Hier wird die eben wieder zusammengesetzte Box tatsächlich auf der Seite platziert, wobei eine Einrückung entsprechend der »optischen Mitte« des Gedichts hinzugefügt wird – sie muss so groß sein wie die Hälfte der Textbreite minus die Hälfte der berechneten mittleren Zeilenlänge. Natürlich sollten Sie darauf achten, dass Sie die linken und rechten Seitenränder so wählen, dass die Mitte des Satzspiegels tatsächlich auch die Mitte des Papiers ist!

Dieser Ansatz zur automatischen optischen Zentrierung funktioniert natürlich auch nur dann, wenn das Gedicht komplett auf eine Seite passt, sonst gibt es wieder Probleme mit der `\vbox`. Man könnte sich eine Erweiterung denken, die außer der Breite auch noch die Höhe der Zeilen misst und ein Gedicht so eventuell auf mehrere Seiten verteilt. Allerdings ergibt die optische Zentrierung unserer Ansicht nach vor allem bei kürzeren Werken Sinn, so dass sich das Problem in der Praxis nicht in allzu großem Umfang stellen mag.

(Dieser Hack beruht auf einer Inspiration durch Axel Reichert.)

Siehe auch

- Axel Reichert beschreibt das optische Zentrierverfahren unter *http://www.axel-reichert.de/richter-typo.html*, ohne jedoch LaTeX-Code dafür zu veröffentlichen. Sie sehen es im Einsatz auf *http://www.axel-reichert.de/richter.html*, in Gestalt einer Gedichtsammlung von Ulla Richter-Federmann.

HACK #12 Programmtexte setzen

Wie Sie Programmtexte setzen, mit Hervorhebungen, in Auszügen und so weiter.

Das `listings`-Paket von Carsten Heinz ermöglicht es Ihnen, Programmcode in den verschiedensten Programmiersprachen zu setzen. Das kann zum Beispiel so aussehen:

```
/* Hauptprogramm */
void
main (void)
{
    greeting ("Hello");
    return EXIT_SUCCESS;
}
```

Die Eingabe hierfür ist einfach:

```
\begin{lstlisting}[language=C,frame=single,
  texidtexcolumns=true]
```

```
/* Hauptprogramm */
void
main (void)
{
    greeting("Hello");
    return EXIT_SUCCESS;
}
\end{lstlisting}
```

Das Paket bietet nahezu unendliche Gestaltungsmöglichkeiten für die Formatierung, die Hervorhebung von Schlüsselwörtern, die Verwendung von Listings als Gleitobjekte und vieles andere mehr. Lesen Sie dazu das dem Paket beigefügte Handbuch.

Wir konzentrieren uns an dieser Stelle auf einen praktischen Aspekt: Das `listings`-Paket macht es einfach, Programmcode zum Beispiel in wissenschaftliche Arbeiten zu übernehmen (die Diplom-Mathematiker und -Informatiker unter Ihnen werden das zu schätzen wissen). Im wirklichen Leben kann es aber durchaus passieren, dass Ihnen beim Schreiben der Arbeit noch Fehler in Programmteilen auffallen, die in Ihrem Text stehen. Es ist dann sehr ärgerlich, dieselbe Änderung an mehreren Stellen machen zu müssen, damit die Arbeit und das tatsächliche Programm übereinstimmen.

Das `listings`-Paket hat hierfür zumindest ansatzweise Abhilfe zu bieten: Mit einem Kommando wie

```
\lstinputlisting[firstline=10,lastline=15]{prog.c}
```

können Sie ein paar Zeilen aus einer Datei extrahieren, die `listings` dann so behandelt, als stünden sie in Ihrem Dokument in einer `lstlisting`-Umgebung. Damit müssen Sie Ihren Code nur noch an einem Ort verwalten, aber wirklich bequem ist es immer noch nicht, da Sie dafür sorgen müssen, dass die Zeilennummern stimmen. Ansonsten könnte es sein, dass Sie (in unserem Beispiel) fünf neue Zeilen hinter Zeile 6 einfügen, und schon holt Ihnen `\lstinputlisting` treudoof den falschen Auszug aus Ihrer Datei. Ferner müssen Sie sich selber merken, welches Stück Code tatsächlich in den Zeilen 10 bis 15 von *prog.c* zu finden war.

Beginnen wir mit einer Lösung für das letztere Problem. Wir möchten unseren Codestücken symbolische Namen geben können, beispielsweise:

```
\lstdef{Hauptprogramm}{prog.c}{10}{15}
```

Im Text verwenden wir dann nicht mehr `\lstinputlisting` mit festen Beginn- und Endzeilennummern, sondern einfach:

```
\lstuse{Hauptprogramm}
```

Immerhin wird dadurch Verwirrung darüber vermieden, was genau das Kommando `\lstinputlisting` sonst lesen würde.

