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

D-35032 Marburg
Hans Meerwein Straße
Lahnberge

8. Dezember 2003

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

Nr. 7 , Abgabe und Besprechung: 15. Dezember in der Übung

Mündliche Aufgaben

7 .1 **Parserkombinatoren:** `bind` vs. `seqp`

Zur Sequentialisierung von Parsern wurde in der Vorlesung der Parserkombinator `bind` vorgestellt. Alternativ kann eine Sequentialisierung aber auch über `seqp` erfolgen:

```
seqp :: Parser tok a -> Parser tok b -> Parser tok (a,b)
seqp p1 p2 = \input -> [((v,w),input'') | (v,input')  <- p1 input
                                        , (w,input'') <- p2 input']
```

Sind `bind` und `seqp` gleich mächtig oder kann ein Kombinator durch den anderen ausgedrückt werden?

7 .2 **Verschachtelt geklammerte Ausdrücke**

Die nebenstehende Grammatik definiert korrekt geschachtelte Klammerstrukturen. Sowohl Korrektheit, als auch maximale Verschachtelungstiefe können mit einem geeigneten Parser festgestellt werden.

$$\begin{aligned} G : S \;&\rightarrow\; () \\ &\mid (S) \\ &\mid SS \end{aligned}$$

(a) Definieren Sie zunächst einen Kombinator, der Inhalte zwischen zwei definierbaren Begrenzungen erkennt:

Haskell Code

```
between :: Parser tok open -> Parser tok close
                -> Parser tok a -> Parser tok a
```

Dabei seien die Parameter von `between` ein Parser für die öffnende Begrenzung, einer für die schließende Begrenzung und einer für den Inhalt. Zurückgegeben wird nur das Parse-Ergebnis für den Teil zwischen den Begrenzungen.

(b) Mit `between` können Sie einen rekursiven Parser `bracket :: Parser Char Int` definieren, der eine Folge von öffnenden und schließenden Klammern erkennt und als Ergebnis die maximale Verschachtelungstiefe zurückgibt.
Sie sollten die Linksrekursion in Regel 3 vorher durch eine geeignete Hilfskonstruktion ersetzen. Die Erzeugung einer LL(1)-Grammatik ist aber nicht erforderlich, da die Kombinatoren stets alle Parsing-Alternativen betrachten.

Schriftliche Aufgaben

7 .3 Transformation von Grammatiken

4 Punkte

(a) Eine Grammatik sei durch die folgenden Produktionen gegeben:

$$\begin{aligned} G: \quad S &\to a \mid (T) \\ T &\to T, S \mid S \end{aligned}$$

Eliminieren Sie die Linksrekursion. Ist die resultierende Grammatik in LL(1)?

(b) Die folgende Grammatik G' ist äquivalent zu G:

$$\begin{aligned} G': \quad S &\to a \mid (T') \\ T' &\to S, T' \mid S \end{aligned}$$

Führen Sie eine Linksfaktorisierung für G' durch.

(c) Bestimmen Sie die look-ahead-Mengen der Regeln für die in b) erhaltene Grammatik. Ist sie aus LL(1)?

7 .4 Parserkombinatoren

5 Punkte

(a) Schreiben Sie einen Parser

```
nat :: Parser Char Int
```

für natürliche Zahlen.

(b) Erweitern Sie `nat` zu einem Parser

```
int :: Parser Char Int
```

, der ganze Zahlen erkennt.

(c) Oft soll ein Parser eine Reihe von Objekten erkennen, die durch Separatoren voneinander getrennt sind, zum Beispiel:

```
Stmt1; Stmt2; Stmt3; Stmt4
```
oder
```
[55, 44, 1, -99]
```

Schreiben Sie einen Parserkombinator

```
sepBy :: Parser tok a -> Parser tok b -> Parser tok [a]
```

, der eine *nichtleere* Reihe von durch Separatoren getrennten Objekten erkennt und nur die Objekte zurückliefert. Als Argumente werden dabei ein Parser zum Erkennen der Objekte und ein Parser zum Erkennen der Separatoren mitgegeben. Erzeugen Sie mit `sepBy` einen Parser, der eine Liste von ganzen Zahlen erkennt.

7 .5 LR(0)-Mengen

3 Punkte

Bestimmen Sie zur folgenden Grammatik G die Menge LR(0)(G):

S → CC
C → cC | d

Ist G LR(0)?

Auf der Vorlesungsseite finden Sie die in der Vorlesung vorgestellte Haskell-Datei *parser.hs* mit Definitionen für Parser-Kombinatoren. Achtung: das dort definierte Modul heißt nicht `Parser`, sondern `AE_Parser`.