.e.d.}}}
}}

```

Allein an der letzten Zeile, den L<sup>A</sup>T<sub>E</sub>X-Befehlen zur Formatierung des „q.e.d.“, lässt sich die Problematik erkennen:

- Vier Makros müssen geschachtelt werden, um das gewünschte Ziel zu erreichen,
- Der Text ist nur schlecht lesbar,
- Der Autor muss sich im Gegensatz zum L<sup>A</sup>T<sub>E</sub>X-Prinzip mit der Formatierung auseinander setzen.

Es ist leicht einzusehen, dass der Arbeitsaufwand sehr hoch ist, wenn obige Befehlsfolgen öfters verwendet werden (denken Sie an die besagten 900 Merkeinheiten!)

Der Ausweg besteht in der Definition eigener Makros (also eigener L<sup>A</sup>T<sub>E</sub>X-Komandos) mit dem `\newcommand{\neuesMakro}{Inhalt}`-Befehl. Der Inhalt des Makros kann natürlich weitere Kommandos enthalten.

Beispiel: Wir wollen, dass mit dem selbst definierten Befehl `\qed` automatisch rechtsbündig in fetter und in kursiver Schrift „q.e.d.“ erscheint:

***q.e.d***

BEISPIEL

```

\newcommand{\qed}{
  \begin{flushright}
    \Large\textbf{\textit{q.e.d.}}
  \end{flushright}
}

\qed

```

Auf diese Weise lassen sich für bestimmte Begriffe die immer wieder verwendet werden, sehr leicht entsprechende Formatierungen definieren. Ein Anwendungsgebiet könnte die Aufrechterhaltung der Corporate-Identity in Geschäftskorrespondenz und in Handbüchern sein. Der Firmenname oder die Produktbezeichnung können einheitlich festgelegt sein, und trotz komplexer Definition immer wieder mit einem Befehl verwendet werden. Beispiel: der L<sup>A</sup>T<sub>E</sub>X-Befehl!

Mit der Definition eigener Makros können wir nun auch die selbst definierten Zähler effektiv verwenden: Wir können nun einen Befehl erstellen, der automatisch die aktuelle Merkeinheit hochzählt und gleichzeitig ausgibt:

BEISPIEL

```

\newcounter{MECount}
\setcounter{MECount}{1}

```



Das Makro erwartet also 4 Parameter und soll unter anderem das Weiterzählen eines Zählers übernehmen.

BEISPIEL

```

\newcommand{\beweis}[4]{
  \stepcounter{MECount}
  \shadowbox{\fcolorbox{red}{yellow}{\parbox{\textwidth}
  {
    \label{#1}
    \large{\textbf{Merkeinheit \theMEcount: #1}} \\
    \begin{tabbing}
      xxxxxxxxxxxxxxx\=xxxxxxxxxxx\kill
      Autor: \> #2 \\
      Jahr: \> #3 \\
    \end{tabbing}
  #4 \\
  \rightline{\Large\textbf{\textit{q.e.d}}}}
  }}}
}

\beweis{NameBeweis}{Autor}{Jahr}{BeweisText}

```

### Merkeinheit 5: Spline-Interpolation

Autor: Schoenberg  
 Jahr: 1946

Folge von Polynomstücken 3. Grades, die in den Stützstellen in den ersten beiden Ableitungen übereinstimmen, und ...

*q.e.d*

### Merkeinheit 6: Fast Fourier Transformation (FFT)

Autor: Cooley und Tukey  
 Jahr: 1965

Das Problem der Berechnung einer DFT der Größe  $n$  wird nun zunächst in zwei Berechnungen der DFT der Größe  $\frac{n}{2}$  aufgeteilt, ...

*q.e.d*

Das Makro formatiert nicht nur alle Eingaben richtig, sondern zählt auch Automatisch den Beweisähler hoch, den wir im Abschnitt 5.1, Seite 8 bereits definiert hatten! Auf diese Weise können in einem Skript leicht Dutzende von

Beweisen einheitlich formatiert und durchgängig nummeriert werden, ohne dass der Autor immer wieder sämtliche Kommandos eingeben muss.

### 5.3 Befehle mit optionalem Parameter

Um eigene Makros effektiv einsetzen zu können, sollten wir in der Lage sein, auch optionale Argumente zu definieren. In obigem Beispiel könnten wir den Namen des Autors optional halten, und nur dann eintragen, wenn er uns bekannt ist. Weitere sinnvolle Ergänzungen könnten sein:

- optionale Breitenangabe
- optionale Angabe der Hintergrundfarbe
- ...

Zunächst einmal ist wichtig, dass wir bei Verwendung eines „normalen“, einfachen, optionalen Parameters (also wenn wir in unserem Makro keine IF-THEN-ELSE- Konstrukte verwenden wollen) für den optionalen Parameter einen Vorgabewert (Defaultwert) angeben *müssen*. Dies schränkt uns in der Funktionalität des Makros natürlich ein, macht uns das Erstellen aber leichter: Man muss eben kein Programmierer sein und sich mit Kontrollstrukturen auskennen, um ein L<sup>A</sup>T<sub>E</sub>X-Makro zu schreiben. Alle Argumente, optional oder nicht werden in einem Makro verwendet, kein Teil der Makrodefinition wird übersprungen! D.h. auch, wenn wir die optionale Breitenangabe weglassen, werden sämtliche Kommandos, die sich darauf beziehen ausgeführt. Daher ergibt sich der Zwang einen entsprechenden Defaultwerte anzugeben.

Solange wir keine „schmutzigen“ Tricks verwenden, haben wir in L<sup>A</sup>T<sub>E</sub>X die Möglichkeit Makros zu definieren, die genau einen optionalen Parameter mit Defaultwert enthalten. Die Verwendung des `\newcommand{}`-Befehls sieht nun so aus:

```
\newcommand{\neuerBefehl}[anzahlParameter] [DefaultWert]{Inhalt}}
```

Als Beispiel wollen wir zunächst ein Kommando `\mybox` erzeugen, das eine Definition mit einem Rahmen umgibt. Dabei soll die Breite der Box optional angegeben werden. Wird keine Breite übergeben, so soll ein Defaultwert von 6cm verwendet werden.

BEISPIEL

```
\newcommand{\mybox}[3] [4cm]{
  \fbox{\parbox{#1}{
    \textbf{\large #2} \\
    \textsl{\footnotesize{#3} }
  }}
}
```

