Inhalt

Lehrziele

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie

- die Begriffe attributierte Grammatik, syntax-gesteuerte Definition, S-attributierte Definition, L-attributierte Definition und semantische Regel erläutern und deren Zusammenhänge darstellen können,
- Datenfluss- und Abhängigkeitsgraphen zu attributierten Grammatiken aufstellen können,
- eine kontextfreie Grammatik zu einer attributierten Grammatik mit vorgegebenen Eigenschaften erweitern können,
- eine kontextfreie Grammatik mit vorgegebenen Eigenschaften in ein Übersetzungsschema umformen können,
- Linksrekursion aus einem Übersetzungsschema beseitigen können,
- aus einem Übersetzungsschema einen syntax-gesteuerten Übersetzer konstruieren können,
- die bei der Implementierung L-attributierter Definitionen oder syntax-gesteuerter Übersetzungsschemata im Rahmen einer Bottom-up-Analyse auftretenden Probleme und Ansätze zu deren Lösung beschreiben können.

Kapitel 4

Syntaxgesteuerte Übersetzung

In den letzten beiden Kapiteln haben wir auf der Basis von Techniken zur Beschreibung der syntaktischen Struktur von Programmiersprachen – regulären Ausdrücken und kontextfreien Grammatiken – Methoden für die lexikalische und syntaktische Analyse entwickelt. Wir sind also nun in der Lage, für eine gegebene Eingabezeichenfolge der Quellsprache einen Ableitungsbaum zu konstruieren. Unser nächstes Ziel muss darin bestehen, aus dem gegebenen Ableitungsbaum Programme der Zielsprache der Übersetzung zu generieren.

Tatsächlich wird ein Ableitungsbaum im Allgemeinen nicht komplett aufgebaut, bevor mit der Übersetzung begonnen wird, sondern Übersetzungsschritte werden ausgeführt, sobald Teilstrukturen des Ableitungsbaumes erkannt sind. Das heißt, Übersetzungsschritte werden verzahnt mit der Analyse, z. B. einer Top-down-Analyse, ausgeführt, und der Ableitungsbaum existiert gewöhnlich nur implizit. In diesem Kapitel betrachten wir zunächst das methodische Problem, Übersetzungsaktionen mit dem Erkennen von Teilstrukturen des Ableitungsbaumes zu verbinden. Es gibt dazu eine elegante Technik, nämlich die der *syntaxgesteuerten Übersetzung*, deren formale Grundlage *attributierte Grammatiken* bilden.

Bei der Besprechung von Lex und Yacc haben wir schon einen Eindruck davon bekommen, wie „semantische Aktionen“ in Regeln der lexikalischen Analyse bzw. der Syntaxanalyse eingebettet werden können. Das sind tatsächlich Anwendungen der Methodik, die wir in diesem Kapitel allgemein studieren. Abschnitt 4.1 beschreibt das Grundkonzept der *attributierten Grammatik* und der daran angelehnten *syntaxgesteuerten Definition*. In Abschnitt 4.2 werden eingeschränkte Klassen syntaxgesteuerter Definitionen betrachtet, die sich effizient in die Syntaxanalyse integrieren lassen. In Abschnitt 4.3 erweitern wir die in Abschnitt 3.2 eingeführte Implementierungstechnik für die Top-down-Analyse, insbesondere die Analyse mit rekursivem Abstieg, um die Einbettung von Übersetzungsaktionen. Schließlich diskutieren wir in Abschnitt 4.4 spezielle Probleme bei der Verbindung von Übersetzungsaktionen mit der Bottom-up-Analyse.

4.1 Attributierte Grammatik, syntaxgesteuerte Definition

Man kann sich leicht klarmachen, dass ein zentrales und ganz allgemeines Problem bei der Übersetzung darin besteht, Information mit Teilstrukturen des Ableitungsbaumes zu assoziieren und diese Information beim Aufbau des Baumes geeignet zu verwalten. Beispielsweise möchte man letztlich jedem Teilbaum, der ausführbare Anweisungen darstellt, ein entsprechendes Code-Fragment zuordnen. Da Ableitungsbäume auf kontextfreien Grammatiken basieren, bietet es sich an, jedem Symbol der Grammatik eine Menge von *Attributen* als „Informationsbehälter" zuzuordnen und jeder Produktion der Grammatik eine Menge von Regeln, die die Berechnung von Attributwerten aus den Werten von Attributen anderer in der Produktion vorkommender Symbole beschreiben. Das ist gerade die Idee der attributierten Grammatik.

Definition 4.1: Gegeben sei eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$. Eine *attributierte Grammatik* enthält für jedes Symbol $X \in (N \cup \Sigma)$ eine Menge von *Attributen* $A(X)$ und für jede Produktion $p: X_0 \to X_1 \ldots X_m$ ($X_i \in N \cup \Sigma$) eine Menge von *semantischen Regeln* $R(p)$ der Form

$$X_i.a := f(X_j.b, \ldots, X_k.c)$$

wobei die X_i usw. in der Regel vorkommende Symbole darstellen und $X_i.a$ usw. deren Attribute. Für jedes Auftreten von X in einem Ableitungsbaum (zu einem Wort in $L(G)$) ist höchstens eine Regel anwendbar, um $X.a$ zu berechnen, für alle $a \in A(X)$. □

Eng verwandt, aber etwas allgemeiner, ist der Begriff der *syntaxgesteuerten Definition*. Der einzige Unterschied besteht darin, dass eine semantische Regel auch Seiteneffekte haben darf, d. h., man darf beliebige Anweisungen benutzen. Wir arbeiten im Folgenden mit syntaxgesteuerten Definitionen.

Die Art der in Attributen aufbewahrten Information ist nicht näher spezifiziert, es könnte z. B. ein Codefragment, die Adresse der Folgeinstruktion, der Name einer Sprungmarke, ein Typ in einer Deklaration sein.

In einer semantischen Regel

$$b := f(c_1, \ldots, c_k)$$

in der b und die c_i Attribute sind, hängt der Wert von b von den Werten der c_i ab. Das heißt, b kann erst berechnet werden, sobald die c_i bekannt sind. Entsprechend muss die Reihenfolge, in der Attributwerte berechnet werden, mit der Reihenfolge beim Aufbau des Ableitungsbaumes oder mit der eines Durchlaufs des Baumes zusammenpassen. Dies führt zu einer wichtigen Unterscheidung von zwei Arten von Attributen. Sei

$$p: X_0 \to X_1 \ldots X_m$$

die Produktion p, zu deren Regelmenge die Regel $b := f(c_1, \ldots, c_k)$ gehört. Das Attribut b heißt *synthetisiert*, falls es Attribut von X_0 ist; es heißt *vererbt*, falls es Attribut eines der X_i auf der rechten Seite von p ist. Die c_i sind Attribute beliebiger in der Produktion vorkommender Symbole, bei einem synthetisierten Attribut sind es gewöhnlich Attribute von Symbolen der rechten Seite.

Damit lassen sich für ein Grammatiksymbol X seine Attribute im Allgemeinen als entweder synthetisiert oder vererbt klassifizieren. Allerdings kann in einer rekursiven Regel $X \to XY$ ein Attribut a von X auf beiden Seiten auftreten; man kann dann nur von einem synthetisierten und einem vererbten Auftreten von $X.a$ sprechen.

Synthetisierte Attribute

Eine syntaxgesteuerte Definition, in der nur synthetisierte Attribute vorkommen, heißt *S-attributierte Definition*. Eine solche Definition ist besonders einfach zu behandeln, da für jeden Knoten des Ableitungsbaumes seine Attribute bottom-up aus den Attributen seiner Söhne zu berechnen sind.

Beispiel 4.2: Wir betrachten, ähnlich dem Yacc-Beispiel in Abschnitt 3.3.4, eine vereinfachte Grammatik, die die Eingabe arithmetischer Ausdrücke auf einem Taschenrechner beschreibt. Es sei nur erlaubt, einzelne Ziffern einzugeben und diese zu addieren oder zu multiplizieren. Mit Hilfe einer syntaxgesteuerten Definition soll der Wert des eingegebenen Ausdrucks berechnet werden. Dazu wird jedem Nichtterminal ein Attribut *val* zugeordnet.

Produktion	*Semantische Regel*
$L \to E$ =	*output* (*E.val)*
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to$ **digit**	$F.val :=$ **digit**.*lexval*

Wie man sieht, kennzeichnen wir innerhalb einer Produktion mehrfach vorkommende Symbole durch Indizierung, um Eindeutigkeit in der semantischen Regel zu erreichen. Es wird angenommen, dass **digit** als Ergebnis der lexikalischen Analyse entsteht und dass im Attribut *lexval* der dort ermittelte Ziffernwert erhältlich ist. Die Eingabe wird mit einem Gleichheitszeichen abgeschlossen, woraufhin der Taschenrechner das Ergebnis liefert. □

Man sieht, dass diese syntaxgesteuerte Definition nur synthetisierte Attribute benutzt. Der folgende Ableitungsbaum (Abb. 4.1) für die Eingabe

6 * (3 + 5)

ist mit den Werten des Attributs *val* (bzw. *lexval*) beschriftet; Pfeile zwischen diesen Beschriftungen zeigen den Datenfluss zwischen den Attributen. Der so erweiterte Ableitungsbaum mit den Belegungen der Attribute heißt *Datenflussgraph*.

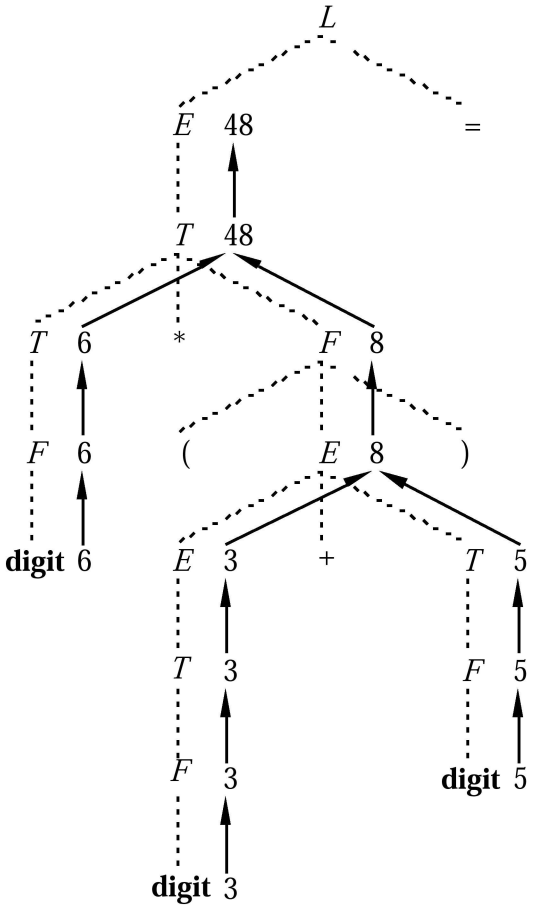

Abb. 4.1. Ableitungsbaum und Datenflussgraph für den Ausdruck 6 * (3 + 5)

Eine solche S-attributierte Definition lässt sich auf recht einfache Art innerhalb einer LR-Syntaxanalyse implementieren. Ansatzpunkt sind die Reduktionsschritte. Dort werden ja die Symbole der rechten Seite einer Produktion, die gerade zuoberst auf dem Stack stehen, durch das Symbol auf der linken Seite der Produktion ersetzt. Um eine S-attributierte Definition zu implementieren, verwaltet man nun auf dem Stack Darstellungen der Symbole, die ihre Attribute beinhalten. (Zum Beispiel könnte jedes Symbol mit seinen Attributen als Record dargestellt sein; der Stack könnte Zeiger auf solche Records enthalten.) Vor dem Reduktionsschritt stehen dann die Attribute der Symbole der rechten Seite auf dem Stack zur Verfügung, die zur Produktion gehörigen semantischen Regeln können ausgewertet werden. Die Ergebnisse werden dann nach der Reduktion den Attributen des neuen obersten Stacksymbols zugewiesen.

Selbsttestaufgabe 4.1: Sei die Grammatik $G = (\{R, B, S\}, \{$ **ziffer**, **.**, **e**, **+**, **-**$\}, P, R)$ zur Darstellung reeller Zahlen im Exponentialformat mit

$$
\begin{aligned}
P = \{\ & R \rightarrow SB.B\mathbf{e}SB \\
& S \rightarrow + \\
& S \rightarrow - \\
& B \rightarrow B\ \mathbf{ziffer} \\
& B \rightarrow \mathbf{ziffer}\ \}
\end{aligned}
$$

gegeben. Es wird angenommen, dass **ziffer** als Ergebnis der lexikalischen Analyse entsteht und in dem Attribut **ziffer**.*lexval* der ermittelte Ziffernwert zu finden ist. Geben Sie eine syntaxgesteuerte Definition an, die den Zahlenwert (nicht String!) jedes Unterausdrucks im Fließkommaformat bestimmt. (Hinweis: Verwenden Sie zwei Attribute.) □

Vererbte Attribute

Vererbte Attribute sind oft nützlich, wenn man an ein Symbol Informationen aus dem Kontext, in dem es auftritt, übertragen will.

Beispiel 4.3: Wir betrachten Typ-Deklarationen der Form

integer *a*, *b*, *c*;
real *e*, *f*;

Hier muss der erkannte Typ der Deklaration an die deklarierten Identifikatoren hinabgereicht werden. Die syntaxgesteuerte Definition sieht so aus:

Produktion	*Semantische Regel*
$D \rightarrow T\ L;$	*L.type* := *T.type*
$T \rightarrow$ **integer**	*T.type* := *integer*
$T \rightarrow$ **real**	*T.type* := *real*
$L \rightarrow L_1$, **id**	L_1.*type* := *L.type* *addtype* (**id**.*entry*, *L.type*)
$L \rightarrow$ **id**	*addtype* (**id**.*entry*, *L.type*)

Die Prozedur *addtype* trägt für einen Identifikator seinen Typ in die Symboltabelle ein.

Hier ist das Attribut *type* des Nichtterminals *L* ein vererbtes Attribut. Das Attribut *type* des Nichtterminals *T* nennt man *intrinsisch*, das heißt, sein Wert ist für eine gewählte Produktion fest vorgegeben. Dies ist ein Spezialfall eines synthetisierten Attributs. □

Wenn man zu einem gegebenen Ableitungsbaum die Attribute der vorkommenden Symbole als Knoten und die Abhängigkeiten zwischen Attributen, die aufgrund der semantischen Regeln entstehen, als Kanten einträgt, erhält man den *Abhängigkeits-*

graphen zu diesem Baum. Für synthetisierte Attribute haben wir einen solchen Graphen schon im vorigen Beispiel gesehen. In Beispiel 4.3 ergibt sich für eine Deklaration

real *a*, *b*, *c*;

der in Abb. 4.2 gezeigte Ableitungsbaum und Abhängigkeitsgraph.

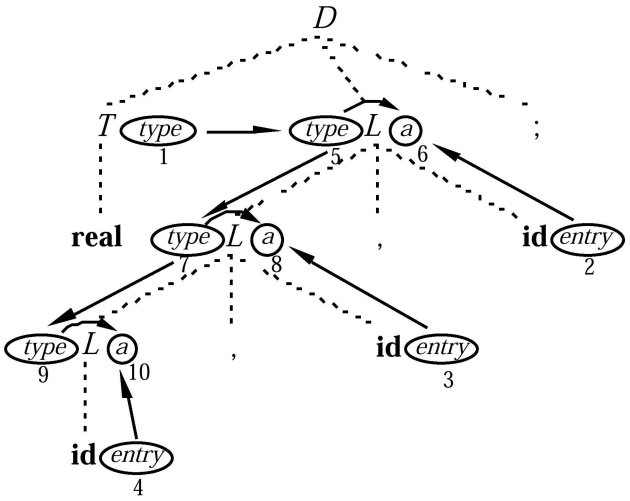

Abb. 4.2. Ableitungsbaum und Abhängigkeitsgraph für die Deklaration **real**: *a*, *b*, *c*;

Um den Abhängigkeitsgraphen zu zeichnen, fassen wir eine semantische Regel, die nur aus einem Prozeduraufruf $f(c_1, \ldots, c_k)$ besteht, als Zuweisung an ein synthetisiertes Dummy-Attribut $d := f(c_1, \ldots, c_k)$ auf. Im Beispiel betrifft das die Prozedur *addtype*; wir haben dort für das Nichtterminal *L* ein Attribut *a* eingeführt, um die *addtype*-Abhängigkeiten darzustellen.

Um aus einem Abhängigkeitsgraphen eine geeignete Reihenfolge für die Auswertung der semantischen Regeln zu erhalten, muss man ihn *topologisch sortieren*. Das heißt, man muss für die Knoten eine Reihenfolge

$n_1, \ldots, n_k$

ermitteln, so dass für jede Kante (n_i, n_j) im Graphen gilt $i \leq j$. Algorithmen für topologisches Sortieren eines azyklischen, gerichteten Graphen findet man in Büchern zu Datenstrukturen, z. B. (Aho, Hopcroft und Ullman 1983) oder (Sedgewick 1990). Man erhält einen solchen Algorithmus auch relativ leicht als Variante des Tiefendurchlaufs durch den Graphen: Wenn die Durchlaufprozedur den Namen des besuchten Knotens jeweils *nach* dem Besuch der Söhne ausgibt, werden die Knoten insgesamt in *umgekehrter* topologischer Ordnung ausgegeben. Das heißt, man erhält die gewünschte Folge, indem man das Ergebnis von hinten nach vorne liest.

Das topologische Sortieren lässt sich allerdings nur dann durchführen, wenn der gegebene Graph azyklisch ist. Es gibt Algorithmen, die überprüfen, ob eine gegebene syntaxgesteuerte Definition für jeden Ableitungsbaum zu einem azyklischen Abhängigkeitsgraphen führt; diese werden hier aber nicht besprochen.

Im Beispiel ist eine derartige Reihenfolge durch Indizierung der Knoten dargestellt. Damit würden die semantischen Regeln in folgender Reihenfolge ausgeführt (Attribute sind mit Knotennummern indiziert):

$type_1$:= *real*;
$type_5$:= $type_1$;
addtype ($entry_2$, $type_5$);
$type_7$:= $type_5$;
addtype ($entry_3$, $type_7$);
$type_9$:= $type_7$;
addtype ($entry_4$, $type_9$);

Die explizite Konstruktion von Abhängigkeitsgraphen ist ein sehr allgemeines, aber auch sehr aufwendiges Verfahren. In der Praxis beschränkt man sich daher oft auf speziellere Klassen syntaxgesteuerter Definitionen, bei denen man ohne diese Konstruktion auskommt. Eine solche Klasse betrachten wir im Folgenden.

4.2 L-attributierte Definition, Übersetzungsschema

Eine syntaxgesteuerte Definition lässt die Reihenfolge der Ausführung ihrer semantischen Regeln offen. Sie erlaubt damit eine Spezifikation eines Übersetzungsvorgangs auf recht hoher Ebene. Wir gehen nun zu einer etwas niedrigeren Spezifikationsebene über, auf der diese Reihenfolge festgelegt wird; dadurch ist es möglich, weitere Implementierungsdetails auszudrücken. An die Stelle der syntaxgesteuerten Definition tritt nun ein *Übersetzungsschema*. Gleichzeitig betrachten wir eine eingeschränkte Klasse syntaxgesteuerter Definitionen, genannt *L-attributierte Definition*, die sich direkt in ein Übersetzungsschema überführen lässt.

Ausgangspunkt ist die Beobachtung, dass es für Bäume, und damit auch für Ableitungsbäume, gewisse Standard-Durchlaufarten gibt und dass man die Auswertung semantischer Regeln mit einem solchen Durchlauf verbinden kann. Die naheliegende Durchlaufart ist der *Tiefendurchlauf*, bei dem die Söhne eines Knotens in der Reihenfolge *von links nach rechts* (also wie üblich) besucht werden. Die Auswertung semantischer Regeln wird wie folgt einbezogen:

```
procedure dfeval (n: node)
begin Seien m_1, …, m_k (von links nach rechts) die Söhne von n
   for i := 1 to k do
      berechne die vererbten Attribute von m_i; dfeval (m_i)
   od;
   berechne die synthetisierten Attribute von n
end
```

Damit das funktioniert, dürfen offensichtlich die Regeln, die vererbten Attributen von m_i einen Wert zuweisen, nur auf vererbte oder synthetisierte Attribute von $m_1, \ldots, m_{i-1}$ und auf vererbte Attribute von n zurückgreifen.

Definition 4.4: Eine syntaxgesteuerte Definition heißt *L-attributiert*, wenn jedes vererbte Attribut eines Symbols X_j auf der rechten Seite einer Produktion $X_0 \rightarrow X_1 \ldots X_m$ nur abhängt von

(i) Attributen der Symbole $X_1, \ldots, X_{j-1}$, und
(ii) vererbten Attributen von X_0.

(das L steht für „von **l**inks nach rechts“). □

Übrigens ist jede S-attributierte Definition auch L-attributiert, da nur Einschränkungen für vererbte Attribute gemacht werden.

Ein *Übersetzungsschema* ist eine kontextfreie Grammatik mit Attributen zu Grammatiksymbolen, in der die rechten Seiten von Produktionen Folgen von Grammatiksymbolen und semantischen Regeln sind.

Beispiel 4.5: Das folgende Übersetzungsschema überführt einfache arithmetische Ausdrücke aus der Infix- in die Postfix-Notation. (In der Postfix-Notation werden Operatoren nachgestellt, d. h. 2+(3*5) wird zu 235*+.) Im Beispiel kommen nur Additions- und Subtraktionsoperatoren und keine Klammern vor.

```
expr   →  term rest
rest   →  + term {write ("+")} rest
          | – term {write ("–")} rest
          | ε
term   →  num {write (num.value)}
```

Einen Ableitungsbaum für ein Übersetzungsschema stellt man in der Form dar, dass semantische Regeln als zusätzliche Blätter auftreten. Für den Ausdruck

17 – 6 – 9

ergibt das den in Abb. 4.3 gezeigten Baum.

