185.A05 Advanced Functional Programming SS 2020

Monday, 18 May 2020

Assignment 6

on Chapter 15 (and related chapters, especially Chapter 12)

Topics: Monadic Parsing (as an example of monadic programming), Combinator Parsing (as an example of higher-order functions programming)

Submission deadline: Monday, 25 May 2020, 12am

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

Important:

1. Carefully read and follow the instructions outlined in the complementary files provided with assignment 1. If you have any questions regarding these instructions, ask your questions in the TISS forum. Following these instructions is paramount to ensure a smooth processing of your submitted file with the test system.

2. Store all functions to be written for this assignment in a top-level file named

   `Assignment6.hs`

   of your group directory starting with the module declaration

   ```
   module Assignment6 where
   ```

   Comment your program meaningfully; use auxiliary functions and constants, where reasonable. The very same file name shall be used for the second submission of assignment 6.

3. *Do not use self-defined modules!* If you want to re-use functions (written for other assignments), copy them to `Assignment6.hs`. Import declarations for self-defined modules will fail: Only `Assignment6.hs` will be copied for the (semi-automatical) evaluation procedure, no other ones.

4. Your programs will be (semi-automatically) evaluated on `g0` using the there installed GHC interpreter of GHC version 8.65. If you use a different tool or a different version of GHC for program development, please, double-check well in time before the submission deadline that your programs behave on `g0` and the there installed GHC interpreter version as you expect.

Programming tasks:

We consider the imperative programming language Mini. The *concrete syntax* of Mini programs is given by the below context-free grammar, where non-terminals are

enclosed in acute brackets (spitze Klammern), and terminal symbols are denoted by (sequences of) uppercase letters:

$$
\begin{array}{rcl}
<program> & ::= & \texttt{PROGRAM} <program\_name><statement\_seq> \ \texttt{.} \\
<program\_name> & ::= & <upper\_char><chardig\_seq> \\
<upper\_char> & ::= & \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid ... \mid \texttt{Z} \\
<statement\_seq> & ::= & <statement> \mid <statement>\texttt{;}<statement\_seq> \\
<statement> & ::= & <assignment> \mid <if> \mid <while> \mid <repeat> \\
 & & \mid \texttt{BEGIN} <statement\_seq> \texttt{END} \\
<assignment> & ::= & <variable> \texttt{ = } <expr> \\
<if> & ::= & \texttt{IF} <pred\_expr> \texttt{THEN} <statement> \texttt{ELSE} <statement> \\
 & & \mid \texttt{IF} <pred\_expr> \texttt{THEN} <statement> \\
<while> & ::= & \texttt{WHILE} <pred\_expr> \texttt{DO} <statement> \\
<repeat> & ::= & \texttt{REPEAT} <statement> \texttt{UNTIL} <pred\_expr> \\
<expr> & ::= & <variable> \mid <integer> \mid <float> \mid <operator><expr><expr> \\
<operator> & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
<pred\_expr> & ::= & <relator><expr><expr> \\
<relator> & ::= & \texttt{==} \mid \texttt{/=} \mid \texttt{>=} \mid \texttt{<=} \\
<variable> & ::= & <char><chardig\_seq> \\
<char> & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid ... \mid \texttt{z} \\
<chardig\_seq> & ::= & \varepsilon \mid <char><chardig\_seq> \mid <digit><chardig\_seq> \\
<integer> & ::= & <digit><digit\_seq> \mid \texttt{-} <digit><digit\_seq> \\
<digit> & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid ... \mid \texttt{9} \\
<digit\_seq> & ::= & \varepsilon \mid <digit><digit\_seq> \\
<float> & ::= & <integer> \ \texttt{.} \ <digit><digit\_seq>
\end{array}
$$

Variables are contiguous non-empty sequences of the lowercase letters `a`, `b`, `c`,..., `z` and digits `0`, `1`,...,`9` starting with a character. Program names must start with an uppercase character optionally followed by lowercase characters and digits. Integers are contiguous non-empty sequences of digits `0`, `1`,..., `9` possibly with leading zeros, and optionally proceeded with the character `-` for negative integers. White space and line breaks might freely occur in MINI programs (except of course in reserved words, program names, variables, integers, floats, and relator symbols). Expressions and predicate expressions are in Polish (prefix) notation (i.e., operator precedes its operands).

Next, we introduce some Haskell types for programs, statements, expressions, and predicate expressions of MINI programs allowing a tree-like representation of MINI programs, called *abstract syntax trees*:

```
type Identifier   = String
type Program_Name = Identifier
type Variable     = Identifier
type PN           = Program_Name
```

```
data P  = P PN [S] deriving (Eq,Show)    -- P for program
data S  = Ass E E                        -- S for statement
          | If PE [S] [S]
          | While PE [S]
          | Repeat [S] PE deriving (Eq,Show)
data E  = I Integer                      -- E for expression
          | F Float
          | V Variable
          | Plu E E
          | Min E E
          | Mul E E
          | Div E E deriving (Eq,Show)
data PE = Equal E E           -- PE for predicate expression
          | NEqual E E
          | GEqual E E
          | LEqual deriving (Eq,Show)
```

Add instance declarations in case of need.

1. *Combinator parsing* (cf. Chapter 15.2). Implement a combinator parser `parser1`

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

   such that `topLevel1` transforms well-formed MINI programs into abstract syntax trees, when called with `parser1` and some input string. If the input string is not a well-formed MINI program, `topLevel1` shall terminate with calling `error "parse unsuccessful"` (cf. function `topLevel`, Example 2, Chapter 15.2.5).

2. *Monadic parsing* (cf. Chapter 15.3). Implement a monadic parser `parser2`

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

   such that `topLevel2` transforms well-formed MINI programs into abstract syntax trees, when called with `parser2` and some input string. If the input string is not a well-formed MINI program, `topLevel2` shall terminate with calling `error "parse unsuccessful"` (cf. function `topLevel`, Example 2, Chapter 15.2.5).

3. **Without submission:** Write a few (well-formed and not well-formed) MINI programs and test both parsers with them. Do you have a preference for the combinator parser or the monadic parser? If so, why?

Iucundi acti labores.
Getane Arbeiten sind angenehm.

Cicero (106 - 43 v.Chr.)
röm. Staatsmann und Schriftsteller