```
\def [Breite]{Titel}{Definition}
```

<p><b>Default</b></p> <p><i>Inhalt</i>, mit <i>Defaultgröße</i>: <i>my-</i>  <code>box{Titel}{Inhalt}</code></p>
--

<p><b>Explizit</b></p> <p><i>Inhalt</i>, mit <i>spezi-</i>  <i>eller Größe</i>: <i>my-</i>  <code>box{3,5cm}{Titel}{Inhalt}</code></p>
--

Bei Versuchen ein Makro mit weniger oder mehr Argumenten als vorgesehen aufzurufen, oder bei der (versuchten) Definition eines Makros mit mehreren optionalen Argumenten, verhielt sich  $\LaTeX$  recht merkwürdig. Nicht immer werden dabei sinnvolle Fehlermeldungen produziert. Zur Ausgabe eigener Fehlermeldungen und Warnhinweise siehe Abschnitt 7. Wenn der auszugebende Inhalt unseres selbst definierten Befehls auch Formeln enthalten soll (man denke an unser Beweismakro), dann sollte im Makro der `\ensuremath`-Befehl verwendet werden.  $\LaTeX$  stellt dann sicher, dass die Formel korrekt dargestellt wird, unabhängig davon, ob unser Makro im Mathematik-Modus (`\$ \dots \$`) oder im normalen Absatzmodus aufgerufen wird. Für komplexere Aufgaben wie unser reichen diese Möglichkeiten jedoch noch nicht aus. Warum sollten wir den Autorennamen optional machen, wenn letztlich doch irgendein Default-Wert angezeigt werden muss? Und wie können wir Autorennamen, Bilder usw. unabhängig voneinander als Argumente ermöglichen? Auf diese Fragen gehen die folgenden Abschnitte ein, aber für viele einfachere Zwecke reicht unser Wissen bereits aus.

## 5.4 Erstellen von eigenen Umgebungen

Um größere Textbereiche – auch mit speziellen, selbst definierten – Befehlen nach festen Regeln zu formatieren, bietet sich als Alternative zur Definition einer Befehls die Definition eigener Umgebungen (Environments) an. Man kann in etwa unterscheiden:

**Kommandos** definieren wir, wenn relativ kurze Textbereiche wiederholt einheitlich formatiert werden sollen, und wenn dieser Text im Allgemeinen nicht durch weitere Befehle verändert werden soll. Insbesondere sollte der Text „logisch“ unstrukturiert sein.

**Umgebungen** definieren wir, wenn größere Textbereiche einheitlich formatiert werden sollen, und diese Textbereiche mit weiteren Befehlen formatiert und strukturiert werden sollen. Insbesondere sind Umgebungen dann notwendig, wenn wir Kommandos definieren wollen, die nur innerhalb des entsprechenden Bereiches gültig sein sollen.

Der Befehl zur Erzeugung einer eigenen Umgebung ist:

```
\newenvironment{UmgebungName}[AnzArgumente]
  {BeginBefehle}
  {EndeBefehle}
```

**UmgebungName** definiert den Namen der neuen Umgebung. Diese Bezeichnung wird dann in `\begin{UmgebungName}` und `\end{UmgebungName}` benutzt.

**AnzArgumente** genau wie bei Makros muss auch hier die Anzahl der Argumente angegeben werden. Innerhalb des Environments kann wie bisher mit `#1` usw. auf diese Argumente zugegriffen werden.

**BeginBefehle** dieser Abschnitt enthält alle Befehle die direkt nach Aufruf des `\begin{UmgebungName}`-Kommandos ausgeführt werden sollen. Insbesondere können hier Label-Deklartiert, Zähler weitergezählt und sogar eigene lokale Anweisungen definiert werden. Die lokalen Anweisungen können nur in der Umgebung verwendet werden, innerhalb der sie definiert wurden.

**EndeBefehle** dieser Abschnitt enthält alle Befehle, die direkt nach Aufruf des `\end{UmgebungName}`-Kommandos ausgeführt werden sollen. Der Sinn kann darin bestehen, dass z.B. im Abschnitt „BeginBefehle“ Formatierungen vorgenommen werden, die nun rückgängig gemacht werden oder z.B. eine Tabelle zu beenden.

Die Formatierung von Merkeinheiten, die wir bisher als Makro definiert hatten, wollen wir nun als Umgebung definieren. Die Gründe dafür liegen auf der Hand, eine Merkeinheit:

- enthält im Allgemeinen viel Text,
- enthält Formeln,
- enthält Bilder und/oder Tabellen,
- wird durch Fomatierungsanweisungen und andere Makros strukturiert
- soll möglicherweise lokale Makros enthalten wie `\qed` oder `\autor`

BEISPIEL

```

\newenvironment{beweisen}[4][\textwidth]
{
  %Begin
  \stepcounter{MECount}
  \label{#1}
  \begin{tabular}{|>{\columncolor{yellow}[5.5pt][5.5pt]}p{#1}|}
\hline
  \large{\textbf{Merkeinheit \theMECount: #2}} \\
  \begin{tabbing}
    xxxxxxxxxxxxxxx\=xxxxxxxxxxx\kill
    Autor: \> #3 \\
    Jahr: \> #4 \\
  \end{tabbing}
}
{
  %Ende
  \\
  \rightline{\Large\textbf{\textit{q.e.d}}}}
  \\
  \hline
  \end{tabular}
}

```

Wie man sieht wird der komplette Inhalt des `beweisen`-Environments in eine Tabelle mit farbigem Hintergrund gepackt.

Von der ursprünglichen Formatierung mittels `shadowbox` und `fcolorbox` wie im 5.2, Seite 10 müssen wir abweichen, da es nicht möglich ist eine öffnende geschweifte Klammer „{“ im Abschnitt „BeginBefehle“ mit einer schließenden geschweiften Klammer „}“ im Abschnitt EndeBefehle zu kombinieren.

Diese Umgebung setzt damit die Verwendung des `colortbl`-Packetes voraus. Angewendet wird unsere neue Umgebung wie folgt:

