CSCE 314: Programming Languages

Homework Assignment 7

Objective

The task in this assignment is to extend the language ***E*** (for *expression*) from the previous assignment to the language ***W*** (for *while*). ***W*** makes a distinction between *expressions* and *statements*. In addition to the expressions supported in ***E***, we add variables, comparison operations for numbers, and the logical operators *and*, *or*, and *not* for booleans. The set of statements supported consists of the empty statement, assignment, a conditional, a while loop, and a block consisting of zero or more statements.

Submission

- Like before, this homework is will be graded as a 0, 1, or 2 for each bit. (There are three tasks listed below, which will be evaluated on a scale with 0 = inadequate or no attempt; 1 = solid effort showing a good attempt at all questions; 2 = faultless/exemplary submission. You may find it helpful to discuss your homework with the peer teachers, TAs, or instructors (say, at their office hours).
- Prepare a **single** Haskell file with a `.hs` extension.
- *Recommended*: Include as a comment at the top of your code: your name and UIN.
- The file should contain functions named precisely as described below. Those functions must have proper type signatures and should follow exactly the ones that have been explicitly given. In addition, you may include any number of additional helper functions you like; you may not include any extra modules, packages, or external libraries.
- Additional documentation in the form of comments throughout your code is strongly encouraged. If you have used outside resources, document these directly in your source file.
- You are encouraged to use the class slack to communicate about/solicit help/ask questions clarifying aspects of the assignment.
- Submission will be via Canvas, due: 29 November 2021.

Read the following sections carefully—they specify in more detail what you need to do, and give guidance and a starting point for your work.

Representing programs

We do not write a parser for ***W*** yet. ***W*** programs are represented using Haskell data types. We use three types, one for each of values, expressions, and statements.

The primitive types of ***W*** are integers and booleans, which gives rise to the following data type:

```
data WValue = VInt Int
            | VBool Bool
              deriving (Eq, Show)
```

The following data type can represent the different kinds of expressions of ***W***:

```
data WExp = Val WValue
```

```
          | Var String

          | Plus WExp WExp
          | Minus WExp WExp
          | Multiplies WExp WExp
          | Divides WExp WExp

          | Equals WExp WExp
          | NotEqual WExp WExp
          | Less WExp WExp
          | Greater WExp WExp
          | LessOrEqual WExp WExp
          | GreaterOrEqual WExp WExp

          | And WExp WExp
          | Or WExp WExp
          | Not WExp
```

All values are expressions. A variable reference is an expression. Two integers can be added, subtracted, multiplied and divided (integer division) to produce an integer. They can be compared for equality and inequality and with *less*, *less than* or *equal*, *greater*, and *greater than or equal* operators to produce a boolean value. Two booleans can be compared for equality and inequality. They can be composed with logical operators *and* and *or*. The unary operator *not* can be applied to a boolean.

A statement can be *empty* (a no-operation), a declaration of a variable with an initializer expression, a variable assignment, if-statement, while-statement, or a block (a list of statements considered as one):

```
data WStmt = Empty
           | VarDecl String WExp
           | Assign String WExp
           | If WExp WStmt WStmt
           | While WExp WStmt
           | Block [WStmt]
```

To give a sense of what ***W*** programs are like, here are two short ***W*** programs. We imagine some intuitive syntax for ***W*** and present the examples both in this imaginary syntax (to make the programs more readable) and using the `WStmt`, `WExp`, and `WValue` data types.

