Assignment 6
Advanced functional Programming
Topic: Parsing – Lexical and Syntactical Analysis
Issued on: 06/14/2007, due date: 06/25/2007

For this assignment a Haskell script named `AssFFP5.hs` shall be written offering functions which solve the problems described below. This file `AssFFP6.hs` shall be stored in your home directory, as usual on the top most level. Comment your programs meaningfully. Use constants and auxiliary functions, where appropriate.

Consider the programming language **Repeat**, whose programs are characterized by the following grammar:

```
Prog       ::= begin Stmt end
Stmt       ::= AssStmt | IfStmt | RepeatStmt | CompStmt
AssStmt    ::= Idf := AExpr
IfStmt     ::= if Bexpr then Stmt else Stmt fi
RepeatStmt ::= repeat Stmt until Bexpr taeper
CompStmt   ::= Stmt ; Stmt
```

We assume that `Idf` denotes an arbitrary identifier and that each identifier is a non-empty sequence of lower case and upper case letters and digits starting with a letter. The set of arithmetic and Boolean expressions is given by the following grammar for expressions.

```
Expr   ::= AExpr | Bexpr

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

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

We assume that `Numeral` denotes an unsigned decimal number (i.e., a natural number).

- Implement either a monadic parser or a combinator parser p. If p is applied to a **Repeat**-program, p yields the corresponding sequence

of tokens. Possible tokens are (where `AssOp` is used to denote the assignment operator `:=`):

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

Take care to implement in particular a function `main1 :: String -> [Token]` allowing to test the functioning of your parser. The token `Err` shall be used, if the input string contains a substring, which does not correspond to one of the tokens above. The remainder of the input string shall then be discarded; `err` is then the last token in the result list of the function `main`.

- Implement a parser, which reads a **Repeat**-program, and yields a list of syntax trees as the result. Each syntax tree occuring in the result shall correspond to one statement of the **Repeat**-program.

```
data STree = Seq [Tree]

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

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 Numeral = Int
```

```
data BExpr = Cb Aexpr RelOp Aexpr

data RelOp = Equal | Unequal
```

Take care to implement in particular a function `main2 :: String -> STree` allowing to test the functioning of your program. Add required `Show`-directives. You can assume that your program will only be tested with syntactically correct **Repeat**-programs.

Note: In case of any remaining name clashes rename identifiers in order resolve these clashes.