CSCE 314 [Sections 202, 502] Programming Languages – Fall 2016

Anandi Dutta

Assignment 4

Assigned on Friday, October 7, 2016

Electronic submission to eCampus due

1. Part I (Interpreter): At **23:59, Tuesday, October 25, 2016**
   You have 4 free days. Use them wisely.
2. Part II (Parser): At **23:59, Tuesday, October 25, 2016**

By electronically submitting this assignment to eCampus by logging in to your account, you are signing electronically on the following Aggie Honor Code:

"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."

In this assignment, you will practice programming language design and implementation using Haskell. In particular, given a grammar for a programming language, you will (i) implement an interpreter that evaluates expressions and executes statements of the language, and (ii) implement a parser that can check the syntactic structure of a given program written in the language.

Below, you will find problem descriptions with specific requirements. Read the problem descriptions and requirements carefully!! There may be significant penalties for not fulfilling the requirements. You will earn total 200 points.

Note 1: This homework set is recommended as a *two-person team work*. Working as a team does not mean that two people split the work, but does mean that two people *work together* – study the problem statements together, discuss possible solutions together, and produce the final code together, so that both of the team members have clear idea on why/how you wrote the code that you submit.
If you prefer to work alone, that is also acceptable, but the same grading rubrics will be applied whether you worked alone or as a team.

Note 2: For Part I, submit electronically exactly one `.tar` or `.zip` file with the naming convention: *teamMember1LastName-teamMember2LastName*-a41.zip (or .tar), and nothing else, to the link "Assignment 4 Part I" on eCampus.tamu.edu. Each team member submits exactly the same contents.
For Part II, submit electronically exactly one `.tar` or `.zip` file with the same naming convention as Part I but "-a42" such as, *teamMember1LastName-teamMember2LastName*-a42.zip (or .tar), and nothing else, to the link "Assignment 4 Part II" on eCampus.tamu.edu. Each team member submits exactly the same contents.
What to include in each zip (or tar) folder is detailed below.

Note 3: Please make sure that the Haskell script (the .hs file) you submit compiles without any error when compiled using the Glasgow Haskell Compiler (ghc), version 8.0.1 (the latest version) or 7.4.2 (that is installed in the departmental servers, linux.cse.tamu.edu and compute.cse.tamu.edu). Each zip (or tar) folder must contain a README file in that the team members' names and UINs are clearly written at the top, and which version of GHC you used to compile-&-run your code is clearly specified. This is very important! Your team will

get points deducted if these informations are not clear. To check the version of the GHC, type `ghc --version` at the command prompt.

Note 4: Remember to put the head comment in your files, including your names, UINs, and *acknowledgements of any help received* in doing this assignment (you do not need to acknowledge your team members). Your team will get points deducted if you do not put the head comment. Again, remember the honor code.

As part of last assignment, you implemented an evaluator `eval` for a tiny language E (for *expression*). In this assignment, we extend E to the language W (for *while*). 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) W also supports the following statements: the empty statement, the variable declaration statement, the assignment statement, the if conditional statement, the while loop, the block statement that consists of zero or more statements, and the print statement (to be only used in Part II).

1 Grammar for W

Grammar for the W language consists of the following rules.

1. ⟨statement⟩ ::= ⟨empty-statement⟩ | ⟨variable-delaration-statement⟩
   | ⟨assignment-statement⟩ | ⟨if-statement⟩ | ⟨while-statement⟩
   | ⟨print-statement⟩ | ⟨block-statement⟩
2. ⟨empty-statement⟩ ::= `;`
3. ⟨variable-delaration-statement⟩ ::= `var` ⟨identifier⟩ `=` ⟨expression⟩ `;`
4. ⟨assignment-statement⟩ ::= ⟨identifier⟩ `=` ⟨expression⟩ `;`
5. ⟨if-statement⟩ ::= `if (` ⟨expression⟩ `)` ⟨statement⟩ [`else` ⟨statement⟩]
6. ⟨while-statement⟩ ::= `while (` ⟨expression⟩ `)` ⟨statement⟩
7. ⟨print-statement⟩ ::= `print` ⟨expression⟩ `;`
8. ⟨block-statement⟩ ::= `{` {⟨variable-delaration-statement⟩} {⟨statement⟩}+ `}`
9. ⟨identifier⟩ ::= ⟨lower-case-letter⟩ {⟨alpha-numeric⟩}
10. ⟨expression⟩ ::= ⟨expression⟩ `||` ⟨and-expr⟩ | ⟨and-expr⟩
11. ⟨and-expr⟩ ::= ⟨and-expr⟩ `&&` ⟨unary-rel-expr⟩ | ⟨unary-rel-expr⟩
12. ⟨unary-rel-expr⟩ ::= `!` ⟨unary-rel-expr⟩ | ⟨rel-expr⟩
13. ⟨rel-expr⟩ ::= ⟨sum-expr⟩ ⟨rel-op⟩ ⟨sum-expr⟩ | ⟨sum-expr⟩
14. ⟨rel-op⟩ ::= `==` | `!=` | `<=` | `>=` | `<` | `>`
15. ⟨sum-expr⟩ ::= ⟨sum-expr⟩ `+` ⟨term⟩ | ⟨term⟩
16. ⟨term⟩ ::= ⟨term⟩ `*` ⟨factor⟩ | ⟨factor⟩
17. ⟨factor⟩ ::= ⟨integer-literal⟩ | ⟨string-literal⟩ | ⟨bool-literal⟩ | ⟨variable⟩ | `(`⟨expression⟩`)`
18. ⟨bool-literal⟩ ::= `true` | `false`

The keywords of W language are `var`, `if`, `else`, `while`, `print`, `true`, and `false`. The symbols used in W language are `;`, `=`, `(`, `)`, `{`, `}`, `!`, `&&`, `||`, `==`, `!=`, `<=`, `>=`, `<`, `>`, `+` and `*`. A variable is an identifier, which is not a keyword. A string literal is anything enclosed in double quotations, for example, `"a string literal"`.

