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

31.Oktober 2007

3. Übung zu „Grundlagen des Compilerbaus“, WS 2007/08

Abgabe der Aufgaben: Mi, 7.November 2007 (vor der Vorlesung)

Reguläre Ausdrücke sollen in Haskell durch folgende Definitionen erfaßt werden:

```
infixl 4 :|                  -- Auswahl
infixr 5 :%                  -- Sequenz

data RExp = Lam              -- Lambda
          | Ch Set           -- Buchstabe(n)
          | RExp :% RExp     -- Sequenz
          | RExp :| RExp     -- Auswahl
          | Star RExp        -- Stern

type Set = Char -> Bool
```

Die ersten beiden Zeilen definieren Assoziativität und Präzedenz der beiden Infix-Konstruktoren zur Darstellung von Sequenz und Auswahl. Mengen von Zeichen werden über ein Prädikat `Set` angegeben. Die Funktion `alphabet` zeigt, wie das einem gegebenen regulären Ausdruck zugrunde liegende Alphabet bestimmt werden kann:

```
alphabet :: RExp -> Alphabet
alphabet Lam         = []
alphabet (Ch p)      = [c | c <- bereich, p c]
alphabet (re :% re') = union (alphabet re) (alphabet re')
alphabet (re :| re') = union (alphabet re) (alphabet re')
alphabet (Star re)   = alphabet re

bereich :: [Char]
bereich = ['\33'..'\127']

union :: Alphabet -> Alphabet -> Alphabet
union a a' = removeDupl (a ++ a')
  where removeDupl []     = []
        removeDupl (c:cs) = if (elem c cs) then     removeDupl cs
                                           else c : removeDupl cs
```

Im Anschluß werden wir mit NFAs arbeiten, deren Definition der DFA-Definition des letzten Übungsblattes ähnelt:

```
type SigmaEps    = String                          -- Epsilon: []
type Delta state = state -> SigmaEps -> [state]
type NFA  state  = ([state], Alphabet, Delta state, state, [state])
```

Aufgaben

3.1 Regulärer Ausdruck → NFA

5 Punkte

(a) Drücken Sie den regulären Ausdruck aus Aufgabe 2.1 in Haskell aus.

(b) Zur Erzeugung des NFA soll nach dem Satz von Kleene vorgegangen werden; dazu ist es notwendig, an mehreren Stellen der Implementierung die Zustandsmenge eines NFA durch eine Verschiebung des Indexbereiches zu verändern. Zustände sollen zunächst über ganze Zahlen repräsentiert werden, so daß eine Indexverschiebung um n Stellen z.B. folgendermaßen aussieht:

```
[1, 2, 3] -> [1+n, 2+n, 3+n]
```

Implementieren Sie die Funktion

```
shift :: Int -> NFA' Int -> NFA' Int
type Delta state = state -> String -> [state]
type NFA'  state = ([state], Delta state, state, [state])
```

Der Aufruf `shift n nfa` soll dann den entsprechend transformierten Automaten zurückliefern. Das Alphabet soll über `alphabet` erkannt werden.

(c) Implementieren Sie nun eine Funktion

```
r2n :: RExp -> NFA Int
```

die induktiv über den Satz von Kleene aus einem regulären Ausdruck einen entsprechenden NFA konstruiert.

Bitte wenden!

3.2 Scanner für eine funktionale Sprache

7 Punkte

Für eine kleine funktionale Programmiersprache sei die Mikrosyntax wie folgt definiert:

- Eine Variable (id) besteht aus einem **Klein**buchstaben, gefolgt von beliebig vielen Buchstaben und Ziffern. Schlüsselwörter dürfen nicht als Variablen verwendet werden.
- Ein Typbezeichner (id') besteht aus einem **Groß**buchstaben, gefolgt von beliebig vielen Buchstaben und Ziffern. Schlüsselwörter dürfen nicht als Typbezeichner verwendet werden.
- Eine Zahl (NumVal) besteht entweder aus der Ziffer 0, oder aus einer nichtleeren 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): == < <= > >=
- Boolesche Operatoren (BoolOp): || && not

Die Syntax sei durch folgende Regeln beschrieben:

$$
\begin{array}{lcl}
P & \rightarrow & \textbf{module}\ \text{id'}\ \textbf{where}\ L \\
 & | & L \\
L & \rightarrow & L\ \textbf{;}\ D \\
 & | & D \\
 & | & L\ \textbf{;}\ \text{id}\ \textbf{::}\ \text{id'}\ F \\
 & | & \text{id}\ \textbf{::}\ \text{id'}\ F \\
F & \rightarrow & \textbf{->}\ \text{id'}\ F \\
 & | & \varepsilon \\
D & \rightarrow & \text{id}\ A = E \\
A & \rightarrow & A\ \text{id} \\
 & | & \varepsilon \\
E & \rightarrow & E\ \text{ArithOp}\ E \\
 & | & E\ \text{RelOp}\ E \\
 & | & E\ \text{BoolOp}\ E \\
 & | & (E) \\
 & | & \text{NumVal} \\
 & | & \text{BoolVal} \\
 & | & A\ \text{id} \\
 & | & \textbf{if}\ E\ \textbf{then}\ E\ \textbf{else}\ E \\
 & | & \textbf{let}\ L\ \textbf{in}\ E
\end{array}
$$

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

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

(c) Erweitern Sie den Scanner und die Token-Definition, so dass die Symbole zusätzlich die Zeilennummer enthalten, in der sie stehen.