Prof. Dr. R. Loogen, M. Dieterle
Fachbereich Mathematik und Informatik
Hans-Meerwein-Straße
D-35032 Marburg

18. November 2009

Philipps Universität Marburg

# Übungen zu „Grundlagen des Compilerbau“, Winter 2009/10

**Nr. 5, Abgabe der Aufgaben: 25. November 2009 vor der Vorlesung**

---

## Aufgaben

### 5.1 Starke LL(k)-Grammatiken — 3 Punkte

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

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

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

Ein Satz der Vorlesung zeigt, daß SLL(1) = LL(1) ist. Dies ist nicht auf beliebige $k$ verallgemeinerbar!

Zeigen Sie, dass für die folgende Grammatik gilt: $G \in LL(2) \setminus SLL(2)$

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

### 5.2 Ein Parserkombinator für Folgen von Paaren *(Programmieraufgabe)* — 3 Punkte

(a) Definieren Sie einen Parserkombinator: / 1

```
alternating :: Parser tok a → Parser tok b → Parser tok [(a,b)]
```

um eine alternierende Folge zweier verschiedener Elemente zu erkennen, in der auf ein Element erster Art immer ein Element zweiter Art folgt. Die Parameter des Kombinators sind Parser für die Elemente, der Parser liefert die erkannte Sequenz als Liste von Elementpaaren.

(b) Definieren Sie mit `alternating` einen Parser, der eine durch einzelne Leerzeichen getrennte und mit einem Leerzeichen beendete Liste von Initialisierungen *Name*=*Wert* erkennt, wobei *Name:* `[a−z]+`, *Wert:* `[0−9]+`. / 2

Bitte wenden!

5.3 **Top-Down-Parser für eine While-Sprache** *(Programmieraufgabe)* 6 Punkte

Die Grammatik in Abb. 1 beschreibt eine While-Sprache ähnlich wie auf Blatt 3 und eignet sich für Top-Down-Parsing. Die Sprache ist allerdings nur partiell definiert, ab dem Level der Ausdrücke wurde die Sprache "flachgeklopft". Ausdrücke werden als Terminalsymbole (Token) behandelt.

**Aufgabe:** Programmieren Sie mit den Parser-Kombinatoren der Vorlesung einen *Recursive-Descent*-Parser für diese While-Sprache, der mit einem Lookahead von 1 deterministisch arbeitet. Ihr Parser soll die Ausgabe des Scanners[1] (`[Token]`) einlesen und einen abstrakten Syntaxbaum für die Eingabe aufbauen bzw. bei Fehlern eine Meldung mit dem falschen Symbol für den *ersten* Fehler ausgeben.

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

| **Grammatik** | | | $la_1$**-Mengen** |
|---|---|---|---|
| *program* | → | **program** Id '{' *dec stmts* '}' | |
| *dec* | → | **var** *decls* | |
| *decls* | → | *varlist* ':' *type A* | |
| *type* | → | **int** | {**int**} |
| | \| | **bool** | {**bool**} |
| *A* | → | ';' *decls* | {';'} |
| | \| | $\varepsilon$ | {Id, **print**, **if**, **while**, '{'} |
| *varlist* | → | Id *B* | |
| *B* | → | ',' Id *B* | {','} |
| | \| | $\varepsilon$ | {':'} |
| *stmt* | → | Id ':=' Expr | {Id} |
| | \| | **print** Expr | {**print**} |
| | \| | **if** Expr **then** '{'*stmts*'}' *maybeElse* | {**if**} |
| | \| | **while** Expr **do** *stmt* | {**while**} |
| | \| | '{' *stmts* '}' | {'{'} |
| *maybeElse* | → | **else** '{' *stmts* '}' | {**else**} |
| | \| | $\varepsilon$ | {';', '}'} |
| *stmts* | → | *stmt C* | |
| *C* | → | ';' *stmts* | {';'} |
| | \| | $\varepsilon$ | {'}'} |

Bezeichner (Id) seien wie auf Blatt 3 beschrieben.

Abbildung 1: Grammatik mit $la_1$-Mengen für eine While-Sprache

[1] eine modifizierte Version des Scanners aus Aufgabe 3.2