2 Representing Programs

The W programs are represented using Haskell data types. We use three types – `WValue` for values, `WExp` for expressions, and `WStmt` for statements. The primitive types of W are integers and booleans, defined as:

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

`VMarker` is not part of W language, but will be used to simulate memory, see below. The following data type represents the different kinds of *expressions* of W:

```
data WExp = Val WValue                  All values are expressions.
          | Var String                  A variable reference is an expression.

          | Plus      WExp  WExp        Two integers can be added or multiplied to
          | Mult      WExp  WExp        produce an integer.

          | Equals    WExp  WExp        Two integers can be compared for equality and
          | NotEqual  WExp  WExp        inequality and with less than, less than or equal,
          | Less      WExp  WExp        greater than and greater than or equal operators
          | LessOrEq  WExp  WExp        to produce a boolean value.
          | Greater     WExp  WExp      Two booleans can also be compared for equality
          | GreaterOrEq WExp  WExp      and inequality.

          | And       WExp  WExp        Logical expressions can be composed using logi-
          | Or        WExp  WExp        cal operators and and or, or involving the unary
          | Not       WExp              operator not.
          deriving Show
```

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

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

To give a flavor of what W programs are like, here are two short W programs. We present the examples first in high-level source code and then using the `WStmt`, `WExp`, and `WValue` data types.

Example program 1 demonstrates the use of the declaration statement, if-else statement, assignment statement, the plus operator, relational (comparison) operators, and logical operators.

In source code:

```
prog1 {
 var x = 0;
 var y = 1;
 var b = x > 0;
 if (b || !(x >= 0))
 {
    x = 1;
    y = y + 1;
 }
 else  x = 2;
}
```

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

```
prog1 = Block
 [ VarDecl "x" (Val (VInt 0)),
   VarDecl "y" (Val (VInt 1)),
   VarDecl "b" (Greater (Var "x") (Val (VInt 0))),
   If (Or (Var "b")
          (Not (GreaterOrEq (Var "x") (Val (VInt 0)))))
      ( Block
        [ Assign "x" (Val (VInt 1)),
          Assign "y" (Plus (Var "y") (Val (VInt 1)))
        ]
      )
      ( 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 source code:

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

As an AST using the three data types for W:

```
factorial = Block
 [ VarDecl "acc" (Val (VInt 1)),
   While (Greater (Var "arg") (Val (VInt 0)))
   ( Block
     [ Assign "acc" (Mult (Var "acc") (Var "arg")),
       Assign "arg" (Plus (Var "arg") (Val (VInt (-1))))
     ]
   ),
   Assign "result" (Var "acc")
 ]
```

3 Part I: 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

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

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)]
```

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 `("|", VMarker)` 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
   `[("|", VMarker)]`
3: After the declaration of `a`, the memory should be
   `[("a", VInt 1),("|", VMarker)]`
4: After entering the inner block, the memory should be
   `[("|", VMarker),("a", VInt 1),("|", VMarker)]`

5: After the declaration of `a`, the memory should be
   `[("a", VInt 2),("|", VMarker),("a", VInt 1),("|", VMarker)]`
6: After the declaration of `b`, the memory should be
   `[("b", VInt 3),("a", VInt 2),("|", VMarker),("a", VInt 1),("|", VMarker)]`
7: After the assignment to `a`, the memory should be
   `[("b", VInt 3),("a", VInt 4),("|", VMarker),("a", VInt 1),("|", VMarker)]`
8: After leaving the inner block, the memory should be
   `[("a", VInt 1),("|", VMarker)]`
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.

3.1 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.

3.2 Part I Tasks

You will earn 100 points for Part I.

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 also fail.

2. (20 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.

4 Part II: Parser for W

The task in Part II is to write a parser for W, extended with strings as a way of outputting values. Given the source program (ref. example programs given in Section 2) as input to your parser (that you will write here), 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 Part I. The `main` function provided shows this flow of execution. Study it carefully. 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.

4.1 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. 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 `WExp` is unchanged from Part I:

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

          | Equals  WExp WExp  | NotEqual    WExp WExp
          | Less    WExp WExp  | LessOrEq    WExp WExp
          | Greater WExp WExp  | GreaterOrEq 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)]
```

4.2 Interpreter

An interpreter for W will be provided[2]. It is very similar to the one you wrote in Part I, 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
```

4.3 Part II Tasks

You will earn total 100 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]It will become available at 1 a.m. on Friday 10/14 when no more late submission of Part I (even with the late penalty) will be accepted.

2. (10 points) Write a test suite of at least ten 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 your 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. (10 points) Rewrite your *Fibonacci* program from the previous assignment in the syntax of W.

4.4 What To Turn In

Create a directory `teamMember1LastName-teamMember2LastName-a42`. 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 *teamMember1LastName-teamMember2LastName-a42.zip* or *teamMember1LastName-teamMember2LastName-a42.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.

4.5 Some Guidance

4.5.1 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.

4.5.2 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)
           right <- term
           termSeq (op left right)
      ) +++ return left
term = factor >>= factorSeq
factorSeq left =
      ( do op <- (symbol "*" >> return Mult)
           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.

4.6 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`.

4.7 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";
```

4.8 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.