```

\begin{beweisen}[Breite]{Titel}{Author}{Zeit}
  Beweistext mit beliebigen Kommandos,
  Bildern,
  Tabellen, etc.
\end{beweisen}

```

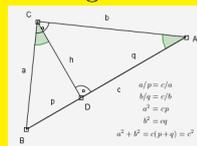
## Merkeinheit 7: Satz von Pythagoras

Autor: Pythagoras  
 Jahr: 500 v. Chr.

Dies ist der Beweistext innerhalb der „beweisen“-Umgebung. Er enthält

- Eine Itemize-Umgebung,
- Weitere *Kommandos*

und sogar Bilder:



und eine Tabelle:

a	b
c	d

*q.e.d*

Um nun Autor, Jahr und das qed optional zu halten wollen wir nun lokale Makros einführen. Lokale Makros werden wie gewohnt mit dem `\newcommand`-Befehl definiert, diesmal jedoch nicht an beliebiger Stelle im Dokument sondern innerhalb des `BeginnBefehle`-Abschnitts unserer Umgebung. Diese Makros sind dann nur innerhalb unserer Umgebung verfügbar.

Wenn wir lokale Makros innerhalb einer Umgebung definieren, müssen wir alle Parameterreferenzen („#“) durch „##“ ersetzen, also verdoppeln. Bei der Analyse des Codes wird der Interpreter alle direkten Referenzen auflösen und alle doppelten # durch einfache ersetzen.

Das folgende Beispiel besteht aus einer Umgebung, die einen Parameter erwartet. Die Umgebung enthält ein lokales Makro `\qedtest` ohne Parameter sowie die beiden Makros `\richtig{}` und `\falsch{}`, die jeweils zwei Parameter erwarten. An diesen beiden Funktionen ist der Effekt der verdoppelten #-Zeichen erkennbar:

```

\newenvironment{test}[1]
{
  %Begin
  \hrule
  \noindent
  \textbf{Beginn der Umgebung "test"\}

```

```

\textit{globaler Parameter: #1 }
\hrule
%
\newcommand{\qedtest}{\rightline{\Large\textbf{\textit{q.e.d}}}}
%
\newcommand{\richtig}[2]
{
  \textbf{Makro "richtig" \\\}
  \textit{Param1: ##1} \\\
  \textit{Param2: ##2 }
}
%
\newcommand{\falsch}[2]
{
  \textbf{Makro "falsch" \\\}
  \textit{Param1: #1} \\\
  \textit{Param2:} #2
}
\noindent
}
{
  %Ende
  \hrule
}

```

Anwendung der Umgebung mit:

BEISPIEL

```

\begin{test}{IchBinGlobal}
  Nun befinden wir uns in der test-Umgebung.\
  Aufruf von richtig{Hallo}{Welt}| \
  \richtig{Hallo}{Welt} \
  Aufruf von falsch{Hallo}{Welt}| \
  \falsch{Hallo}{Welt}
\end{test}

```

Liefert nun folgendes Resultat:

---

**Beginn der Umgebung „test“**

*globaler Parameter: IchBinGlobal*

---

Nun befinden wir uns in der test-Umgebung.

Aufruf von \richtig{Hallo}{Welt}

**Makro „richtig“**

*Param1: Hallo*

*Param2: Welt*

Aufruf von \falsch{Hallo}{Welt}

**Makro „falsch“**

*Param1: IchBinGlobal*

---

*Param2: Welt*

Nach diesem einfachen Beispiel wollen wir uns nun einer etwas komplexeren Aufgabe widmen: Wer im richtigen (oder falschen?) Studiengang studiert, kommt bisweilen in die Verlegenheit UML-Klassendiagramme in eine Mitschrift einfügen zu müssen. Im Folgenden versuchen wir eine Umgebung zu definieren die:

- Einen Klassennamen erwartet sowie
- Durch Verwendung lokaler Kommandos Definitionen für:
  - Methoden und
  - Attribute ermöglicht

```

\newenvironment{UMLClass}[1]
{
  %begin -lokale Kommandos
  \newcommand{\attrib}[2]{
    ##1 & ##2 \\
    \hline
  }
  %
  \newcommand{\method}[3]{
    ##1 & ##2 (##3)\\
    \hline
  }
  %begin - anzeigen
  \begin{tabular}{|c|l|}
  \hline
    \multicolumn{2}{|c|}{\textbf{Klasse: #1}} \\
  \hline
  \hline
  \hline
  \end{tabular}
}
{
  %end
  \hline
  \end{tabular}
}

```

Klasse: myClass1	
-	attribut1
-	attribut2
-	attribut3
+	getAttrib1 (void)
+	setAttrib1 (int)

BEISPIEL

Erzeugt wurde diese Ausgabe mittels:

```
\begin{UMLClass}{myClass1}
  \attrib{-}{attribut1}
  \attrib{-}{attribut2}
  \attrib{-}{attribut3}
  \method+}{getAttrib1}{void}
  \method+}{setAttrib1}{int}
\end{UMLClass}
```

Natürlich bleiben noch viele Fragen ungeklärt:

- Können wir die Reihenfolge sichern? Momentan können Attribute und Methoden beliebig kombiniert werden!
- Wie steht es mit bedingten Anweisungen? Die Formatierung sollte sich ändern, je nachdem ob die Option „Interface“ angegeben wurde oder nicht.
- Variable Anzahl von optionalen Argumenten?

## 5.5 Redefinieren vorhandener Makros

Natürlich ist es nicht nur möglich eigene Kommandos und Umgebungen zu erzeugen, sondern auch vorhandene  $\LaTeX$ -Kommandos können jederzeit verändert werden. Bei einigen Befehlen ist dies problemlos möglich, ja teilweise sogar gewünscht. Beispielsweise müssen wir für eine veränderte Formatierung von Zählern das `\the...`-Kommando redefinieren. Andere Befehle und Umgebungen sind jedoch so komplex, dass eine Redefinition nicht zu empfehlen ist. Einem Beispiel für Redefinition, das innerhalb von  $\LaTeX$  verwendet wird, sind wir bereits begegnet: Innerhalb einer tabbing-Umgebung haben `\>`, `\=`, usw. eine andere Bedeutung als außerhalb, die entsprechenden Kommandos wurden also redefiniert. In diesem Fall wurde aber sichergestellt, dass die Kommandos nach Ende der Umgebung wiederhergestellt werden. Zur Redefinition eines Makros verwenden wir den Befehl: `\renewcommand{\Makro}{Inhalt}` Um ein Befehl zu kopieren (um die ursprüngliche Funktion später wiederherzustellen) kann der `\let`-Befehl von  $\TeX$  verwendet werden: `\let\KopieBefehl\OriginalBefehl`. Auf diese Weise kann auch die Funktionalität eines vorhandenen Kommandos erweitert werden:

- Das ursprüngliche Makro wird kopiert,
- anschließend unter Verwendung neuer Anweisungen sowie der Kopie redefiniert