Example 1

This example shows the use of the conditional statement and logical operations. First with our conjured-up syntax:

```
var x = 0;
var b = x > 0;
if (b || !(x >= 0)) {
  x = 1;
} else {
  x = 2;
}
```

Then as a Haskell value constructed with the `WExp`, `WStmt`, and `WValue` data types:

```
p1 = Block
     [
       VarDecl "x" (Val (VInt 0)),
       VarDecl "b" (Greater (Var "x") (Val (VInt 0))),
       If (Or (Var "b") (Not (GreaterOrEqual (Var "x") (Val (VInt 0)))))
         ( Block [ Assign "x" (Val (VInt 1)) ] )
         ( Block [ Assign "x" (Val (VInt 2)) ] )
     ]
```

Example 2

This example program demonstrates ***W***'s loop construct. Note that the program refers to the variables `x` and `result` that are not declared. Undeclared variables are our mechanism for providing input to and output from a program: the program is expected to be launched with a memory that has these variables declared; they are left in the memory when the program exits.

```
var acc = 1;
while ( x > 0 ) {
    acc = acc * x;
    x = x - 1;
}
result = acc;
```

The same example as an AST (abstract syntax tree); we name the program to suggest its intent.

```
factorial =
  Block
  [
    VarDecl "acc" (Val (VInt 1)),
    While (Greater (Var "x") (Val (VInt 1)))
    (
      Block
      [
        Assign "acc" (Multiplies (Var "acc") (Var "x")),
        Assign "x" (Minus (Var "x") (Val (VInt 1)))
      ]
    ),
    Assign "result" (Var "acc")
  ]
```

Interpreter

Memory

The addition of variables makes the ***W*** language notably more complicated than the ***E*** language in the previous assignment—we need to somehow represent memory. A simple (and inefficient) representation will do for this assignment. We treat memory as a list of key-value pairs:

```
type Memory = [(String, WValue)]
```

To declare a variable means prepending a new key-value pair to the list. To assign to a variable could mean finding the key equal to the variable's name and modifying the value corresponding to that key. Haskell does not allow us to do this (it does, but we have to learn new tricks first to know how). Therefore, every assignment reconstructs the entire memory, with everything else the same except for the value of the assigned to variable. This is really inefficient; for simplicity, we ignore this efficiency concern.

Somehow the correct scoping of variables must be ensured. One way to do this is to add a marker of some kind to the memory whenever entering a new scope, and popping elements off the memory until the first marker whenever leaving a scope. (This scheme shares some resemblance with how activation records are handled in stack-based languages.) Let's use the value `("|", undefined)` as the marker.

Consider this program:

```
var a = 1;
{
  var b = 2;
  var c = 3;
  b = 4;
}
```

- At the beginning of the program, the memory should be `[]`.
- The entire program is a block; after entering that block, the memory should be `[("|", undefined)]`.
- After the declaration of `a`, the memory should be `[("a", VInt 1), ("|", undefined)]`.
- After the `{`, the memory should be `[("|", undefined), ("a", VInt 1), ("|", undefined)]`.
- After the declaration of `b`, the memory should be `[("b", VInt 2), ("|", undefined), ("a", VInt 1), ("|", undefined)]`.
- After the declaration of `c`, the memory should be `[("c", VInt 3), ("b", VInt 2), ("|", undefined), ("a", VInt 1), ("|", undefined)]`.
- After the assignment to `b`, the memory should be `[("c", VInt 3), ("b", VInt 4), ("|", undefined), ("a", VInt 1), ("|", undefined)]`.
- After the `}`, the memory should be `[("a", VInt 1), ("|", undefined)]`.
- After exiting the program (the main block), the memory should be `[]`.

Values, expressions, and statements

W has three syntactic categories: values, expressions, and statements. We observe the following:

> An expression evaluates to a value. Expression evaluation may need to read from the memory, but it does not modify it. Statements do not have a value. Executing a statement may modify the memory.

Hence, the types of the evaluator and executor functions are:

```
eval :: WExp -> Memory -> WValue
```

```
exec :: WStmt -> Memory -> Memory
```

A *program* in ***W*** is any value of type `WStmt`.

Running a program

To run a program means calling the `exec` function. Specifying input to a program means passing `exec` a value of type `Memory` with some predefined variables. Observing the result of a program means looking up from the memory the values of variables of interest.

