Homework 8: An interpreter for IMP

CS3490: Programming Languages

1. Overview

In this assignment, you will implement an interpreter for a simple programming language called IMP. It is an *imperative* language that is designed as a minimalist version of C. Thus, IMP uses syntax similar to C, and is based on the same basic primitives.

This homework synthesizes various concepts we covered over the last several weeks, including:

- Encoding syntax via expression trees
- Searching and updating environments
- Evaluating arithmetical and logical expressions
- Parsing and lexical analysis
- Terminal I/O and read-eval-print-loop

You are allowed to reuse code from the lectures or prior assignments. You may also work in groups or with a partner — as long as all the code you submit is your own. (That is, you can share ideas or discuss code segments, but you should each write your own programs and submit them separately.) Make sure you understand what every function does, how it does it, and can write it yourself if asked during the next test or quiz.

This assignment is longer than usual and will count both as a homework and as a 50-point quiz. It also covers important concepts that you should understand well going into the final weeks of the course.

Therefore, it is important to budget enough time for this assignment, start early, and make sure to master all the material needed to successfully complete it.

Don't wait to ask questions if you get stuck!

Use asulearn's homework discussion forum when possible.

2. IMP: Language Definition

An IMP program consists of a list of *instructions*.

In turn, instructions consist of keywords and boolean or arithmetic expressions.

Running a program amounts to executing the list of instructions, one at a time.

Executing an instruction transforms the current environment into a new one, with some variables potentially being updated in the process.

2.1. Types

The language IMP contains two types of data: 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.

Therefore, there is no need for type declarations — the type of any expression can be read off from the topmost operator.

2.2. Syntax

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

```
type Vars  = String                          -- Variables
type Value = Integer                         -- Values

data AExpr = Var Vars | Num Value            -- Arithmetic expressions
           | Add AExpr AExpr | Sub AExpr AExpr | Mul AExpr AExpr
           | Div AExpr AExpr | Mod AExpr AExpr | Exp AExpr AExpr
  deriving (Eq,Show)

data BExpr = Const Bool                      -- Boolean expressions
           | And BExpr BExpr | Or BExpr BExpr | Not BExpr
           | Eq  AExpr AExpr | Lt AExpr AExpr | Lte AExpr AExpr
           | Neq AExpr AExpr | Gt AExpr AExpr | Gte AExpr AExpr
  deriving (Eq,Show)

data Instr = Assign Vars AExpr            -- assignment
           | IfThen BExpr Instr           -- conditional
           | IfThenElse BExpr Instr Instr -- another conditional
           | While BExpr Instr            -- looping construct
           | Do [Instr]                   -- a block of several instructions
           | Nop                          -- the "do nothing" instruction
           | Return AExpr                 -- the final value to return
  deriving (Eq,Show)
```

3. Lexical analysis

Token specification

The tokens for lexical analysis are based on the following datatypes.

```
data Keywords = IfK | ThenK | ElseK | WhileK | NopK | ReturnK
  deriving (Eq,Show)
data BOps = AddOp | SubOp | MulOp | DivOp | ModOp | ExpOp
          | AndOp | OrOp  | EqOp  | NeqOp
          | LtOp  | LteOp | GtOp  | GteOp
  deriving (Eq,Show)
data Token = VSym String | CSym Integer | BSym Bool
           | LPar | RPar | LBra | RBra | Semi
           | BOp BOps | NotOp | AssignOp
           | Keyword Keywords
           | Err String
           | PA AExpr | PB BExpr | PI Instr | Block [Instr]
  deriving (Eq,Show)
```

Note that the last four constructors in the `Token` type are auxiliary, and are only used internally by the parser.

The tokens `PA`, `PB`, and `PI` are used to put on the parse stack arithmetic expressions, boolean expressions, and instructions corresponding to fragments of the code that have been parsed successfully.

The `Block` token is used for chaining together instructions appearing inside blocks.

Lexical rules

As the first stage of the parsing process, each valid piece of the input code should be classified into an appropriate token.

The rules specifying what every token should look like on the input level are as follows.

Variables: Each variable should begin with a *lower-case letter*. It can then be followed by zero or more *letters or numbers*.

Constants: An integer constant is a number. A boolean constant is either `tt` or `ff`.

Punctuation marks: These include parentheses, braces, and the semicolon.

Keywords: The keywords appear as one of these *lowercase* strings:

```
while if then else nop return
```

The lexer should match each of the above strings to the corresponding constructor of the `Keywords` datatype:

```
WhileK IfK ThenK ElseK NopK ReturnK
```

Operators: The concrete syntax of operators is given in Figure 1.

	Arithmetical		Logical	
Binary Operators	Addition	`+`	Conjunction	`&&`
	Subtraction	`-`	Disjunction	`||`
	Multiplication	`*`	Equality	`==`
	Division	`/`	Disequality	`!=`
	Modulus	`%`	Less-than	`<`
	Exponentiation	`^`	Less-or-equal	`<=`
			Greater-than	`>`
			Greater-or-equal	`>=`
Other operators	Assignment	`:=`	Negation	`!`

Figure 1: Representation of operators on the lexical level

Question 1 (20 pts)

Implement the function `lexer :: String -> [Token]`

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

which produces a list of tokens from the given string.

