4. Übungsaufgabe zu
Fortgeschrittene Funktionale Programmierung
Themen: Parsing – Lexikalische und Syntaxanalyse
ausgegeben: 12.06.2006, fällig: 21.06.2006

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften zur Lösung der im folgenden angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `AufgabeFFP4.hs` in Ihrem Homeverzeichnis ablegen, wie gewohnt auf oberstem Niveau. Kommentieren Sie Ihre Programme aussagekräftig und benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

Wir betrachten die Programmiersprache **While**, deren Programme durch folgende Grammatik beschrieben seien:

```
Prog      ::= begin Stmt end
Stmt      ::= AssStmt | IfStmt | WhileStmt | CompStmt
AssStmt   ::= Idf := AExpr
IfStmt    ::= if Bexpr then Stmt else Stmt fi
WhileStmt ::= while Bexpr do Stmt od
CompStmt  ::= Stmt ; Stmt
```

Dabei sei `Idf` ein Repräsentant für einen beliebigen Identifikator, wobei wir annehmen, dass ein Identifikator aus einer nichtleeren Folge von Klein- und Großbuchstaben und Ziffern besteht, die mit einem Buchstaben beginnt. Die Menge der arithmetischen und Booleschen Ausdrücke werde durch folgende Grammatik für Ausdrücke beschrieben.

```
Expr   ::= AExpr | Bexpr

AExpr  ::= Term | AExpr Aop Term
Term   ::= Factor | Term Mop Factor
Factor ::= Opd | (AExpr)
Opd    ::= Zahl | Idf
Aop    ::= + | -
Mop    ::= * | /

Bexpr  ::= (Aexpr Relop Aexpr)
Relop  ::= = | /= | > | <
```

Dabei sei `Zahl` ein Repräsentant für beliebige Dezimalzahlen ohne Vorzeichen (also natürliche Zahlen).

- Schreiben Sie wahlweise einen monadischen oder Kombinatorparser, der angesetzt auf ein **While**-Programm die zugehörige Tokenfolge ausgibt. Mögliche Token sind (wobei `ZuwOp` für den Zuweisungsoperator `:=` steht):

  ```
  data Token = Id | ZuwOp | Num |
               OeffKlammer | SchliessKlammer |
               Plus | Minus | Mal | Durch |
               Gleich | Ungleich | Groesser | Kleiner |
               BeginSymb | EndSymb |
               IfSymb | ThenSymb | ElseSymb | FiSymb |
               WhileSymb | DoSymb | OdSymb |
               Err
               deriving Show
  ```

  Sehen Sie in Ihrer Lösung insbesondere eine Funktion `main1 :: String -> [Token]` vor, anhand derer die Arbeitsweise Ihres Parsers getestet werden kann. Das Token `Err` soll verwendet werden, wenn die Eingabezeichenreihe eine Teilzeichenreihe enthält, die nicht einem der anderen Token entspricht. Der Rest der Eingabezeichenreihe soll dann verworfen werden; `err` ist also das letzte Token in der Resultatliste der Funktion `main`.

- Schreiben Sie einen Parser, der ein **While**-Programm einliest und eine Liste von Strukturbäumen als Resultat ausgibt. Jeder Strukturbaum im Resultat entspricht dabei einer Anweisung des **While**-Programms.

  ```
  data STree = Seq [Tree]

  data Tree = AssStmt Idf AExpr |
              IfStmt BExpr Tree Tree |
              WhileStmt BExpr Tree

  data AExpr = Term | Ce AExpr Aop Term
  data Term = Factor | Ct Term Mop Factor
  data Factor = Opd | AExpr
  data Opd = Zahl | Idf

  data Aop = Plus | Minus
  data Mop = Mal | Durch
  ```

```
type Idf = String
type Zahl = Int

data BExpr = Cb Aexpr RelOp Aexpr

data RelOp = Gleich | Ungleich | Groesser | Kleiner
```

Sehen Sie in Ihrer Lösung insbesondere eine Funktion `main2 :: String -> STree` vor, anhand derer die Arbeitsweise Ihres Programms getestet werden kann. Ergänzen Sie dazu auch erforderliche `Show`-Direktiven. In Ihrer Lösung können Sie davon ausgehen, dass nur syntaktisch korrekte **While**-Programme zum Testen verwendet werden.