Advanced Functional Programming: Assignment 5 (Wed, 05/22/2019)
Topic: Combinator and Monadic Parsing
Submission deadline: Wed, 05/29/2019 (3pm)

Regarding the deadline for the second submission: Please, refer to „Hinweise zu Organisation und Ablauf der Übung“ available at the homepage of the course.

Store all functions to be written for this assignment in a top-level file `assignment5.hs` of your group directory. Comment your program meaningfully; use auxiliary functions and constants, where reasonable.

1. We consider the imperative programming language While. The *concrete syntax* of While programs is given by the context-free grammar below, where non-terminals are enclosed in acute brackets (spitze Klammern), and terminal symbols are denoted by (sequences of) uppercase letters:

$$
\begin{array}{rcl}
<program> & ::= & \texttt{PROGRAM} <statement\_seq> \texttt{ .} \\
<statement\_seq> & ::= & <statement> \mid <statement>\texttt{;}<statement\_seq> \\
<statement> & ::= & <assignment> \mid <ifels> \mid <while> \mid \texttt{skip} \\
<assignment> & ::= & <identifier> \texttt{ := } <expr> \\
<ifels> & ::= & \texttt{IF } <pred\_expr> \texttt{ THEN } <statement\_seq> \texttt{ ELSE } <statement\_seq> \texttt{ FI} \\
<while> & ::= & \texttt{WHILE } <pred\_expr> \texttt{ DO } <statement\_seq> \texttt{ OD} \\
<pred\_expr> & ::= & <expr> \texttt{ = } <expr> \mid <expr> \texttt{ > } <expr> \\
<expr> & ::= & <term> \mid <expr> \texttt{ + } <term> \\
<term> & ::= & <factor> \mid <term> \texttt{ * } <factor> \\
<factor> & ::= & \texttt{(} <expr> \texttt{)} \mid <identifier> \mid <integer> \\
<identifier> & ::= & <char><char\_seq> \\
<char> & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid ... \mid \texttt{z} \\
<char\_seq> & ::= & \varepsilon \mid <char><char\_seq> \\
<integer> & ::= & <digit><digit\_seq> \mid \texttt{-}<digit><digit\_seq> \\
<digit> & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid ... \mid \texttt{9} \\
<digit\_seq> & ::= & \varepsilon \mid <digit><digit\_seq>
\end{array}
$$

   Identifiers are contiguous non-empty sequences of the lowercase letters `a`, `b`, `c`,..., `z`; integers are contiguous non-empty sequences of digits `0`, `1`,..., `9`, possibly with leading zeros, and possibly proceeded with the character `-` for negative integers. White space and line breaks might freely occur in While programs (except of course in reserved words, identifers, integers, and the assignment operator symbol `:=`).

   We complement the concrete syntax of While programs by a more concise *abstract syntax*, which is given by the following grammar, where non-terminals

are denoted by uppercase letters:

$$\begin{array}{lcl} P & ::= & S \\ S & ::= & S_1; S_2 \mid V = E \mid \text{ if } E\ S_1\ S_2 \mid \text{while } E\ S \mid \text{skip} \\ E & ::= & E_1 == E_2 \mid E_1 > E_2 \mid E_1 + E_2 \mid E_1 * E_2 \mid V \mid I \end{array}$$

Unlike the concrete syntax of WHILE programs, their abstract syntax does not contain white spaces and line breaks freely. The abstract syntax of WHILE programs does not contain any line breaks, and white space only (strictly limited) within if and while statements: Given an if statement, if, E, S_1, and S_2 are separated by exactly one blank each; analogously, given a while statement, while, E, and S are separated by exactly one blank each. White space other than that does not occur. Moreover, (superfluous) leading zeros in the representations of integers are removed as well as the symbol - if it is only followed by zeros. Hence, the abstract syntax of the WHILE language provides a unique representation of the integer value 0.

Implement a

1.1 combinator parser `parser1` (cf. Chapter 13.2)

```
type Parse1 a b = [a] -> [(b,[a])]
parser1   :: Parse1 Char String
topLevel1 :: Parse1 a b  -> [a] -> b
```

1.2 monadic parser `parser2` (cf. Chapter 13.3)

```
newtype Parse2 a = Parse (String -> [(a,String)])
parser2   :: Parse2 String
topLevel2 :: Parse2 a -> String -> a
```

such that `topLevel1` and `topLevel2` transform well-formed WHILE programs given in concrete syntax into abstract syntax, when called with `parser1` and `parser2`, respectively, and some input string. If the input string is not well-formed, `topLevel1` and `topLevel2` shall terminate with calling `error "parse unsuccessful"` (cf. function `topLevel`, Example 2, Chapter 13.2.5).

2. We introduce the following Haskell types for programs, statements, and expressions of WHILE programs allowing a tree-like representation of WHILE programs, called *abstract syntax tree* representation:

```
data P = P [S] deriving (Eq,Show)
data S = Ass E E
       | If E [S] [S]
       | While E [S]
       | Skip deriving (Eq,Show)
data E = Val Int
       | Idf Char
       | Add E E
       | Mul E E deriving (Eq,Show)
```

Implement a

2.1 combinator parser `parser3` (cf. Chapter 13.2)

```
parser3   :: Parse1 Char P
```

2.2 monadic parser `parser4` (cf. Chapter 13.3)

```
parser4   :: Parse2 P
```

such that `topLevel1` and `topLevel2` transform well-formed WHILE programs given in abstract syntax into abstract syntax trees, when called with `parser3` and `parser4`, respectively, and some input string. If the input string is not well-formed, `topLevel1` and `topLevel2` shall terminate with calling `error "parse unsuccessful"` (cf. function `topLevel`, Example 2, Chapter 13.2.5).

