CSCE 314 Programming Languages – Fall 2015
Hyunyoung Lee

Assignment 5

Assigned on Wednesday, October 14, 2015

Electronic submission on eCampus due at 9:00 a.m., Friday, 10/30/2015
Honor code signed coversheet due at the beginning of class on Friday, 10/30/2015
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 `.tar` or `.zip` file, namely, *yourLastName-yourFirstName*-a5.zip (or .tar), and nothing else, on eCampus.tamu.edu. What to submit is detailed below.

Note 3: Make sure that all Haskell programs that you submit must compile without errors. 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 write a parser for a simple language. The language we use is the language W from the previous assignment, extended with strings as a way of outputting values. We do not define the grammar for W formally; you can infer the grammar for W from the example programs given in Assignment 4, and apply usual (say, C-like) conventions.

Essentially, the pseudocode in those examples are now the source program written in W language in this assignment. Then, given the source program as input to your parser (that you will write in this assignment), your parser returns an AST for the source program, which will then be executed and evaluated using the interpreter that (is similar to what) you wrote in Assignment 4. The `main` function provided shows this flow of execution clearly.

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.

1.1 Representing Programs

In addition to integers and booleans as before, now *strings* are supported too. An explicit instance declaration defines a `show` that shows only the underlying value, not the AST node type. Note that a new constructor `VMarker` has been added to `WValue`, to be used as the value of a scope marker in lieu of `undefined`. The benefit of this is that the contents of the memory can be shown for debugging purposes with the compiler generated `show` function.

```
data WValue = VInt Int
            | VBool Bool
            | VString String
            | VMarker
              deriving Eq

instance Show WValue where
   show (VInt i)    = show i
   show (VBool b)   = show b
   show (VString s) = s
   show (VMarker)   = "_"
```

The expression type is unchanged:

```
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
            deriving Show
```

One additional *W* statement constructor is added: `Print`. When executing `Print e`, `e` is evaluated and the resulting value is printed to the standard output stream.

```
data WStmt = Empty
           | VarDecl String WExp
           | Assign  String WExp
           | If      WExp   WStmt   WStmt
```

```
          | While   WExp   WStmt
          | Block   [WStmt]
          | Print   WExp
            deriving Show
```

We continue to represent memory as a list of key-value pairs:

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

1.2 Interpreter

An interpreter for W will be provided[1]. It is very similar to the one you wrote in the previous assignment, but because of the newly added output capabilities, statements must now be executed within the `IO` monad. Thus the type of `exec` changes to:

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

The type of `eval` remains the same:

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

2 Tasks

You will earn total 140 points.

1. (80 points) Write a parser for W. The type signature of your parser should be as follows:

   ```
   wprogram :: Parser WStmt
   ```

   Given are three files: `Main.hs`, `W.hs`, and `WParser.hs`, each defining a module. The skeleton of the parser, as well as all the parsing tools we discussed in class, are in `WParser.hs`. Of these files, you likely only need to modify WParser.hs.

   As given, the parser accepts empty statements and print statements. It also understands C++ style single-line comments. The following program should work:

   ```
   // The skeleton parser accepts only print and empty statements
   ;;;;;
   print "Testing...\n";
   ;;;;;
   ```

2. (40 points) Write a test suite of W-programs that tests each feature as well as combinations of features of W. Use the suffix `.w` for W-program files. Write a script `testw` that parses and interprets each of the test programs in your test suite and compares their output to what is expected, and reports errors. Use bash, Make, Perl, Python, Haskell, whatever you like.

3. (20 points) Rewrite your *Fibonacci* program from the previous assignment in the syntax of W.

[1]It will become available at 1 p.m. on Sunday 10/18 when no more late submission of Assignment 4 (even with the late penalty) will be accepted.

2.1 What To Turn In

Create a directory `yourLastName-yourFirstName-a5`. Its contents should be the files `Main.hs`, `W.hs`, `WParser.hs`, your script `testw`, and all your `.w` files, including the `fibonacci.w` file.

