PHILIPPS-UNIVERSITÄT MARBURG
Fachbereich Mathematik und Informatik
Prof. Dr. R. Loogen
J.Berthold

D-35032 Marburg
Hans Meerwein Straße

1.Dezember 2005

6. Übung zu „Grundlagen des Compilerbaus", WS 2005/06

Abgabe schriftlicher Aufgaben: **Do, 8.Dezember 2005 (vor der Vorlesung)**
Besprechung mündlicher Aufg.: **ab 5.Dezember 2005 in der Übung**

Mündliche Aufgaben

6.1 LR(k)-Grammatiken

(a) Zeigen Sie: Eine mehrdeutige Grammatik G ist für kein $k \in \mathbb{N}$ aus $LR(k)$.

(b) Kann eine Grammatik G mit ε-Produktionen aus $LR(0)$ sein?

6.2 Bottom-Up-Parsing

Gegeben sei die Grammatik aus Aufgabe 5.4, welche für das LR-Parsing zusätzlich startsepariert ist.

Geben Sie die Konfigurationsfolgen eines nichtdeterministischen Bottom-Up-Analyseautomaten für die Eingabe `(((),(a)),())` an. Gibt es Stellen, an denen Übergänge nichtdeterministisch bestimmt werden?

$$
\begin{array}{lll}
G: & S' \rightarrow & S \\
 & S \rightarrow & (a) \mid () \mid (T) \\
 & T \rightarrow & T, S \mid S
\end{array}
$$

Schriftliche Aufgaben

6.3 Top-Down-Parser für eine While-Sprache (8 Punkte)

Die Grammatik in Abb. 1 beschreibt eine While-Sprache ähnlich wie auf Blatt 4 und eignet sich für Top-Down-Parsing.

(a) Bestimmen Sie die Lookahead-Mengen und erstellen Sie eine LL(1)-Analysetabelle. / 3

(b) Programmieren Sie

- mit den Parser-Kombinatoren der Vorlesung / 5

oder • in einer Programmiersprache Ihrer Wahl

einen *Recursive-Descent*-Parser[1] für diese While-Sprache

Ihr Parser soll die Ausgabe des Scanners aus Aufgabe 5.4 (`[Token]`) einlesen und einen abstrakten Syntaxbaum für die Eingabe aufbauen bzw. bei Fehlern eine Meldung mit Zeilennummer und dem falschen Symbol für den *ersten* Fehler ausgeben.

Auf der Webseite der Vorlesung finden Sie eine Haskell-Datei `ProgType.hs` mit Datentypdefinitionen für einen abstrakten Syntaxbaum der While-Sprache, zwei (fehlerhafte) Testeingaben sowie bei Bedarf ein passendes Scannermodul.

---

[1] Das sog. *recursive-descent*-Parsing ist ein spezielles Implementierungsverfahren des Top-Down-Parsing mit einer Implementierungssprache, welche Rekursion erlaubt. Es existiert für jedes Nonterminal eine eigene Parse-Funktion, der Parser startet mit derjenigen des Startsymbols. Der Stack des TDA wird damit durch die rekursiven Funktionsaufrufe ersetzt, die ohnehin mit Hilfe eines Stacks verwaltet werden.

Bitte wenden!

program	$\rightarrow$	**program** Id '{' *dec stmts* '}'
dec	$\rightarrow$	**var** *decls*
decls	$\rightarrow$	*varlist* ':' *type A*
type	$\rightarrow$	**int** | **bool**
A	$\rightarrow$	';' *decls* | ε
varlist	$\rightarrow$	Id *B*
B	$\rightarrow$	',' Id *B* | ε
stmt	$\rightarrow$	Id ':=' *expr*
	|	**print** *expr*
	|	**if** *cond* **then** *stmt*
	|	**while** *cond* **do** *stmt*
	|	'{' *stmts* '}'
stmts	$\rightarrow$	*stmt C*
C	$\rightarrow$	';' *stmts* | ε
cond	$\rightarrow$	*expr* RelOp *expr* D
D	$\rightarrow$	'&' *cond* | '|' *cond* | ε
expr	$\rightarrow$	(gemäß Beispiel G'_{AE} der Vorlesung)

Bezeichner (Id), Zahlen (Num) und Operatoren (*Op) seien wie auf Blatt 4 beschrieben.
Die Sprache kann außerdem Zeilenkommentare erlauben, welche mit '//' eingeleitet werden.

Abbildung 1: Grammatik für eine While-Sprache

6.4 Umgang mit Parse-Fehlern

4 Punkte

Ein Parser, der gleich beim ersten Fehler die Erkennung abbricht, ist für die praktische Arbeit sehr unkomfortabel: Pro Parse wird nur ein einziger Fehler gemeldet, obwohl das Programm unter Umständen noch weitere (davon unabhängige, also gleichzeitig erkennbare) Fehler enthält.

Ein benutzerfreundlicher Parser versucht daher, den Fehler zu melden, aber die Erkennung trotzdem fortzusetzen. Falls Nonterminal N erkannt werden soll, aber der look-ahead keines der für N erwarteten Token ist, werden Token verworfen, bis ein Element aus $follow(N)$ erreicht ist. Danach wird die Erkennung fortgesetzt, als ob N erkannt worden wäre.

(a) Programmieren Sie einen Parser, der so lange Token verwirft, bis ein Token einem der erwarteten Werte (Parameter 1) entspricht. Sobald ein erwarteter Wert in der Eingabe erscheint, wird das Resultat (Parameter 2) geliefert, die Eingabe dabei aber nicht konsumiert.

```
skipUntil :: Eq tok => [tok] -> tree -> Parser tok tree
skipUntil expected res  []   = [] -- EOF, aufgeben
skipUntil expected res input ...
```

(b) Bauen Sie eine entsprechende Fehlerbehandlung in den Parser `AETree.hs` für arithmetische Ausdrücke ein, den Sie auf der Vorlesungsseite finden.

Testen Sie Ihren Parser mit einer fehlerhaften Eingabe, etwa `a*a+*(a+aa)*a*(a+)`. Wie könnten Folgefehler entstehen?