3. Identifying the non-terminals of the grammar defining the abstract syntax of WHILE programs with the set of all programs, statements, and expressions, respectively, i.e.:

P :	**PROG**	(set of all) programs
S :	**STMT**	(set of all) statements
E :	**EXPR**	(set of all) expressions
V :	**IDF**	(set of all) (variable) identifiers
I :	**INT**	(set of all) integer constant representations as digit sequences

and introducing additionally the (notational) conventions:

State	(set of all) program states (i.e., maps from (variable) identifiers to values)
Val	(set of all) expression values (i.e., Boolean and integer values)
σ_0	with $\sigma_0 \in$ **State** defined by: $\sigma_0 = \lambda\, v.\ 0$
Id_{State}	Identity function on the set of states defined by: $Id_{State} = \lambda\, \sigma.\ \sigma$

we can define a semantics for WHILE programs which are given in terms of their abstract syntax trees:

$$
\begin{array}{rcl}
\mathcal{P} : \mathbf{PROG} \to \mathbf{State} & & \\
\mathcal{P}[\![\, S \,]\!] & = & \mathcal{S}[\![\, S \,]\!](\sigma_0) \\
 & & \\
\mathcal{S} : \mathbf{STMT} \to (\mathbf{State} \to \mathbf{State}) & & \\
\mathcal{S}[\![\, S_1; S_2 \,]\!] & = & \mathcal{S}[\![\, S_2 \,]\!] \circ \mathcal{S}[\![\, S_1 \,]\!] \\
\mathcal{S}[\![\, V = E \,]\!](\sigma) & = & \sigma' \text{ with } \sigma' = \lambda\, x.\ \text{if } x \equiv V \text{ then } \mathcal{E}[\![\, E \,]\!](\sigma) \text{ else } \sigma(x) \\
\mathcal{S}[\![\, \text{if } E\ S_1\ S_2 \,]\!](\sigma) & = & \begin{cases} \mathcal{S}[\![\, S_1 \,]\!](\sigma) & \text{if } \mathcal{E}[\![\, E \,]\!](\sigma) \neq 0 \\ \mathcal{S}[\![\, S_2 \,]\!](\sigma) & \text{otherwise} \end{cases} \\
\mathcal{S}[\![\, \text{while } E\ S \,]\!](\sigma) & = & \begin{cases} (\mathcal{S}[\![\, \text{while } E\ S \,]\!] \circ \mathcal{S}[\![\, S \,]\!])(\sigma) & \text{if } \mathcal{E}[\![\, E \,]\!](\sigma) \neq 0 \\ \sigma & \text{otherwise} \end{cases} \\
\mathcal{S}[\![\, \text{skip} \,]\!] & = & Id_{State} \\
 & & \\
\mathcal{E} : \mathbf{EXPR} \to (\mathbf{State} \to \mathbf{Val}) & & \\
\mathcal{E}[\![\, E_1 == E_2 \,]\!](\sigma) & = & equal(\mathcal{E}[\![\, E_1 \,]\!](\sigma), \mathcal{E}[\![\, E_2 \,]\!](\sigma)) \\
\mathcal{E}[\![\, E_1 > E_2 \,]\!](\sigma) & = & greater(\mathcal{E}[\![\, E_1 \,]\!](\sigma), \mathcal{E}[\![\, E_2 \,]\!](\sigma)) \\
\mathcal{E}[\![\, E_1 + E_2 \,]\!](\sigma) & = & plus(\mathcal{E}[\![\, E_1 \,]\!](\sigma), \mathcal{E}[\![\, E_2 \,]\!](\sigma)) \\
\mathcal{E}[\![\, E_1 * E_2 \,]\!](\sigma) & = & times(\mathcal{E}[\![\, E_1 \,]\!](\sigma), \mathcal{E}[\![\, E_2 \,]\!](\sigma)) \\
\mathcal{E}[\![\, V \,]\!](\sigma) & = & \sigma(V) \\
\mathcal{E}[\![\, I \,]\!](\sigma) & = & NaturalInterpretation(I)
\end{array}
$$

where *equal* and *greater* denote the equality and greater test on integers, and *plus* and *times* the addition and multiplication operation on integers. Last but not least, $NaturalInterpretation$ interpretes a digit sequence (possibly proceeded by the sign symbol `-`) in the 'natural sense' as the integer it represents.

3.1 Implement a Haskell function `semantics`

```
type Var   = String
type State = (Var -> Int)

sigma0 :: State
sigma0 = \_ -> 0

semantics :: P -> State
```

which, applied to a well-formed abstract syntax tree of a WHILE program p, computes the semantics of p with respect to the initial program state σ_0 implemented by `sigma0`. If the abstract syntax tree is not well-formed, no specific behaviour of `semantics` is requested. Note that input programs may result from the (composition of the) parsers of exercises 1 and 2.

Important: *Do not use self-defined modules!* If you want to re-use functions (written for earlier assignments), copy these functions to the new submission file. An `import` declaration for self-defined modules will fail, since only the submission file `assignment`i`.hs` , where i, $1 \leq i \leq 8$ (*tentatively*), denotes the running number of the assignment, willl be copied for the (semi-automatic) evaluation. No other file in addition to `assignment`i`.hs` will be copied.