```
\let\oldEmph\emph
\renewcommand{\emph}[1]{\oldEmph{\textbf{#1}}}
\emph{das neue emph-Kommando} und
\let\emph\oldEmph
\emph{das ursprüngliche emph-Kommando}
```

Zum Vergleich: *das neue emph-Kommando* und *das ursprüngliche emph-Kommando* BEISPIEL

## 6 Listen

Bevor wir uns mit weiteren Details der Makroprogrammierung beschäftigen, wollen wir hier noch auf ein wichtiges, für die Praxis relevantes Thema eingehen: Die Definition eigener Listen.

### 6.1 Verändern von Listen

Standardmäßig haben wir in  $\text{\LaTeX}$  drei Listen zur Verfügung: die Spiegelstrichliste (Itemize), die Aufzählungsliste (Enumerate) und die Definitionsliste (Description). Unter Verwendung der Redefinition von Kommandos, die wir im Abschnitt 5.5, Seite 17 kennengelernt haben, wollen wir diese Listen nun auf unsere speziellen Bedürfnisse anpassen.

#### 6.1.1 Itemize-Liste

In der normalen Itemize-Liste wird vor jedes  $\text{\Item}$  in der ersten Ebene ein Punkt als Aufzählungszeichen gesetzt.  $\text{\LaTeX}$  erlaubt auch die Veränderung dieses Aufzählungszeichens durch Angabe des Optionalen Parameters:  $\text{\item[Aufzählungszeichen]}$ . Dabei kann dieses Aufzählungszeichen nicht nur aus einem Symbol, sondern auch aus mehreren Worten bestehen:

BEISPIEL

```
\begin{itemize}
  \item normales Aufzählungszeichen
  \item auch hier
  \item[+] eher Positiv oder Pro
  \item[-] und Kontra
  \item[vorletzter Punkt] Aufzählungszeichen mit zwei Wörtern
  \item[letzter Punkt] damit endet das Beispiel
\end{itemize}
```

- normales Aufzählungszeichen
- auch hier
- + eher Positiv oder Pro
- und Kontra

vorletzter Punkt Aufzählungszeichen mit zwei Wörtern

letzter Punkt damit endet das Beispiel

Diese Möglichkeit ist jedoch ein wenig unschön, da wir bei jedem item den entsprechenden Parameter angeben müssen. Das einheitliche Schriftbild des Dokumentes ist daher nicht mehr gewährleistet. Die Alternative besteht in der Redefinition des Standardlabels. In  $\text{\LaTeX}$  ist jeder Auflistungsebene ein Standardsymbol zugeordnet:

Stufe	Befehl	Zeichen
1	$\text{\labelitemi}$	•
2	$\text{\labelitemii}$	–
3	$\text{\labelitemiii}$	*
4	$\text{\labelitemiv}$	·

Die Redefinition erfolgt wie bereits bekannt mit dem `\renewcommand{}`-Befehl:

```
\renewcommand{\labelitemi}{$\longrightarrow$}
\renewcommand{\labelitemii}{\ding{227}}
```

BEISPIEL

- erste Ebene
- auch noch erste Ebene
  - zweite Ebene
  - mit neuem Symbol
- wieder auf erster Ebene

### 6.1.2 Enumerate-Liste

Auch die Enumerate-Liste kennt verschiedene Ebenen. Intern wird mit jedem Aufruf von `\item` ein Zähler für die entsprechende Ebene hochgezählt und dann mit dem `\the...`-Kommando angezeigt. Dabei ist das angezeigte Format abhängig von der Ebene. Die Änderung wird auch hier mit `renewcommand` durchgeführt, allerdings wird diesmal nicht einfach nur ein neues Zeichen definiert, sondern ein Makro, das den Zählerstand anzeigt (oder auch verändert):

Stufe1	<code>\labelenumi</code>	arabic 1.
Stufe2	<code>\labelenumii</code>	alph (a)
Stufe3	<code>\labelenumiii</code>	roman i
Stufe4	<code>\labelenumiv</code>	Alph A

```
\renewcommand{\labelenumi}{Teil -- \theenumi:}
\renewcommand{\labelenumii}{Abschnitt -- \roman{enumii}}
```

BEISPIEL

- Teil – 1: Auf höchster Ebene
- Teil – 2: Mit verändertem Zähler
- Abschnitt – i: Andere Formatierung auf der 2. Ebene
- Abschnitt – ii: In römischer Nummerierung
- Teil – 3: Wieder auf der 1. Ebene

Ein weiteres, etwas komplexeres, Beispiel (übernommen aus [1]): Die Nummerierung der 1. Ebene soll eingerahmt erfolgen, und aus der römischen Zahl für den aktuellen Abschnitt (Section) sowie arabischen Ziffern für das aktuelle Item bestehen. Die Nummerierung der 2. Ebene soll durch spezielle Symbole erfolgen, die abhängig vom Zählerstand verändert werden.

```
\renewcommand{\labelenumi}
{
  \fbox{\Roman{section}.\theenumi}}
}
```

BEISPIEL

```

\newcounter{zaehler}
\renewcommand{\labelenumii}
{
  \setcounter{zaehler}{\value{enumii}}
  \addtocounter{zaehler}{181}
  \ding{\value{zaehler}}
}

```

VI.1 1. Ebene, ab hier wird der Zähler umrahmt und

VI.2 besteht aus „section“ (Römisch) sowie „Item“-Nummer (arabisch)

- ❶ Auf der 2. Ebene
- ❷ werden Symbole aus
- ❸ dem „ding“-Package verwendet

VI.3 Wieder auf der 1. Ebene

### 6.1.3 Description-Liste

Die Description-Liste wird im Prinzip genauso wie Enumerate- und Itemize-Liste verwendet, nur muss das anzuzeigende Label zusätzlich angegeben werden (obwohl es in eckigen-Klammern steht). In der Standardeinstellung wird L<sup>A</sup>T<sub>E</sub>X die Label in fetter Schrift linksbündig untereinander schreiben.

Diese Einstellungen können wir durch Redefinition von `\descriptionlabel` verändern. Dabei müssen wir natürlich einen Parameter angeben, da die Definitionsliste ohne Angabe des zu beschreibenden Begriffes ja auch sinnlos wäre.

**normal** sind Einträge in fetter Schrift

**so wie** hier.

