CSCE 314 Programming Languages – Fall 2015

Hyunyoung Lee

Assignment 4

Assigned on Wednesday, September 30, 2015

Electronic submission on eCampus due at 9:00 a.m., Wednesday, 10/14/2015
Honor code signed coversheet due at the beginning of class on Wednesday, 10/14/2015
If you do not turn in a Honor code signed coversheet your work will not be graded.

"On my honor, as an Aggie, I have neither given nor received any unauthorized aid on any portion of the academic work included in this assignment."

Typed or printed name of student	Section (501 or 502)
Signature of student	UIN

Note 1: This homework set is *individual* homework, not a team-based effort. Discussion of the concept is encouraged, but actual write-up of the solutions must be done individually.

Note 2: Submit electronically exactly one file, namely, *yourLastName-yourFirstName*-a4.hs, and nothing else, on eCampus.tamu.edu.

Note 3: Make sure that the Haskell script (the .hs file) you submit compiles without any error. If your program does not compile, you will likely receive zero points for this assignment.

Note 4: Remember to put the head comment in your file, including your name and *acknowledgements of any help received* in doing this assignment. Again, remember the honor code.

1 The Problem

The task in this assignment is to extend the language E (for *expression*) from the previous assignment to the language W (for *while*). The language W makes a distinction between expressions and statements: (1) In addition to the expressions in E, W supports variables, comparison operations for numbers, and the logical operators *and*, *or*, and *not* for booleans. (2) The language W also supports the following statements: the empty statement, the assignment statement, the if conditional statement, the while loop, and the block statement that consists of zero or more statements.
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.

2 Representing Programs

The *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, defined as follows:

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

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

```
data WExp = Val WValue
          | Var String
```

All values are expressions.
A variable reference is an expression.

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

Two integers can be added, subtracted, multiplied and divided (integer division) to produce an integer.

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

They can be compared for equality and inequality and with *less than*, *less than or equal*, *greater than*, and *greater than or equal* operators to produce a boolean value. Two booleans can also be compared for equality and inequality.

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

They can be composed using logical operators *and* and *or*, or involving the unary operator *not*.

A statment 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):

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

To give a flavor of what *W* programs are like, here are two short *W* programs. We present the examples first in some pseudocode and then using the `WStmt`, `WExp`, and `WValue` data types.

Example program 1. The first example shows the use of the if statement and logical operations.

In pseudocode: As a Haskell value constructed with the `WStmt`, `WExp`, and `WValue` data types as an AST:

```
prog1 {             prog1 = Block
 var x = 0;          [ VarDecl "x" (Val (VInt 0)),
 var b = x > 0;        VarDecl "b" (Greater (Var "x") (Val (VInt 0))),
 if (b || !(x >= 0))   If (Or (Var "b")
                              (Not (GreaterOrEqual (Var "x") (Val (VInt 0)))))
 then {  x = 1; }         ( Block [ Assign "x" (Val (VInt 1)) ] )
 else {  x = 2; }         ( Block [ Assign "x" (Val (VInt 2)) ] )
}                    ]
```

Example program 2. This example demonstrates the loop construct of W. Note that the program refers to the variable `arg` and `result` that are not declared. These two variables are our mechanism for providing input to and output from a W program: we launch the program with a memory that has these variables declared, and leave them in the memory when the program exits.

In pseudocode: As an AST using the three data types for W:

```
factorial ( arg ) {   factorial = Block
 var acc = 1;          [ VarDecl "acc" (Val (VInt 1)),
 while ( arg > 0 )       While (Greater (Var "arg") (Val (VInt 1)))
 {                       ( Block
    acc = acc * arg;       [ Assign "acc" (Multiplies (Var "acc") (Var "arg")),
    arg = arg - 1;           Assign "arg" (Minus (Var "arg") (Val (VInt 1)))
                           ]
 }                       ),
 result = acc;           Assign "result" (Var "acc")
}                      ]
```

3 Interpreter

Memory. The addition of variables makes the W language notably more complicated than the E language in the previous assignment because we now need to represent memory. We will represent memory as a list of key-value pairs as below:[1]

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

A variable cannot be declared more than once within the same scope. Thus, declaring a variable involves the following steps: First, check whether the same variable has already been defined in the current scope; If so, raise an exception with an error message saying that the variable was already declared in the current scope. Otherwise, prepend a new key-value pair to the memory (list).

Assigning to a variable means finding the first key in the memory, that is equal to the variable's name and modifying the value associated with the key. Here we enforce the type of the new value to be consistent with that of the existing value for the key. For example,

```
> exec (Assign "x" (Val (VInt 5))) [("x",VInt 1)]
```

[1]This is a simple and inefficient representation of memory, but it will do for this assignment.

should modify the value of the key `"x"` in the list with the new value `VInt 5` and return the new memory:

```
[("x",VInt 5)]
```

but

```
> exec (Assign "x" (Val (VInt 5))) [("x",VBool False)]
```

should raise an exception saying something like,

```
*** Exception: Type error in assignment
```

What you learned so far in Haskell does not allow you to "modify" existing object (here list), thus for every assignment we will reconstruct the entire memory in a way that you prepend the new key-value pair (with the new value) to the rest of memory unchanged. Furthermore, we need to ensure the correct scoping of variables. To do so, we will add a marker to the memory whenever entering a new scope, and whenever leaving a scope, pop elements off the memory until the first encountered marker is popped off because this marker indicates the beginning of the current scope. This scoping scheme simulates how activation records are handled in stack based languages. We use the value `("|", undefined)` as the marker. Consider the following program and the explanation of the scoping scheme.

```
1: prog2
2: {
3:    var a = 1;
4:    {
5:       var a = 2;
6:       var b = 3;
7:       a = 4;
8:    }
9: }
```

1: At the beginning of the program, the memory is empty `[]`.
2: The entire program is a block. After entering the block, the memory should be `[("|", undefined)]`
3: After the declaration of `a`, the memory should be `[("a", VInt 1),("|", undefined)]`
4: After entering the inner block, the memory should be `[("|", undefined),("a", VInt 1),("|", undefined)]`
5: After the declaration of `a`, the memory should be `[("a", VInt 2),("|", undefined),("a", VInt 1),("|", undefined)]`
6: After the declaration of `b`, the memory should be `[("b", VInt 3),("a", VInt 2),("|", undefined),("a", VInt 1),("|", undefined)]`
7: After the assignment to `a`, the memory should be `[("b", VInt 3),("a", VInt 4),("|", undefined),("a", VInt 1),("|", undefined)]`
8: After leaving the inner block, the memory should be `[("a", VInt 1),("|", undefined)]`
9: After exiting the program, the memory should be empty `[]`.

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

1. An expression is *evaluated* to a value. Expression evaluation may need to read from the memory, but it does not modify it.
2. Statements are *executed* and do not results in 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.

4 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 the function `lookup :: String -> Memory -> Maybe WValue` 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", VInt (-1)),("arg", VInt 10)] )
```

The type of `result` is `Maybe WValue`, and the value is `Just (VInt 3628800)`. To access the integer `3628800` from `Just (VInt 3628800)` one has to do some unwrapping.

5 Tasks

1. (80 points) Write an interpreter for W. 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 function `error :: String -> a` defined in `Prelude`.

   Your interpreter should also abort if a variable is used before it is declared. W makes a difference 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. (30 points) Write a large number (at least 15) of tests that test all language constructs of your interpreter, using many different input programs.

3. (30 points) 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.

You will earn total 140 points.