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

D-35032 Marburg
Hans Meerwein Straße
Lahnberge

1. Dezember 2003

Übungen zu „Grundlagen des Compilerbaus“, WS 2003/04

Nr. 6 , Abgabe und Besprechung: 8. Dezember in der Übung

Mündliche Aufgaben

6 .1 **Reduktion von Grammatiken, nützliche Symbole**
Betrachten Sie die folgende kontextfreie Grammatik:

$$\begin{aligned} G: \quad S &\to AB \mid CD \\ A &\to a \mid \varepsilon \\ B &\to BC \\ C &\to DD \\ D &\to E \mid \varepsilon \\ E &\to D \mid b \end{aligned}$$

Identifizieren Sie unnütze Symbole der Grammatik (nicht erreichbare und nicht produktive Nonterminale). Reduzieren Sie die Grammatik (begründen Sie Ihre Vorgehensweise). Ist die resultierende Grammatik eine LL(1)-Grammatik?

6 .2 **Grammatiken** $G \notin LL(k)$

(a) Gegeben sei die Grammatik $G = (\{S\}, \{a, b\}, P, S)$ mit

$$P = \{S \to aSab \mid aSabb \mid b\}$$

Zeigen Sie: Es existiert *kein* k mit $G \in LL(k)$.

(b) Geben Sie weitere Beispiele von Grammatiken G_i mit $G_i \notin LL(k)\ \forall k \in I\!N$.

Schriftliche Aufgaben

6 .3 **WHILE-Sprache** 8 Punkte

Gegeben sei die nebenstehende Grammatik für eine While-Sprache.

(a) Wie unterscheidet sich die von dieser Grammatik erzeugte Sprache von der Sprache aus Aufgabe 4.4 ? / 2

(b) Bestimmen Sie die Lookahead-Mengen aller Regeln und transformieren Sie die Grammatik nötigenfalls in eine äquivalente LL(1)-Grammatik. Erstellen Sie die Analysetabelle zur modifizierten Grammatik. / 3

Grammatik für While-Programme:

program	→	**program** Id **{ var** Id **;** *A* **}**
A	→	Id *B*
	|	*stmt'*
B	→	**;** *A*
	|	**:=** *expr*
stmt'	→	**{** *stmts* **}**
	|	*ifthen*
	|	*whiledo*
stmt	→	Variable **:=** *expr*
	|	**print** *expr*
	|	*stmt'*
stmts	→	*stmts* **;** *stmt*
	|	*stmt*
ifthen	→	**if** *cond* **then** *stmt*
whiledo	→	**while** *cond* **do** *stmt*
cond	→	*expr* RelOp *expr*
expr	→	Id
	|	Num
	|	**(** *expr* AOp *expr* **)**

Bezeichner (Id), Zahlen (Num) und Operatoren (*Op) seien wie in Aufgabe 4.4.

(c) Programmieren Sie, ausgehend von dieser Grammatik, einen *Recursive-Descent*-Parser[1] mit look-ahead-Mengen in einer Programmiersprache Ihrer Wahl. / 3

Der Parser sollte die Ausgabe des Scanners aus Aufgabe 4.4 (`[Token]`) einlesen können, die Syntax überprüfen und bei Fehlern eine Meldung mit Zeilennummer und dem falschen Symbol für den *ersten* Fehler ausgeben.

6 .4 Starke LL(k)-Grammatiken

4 Punkte

Für reduzierte kontextfreie Grammatiken sei die *starke LL(k)-Eigenschaft* (SLL(k)) folgendermaßen definiert:

Sei $G = (N, \Sigma, S, P) \in CFG$ reduziert.

$$G \in \mathrm{SLL}(k) \quad :\Longleftrightarrow \quad \forall A \in N; \beta, \gamma \in \Sigma^* \text{ gilt:}$$
$$A \to \beta,\ A \to \gamma \in P \wedge \beta \neq \gamma$$
$$\Longrightarrow \mathit{first}_k(\beta\ \mathit{follow}_k(A)) \cap \mathit{first}_k(\gamma\ \mathit{follow}_k(A)) = \emptyset$$

Ein Satz der Vorlesung zeigt, daß SLL(1) = LL(1) ist. Zeigen Sie am Beispiel der folgenden Grammatik, daß dies nicht verallgemeinerbar ist.

G: $S \to aAab \mid bAbb$
$A \to a \mid \epsilon$

(Bestimmen Sie $k, k_s \in I\!N$ mit $G \in LL(k), G \in SLL(k_s)$)

[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.