Assignment 5
Advanced functional Programming
Topic: Parsing – Lexical and Syntactical Analysis
Issued on: 05/11/2010, due date: 05/18/2010

For this assignment a Haskell script named `AssFFP5.hs` shall be written offering functions which solve the problems described below. This file `AssFFP5.hs` shall be stored in the home directory of your individual account (not of your group account), as usual on the top most level. Comment your programs meaningfully. Use constants and auxiliary functions, where appropriate.

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

```
Prog       ::= begin Stmt end
Stmt       ::= AssStmt | IfStmt | RepeatStmt | CompStmt | Skip
AssStmt    ::= Idf := AExpr
IfStmt     ::= if Bexpr then Stmt else Stmt fi
WhileStmt  ::= while Bexpr do Stmt od
Skip       ::= skip
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. Different language tokens (such as identifiers, keywords, etc.) can be separated by an arbitrary number of blanks, but at least by one. E.g. the string `doskip` represents an identifier, while `do skip` represents the keyword `do` followed by the empty statement `skip`, i.e., the longest possible token has to be taken. 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

  1. a combinator parser *pc* and
  2. a monadic parser *pm*

  If *pc* and *pm* are applied to a **While**-program, they yield the corresponding sequence of tokens. Possible tokens are (where `AssOp` is used to denote the assignment operator ":="):

```
data Token = Id | AssOp | Num |
             LeftParenth | RightParenth |
             Plus | Minus | Mult | Div |
             Equal | Unequal | Greater | Less |
             BeginSymb | EndSymb |
             IfSymb | ThenSymb | ElseSymb | FiSymb |
             WhileSymb | DoSymb | OdSymb |
             SkipSymb |
             SemicolonSymb |
             Err
             deriving Show
```

  Take care to implement in particular two functions `main_pc :: String -> [Token]` and `main_pm :: String -> [Token]` allowing to test the functioning of your parsers. The token `Err` shall be used by both parsers, 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 functions `main_pc` and `main_pm`.