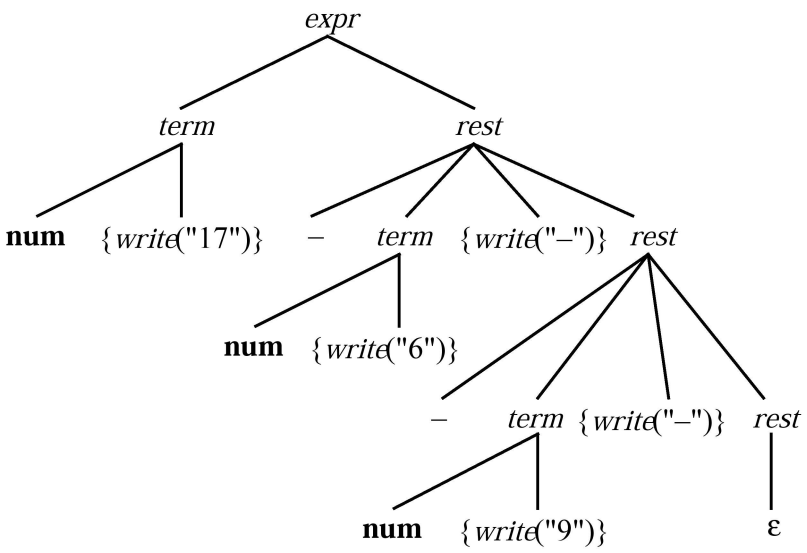

Abb. 4.3. Ableitungsbaum für das Übersetzungsschema für 17 – 6 – 9

Ein Tiefendurchlauf durch diesen Baum, bei dem semantische Aktionen ausgeführt werden, sobald das entsprechende Blatt besucht wird, ergibt:

17 6 – 9 –

Das ist die Postfix-Darstellung des Ausdrucks 17 – 6 – 9. □

Man kann ein Übersetzungsschema direkt angeben. Man kann es aber auch aus einer gegebenen L-attributierten Definition systematisch erzeugen, und zwar nach folgenden Regeln:

1. Der Wert eines vererbten Attributs eines Symbols auf der rechten Seite einer Produktion muss in einer semantischen Regel *vor* diesem Symbol zugewiesen werden.
2. Der Wert eines synthetisierten Attributs wird in einer semantischen Regel zugewiesen, die am Ende der rechten Seite steht.

Die syntaxgesteuerte Definition aus Beispiel 4.3 ist L-attributiert. Mit den obigen Regeln erhalten wir dazu folgendes Übersetzungsschema:

$D \rightarrow$ T $\{L.type := T.type\}$
L
;
$T \rightarrow$ **integer** $\{T.type := integer\}$
$T \rightarrow$ **real** $\{T.type := real\}$

$L \rightarrow$ L_1 , **id** $\quad \{L_1.type := L.type\}$
$\{addtype\ (\mathbf{id}.entry, L.type)\}$

$L \rightarrow$ **id** $\quad \{addtype\ (\mathbf{id}.entry, L.type)\}$

4.3 Top-down-Übersetzung

Ein vorgreifender Analysator (predictive parser, Abschnitt 3.2) besucht die Knoten des Ableitungsbaumes gerade in der Tiefendurchlauf-Reihenfolge, die die Basis für eine L-attributierte Definition und ein Übersetzungsschema ist. Das führt dazu, dass ein Übersetzungsschema direkt im Rahmen einer Top-down-Analyse implementiert werden kann, falls die zugrundeliegende Grammatik vom Typ LL(1) ist. In diesem Abschnitt erweitern wir das Schema aus Abschnitt 3.2.5 zur Implementierung eines vorgreifenden Analysators durch ein System rekursiver Prozeduren in der Weise, dass wir die Implementierung eines Übersetzungsschemas erhalten. Damit erhalten wir insgesamt eine systematische Technik zur Implementierung einer L-attributierten syntaxgesteuerten Definition auf der Basis einer LL(1)-Grammatik.

In der Praxis kommt es jedoch oft vor, insbesondere bei der Übersetzung von Ausdrücken, dass man eine syntaxgesteuerte Definition am einfachsten für eine linksrekursive Grammatik erstellen kann. Als letzter Baustein in unserer Methodik fehlt also noch eine Technik zur *Beseitigung von Linksrekursion aus einem Übersetzungsschema*. Wir beschränken uns allerdings auf die Behandlung eines linksrekursiven Übersetzungsschemas mit synthetisierten Attributen.

In Abschnitt 3.2.1 wurde ein Paar linksrekursiver Produktionen

$$A \rightarrow A\alpha \mid \beta$$

umgewandelt in

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Wir betrachten nun ein linksrekursives Übersetzungsschema

$A \rightarrow A_1\ Y \quad \{A.a := g(A_1.a, Y.y)\}$
$\mid X \quad \{A.a := f(X.x)\}$

Dabei gibt es zu jedem Grammatiksymbol ein synthetisiertes Attribut, das mit einem entsprechenden Kleinbuchstaben bezeichnet wird. Der durch dieses Schema erzeugte Datenfluss- bzw. Abhängigkeitsgraph ist in Abb. 4.4 veranschaulicht.

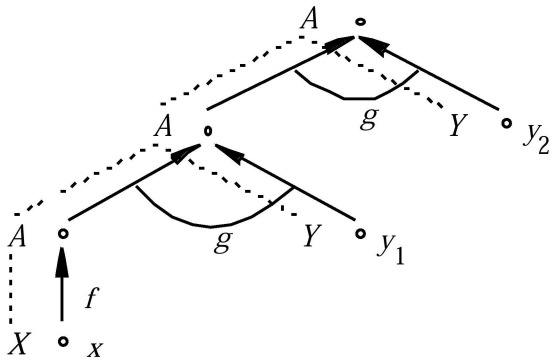

Abb. 4.4. Datenflussgraph für linksrekursives Übersetzungsschema

Wenn wir die übliche Technik zur Beseitigung von Linksrekursion einsetzen, erhalten wir zunächst die Produktionen

$$A \rightarrow X\,R$$
$$R \rightarrow Y\,R \mid \varepsilon$$

und es würde für die gleiche Blattfolge *XYY* der Baum erzeugt:

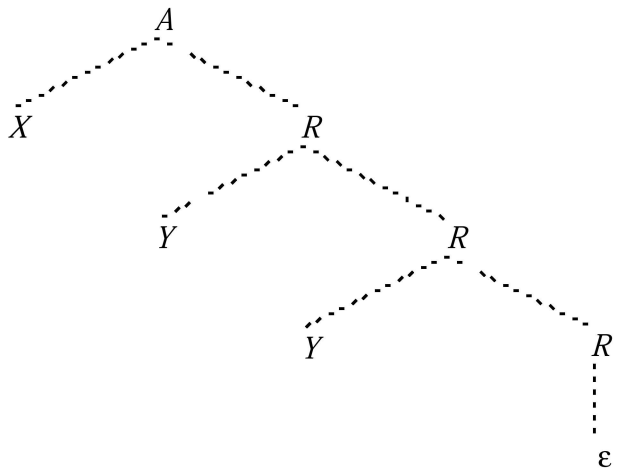

Abb. 4.5. Ableitungsbaum für modifizierte Grammatik

Die Idee besteht nun darin, das Symbol *R* mit einem vererbten Attribut *R.v* und einem synthetisierten Attribut *R.s* auszustatten und dann den Datenfluss so zu organisieren, wie in Abb. 4.6 gezeigt. *R.v* und *R.s* sind dort links und rechts von *R* gezeichnet.

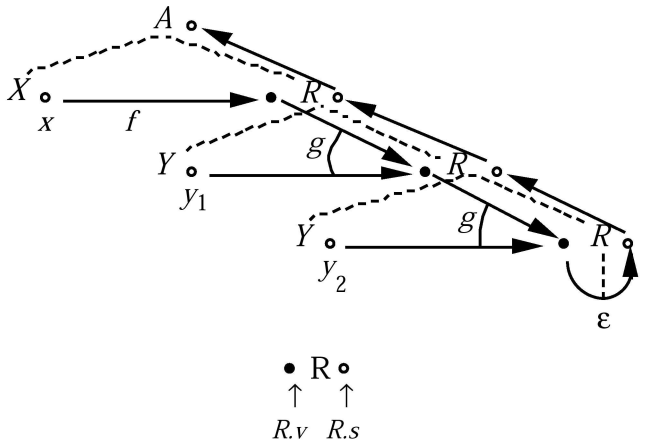

Abb. 4.6. Datenflussgraph für modifiziertes Übersetzungsschema

Man sieht, dass dieses Schema äquivalent ist, da der resultierende Wert für $A.a$ in der Wurzel in beiden Fällen gleich ist, nämlich $g(g(f(x), y_1), y_2)$. Das entsprechende Übersetzungsschema ist dann:

A	$\rightarrow$	X	$\{R.v := f(X.x)\}$
		R	$\{A.a := R.s\}$
R	$\rightarrow$	Y	$\{R_1.v := g(R.v, Y.y)\}$
		R_1	$\{R.s := R_1.s\}$
R	$\rightarrow$	ε	$\{R.s := R.v\}$

Beispiel 4.6: Wir wenden die Technik nun auf ein konkretes linksrekursives Übersetzungsschema an, das wiederum Ausdrücke mit Addition und Subtraktion darstellt. Diesmal berechnet das Schema aber nicht Postfix-Notation, sondern den Wert des Ausdrucks.

E	$\rightarrow$	$E_1 + T$	$\{E.val := E_1.val + T.val\}$
E	$\rightarrow$	$E_1 - T$	$\{E.val := E_1.val - T.val\}$
E	$\rightarrow$	T	$\{E.val := T.val\}$
T	$\rightarrow$	(E)	$\{T.val := E.val\}$
T	$\rightarrow$	**num**	$\{T.val :=$ **num**$.lexval\}$

Als umgeformtes Schema ergibt sich:

E	$\rightarrow$	T	$\{R.v := T.val\}$
		R	$\{E.val := R.s\}$
R	$\rightarrow$	$+$	
		T	$\{R_1.v := R.v + T.val\}$
		R_1	$\{R.s := R_1.s\}$

$$
\begin{array}{lcll}
R & \rightarrow & - & \\
 & & T & \{R_1.v := R.v - T.val\} \\
 & & R_1 & \{R.s := R_1.s\} \\
R & \rightarrow & \varepsilon & \{R.s := R.v\} \\
T & \rightarrow & (& \\
 & & E & \\
 & &) & \{T.val := E.val\} \\
T & \rightarrow & \textbf{num} & \{T.val := \textbf{num}.lexval\}
\end{array}
$$

□

Selbsttestaufgabe 4.2:

(a) Geben Sie eine zu der Grammatik aus Aufgabe 4.1 äquivalente LL(1)-Grammatik G_1 an.

(b) Geben Sie zu G_1 ein entsprechendes LL(1)-Übersetzungsschema an, das ebenfalls den Zahlwert im Fließkommaformat bestimmt. Geben Sie auch die von Ihnen verwendeten vererbten Attribute an. □

Wir gehen nun davon aus, dass ein Übersetzungsschema auf der Basis einer LL(1)-Grammatik gegeben ist, und erweitern die Methode zur Konstruktion eines vorgreifenden Analysators (mit rekursivem Abstieg) aus Abschnitt 3.2.5 zu einer Methode zur Konstruktion eines entsprechenden *syntaxgesteuerten Übersetzers*. Gegeben sei also für jedes Nichtterminal A die Menge der A-Produktionen

$$
\begin{array}{lcll}
A & \rightarrow & \alpha_1 & \mid D_1 \\
 & & \alpha_2 & \mid D_2 \\
 & & \cdots & \\
 & & \alpha_m & D_m
\end{array}
$$

mit zugeordneten Steuermengen D_i (die Produktionsnummern interessieren uns hier nicht, da das Ziel nicht darin besteht, eine Ableitung auszugeben). Die Symbole der Grammatik haben nun natürlich Attribute und die α_i sind entsprechende rechte Seiten eines Übersetzungsschemas.

Algorithmus 4.7: Konstruktion eines syntaxgesteuerten Übersetzers.

Eingabe Ein syntaxgesteuertes Übersetzungsschema auf der Basis einer LL(1)-Grammatik.

Ausgabe Programmtext für einen syntaxgesteuerten Übersetzer.

Methode

1. Erzeuge für jedes Nichtterminal A eine Funktionsprozedur. Diese Prozedur hat einen formalen Parameter für jedes vererbte Attribut von A, und sie liefert die Werte der synthetisierten Attribute von A als Ergebnis zurück. Falls A nur *ein* synthetisiertes Attribut besitzt, ist das ohne weiteres realisierbar. Falls mehrere synthetisierte Attribute vorkommen, kann die Prozedur z. B. einen Zeiger auf einem Record zurückgeben, der die Werte dieser Attribute enthält. Wir nehmen

im Folgenden der Einfachheit halber an, dass A nur *ein* synthetisiertes Attribut besitzt. Die Prozedur hat weiterhin lokale Variablen für alle in A-Produktionen vorkommenden Attribute.

2. Die Struktur des Prozedurrumpfes ist ebenso wie in Abschnitt 3.2.5, also

```
if symbol ∈ D1 then
   bearbeite α1
elsif symbol ∈ D2 then
…
else error
fi
```

3. Die Behandlung der rechten Seiten, also „bearbeite α_i“ wird nun etwas anders verfeinert. Die rechten Seiten sind Folgen von Terminalsymbolen, Nichtterminalen und semantischen Regeln:

   (i) Terminalsymbol X:

   Falls X Attribute besitzt, deren Werte hier benötigt werden, erzeuge Anweisungen, die diese Werte in lokale Variablen übertragen, z. B.

   $Xx := symbol.x$

   Erzeuge anschließend eine Anweisung

   $match(X)$

   (ii) Nichtterminal X:

   Erzeuge eine Zuweisung

   $Xs := X\,(Xv_1, \ldots, Xv_n)$

   wobei Xs die lokale Variable zur Aufnahme des synthetisierten Attributs von X ist und rechts ein Aufruf der Funktionsprozedur für X steht, dessen Parameter die Variablen sind, die zu den vererbten Attributen gehören.

   (iii) Semantische Regel $b := f(c_1, \ldots, c_k)$:

   Kopiere die Regel in den Programmtext, wobei alle Bezüge auf Attribute durch die entsprechenden Variablen ersetzt werden. □

Wir wenden den Algorithmus auf das Übersetzungsschema aus Beispiel 4.6 an. Nach dem Verfahren aus Abschnitt 3.2.3 konstruiert man zunächst die Steuermengen:

E	$\rightarrow$	$T\,R$	{(, **num**}
R	$\rightarrow$	$+\,T\,R$	{+}
		$\mid -\,T\,R$	{–}
		$\mid \varepsilon$	{$,)}
T	$\rightarrow$	$(\,E\,)$	{(}
		$\mid$ **num**	{**num**}

Der Algorithmus erzeugt dann folgende Funktionsprozeduren. Wir nehmen an, dass numerische Werte als *integer* darstellbar sind.

```
function E : integer;
var Tval, Eval, Rv, Rs: integer;
begin
   if symbol = (  or symbol = num
   then Tval := T; Rv := Tval; Rs := R(Rv); Eval := Rs; return Eval
   else error
   fi
end;

function R (Rv :integer): integer;
var Tval, R1v, R1s, Rs: integer;
begin
   if symbol = + then
      match (+); Tval := T; R1v := Rv + Tval; R1s := R(R1v); Rs := R1s;
      return Rs
   elsif symbol = - then
      (analog)
   elsif symbol = $ or symbol = ) then
      Rs := Rv; return Rs;
   else error
   fi
end;

function T : integer;
var Tval, Eval, numlexval: integer;
begin
   if symbol = ( then
      match (); Eval := E; match := ()); Tval := Eval; return Tval
   elsif symbol = num then
      numlexval := symbol.lexval; match (num); Tval := numlexval;
      return Tval;
   else error
   fi
end;
```

Es gelten ähnliche Bemerkungen wie in Abschnitt 3.2.5: Der hier gezeigte Programmtext wurde streng nach dem Schema des Algorithmus 4.7 erzeugt; Effizienzverbesserungen könnten von Hand oder automatisch vorgenommen werden.

Selbsttestaufgabe 4.3: Konstruieren Sie zu der Grammatik G_1 aus Aufgabe 4.2 einen syntaxgesteuerten Übersetzer. □

4.4 Bottom-up-Übersetzung

Das Grundmuster der syntaxgesteuerten Übersetzung im Rahmen einer Bottom-up-Analyse ist sehr einfach, und wir haben es schon bei der Einführung S-attributierter Definitionen beschrieben: Zusammen mit den Darstellungen der Grammatiksymbole auf dem Parserstack werden ihre Attribute verwaltet; bei einem Reduktionsschritt wird ein neues Symbol (das der linken Seite) auf den Parserstack gelegt, dessen Attribute aus denen der Symbole der rechten Seite berechnet werden. Die Behandlung synthetisierter Attribute bzw. S-attributierter Definitionen ist also kein Problem.

Noch offen ist die Frage, ob und wie ggf. L-attributierte Definitionen oder syntaxgesteuerte Übersetzungsschemata implementiert werden können, die für einige konkrete Übersetzungsprobleme dringend benötigt werden. Dabei gibt es zwei Aspekte:

1. Es werden auch den Symbolen der *rechten Seite* Attributwerte zugewiesen, die sich aus den Attributen weiter links stehender Symbole der rechten Seite sowie aus vererbten Attributen des Symbols der linken Seite ergeben können.
2. Übersetzungsaktionen sind in rechte Seiten eingebettet, müssen also ausgeführt werden, *bevor* eine komplette rechte Seite auf dem Stack liegt und die Reduktion ausgeführt wird.

Betrachten wir zunächst den zweiten Aspekt. Dies kann man relativ einfach durch eine Transformation des Übersetzungsschemas lösen, die dafür sorgt, dass Übersetzungsaktionen nur noch am Ende von rechten Seiten auftreten, also beim Reduktionsschritt ausgeführt werden können. Die Idee ist, sogenannte *Markierungs-Nichtterminale* einzuführen, die ins leere Wort ε ableiten und den einzigen Zweck haben, bei ihrer Reduktion eine Übersetzungsaktion auszulösen. Nehmen wir also an, es gebe eine Regel

$$A \quad \rightarrow \quad X \quad \{action_1\} \quad Y$$

Dann führen wir ein Markierungs-Nichtterminal M ein und eine Produktion $M \rightarrow \varepsilon$ und transformieren in ein neues Übersetzungsschema

$$\begin{array}{lcll} A & \rightarrow & X & M \quad Y \\ M & \rightarrow & \varepsilon & \{action_1\} \end{array}$$

Betrachten wir nun den ersten oben genannten Aspekt. Daran ist weniger die Tatsache problematisch, dass Attribute von weiter links stehenden Symbolen der rechten Seite benötigt werden; diese Symbole stehen ja bereits auf dem Stack, und ihre Attributwerte können benutzt werden. Das Problem entsteht dadurch, dass wir vererbte Attribute des *Symbols der linken Seite* benutzen wollen. Dieses Symbol ist aber noch nicht vorhanden; es wird erst dann auf dem Stack stehen, wenn die gerade aktuelle rechte Seite komplett bearbeitet ist, also zu spät. Außerdem erhält es seinen Wert ja nicht durch Auswertung dieser rechten Seite (dann wäre es ein synthetisiertes Attribut), sondern aus der Umgebung.

Auch dieses Problem kann man durch eine Transformation des Übersetzungsschemas lösen, jedenfalls dann, wenn die Grammatik vom Typ LL(1) ist. Wir nehmen zunächst der Einfachheit halber an, dass jedes Terminalsymbol T ein synthetisiertes Attribut $T.s$ besitzt (dessen Wert in der lexikalischen Analyse bestimmt wird) und jedes Nichtterminal A genau ein synthetisiertes Attribut $A.s$ und ein vererbtes Attribut $A.v$. Die Idee besteht dann wieder darin, Markierungs-Nichtterminale einzuführen, die nach ε ableiten, und zwar auf folgende Art. Für jede Produktion

$$A \rightarrow X_1 \ldots X_n$$

führen wir neue Markierungssymbole M_1, …, M_n ein und ersetzen sie durch die Produktion

$$A \rightarrow M_1 X_1 \ldots M_n X_n$$

Auf dem Parserstack speichern wir nun für jedes Nichtterminal X_i sein synthetisiertes Attribut $X_i.s$ zusammen mit der Darstellung des Symbols X_i, sein vererbtes Attribut $X_i.v$ aber mit der Darstellung des Symbols M_i !

Das bedeutet, dass nun während der Analyse einer aus X_i abgeleiteten rechten Seite, sagen wir mit der Produktion

$$X_i \rightarrow Y_1 \ldots Y_k ,$$

aus der durch die Transformation eine Produktion

$$X_i \rightarrow N_1 Y_1 \ldots N_k Y_k$$

geworden ist, bekannt ist, dass das vererbte Attribut von X_i, also $X_i.v$, auf dem Parserstack auf der Position links von N_1 steht (denn das ist die Position von M_i). Für die Berechnung eines vererbten Attributs von Y_j stehen also nun alle synthetisierten und vererbten Attribute von $Y_1 \ldots Y_{j-1}$ auf den $2 \cdot (j-1)$ Stackpositionen links von der obersten Stackposition zur Verfügung sowie das vererbte Attribut von X_i auf der Position $-2 \cdot (j-1) - 1$ (wenn wir die oberste Stackposition als Position 0 bezeichnen, die links davon als –1 usw.). Der Parserstack sieht also gerade so aus:

$$\ldots \quad M_i \quad N_1 \quad Y_1 \quad \ldots \quad N_{j-1} \quad Y_{j-1} \quad N_j$$

Das vererbte Attribut von Y_j wird ja bei der Reduktion mit der Regel $N_j \rightarrow \varepsilon$ berechnet und ist N_j zugeordnet. Die Berechnung eines synthetisierten Attributs, etwa des Attributs $X_i.s$, erfolgt bei der Reduktion mit der Regel $X_i \rightarrow N_1 Y_1 \ldots N_k Y_k$; auch hier sind die Stackpositionen aller synthetisierten und vererbten Attribute der Y_j bekannt.

Diese Technik kann man noch etwas verbessern:

1. Man führt Markierungssymbole nur ein für Nichtterminale, die tatsächlich vererbte Attribute besitzen. Dann muss man natürlich mit den Stackpositionen entsprechend mitrechnen, das ist aber machbar.
2. Falls das erste Symbol einer rechten Seite, also Y_1 in der Regel $X_i \rightarrow Y_1 \ldots Y_k$, ein vererbtes Attribut besitzt, das sich als $Y_1.v := X_i.v$ ergibt, so braucht man kein

Markierungssymbol N_1 für Y_1 einzuführen, da der benötigte Wert ja schon links von Y_1 auf dem Stack steht, nämlich in M_i. Das ist vor allem deshalb von Bedeutung, weil man bei linksrekursiven Regeln sonst Parse-Konflikte erzeugt.

Weitere Details hierzu findet man bei (Aho, Sethi und Ullman 1986, Abschnitt 5.6).

Auswirkung auf Yacc-Spezifikationen

In Ergänzung zu der Beschreibung in Abschnitt 3.3.4 ist es in Yacc-Spezifikationen erlaubt, semantische Aktionen innerhalb rechter Seiten (also nicht am Ende) auftreten zu lassen. Dabei darf auf Attributwerte von Symbolen links von der semantischen Aktion zugegriffen werden. Damit ist z. B. folgende Spezifikation erlaubt:

```
A       :       B       {$$ = 1;}
                C       {x = $2; y = $3;}
        ;
```

Yacc sorgt selbst durch Einsatz von Markierungssymbolen für die Auswertung eingebetteter semantischer Regeln. – Zu beachten ist, dass bei der Nummerierung der Symbole auf der rechten Seite die semantischen Aktionen mitgezählt werden und auch einen Wert liefern. Deshalb bezieht sich oben \$2 auf die Regel `{$$ = 1;}` und liefert den Wert 1; \$3 ist dem Symbol *C* zugeordnet.

Schließlich kann man die beschriebene Technik zur Behandlung vererbter Attribute in Yacc selbst implementieren, indem man auf Stacksymbole links von denen der aktuellen Regel zugreift, insbesondere auf das Symbol \$0, das unmittelbar links von \$1 steht. Dort kann man also ein vererbtes Attribut des Symbols der linken Seite unterbringen.

