Übungen zu Vorlesung
Praktische Informatik I
Wintersemester 2000/01

Serie 5

Grammatiken und Sortieren

Aufgabe 1: Syntaxcheck von Java Konstrukten (30%)

Ziehen Sie das Kapitel 18 „Syntax" der Java Language Specification (zweite Ausgabe) sowie Kapitel 3.8 und 3.9 für die Definition von *Identifier* und Kapitel 3.10 für die Definition der Literale heran, um Java Konstrukte auf ihre syntaktische Korrektheit zu überprüfen:

1. Handelt es sich bei dem Programmfragment
   ```
   for(int i=5, i<9; j<20; i++);
   ```
   um ein korrektes *Statement* ?

2. Handelt es sich bei dem Programmfragment
   ```
   {
    if (x==y); {
     x++;
    } else x=y;
   }
   ```
   um einen korrekten *Block*?

Geben Sie bei beiden Konstrukten genau an, welche(s) Zeichen(ketten) von welcher (Kombination von) Produktionsregel(n) erkannt werden (dabei brauchen die vorkommenden Literale und Identifier *nicht* in ihre Einzelteile aufgebrochen zu werden) und geben Sie ebenfalls an, in welcher Regel ein eventueller Fehler auftritt.
Auf der PI1–WWW–Seite befindet sich ein kleines LaTeX Beispiel, welches zeigt, wie die Lösung zu einem derartigen Problem gesetzt werden könnte.

ACHTUNG: Die Syntaxdefinitionen in Kapitel 18 sind an einigen Stellen fehlerhaft bzw. inkonsistent notiert:

- In *Statement* muß *ExpressionStatement* durch *StatementExpression*; ersetzt werden.
- Manchmal bedeutet ein nachgestelltes *Opt*, daß es sich um ein optionales Nichtterminal handelt, welches dann den Namen ohne *Opt* trägt (z. B. bei *ForInitOpt* in *Statement*, wo dann das Nichtterminal *ForInit* gemeint ist) und manchmal ist das nachgestellte *Opt* einfach nur Teil des Namens (z. B. bei *BracketsOpt* in *VariableDeclaratorRest*).
- Es ist sehr schwierig, kursive geschweifte Klammern ({, }) zum Ausdruck der beliebig häufigen Wiederholung von normal gesetzten geschweiften Klammern zu unterscheiden. In *Block* sind die geschweiften Klammern terminale Symbole, in *BlockStatements* jedoch stehen sie für die beliebig häufige Wiederholung eines *BlockStatement* Nichtterminals.

Aufgabe 2: Quergestreifte Fliegende Teppiche (30%)

Die uralte Suleika ist die letzte Weberin in der Stadt Samarkand, die sich noch auf die Kunst des Webens von quergestreiften fliegenden Teppichen versteht. Als Suleika ihren Tod nahen fühlt, ruft sie ihre Tochter Hurinka zu sich, um ihr das Geheimnis ihrer Zunft anzuvertrauen. Dieses sind die geheimen Regeln, nach denen quergestreifte fliegende Teppiche gewebt werden:

1. Es gibt zwei Basismuster. Jedes Basismuster ist ein mögliches Teilmuster.
2. Es gibt ein vorne ergänzendes Muster. Aus einem gegebenen Teilmuster entsteht ein neues Teilmuster, wenn dieses Muster vorne angefügt wird.
3. Es gibt ein hinten ergänzendes Muster. Aus einem gegebenen Teilmuster entsteht ein neues Teilmuster, wenn dieses Muster hinten angefügt wird.
4. Das Teppichmuster ist eine beliebig lange Folge von Teilmustern.

Die Basismuster B1 und B2 sind:

```
B1 = ROSENROT LICHTBLAU LICHTBLAU HONIGGOLD
B2 = HONIGGOLD SCHNEEWEISS HONIGGOLD
```

Das vorn ergänzende Muster ist:

```
V = ROSENROT LICHTBLAU
```

Das hinten ergänzende Muster ist:

```
H = LICHTBLAU HONIGGOLD SCHNEEWEISS
```

Schreiben Sie eine Grammatik mit Hilfe von Syntaxdiagrammen, welche korrekte quergestreifte fliegende Teppiche beschreibt. Die Farben sollen dabei als terminale Symbole (Schlüsselwörter) betrachtet werden.
Geben Sie drei verschiedene korrekte Muster für quergestreifte fliegende Teppiche, so wie sie von Ihrer Grammatik erzeugt werden, an.
Verwenden Sie auch hier zur Dokumentation `latex`. Die Syntaxdiagramme können dabei mit dem Grafiktool `xfig` erstellt werden. Eine kurze Anleitung zum Einbinden von Bildern in LaTeX–Dokumente befindet sich auf der PI1–WWW–Seite.

Aufgabe 3: Bubblesort (40%)

Der Bubblesort-Algorithmus gilt als eines der am einfachsten zu implementierenden Sortierverfahren. Eine verbreitete Variante funktioniert folgendermaßen:

Durchlaufe immer wieder das gesamte zu sortierende Feld von vorne nach hinten und vertausche dabei benachbarte Elemente, falls diese nicht in der gewünschten Reihenfolge sind. Wenn bei einem Durchlauf kein Austausch mehr erforderlich ist, ist das Feld komplett sortiert und der Algorithmus beendet.

Implementieren Sie diesen Algorithmus in Java, um ein Array von Integer–Zahlen in aufsteigender Reihenfolge zu sortieren. Dabei sollen bei Programmstart sowohl die Anzahl der zu sortierenden Elemente als auch die Elemente selbst vom Benutzer eingegeben werden. Verwenden Sie hierfür die auf der

PI1–WWW–Seite liegende Klasse `Input.java`, welche es ermöglich, mit Hilfe der Methode `Input.readInt()` einen Integer Wert von der Tastatur einzulesen. Es ist dabei ausreichend, wenn die Datei `Input.class` im gleichen Verzeichnis liegt wie das von Ihnen entwickelte Programm. Ein kurzes Beispiel zur Verwendung dieser Inputklasse befindet sich in dem Programm `Test.java`, welches ebenfalls auf der PI1–WWW–Seite liegt.
Sowohl das eingelesene als auch das sortierte Array sollen von dem Programm auf dem Bildschirm in der Form `[x1, x2, ..., xn]` ausgegeben werden.

Abgabe: 18. – 22.12. nach den jeweiligen Praktika. Die Abgabe soll sowohl elektronisch (Programm-Quellcode zu Aufgabe 3) als auch in gedruckter Form (mit LaTeX gesetzter kommentierter und erläuterter Quellcode sowie die Lösungen zu Aufgaben 1 und 2) erfolgen!