Homework 9: An interpreter for IMP

CS3490: Programming Languages

Due: Friday, November 3rd, 6:00PM

In this homework, you will implement an interpreter for a rudimentary *imperative* programming language, **IMP**.

The purpose of this homework is to integrate different language processing components into a single whole. Different sections of this homework may require you to rewatch different parts of lectures given in the last few weeks. You will need to review the concepts of environment, evaluation, lexical analysis, parsing, and I/O.

IMP Overview

The language **IMP** contains two datatypes: Integers and Booleans.

Integers can be combined into *arithmetic expressions*.

Booleans can be combined into *boolean expressions*.

All variables are of the *integer* type; the boolean type is only used in conditionals and control statements.

In addition to these two kinds of expressions, **IMP** programs consist of a sequence of *instructions*. These instructions can be executed, which has a practical effect of changing the values of some variables.

Executing a program will result in some final assignment of values to variables.

Syntax

The syntax of the language is based on the following datatypes.

```
-- Variables
type Vars = String
-- Arithmetic expressions
data AExpr = Var Vars | Const Integer           -- variables and constants
           | Add AExpr AExpr | Sub AExpr AExpr  -- addition and subtraction
           | Mul AExpr AExpr | Div AExpr AExpr  -- multiplication and division
           | Exp AExpr AExpr | Mod AExpr AExpr  -- exponential and remainder
  deriving Show
```

```
-- Boolean expressions
data BExpr = TT | FF | Not BExpr               -- the true and false constants
           | And BExpr BExpr | Or BExpr BExpr  -- and the boolean operations
           | Eq  AExpr AExpr     -- equality of arithmetic expressions
           | Lt  AExpr AExpr     -- true if the first is less than the second
           | Lte AExpr AExpr     -- true if it's less than or equal to
  deriving Show
-- Instructions
data Instr = Assign Vars AExpr            -- assign X to the value of an expression
           | IfThenElse BExpr Instr Instr -- conditional
           | While BExpr Instr            -- looping construct
           | Do [Instr]                   -- a block of several instructions
           | Nop                          -- the "do nothing" instruction
  deriving Show
```

Environments

You will use a lookup table to implement the environment.

```
type Env = [(Vars,Integer)]
```

Make sure you implement the following helper function before proceeding.

```
-- update (x,v) e sets the value of x to v and keeps other variables in e the same
update :: (Vars, Integer) -> Env -> Env
```

Question 1. Evaluating arithmetic and boolean expressions

Implement the following functions, by interpreting each arithmetic and logical operator by its usual meaning.

```
evala :: Env -> AExpr -> Integer
evalb :: Env -> BExpr -> Bool
```

Question 2. Executing instructions

Write the function `exec` of the following type:

```
exec :: Instr -> Env -> Env
```

You can think of the argument `e :: Env` as a snapshot of the state of computer memory, which assigns values to some of the variables. Then, running a single instruction may update this memory.

Think about what effect every constructor of type `Instr` should have, and then give a recursive implementation of `exec` using pattern-matching on this datatype.

Here are some examples of running this function:

```
exec (Assign "X" (Const 3)) [] = [("X",3)]
exec (Assign "X" (Const 4)) [("X",3)] = [("X",4)]
exec (Assign "X" (Const 4)) [("Y",3)] = [("X",4),("Y",3)]
exec (Assign "Z" (Const 4)) [("X",2),("Y",3)] = [("X",2),("Y",3),("Z",4)]
exec (IfThenElse (Lt (Var "X") (Const 10))
                 (Assign "X" (Add (Var "X") (Const 1)))
                 Nop)
     [("X",9)] = [("X",10)]
exec (IfThenElse (Lt (Var "X") (Const 10))
                 (Assign "X" (Add (Var "X") (Const 1)))
                 Nop)
     [("X",12)] = [("X",12)]
exec (While (Not (Eq  (Var "X") (Const 0)))
            (Assign "X" (Sub (Var "X") (Const 1))))
     [("X",5)] = [("X",0)]
exec (While (Not (Eq  (Var "X") (Const 0)))
            (Do [ (Assign "Y" (Mul (Var "X") (Var "Y")))
                , (Assign "X" (Sub (Var "X") (Const 1)))]))
     [("X",5),("Y",1)] = [("X",0),("Y",120)]
```

Notice that the last example computes the factorial of 5.

Example

Programs are executed starting from the empty environment. After all the statements are executed in sequence, the result is found in the final environment returned by the last instruction.

```
run :: [Instr] -> Env
run p = exec (Do p) []

sum100 :: [Instr]     -- a program to add together all the numbers up to 100
sum100 = [
  Assign "X" (Const 0),         -- initialize the sum     at X=0
  Assign "C" (Const 1),         -- initialize the counter at C=1
  While (Lt (Var "C") (Const 101))    -- while C < 101, do:
        (Do [Assign "X" (Add (Var "X") (Var "C")),  -- X := X + C;
             Assign "C" (Add (Var "C") (Const 1))]  -- C := C + 1
        )]

sum100output = lookUp "X" (run sum100)
```

Question 3. Lexical analysis

Use the following datatype of tokens:

```
data UOps = NotOp deriving Show
data BOps = AddOp | SubOp | MulOp | DivOp | ModOp | ExpOp
          | AndOp | OrOp  | EqOp  | LtOp  | LteOp
  deriving Show
data Keywords = ForK | FromK | ToK | IfK | ThenK
  deriving Show
data Token = VSym String | CSym Integer | BSym Bool
           | UOp UOps | BOp BOps | AssignOp
           | LPar | RPar | LBra | RBra | Semi
           | Keyword Keywords
           | Err String
           | PA AExpr | PB BExpr | PI Instr | Block [Instr]
  deriving Show
```

Note that the final three constructors in the above type are auxiliary; they will be used during the parsing stage to admit the three datatypes into the list of tokens on the parse stack.

Implement the function

```
lexer :: String -> [Token]
```

which, given a string, produces a list of appropriate tokens.

You are free to use helper functions to identify alphanumeric characters, add spaces, etc. — just as we did in class.

Lexical rules

Variables should begin with a *lower-case letter*, and then can be followed by zero or more letters or numbers.

Constants An integer constant is a number. A boolean constant is *upper-case* T or F.

Punctuation marks include left and right parentheses and braces, and the semicolon.

Keywords These should be tagged with one of the following strings:

```
while if then else nop
```

Operators Use the following table:

Warning

Again, remember that the backslash character `'\'` has to be escaped when used inside a Haskell string or a character — but not when used inside the input file, or entered into `getLine`.

Add	`+`	And	`/\`
Sub	`-`	Or	`\/`
Mul	`*`	Not	`!`
Div	`/`	Eq	`==`
Exp	`^`	Lt	`<`
Mod	`%`	Lte	`<=`
Assign	`:=`		

Figure 1: Representation of operators on the lexical level

Question 4. Parsing

Write a function `parser :: [Token] -> Instr` which takes a list of tokens and returns an internal representation of an **IMP** program. (Hint: If needed, you can turn a list of instructions into a single instruction using Do.)

The most tricky part is parsing blocks of statements.

Here are the syntactic rules:

- A block of instructions begins with a left brace and ends with a right brace.
- The instructions must be separated from each other with a semicolon.
- The semicolon following the last instruction in a block is *mandatory*.
- All the parsed instructions should be put in a list given to `Do` constructor.

You are allowed to introduce additional tokens for parsing lists of instructions.

Question 5. I/O

Write a *main* method which asks for a file to load, parses and loads the file into the `[Instr]` datatype, executes this program, and then outputs the final list of variable assignments. Two input files have been provided for testing.