4.5 Literaturhinweise

Die Darstellung in diesem Kapitel orientiert sich an dem Buch von Aho et al. (2006), das das Thema natürlich sehr viel eingehender behandelt. Gute, weiterführende Darstellungen finden sich z. B. auch in (Wilhelm und Maurer 2007) und (Alblas und Nymeyer 1996).

Das Konzept der attributierten Grammatik stammt von Knuth (1968, 1971a). Dort werden auch vererbte Attribute und Abhängigkeitsgraphen eingeführt. L-attributierte Grammatiken wurden von Lewis, Rosenkrantz und Stearns (1974) definiert mit dem Ziel, die Auswertung der Attributdefinitionen gleichzeitig mit dem Aufbau das Ableitungsbaums vornehmen zu können; die gleiche Absicht liegt den in (Bochmann 1976) beschriebenen Baumdurchlauftechniken zugrunde. Verallgemeinerungen mit mehreren Baumdurchläufen wurden von Alblas (1981) untersucht. Die Konstruktion eines vorgreifenden Analysators mit rekursivem Abstieg

aus einem syntaxgesteuerten Übersetzungsschema gemäß Alg. 4.7 wird in ähnlicher Form in (Bochmann und Ward 1978) beschrieben.

Die von Kastens (1980) definierte Klasse der *geordneten attributierten Grammatiken* erlaubt es, für jedes Grammatiksymbol eine Ordnung auf der Menge seiner Attribute zu bestimmen, nach der dann Attributdefinitionen in jedem entsprechenden Knoten eines Ableitungsbaumes ausgewertet werden können. Diese Auswertungstechnik wurde von Engelfriet und Filé (1982) weiter analysiert. Einen Überblick zu weiteren Auswertungsstrategien bietet (Alblas 1991).

Unter anderem im Zusammenhang mit syntaxgesteuerten Editoren ist man daran interessiert, nach Transformationen des Ableitungsbaumes Attribute neu auszuwerten, wobei sich die Auswertung auf die betroffenen Teile beschränken sollte. Man bezeichnet das als *inkrementelle Attributauswertung*. Ein bekanntes System, das inkrementelle Auswertung unterstützt, ist der *Synthesizer Generator* (Demer, Reps und Teitelbaum 1981, Reps und Teitelbaum 1989). Ähnliche Funktionalität bietet das System OPTRAN (Lipps et al. 1988).

Überblicksartikel zum Gebiet der attributierten Grammatiken und dazugehörigen Systeme sind (Courcelle 1984, Engelfriet 1984, Deransart, Jourdon und Lorho 1988, Paakki 1995).

Kapitel 5

Übersetzung einer Dokument-Beschreibungssprache

Die nun folgenden drei Kapitel sind verschiedenen Anwendungsgebieten von Compilertechnik gewidmet. Kapitel 6 behandelt das klassische Thema der Implementierung imperativer Programmiersprachen, Kapitel 7 die Übersetzung funktionaler Sprachen. In diesem Kapitel beginnen wir bewusst mit einer weniger „klassischen“ Anwendung, nämlich der Übersetzung einer „Formatierungs-“ oder Markup-Sprache, in der zu setzende Dokumente in Form von ASCII-Texten beschrieben werden. Dafür gibt es zwei Gründe:

- Es ist wesentlich wahrscheinlicher, dass Sie die in diesem Kurs gewonnenen Kenntnisse praktisch einsetzen werden, indem Sie eine „kleine Sprache“ für eine spezielle Anwendung implementieren, als dass Sie einen ausgewachsenen Compiler für eine normale Programmiersprache schreiben. Das in diesem Kapitel gezeigte Beispiel steht stellvertretend für viele solcher Nicht-Standard-Anwendungen. Es geht also nicht darum, speziell etwas über die Übersetzung von Markup-Sprachen zu lernen. Es ist vielmehr wichtig, einen Blick dafür zu gewinnen, dass man manche Probleme relativ einfach lösen kann, indem man eine Sprache definiert und einen Compiler dafür schreibt.
- Wir wollen gern eine vollständige Compiler-Implementierung vorführen. Sie sollen sehen, wie die Werkzeuge Lex und Yacc zusammenarbeiten und wie sie in eine Umgebung eingebunden werden. Das ist nur im Rahmen einer solchen kleinen Anwendung möglich, bei der Quell- und Zielsprache relativ einfach sind. Sie sollen dabei auch lernen, den Aufwand für eine solche Implementierung einzuschätzen.

Im Folgenden motivieren wir zunächst kurz die Anwendung, erklären Quell- und Zielsprache und beschreiben die Komponenten des Systems einschließlich der Lex- und Yacc-Spezifikationen. Im Anhang A ist das komplette (in Englisch) dokumentierte und lauffähige Programm wiedergegeben.

5.1 Integration von Programmen und Dokumentation

Das Ziel der hier beschriebenen Entwicklung bestand darin, die Möglichkeiten zur Dokumentation von Programmen zu verbessern und insbesondere, Programme und

ihre Dokumentation integriert darzustellen; das heißt konkret: Programm und zugehörige Dokumentation sollten in der gleichen Datei stehen.

Einerseits möchte man für Dokumentation (also für Texte, die erklären, wie ein Programm funktioniert) die gleichen Möglichkeiten haben, wie für jeden anderen Text, also z. B. Überschriften, Einrückungen, Aufzählungen, Zeichnungen, verschiedene Schrifttypen usw. Diese Möglichkeiten werden von Textsystemen angeboten. Andererseits sind die Möglichkeiten, die übliche Programmiersprachen bzw. ihre Compiler dazu anbieten, sehr beschränkt: Das Programmfile muss ein ASCII-File sein; als Dokumentation kann man ASCII-Texte in Kommentarklammern einfügen, die vom Compiler ignoriert werden.

Eine offensichtliche, aber schlechte Lösung besteht darin, Programme wie üblich, also nur geringfügig kommentiert, zu schreiben und Dokumentation getrennt davon mit einem Textsystem anzufertigen. Das führt zu mehreren Problemen:

- Es gibt eine Tendenz, dass das lästige Schreiben von Dokumentation überhaupt unterbleibt, da es eine von der Programmentwicklung getrennte Aktivität ist. Es ist auch nicht leicht, zu erklären, auf welche Stelle des Programms man jeweils Bezug nimmt.
- Programme sind dynamisch und werden in der Entwicklung und später in der Wartung immer wieder geändert, Programmstücke neu zusammengebaut usw. Wenn die Dokumentation getrennt geführt wird, ist es so gut wie unausweichlich (oder nur mit strikten organisatorischen Maßnahmen zu verhindern), dass Programme und Dokumentation auseinanderdriften.

Einen prinzipiellen Ausweg aus dem Dilemma bieten Markup-Sprachen wie LaTeX oder HTML, die in einem ASCII-File den eigentlichen Text mit „Markup“ vermischen, also mit Markierungen, die anzeigen: „Das Folgende ist eine Überschrift“, „dieser Absatz ist einzurücken“ usw. Ein Beispiel für HTML wurde schon in der Einführung (Abschnitt 1.1) gezeigt. Eine einfache Idee besteht darin, ein Programmfile in zwei Arten von Abschnitten zu zerlegen, nämlich *Dokumentations-* und *Programmabschnitte*, getrennt durch öffnende und schließende Kommentarklammern der Programmiersprache. In Dokumentationsabschnitten steht dann markierter Text, z. B. in LaTeX, in Programmabschnitten eben normaler Programmcode. Ein solches File ist einerseits direkt übersetzbar; der Inhalt der Kommentarabschnitte interessiert den Compiler ja nicht. Andererseits kann man mit einem kleinen Präprozessor das File so umformen, dass das Formatiersystem (also z. B. das LaTeX/TeX-System) die Programmtexte als wörtlich wiederzugebende Textstücke interpretiert (dargestellt mit einem speziellen Schrifttyp, mit allen Einrückungen usw.) und so in der Lage ist, das Ganze hübsch zu formatieren.

So weit, so gut. Das einzige verbleibende Problem ist noch, dass es zwei Klassen von Menschen gibt: solche, die LaTeX (oder ähnliche Markup-Sprachen) mögen, und solche, die sie nicht mögen. Die Leute, die sie nicht mögen, stört, dass durch die Vermischung von Markup und Text der eigentliche Text nur noch schwer lesbar ist. Eingefleischte LaTeX-Fans sind für diese Tatsache allerdings blind geworden:

```
f\"{u}r sie ist es {\em genauso leicht\/}, LaTeX zu lesen, wie
f{\"{u}r andere Leute, ein Ausrufezeichen.
```

Um auch eine zufriedenstellende Lösung für die zweite Klasse von Menschen zu finden, ist die Idee nun, in den Dokumentationsabschnitten ein weniger aufdringliches, möglichst „implizites“ Markup zu verwenden. Man kann in ASCII-Files gewisse Textelemente auch ohne explizites Markup kenntlich machen. So wird jeder verstehen, dass

```
5.1 Integration von Programmen und Dokumentation
```

wohl eine Überschrift ist, und dass

```
  * Es gibt eine Tendenz, da[ss] das l[ae]stige Schreiben von
Dokumentation ...


  * Programme sind dynamisch und werden in der Entwicklung und
sp[ae]ter in der Wartung immer wieder ...
```

wohl eine Aufzählung von Punkten ist, wie auf der vorigen Seite. Diese Idee wird in der im folgenden beschriebenen Sprache der *PD-Texte* (**P**rogramme mit **D**okumentation) weiter verfolgt. Das in diesem Kapitel vorgestellte *PD-System* ist ein Compiler, der PD-Texte nach LaTeX übersetzt.

5.2 Die Quellsprache: PD-Texte

Wir kommen nicht sofort mit einer Grammatik daher, sondern definieren die Sprache zunächst nicht-formal, aus Anwendungssicht. Auf den ersten Blick ist es übrigens gar nicht so leicht zu sehen, dass es tatsächlich eine formale Sprache ist.

Die Grundstruktur eines PD-Textes (bzw. eines PD-Files) sind alternierende Dokumentations- und Programmabschnitte, getrennt durch Kommentarklammern, die jeweils allein in einer Zeile stehen.

```
(*************************************************************
*
Hier ist ein Doku-Abschnitt. Dies ist ein erster Absatz.

Ein zweiter Absatz darin.

*************************************************************
*)
(* normaler Programmtext, hier in Modula-2 *)
```

```
MODULE Example;

FROM ... IMPORT ...

(*
Ein zweiter Doku-Abschnitt.

*)
...
```

Man darf zu der Grundform der Klammern – in Modula-2 „(*“ und „*)“, in C „/*“ und „*/“ noch beliebig viele Sterne in der Zeile hinzufügen, um den Quelltext übersichtlich zu machen. Wir betrachten nun den Inhalt der Dokumentationsabschnitte.

Ein Textsystem bietet mindestens die folgenden Formatierungsmöglichkeiten:

1. Es gibt Textelemente wie normale Absätze, Überschriften, Aufzählungen usw. Derartige Elemente werden durch *Absatzformate* beschrieben.
2. Die Schrift selbst kann verschiedene Eigenschaften haben wie Schrifttyp, kursiv, fett, hochgestellt usw. Dies wird durch eine Reihe von *Zeichenformaten* beschrieben.
3. Schließlich möchte man *Sonderzeichen* wie „ü“, α, $\rightarrow$ usw. benutzen, die alle im ASCII-Alphabet nicht vorkommen, die man also auch irgendwie beschreiben muss.

Wir entwickeln jetzt möglichst wenig störende ASCII-Beschreibungen für diese drei Arten von Formatierung.

5.2.1 Absatzformate

Ein Absatz ist in einem PD-File beschrieben durch eine Folge von (nichtleeren) Zeilen, gefolgt von mindestens einer Leerzeile. Die gleiche Konvention gilt übrigens in LaTeX. Wir legen eine kleine Anzahl häufig benötigter „vordefinierter“ Absatzformate fest und bieten zusätzlich die Möglichkeit, weitere Absatzformate in einem PD-File zu definieren. Absatzformate werden beschrieben durch die ersten Zeichen (in der ersten Zeile) des Absatzes. Vordefinierte Absatzformate sind:

- Standard
- Überschriften (3 Ebenen)
- eingerückter Absatz
- nummerierte und nichtnummerierte Aufzählungen
- (Verweis auf einzubettende) Abbildung
- Programmzitat

Diese Formate werden durch folgende Konventionen beschrieben:

Jeder Absatz, der nicht durch eine der im Folgenden genannten speziellen Zeichenfolgen eingeleitet wird, ist ein *Standardabsatz* (*standard_paragraph*). Er wird im Blocksatz mit dem Standardzeichensatz gesetzt wie dieser Absatz hier auch.

```
Dies ist ein kleiner Standardabsatz. Die ihn abschliessende
Leerzeile lassen wir aus Platzgr[ue]nden weg.
```

Überschriften (*headings*) werden durch ein- oder zweiziffrige Zahlen und Punkte sowie ein darauffolgendes Leerzeichen gekennzeichnet.

```
1 Dies ist eine [Ue]berschrift der ersten Ebene


1.1 Zweite Ebene


3.12.6 Dritte Ebene
```

Ein *eingerückter Absatz* (*display*) wird durch ein Tabulatorsymbol oder durch 8 führende Leerzeichen gekennzeichnet (der Standard-Texteditor unter UNIX springt mit Tabulator-Zeichen jeweils auf durch 8 teilbare Positionen, also auf 8, 16, 24 usw., wenn wir die Positionen in der Zeile ab 0 nummerieren).

```
        Dies ist ein einger[ue]ckter Absatz. Man kann die
Folgezeilen auch einr[ue]cken, mu[ss] es aber nicht.
```

Es gibt zwei Arten von Aufzählungen (*lists*), nämlich nummerierte (*enumeration lists*), in denen jeder Absatz durch eine Ziffer eingeleitet wird, und nichtnummerierte (*itemized lists*), in denen jeder Absatz durch einen fetten Punkt eingeleitet wird. Beispiele für beide Arten sind auf den vorigen Seiten zu sehen. Darüber hinaus kann man Aufzählungen noch ineinander schachteln (aber nur bis zur Tiefe 2) und es kann zu einem Element einer Aufzählung Folgeabsätze (*followups*) geben. Wir kennzeichnen sie wie folgt:

```
  * Auf der ersten Ebene wird ein Absatz einer unnummerierten
Aufz[ae]hlung durch eine Gruppe von vier Zeichen (zwei
Leerzeichen, Stern, Leerzeichen) eingeleitet.


    Ein Folgeabsatz beginnt mit 4 Leerzeichen.


  * Abs[ae]tze nummerierter Aufz[ae]hlungen (dies ist selbst
keiner) werden durch zwei Leerzeichen, Ziffer, Leerzeichen oder
durch Leerzeichen, zwei Ziffern, Leerzeichen eingeleitet.


    1 Eine Ebene tiefer geht alles genauso, aber es gibt zwei
weitere f[ue]hrende Leerzeichen.
```

```
  17 Welche Ziffern gew[ae]hlt werden, spielt keine Rolle, da
das dahinterliegende Formatiersystem die Elemente selbst
nummeriert.

    Auch auf der zweiten Ebene kann es Folgeabs[ae]tze geben.
```

Alles dies würde etwa so gesetzt:

- Auf der ersten Ebene wird ein Absatz einer unnummerierten Aufzählung durch eine Gruppe von vier Zeichen (zwei Leerzeichen, Stern, Leerzeichen) eingeleitet.

  Ein Folgeabsatz beginnt mit 4 Leerzeichen.

- Absätze nummerierter Aufzählungen (dies ist selbst keiner) werden durch zwei Leerzeichen, Ziffer, Leerzeichen oder durch Leerzeichen, zwei Ziffern, Leerzeichen eingeleitet.

  1. Eine Ebene tiefer geht alles genauso, aber es gibt zwei weitere führende Leerzeichen.

  2. Welche Ziffern gewählt werden, spielt keine Rolle, da das dahinterliegende Formatiersystem die Elemente selbst nummeriert.

     Auch auf der zweiten Ebene kann es Folgeabsätze geben.

Abbildungen werden eingebettet durch einen Absatz, der mit zwei Tabulatorsymbolen oder 16 Leerzeichen beginnt (hier sollen die Punkte die Leerzeichen zeigen).

```
................Abbildung 1: Baumdarstellung eines
                Listenausdrucks [mytext.Figure1.eps]
```

Der Rest dieses Absatzes hat die Struktur <$Text_1$> : <$Text_2$> [<$Text_3$>]. $Text_1$ wird ignoriert (der Formatierer erzeugt automatisch einen Text „Figure *n*“), $Text_2$ wird als Bildunterschrift verwendet, und $Text_3$ ist der Name der Datei, in der die Abbildung in Form von „encapsulated postscript“ steht. Die Postscript-Datei muss man mit einem geeigneten Zeichenprogramm erzeugen.

Schließlich möchte man manchmal *Programmstücke* im Text *zitieren* (*program display*), also innerhalb der Dokumentationsabschnitte. Dies ist genaugenommen kein Absatzformat, da auch beliebig viele Leerzeilen innerhalb eines solchen Textstückes vorkommen können. Wir klammern diesen Text ein in jeweils 4 Bindestriche am Anfang der Zeile, also z. B.

```
----    PROCEDURE CreateSegment
          (VAR SegId : INTEGER       (* out *);
           VAR Error : SMErrorType   (* out *));
----
```

Dies würde formatiert fast genauso aussehen:

```
PROCEDURE CreateSegment
  (VAR SegId : INTEGER       (* out *);
   VAR Error : SMErrorType   (* out *));
```

Die horizontalen Linien in der formatierten Darstellung haben den Zweck, dies als Programmzitat kenntlich zu machen; ansonsten wäre es nicht unterscheidbar von einem „echten“ Programmstück, das auch der Compiler sieht (die Kommentarklammern, die Dokumentations- und Programmabschnitte trennen, sieht man nicht mehr in der formatierten Version).

Spezielle Absatzformate

Wenn dieser Grundbestand an Absatzformaten nicht ausreicht, kann der Benutzer des PD-Systems spezielle Absatzformate definieren. Absätze mit einem solchen Format werden im Text eingeleitet durch eine ein- oder zweiziffrige Zahl in eckigen Klammern.

```
[10] Dies k[oe]nnte zum Beispiel ein Absatz sein, der zentriert
darzustellen ist.
```

Das Absatzformat wird definiert – unter Rückgriff auf die Möglichkeiten der dahinterliegenden Formatiersprache, also z. B. LaTeX – durch eine Zeile am Anfang des PD-Files (im ersten Dokumentationsabschnitt) der Form:

```
//paragraph [10] centered:   [\begin{center}]    [\end{center}]
```

Das einleitende Schlüsselwort *paragraph* sagt, dass hier ein Absatzformat definiert wird, 10 ist die Nummer, *centered* der Name des Formats. Es folgt in zwei Paaren von eckigen Klammern LaTeX-Code, der *vor* den Absatz zu setzen ist, und LaTeX-Code, der *hinter* den Absatz zu setzen ist. Das PD-System wird den obigen Absatz aufgrund der Definition also konvertieren in einen Text

```
\begin{center}Dies k\"{o}nnte zum Beispiel ein Absatz sein, der
zentriert darzustellen ist.\end{center}
```

LaTeX sorgt dann für die gewünschte zentrierte Darstellung. – Es war geplant, das Format nicht nur über die Nummer, sondern auch über den Namen (*centered*) aufrufbar zu machen; dies ist aber bisher nicht implementiert.

5.2.2 Zeichenformate

Hier führen wir nur Möglichkeiten ein, *Kursivschrift* und **Fettdruck** direkt zu kennzeichnen, da diese Zeichenformate häufig gebraucht werden. Zusätzlich kann man spezielle Zeichenformate als Benutzer definieren, ähnlich wie für Absatzformate.

Um Zeichenformate zuzuweisen, muss man innerhalb von Absätzen Zeichenfolgen kennzeichnen, die dieses Format bekommen sollen. Dies geschieht durch Einklammern der Zeichenfolge auf besondere Art:

```
Wir notieren ~Kursivschrift~ durch Einklammern in Tilde-
Zeichen, *Fettdruck* durch Einklammern in Sterne.
"Benutzerdefinierte Zeichenformate"[1] werden in normale
Doppelanf[ue]hrungsstriche gesetzt, wobei am Ende eine
Zeichenformatnummer (wieder eine oder zwei Ziffern in eckigen
Klammern) folgt. Bei einer "wiederholten Verwendung" desselben
Formats kann man die Formatnummer weglassen.
```

Ein Zeichenformat wird ebenso definiert wie ein Absatzformat, nur mit einem Schlüsselwort *characters*. Auch hier wird einfach LaTeX-Code angegeben, der davor- bzw. dahinterzuhängen ist.

```
//characters [10] underlined:   [\underline{]    [}]
```

Manchmal möchte man die Zeichen ~, * und " natürlich auch mit ihrer normalen Bedeutung verwenden.

```
Wir erlauben es, sie entweder in eckigen Klammern zu schreiben,
also [~], [*], ["], in diesem Fall werden die Klammern vom
System bei der [Ue]bersetzung entfernt. Oder sie k[oe]nnen in
Leerzeichen eingeschlossen werden, also ~ , * , " ; die
Leerzeichen erscheinen dann auch in der Ausgabe.
```

Die doppelten Anführungsstriche braucht man übrigens normalerweise nicht, da in LaTeX und auch in PD-Texten Paare von einfachen Anführungsstrichen verwendet werden:

```
So kann man etwas ``zitieren´´.
```

5.2.3 Sonderzeichen

Sonderzeichen werden durch Zeichenfolgen in eckigen Klammern beschrieben, für „ü“, „ä“, „ö“ kam das oben schon in verschiedenen Beispielen vor. Entsprechende Definitionen am Anfang des PD-Files sehen so aus:

```
//[ae]          [\"{a}]
//[alpha]       [$\alpha $]
//[->]          [$\rightarrow $]
```

Man gibt also einfach zwei Paare von eckigen Klammern an. Das erste Paar enthält die Zeichenfolge, die man im Text verwenden will, das zweite Paar entsprechenden LaTeX-Code, der bei der Übersetzung dafür einzusetzen ist. – Damit sind PD-Texte als Quellsprache der Übersetzung vollständig beschrieben.

5.3 Die Zielsprache: LaTeX

Das Ziel dieses Abschnitts kann natürlich nicht sein, einen LaTeX-Kurs zu geben.[1] Wir wollen nur kurz zeigen, wie die in PD-Texten vorkommenden Elemente zu übersetzen sind. Ein LaTeX-File beginnt mit einem Vorspann, der allgemeine Parameter setzt und insbesondere einen Dokumentstil auswählt. Die Grundstruktur ist:

```
\documentstyle{article}
...
\begin{document}
...
\end{document}
```

Der für das Setzen von PD-Texten gewählte Vorspann findet sich in Anhang A in Abschnitt 6.1. Der eigentliche Text des Dokumentes wird durch `\begin{document}` und `\end{document}` eingeklammert. In diesem Teil ist also der Inhalt eines PD-Files unterzubringen. Allgemein ist ein LaTeX-File eine geschachtelte Klammerstruktur, mit Klammerungen ähnlich den gerade gezeigten.