```
\renewcommand{\descriptionlabel}[1]{\textbf{\emph{\textsf{#1}}}}
```

BEISPIEL

**besser** und schöner sieht die neue

**Liste** mit schräger Schrift aus.

## 6.2 Neue Listen

Abgesehen von der Redefinition der drei Standardlisten können wir auch mit der generischen Umgebung „list“ arbeiten. Eine solche allgemeine Liste wird folgendermaßen verwendet:

```

\begin{list}{DefaultMarke}{LayoutEinstellungen}
  \item[Marke] Text
  \item Text
\end{list}

```

Eine solche Liste besteht entweder aus der DefaultMarke oder der optional angegebenen Marke und dem zugehörigen Text. Durch Redefinition des `\makeLabel`-Kommandos kann die Darstellung der Marken definiert werden. Als Layout-Einstellungen (Abstände und Größen) können folgende Parameter verändert werden:

<code>\parsep</code>	Abstand zwischen den Absätzen innerhalb eines Listeneintrages
<code>\itemsep</code>	Abstand zwischen zwei Listeneinträgen
<code>\leftmargin</code>	Abstand zwischen dem Textrand auf der linken Seite und dem Beginn des Listeneintrages
<code>\rightmargin</code>	Abstand zwischen dem Textrand auf der rechten Seite und der Begrenzung der Listeneinträge
<code>\labelsep</code>	Abstand zwischen der Marke und dem Beginn des Listeneintrages
<code>\labelwidth</code>	Breite der Box für die Marke
<code>\usecounter</code>	Angabe welcher Zähler automatisch erhöht werden soll, sobald <code>\item</code> aufgerufen wird.

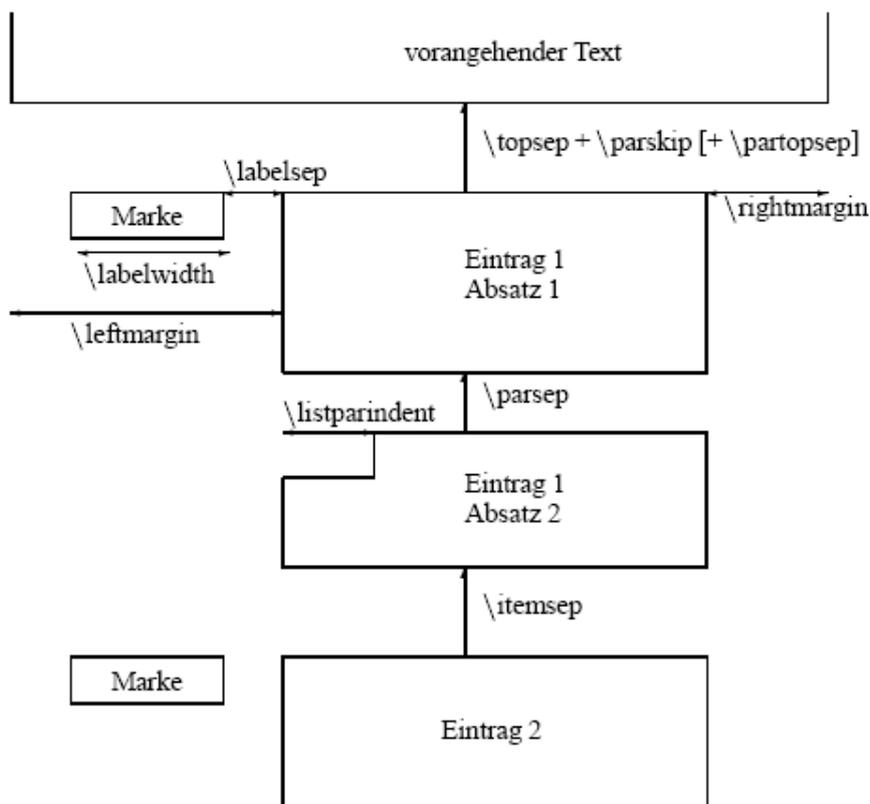


Abbildung 1: Parameter zur Beeinflussung von Listen

```

\begin{list}{\ding{42}}
{
  \setlength{\topsep}{0.5cm}
  \setlength{\itemsep}{0.5cm}
  \setlength{\leftmargin}{5cm}
  \setlength{\labelwidth}{3cm}
  \setlength{\labelsep}{1cm}
  \renewcommand{\makelabel}[1]{\textbf{\textsf{\large #1}}}
}
\item[Begriff1] Hier ist die Definition des 1. Begriffes
\item[Begriff2] Hier ist die Definition des 2. Begriffes
\item[Begriff3] Hier ist die Definition des 3. Begriffes
\item Hier eine Eintrag ohne eigene Marke,
\item in diesem Fall wird die Standardmarke verwendet

\end{list}

```

BEISPIEL

- |                 |   |
|-----------------|---|
| <b>Begriff1</b> | Hier ist die Definition des 1. Begriffes        |
| <b>Begriff2</b> | Hier ist die Definition des 2. Begriffes        |
| <b>Begriff3</b> | Hier ist die Definition des 3. Begriffes        |
| ☛               | Hier eine Eintrag ohne eigene Marke,            |
| ☛               | in diesem Fall wird die Standardmarke verwendet |

Die Verwendung eines Counters in einer Liste im folgenden Beispiel gezeigt:

```

\newcounter{strophe}
\begin{list}{\thestrophe. Strophe}
{
  \usecounter{strophe}
  \setlength{\labelwidth}{2cm}
  \setlength{\leftmargin}{4cm}
  \setlength{\labelsep}{1cm}
  \renewcommand{\makelabel}[1]{\textsf{\Large #1}}
}
\item abcdefg
\item hijklmn
\item opqrstu
\item vwxyzab
\end{list}

```

BEISPIEL

1. **Vers**      Wer routet so spät durch Nacht und Wind?  
Es ist der Router, er routet geschwind!
2. **Vers**      Bald routet er hier, bald routet er dort  
Jedoch die Pakete, sie kommen nicht fort.
3. **Vers**      Sie sammeln und drängeln sich, warten recht lange  
in einer zu niedrig priorisierten Schlange.
4. **Vers**      Die Schlangen sind voll, der Router im Streß,  
da meldet sich vorlaut der Routingprozeß
5. **Vers**      und ruft: „All Ihr Päckchen, Ihr sorgt Euch zu viel,  
nicht der IP-Host, nein, der Weg ist das Ziel!“

## 7 Kommunikation mit L<sup>A</sup>T<sub>E</sub>X, Ein- und Ausgabe

Wenn wir Makros für L<sup>A</sup>T<sub>E</sub>X schreiben oder später gar mit Kontrollstrukturen programmieren, ist es ganz nützlich (und eventuell auch guter Stil), wenn wir Informationen in der Statusanzeige des Compilers ausgeben oder abfragen können. Wir könnten z.B. bei jedem Aufruf eines durch uns definierten Makros das Datum, den Autor sowie die benötigten Packages ausgeben lassen. Der Befehl für Ausgaben lautet:

```
\typeout{Text der Angezeigt wird}
```

Eine interessante Verwendung könnte auch darin bestehen die Anzahl der Aufrufe eines Makros auszugeben. Wir könnten beispielsweise einen Zähler „UML-Count“ erzeugen, der bei jedem Aufruf unseres `\umlclass`-Makros erhöht wird. In der Statusanzeige könnte unser Makro dann den jeweiligen Zählerstand anzeigen, selbst wenn diese Information im ausdrückbaren Dokument nirgends angezeigt werden soll.

L<sup>A</sup>T<sub>E</sub>X erlaubt sogar die Eingabe von Werten über die Kommandozeile. Der Befehl für Eingaben lautet:

```
\typein[\Variable]{Dieser Text wird angezeigt}
```

L<sup>A</sup>T<sub>E</sub>X hält nun den Kompilerlauf an, gibt den angegebenen Text aus und erwartet eine Eingabe. Der eingegebene Text wird in der Variablen gespeichert, und kann mit dem Befehl `\Variable` angezeigt bzw. verwendet werden.

Zum Beispiel könnten auf diese Weise Textabschnitte bei jedem L<sup>A</sup>T<sub>E</sub>X-Lauf dynamisch eingefügt werden. Ein sinnvolle Anwendung kann auch darin bestehen das Einfügen externer Dateien von Benutzerangaben abhängig zu machen:

```
\typein[\File]{Welche Datei soll eingebunden werden?}  
\includeonly{\File}
```

BEISPIEL

## 8 Kontrollstrukturen in L<sup>A</sup>T<sub>E</sub>X bzw. T<sub>E</sub>X

Um wirklich mächtige Makros und Erweiterungen zu L<sup>A</sup>T<sub>E</sub>X zu schreiben, ist es teilweise notwendig, plain-T<sub>E</sub>X zu verstehen und die Arbeitsweise von T<sub>E</sub>X zu kennen. Es lassen sich schließlich nur dann Erweiterungen zu L<sup>A</sup>T<sub>E</sub>X – die über den vorhandenen Funktionsumfang hinausgehen – schreiben, wenn man direkt auf T<sub>E</sub>X-Befehle zugreift. Ansonsten ist man beim Erstellen eigener Makros auf die ohnehin vorhandenen Funktionen von L<sup>A</sup>T<sub>E</sub>X und entsprechender Pakete beschränkt.

T<sub>E</sub>X (und einige Pakete von L<sup>A</sup>T<sub>E</sub>X) bieten uns tatsächlich viele Möglichkeiten an, die wir auch aus anderen Programmiersprachen kennen:

- Variablen (Register)
- Operationen
- Kontrollstrukturen (Schleifen, bedingte Anweisungen)
- Funktionen und Prozeduren (Makros)

Mit diesen Werkzeugen sind wir in der Lage, komplexe Makros zu erstellen, die auch auf komplexe Sachverhalte reagieren können. Es würde den Rahmen der vorliegenden Arbeit sprengen, detailliert auf die Möglichkeiten der Programmierung in T<sub>E</sub>X einzugehen (Hierfür sei auf [4] und [5] verwiesen), im folgenden wird nur ein kurzer Überblick über die Möglichkeiten gegeben. Der Leser sollte sich vor Augen halten, dass komplexe Pakete wie BEAMER auf diese Weise erstellt werden können.

### 8.1 Variablen

Zum Arbeiten mit Werten bietet T<sub>E</sub>X eine Reihe von Speicherbereichen für Werte von unterschiedlichen Typen an. Diese Speicherbereiche werden Register genannt und ähneln im Prinzip Arrays. Eine (kleine) Auswahl der vorhandenen Speicherbereiche:

<code>\count</code>	Integerwerte (number) in ] – 231, 231[
<code>\dimen</code>	Dimensionswerte (pt, pc, cm, ...; siehe unten)
<code>\skip</code>	Für glue-Werte

Jeder dieser Speicherbereiche ist aufgeteilt in 256 Zellen (Speicherplätzen)<sup>8</sup>. Einem solchen Speicherplatz kann für die weitere Verwendung ein Bezeichner (Variable) zugewiesen werden:

Befehl	Beispiel
<code>\countdef\Name = zelle</code>	<code>\countdef\X = 255</code>
<code>\dimendef\Name = zelle</code>	<code>\dimendef\Y = 255</code>
<code>\skipdef\Name = zelle</code>	<code>\skipdef\Z = 255</code>

Die so definierten Variablen können dann für Wertzuweisungen oder zur Ausgabe genutzt werden:

<sup>8</sup>nach [5] bezeichnet Knuth diese Beschränkung als den größten Fehler, der bei der Entwicklung von T<sub>E</sub>X gemacht wurde.

BEISPIEL

```
\countdef\X = 255
\X = 0815
\the\X
```

Allerdings werden viele der vorhandenen Zellen bereits von TeX verwendet. Nur die Zelle 255 ist im Allgemeinen frei verfügbar. Damit es zu keinen Kollisionen kommt, sollten neue Variablen nicht durch Angabe einer Zelle, sondern mit Hilfe des `\new...` Befehls deklariert werden:

```
\newcount\X
\newdimen\Y
```

## 8.2 Operationen

Arithmetische Operationen lassen sich mit folgenden Befehlen durchführen:

Befehl	Beispiel
<code>\advance\Name by Wert</code>	<code>\Advance\X by -42</code>
<code>\multiply\Name by Wert</code>	<code>\multiply\Y = 255</code>
<code>\divide\Name by Wert</code>	<code>\divide\Z by 2</code>

Der Advance-Befehl kann mit unterschiedlichen Datentypen (glue, number, etc.) arbeiten, die Multiplikation und die Division funktionieren dabei jedoch nur mit Integer-Werten. Bei der Division handelt es sich also um eine Ganzzahlendivision. Die folgenden Beispiele zeigen die Verwendung dieser Kommandos:

```
X = 20
Y = 30
Y = Y + X = 50
Y = Y * 2 = 100
X = X / Y = 0
```

BEISPIEL

Die nötige Befehlsfolge für diese Rechnung ist:

```
\newcount\X
\newcount\Y
\X = 20
\Y = 30

X = 20 \\
Y = 30 \\

\advance\Y by \X
Y = Y + X = \the\Y \\

\multiply\Y by 2
Y = Y * 2 = \the\Y \\

\divide\X by \Y
X = X / Y = \the\X
```

Eine Möglichkeit, um dennoch eine Fließkommadivision durchzuführen, besteht in der Verwendung von Dimen-Zellen (Variablen). In Wahrheit wird auch dort eine Ganzzahldivision durchgeführt, alle Werte werden jedoch zuvor in „scaled Points“ (sp) umgewandelt, das Ergebnis wird jedoch wieder in Points (pt) umgewandelt und ausgegeben:

```
Textwidth = \the\textwidth \\\
```

BEISPIEL

```
\newdimen\A
\A = 60
\divide\A by 7
A = 60 / 7 = \the\A \\\
```

```
Textwidth = 345.0pt
A = 60 / 7 = 8.57143pt
```

Das Rechnen in  $\TeX$  ist aus mehreren Gründen nicht wirklich komfortabel, zum einen werden alle Befehle in einer Prefixnotation angegeben, anstelle simpler Operator-Symbole müssen komplette Befehle verwendet werden, und die Beschränkung auf Ganzzahlenarithmetik ist sehr störend. In  $\LaTeX$  sind aus diesen Gründen verschiedene Pakete verfügbar, mit deren Hilfe auch komplexe arithmetische Ausdrücke berechnet werden können.

Die Verwendung des „calc“-Paketes erlaubt unter anderem binäre Operatoren:

```
\newcounter{test}
\setcounter{test}{10}
\setcounter{test}{(\value{test} + 1) * \value{test} / 2 }
Ergebnis: (10 + 1) * (10 / 5) = \thetest
```

BEISPIEL

```
Ergebnis: (10 + 1) * (10/5) = 55
```

Das Paket „fp“ ermöglicht auch das Rechnen mit Dezimalbrüchen:

```
\FPset\x{60.3}
\FPset\y{7.5}
\FPset\erg{0.0}
\FPdiv\erg\x\y
Ergebnis: 60.3 / 7.5 = \the\erg \\\
```

BEISPIEL

```
Ergebnis: 60.3/7.5 = 8.04000000000
```

## 8.3 Kontrollstrukturen

### 8.3.1 Bedingte Anweisungen

Wie bereits erwähnt ist eine unabdingbare Voraussetzung für die Definition komplexer Makros die Möglichkeit bestimmte Anweisungen in Abhängigkeit von Bedingungen auszuführen.  $\TeX$  stellt unter anderem folgende Befehle zur Verfügung:

Befehl	Beispiel
<code>\ifnum Int1 Operator Int2</code>	<code>\ifnum\X=\Y</code>
<code>\ifdim Dim1 Operator Dim2</code>	<code>\ifdim\textwidth&gt;5cm</code>
<code>\ifodd Int</code>	<code>\ifodd 3</code>

Der Syntax zur Verwendung der verschiedenen If-Anweisungen lautet:

```
\ifnum Int1 Operator Int2 {Anweisungen1} \else {Anweisungen2} \fi
```

BEISPIEL

```
\newcount\A
\newcount\B
\A=20
\B=\A
\ifnum\A=\B {A (\the\A) gleich B (\the\B)}
           \else {A (\the\A) ungleich B (\the\B\ ) } \fi
```

A (20) gleich B (20)

Auch für diese bedingten Anweisungen kennt  $\LaTeX$  ein analoges Paket um uns die Arbeit zu erleichtern: das Package „ifthen“ das unter anderem folgende Makros zur Verfügung stellt:

Befehl	Erklärung
<code>\ifthenelse {Bedingung}{Dann}{Sonst}</code>	<code>\ifnum\X=\Y</code>
<code>\newboolean Name</code>	Erzeugen einer Boole'schen Variablen
<code>\setboolean Name Wert</code>	Zuweisung an eine Boole'sche Variable
<code>\and, \or, \not</code>	Verknüpfen logischer Ausdrücke
<code>\equal String1 String2</code>	Überprüft zwei Strings auf Gleichheit
<code>\isodd Zahl</code>	Überprüfen ob eine Zahl ungerade ist
<code>\lengthtest Dimension</code>	Vergleichen von Maßen
<code>\(, \)</code>	Klammerung von logischen Ausdrücken

Als Beispiel soll ein Makro aus [5] gezeigt werden, das die aktuelle Zeit des Übersetzungslaufes ausgibt:

```
\newcommand{\printtime}
{
  \newcounter{std}
  \newcounter{min}
  \setcounter{std}{ \time / 60 }
  \setcounter{min}{ \time - \value{std}*60 }
```

```

\ifthenelse{\value{std}<10}{0}{}\thestd:
\ifthenelse{\value{min}<10}{0}{}\themin
}

```

Die aktuelle Zeit ist `\printtime`

Die aktuelle Zeit ist 18:37

### 8.3.2 Schleifen

In  $\text{T}_{\text{E}}\text{X}$  werden Schleifen mit Hilfe des `\loop \dots \repeat` Konstruktes realisiert. Dabei wird dem `\repeat` ein `|\if|` vorangestellt, das die Abbruchbedingung überprüft. Auch die folgenden Beispiele wurden aus [5] übernommen:

BEISPIEL

```

\newcount\i
\i=1

\loop
  \the\i,
  \advance\i by 1
\ifnum \i<11 \repeat

```

Zählen von 1 bis 10:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

Ein weiteres Beispiel, diesmal mit Rechnung:

BEISPIEL

```

\newcount\val
\newcount\step
\val=1
\step=2

\loop
  \ifodd\val
    \else{\the\val, }\fi
  \advance\val by \step
  \ifnum\step<101\advance\step by 1
\repeat

```

6, 10, 28, 36, 66, 78, 120, 136, 190, 210, 276, 300, 378, 406, 496, 528, 630, 666, 780, 820, 946, 990, 1128, 1176, 1326, 1378, 1540, 1596, 1770, 1830, 2016, 2080, 2278, 2346, 2556, 2628, 2850, 2926, 3160, 3240, 3486, 3570, 3828, 3916, 4186, 4278, 4560, 4656, 4950, 5050,

Auch hier bietet uns L<sup>A</sup>T<sub>E</sub>X wieder ein einfacheres Makros an:

```
\whiledo {Bedingung}{Körper}
```

Es ermöglicht uns die einfache Definition von kopfgesteuerten While-Schleifen.

Das folgende abschließende, komplexe Beispiel stammt von Markus Rahn [7] und kombiniert die Möglichkeiten, eigene Makros zu definieren mit While-Schleifen. Die Makros erlauben es, eine beliebige Anzahl von Fibonacci-Zahlen zu berechnen:

BEISPIEL