Dazu muss das `\lstdef`-Kommando die angegebenen Parameter protokollieren, damit `\lstuse` sie unter dem Namen des Listings (hier »`Hauptprogramm`«) wiederfinden kann. Die übliche Vorgehensweise ist, dafür einfach ein entsprechendes LaTeX-Kommando zu definieren. Unser

```
\lstdef{Hauptprogramm}{prog.c}{10}{15}
\lstuse{Hauptprogramm}
```

zum Beispiel soll unter dem Strich nichts anderes tun, als

```
\lstinputlisting[firstline=10,lastline=15]{prog.c}
```

auszuführen, also sollte `\lstdef` dieses Kommando konstruieren und zur späteren Verwendung bereithalten. Wir legen einfach fest, dass eine Definition der Form

```
\lstdef{Name}
```

ein Kommando namens `\lst@l@`*Name* definiert (das Präfix `\lst@l@` kommt nirgendwo sonst vor, so dass eine Kollision mit anderen Kommandos unwahrscheinlich ist), dessen Bedeutung exakt dem `\lstinputlisting`-Kommando mit den entsprechenden Parametern entspricht. `\lstuse` muss dieses Kommando dann nur noch aufrufen.

Wenn Sie sich an unsere Diskussion von `\@nameuse` in Die Nummerierung von Listen ändern **[Hack #8]** erinnern, dann ist Ihnen längst klar, wie `\lstuse` aussehen könnte:

```
\newcommand*{\lstuse}[1]{\@nameuse{lst@l@#1}}
```

genügt, um zum Beispiel über

```
\lstuse{Hauptprogramm}
```

das Kommando `\lst@l@Hauptprogramm` aufzurufen.

Jetzt müssen wir nur noch `\lstdef` erklären. Dafür machen wir Sie erst mit dem `\@namedef`-Kommando bekannt, quasi dem Gegenstück zu `\@nameuse`. Mit

```
\@namedef{Name}{Ersatztext}
```

definieren Sie ein Kommando »`\`*Name*«, das den *Ersatztext* ausführt. Der *Name* darf dabei wie bei `\@nameuse` auch Ziffern enthalten und veränderlichen Bestand-

```
#!/usr/bin/perl
# lst-find: Findet benannte Bereiche in einer Datei

$comment = quotemeta('/*');

while (<>) {
    if (m,${comment}lst:([-\w]*),) { # Eine interessante Zeile
        if ($1 ne '-') {                    # Bereichsanfang
            ($name, $start) = ($1, $.+1);
        } else {                            # Bereichsende
            printf "\\lstdef{%s}{%s}{%d}{%d}\n",
                $name, $ARGV, $start, $.-1;
        }
    }
}
```

Abbildung 1-6: lst-find findet benannte Bereiche in Programmcodedateien

teilen zusammengebastelt werden. Die Definition von `\lstdef` könnte damit zum Beispiel so aussehen:

```
\newcommand{\lstdef}[3]{%
  \@namedef{lst@l@#1}{%
    \lstinputlisting[firstline=#2,lastline=#3]{#4}}}
```

Die Kombination aus `\lstdef` und `\lstuse` löst das Problem der über Ihr Dokument verteilten willkürlichen Zeilennummern. Gegen das eher noch unangenehmere Problem der willkürlichen Zeilennummern an sich, die anfällig sind gegenüber Änderungen im Programmcode, tut sie dagegen erst mal nichts. Allerdings geben die beiden Kommandos uns das Handwerkszeug, dieses Problem auch noch zu lösen, wenngleich – der Bequemlichkeit halber – unter Rückgriff auf ein paar Zeilen Perl.[2] Der Trick besteht einfach darin, interessante Stellen im Programmcode zu markieren, etwa so:

```
/*lst:Hauptprogramm*/
int
main (void)
{
  ...
}
/*lst:-*/
```


[2] Sie könnten dasselbe sicher auch mit Awk, C oder Visual Basic erreichen, aber wir machen es uns gerne einfach.

Ein Programm wie das in Abbildung 1-6 gezeigte `lst-find` kann dann die entsprechenden Stellen finden, ihnen Zeilennummern zuordnen und alle nötigen `\lstdef`-Kommandos in eine Datei schreiben. Diese Datei lesen Sie mit `\input` am Anfang Ihres Dokuments ein und haben anschließend die markierten Stellen für `\lstuse` zur Verfügung. Sie müssen dann nur noch darauf achten, nach jeder Programmänderung `lst-find` aufzurufen – aber dafür gibt es ja `make`: Etwas wie

```
SOURCEFILES = prog.c prog1.c prog2.c
paper.dvi: paper.tex paper.lst
    latex paper
paper.lst: $(SOURCEFILES)
    lst-find $(SOURCEFILES) >paper.lst
```

in Ihrem *Makefile* hilft gegen Vergesslichkeit.