Programmabschnitte sollten exakt so gesetzt werden wie im Quelltext, insbesondere Einrückungen, die die Schachtelungsstruktur des Programms zeigen. Es muss ein Zeichensatz verwendet werden, bei dem alle Zeichen gleich breit sind (also kein Proportionalfont). Außerdem wollen wir den Programmtext insgesamt einrücken und eine etwas kleinere Schriftgröße wählen. In LaTeX erreicht man alles dies so:

```
{\small \begin{quote} \begin{verbatim}
MODULE Example;

FROM ... IMPORT ...
...
\end{verbatim} \end{quote} }
```

Absatzformate. Für *Standardabsätze* ist gar nichts zu tun; sie sehen in LaTeX genauso aus. *Überschriften* werden so übersetzt:

```
\section{Dies ist eine \"{U}berschrift der ersten Ebene}
\subsection{Zweite Ebene}
\subsubsection{Dritte Ebene}
```

[1] Die offizielle Schreibweise des Wortes „LaTeX“ ist nicht leicht zu setzen, außer mit LaTeX selbst. Näherungsweise sieht es so aus, wie von uns geschrieben, die genaue Form findet sich z. B. im Anhang A in der Überschrift 6. Gesprochen „Latech“ .

Eingerückte Absätze:

```
\begin{quote}Dies ist ein einger\"{u}ckter Absatz. Man kann die
Folgezeilen auch einr\"{u}cken, mu{\ss} es aber nicht.

\end{quote}
```

Aufzählungen:

```
\begin{itemize}
\item Auf der ersten Ebene wird ein Absatz einer unnummerierten
Aufz\"{a}hlung durch eine Gruppe von vier Zeichen (zwei
Leerzeichen, Stern, Leerzeichen) eingeleitet.

Ein Folgeabsatz beginnt mit 4 Leerzeichen.

\item Abs\"{a}tze nummerierter Aufz\"{a}hlungen (dies ist
selbst keiner) werden durch zwei Leerzeichen, Ziffer,
Leerzeichen oder durch Leerzeichen, zwei Ziffern, Leerzeichen
eingeleitet.

\end{itemize}
```

Um nummerierte Aufzählungen zu erhalten, klammert man mit

```
\begin{enumerate} ... \end{enumerate}
```

Diese Darstellungen kann man auch ineinanderschachteln, um Aufzählungen auf der zweiten Ebene zu erhalten.

Abbildungen werden mit folgendem Code eingebettet:

```
\begin{figure}[htb]
\begin{center} \leavevmode
        \epsfbox{Figures/mytext.Figure1.eps}
\end{center}
        \caption{ Baumdarstellung eines Listenausdrucks}
\end{figure}
```

Programmzitate behandeln wir ähnlich wie Programmabschnitte. Wir erzeugen zusätzlich die kennzeichnenden horizontalen Linien (die genauen Maße ergeben sich nach etwas Experimentieren) und lassen die Einrückung weg, da dies im Quelltext normalerweise schon eingerückt geschrieben wird.

```
\hspace{0.9cm} \rule{2in}{0.1pt}
{\small \begin{verbatim}
        PROCEDURE CreateSegment
          (VAR SegId : INTEGER       (* out *);
           VAR Error : SMErrorType   (* out *));

\end{verbatim} }
\hspace{0.9cm} \rule{2in}{0.1pt}
```

Um benutzerdefinierte Absatzformate müssen wir uns hier nicht kümmern; der LaTeX-Code wird ja explizit in der Definition angegeben.

Zeichenformate. Auch hier sind nur die beiden vordefinierten Formate von Interesse. Dabei erzeugt der Befehl „\/“ etwas Abstand nach kursivem Text.

```
Wir notieren {\em Kursivschrift\/} durch Einklammern in Tilde-
Zeichen, {\bf Fettdruck} durch Einklammern in Sterne.
```

Sonderzeichen. Hier benötigen wir keine speziellen LaTeX-Kenntnisse; wir müssen nur den Mechanismus für die Einbettung des vom Benutzer angegebenen LaTeX-Codes implementieren.

Dies genügt als Beschreibung der Zielsprache unserer Übersetzung.

5.4 Entwurf des Übersetzers

Nachdem wir Quell- und Zielsprache kennen, können wir mit dem Entwurf des Übersetzers beginnen. Mit etwas Nachdenken findet man, dass folgende Fragen zu klären sind:

1. Wie sieht die Aufgabenverteilung zwischen lexikalischer Analyse und Parsen aus? Also welche lexikalischen Symbole werden benötigt, und wie kann man dann die Grammatik definieren?
2. Was soll das zu konstruierende System eigentlich aus Benutzersicht tun, und wie arbeitet es mit anderen Programmen zusammen (Systemumgebung)?
3. Kann man in den Übersetzungsaktionen vom Parser analysierte Textteile unmittelbar ausgeben, oder muss Text zunächst in irgendwelchen Datenstrukturen gesammelt werden?
4. Offenbar werden Datenstrukturen benötigt, um benutzerdefinierte Absatzformate, Zeichenformate und Sonderzeichen zu speichern. Wie sollen sie aussehen?

Die erste Frage ist offenbar die zentrale und technisch schwierigste Frage, die uns eine Weile beschäftigen wird. Denken wir zunächst über die einfacheren Fragen (2) bis (4) nach und anschließend über (1). Dies sind die Themen der folgenden Abschnitte.

5.4.1 Systemfunktion und Umgebung

Die Grundfunktion des Systems, das wir *PD-System* nennen, besteht darin, ein PD-File *pdfile* zu lesen und ein entsprechendes LaTeX-File *pdfile.tex* zu schreiben. Als Benutzer würde man gern durch einen Programmaufruf erreichen, dass das im PD-File beschriebene Dokument formatiert und am Bildschirm angezeigt wird. Das kann man erreichen, indem man für das erzeugte LaTeX-File *pdfile.tex* das LaTeX/TeX-System aufruft:

```
latex pdfile.tex
```

Als Ergebnis erhält man ein File *pdfile.dvi*. Weiterhin gibt es ein Programm, das *dvi*-Files am Bildschirm anzeigen kann, nämlich *xdvi*:

```
xdvi pdfile.dvi
```

Außerdem kann man *dvi*-Files drucken (bzw. in Postscript umwandeln und dem Drucker schicken) mit *dvips*:

```
dvips pdfile.dvi
```

Wir werden uns geeignete Kommandoprozeduren auf UNIX-Ebene ausdenken, die diese Befehle kombinieren, so dass z. B. ein Kommando

```
pdview pdfile
```

alle Schritte durchführt, bis das formatierte Dokument am Bildschirm zu sehen ist, und

```
pdprint pdfile
```

ein solches File formatiert ausdruckt.

Betrachten wir nun das Erzeugen des LaTeX-Files etwas genauer. In Abschnitt 5.3 haben wir gesehen, dass das LaTeX-File mit einem Vorspann beginnt, in dem gewisse Parameter gesetzt werden. Wenn wir die Schreibbefehle für den Vorspann fest in das Übersetzungsprogramm hineinschreiben, gibt es für den Benutzer keine Möglichkeit, an den Parametern etwas zu ändern. Besser ist es, wenn der Vorspann in einem File *pd.header* steht, das der Benutzer ändern kann, und von dort in das Zielfile kopiert wird. Dazu brauchen wir kein Programm zu schreiben; das tut das Systemkommando *cat*:

```
cat pd.header > pdfile.tex
```

Anschließend muss die Übersetzung des Eingabefiles *pdfile* dahintergeschrieben werden.

Wir müssen noch ein Problem lösen, das man allerdings erst entdeckt, wenn man mit dem hier konstruierten System zu arbeiten beginnt. Das Problem sind Tabulatorsymbole. Oben wurde schon erwähnt, dass der UNIX-Texteditor mit Tabsymbolen auf durch 8 teilbare Positionen springt. Leider weiß LaTeX nichts davon. Das hat den

Effekt, dass Programmtexte, in denen Tabsymbole vorkommen und die von LaTeX ja „verbatim“ (wörtlich) gesetzt werden sollen, in der formatierten Version ziemlich durcheinandergeraten. Die Lösung für dieses Problem sieht so aus, dass wir vor den Übersetzer einen kleinen Präprozessor schalten, ein Programm, das lediglich Tabsymbole in entsprechende Folgen von Leerzeichen umwandelt. Dieses Programm *tabs* findet sich in Anhang A.7.1. Wir können also jetzt davon ausgehen, dass die Eingabe für den Übersetzer keine Tabsymbole mehr enthält.

Noch eine andere Kleinigkeit: Wir wollen beim Schreiben von PD-Texten normalerweise in Absätzen keine Zeilenendesymbole benutzen (das Zeichen '\n'), damit der Zeilenumbruch vom Texteditor vorgenommen wird und auch nach Änderungen im Absatz noch stimmt. Andererseits sollte man dem (Zeilen-)Drucker nur Zeilen begrenzter Länge schicken. Wir schreiben dazu ein kleines Programm *linebreaks*, das ein File liest (egal ob *pdfile* oder *pdfile.tex*) und durch Einfügen von Zeilenumbrüchen beim Kopieren in ein Ausgabefile dafür sorgt, dass Zeilen nicht länger als 80 Zeichen werden. Dieses Programm ist im Anhang A.7.2 abgedruckt. Wir benutzen es einerseits, um PD-Files als Quelltext zu drucken, und andererseits, um die Zeilenlänge in den erzeugten LaTeX-Files zu begrenzen.

Unsere Strategie zum Erzeugen des LaTeX-Files sieht jetzt insgesamt so aus:

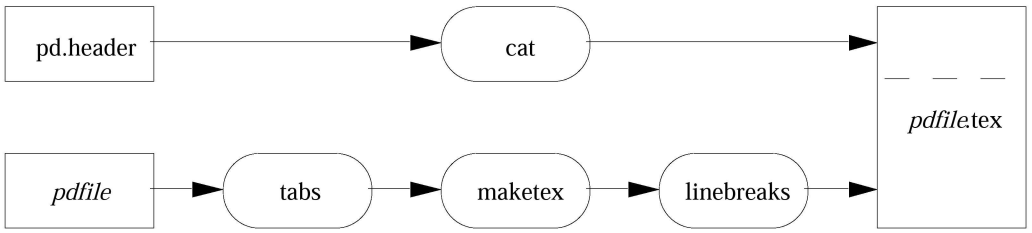

Abb. 5.1. Schritte beim Erzeugen des Files *pdfile*.tex aus *pdfile*

Dabei ist *maketex* das eigentliche Übersetzungsprogramm.

5.4.2 Text-Datenstrukturen

In Abschnitt 3.3.4 haben wir gesehen, wie man Übersetzungsaktionen in eine Yacc-Spezifikation aufnehmen kann. Die Frage ist nun, ob man mit einfacher sequentieller Ausgabe (*printf* -Befehlen) in den Übersetzungsaktionen auskommt. Das könnte vielleicht etwa so funktionieren:

```
chars   :
        | chars CHAR            {print($2);}
        ;
```

Nehmen wir an, dass die lexikalische Analyse jeweils ein einzelnes Zeichen als Token CHAR liefert (das Zeichen selbst steht als Attribut in $2), und sei *print* eine

Prozedur (Funktion in C), die ein Attributzeichen ausgibt. Dann würde diese Regel einen beliebig langen Text reduzieren und in der richtigen Reihenfolge ausgeben.

Dies wäre sicherlich die einfachste Strategie. Leider gibt es Situationen, in denen man Texte zunächst bis zum Ende lesen muss, bis man entscheiden kann, welcher LaTeX-Code *davor* zu schreiben ist. Ein Beispiel dafür sind benutzerdefinierte Zeichenformate.

```
"Hier steht irgendein beliebig langer Text"[5]; erst wenn die
Zeichenformatnummer 5 gelesen ist, kann der er[oe]ffnende
LaTeX-Code geschrieben werden.
```

Selbst wenn man dieses Problem nicht hätte, wäre es ziemlich mühsam, die Übersetzungsregeln so zu schreiben, dass bei Reduktionen Text stets unmittelbar ausgegeben wird. Wir benötigen also eine Datenstruktur, in der Text als Attribut eines Grammatiksymbols dargestellt werden kann. Die wesentliche Operation auf dieser Datenstruktur ist *Konkatenation* von Texten, da sich der Text des Symbols auf der linken Seite einer Regel meist als Konkatenation der Texte der rechten Seite ergibt. Damit können wir so vorgehen:

```
text    : chars                 {print($1);}
        ;
chars   :                       {$$ = empty();}
        | chars CHAR            {$$ = concat($1, $2);}
        ;
```

Hier ist *print* eine Funktion, die einen Wert des Attribut-Datentyps *Text* ausgibt, *concat* eine Funktion, die zwei Argumenttexte aneinanderhängt, und *empty* eine Funktion, die einen leeren Text erzeugt.

Eine Datenstruktur, mit der man effizient Texte darstellen und konkatenieren kann, ist ein Binärbaum, in dessen Blättern Textstücke („Textatome“) stehen (Abb. 5.2).

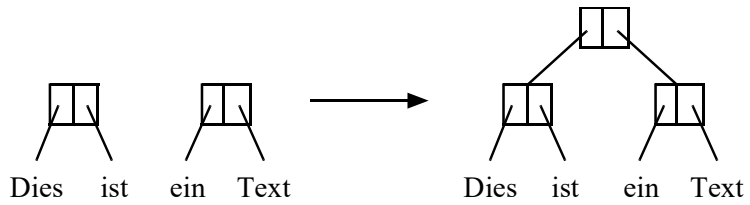

Abb. 5.2. Binärbaum zur Darstellung von Text

Um zwei Binärbäume zu konkatenieren, braucht man nur eine neue Wurzel zu erzeugen und die beiden gegebenen Bäume zu ihren Söhnen zu machen. Wir entwerfen die Datenstruktur als einen abstrakten Datentyp *listexpr* (das hat damit zu tun,

dass in der Sprache Lisp ähnliche Strukturen zur Darstellung geschachtelter Listen benutzt werden) und bieten folgende Operationen an:

atom:	*string* × *int*	→ *listexpr*
atomc:	*string*	→ *listexpr*
concat:	*listexpr* × *listexpr*	→ *listexpr*
print:	*listexpr*	→ ε
copyout:	*listexpr* × *int*	→ *string*
release-storage		

Die Funktion *atom* erzeugt ein Textatom, d. h. ein Blatt der Baumstruktur, aus einem String und einer Längenangabe. In der Implementierung wird der Typ *string* ein Zeiger in einen *array of char* sein. Die Funktion ist in dieser Form vor allem deshalb von Interesse, weil die lexikalische Analyse über die Variablen *yytext* und *yyleng* die gerade als lexikalisches Symbol erkannte Zeichenfolge bekanntmacht. Wir können also mit *atom*(*yytext, yyleng*) in der lexikalischen Analyse ein Textatom für das gerade erkannte Symbol erzeugen.

Die Funktion *atomc* ist eine Variante von *atom*, die die Länge selbst berechnet; hiermit kann man in Übersetzungsaktionen Atome für konstante Texte erzeugen, z. B.

```
atomc("\\subsection{")
```

Man beachte, dass das Backslash-Symbol „\“ innerhalb von Stringkonstanten als Escape-Symbol benutzt wird; um also einen Backslash selbst zu notieren (der in LaTeX häufig benötigt wird), muss man „\\“ schreiben. Das hier erzeugte Atom wird die Zeichenfolge

```
\subsection{
```

enthalten. Die Funktion *concat* ist klar, *print* schreibt einen als Baum dargestellten Text auf die Standardausgabe, *copyout* schreibt einen Text in einen Array (of char) und kontrolliert dabei, dass die im zweiten Parameter angegebene Schranke für die Länge des Textes (nämlich der vorhandene Platz im Array) nicht überschritten wird. *Release-storage* gibt sämtlichen Speicherplatz für innere Knoten und Textatome für alle existierenden Bäume frei.

Zur Implementierung benutzen wir einen großen Array für Knoten *nodespace* (jedes Element ist ein Knoten), in dem die Knoten von Index 0 bis *first-free-node* – 1 bereits vergeben sind; wann immer ein Knoten benötigt wird, belegen wir das Element *first-free-node* und erhöhen diesen Index um 1. Für die Speicherung der eigentlichen Zeichenfolgen der Textatome benutzen wir einen zweiten Array *text*; ein Blatt des Baumes enthält einen Text als Zeiger in den Array *text* plus Längenangabe.

Ein Wert des Datentyps *listexpr* wird damit beschrieben durch den Wurzelknoten seines Baumes, dieser wiederum durch einen Index in den Array *nodespace.* Der Datentyp *listexpr* wird damit technisch realisiert durch den Typ *integer*. Das ist sehr

praktisch, weil Attribute in Lex und Yacc ohne weiteres vom Typ *integer* sind. Wir können also Texte in Form von Zahlen an Grammatiksymbole zuweisen.

Die Implementierung des Datentyps *listexpr* findet sich in einem „Modul“ *NestedText*, in C dargestellt durch zwei Files *NestedText.h* und *NestedText.c*, die im Anhang A in den Abschnitten 2.1 und 2.2 wiedergegeben sind.

5.4.3 Datenstrukturen für benutzerdefinierte Formate und Zeichen

Datenstrukturen für benutzerdefinierte Absatzformate, Zeichenformate und Sonderzeichen können sehr einfach sein; es werden Tabellen benötigt, um die jeweiligen Informationen aus der Definition aufzubewahren. Wir brauchen also für Definitionen von Absatzformaten

```
//paragraph [10] centered:   [\begin{center}]    [\end{center}]
```

eine Tabelle

Formatnummer	*Formatname*	*Vorspann*	*Nachspann*
`10`	`centered`	`\begin{center}`	`\end{center}`
`...`	`...`	`...`	`...`

Das kann man einfach als Array definieren, in dem man sequentiell nach der Formatnummer sucht. Sequentielle Suche genügt, da nur wenige Formatdefinitionen existieren werden (und maximal 100, da nur zwei Ziffern erlaubt sind). In der Implementierung wird von hinten nach vorne gesucht. Dies hat mit der Absicht zu tun, Absatzformate über den Namen aufrufbar zu machen, was bisher nicht implementiert ist. Bei der gegenwärtigen Implementierung könnte man die Tabelle auch von vorne an durchlaufen.

Datenstrukturen für Zeichenformate und Sonderzeichen gehen analog. Diese Strukturen sind in einem File *ParserDS.c* implementiert, das in Anhang A.4 wiedergegeben ist.

5.4.4 Lexikalische Symbole und Grammatik

Wir kommen jetzt zum Kernproblem, nämlich dem Entwurf einer Grammatik für unsere Anwendung. Die spannende Frage dabei ist die Aufgabenverteilung zwischen lexikalischer Analyse und Syntaxanalyse, also die Frage, welche lexikalischen Symbole benötigt werden. Kandidaten für lexikalische Symbole (Token) sind offenbar alle Zeichen oder Zeichenfolgen, über die wir in der Spezifikation der Quellsprache (Abschnitt 5.2) explizit gesprochen haben, also die Folgenden:

`(**********`	öffnende Klammer Doku-Abschnitt
`**********)`	schließende Klammer Doku-Abschnitt
`\n\n`	Absatzende
`1 17.5 3.1.1`	Überschriften
`¬¬¬¬¬¬¬¬`	(8 Leerzeichen) eingerückter Absatz

`¬¬*¬`
`¬¬1¬`
`¬10¬`
`¬¬¬¬`
`¬¬¬¬*¬`
`¬¬¬¬1¬`
`¬¬¬10¬`
`¬¬¬¬¬¬`
	Elemente von Aufzählungen („¬“ stellt das Leerzeichen dar)

`¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬` `: [ ]`	(16 Leerzeichen) Abbildungen
`----`	Programmzitat
`[1] [20]`	spezielles Absatzformat

`~ [~] ¬~¬`
`* [*] ¬*¬`
`" ["] ¬"¬`
`[1] [20]`
	Zeichenformate und Escape-Mechanismus

`[ ]`	Sonderzeichen
`// [1] [20] : [ ]` `paragraph characters`	Formatdefinitionen des Benutzers

Es ist sehr instruktiv, zu betrachten, wie man die Interaktion zwischen lexikalischer Analyse und Syntaxanalyse *falsch* anlegen könnte (tatsächlich hat der Autor es im ersten Entwurf so gemacht). Man könnte versucht sein, z. B. eingerückte Absätze, die verschiedenen Arten von Aufzählungselementen und Abbildungen in Yacc so zu spezifizieren:

```
display         : "        " paragraph_rest
                ;
bulletpar1      : "  * " paragraph_rest
                ;
enum1           : "  " DIGIT " " paragraph_rest
                | " " DIGIT DIGIT " " paragraph_rest
                ;
```

```
followup1       : "    " paragraph_rest
                ;
```

Wenn man so vorgeht, meldet Yacc eine beträchtliche Anzahl von shift/reduce- und reduce/reduce-Konflikten. Das ist auch klar, denn die explizit in den Regeln notierten Zeichen (z. B. 8 Leerzeichen in der ersten Regel) kommen alle einzeln aus der lexikalischen Analyse. Die Vorausschau umfasst aber nur ein einziges Zeichen, also das erste Leerzeichen für alle diese Regeln, die damit sämtlich miteinander in Konflikt stehen. Wir müssen daher unbedingt die lexikalische Analyse so anlegen, dass die jeweiligen Anfänge dort zu einem einzigen Token zusammengefasst werden! Damit sehen die obigen Regeln etwa so aus:

```
display         : DISPLAY paragraph_rest
                ;
bulletpar1      : BULLET1 paragraph_rest
                ;
```

(usw.)

Wir betrachten im Folgenden die Yacc- und Lex-Spezifikationen der Dokumentstruktur, einiger Absatzformate und der Struktur von Aufzählungen.

Dokumentstruktur

Nehmen wir an, es gebe für die öffnenden und schließenden Klammern der Dokumentationsabschnitte Token OPEN und CLOSE. Dann können wir die Dokumentstruktur so definieren:

```
doc             : doc_section
                | doc program_section doc_section
                ;
doc_section     : OPEN elements CLOSE
                ;
```

Damit erreichen wir, dass ein PD-Dokument mit einem Doku-Abschnitt beginnen muss; es können dann jeweils Paare von Programm- und Doku-Abschnitten folgen. Ein Dokumentationsabschnitt beinhaltet auch seine öffnenden und schließenden Klammern, dazwischen können die verschiedenen Arten von Absätzen usw. stehen.

Die Token OPEN und CLOSE lassen sich in Lex so spezifizieren:

```
lbracket        ("(*"|"/*")
rbracket        ("*)"|"*/")
star            [*]
open            {lbracket}{star}*(" "*[\n])+
close           {star}*{rbracket}(" "*[\n])+
%%
```

```
^{open}          {return(OPEN);}
^{close}         {return(CLOSE);}
```

Die Definition erzwingt, dass *open*- und *close*-Symbole jeweils eine vollständige Zeile beinhalten; sie müssen am Anfang der Zeile beginnen, können nach *lbracket* und vor *rbracket* beliebig viele zusätzliche Sterne enthalten und beinhalten dann auch beliebig viele Leerzeichen bis zum Zeilenendesymbol. *Open* und *close* „verzehren“ darüber hinaus noch beliebig viele folgende Leerzeilen. Ihr Bereich (also die darin aufgesammelten Eingabezeichen) endet exakt vor dem ersten Zeichen einer Zeile, die nicht vollständig leer ist.