```
\newcounter{fiba}
\newcounter{fibb}
\newcounter{fibc}
\newcounter{fibrun}

\newcommand{\init}{
  \setcounter{fiba}{1}
  \setcounter{fibb}{1}
  \setcounter{fibrun}{0}
}

% fib{k} gibt die k-te Fibonaccizahl F_k
\newcommand{\fib}[1]{
  \init
  \whiledo{\thefibrun < #1}{ \stepnext \stepcounter{fibrun} }
  \thefiba
}

\newcommand{\stepnext}{
  \add
  \rotate
}

\newcommand{\add}{
  \setcounter{fibc}{\thefiba}
  \addtocounter{fibc}{\thefibb}
}

\newcommand{\rotate}{
  \setcounter{fiba}{\thefibb}
  \setcounter{fibb}{\thefibc}
}

\newcounter{i}
\newcounter{en}

\setcounter{en}{20}
Die ersten \theen\ Fibonacci-Zahlen lauten:\\
\whiledo{\thei < \theen}{\fib{\thei}$ \stepcounter{i}}
```

Die ersten 20 Fibonacci-Zahlen lauten:

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

## 9 Fazit

In der vorliegenden Arbeit konnte gezeigt werden, wie sich mithilfe der  $\LaTeX$  eigenen Möglichkeiten neue, auf die individuellen Anforderungen abgestimmte Makros erzeugen lassen. In den Abschnitten 4 bis 5.5 wurde erläutert, wie parametrisierbare Befehle und Umgebungen erstellt werden können. Im Abschnitt 6.1 wurde darauf aufbauend erläutert, wie die Standardformatierung für Listenelemente verändert und optimiert werden kann. Abschließend konnte anhand der angebotenen Kontrollstrukturen gezeigt werden, dass  $\LaTeX$  in Verbindung mit einigen Paketen alle Merkmale einer vollwertigen Programmiersprache erfüllt und damit über die gezeigten Anwendungen hinaus frei anpassbar ist. Wie diese Arbeit dokumentiert, kann das Ziel von  $\LaTeX$  – die Konzentration des Autors auf den Text anstelle des Designs – nur eingeschränkt erreicht werden:

1. Entweder der Autor beschränkt sich auf das Schreiben eines reinen Fließtextes, der nur wenige Formatierungsanweisungen enthält und dessen Struktur und Format durch einige wenige  $\LaTeX$ -Befehle beschrieben werden können,
2. oder der Autor kommt nicht umhin, komplexe Befehlsstrukturen zu erlernen und möglicherweise durch individuelle Makros zu ergänzen.

Insgesamt ist  $\LaTeX$  zum Erstellen komplexer Dokumente für den durchschnittlichen PC-User ungeeignet. Auch nach dem Erlernen der häufigsten benötigten Befehle ist die Produktivität des Autors mit  $\LaTeX$  im Vergleich zu einem WYSIWYG-Programm eher gering.

Als Analogie mag hier der Bereich des Web-Designs herangezogen werden: „*Jeder Web-Designer, der etwas auf sich hält kann HTML verstehen, doch niemand auf der Welt käme auf den Gedanken eine Internetpräsenz ohne Dreamweaver, GoLive oder Frontpage zu erstellen*“

Für Dokumente mit gleich bleibendem Design und ähnlicher Struktur kann  $\LaTeX$  dennoch produktiv eingesetzt werden, falls entsprechende Makro-Pakete oder Klassen aufgebaut werden. Unter diesen Umständen kann  $\LaTeX$  für das Einhalten der Corporate-Identity in Dokumentations-, Präsentation- und Schulungsunterlagen eingesetzt werden.

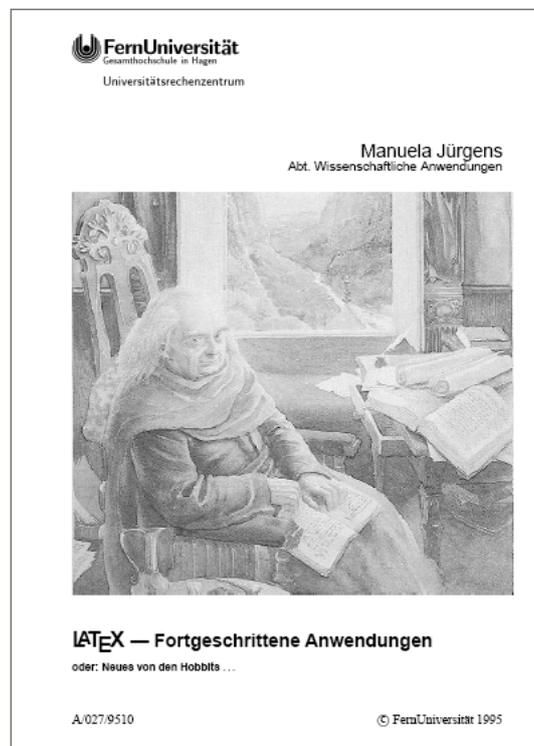
Das hierfür notwendige know-how grenzt den möglichen Nutzerkreis von  $\LaTeX$  jedoch stark ein, was durch die Existenz als „Nischen-Lösung“ im wissenschaftlichen und technischen Bereich bestätigt wird.

Abhilfe könnten hier langfristig möglicherweise Hybridlösungen wie LyX schaffen, die den  $\LaTeX$ -Hintergrund mit einer WYSIWYG-Oberfläche verbinden.

## 10 Quellen

### Literatur

- [1]  $\LaTeX$  – Fortgeschrittene Anwendungen, Manuela Jürgens, Fernuniversität Hagen
- [2] Makroprogrammierung mit  $\TeX$  (Anwenderkreis Dortmund)
- [3]  $\LaTeX 2\epsilon$  for Class and Package Writers (CLS-guide)
- [4] The TeXbook, Donald E. Knuth, Addison-Wesley
- [5] Rechnen und Programmieren in  $\TeX$ , M. Biedermann, Uni Koblenz-Landau
- [6] UK List of  $\TeX$ -Frequently Asked Questions, (<http://www.tex.ac.uk/cgi-bin/texfaq2html>)
- [7] 433 Beispiele in 132 Programmiersprachen, Markus Rahn, (<http://www.ntecs.de/old-hp/uu9r/lang/html/latex.de.html>)



Hiermit versichere ich, die vorliegende Seminararbeit selbständig und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden.

Gießen, den 30. Juni 2005,

Jan-Erek Thiede