Dec@ding style files – Übungen

Andres Löh
Universiteit Utrecht
andres@cs.uu.nl

8. November 2002

1 Kommentare (wichtig)

Bestimmt ist Dir bereits aufgefallen, daß viele Zeilenenden in den Codebeispielen mit einem Prozentzeichen versehen sind, obwohl dann gar kein Kommentar folgt. Der Zweck dieser Übung ist, zu erkennen, daß dies eine gute Praxis ist und tatsächlich notwendig, um die gewünschten Ergebnisse zu erzielen.

Versuche folgende Makrodefinition (genau so abtippen!):

```
\newcommand{\ProzentTestOhne}{
  Was nun?}
\newcommand{\ProzentTestMit}{%
  Was nun?}
```

Probiere das nun an folgendem Beispiel aus:

```
\par
\rule{1pt}{\lineheight} \ProzentTestOhne
\par
\rule{1pt}{\lineheight} \ProzentTestMit
```

Was kannst Du erkennen?

Der Effekt liegt in der Art und Weise begründet, in der TEX die Eingabe verarbeitet. Normalerweise ist es günstig, das Zeilenende als Leerraum zu betrachten. Wenn man einen fortlaufenden Text schreibt, dann ist das genau das gewünschte Verhalten.

Bei der Programmierung können ungewollte Leerräume jedoch zu sehr schwer diagnostizierbaren Fehlern führen, daher kann man das Einfügen des Leerraums durch das Kommentieren von Zeilenenden verhindern. Dazu muß das Kommentarzeichen *direkt* hinter dem Code der Zeile stehen, ohne weitere Leerzeichen dazwischen! Du solltest Dir daher angewöhnen, bei eigenen Makrodefinitionen *grundsätzlich* die Zeileenden auszukommentieren. (Alternativ kannst Du Dir im TEXbook die genauen Regeln ansehen, wann dann doch kein Leerzeichen eingefügt wird.)

2 Die Wahrheit über `@` (informativ)

Im Vortrag wurde gesagt, daß Kommandos mit einem `@` im Namen nur in Style-Files verwendet werden können. Das ist nicht die ganze Wahrheit! Ein Kommando, welches im LaTeX-Kernel definiert wird, ist `\@gobble`.

Versuche doch mal,

```
\par
\@gobble{Hello world}
```

einzugeben. Natürlich bekommt man eine Fehlermeldung, nur leider eine, die alles andere als deutlich auf die Natur des Fehlers hinweist. Es ist also Teil dieser Übung, sich die Fehlermeldung genau anzusehen. Wenn Du eine ähnliche Meldung später einmal wieder bekommst, so liegt das ziemlich sicher daran, daß Du ein Kommando mit `@` versucht hast zu verwenden (oder aus Versehen ein `@` in einen Kommandonamen eingefügt hast).

LaTeX bietet nun aber zwei Befehle, mit denen man vorübergehend so tun kann, als sei man in einem Style-File. Code, der zwischen den beiden Befehlen steht, darf Kommandos mit `@` enthalten!

Versuche jetzt

```
\makeatletter
\par
\@gobble{Hello world}
\makeatother
```

und staune darüber, daß es jetzt funktioniert.

Lies die Befehle als *make "at" letter* und *make "at" other*, wobei "at" für das Zeichen `@` steht. Die Namen beschreiben die Wirkungsweise des Tricks. Mit `\makeatletter` wird das Zeichen `@` gegenüber TeX vorübergehend als Buchstabe ausgewiesen, und Buchstaben dürfen in Kommandonamen vorkommen! Mit `\makeatother` wird diese Änderung wieder rückgängig gemacht, und `@` ist daraufhin wieder ein „anderes" Zeichen, welches nicht in Befehlsnamen vorkommen darf.

Nun läßt sich auch die Fehlermeldung aufklären. TeX erlaubt symbolische Befehlsnamen genau dann, wenn sie aus einem Zeichen bestehen. Da also `@` in `\@gobble` kein Buchstabe ist, denkt TeX, der Befehl sei nach `\@` bereits zu Ende. Diesen Befehl gibt es auch tatsächlich. Er ist definiert als

```
\def\@{\spacefactor\@m}
```

Der primitive TeX-Befehl `\spacefactor` ist nun ungünstigerweise in dieser Situation nicht zulässig und erzeugt daher den Fehler, den Du vorhin gesehen hast.

3 LaTeX-Kernel (harmlos)

Versuche, die Quellen des LaTeX-Kernels in Deiner Linux-Distribution zu finden. Zu erwarten wäre, daß sie innerhalb des TeX-Baums etwa unter `/temxf/source/latex/base`

zu finden sind. Es ist möglich, daß manche Distributionen die Quellen nicht oder nur wahlweise installieren. In letzterem Fall solltest Du beim nächsten Update die Quellen mal mitinstallieren und die Übung nachholen.