Wichtig ist hier, dass *jedes* Zeichen der Eingabe irgendwo verarbeitet werden muss. Wir wollen in Doku-Abschnitten nach der öffnenden Klammer Absätze verarbeiten; der Absatz wird jeweils mit dem ersten Zeichen seiner ersten (per Definition nichtleeren) Zeile beginnen; also müssen Leerzeilen davor von *open* aufgesammelt werden. Eine ähnliche Überlegung gilt für Leerzeilen am Anfang von Programmabschnitten, d. h. für *close*.

Nicht so schön ist, dass ein PD-Text nun mit einem Doku-Abschnitt enden muss; es wäre durchaus in Ordnung, mit einem Programmabschnitt zu schließen. Das lässt sich durch ein neues Startsymbol *document* und eine weitere Regel leicht erreichen:

```
document         : doc
                 | doc program_section
                 ;
```

Nach diesen Definitionen muss das erste Zeichen eines PD-Files eine öffnende Klammer eines Doku-Abschnittes sein, genauer: das erste Zeichen des lexikalischen Symbols *lbracket*, also „(“ oder „/“. Führende Leerzeilen ergeben einen Syntaxfehler, was etwas ärgerlich ist. Deshalb wollen wir führende Leerzeilen im ersten *open*-Symbol aufsammeln. Dies sollte allerdings nur am Anfang des Dokuments passieren. Wir erreichen das wie folgt: In der Syntaxanalyse modifizieren wir die Regel für *doc*, so dass am Anfang des Dokuments Leerzeichen bis zum ersten Zeilenendesymbol in einem Nichtterminal *space* aufgenommen werden.

```
doc              : space doc_section
                 | doc program_section doc_section
                 ;
space            :
                 | space ' '
                 ;
```

Das nun folgende Zeilenendesymbol, ggf. darauffolgende Leerzeilen und schließlich die öffnende Klammer des Doku-Abschnittes reduzieren wir in der lexikalischen Analyse mit einer Variante von *open*:

```
open2            ([\n]" "*)*[\n]{lbracket}{star}*(" "*[\n])+
%%
```

```
{open2}          {return(OPEN);}
```

Auch hier wird deutlich, dass man sich das Zusammenspiel von lexikalischer und Syntaxanalyse genau überlegen muss.

Wir betrachten nun die Programmabschnitte:

```
program_section :       {printf("{\\small \\begin{quote}
                        \\begin{verbatim}\n");}
                  chars
                        {printf("\n\\end{%s}
                        \\end{quote}}\n\n", "verbatim");}
                ;
chars           :
                | chars text_char       {print($2);}
                ;
```

Die Regel für *program_section* ist etwas unübersichtlich; auf der rechten Seite steht nur das Nichtterminal *chars* zwischen zwei semantischen Regeln, die den LaTeX-Vorspann bzw. Nachspann (vgl. Abschnitt 5.3) schreiben. Die *print*-Funktion stammt aus dem *NestedText*-Modul und druckt hier jeweils ein einzelnes Zeichen. Beide Yacc-Regeln zusammen schreiben Vorspann, Programmtext und Nachspann in der richtigen Reihenfolge. Die erste Regel ist übrigens ein Beispiel dafür, dass eine semantische Aktion auch innerhalb einer rechten Seite – hier am Anfang – auftreten darf (vgl. Abschnitt 4.4).

Die zweite semantische Aktion innerhalb von *program_section* ist deshalb so trickreich formuliert, damit man diesen Programmtext (des Parsers) selbst auch in LaTeX „verbatim“ setzen kann. Das Einzige, was nämlich innerhalb eines verbatim zu setzenden Textes nicht vorkommen darf, ist die Zeichenfolge

```
\end{verbatim}
```

Durch die gewählte Formulierung wird diese Folge zwar geschrieben, ist aber im Text für LaTeX nicht zu sehen. – Damit ist die äußere Struktur des PD-Files vollständig definiert; zu beschreiben bleibt die Struktur der Dokumentationsabschnitte.

Absatzformate

Inhalt von Dokumentationsabschnitten sind vor allem Absätze der verschiedenen vordefinierten Formate. Die Formate sollen anhand der ersten Zeichen des Absatzes erkannt werden. Jeder Absatz endet mit einem Zeilenendesymbol, gefolgt von einer Leerzeile; eine Leerzeile ist wieder eine möglicherweise leere Folge von Leerzeichen gefolgt von einem Zeilenendesymbol. Der Absatz sollte weitere folgende Leerzeilen mitverzehren (ähnlich wie oben *open* und *close*). Wir führen ein lexikalisches Symbol *epar* ein, um das Absatzende zu beschreiben:

```
epar            [\n]" "*[\n](" "*[\n])*
%%
{epar}          {yylval = atom(yytext, yyleng); return(EPAR);}
```

Die zum erkannten Symbol *epar* gehörige Eingabezeichenfolge wird als Textatom-Attribut dem Token EPAR zugewiesen (über die Variable *yylval*). Warum?

Der Grund ist, dass die lexikalische Analyse *epar*-Symbole auch innerhalb von Programmtexten erkennen wird. Dort sollten die entsprechenden Zeichenfolgen aber völlig unverändert in die Ausgabe übernommen werden. Wir müssen deshalb den Text in der lexikalischen Analyse erfassen und die Regel für *chars* erweitern:

```
chars           :
                | chars text_char       {print($2);}
                | chars EPAR            {print($2);}
                ;
```

Nach dem gleichen Prinzip werden wir jedes weitere benötigte Symbol der lexikalischen Analyse behandeln.

Ein Absatz wird also irgendeine spezielle Einleitung haben, gefolgt von beliebigem Text, und durch ein EPAR-Token abgeschlossen werden. Bei einem Standardabsatz fehlt die spezielle Einleitung. Ihn können wir so beschreiben:

```
standard_paragraph : paragraph_rest     {print($1);
                                         printf("\n\n");}
                ;
paragraph_rest  : text EPAR             {$$ = $1;}
                ;
```

Was genau *text* ist, ist natürlich noch zu definieren. Wie man sieht, wird in diesem Kontext die EPAR zugeordnete Zeichenfolge ignoriert; es werden stets zwei Zeilenendesymbole geschrieben, um den Absatz in der Ausgabe abzuschließen.

Es würde zu weit führen, hier die Spezifikationen sämtlicher Absatzformate zu besprechen. Wir betrachten als Beispiele nur Überschriften und eingerückte Absätze.

```
heading         : heading1
                | heading2
                | heading3
                ;
heading1        : HEAD1 paragraph_rest
                        {printf("\\section {"); print($2);
                        printf("}\n\n");}
                ;
```

```
heading2        : HEAD2 paragraph_rest
                        {printf("\\subsection {"); print($2);
                        printf("}\n\n");}
                ;
heading3        : HEAD3 paragraph_rest
                        {printf("\\subsubsection {");
                        print($2); printf("}\n\n");}
                ;
display         : DISPLAY paragraph_rest
                        {printf("\\begin{quote}\n");
                        printf("        "); print($2);
                        printf("\n\\end{quote}\n\n");}
                ;
```

Dabei sind die lexikalischen Symbole so definiert:

```
digit           [0-9]
num             ({digit}{digit}|{digit})
head1           {num}" "
head2           {num}"."{head1}
head3           {num}"."{head2}
display         "        "
%%
^{head1}        {yylval = atom(yytext, yyleng);
                return(HEAD1);}
^{head2}        {yylval = atom(yytext, yyleng);
                return(HEAD2);}
^{head3}        {yylval = atom(yytext, yyleng);
                return(HEAD3);}
^{display}      {yylval = atom(yytext, yyleng);
                return(DISPLAY);}
```

Auch hier werden die tatsächlichen Zeichenfolgen den lexikalischen Symbolen zugeordnet, um sie in anderen Kontexten ausgeben zu können. Zum Beispiel darf eine Folge „3.17“ natürlich auch als normaler Text innerhalb eines Absatzes vorkommen oder in einem Programm als Konstante auftreten.

Aufzählungen

Wir betrachten nur einen kleinen Teil davon, nämlich nichtnummerierte Aufzählungen auf der ersten Ebene; der Rest geht analog.

```
list            : itemized1
                        {printf("\\begin{itemize}\n");
                        print($1);
                        printf("\n\n\\end{itemize}\n\n");}
                | enum1
                        {printf("\\begin{enumerate}\n");
                        print($1);
                        printf("\n\n\\end{enumerate}\n\n");}
                ;
itemized1       : bulletitem1   {$$ = $1;}
                | itemized1 bulletitem1
                                {$$ = concat($1, $2);}
                ;
bulletitem1     : bulletpar1    {$$ = $1;}
                | bulletitem1 followup1
                                {$$ = concat($1, $2);}
                | bulletitem1 list2
                                {$$ = concat($1, $2);}
                ;
bulletpar1      : BULLET1 paragraph_rest
                        {$$ = concat(atomc("\n  \\item "),
                        concat($2, atomc("\n\n")));}
                ;
followup1       : FOLLOW1 paragraph_rest
                        {$$ = concat($1, concat($2,
                        atomc("\n\n")));}
                ;
```

Eine nichtnummerierte Liste der ersten Ebene (*itemized1*) besteht aus einer Folge von *bulletitem1*. Jedes solche Element hat mindestens einen ersten Absatz mit Einleitung „ * “ (*bulletpar1*); darauf folgen können beliebig viele *followup1*-Absätze oder hineingeschachtelte Listen der 2. Ebene (*list2*). Die lexikalischen Symbole dazu sind:

```
bullet1         "  * "
follow1         "    "
%%
^{bullet1}      {yylval = atom(yytext, yyleng);
                return(BULLET1);}
^{follow1}      {yylval = atom(yytext, yyleng);
                return(FOLLOW1);}
```

Abschließende Bemerkungen

Die übrigen Teile der Grammatik wollen wir nicht mehr im Einzelnen besprechen. Die komplette Spezifikation des Parsers findet sich in Abschnitt 5 des Anhangs A, die Spezifikation der lexikalischen Analyse in Abschnitt A.3. Sie können sich dort alle interessierenden Mechanismen im Detail ansehen. Die folgenden Hinweise können dabei nützlich sein:

Die Struktur von Texten, die in Absätzen und dort in bestimmten Zusammenhängen auftreten können, ist relativ komplex zu beschreiben (Abschnitt A.5.7 bis A.5.10). Wir erklären kurz die dort auftretenden syntaktischen Kategorien:

- *text* ist eine Folge, die mit einem *start_elem* beginnt; es dürfen *start_elems* und *follow_elems* folgen.
- *follow_elem* ist ein lexikalisches Symbol, das innerhalb eines Absatzes vorkommen darf und dort keine besondere Interpretation hat; es darf aber nicht am Anfang von *paragraph_rest* auftreten, da sonst Konflikte entstehen. Wenn z. B. ein HEAD1-Symbol am Anfang von *paragraph_rest* stehen dürfte, gäbe es einen Konflikt, da man dann *paragraph_rest* auch zu einem *standard_paragraph* reduzieren könnte.
- *start_elem*: lexikalisches Symbol, das auch am Anfang von *paragraph_rest* stehen darf.
- *text_char*: normales Zeichen im Absatz.
- *ftext_char*: Zeichen, die in Abbildungsunterschriften verwendet werden können. Das sind alle normalen Zeichen außer dem Doppelpunkt.
- *emphasized* und *bold_face* bezeichnen Kursivschrift und Fettdruck. Innerhalb der Klammern darf dann nicht noch einmal *emphasized* oder *bold_face* stehen, da sonst der Parser beim Auftreten der schließenden Klammer (also des zweiten „~“ oder „*“-Zeichens) einen shift/reduce-Konflikt bekommt und sich für shift entscheidet. Deshalb die etwas mühsame Konstruktion, innerhalb der Klammern eine *unemph_list* (Liste von Material außer *emphasized*) bzw. *unbold_list* zu verwenden.
- *plain_list*: Material, das innerhalb eines benutzerdefinierten Zeichenformats vorkommen darf.
- *btext*: Text, der innerhalb eckiger Klammern (Sonderzeichendefinitionen) stehen darf. Er darf eckige Klammern enthalten, aber nur korrekt geklammert.
- *btext2*: wie *btext*, aber zur Verwendung in Formatdefinitionen mit anderen semantischen Regeln.

Formatdefinitionen des Benutzers werden durch Yacc-Regeln in ihre Bestandteile zerlegt; diese werden mit einer Funktion *enter_def* in die in Abschnitt 5.4.3 skizzierten Tabellen-Datenstrukturen eingetragen (Anhang A.5.4). Beim Erkennen spezieller Absatzformate oder Zeichenformate wird mit einer Funktion *lookup_def* in der Tabelle nachgesehen, ob eine entsprechende Formatnummer existiert und ggf. der Index in der Tabelle zurückgegeben. Der Absatz oder die Zeichenfolge wird dann mit dem in der Tabelle nachgeschlagenen Vorspann bzw. Nachspann eingeklammert (Anhang A, Abschnitte 5.5.3 und 5.8). Analog verfährt man mit

Definitionen von Sonderzeichen (Anhang A.5.4 und A.5.9); hier heißen die Funktionen zum Eintragen und Nachschlagen *enter_schar* und *lookup_schar* (*schar* = ***s**pecial **char**acter*).

Abschnitt 9 des Anhangs A enthält das sogenannte *Makefile* für das gesamte PD-System und seine Dokumentation. In einem Makefile kann man beschreiben, welche Komponenten eines Programmsystems von welchen anderen Komponenten abhängen und was zu tun ist, wenn irgendeines der beteiligten Files sich ändert. Beispielsweise hängt das C-Programm *lex.yy.c* von der Spezifikation *PDLex.l* und auch von den Datenstrukturen in *PDNestedText.h* ab. Wenn z. B. *PDLex.l* geändert wird, muss mit dem Kommando

```
lex PDLex.l
```

das File *lex.yy.c* neu erzeugt werden.

Die Dokumentation für das gesamte Programmsystem wird erstellt, indem mit dem Systemkommando *cat* alle Files, die Quelltexte, Spezifikationen, reine Dokumentation, Kommandoprozeduren usw. enthalten, in geeigneter Reihenfolge verkettet, also in ein einziges File geschrieben werden. Das so erzeugte File ist *docu*, ein PD-File, das exakt den PD-Text für den Anhang A enthält. Der Anhang lässt sich nach Aufruf von *make*, das das Makefile auswertet, durch die Folge von Kommandos

```
pdview docu
pdprint docu
```

erzeugen. Sie finden im Anhang A also nicht nur eine Beschreibung des PD-Systems, von der das tatsächliche Programm inzwischen durch ein paar Änderungen etwas abweicht, sondern Sie sehen exakt das vollständige Programm, das auch der Compiler sowie Lex und Yacc verarbeiten. Diese Einheit von Programm und Dokumentation war ja das in Abschnitt 5.1 beschriebene Ziel.

Anhang B zeigt einen Teil des Quelltextes der Dokumentation, nämlich das File *PDNestedText.h*; dies entspricht dem Abschnitt 2.1 des Anhangs A. Anhang C enthält das vom PD-System daraus mit dem Befehl

```
pd2tex PDNestedText.h
```

erzeugte LaTeX-File *PDNestedText.h.tex*.

5.5 Literaturhinweise

Das PD-System wurde von einem der Autoren dieses Kurses entworfen und implementiert; der Bericht (Güting 1995) ist so etwas wie ein „Benutzerhandbuch“ dazu. Es wird in der Software-Entwicklung am Lehrgebiet „Datenbanksysteme für neue Anwendungen“ der FernUniversität Hagen regelmäßig zur Programmdokumentation eingesetzt, insbesondere in Abschlussarbeiten. Über die hier

gezeigte Implementierung hinaus wurden noch Varianten des Parsers geschrieben, die PD-Files nach HTML sowie nach ASCII übersetzen (wobei letzteres einem Entfernen des Markups entspricht). Da man hier nur Ausgabeanweisungen in den Yacc-Spezifikationen in die neue Zielsprache umsetzen muss, ist eine solche Erweiterung mit sehr wenig Aufwand, d. h. in einigen Stunden, zu realisieren. Das PD-System ist frei verfügbare Software; man kann es über die Web-Adresse

```
http://dna.fernuni-hagen.de/software.html
```

herunterladen. Derzeit läuft es unter UNIX und Linux.

Kurze Beschreibungen zu Lex und Yacc finden sich z. B. in den Übersetzerbau-Büchern von Aho et al. (2006) und Parsons (1992). Speziell Lex und Yacc gewidmete Bücher sind (Levine, Mason und Brown 1992) und (Herold 1995). Gute Einführungen zu LaTeX bieten (Lamport 1986) und (Kopka 1996).

Das Buch von Schmitz (1995) betont, ähnlich wie wir, Anwendungen von Compilertechnik in anderen Bereichen als der klassischen Implementierung von Programmiersprachen. Dort wird u. a. die Übersetzung einer kleinen Sprache zur Beschreibung von mathematischen Formeln (ein modifizierter Ausschnitt aus LaTeX) nach Postscript vorgeführt.

Lösungen zu den Selbsttestaufgaben

Aufgabe 4.1

Wir führen zwei Attribute ein. Das Attribut *val* enthält stets eine Zahl, das zweite Attribut *len* gibt die Anzahl der Ziffern dieser Zahl an. Wir benötigen *len*, um in einfacher Weise den Nachkommateil der Zahl umzuformen.

Produktion	*Semantische Regel*
$R \rightarrow S_1 B_1 . B_2 \mathbf{e} S_2 B_3$	$R.val := S_1.val * (B_1.val + B_2.val / 10^{B_2.len}) * 10^{(S_2.val * B_3.val)}$
$S \rightarrow +$	$S.val := 1$
$S \rightarrow -$	$S.val := -1$
$B \rightarrow B_1$ **ziffer**	$B.val := B_1.val * 10 +$ **ziffer**.*lexval* $B.len := B_1.len + 1$
$B \rightarrow$ **ziffer**	$B.val :=$ **ziffer**.*lexval* $B.len := 1$

Da ausschließlich synthetisierte Attribute auftreten, liegt hier eine S-attributierte Definition vor.

Aufgabe 4.2

(a) Die Grammatik G ist linksrekursiv, also nicht LL(1). Eine äquivalente LL(1)-Grammatik ist $G_1 = (\{R, B, S, F\}, \{$ **ziffer**, **.**, **e**, +, –$\}, P_1, R)$ mit

$$
\begin{array}{lll}
P_1 = \{ & R \rightarrow & SB\mathbf{.}B\mathbf{e}SB \\
& S \rightarrow & + \\
& S \rightarrow & - \\
& B \rightarrow & \mathbf{ziffer}\ F \\
& F \rightarrow & \mathbf{ziffer}\ F \\
& F \rightarrow & \varepsilon \qquad \}
\end{array}
$$

(b) Seien *sval* und *slen* synthetisierte und *vval* und *vlen* vererbte Attribute von F. Dann erhält man das folgende Übersetzungsschema:

$R \rightarrow S_1B_1\mathbf{.}B_2\mathbf{e}S_2B_3$ { $R.val := S_1.val * (B_1.val + B_2.val / 10^{B_2.len}) * 10^{(S_2.val * B_3.val)}$ }

$S \rightarrow +$ { $S.val := 1$ }

$S \rightarrow -$ { $S.val := -1$ }

$B \rightarrow$ **ziffer** { $F.vval :=$ **ziffer**$.lexval$; $F.vlen := 1$ }

F { $B.val := F.sval$; $B.len := F.slen$ }

$F \rightarrow$ **ziffer** { $F_1.vval := F.vval * 10 +$ **ziffer**$.lexval$; $F_1.vlen := F.vlen + 1$ }

F_1 { $F.sval := F_1.sval$; $F.slen := F_1.slen$ }

$F \rightarrow \varepsilon$ { $F.sval := F.vval$; $F.slen := F.vlen$ }

Aufgabe 4.3

Wir benötigen zunächst die Steuermengen der Produktionen. Anwendung der Verfahren aus Kapitel 3 liefert uns

Produktionen			Steuermenge
R	→	*A*.*B*e*A*	{+, –}
A	→	*SB*	{+, –}
S	→	+	{+}
	→	–	{–}
B	→	**ziffer** *F*	{**ziffer**}
F	→	**ziffer** *F*	{**ziffer**}
	→	ε	{., **e**, $}

Nun wenden wir Algorithmus 4.7 an. Da wir zwei Attribute übergeben müssen, definieren wir zunächst

```
type  attr_rec =  record
                     val: real;
                     len: integer;
                  end;
      attr    =  pointer to attr_rec;
```

und erhalten dann:

```
function R: attr;
varRval, A1val, A2val, Bval: real;
   Blen: integer;
   help: attr;
begin
   if symbol = + or symbol = – then
      help := A; A1val := help.val;
      match(.)
      help := B; Bval := help.val; BLen := help.len;
      match(e)
      help := A; A2val := help.val;
      Rval := sign(A1val) * (abs(A1val) + Bval / 10^Blen) * 10^A2val;
      help.val := Rval;
      return help
   else error
   fi
end;
```

```
function A: attr;
var Aval, Sval, Bval: real;
    Alen, Blen: integer;
    help: attr;
begin
    if symbol = + or symbol = – then
        help := S; Sval := help.val;
        help := B; Bval := help.val; Blen := help.len;
        Aval := Sval * Bval; Alen := Blen;
        help.val := Aval; help.len := Alen;
        return help
    else error
    fi
end;

function S: attr;
var Sval: real;
    help: attr;
begin
    if symbol = + then
        match(+);
        Sval := 1; help.val := Sval;
        return help
    elsif symbol = – then
        match(–);
        Sval := –1; help.val := Sval;
        return help
    else error
    fi
end;

function B: attr;
varBval, Fsval, Fvval, zifferlexval: real;
    Blen, Fslen, Fvlen: integer;
    vhelp, help: attr;
begin
    if symbol = ziffer then
        zifferlexval := symbol.lexval;
        match(ziffer);
        Fvval := zifferlexval;
        Fvlen := 1;
        vhelp.val := Fvval; vhelp.len := Fvlen;
        help := F(vhelp);
        Fsval := help.val; Fslen := help.len;
        Bval := Fsval; Blen := Fslen;
```

```
        help.val := Bval; help.len := Blen;
        return help
    else error
    fi
end;

function F( VF: attr ): attr;
var F1vval, Fsval, F1sval, zifferlexval: real;
    F1vlen, Fslen, F1slen: integer;
    help, vhelp: attr;
begin
    if symbol = ziffer then
        zifferlexval := symbol.lexval;
        match(ziffer);
        F1vval := VF.val * 10 + zifferlexval;
        F1vlen := VF.len + 1;
        vhelp.val := F1vval; vhelp.len := F1vlen;
        help := F(vhelp);
        F1sval := help.val; F1slen := help.len;
        Fsval := F1sval; Fslen := F1slen;
        help.val := Fsval; help.len := Fslen;
        return help
    elsif symbol = . or symbol = e or symbol = $ then
        Fsval := VF.val; Fslen := VF.len;
        help.val := Fsval; help.len := Fslen;
        return help
    else error
    fi
end;
```

Die angegebenen Prozeduren sind hier zur Verdeutlichung sehr ausführlich dargestellt und können leicht optimiert werden.