Assume that `lookup :: String -> Memory -> Maybe WValue` function looks up the value of a variable from memory. Then, computing, for example, `10!` with the factorial program is achieved as:

```
result = lookup "result" ( exec factorial [("result", undefined), ("x", VInt 10)] )
```

The type of result is `Maybe WValue`, the value `Just (VInt 3628800)`. To access the integer `3628800` consider this program:

Your tasks

1. Write an interpreter for ***W***. Concretely this means that you will implement the functions `eval` and `exec`.

   Note that ***W*** allows nonsensical programs, such as `Plus (VBool True) (Vint 1)`. Make sure that for such programs the evaluator aborts with some indicative error message of what went wrong. For aborting, you can use the `error :: String -> a` function. It is defined in `Prelude`.

   Your interpreter should also abort if a variable is used before it is declared. ***W*** distinguishes between a variable declaration and an assignment. An assignment should fail if a variable has not been declared. Declaring a variable twice in the same block should fail.

2. Write some tests that test all language constructs of your interpreter, using many different input programs.
3. Implement a ***W*** program for computing the **n***th* Fibonacci number. Implement a Haskell function `fibonacci :: Int -> Int` that uses your ***W*** program to compute the **n***th* Fibonacci number.

The code below can serve as a starting point for your tasks:

Skeleton Code

```
-- Assignment 7, CSCE-314

module Main where

import Prelude hiding (lookup)

import Test.HUnit
import System.Exit

-- AST definition for W
data WValue = VInt Int
            | VBool Bool
              deriving (Eq, Show)

data WExp = Val WValue

          | Var String

          | Plus WExp WExp
          | Minus WExp WExp
          | Multiplies WExp WExp
          | Divides WExp WExp

          | Equals WExp WExp
          | NotEqual WExp WExp
          | Less WExp WExp
          | Greater WExp WExp
          | LessOrEqual WExp WExp
          | GreaterOrEqual WExp WExp

          | And WExp WExp
          | Or WExp WExp
          | Not WExp

data WStmt = Empty
           | VarDecl String WExp
           | Assign String WExp
           | If WExp WStmt WStmt
           | While WExp WStmt
           | Block [WStmt]

type Memory = [(String, WValue)]
marker = ("|", undefined)
isMarker (x, _) = x == "|"

-- eval function
eval :: WExp -> Memory -> WValue
eval = undefined

-- exec function
exec :: WStmt -> Memory -> Memory
exec = undefined

-- example programs
factorial =
  Block
  [
    VarDecl "acc" (Val (VInt 1)),
    While (Greater (Var "x") (Val (VInt 1)))
    (
      Block
      [
        Assign "acc" (Multiplies (Var "acc") (Var "x")),
```

```
        Assign "x" (Minus (Var "x") (Val (VInt 1)))
      ]
    ),
    Assign "result" (Var "acc")
  ]

p1 = Block
     [
       VarDecl "x" (Val (VInt 0)),
       VarDecl "b" (Greater (Var "x") (Val (VInt 0))),
       If (Or (Var "b") (Not (GreaterOrEqual (Var "x") (Val (VInt 0)))))
         ( Block [ Assign "x" (Val (VInt 1)) ] )
         ( Block [ Assign "x" (Val (VInt 2)) ] )
     ]

-- some useful helper functions
lookup s [] = Nothing
lookup s ((k,v):xs) | s == k = Just v
                    | otherwise = lookup s xs

asInt (VInt v) = v
asInt x = error $ "Expected a number, got " ++ show x

asBool (VBool v) = v
asBool x = error $ "Expected a boolean, got " ++ show x

fromJust (Just v) = v
fromJust Nothing = error "Expected a value in Maybe, but got Nothing"

-- unit tests
myTestList =

  TestList [
    test $ assertEqual "p1 test" [] (exec p1 []),

    let res = lookup "result" (exec factorial [("result", undefined), ("x", VInt 10)])
    in test $ assertBool "factorial of 10" (3628800 == asInt (fromJust res))
    ]

-- main: run the unit tests
main = do c <- runTestTT myTestList
          putStrLn $ show c
          let errs = errors c
              fails = failures c
          if (errs + fails /= 0) then exitFailure else return ()
```


Acknowledgements

This assignment was based the one produced by Dr. Jaakko Järvi.