Do not include `.hi`, `.o`, or any executable files. Then package the directory into *yourLastName-yourFirstName-a5.zip* or *yourLastName-yourFirstName-a5.tar*.

Submit only the `.zip` or `.tar` file, nothing else. We will deduct two points for each `.hi` and `.o` file in your tar or zip file, and two points for each violation of the file naming conventions.

3 Some Guidance

3.1 Parsing whitespace

The parser removes whitespace before the start of the program and then after every token. Then each token is assumed to begin with a non-whitespace character. This is taken care of by the `symbol`, `identifier`, and `stringLiteral` parsers, which all other parsers eventually rely on. You should nevertheless be careful to maintain this property.

3.2 Expression parsing

Recursive descent parsers, which is what we write with our parser combinators, cannot handle left recursive production rules. E.g., you cannot write an expression parser like this: `expr = expr +++ (expr >>= \e1 -> symbol "+" ...` A typical pattern for implementing binary operators (left-associative ones) is as follows:

```
expr = term >>= termSeq
termSeq left =
      ( do op <- (symbol "+" >> return Plus) +++ (symbol "-" >> return Minus)
           right <- term
           termSeq (op left right)
      ) +++ return left
term = factor >>= factorSeq
factorSeq left =
      ( do op <- (symbol "*" >> return Multiplies)
                 +++ (symbol "/" >> return Divides)
           right <- factor
           factorSeq (op left right)
      ) +++ return left
factor = (nat >>= \n -> return $ Val (VInt n))
        +++ stringLiteral +++ parens expr
```

The *W* language has more precedence levels, so there will be more "layers" in the expression parsers for *W*, but the structure can be as above.

3.3 Parsing and interpreting programs

Equipped with a parser, the interpreter can read the program to be executed from a file. The provided `main` function is already setup to do that.

```
main = do
  args <- getArgs -- get the command line arguments

  let (first:rest) = args
  let debug = first == "-d"
  let fileName = if debug then head rest else first

  str <- readFile fileName

  let prog = parse wprogram str

  let ast = case prog of
              [(p, [])]  -> p
              [(_, inp)] -> error ("Unused program text: " ++
                                  take 256 inp) -- this helps in debugging
              []         -> error "Syntax error"
              _          -> error "Ambiguous parses"
  result <- exec ast []

  when debug $ print "AST:"    >> print prog
  when debug $ print "RESULT:" >> print result
```

Once you compile this program (with the interpreter and your parser implementation) to an executable named `w`, you can execute a *W* program in file `prog.w` as:

```
> ./w prog.w
```

The `-d` option, as in `./w -d prog.w`, shows a bit of debug information (the AST that was parsed and the contents of the memory at the end of the program).

Compiling the skeleton: Now that the interpreter consists of several modules, we must instruct `ghc` to build the entire program. Compiling with the command `ghc --make main.hs -o w` will achieve this, creating an executable program named `w`.

3.4 Input and output

Each program now starts with an empty memory, so the only source of input to a *W* program is the program text itself. For output, *W* provides the `print` statement.

The following example program computes the factorial of 12:

```
var x = 12;

var acc = 1;
while (x > 1) {
  acc = acc * x;
  x = x - 1;
}
print "result is ";
print acc;
print "\n";
```

3.5 About syntax errors in W programs

The parser provides very limited diagnostics if a program is syntactically incorrect. The `main` function prints out the beginning of the remaining input string not yet parsed. That information should be enough to indicate which statement contains the error.

More realistic parser implementations carry information about the source position of tokens and try to give accurate information of what is wrong. Here we wish to keep the parser as simple as possible and thus forgo such conveniences.

A good set of tests can mitigate the pain caused by such lack of error messages. You should thus start writing your test suite as soon as you can parse the simplest possible program, and extend it as you make progress. By running your test suite regularly, you can detect changes that may have broken existing working functionality.