VI Lösungen zu den Selbsttestaufgaben

Literatur

Aho, A.V., Hopcroft, J.E. und Ullman, J.D. (1983). *Data Structures and Algorithms*. Addison-Wesley, Reading, MA.

Aho, A.V., Lam, M.S., Sethi, R. und Ullman, J.D. (2006). *Compilers: Principles, Techniques, and Tools*. 2nd Edition, Addison-Wesley, Reading, MA.

Alblas, H. (1981). A Characterization of Attribute Evaluation in Passes. *Acta Informatica 16*, 427-464.

Alblas, H. (1991). Attribute Evaluation Methods. In: (Alblas und Melichar 1991), S. 48-113.

Alblas, H. und Melichar, B., Hrsg. (1991). *Proceedings of the International Summer School SAGA: Attribute Grammars, Applications, and Systems*. Lecture Notes in Computer Science 545, Springer-Verlag, Berlin.

Alblas, H. und Nymeyer, A. (1996). *Practice and Principles of Compiler Building with C*. Prentice-Hall International, London, UK.

Bochmann, G.V. (1976). Semantic Evaluation from Left to Right. *Communications of the ACM 19*, 55-62.

Bochmann, G.V. und Ward, P. (1978). A Compiler Writing System for Attribute Grammars. *The Computer Journal 21*, 144-148.

Courcelle, B. (1984). Attribute Grammars: Definitions, Analysis of Dependencies. In: (Lorho 1984).

Demer, A., Reps, T. und Teitelbaum, T. (1981). Incremental Evaluation for Attribute Grammars with Application to Syntax-directed Editors. *8th ACM Symposium on Principles of Programming Languages*, S. 105-116.

Deransart, P., Jourdan, M. und Lorho, B., Hrsg. (1988). *Attribute Grammars, Definitions, Systems, and Bibliography.* Lecture Notes in Computer Science 323, Springer-Verlag, Berlin.

Engelfriet, J. (1984). Attribute Grammars: Attribute Evaluation Methods. In: (Lorho 1984).

Engelfriet, J. und Filé, G. (1982). Simple Multi-Visit Attribute Grammars. *Journal of Computer and System Sciences 24*, 283-314.

Güting, R.H. (1995). Integrating Programs and Documentation. FernUniversität Hagen, Informatik-Report 182.

Herold, H. (1995). *lex und yacc: Lexikalische und syntaktische Analyse*. 2. Aufl., Addison-Wesley (Deutschland), Bonn.

Kastens, U. (1980). Ordered Attribute Grammars. *Acta Informatica 13*, 229-256.

Knuth, D.E. (1968). Semantics of Context-free Languages. *Mathematical Systems Theory 2*, 127-145.

Knuth, D.E. (1971a). Semantics of Context-free Languages, Correction. *Mathematical Systems Theory 5*, 95-96.

Kopka, H. (1996). *LaTeX, Band 1: Einführung*. 2. Aufl., Addison-Wesley (Deutschland), Bonn.

Lamport, L. (1986). *LATEX. A Document Preparation System*. Addison-Wesley, Reading, MA.

Levine, J.R., Mason, T. und Brown, D. (1992). *lex & yacc*. 2nd Edition, O'Reilly & Associates, Sebastopol.

Lewis, P.M. II, Rosenkrantz, D.J. und Stearns, R.E. (1974). Attributed Translations. *Journal of Computer and System Sciences 9*, 279-307.

Lipps, P., Olk, M., Möncke, U. und Wilhelm, R. (1988). Attribute (Re)evaluation in the OPTRAN System. *Acta Informatica 26*, 213-239.

Lorho, B., Hrsg. (1984). *Methods and Tools for Compiler Construction*. Cambridge University Press, 1984.

Paakki, J. (1995). Attribute Grammar Paradigms – A High Level Methodology in Language Implementation. *ACM Computing Surveys 27*, 196-255.

Reps, T. und Teitelbaum, T. (1989). *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, Berlin.

Schmitz, L. (1995). *Syntaxbasierte Programmierwerkzeuge*. Teubner-Verlag, Stuttgart.

Sedgewick, R. (1990). *Algorithms in C*. Addison-Wesley, Reading, MA.

Wilhelm, R. und Maurer, D. (2007). *Übersetzerbau: Theorie, Konstruktion, Generierung*. 2. Aufl., Springer-Verlag, Berlin.

Index

Anhang A. Das dokumentierte PD-System

The PD System: Integrating Programs and Documentation

Ralf Hartmut Güting

May 1995

1 Overview

The purpose of *PDSystem* is to allow a programmer to write ASCII program files with embedded documentation (*PD* stands for *Programs with Documentation*). Such files are called *PD files*. Essentially a PD file consists of alternating *documentation sections* and *program sections*. Within documentation sections, one can describe a number of paragraph formats (such as headings, displayed material, etc.), character formats (e.g. italics), and special characters (e.g. "ü"). How this is done, is described in the document "Integrating Programs and Documentation" [Gü95].

The main component of PDSystem is an executable program *maketex* which converts a PD file into a LaTeX file. More precisely, given a file *pdfile*, a LaTeX file *pdfile.tex* is created as follows: First, a standard header for LaTeX is copied from a file *pd.header* into *pdfile.tex*. Then, *pdfile* is first run through a filter program called *pdtabs* which converts tabulator symbols into corresponding sequences of blanks. The output of this filter is fed into *maketex* (which can therefore be sure not to encounter any tab symbols) which converts material in documentation sections into LaTeX code and encloses program sections by "verbatim" commands to force TeX to typeset them exactly as they have been written.

A PD file may contain very long lines, because CR (end of line) symbols need only be present to delimit paragraphs. To make the output file *pdfile.tex* easily printable, the output of *maketex* is run through a further filter called *linebreaks* which introduces CR symbols such that no lines with more than 80 characters are in the output. In total, we have the processing steps illustrated in Figure 1.

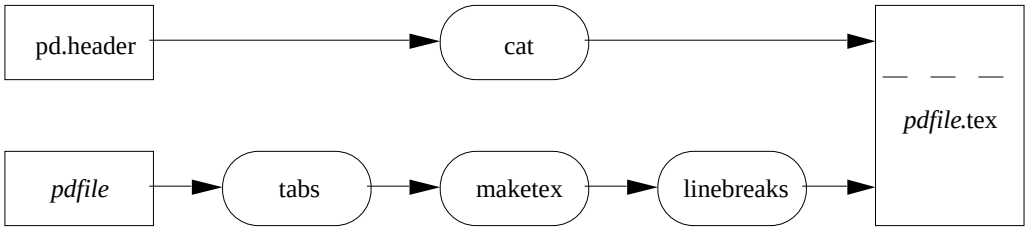

Figure 1: Construction of a TeX file from a PD file

The file *pd.header* is shown in Section 6.1. *Cat* is a UNIX system command. Executables *pdtabs* and *linebreaks* are created from the corresponding C programs *pdtabs.c* and *linebreaks.c* shown in Section 7. The programs leading to *maketex* are described below. The complete process shown in Figure 1 is executed by a command procedure called *pd2tex* given in Section 8.2.

There is a further command procedure:

- *pdview* allows one to view a PD file under the *yap* viewer (after it has been processed by LaTeX, Section 8.1).

We now consider the construction of *maketex*, the central component of the PD system. *Maketex* depends on the components shown in Figure 2.

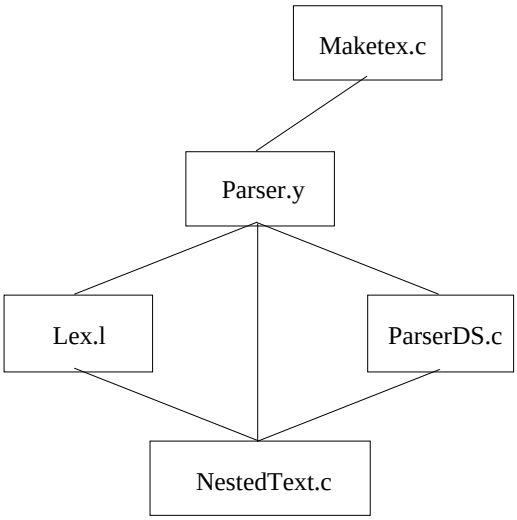

Figure 2: Components for building *maketex*

These components play the following roles:

- *Lex.l* is a *lex* specification of a lexical analyser (which is transformed by the UNIX tool *lex* into a C program *lex.yy.c*). The lexical analyser reads the input PD file and produces a stream of tokens for the parser.
- *Parser.y* is a *yacc* specification of a parser (transformed by the UNIX tool *yacc* into a C program *y.tab.c*). The parser consumes the tokens produced by the lexical analyser. On recognizing parts of the structure of a PD file, it writes corresponding LaTeX code to the output file.
- *NestedText.c* is a "module" in C providing a data structure for "nested text" together with a number of operations. This is needed because text for the output file cannot always be created sequentially. Sometimes it is necessary, for example, to collect a piece of text from the source file into a data structure and then to create enclosing pieces of LaTeX code before and after it. The *NestedText* data structure corresponds to a LISP "list expression" and is a binary tree with character strings in its leaves. There are operations available to create a leaf from a string, to concatenate two trees, or to write the contents of a tree in tree order to the output.
- *ParserDS.c* contains a number of data structures needed by the parser. These data structures are used to keep definitions of special paragraph formats, special character formats, and special characters, which can be defined in header documentation sections of PD files.

- *Maketex.c* is the main program. It does almost nothing. It just calls the parser; after completion of parsing (which includes translation to Latex) a final piece of Latex code is written to the output.

Figure 2 describes the dependencies among these components at a logical level. An edge describes the "uses" relationship. For example, the *NestedText* module is used in the lexical analyser, the parser and in the parser data structure component. Figure 3 shows these dependencies at a more technical level, as they are reflected in the *makefile* (see Section 9).

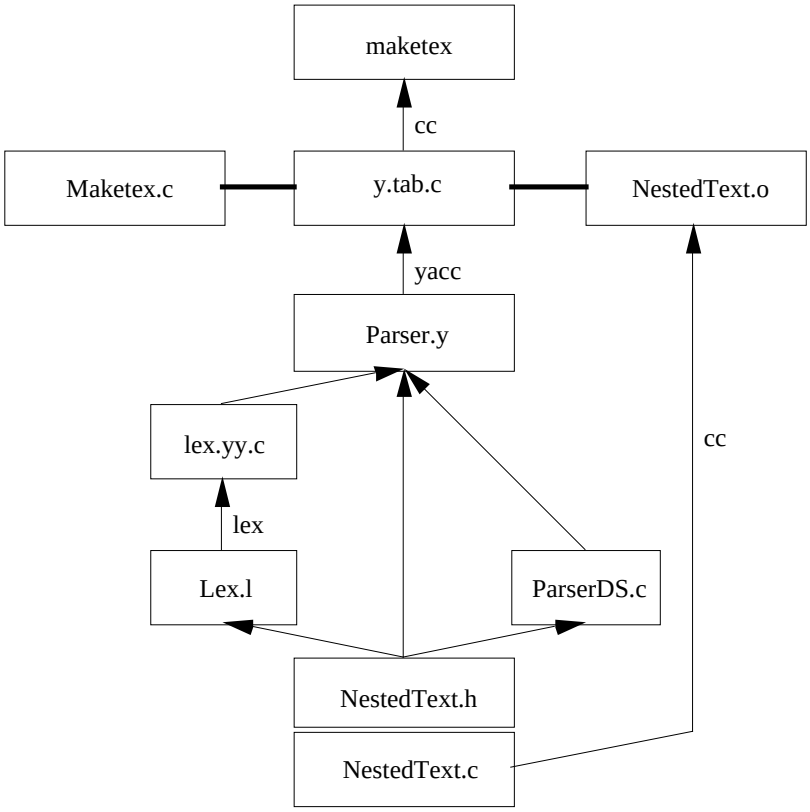

Figure 3: Technical dependencies in the construction of the executable *maketex*

Here each box corresponds to a file. An unlabeled arrow means the "include" relationship (for example, *NestedText.h* is included into *Lex.l*, *Parser.y*, and *ParserDS.c*. Edges labeled with *lex*, *yacc*, and *cc* mean that the tools *lex* and *yacc* or the C compiler, respectively, produce the result files. Fat edges connecting files indicate that these files are compiled together.

The rest of this document is structured as follows: Section 2 describes the *NestedText* module (files *NestedText.h* and *NestedText.c*), Section 3 the lexical analyser (*Lex.l*), Section 4 *ParserDS.c*, Section 5 the parser itself (*Parser.y*). Section 6 shows the header file for Latex and the rather trivial program *Maketex.c*. Section 7 contains the auxiliary functions *pdtabs.c* and *linebreaks.c*. Section 8 gives the command procedures *pdview* and *pdtex*. Finally, Section 9 contains the *makefile*.

2 The Module NestedText

2.1 Definition Part

(File *PDNestedText.h*)

This module allows one to create nested text structures and to write them to standard output. It provides in principle a data type *listexpr* (representing such structures) and operations:

atom: string × int → listexpr
atomc: string → listexpr
concat: listexpr × listexpr → listexpr
print: listexpr → e
copyout: listexpr → string × int
release-storage

However, for use by *lex* and *yacc* generated lexical analysers and parsers which only allow to associate integer values with grammar symbols, we represent a *listexpr* by an integer (which is, in fact, an index into an array for nodes). Hence we have a signature:

atom: string × int → int
atomc: string → int
concat: int × int → int
print: int → e
copyout: int → string × int
release-storage

The module uses two storage areas. The first is a buffer for text characters, it can take up to STRINGMAX characters, currently set to 30000. The second provides nodes for the nested list structure; currently up to NODESMAX = 30000 nodes can be created.

The operations are defined as follows:

```
int atom(char *string, int length)
```

List expressions, that is, values of type *listexpr* are either atoms or lists. The function *atom* creates from a character string *string* of length *length* a list expression which is an atom containing this string. Possible errors: The text buffer or storage space for nodes may overflow.

```
int atomc(char *string)
```

The function *atomc* works like *atom* except that the parameter should be a null-terminated string. It determines the length itself. To be used in particular for string constants written directly into the function call.

```
int concat(int list1, int list2)
```

Concats the two lists; returns a list expression representing the concatenation. Possible error: the storage space for nodes may be exceeded.

```
print(int list)
```

Writes the character strings from all atoms in *list* in the right order to standard output.

```
copyout(int list, char *target, int lengthlimit)
```

Copies the character strings from all atoms in *list* in the right order into a string variable *target*. Parameter *lengthlimit* ensures that the maximal available space in *target* is respected; an error occurs if the list expression *list* contains too many characters.

```
release_storage()
```

Destroys the contents of the text and node buffers. Should be used only when a complete piece of text has been recognized and written to the output. Warning: Must not be used after pieces of text have been recognized for which the parser depends on reading a look-ahead token! This token will be in the text and node buffers already and be lost. Currently this applies to lists.

The following is what is technically exported from this file:

```
int atom(), atomc(), concat();
int print(), copyout(), release_storage(),
show_storage();                 /* show_storage used only for testing */
```

2.2 Implementation Part

(File *PDNestedText.c*)

```
#include <string.h>
#include "PDNestedText.h"

#define AND &&
#define TRUE 1
#define FALSE 0
#undef NULL
#define NULL -1

#define STRINGMAX 30000
```

Maximal number of characters in buffer *text*.

```
#define NODESMAX 30000
```

Maximal number of nodes available from *nodespace*.

```
struct listexpr {
    int left;
    int right;
    int atomstring;             /* index into array text */
    int length;                 /* no of chars in atomstring*/
};
```

If *left* is NULL then this represents an atom, otherwise it is a list in which case *atomstring* must be NULL.

```
struct listexpr nodespace[NODESMAX];
int first_free_node = 0;

char text[STRINGMAX];
int first_free_char = 0;
```

The function *atom* creates from a character string *string* of length *length* a list expression which is an atom containing this string. Possible errors: The text buffer or storage space for nodes may overflow.

```
int atom(char *string, int length)
{
    int newnode;
    int i;

    /* put string into text buffer: */

        if (first_free_char + length > STRINGMAX)
            {printf("Error: too many characters.\n"); exit(1);}

        for (i = 0; i< length; i++)
            text[first_free_char + i] = string[i];

    /* create new node */

        newnode = first_free_node++;
        if (first_free_node > NODESMAX)
            {printf("Error: too many nodes.\n"); exit(1);}

        nodespace[newnode].left = NULL;
        nodespace[newnode].right = NULL;
        nodespace[newnode].atomstring = first_free_char;
            first_free_char = first_free_char + length;
        nodespace[newnode].length = length;
        return(newnode);
}
```

The function *atomc* works like *atom* except that the parameter should be a null-terminated string. It determines the length itself. To be used in particular for string constants written directly into the function call.

```
int atomc(char *string)
{   int length;

    length = strlen(string);
    return(atom(string, length));
}
```

The function *concat* concats two lists; it returns a list expression representing the concatenation. Possible error: the storage space for nodes may be exceeded.

```
int concat(int list1, int list2)
{
    int newnode;

    newnode = first_free_node++;

    if (first_free_node > NODESMAX)
        {printf("Error: too many nodes.\n"); exit(1);}

    nodespace[newnode].left = list1;
    nodespace[newnode].right = list2;
    nodespace[newnode].atomstring = NULL;
    nodespace[newnode].length = 0;
    return(newnode);
}
```

Function *print* writes the character strings from all atoms in *list* in the right order to standard output.

```
print(int list)
{
    int i;

    if (isatom(list))
        for (i = 0; i < nodespace[list].length; i++)
            putchar(text[nodespace[list].atomstring + i]);
    else
        {print(nodespace[list].left); print(nodespace[list].right);};
}
```

Function *isatom* obviously checks whether a list expression *list* is an atom.

```
int isatom(int list)
{
    if (nodespace[list].left == NULL) return(TRUE);
    else return(FALSE);
}
```

The function *copyout* copies the character strings from all atoms in *list* in the right order into a string variable *target*. Parameter *lengthlimit* ensures that the maximal available space in *target* is respected; an error occurs if the list expression *list* contains too many characters. *Copyout* just calls an auxiliary recursive procedure *copylist* which does the job.

```
copyout(int list, char *target, int lengthlimit)
{   int i;

    i = copylist(list, target, lengthlimit);
    target[i] = '\0';
}

int copylist(int list, char *target, int lengthlimit)
{   int i, j;

    if (isatom(list))
        if (nodespace[list].length <= lengthlimit - 1)
            {for (i = 0; i < nodespace[list].length; i++)
                target[i] = text[nodespace[list].atomstring + i];
            return nodespace[list].length;
            }
        else
            {printf("Error in copylist: too long text.\n"); print(list);
            exit(1);
            }
    else
        {i = copylist(nodespace[list].left, target, lengthlimit);
        j = copylist(nodespace[list].right, &target[i], lengthlimit - i);
        return (i+j);
        }
}
```

Function *release-storage* destroys the contents of the text and node buffers. Should be used only when a complete piece of text has been recognized and written to the output. Do not use it for text pieces whose recognition needs look-ahead!

```
release_storage()
{   first_free_char = 0;
    first_free_node = 0;
}
```

Function *show-storage* writes the contents of the text and node buffers to standard output; only used for testing.

```
show_storage()
{   int i;
    for (i = 0; i < first_free_char; i++) putchar(text[i]);

    for (i = 0; i < first_free_node; i++)
         printf("node: %d, left: %d, right: %d, atomstring: %d, length: %d\n",
                i, nodespace[i].left, nodespace[i].right,
                nodespace[i].atomstring, nodespace[i].length);
}
```

3 Lexical Analysis

3.1 Introduction

The file *PDLex.l* contains a specification of a lexical analyser. A description of the structure of *lex* specifications can be found in [ASU86, Section 3.5]. More detailed information is given in [SUN88, Section 10]. From such a specification, the UNIX tool *lex* creates a file *lex.yy.c* which can then be compiled to obtain a lexical analyser. This analyser is given as a function

```
int yylex()
```

On each call, this function matches a piece of the input string and returns a *token*, which is an integer constant.

A lex specification has the following structure:

```
declarations
%%
translation rules
%%
auxiliary procedures
```

The *declarations* section provides definitions of tokens to be returned (within brackets of the form %{ ... }%. The remainder of this section contains definitions of regular symbols (nonterminals of a regular grammar). For example, the definitions

```
digit           [0-9]
num             ({digit}{digit}|{digit})
```

introduce two nonterminals called *digit* and *num* respectively, by giving regular expressions on the right hand side for them. Terminal symbols can be described by character classes such as [0-9] (for digits), [\n] (matches just the end of line symbol), [ab?!] (contains just these four characters), etc. Nonterminal symbols used in regular expressions have to be enclosed by braces. Hence here a number is defined to consist of either one or two digits. The characters "|", "+", and "*" have the usual meaning for the composition of regular expressions.

The *translation rules* section contains a list of rules. Each rule describes some action to be taken whenever a piece of the input string has been matched. For example, the rule

```
^{head1}        {yylval = atom(yytext, yyleng); return(HEAD1);}
```

states the action to be taken when a *head1* regular symbol has been recognized which is defined in the *declarations* section by:

```
head1           {num}" "
```

So the input string matching *head1* consists of one or two digits followed by a blank. A translation rule consists of a regular symbol or regular expression on the left hand side, and some C code in braces on the right hand side. The C code says which token is to be returned (if any, one can also just skip a part of the input string and not return a token). The rule above says that a *HEAD1* token is to be returned from lexical analysis on recognizing a *head1* regular symbol which must occur at the beginning of a line (this is specified by the leading caret character).

In addition to returning a token, one often wants to give further information. For the communication with a *yacc* generated parser, a global integer variable *yylval* is predefined. A value assigned to this variable can be accessed in parsing (see Section 5). The input string matched by a regular expression is given by *lex* variables

```
char *yytext;
int yyleng;
```

where *yytext* points to the first character and *yyleng* gives the number of characters matched. The lexical analyser specified here usually returns the text that has been matched by creating a corresponding atom in the *NestedText* data structure and returning its node index. This happens also in the rule shown above.

The last section *auxiliary procedures* contains C code that is copied directly into the program *lex.yy.c*. Here one can define support variables or functions for the action parts of translation rules.

Some specific comments about the definitions given below:

- The regular symbols *open*, *open2*, and *close* consume preceding and following empty lines (consisting of space and end-of-line characters). Similarly, *epar* (paragraph end symbol) consumes subsequent empty lines. This is to avoid superfluous space in "verbatim" sections of the target Latex document.
- A *close* followed by an *open* commentary bracket is omitted completely. This means that adjacent documentation sections are merged into one. This is needed in particular for documents obtained by concatenating several files. Be aware of this in testing lexical analysis! This is the only case when parts of the input string are "swallowed" without returning tokens.