You should ignore whitespace, and whenever any unrecognized symbol is encountered, use the `Err` token to return it and the substring of length 10 within which it occurs.

4. Parsing

The next step is to implement the parser.

You will use the same bottom-up strategy we employed in class, where you first write the function `sr :: [Token] -> [Token] -> [Token]` and then use that function to build a parse tree for the whole IMP program.

Parsing rules

The arithmetic and boolean expressions should be parsed in the same manner as in the lectures and previous homeworks.

The arithmetic operators should follow the PEMDAS convention. You can give the modulus operator the same precedence level as division.

For parsing instructions, the following rules should be adhered to.

- An assignment instruction looks like $v := e$, where v is a variable symbol, and e is any arithmetic expression. For example, `v := v + x%2` would give rise to

```
ghci> lexer "v := v + x%2;"
[VSym "v",AssignOp,VSym "v",BOp AddOp,VSym "x",BOp ModOp,CSym 2,Semi]
ghci> sr [] it
[PI (Assign "v" (Add (Var v) (Mod (Var "x") (Num 2))))]
```

- Note that, by the time you have all the components of an assignment on the stack, the variable symbol is likely to have already been promoted to a `PA` via `Var`. So you will either need to match *that*, or block the variable from being promoted in the first place, if there's an assignment operator coming after it in the input queue.
- A block of instructions begins with a left brace and ends with a right brace.
- Most instructions should end with a semicolon. However, care should be taken when parsing conditionals and blocks, because they take as input instructions that should already have the semicolon.

  Suppose you encounter a condition of the form `"if c then i;"`. Once you parse `"i;"` as a single token `PI i`, the semicolon that came after the original `i` is no longer present on the stack. Hence, the parse rule for `if-then` should not require the semicolon.

  Similarly, since all instructions inside a do block should end with a semicolon, there is no semicolon expected after the closing brace:

  ```
  { i1 ; i2 ; ... iN ; }
  ```
- If you have successfully parsed an `if-then` instruction, but it is followed by the `else` keyword and another instruction (with a semicolon), you should combine them both into a single `if-then-else` instruction.

Question 2 (20 pts)

Write a function `sr :: [Token] -> [Token] -> [Token]` which applies the grammar rules for expressions and instructions "in reverse", in an attempt to build a parse tree from a given list of tokens.

Notice that, if the input consists of a sequence of instructions

```
i1; i2; ... iN;
```

then the final stack after running `sr` should look something like

```
[Semi,PI iN,Semi,...,Semi,PI i1]
```

We now wish to transform this list into `[i1,i2,..,iN] :: [Instr]`.

To that end, write the function `readProg :: [Token] -> Either [Instr] String` which takes a list of tokens and returns either an internal representation of an IMP program, or an error string.

(*Hint.* You can do this directly via list recursion. Or you can let `sr` do this for you as well, by putting the list into a context that makes it look like part of a `do`-block.)

5. Evaluating expressions

As usual, an environment is a lookup table of variable–value pairs.

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

First, make sure you have implemented the following helper function.

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

Question 3 (20 pts)

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

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

6. Executing instructions

Question 4 (20 pts)

Implement two mutually recursive functions

```
exec     ::  Instr  -> Env -> Env
execList :: [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" (Num 3)) [] = [("X",3)]
exec (Assign "X" (Num 4)) [("X",3)] = [("X",4)]
exec (Assign "X" (Num 4)) [("Y",3)] = [("Y",3),("X",4)]
exec (Assign "Z" (Num 4)) [("X",2),("Y",3)] = [("X",2),("Y",3),("Z",4)]
exec (IfThenElse (Lt (Var "X") (Num 10))
                 (Assign "X" (Add (Var "X") (Num 1)))
                 Nop)
     [("X",9)] = [("X",10)]
exec (IfThenElse (Lt (Var "X") (Num 10))
                 (Assign "X" (Add (Var "X") (Num 1)))
                 Nop)
     [("X",12)] = [("X",12)]
exec (While (Not (Eql (Var "X") (Num 0)))
            (Assign "X" (Sub (Var "X") (Num 1))))
     [("X",5)] = [("X",0)]
exec (While (Not (Eql (Var "X") (Num 0)))
            (Do [ (Assign "Y" (Mul (Var "X") (Var "Y")))
                , (Assign "X" (Sub (Var "X") (Num 1)))]))
     [("X",5),("Y",1)] = [("X",0),("Y",120)]
```

(Notice that the last example computes the factorial of 5.)

Programs are lists of instructions and are executed starting from the empty environment. The `return` keyword will store the final value to be returned into the special variable `""`, which cannot be entered by the user. After all the statements are executed in sequence, the result is found by looking up the value of this variable in the final environment returned by the last instruction.

```
run :: [Instr] -> Integer
run p = case lookup "" (execList p []) of
  Just x  -> x
  Nothing -> error "No value returned."
```

In order to make the `return` keyword jump out of the given block being executed, modify `execList` so that, before executing any instructions, it checks whether the special variable `""` has already been assigned a value.

Example

```
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
  Return (Var "X")]

sum100output = run sum100
```

7. Loading the source file

Question 5 (20 pts)

Write a *main* method which asks for a file to load, parses and loads the file into the `[Instr]` type, executes this program, and then outputs the value of the special variable.