Das `lst-find` in Abbildung 1-6 ist mit C-artigen Programmiersprachen verheiratet – es findet seine Marken in Kommentaren, die mit

```
/*lst:
```

anfangen. Sie können das aber am Anfang des Programms ändern. Das Programm besteht nicht auf einer speziellen Ende-Markierung für Kommentare, sondern interpretiert eine beliebige Folge von alphanumerischen Zeichen hinter dem »`lst:`« als Marke.

Es wäre natürlich auch ohne Weiteres möglich, automatisch den Anfang und das Ende von C-Funktionen zu finden. Solange Sie sich an gewisse lexikalische Konventionen halten – etwa Namen in Funktionsdefinitionen und die schließende geschweifte Klammer einer Funktion immer an den linken Rand zu schreiben, was ohnehin guter C-Stil ist –, schaffen Sie das auch, ohne `lst-find` zu einer Art C-Übersetzer zu machen. Der Ansatz mit den expliziten Marken ist dagegen dann vorteilhaft, wenn Sie nicht nur ganze Funktionen zitieren wollen, sondern interessante Stellen innerhalb einer Funktion oder Bereiche, die sich über Funktionsgrenzen hinaus erstrecken.

HACK
#13

Programm-Listings mit Stil

Wie Sie Ihren Code-Auszügen listings-Optionen mitgeben.

Der im vorigen Hack gezeigte Ansatz ist nützlich, aber erlaubt es nicht, individuelle Listings über `listings`-Parameter zu beeinflussen – es gibt keine Möglichkeit, sie an das `\lstinputlisting`-Kommando durchzureichen. (Allgemeingültige Vorgaben können Sie nach wie vor über das `\lstset`-Kommando machen – sehen Sie in der Dokumentation nach – aber das ist nicht wirklich

bequem.) Hier zeigen wir Ihnen noch, wie Sie diesen Mangel beheben können, und zwar um den Preis eines etwas unverständlicheren TEX-Codes.

Zuerst muss das `\lstuse`-Kommando ein optionales Argument bekommen, mit dem Sie zusätzliche Parameter angeben können, etwa so:

```
\lstuse[frame=L]{Hauptprogramm}
```

Dazu genügt uns ein wie folgt definiertes `\lstuse`:

```
\newcommand{\lstuse}[2][]{\@nameuse{lst@l@#2}{#1}}
```

(Erinnern Sie sich daran, dass Sie LATEX-Kommandos mit einem optionalen Argument definieren können, indem Sie den Ersatzwert dafür in eckigen Klammern hinter die Argumentanzahl schreiben.) Da die mit `\@namedef` eingeführten Kommandos diesen Mechanismus nicht unterstützen, geben wir das optionale Argument – ob es nun tatsächlich vorhanden war oder nur der leere Standardwert angenommen wurde – als »echtes« Argument an das `\lst@l@...`-Kommando weiter.

So weit, so gut – nur sieht `\@namedef` nicht offiziell vor, dass die damit definierten Kommandos überhaupt Argumente übernehmen. Trotzdem geht das; Sie müssen nur darauf achten, die Syntax des TEX-Kommandos `\def` einzuhalten und nicht die des `\newcommand` von LATEX. Unser `\lstdef` wird dann zu etwas wie:

```
\newcommand{\lstdef}[4]{%
  \@namedef{lst@l@#1}##1{%
    \lstinputlisting[##1,firstline=#3,lastline=#4]{#2}}}
```

Das »#« in »`\@namedef{...}##{`« muss verdoppelt werden, da es sich nicht auf das erste Argument von `\lstdef` bezieht, sondern Platzhalter für das erste Argument des Kommandos ist, das gerade mit `\@namedef` definiert wird – wir machen ja Folgendes:

```
\def\lst@l@Hauptprogramm#1{...}
```

Es ist übrigens nicht schlimm, wenn wir in `\lstuse` kein optionales Argument angeben. Das `keyval`-Paket, das `listings` und viele andere LATEX-Pakete zur Auswertung ihrer Parameter verwenden, stört sich nicht an überzähligen Kommas, so dass ein Kommando wie

```
\lstinputlisting[,firstline=100,lastline=110]{prog.c}
```

keinen Anstoß erregt.