2. Übungsaufgabe zu
Fortgeschrittene Funktionale Programmierung
Themen: Parsing – Lexikalische und Syntaxanalyse
ausgegeben: 17.05.2004, fällig: 02.06.2004

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften zur Lösung der im folgenden angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `AufgabeFFP2.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**. Die Programme dieser Sprache sind durch folgende Grammatik beschrieben:

```
Prog ::= begin S end
S    ::= Idf := AExpr | skip | if Bexpr then S else S |
         while Bexpr do S | S ; S
```

Dabei sei `Idf` ein Repräsentant für einen beliebigen Identifikator. Wir nehmen an, dass Identifikatoren aus einer nichtleeren mit einem Klein- oder Großbuchstaben beginnenden Folge von Klein- und Großbuchstaben und Ziffern bestehen. Die Schlüsselwörter `if`, `then`, `else`, `while` und `do`, sowie der Standardbezeichner `skip` kommen dabei nicht als Identifikatoren vor. Die Menge der arithmetischen und Booleschen Ausdrücke in **while**-Programmen wird durch folgende Grammatik für Ausdrücke beschrieben, wobei `Zahl` ein Repräsentant für beliebige Dezimalzahlen ohne Vorzeichen (also natürliche Zahlen) sei.

```
Expr   ::= AExpr | Bexpr

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

Bexpr  ::= true | false | (Aexpr Relop Aexpr)
Relop  ::= == | <>
```

- Schreiben Sie nach dem Muster der Vorlesung einen Kombinator-Parser `lex`, der ein **while**-Programm einliest, lexikalisch analysiert und die zugehörige Tokenfolge als Liste zurückgibt. Die Eingabe soll dabei durch wiederholtes Zeilenlesen von `stdin` erfolgen. Das Ende der Eingabe werde dabei durch einen allein auf einer Zeile stehenden "." (Punkt) markiert. Dieser Abschlusspunkt wird nicht verarbeitet.

  Als Token sollen dabei verwendet werden: `Invalid, Plus, Minus, Mult, Div, True, False, Eq, Neq, Number, Var, If, Then, Else, While, Do, Ass, Seq, Lbrack, Rbrack, Begin` und `End`. Dabei werde der Zuweisungsoperator `:=` auf `Ass`, der Kompositionsoperator `;` auf `Seq`, die öffnende Klammer `(` auf `Lbrack` und die schließende Klammer `)` auf `Rbrack` abgebildet, für die übrigen Zeichenreihen und Token betrachten wir den Zusammenhang als evident. Führen Sie einen geeigneten Datentyp `Token` mit den entsprechenden nullstelligen Datenkonstruktoren ein.

  Sehen Sie zusätzlich eine Funktion `showToken` vor, die eine Liste von Token auf `stdout` ausgibt. Die sequentielle Komposition aus `lex` gefolgt von `showToken` soll also die Ausgabe des Tokenstroms des eingelesenen Programms bewirken.

- Erweitern Sie den Kombinator-Parser aus der vorigen Teilaufgabe zu einem Kombinator-Parser `syn`, der aus dem Strom der Eingabetoken den Syntaxbaum des zugrundeliegenden **while**-Programms aufbaut entsprechend der Grammatiken der Ausdrücke und der **while**-Programme. Führen Sie dafür einen Ihnen geeignet erscheinenden Datentyp zur Repräsentation von Syntaxbäumen ein und sehen Sie eine Funktion `showTree` zur Ausgabe von Syntaxbäumen vor. Die sequentielle Komposition aus `lex` gefolgt von `syn` gefolgt von `showTree` soll also die Ausgabe des Syntaxbaums des eingelesenen Programms auf `stdout` bewirken, wobei die Baumstruktur in der Ausgabe deutlich werden soll, d.h., sich wiederspiegeln soll.