In diesem Verzeichnis gibt es einen Haufen von `.dtx`-Dateien und im Idealfall eine Datei mit dem Namen `source2e.tex`. Schau nach, ob irgendwo auf dem Computer bereits eine übersetzte Fassung dieses Dokumentes herumliegt, suche also nach `source2e.dvi` oder `source2e.ps` oder gar `source2e.pdf`.

Falls nicht, mußt Du selbst die Dokumentation kompilieren. Da Du auf das Verzeichnis eventuell keine Schreibberechtigung hast, kopiere alle Dateien aus dem LaTeX-Quellverzeichnis in ein Unterverzeichnis Deines Arbeitsverzeichnisses und lasse dann LaTeX auf `source2e` los.

Auf einem Unix-System also ungefähr so:

```
$ locate source2e
/usr/texmf/source/latex/base/source2e.tex
# (die Quelle existiert, die kompilierte Fassung nicht)
$ cd # (ins Home-Verzeichnis wechseln)
$ mkdir LaTeX-Kernel
$ cd LaTeX-Kernel
$ cp /usr/texmf/source/latex/base/* .                    (her damit!)
$ latex source2e                                    (LaTeX aufrufen)
```

Schaue Dir dann das fertige `source2e` mit einem Previewer an und versuche, darin die Definitionen von `\@gobble` und `\newcommand` wiederzufinden. Die wirklichen Codezeilen sind diejenigen, die numeriert sind – alles andere ist Kommentar!

Suche auch einige andere Befehle, die Du bereits kennst (zum Beispiel `\section` oder `\emph`.

Versuche auch, die Definition der `itemize`-Umgebung zu finden. Denke daran, daß Du im Vortrag gelernt hast, daß Umgebungen nicht notwendigerweise mit den Befehlen `\newenvironment` oder `\renewenvironment` definiert werden müssen.

4 Debugging mit `\show` und `\showthe` (nützlich)

Im Vortrag wurde erwähnt, daß man mit Hilfe des primitiven TeX-Befehls `\show` die Definition eines Makros anzeigen lassen kann. Während des Aufrufs von TeX wird dann die Abarbeitung unterbrochen und das Ergebnis des `\show`-Befehls in der Konsole angezeigt. Falls Du eine Shell benutzt, um TeX zu starten, dann kann es sein, daß es schwierig ist, die Ausgaben von TeX direkt zu begutachten.

Wir diskutieren hier daher die interaktive Variante. Öffne eine Kommandozeile und tippe `latex` ein, etwa so:

```
$ latex
This is TeX, Version 3.14159 (Web2C 7.3.3.1)
**
```

Der doppelte Stern signalisiert, daß Du einen Dateinamen eingeben kannst, der bearbeitet werden soll, aber Du kannst auch einfach mit einem Befehl loslegen.

Weil nach dem ersten Befehl noch zusätzliche Ausgaben kommen, beginnen wir mit etwas, das keine Auswirkungen hat.

```
** \relax
LaTeX2e <2000/06/01>
Babel <v3.7h> and hyphenation patterns for english, dumylang, nohyphenation,
german, ngerman, ukenglish, greek, latin, dutch, loaded.
*
```

Der einzelne Stern ist TEXs normaler Kommandoprompt. Man kann hier beliebige TEX-Befehle eingeben, im Prinzip also auch ein gesamtes LATEX-Dokument, Zeile für Zeile. Das ist natürlich ziemlich unkomfortabel, aber für ein bißchen „Frage und Antwort“ ist es sehr gut geeignet.

Überprüfen wir die Erkenntnisse, die wir in der letzten Übung gesammelt haben. Versuche

```
* \makeatletter                                  (wir wollen auch Befehle mit @ eingeben!)
* \show\@gobble
> \@gobble=\long macro
#1->.
<*> \show\@gobble

?
```

Dies muß man nun übersetzen: `\@gobble` ist ein Makro, welches mit `\long\def` definiert wurde (dazu später noch mehr). Es hat ein Argument (links vom Pfeil `->`). Rechts vom Pfeil steht die Ersetzung, beendet durch einen `.`, der selbst nicht zur Ersetzung gehört. Folglich ist der Ersetzungstext von `\@gobble` leer.

Die Zeile danach sagt aus, daß TEXbei der Abarbeitung des Befehls `\show\@gobble` unterbrochen wurde, das `?` wartet auf eine Bestätigung. Mit der Return-Taste kommst Du zum Stern=Prompt zurück.