Note that the characters

```
~, *, ", [, ], :
```

---

are returned directly to the parser (each is its own token) because they are used there directly in grammar rules. These characters are matched by the rule with a "." on the left hand side. The dot matches everything unmatched otherwise.

3.2 The Specification

(File *PDLex.l*)

```
%{
#include "PDNestedText.h"
%}


/* regular definitions */

lbracket        ("(*"|"/*")
rbracket        ("*)"|"*/")
star            [*]
other           [-;,?!\‘\’()/@#$%_\^{}+=|<>\n&]
open            {lbracket}{star}*(" "*[\n])+
open2           ([\n]" "*)*[\n]{lbracket}{star}*(" "*[\n])+
close           {star}*{rbracket}(" "*[\n])+
epar            [\n]" "*[\n](" "*[\n])*
defline1        [\n]" "*"//"
defline2        " "*"//"
digit           [0-9]
num             ({digit}{digit}|{digit})
ref             "["{num}"]"
verbatim        "----"

head1           {num}" "
head2           {num}"."{head1}
head3           {num}"."{head2}
head4           {num}"."{head3}
head5           {num}"."{head4}

enum1           "  "{digit}" "|" "{digit}{digit}" "
enum2           "  "{enum1}
bullet1         "  * "
bullet2         "  "{bullet1}
follow1         "    "
follow2         "  "{follow1}

display         "        "
figure          "                "

%%
```

```
^{open}                 {return(OPEN);}
{open2}                 {return(OPEN);}
^{close}                {return(CLOSE);}
^{close}{open}          { }
^{verbatim}             {return(VERBATIM);}
{epar}                  {yylval = atom(yytext, yyleng); return(EPAR);}
{defline1}              {yylval = atom(yytext, yyleng); return(DEFLINE);}
^{defline2}             {yylval = atom(yytext, yyleng); return(DEFLINE);}
[A-Za-z]                {yylval = atom(yytext, yyleng); return(LETTER);}
^{head1}                {yylval = atom(yytext, yyleng); return(HEAD1);}
^{head2}                {yylval = atom(yytext, yyleng); return(HEAD2);}
^{head3}                {yylval = atom(yytext, yyleng); return(HEAD3);}
^{head4}                {yylval = atom(yytext, yyleng); return(HEAD4);}
^{head5}                {yylval = atom(yytext, yyleng); return(HEAD5);}
^{enum1}                {yylval = atom(yytext, yyleng); return(ENUM1);}
^{enum2}                {yylval = atom(yytext, yyleng); return(ENUM2);}
^{bullet1}              {yylval = atom(yytext, yyleng); return(BULLET1);}
^{bullet2}              {yylval = atom(yytext, yyleng); return(BULLET2);}
^{follow1}              {yylval = atom(yytext, yyleng); return(FOLLOW1);}
^{follow2}              {yylval = atom(yytext, yyleng); return(FOLLOW2);}
^{display}              {yylval = atom(yytext, yyleng); return(DISPLAY);}
^{figure}               {yylval = atom(yytext, yyleng); return(FIGURE);}
^({ref}" "|"[] ")       {yylval = atom(yytext, yyleng); return(STARTREF);}
{ref}                   {yylval = atom(yytext, yyleng); return(REF);}
[0-9]                   {yylval = atom(yytext, yyleng); return(DIGIT);}
"[~]"                   {yylval = atom(yytext, yyleng); return(TILDE);}
"[*]"                   {yylval = atom(yytext, yyleng); return(STAR);}
"[\"]"                  {yylval = atom(yytext, yyleng); return(QUOTE);}
" ~ "                   {yylval = atom(yytext, yyleng); return(BLANKTILDE);}
" * "                   {yylval = atom(yytext, yyleng); return(BLANKSTAR);}
" \" "                  {yylval = atom(yytext, yyleng); return(BLANKQUOTE);}
{other}                 {yylval = atom(yytext, yyleng); return(OTHER);}
.                       {yylval = atom(yytext, yyleng); return(yytext[0]);}
"paragraph"             {yylval = atom(yytext, yyleng); return(PARFORMAT);}
"characters"            {yylval = atom(yytext, yyleng); return(CHARFORMAT);}

%%
```

3.3 Testing the Lexical Analyser

One can test lexical analysis separately from the rest of the system. The files *PDTokens.h* and *PDLexTest.c* are needed. The file *PDTokens.h* needs to be included in the *declarations* section of *Lex.l*:

```
%{
#include "PDTokens.h"
}%
```

This file just defines each token as an integer constant:

```
# define OPEN 257
# define CLOSE 258
# define EPAR 259
# define DEFLINE 260
# define LETTER 261
# define DIGIT 262
# define OTHER 263
# define TILDE 264
# define STAR 265
# define QUOTE 266
# define BLANKTILDE 267
# define BLANKSTAR 268
# define BLANKQUOTE 269
# define HEAD1 270
# define HEAD2 271
# define HEAD3 272
# define HEAD4 273
# define HEAD5 274
# define ENUM1 275
# define ENUM2 276
# define BULLET1 277
# define BULLET2 278
# define FOLLOW1 279
# define FOLLOW2 280
# define DISPLAY 281
# define FIGURE 282
# define STARTREF 283
# define REF 284
# define VERBATIM 285
# define PARFORMAT 286
# define CHARFORMAT 287
```

The file *PDLexTest.c* contains the main function for testing lexical analysis. It prints all letters directly and prints the other tokens as integers.

```
#include "PDTokens.h"
#include "PDNestedText.h"

main()
{

        int token;
        yylval = 0;

        token = yylex();
        while (token != 0) {

                if (token == LETTER) print(yylval);
                else printf("%d \n", token);

                token = yylex();
        }
}
```

To produce a lexical analyser for testing one can issue the following commands (after including *PDTokens.h* in *PDLex.l*):

```
lex PDLex.l
cc PDLexTest.c lex.yy.c PDNestedText.o -ll
```

The file *a.out* will then contain the analyser.

4 Data Structures for the Parser

(File *PDParserDS.c*)
This file contains data structures for special paragraph and character formats (Section 4.2) and special characters (Section 4.3), also some auxiliary functions for the parser (Section 4.4).

```
#include "PDNestedText.h"
#include <string.h>

#define AND &&
```

4.1 Global Constants and Variables

```
#define BRACKETLENGTH 500
```

Length of text that may occur in square brackets within normal text.

```
int pindex = -1;
```

Index (in array *definitions*, see below) of the most recently used special paragraph format.

```
int cindex = -1;
```

Index (in array *definitions*, see below) of the most recently used special character format.

4.2 Data Structure for Definitions of Special Paragraph or Character Formats

```
#define DEFMAX 100
#define NAMELENGTH 30
#define COMLENGTH 100

struct def {
        int index;
        char name[NAMELENGTH];
        char open[COMLENGTH];
        char close[COMLENGTH];
} definitions[DEFMAX];

int first_free_def = 0;
int last_global_def = -1;
```

Contains definitions of special paragraph or character formats, such as

```
paragraph [1] Title: [{\bf \Large \begin{center}] [\end{center} }]
```

Here 1 would become the *index*, *Title* would be the *name*, the material enclosed in the first pair of square brackets would be the *open*, and that in the second pair of brackets the *close* component.

Into this array are put first definitions from the header documentation section. When these are complete, the variable *last-global-def* is set to the array index of the last entry. Then, for each paragraph which has annotation lines further definitions may be appended. The lookup procedure *lookup-def* (see below) searches from the end so that if finds first paragraph annotations. (However, paragraph annotations are not yet implemented.)

The following function *enter-def* puts a quadruple into this array. The last three parameters are (indices of) list expressions.

```
enter_def(int index, int name, int open, int close)
{   definitions[first_free_def].index = index;
    copyout(name, definitions[first_free_def].name, NAMELENGTH);

        /* This function copies the first parameter list expression into a
        string given as a second parameter. Part of NestedText.*/

    copyout(open, definitions[first_free_def].open, COMLENGTH);
    copyout(close, definitions[first_free_def].close, COMLENGTH);

    first_free_def++;

    if (first_free_def >= DEFMAX)
        {printf("Error in enter_def: table full.\n");
        exit(1);
        }
}
```

Function *lookup-def* finds an array index in array *definitions* such that its *index* component has value i. Starts at the end and searches backwards in order to find paragraph annotations first. Returns either the array index, or -1 if entry was not found.

```
int lookup_def(int i)
{   int j;
    j = first_free_def;
    do j--;
    while ((j>=0) AND (definitions[j].index != i));
    return j;
}
```

4.3 Data Structure for Definitions of Special Characters

```
#define CODELENGTH 31           /* 30 usable characters + 0C */
#define SCMAX 100

struct schar {
        char code[CODELENGTH];
        char command[COMLENGTH];
} schars[SCMAX];

int first_free_schar = 0;
```

Contains definitions of special characters such as:

```
   [ue]    [\"{u}]
```

The function *enter-schar* puts such a pair into the data structure. Parameters are again indices of list expressions in array *nodespace*.

```
enter_schar(int code, int command)
{   copyout(code, schars[first_free_schar].code, CODELENGTH);
    copyout(command, schars[first_free_schar].command, COMLENGTH);

    first_free_schar++;

    if (first_free_schar >= SCMAX)
        {printf("Error in enter_schar: table full.\n");
        exit(1);
        }
}
```

The function *lookup-schar* tries to find a parameter *string* as a *code* component under some index j in the array *schars* with special character definitions. It returns this index. If such an entry was not found, it returns a negative index value.

```
int lookup_schar(char *string)
{   int j;
    j = first_free_schar;
    do j--;
    while ((j>=0) AND (strcmp(string, schars[j].code) != 0));
    return j;
}
```

4.4 Auxiliary Functions

The function *get-startref-index* gets as a parameter a list expression *listexpr* containing a special paragraph format number in square brackets followed by a blank (the number can have one or two digits), or an empty pair of square brackets (as a reference to a special format used previously). Hence examples are:

```
[15] [7] or []
```

The function returns either the numeric value (the format number) or 0 for an empty pair of brackets.

```
int get_startref_index(int listexpr)
{   char ref[6];
    int length;

    copyout(listexpr, ref, 6);
    length = strlen(ref);

    switch(length) {
        case 3:                                 /* empty brackets */
                return 0;
        case 4:                                 /* one digit */
                return (ref[1] - '0');
        case 5:                                 /* two digits */
                return (10 * (ref[1] - '0') + (ref[2] - '0'));
        default:
                {printf ("Error in get_startref_index: length is %d.\n",
                        length);
                exit(1);
                }
    }
}
```

Function *get-ref-index* does the same for a reference without the trailing blank.

```
int get_ref_index(int listexpr)
{   return(get_startref_index( concat(listexpr, atom(" ", 1)) ));
}
```

5 The Parser

(File *PDParser.y*)

5.1 Introduction

This file contains a *yacc* specification of a parser which is transformed by the UNIX tool *yacc* into a program file *y.tab.c* which in turn is compiled to produce a parser. For an introduction to *yacc* specifications see [ASU86, Section 4.9]. Detailed information is given in [SUN88, Section 11].

In fact, the specification contains "semantic rules" (or *actions*); hence, the generated program is a compiler (from PD files into Latex). The parsing technique used is bottom-up LR parsing. Let us briefly consider two example rules:

```
heading          : heading1
                 | heading2
                 | heading3
                 | heading4
                 | heading5
                 ;

heading1         :  HEAD1 paragraph_rest {printf("\\section {");
                                          print($2);
                                          printf("}\n\n");}
                 ;
```

The first is a grammar rule stating that a *heading* (nonterminal) can be derived into either a *heading1* or a *heading2* or a *heading3* (nonterminal). The second rule says that a *heading1* consists of a *HEAD1* token followed by a *paragraph-rest*. This rule has an associated action which is C code enclosed by braces.

Attached to each grammar symbol (nonterminal, or terminal token) is an integer-valued *attribute*. One can refer to these attributes in a grammar rule by the names *$$*, *$1*, *$2*, etc. where *$$* refers to the attribute of the nonterminal on the left hand side and *$1*, *$2*, etc. to the grammar symbols on the right hand side. Hence, in the second rule, *$$* is the attribute attached to *heading1*, *$1* belongs to *HEAD1*, and *$2* to *paragraph-rest*.

For the terminal tokens which are generated in lexical analysis the attribute value is set by an assignment to the variable *yylval*. The lexical analyser used here assigns always the index of a node of the *NestedText* data structure containing the character string matching the token.

For the nonterminals, the attribute value is set by an assignment to *$$* in the action part of a grammar rule. The bottom-up parser basically works as follows. Terminal tokens and nonterminals recognized earlier are kept on a stack. Whenever the top symbols on the stack correspond to the right hand side of a grammar rule which is applicable with the current derivation, a *reduction* is made: The right hand side symbols are removed from the stack and the left hand side nonterminal is put on the stack. In addition, the action associated with the rule (the C code in braces) is executed.

Therefore, in our example rule the *HEAD1* and *paragraph-rest* symbols are removed from the stack and a *heading1* symbol is put on top. The action is executed, namely:

1. The text "\section {" is written to the output file.
2. The text associated with *paragraph-rest* (a *NestedText* node whose index is given in *$2*) is written (by the *print* function from *NestedText*).
3. The text "}\n\n" (a closing brace followed by two end-of-line characters) is written to the output.

Hence for a heading described in the PD file by

```
5 The Parser
```

---

a piece of Latex code

```
\section {The Parser}
```

---

followed by an empty line is written to the output file. To understand the output created in the grammar rules you need to know (some) LaTeX, see [La86].

5.2 Declaration Section: Definition of Tokens

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "PDNestedText.h"
#include "PDParserDS.c"
%}

%token OPEN, CLOSE, EPAR, DEFLINE, LETTER, DIGIT, OTHER, TILDE, STAR,
        QUOTE, BLANKTILDE, BLANKSTAR, BLANKQUOTE,
        HEAD1, HEAD2, HEAD3, HEAD4, HEAD5, ENUM1, ENUM2, BULLET1, BULLET2,
        FOLLOW1, FOLLOW2, DISPLAY, FIGURE, STARTREF, REF, VERBATIM,
        PARFORMAT, CHARFORMAT
%%
```

5.3 Document Structure and Program Sections

```
document        : doc
                | doc program_section
                ;

doc             : space doc_section
                | doc program_section doc_section
                ;

doc_section     : OPEN elements CLOSE
                ;

program_section :       {printf("{\\small \\begin{quote}
\\begin{verbatim}\n");}
                  chars {printf("\n\\end{%s} \\end{quote}}\n\n", "verbatim");}
                ;

chars           :
                | chars text_char       {print($2);}
                | chars TILDE           {print($2);}
                | chars STAR            {print($2);}
                | chars QUOTE           {print($2);}
                | chars '\"'            {print($2);}
```

```
                | chars '*'             {print($2);}
                | chars '~'             {print($2);}
                | chars '['             {print($2);}
                | chars ']'             {print($2);}
                | chars follow_elem     {print($2);}
                | chars EPAR            {print($2);}
                | chars DEFLINE         {print($2);}
                ;

elements        :
                | elements definitions
                | elements element      {release_storage();}
                | elements list         /* storage cannot be released after
                                        lists because of look-ahead */
                ;
```

5.4 Definitions of Special Formats and Characters

```
definitions     : defs EPAR
                ;

defs            : defline
                | defs defline
                ;

defline         : DEFLINE par_format
                | DEFLINE char_format
                | DEFLINE special_char_def
                ;

par_format      : PARFORMAT space REF space ident space ':'
                        space bracketed2 space bracketed2
                                {
                                /* test only:
                                print($3);
                                printf("paragraph definition: %d ",
                                        get_ref_index($3));
                                print($5);
                                print($9);
                                print($11);
                                */
                                enter_def(get_ref_index($3), $5, $9, $11);}
                ;

char_format     : CHARFORMAT space REF space ident space ':'
                        space bracketed2 space bracketed2
                                {
                                /*
                                printf("characters definition: %d ",
                                        get_ref_index($3));
                                print($5);
                                print($9);
```

```
                                print($11);
                                */

                                enter_def(get_ref_index($3), $5, $9, $11);}
                ;

special_char_def: bracketed2 space bracketed2
                                        {enter_schar($1, $3);}
                ;

space           :
                | space ' '
                | space FOLLOW1
                | space FOLLOW2
                | space DISPLAY
                | space FIGURE
                ;

ident           : LETTER                {$$ = $1;}
                | ident LETTER          {$$ = concat($1, $2);}
                | ident DIGIT           {$$ = concat($1, $2);}
                ;
```

5.5 Text Elements

5.5.1 Predefined Paragraph Types

```
element         : standard_paragraph
                | heading
                | verb
                | display
                | figure
                | special_paragraph
                ;

standard_paragraph : paragraph_rest     {print($1); printf("\n\n");}
                ;

heading         : heading1
                | heading2
                | heading3
                | heading4
                | heading5
                ;

heading1        : HEAD1 paragraph_rest  {printf("\\section {");
                                        print($2);
                                        printf("}\n\n");}
                ;

heading2        : HEAD2 paragraph_rest  {printf("\\subsection {");
                                        print($2);
```

```
                                        printf("}\n\n");}
                ;

heading3        : HEAD3 paragraph_rest  {printf("\\subsubsection {");
                                        print($2);
                                        printf("}\n\n");}
                ;

heading4        : HEAD4 paragraph_rest  {printf("\\paragraph {");
                                        print($2);
                                        printf("}\n\n");}
                ;

heading5        : HEAD5 paragraph_rest  {printf("\\subparagraph {");
                                        print($2);
                                        printf("}\n\n");}
                ;

verb            : verb_start verb_end
                ;

verb_start      : VERBATIM      {printf("\\hspace{0.9cm}
\\rule{2in}{0.1pt}\n{\\small \\begin{verbatim}\n    ");}
                ;

verb_end        : chars VERBATIM        {printf("\\end{%s}\n\\hspace{0.9cm}
\\rule{2in}{0.1pt}\n", "verbatim");}
                ;


display         : DISPLAY paragraph_rest {printf("\\begin{quote}\n");
                                        printf("        ");
                                        print($2);
                                        printf("\n\\end{quote}\n\n");}
                ;
```

5.5.2 Figures

```
figure          : FIGURE figure_text optional_caption
                  bracketed2 annotations
                                        {printf("\\begin{figure}[htb]\n");
                                        printf("\\begin{center}\n");
                                        printf("\\leavevmode\n");
                                        printf("        \\epsfbox{Figures/");
                                                print($4);
                                        printf("}\n");
                                        printf("\\end{center}\n");
                                        printf("        \\caption{");
                                                print($3);
                                        printf("}\n");
                                        printf("\\end{figure}\n");}
                ;
```

```
optional_caption:                       {$$ = atomc("");}
                | ':' figure_text       {$$ = $2;}
                ;

figure_text     :                       {$$ = atomc("");}
                | figure_text ftext_char {$$ = concat($1, $2);}
                | figure_text TILDE     {$$ = concat($1, atomc("~"));}
                | figure_text STAR      {$$ = concat($1, atomc("*"));}
                | figure_text QUOTE     {$$ = concat($1, atomc("\""));}
                | figure_text emphasized {$$ = concat($1, $2);}
                | figure_text bold_face {$$ = concat($1, $2);}
                | figure_text special_char_format {$$ = concat($1, $2);}
                | figure_text follow_elem {$$ = concat($1, $2);}
                ;
```

5.5.3 Special Paragraph Formats

```
special_paragraph: STARTREF paragraph_rest

                {int i;

                i = get_startref_index($1);
                if (i > 0)                      /* not an empty start ref */
                    pindex = lookup_def(i);
                                                /* otherwise use previous
                                                pindex value */
                if (pindex >= 0)                /* def was found */
                    {printf("%s ", definitions[pindex].open);
                    print($2);
                    printf("%s \n\n", definitions[pindex].close);
                    }
                else print($2);         /* make it a standard paragraph */
                }
                ;
```

5.6 Lists

```
list            : itemized1     {printf("\\begin{itemize}\n");
                                 print($1);
                                 printf("\n\n\\end{itemize}\n\n");}
                | enum1         {printf("\\begin{enumerate}\n");
                                 print($1);
                                 printf("\n\n\\end{enumerate}\n\n");}
                ;

itemized1       : bulletitem1           {$$ = $1;}
                | itemized1 bulletitem1 {$$ = concat($1, $2);}
                ;

bulletitem1     : bulletpar1            {$$ = $1;}
                | bulletitem1 followup1 {$$ = concat($1, $2);}
```

```
                | bulletitem1 list2     {$$ = concat($1, $2);}
                ;

bulletpar1      : BULLET1 paragraph_rest
                                {$$ = concat(atomc("\n  \\item "),
                                        concat($2, atomc("\n\n")));}
                ;

followup1       : FOLLOW1 paragraph_rest
                                {$$ = concat($1,
                                          concat($2, atomc("\n\n")));}
                ;

enum1           : enumitem1             {$$ = $1;}
                | enum1 enumitem1       {$$ = concat($1, $2);}
                ;

enumitem1       : enumpar1              {$$ = $1;}
                | enumitem1 followup1   {$$ = concat($1, $2);}
                | enumitem1 list2       {$$ = concat($1, $2);}
                ;

enumpar1        : ENUM1 paragraph_rest
                                {$$ = concat(atomc("\n  \\item "),
                                        concat($2, atomc("\n\n")));}
                ;

list2           : itemized2 {$$ = concat(atomc("\n    \\begin{itemize}\n"),
                                concat($1,
                                atomc("\n    \\end{itemize}\n\n")));}
                | enum2 {$$ = concat(atomc("\n    \\begin{enumerate}\n"),
                                concat($1,
                                atomc("\n    \\end{enumerate}\n\n")));}
                ;

itemized2       : bulletitem2           {$$ = $1;}
                | itemized2 bulletitem2 {$$ = concat($1, $2);}
                ;

bulletitem2     : bulletpar2            {$$ = $1;}
                | bulletitem2 followup2 {$$ = concat($1, $2);}
                ;

bulletpar2      : BULLET2 paragraph_rest
                                {$$ = concat(atomc("\n      \\item "),
                                        concat($2, atomc("\n\n")));}
                ;

followup2       : FOLLOW2 paragraph_rest
                                {$$ = concat($1, concat($2, atomc("\n\n")));}
                ;

enum2           : enumitem2             {$$ = $1;}
```

```
                | enum2 enumitem2       {$$ = concat($1, $2);}
                ;

enumitem2       : enumpar2              {$$ = $1;}
                | enumitem2 followup2   {$$ = concat($1, $2);}
                ;

enumpar2        : ENUM2 paragraph_rest
                                {$$ = concat(atomc("\n      \\item "),
                                        concat($2, atomc("\n\n")));}

                ;
```

5.7 Text Structure

Unfortunately, this gets a bit complex because we must take care of the following:

- Most tokens that are recognized in lexical analysis may occur in normal text; we must make sure that they can be reduced there. In particular, the tokens defining paragraph formats must be allowed to occur in the middle of a paragraph and not be interpreted there.
- We must take care of escaping characters with a special meaning. This concerns the *TILDE*, *STAR*, and *QUOTE* tokens which are formed in LA from the corresponding characters in square brackets. In normal text, the square brackets must be stripped off. On the other hand, in program text or in definitions (given in square brackets themselves) the character strings should be left untouched.
- Emphasized text (enclosed by tilde characters) and bold face text (enclosed by stars) may be nested, but we must make sure through the grammar rules that emphasized cannot occur within emphasized etc. Otherwise a second tilde meaning a closing bracket of the emphasized text would be shifted on the stack by the parser rather than reduced as we want.

