PHILIPPS-UNIVERSITÄT MARBURG D-35032 Marburg
Fachbereich Mathematik und Informatik Hans Meerwein Straße
Prof. Dr. R. Loogen
J.Berthold

17.November 2005

4. Übung zu „Grundlagen des Compilerbaus“, WS 2005/06

Abgabe schriftlicher Aufgaben: Do, 24.November 2005 (vor der Vorlesung)
Besprechung mündlicher Aufg.: ab 21.November 2005 in der Übung

---

Mündliche Aufgaben

4.1 **Pragmatik und Syntax von Programmiersprachen**
Diskutieren Sie die Eigenschaften der in Abbildung 1 definierten Sprache. Welche Schwachpunkte sehen Sie? An welchen Stellen würden Sie, aus welchen Gründen, anders vorgehen?

Für eine kleine WHILE-Programmiersprache sei die Mikrosyntax wie folgt definiert:

- Eine Variable besteht aus einem Buchstaben, gefolgt von beliebig vielen Buchstaben und Ziffern. Schlüsselwörter dürfen nicht als Variablen verwendet werden.
- Eine Zahl (NumVal) besteht entweder aus der Ziffer 0, oder aus einer nicht-leeren Ziffernfolge, die nicht mit einer 0 beginnt. Eine Zahl kann ein Vorzeichen + oder - haben.
- Ein Wahrheitswert (BoolVal) ist entweder True oder False.
- Arithmetische Operatoren (ArithOp): + – * /
- Relationale Operatoren (RelOp): = != < <= > >=

program	→	**program** Variable '{' *dec stmts* '}'
dec	→	**var** *decls*
decls	→	*decl* ';' *decls*
decls	→	*decl* ';'
decl	→	*varlist* ':' *type*
varlist	→	Variable ',' *varlist*
varlist	→	Variable
type	→	**int**
type	→	**bool**
stmts	→	*simple* ';' stmts
stmts	→	*simple*
simple	→	Variable ':=' *expr*
simple	→	**print** *expr*
simple	→	*cond*
simple	→	*whiledo*
cond	→	**if** *expr* **then** *stmts*
cond	→	**if** *expr* **then** *stmts* **else** *stmts*
whiledo	→	**while** '(' *expr* ')' **do** *stmts*
expr	→	Variable
expr	→	NumVal
expr	→	BoolVal
expr	→	*expr* ArithOp *expr*
expr	→	*expr* RelOp *expr*

Abbildung 1: Syntax einer While-Sprache in Form kontextfreier Grammatikregeln

Bitte wenden!

4.2 **Reduktion von Grammatiken, nützliche Symbole**

(a) Identifizieren Sie unnütze Symbole der Grammatik (nicht erreichbare und nicht produktive Nonterminale).

(b) Reduzieren Sie die Grammatik (begründen Sie Ihre Vorgehensweise).

(c) Ist die resultierende Grammatik eine LL(1)-Grammatik?

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

Schriftliche Aufgaben

4.3 **(Erweiterte) Backus-Naur-Form** 4 Punkte

Die Backus-Naur-Form BNF (auch: Backus-Normalform) ist eine gebräuchliche Form der Definition kontextfreier Sprachen (insbes. Programmiersprachen hinsichtlich der Syntax). Üblicherweise verwendet man eine erweiterte Form EBNF, in der die folgenden Definitionen gelten:

- Produktionsregeln haben syntaktisch die Form: $X = \alpha$;
  α besteht aus beliebig vielen Nonterminalen (Elementen auf linken Regelseiten) und Terminalen (Scanner-Token), ein Semikolon kennzeichnet das Ende einer Regel.
- Alternativen werden mit | zusammengefasst.
- Optional auftretende Elemente stehen in eckigen Klammern.
- Beliebig oft wiederholbare Elemente stehen in geschweiften Klammern.
- Runde Klammern dienen nur der Gruppierung.
- Um zwischen diesen Metazeichen der EBNF und Terminalsymbolen zu unterscheiden, stehen letztere immer in Anführungszeichen. Terminale aus mehreren Zeichen stehen auch oft in Fettdruck (was die Lesbarkeit erheblich verbessert).

Beispiel:

```
ZifferAußerNull   = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
Ziffer            = "0" | ZifferAußerNull ;
NatuerlicheZahl   = ZifferAußerNull { Ziffer } ;
Waehrung          =  EUR | "$" | "£" ;
Betrag            = ( "0" | [ "-" ] NatuerlicheZahl ) Waehrung;
```

(a) Beschreiben Sie, wie eine Syntaxbeschreibung in EBNF systematisch in eine äquivalente kontextfreie Grammatik übersetzt werden kann.

(b) Geben Sie die Syntax der While-Sprache aus Abb. 1 in EBNF an. Versuchen Sie, mit möglichst wenig Nonterminalen auszukommen.

4.4 **Scanner für eine imperative Sprache** 8 Punkte

(a) Geben Sie sinnvolle Symbolklassen für die lexikalische Beschreibung der While-Sprache aus Abbildung 1 an. Definieren Sie dann einen Haskell-Datentyp `Token` analog zu Ihren Symbolklassen. / 3

(b) Benutzen Sie den Scanner-Generator Alex, um einen Scanner zu erzeugen, der ein While-Programm in eine Liste von Symbolen umwandelt. / 5

Die Symbole sollen zusätzlich die Zeilennummer enthalten, in der sie stehen. Definieren Sie Token und Scanner entsprechend.