Suche Dir nun Befehle aus, die Du gesehen hast, die ich aber Deiner Meinung nach nicht ausreichend erklärt habe. Hilft Dir die Erklärung weiter? Was passiert, wenn Du ein primitives Kommando eingibst, welches gezeigt werden soll, etwa

```
* \show\def
```

TEX kennt außerdem den Befehl `\showthe`, mit dem Registerinhalte ausgegeben werden. Versuche, in einem Dokument irgendwo im Text die Zeile

```
\showthe\linewidth
```

unterzubringen, und Du kannst in der Ausgabe nachlesen, wie groß diese Länge zu diesem Zeitpunkt ist. Dies kann sinnvoll sein, wenn irgendwelche Abstände im Dokument anders ausfallen als man das erwartet . . .

5 Scannen von Environments contra `\newenvironment` (herausfordernd)

Mit `\newenvironment` kann man neue Umgebungen definieren. Man könnte also etwa die folgende Umgebung wollen:

```
\newenvironment{mytx}{%
  \begin{center}
  \small
  \begin{tabularx}}{%
  \end{tabularx}
  \end{center}}
```

Probiere das aus und verwende die Umgebung. Das sollte eine Fehlermeldung geben. Kannst Du Dir vorstellen warum?

Modifiziere nun die Umgebung zu folgender:

```
\newenvironment{mytx}{%
  \center
  \small
  \tabularx}{%
  \endtabularx
  \endcenter}
```

Nun sollte es funktionieren. Versuche abermals, eine Erklärung für dieses Verhalten zu finden. (Hinweis: Es hat etwas damit zu tun, daß `\@currenvir` durch `\begin` gesetzt wird und nirgendwo sonst . . .)

6 Lange Argumente mit `\long` (kurz)

Probiere die folgenden Befehle aus:

```
\newcommand*{\Kurz}[1]{Kurz}
\newcommand{\Lang}[1]{Lang}
```

Schreibe dann einen Text, der `\Kurz` aufruft, aber ohne eine schließende klammar zu. Lasse diesen Text mehrere Absätze umfassen. Welche Fehlermeldung bekommst Du (gut merken, die wird Dir nocht häufiger begegnen)?

Ändere danach den Text so, daß Du `\Lang` statt kurz aufrufst, aber die Klammer immer noch fehlt. Wie verändert sich die Fehlermeldung?

Erinnere Dich, daß `\newcommand` ein `\long\def` verwendet, `\newcommand*` nur `\def`. Was ist also die Ursache für die unterschiedlichen Fehlermeldungen?

In einem langen Text, in dem der Befehl vielleicht hundertfach aufgerufen wird, kann eine Fehlermeldung wie die zweite sehr schwer zuzuordnen sein. Die erste Fehlermeldung wird viel näher bei dem falsch abgeschlossenen Befehl erzeugt. Verwende daher `\newcommand*`, wenn Du weißt, daß die Argumente kurz sind.

7 Expansionskontrolle mit `\expandafter` (kompliziert)

Auch wenn das Konstrukt `\expandafter` nur auf den zwei nächsten Tokens arbeitet und deren Expansionsreihenfolge vertauscht, kann man durch Aneinanderreihungen mehrerer `\expandafter`-Befehle erstaunlich viel Kontrolle auf die Expansionsreihenfolge ausüben.

So sorgt etwa die Befehlsfolge

```
\expandafter\expandafter\expandafter\A\B
```

dafür, daß `\B` *zweimal* expandiert wird, bevor `\A` an die Reihe kommt. Verstehst Du, warum? Male die Expansionen schrittweise auf.

`\TeX` kennt einige Befehle, um ausführlich zu veranschaulichen, was es tut. Füge in Deinem Dokument (kurz nach `\begin{document}`) die folgenden Befehle ein

```
\tracingmacros=1
\tracingcommands=2
\tracingonline
```

Nun kannst Du in der Ausgabe von `\TeX` erkennen, welche Makros wann expandiert werden und was wann geschieht. Experimentiere nun mit folgendem Code-Schnipsel:

```
\def\A#1{#1a}
\def\B#1{#1b}
\def\C#1{#1c}
\let\EA\expandafter
```

Die letzte Zeile definiert `\EA` als Abkürzung für `\expandafter`. Schreibe nun

```
\B\C\A d
```

und schaue auf das Ergebnis. Füge in diese Zeile jetzt `\expandafter`-Befehle ein (und *nur* solche, *keine* Klammern, *kein* Vertauschen der Befehle oder sonstiges) und versuche damit, andere Ausgaben zu erzeugen. Wieviele verschiedene Ergebnisse kannst Du erreichen? Schafft Du die Ausgabe “dbca”? Und als große Herausforderung: Schaffst Du “abcd”? (Hinweis: Für letzteres sind mindestens 11 (!) `\expandafter`-Befehle nötig.)