```
paragraph_rest  : text annotations EPAR {$$ = $1;}
                ;

text            :                       {$$ = atomc("");}
                | netext                {$$ = $1;}
                ;

netext          : start_elem            {$$ = $1;}
                | netext start_elem     {$$ = concat($1, $2);}
                | netext follow_elem    {$$ = concat($1, $2);}
                ;

start_elem      : text_char             {$$ = $1;}
                | TILDE                 {$$ = atomc("~");}
                | STAR                  {$$ = atomc("*");}
                | QUOTE                 {$$ = atomc("\"");}
                | emphasized            {$$ = $1;}
```

```
                | bold_face             {$$ = $1;}
                | special_char_format   {$$ = $1;}
                | bracketed             {$$ = $1;}
                ;

follow_elem     : HEAD1                 {$$ = $1;}
                | HEAD2                 {$$ = $1;}
                | HEAD3                 {$$ = $1;}
                | HEAD4                 {$$ = $1;}
                | HEAD5                 {$$ = $1;}
                | ENUM1                 {$$ = $1;}
                | ENUM2                 {$$ = $1;}
                | FOLLOW1               {$$ = $1;}
                | FOLLOW2               {$$ = $1;}
                | BULLET1               {$$ = $1;}
                | BULLET2               {$$ = $1;}
                | DISPLAY               {$$ = $1;}
                | FIGURE                {$$ = $1;}
                | STARTREF              {$$ = $1;}
                | REF                   {$$ = $1;}
                ;

text_char       : ftext_char            {$$ = $1;}
                | ':'                   {$$ = $1;}
                ;

ftext_char      : LETTER                {$$ = $1;}
                | DIGIT                 {$$ = $1;}
                | OTHER                 {$$ = $1;}
                | BLANKTILDE            {$$ = $1;}
                | BLANKSTAR             {$$ = $1;}
                | BLANKQUOTE            {$$ = $1;}
                | '\\'                  {$$ = $1;}
                | ' '                   {$$ = $1;}
                | '.'                   {$$ = $1;}
                | PARFORMAT             {$$ = $1;}
                | CHARFORMAT            {$$ = $1;}
                ;

emphasized      : '~' unemph_list '~'   {$$ = concat(atomc("{\\em "),
                                                concat($2,
                                                atomc("\\/}")));}
                ;

bold_face       : '*' unbold_list '*'   {$$ = concat(atomc("{\\bf "),
                                                concat($2,
                                                atomc("}")));}
                ;

unemph_list     :                       {$$ = atomc("");}
                | unemph_list unemph    {$$ = concat($1, $2);}
                ;
```

```
unemph          : text_char             {$$ = $1;}
                | TILDE                 {$$ = atomc("~");}
                | STAR                  {$$ = atomc("*");}
                | QUOTE                 {$$ = atomc("\"");}
                | follow_elem           {$$ = $1;}
                | '*' unboldemph_list '*'       {$$ = concat(atomc("{\\bf "),
                                                  concat($2,
                                                  atomc("}")));}
                | special_char_format   {$$ = $1;}
                | bracketed             {$$ = $1;}
                ;

unbold_list     :                       {$$ = atomc("");}
                | unbold_list unbold    {$$ = concat($1, $2);}
                ;

unbold          : text_char             {$$ = $1;}
                | TILDE                 {$$ = atomc("~");}
                | STAR                  {$$ = atomc("*");}
                | QUOTE                 {$$ = atomc("\"");}
                | follow_elem           {$$ = $1;}
                | '~' unboldemph_list '~'       {$$ = concat(atomc("{\\em "),
                                                  concat($2,
                                                  atomc("\\/}")));}
                | special_char_format   {$$ = $1;}
                | bracketed             {$$ = $1;}
                ;

unboldemph_list :                       {$$ = atomc("");}
                | unboldemph_list unboldemph    {$$ = concat($1, $2);}
                ;

unboldemph      : text_char             {$$ = $1;}
                | TILDE                 {$$ = atomc("~");}
                | STAR                  {$$ = atomc("*");}
                | QUOTE                 {$$ = atomc("\"");}
                | follow_elem           {$$ = $1;}
                | special_char_format   {$$ = $1;}
                | bracketed             {$$ = $1;}
                ;

plain_list      :                       {$$ = atomc("");}
                | plain_list plain      {$$ = concat($1, $2);}
                ;

plain           : text_char             {$$ = $1;}
                | TILDE                 {$$ = atomc("~");}
                | STAR                  {$$ = atomc("*");}
                | QUOTE                 {$$ = atomc("\"");}
                | bracketed             {$$ = $1;}
                ;
```

5.8 Special Character Formats

```
special_char_format :   '\"' plain_list '\"' REF

                        {int i;

                        i = get_ref_index($4);
                        cindex = lookup_def(i);
                        if (cindex >= 0)        /* def was found */
                            {$$ = concat(
                                   atomc(definitions[cindex].open),
                                   concat($2,
                                     atomc(definitions[cindex].close)  ));
                            }
                        else                    /* ignore special format */
                            $$ = $2;
                        }

                    | '\"' plain_list '\"'

                        {if (cindex >= 0)       /* def exists */
                            {$$ = concat(
                                   atomc(definitions[cindex].open),
                                   concat($2,
                                     atomc(definitions[cindex].close)  ));
                            }
                        else                    /* ignore special format */
                            $$ = $2;
                        }

                    ;
```

5.9 Text in Square Brackets: Checking for Special Characters

```
bracketed       : '[' btext ']'

                {char bracketstring[BRACKETLENGTH];
                int i;
                int length;

                copyout($2, bracketstring, BRACKETLENGTH);
                length = strlen(bracketstring);

                if (length <= CODELENGTH - 1)
                    {i = lookup_schar(bracketstring);
                    if (i >= 0)                 /* found */
                        $$ = atomc(schars[i].command);
                    else
                        $$ = concat($1,
                                concat($2, $3));
                    }
                else
```

```
                    $$ = concat($1, concat($2, $3));
                }
                ;

btext           :                       {$$ = atomc("");}
                | btext text_char       {$$ = concat($1, $2);}
                | btext TILDE           {$$ = concat($1, atomc("~"));}
                | btext STAR            {$$ = concat($1, atomc("*"));}
                | btext QUOTE           {$$ = concat($1, atomc("\""));}
                | btext follow_elem     {$$ = concat($1, $2);}
                | btext '\"'            {$$ = concat($1, $2);}
                | btext '*'             {$$ = concat($1, $2);}
                | btext '~'             {$$ = concat($1, $2);}
                | btext '[' btext ']'

                {char bracketstring[BRACKETLENGTH];
                int i;
                int length;

                copyout($3, bracketstring, BRACKETLENGTH);
                length = strlen(bracketstring);

                if (length <= CODELENGTH - 1)
                    {i = lookup_schar(bracketstring);
                    if (i >= 0)                 /* found */
                        $$ = concat($1, atomc(schars[i].command));
                    else
                        {$$ = concat($1,
                                concat($2,
                                        concat($3, $4)));}
                    }
                else
                    {$$ = concat($1,
                                concat($2,
                                        concat($3, $4)));}
                }
                ;
```

5.10 Uninterpreted Square Brackets (Used in Definitions)

```
bracketed2      : '[' btext2 ']'        {$$ = $2;}
                ;

btext2          :                       {$$ = atomc("");}
                | btext2 text_char      {$$ = concat($1, $2);}
                | btext2 TILDE          {$$ = concat($1, $2);}
                | btext2 STAR           {$$ = concat($1, $2);}
                | btext2 QUOTE          {$$ = concat($1, $2);}
                | btext2 follow_elem    {$$ = concat($1, $2);}
                | btext2 '\"'           {$$ = concat($1, $2);}
                | btext2 '*'            {$$ = concat($1, $2);}
                | btext2 '~'            {$$ = concat($1, $2);}
```

```
                | btext2 '[' btext2 ']' {$$ = concat($1,
                                              concat($2,
                                              concat($3, $4)));}
                ;

annotations     :
                ;


%%

#include "lex.yy.c"

int yyerror(const char *msg)
{
  if (yytext[0] == '\n' && yytext[1] == '\n')
    fprintf(stderr, "%s in paragraph before line %d.\n", msg, yylineno);
  else
    fprintf(stderr,
            "%s at line %d reading symbol '%s'.\n", msg, yylineno, yytext);

}
```

6 LaTeX Header and Main Program

6.1 The LaTeX Header File: pd.header

```
\documentclass[11pt]{article}
\usepackage{epsfig, a4wide, xspace}
\parindent0em
\parskip0.8ex plus0.4ex minus 0.4ex
\setcounter{secnumdepth}{4}
\setcounter{tocdepth}{3}
\fussy
\begin{document}
```

6.2 Main Program: Maketex.c

Use the parser to transform from implicitly formatted text to TeX.

```
main()
{       int error;

        error = yyparse();
        print_tail();
}

print_tail()
{
        printf("\\end{document}\n");
}
```

7 Two Auxiliary Programs

7.1 Program pdtabs.c

This program converts tab symbols into corresponding sequences of blanks. With the standard text editor, each tab corresponds to 8 blanks.

```
#define TABLENGTH 8
#define EOF -1

main()
{
        int c, position, nblanks, i;

        position = 0;
        while ((c = getchar()) != EOF)
                if (c == '\n')
                        {position = 0; putchar(c);}
                else if (c == '\t') {
                        nblanks = TABLENGTH - (position % TABLENGTH);
                        for (i = 0; i < nblanks; i++)
                                {position++; putchar(' ');}
                        }
                else {position++; putchar(c);}
}
```

7.2 Program *linebreaks.c*

This program reads a file from standard input and writes it to standard output. Whenever lines longer than LINELENGTH (which is 80 characters) occur, it puts a line break to the position of the last blank read before character 80 and continues in a new line. If there was no blank in such a line, it introduces a line break anyway (possibly in the middle of a word).

```
#define LINELENGTH 80
#define EOF -1
#define OR ||

main()
{
        int position, c, lastblank, i;
        int line[LINELENGTH];

        position = 0;
        lastblank = -1;

        while ((c = getchar()) != EOF) {
                line[position] = c;
                if (c == ' ') lastblank = position;
                position++;

                if ((c == '\n') OR (position == LINELENGTH))
```

```
                        if (c == '\n') {

                                /* output a complete line */

                                for (i = 0; i < position; i++)
                                        putchar(line[i]);
                                position = 0;
                                lastblank = -1;}

                        else if (lastblank > 0) {       /* a blank exists */

                                /* output line up to blank */

                                for (i = 0; i < lastblank; i++)
                                        putchar(line[i]);
                                putchar('\n');

                                /* move rest of line to the front */

                                for (i = lastblank + 1; i < position; i++)
                                        line[i - (lastblank + 1)] = line[i];
                                position = position - (lastblank + 1);
                                lastblank = -1;
                                }

                             else {                     /* no blank exists */

                                /* output line anyway */

                                for (i = 0; i < position; i++)
                                        putchar(line[i]);
                                putchar('\n');
                                position = 0;
                                lastblank = -1;
                                }
        }

}
```

8 Command Procedures

8.1 Procedure *pdview*

```
cat $HOME/pd.header > $1.tex
pdtabs < $1 | maketex | linebreaks >> $1.tex
latex $1.tex
yap $1.dvi &
rm $1.log
rm $1.tex
```

8.2 Procedure *pd2tex*

```
cat $HOME/pd.header > $1.tex
pdtabs < $1 | maketex | linebreaks >> $1.tex
```

9 The Makefile

```
OPTIONS = -g

# The first line is used when compiling with FLEX and BISON
# The second line when compiling with LEX and YACC
LINKLIBS = -lfl
#LINKLIBS = -ll -ly

# The first line is used when compiling with FLEX and BISON
# The second line when compiling with LEX and YACC
LEX = flex -l
#LEX = lex

# The first line is used when compiling with FLEX and BISON
# The second line when compiling with LEX and YACC
YACC = bison --yacc
#YACC = yacc

all:    maketex pdtabs linebreaks docu makehtml docuhtml makeascii

maketex: PDMaketex.c PDParser.y lex.yy.c PDNestedText.h PDParserDS.c
PDNestedText.o
        $(YACC) PDParser.y
        gcc $(OPTIONS) -o maketex PDMaketex.c y.tab.c PDNestedText.o
$(LINKLIBS)

makehtml: PDMakeHTML.c PDNestedText.o PDParserHTML.y lex.yy.c \
                PDNestedText.h PDParserDS.c
        $(YACC) PDParserHTML.y
        gcc $(OPTIONS) -o makehtml PDMakeHTML.c y.tab.c PDNestedText.o
$(LINKLIBS)

makeascii: PDMakeASCII.c PDNestedText.o PDParserASCII.y lex.yy.c \
                PDNestedText.h PDParserDS.c
        $(YACC) PDParserASCII.y
        gcc $(OPTIONS) -o makeascii PDMakeASCII.c y.tab.c PDNestedText.o
$(LINKLIBS)


#
lex.yy.c: PDLex.l PDNestedText.h
        $(LEX) PDLex.l

PDNestedText.o: PDNestedText.c PDNestedText.h
        gcc -c -g PDNestedText.c
```

```
pdtabs: pdtabs.c
        gcc -o pdtabs pdtabs.c

linebreaks: linebreaks.c
        gcc -o linebreaks linebreaks.c

docu:   PD1 PDNestedText.h PDNestedText.c PD3 PDLex.l PDTokens.h PDLexTest.c \
                PDParserDS.c PDParser.y PD6 pd.header \
                PDMaketex.c PD7 pdtabs.c linebreaks.c PD8 pdview \
                PD8.2 pd2tex  PD9 makefile \
                PDRefs
        cat PD1 PDNestedText.h PDNestedText.c PD3 PDLex.l PDTokens.h \
                PDLexTest.c PDParserDS.c \
                PDParser.y PD6 pd.header \
                PDMaketex.c PD7 pdtabs.c linebreaks.c PD8 pdview \
                PD8.2 pd2tex PD9 makefile \
                PDRefs > docu

docuhtml: HTML1 PDParserHTML.y PDMakeHTML.c HTML4 pd2html HTML5 makefile \
                PDRefsHTML
        cat HTML1 PDParserHTML.y PDMakeHTML.c HTML4 pd2html HTML5 \
                makefile PDRefsHTML > docuhtml

save:
        tar -cvf pd.tar HTML1 HTML4 HTML5 PD1 PD3 PD6 PD7 PD8 PD8.2 PD8.3 \
        PD8.4 PD8.5 PD9 PDLex.l PDLexTest.c PDMakeHTML.c PDMaketex.c \
        PDNestedText.c PDNestedText.h PDParser.y PDParserDS.c PDParserHTML.y \
        PDRefs PDRefsHTML PDTokens.h \
        linebreaks.c makefile makehtml maketex pd.header pd2html pd2tex \
        pdtabs.c pdview \
        PDMakeASCII.c PDParserASCII.y docu.ascii makeascii pd2ascii \
        INSTALL COPYRIGHT renumber \Figures
```

References

[ASU86] Aho, A.V., R. Sethi, and J.D. Ullman, Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

[Gü95] Güting, R.H., Integrating Programs and Documentation. FernUniversität Hagen, Informatik-Report 182, May 1995.

[La86] Lamport, L., LaTeX: A Document Preparation System. User's Guide & Reference Manual. Addison-Wesley, 1986.

[SUN88] Sun Microsystems, Programming Utilities and Libraries. User Manual. Sun Microsystems, 1988.

Anhang B. File PDNestedText.h (Quelltext, PD-File)

```
/************************************************
//[x] [$\times $]
//[->] [$\rightarrow $]
//paragraph [2] verse:  [\begin{verse}] [\end{verse}]
```

2 The Module NestedText

2.1 Definition Part

(File ~PDNestedText.h~)

This module allows one to create nested text structures and to write them to standard output. It provides in principle a data type ~listexpr~ (representing such structures) and operations:

```
[2]     atom: string [x] int [->] listexpr              \\
        atomc: string [->] listexpr                     \\
        concat: listexpr [x] listexpr [->] listexpr     \\
        print: listexpr [->] e                          \\
        copyout: listexpr [->] string [x] int           \\
        release-storage                                 \\
```

However, for use by ~lex~ and ~yacc~ generated lexical analysers and parsers which only allow to associate integer values with grammar symbols, we represent a ~listexpr~ by an integer (which is, in fact, an index into an array for nodes). Hence we have a signature:

```
[2]     atom: string [x] int [->] int                   \\
        atomc: string [->] int                          \\
        concat: int [x] int [->] int                    \\
        print: int [->] e                               \\
        copyout: int [->] string [x] int                \\
        release-storage                                 \\
```

The module uses two storage areas. The first is a buffer for text characters, it can take up to STRINGMAX characters, currently set to 30000. The second provides nodes for the nested list structure; currently up to NODESMAX = 30000 nodes can be created.

The operations are defined as follows:

```
----    int atom(char *string, int length)
----
```

List expressions, that is, values of type ~listexpr~ are either atoms or lists. The function ~atom~ creates from a character string ~string~ of length ~length~ a list expression which is an atom containing this string. Possible errors: The text buffer or storage space for nodes may overflow.

```
----    int atomc(char *string)
----
```

The function ~atomc~ works like ~atom~ except that the parameter should be a null-terminated string. It determines the length itself. To be used in particular for string constants written directly into the function call.

```
----    int concat(int list1, int list2)
----
```

Concats the two lists; returns a list expression representing the concatenation. Possible error: the storage space for nodes may be exceeded.

```
----    print(int list)
----
```

Writes the character strings from all atoms in ~list~ in the right order to standard output.

```
----    copyout(int list, char *target, int lengthlimit)
----
```

Copies the character strings from all atoms in ~list~ in the right order into a

string variable ~target~. Parameter ~lengthlimit~ ensures that the maximal available space in ~target~ is respected; an error occurs if the list expression ~list~ contains too many characters.

---- release_storage()

Destroys the contents of the text and node buffers. Should be used only when a complete piece of text has been recognized and written to the output. Warning: Must not be used after pieces of text have been recognized for which the parser depends on reading a look-ahead token! This token will be in the text and node buffers already and be lost. Currently this applies to lists.

The following is what is technically exported from this file:

```
************************************/

int atom(), atomc(), concat();
int print(), copyout(), release_storage(),
show_storage();                 /* show_storage used only for testing */
```

Anhang C. File PDNestedText.h.tex (aus PDNestedText.h generierter LaTeX-Code)

```
\documentclass[11pt]{article}
\usepackage{epsfig, a4wide, xspace}
\parindent0em
\parskip0.8ex plus0.4ex minus 0.4ex
\setcounter{secnumdepth}{4}
\setcounter{tocdepth}{3}
\fussy
\begin{document}
\section {The Module NestedText}

\subsection {Definition Part}

(File {\em PDNestedText.h\/})

This module allows one to create nested text structures and to write them to
standard output. It provides in principle a data type {\em listexpr\/}
(representing such structures) and operations:

\begin{verse}     atom: string $\times $ int $\rightarrow $ listexpr
  \\
        atomc: string $\rightarrow $ listexpr                       \\
        concat: listexpr $\times $ listexpr $\rightarrow $ listexpr     \\
        print: listexpr $\rightarrow $ e                            \\
        copyout: listexpr $\rightarrow $ string $\times $ int           \\
        release-storage                                  \\\end{verse}

However, for use by {\em lex\/} and {\em yacc\/} generated lexical analysers
and parsers which only allow to associate integer values with grammar symbols,
we represent a {\em listexpr\/} by an integer (which is, in fact, an index into
an array for nodes). Hence we have a signature:

\begin{verse}     atom: string $\times $ int $\rightarrow $ int
  \\
        atomc: string $\rightarrow $ int                            \\
        concat: int $\times $ int $\rightarrow $ int                    \\
        print: int $\rightarrow $ e                                 \\
        copyout: int $\rightarrow $ string $\times $ int                \\
        release-storage                                  \\\end{verse}

The module uses two storage areas. The first is a buffer for text characters,
it can take up to STRINGMAX characters, currently set to 30000. The second
provides nodes for the nested list structure; currently up to NODESMAX = 30000
nodes can be created.

The operations are defined as follows:

\hspace{0.9cm} \rule{2in}{0.1pt}
{\small \begin{verbatim}
        int atom(char *string, int length)
\end{verbatim}}
\hspace{0.9cm} \rule{2in}{0.1pt}


List expressions, that is, values of type {\em listexpr\/} are either atoms or
lists. The function {\em atom\/} creates from a character string {\em string\/}
of length {\em length\/} a list expression which is an atom containing this
string. Possible errors: The text buffer or storage space for nodes may
overflow.

\hspace{0.9cm} \rule{2in}{0.1pt}
{\small \begin{verbatim}
        int atomc(char *string)
\end{verbatim}}
\hspace{0.9cm} \rule{2in}{0.1pt}


The function {\em atomc\/} works like {\em atom\/} except that the parameter
should be a null-terminated string. It determines the length itself. To be used
in particular for string constants written directly into the function call.

\hspace{0.9cm} \rule{2in}{0.1pt}
```

{\small \begin{verbatim}
        int concat(int list1, int list2)
\end{verbatim}}
\hspace{0.9cm} \rule{2in}{0.1pt}

Concats the two lists; returns a list expression representing the concatenation. Possible error: the storage space for nodes may be exceeded.

\hspace{0.9cm} \rule{2in}{0.1pt}
{\small \begin{verbatim}
        print(int list)
\end{verbatim}}
\hspace{0.9cm} \rule{2in}{0.1pt}

Writes the character strings from all atoms in {\em list\/} in the right order to standard output.

\hspace{0.9cm} \rule{2in}{0.1pt}
{\small \begin{verbatim}
        copyout(int list, char *target, int lengthlimit)
\end{verbatim}}
\hspace{0.9cm} \rule{2in}{0.1pt}

Copies the character strings from all atoms in {\em list\/} in the right order into a string variable {\em target\/}. Parameter {\em lengthlimit\/} ensures that the maximal available space in {\em target\/} is respected; an error occurs if the list expression {\em list\/} contains too many characters.

\hspace{0.9cm} \rule{2in}{0.1pt}
{\small \begin{verbatim}
        release_storage()
\end{verbatim}}
\hspace{0.9cm} \rule{2in}{0.1pt}

Destroys the contents of the text and node buffers. Should be used only when a complete piece of text has been recognized and written to the output. Warning: Must not be used after pieces of text have been recognized for which the parser depends on reading a look-ahead token! This token will be in the text and node buffers already and be lost. Currently this applies to lists.

The following is what is technically exported from this file:

{\small \begin{quote} \begin{verbatim}
int atom(), atomc(), concat();
int print(), copyout(), release_storage(),
show_storage();   /* show_storage used only for testing */

\end{verbatim} \end{quote}}

\end{document}