TEXAS A&M UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

# CSCE 314: Programming Languages

## Homework Assignment 8

### Objective

The task in this assignment is to write a parser for the simple language ***W*** from the previous assignment, extended with strings as a mechanism for outputting values. We do not define ***W***'s grammar formally; infer it from the example programs given in Homework Assignment 7, and apply usual conventions.

### Submission

- Like before, this homework is will be graded as a 0, 1, or 2 for each aspect, with 0 = inadequate or no attempt; 1 = solid effort showing a good attempt at all questions; 2 = faultless/exemplary submission. There are three tasks listed below; the second one is difficult for us to assess, so we will give points for the following three aspects: (1) implementation of the parser so that parses ***W*** correct programs; (2) helpful error messages when parsing ***W*** programs that have (syntax) errors; (3) The fibonacci program, and the ability to interpret it using your code. You may find it helpful to discuss your homework with the peer teachers, TAs, or instructors (say, at their office hours).
- *Recommended*: Include as a comment at the top of your code: your name and UIN.
- 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(s).
- 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: 8 December 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

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 be used as the value of a scope marker in lieu of `undefined`. The benefit is that the contents of the memory can now 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
```

One additional 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)]
```

### Interpreter

An interpreter for ***W*** is provided. 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. The type of exec changes:

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

The type of `eval` remains the same:

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

The implementation is given below. There should not be a need to modify it, but please study it nevertheless and compare to Assignment 7.

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

marker = ("|", VMarker)
isMarker (x, _) = x == "|"
```

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

eval (Val v) _ = v

eval (Var s) m =
  case lookup s m of
    Nothing -> error $ "Unknown variable " ++ s ++ " in memory " ++ show m
    Just v -> v

eval (Plus e1 e2) m =
  let e1' = eval e1 m
      e2' = eval e2 m
  in
   VInt $ asInt e1' + asInt e2'

eval (Minus e1 e2) m =
  let e1' = eval e1 m
      e2' = eval e2 m
  in
   VInt $ asInt e1' - asInt e2'

eval (Multiplies e1 e2) m =
  let e1' = eval e1 m
      e2' = eval e2 m
  in
   VInt $ asInt e1' * asInt e2'

eval (Divides e1 e2) m =
  let e1' = eval e1 m
      e2' = eval e2 m
  in
   VInt $ asInt e1' `div` asInt e2'

eval (Equals e1 e2) m =
  let e1' = eval e1 m
      e2' = eval e2 m
  in
   VBool $ e1' == e2'

eval (NotEqual e1 e2) m = VBool $ not $ asBool $ eval (Equals e1 e2) m

eval (Less e1 e2) m =
  let e1' = eval e1 m
      e2' = eval e2 m
  in
   VBool $ asInt e1' < asInt e2'

eval (LessOrEqual e1 e2) m =
  let e1' = eval e1 m
      e2' = eval e2 m
  in
   VBool $ asInt e1' <= asInt e2'

eval (Greater e1 e2) m =
  let e1' = eval e1 m
      e2' = eval e2 m
  in
   VBool $ asInt e1' > asInt e2'

eval (GreaterOrEqual e1 e2) m =
  let e1' = eval e1 m
      e2' = eval e2 m
  in
   VBool $ asInt e1' >= asInt e2'

eval (And e1 e2) m | not (asBool (eval e1 m)) = VBool False
                   | otherwise = VBool (asBool (eval e2 m))

eval (Or e1 e2) m | asBool (eval e1 m) = VBool True
                  | otherwise = VBool (asBool (eval e2 m))

eval (Not e) m = VBool $ not $ asBool $ eval e m

----------
```

```
-- exec --
----------
exec :: WStmt -> Memory -> IO Memory
exec Empty m = return m

exec (VarDecl s e) m | not (definedInThisScope m) = return $ (s, eval e m) : m
                     | otherwise = error $ "Variable " ++ s ++ " already defined in this scope"
    where
      definedInThisScope (hd@(d, _):ds) | isMarker hd = False
                                        | d == s = True
                                        | otherwise = definedInThisScope ds

exec (Assign s e) m = return $ replaceFirstDef (eval e m) m
    where replaceFirstDef _ [] = error $ "Undefined variable " ++ s ++ " in assignment"
          replaceFirstDef v (hd@(n, _):m) | n == s = (n, v):m
                                          | otherwise = hd:replaceFirstDef v m

exec (If e s1 s2) m | eval e m == VBool True = exec s1 m
                    | eval e m == VBool False = exec s2 m
                    | otherwise = error "Non-boolean in condition of if"

exec (While e s) m | eval e m == VBool True = exec s m >>= \m' ->
                                              exec (While e s) m'
                   | eval e m == VBool False = return m
                   | otherwise = error "Non-boolean in condition of while"

exec (Block ss) m = bexec ss (marker:m) >>= \m' -> return (popMarker m')
    where bexec [] m = return m
          bexec (s:ss) m = exec s m >>= \m' ->
                           bexec ss m'
          popMarker [] = []
          popMarker (x:xs) | isMarker x = xs
                           | otherwise = popMarker xs

exec (Print e) m = putStr (show (eval e m)) >> return m
```

## Your tasks

1. 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`, `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. Write a test suite of ***W***-programs that tests each feature of ***W***, and combinations of features as well. 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. Rewrite your *Fibonacci* program from the previous assignment in ***W***'s syntax.

## Some guidance

### Parsing whitespace

The parser removes whitespace before the start of the program and then after every token. Each token parser can then assume that its input starts with a non-whitespace character. This is taken care of automatically by the `symbol`, `identifier`, and `stringLiteral` parsers, on which all other parsers eventually rely on. You should nevertheless be careful to maintain this property.

## 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 >>= \e1 -> symbol "+"` .... A typical pattern for implementing binary operators (left-associative ones) is as follows:

```
expr = term >>= termSeq

termSeq left = ( (symbol "+" +++ symbol "-") >>= \s ->
                 term >>= \right ->
                 termSeq ((toOp s) left right)
               ) +++ return left

term = factor >>= factorSeq

factorSeq left = ( (symbol "*" +++ symbol "/") >>= \s ->
                   factor >>= \right ->
                   factorSeq ((toOp s) left right)
                 ) +++ return left

factor = ( nat >>= \i ->
           return (Val (VInt i))
         ) +++ parens expr

toOp "+" = Plus
toOp "-" = Minus
toOp "*" = Multiplies
toOp "/" = Divides
```

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

## 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 (a:as) = args
  let debug = a == "-d"
  let fileName = if debug then head as else a

  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`.

## Input and output

Each program 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 20:

```
var x = 20;

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

## 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. The lack of proper diagnostics is a bit painful, and of course (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 mitigate the lack of good compiler error messages. You should 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.

## Acknowledgements

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