Laboratory 2 - JAVA

Starting with this laboratory you are going to implement an interpreter for a toy language. You have to use the model-view-controller pattern and the object-oriented concepts.

Toy Language Description

A program (Prg) in this language consists of a statement (Stmt) as follows:
Prg := Stmt where the symbol "::=" means "a Prg is defined as a Stmt".

A statement can be either a compound statement (CompStmt), or an assignment statement (AssignStmt), or a print statement (PrintStmt), or a conditional statement (IfStmt) as follows:

```
Stmt ::= Stmt1 ; Stmt2                          /* (CompStmt)*/
       | Id = Exp                               /* (AssignStmt)*/
       | Print(Exp)                             /* (PrintStmt)*/
       | If  Expr Then Stmt1  Else Stmt2        /*   (IfStmt)*/
```

where the symbol "|" denotes the possible definition alternatives.

An expression (Exp) can be either an integer number (Const), or a variable name (Var), or an arithmetic expression (ArithExp) as follows:

```
Exp ::= Number                    /*(Const)*/
       | Id                       /*(Var)*/
       | Exp1 + Exp2              /*(ArithExp)*/
       | Exp1  - Exp2
       | Exp1 * Exp2
       | Exp1 / Exp2
```

where Number denotes an integer constant, and Id denotes a variable name.

Example1:
```
 v=2;
Print(v)
```

Example2:
```
a=2+3*5;
b=a-4/2 + 7;
Print(b)
```

Example3:
```
a=2-2;
If a Then v=2 Else v=3;
Print(v)
```

note that we used C convention, a zero expression means false, otherwise it means true.

Toy Language Evaluation (Execution):

Our mini interpreter uses three main structures:

- Execution Stack (ExeStack): a stack of statements to execute the currrent program
- Table of Symbols (SymTable): a table which keeps the variables values
- Output (Out): that keeps all the mesages printed by the toy program

All these three main structures denote the given program state (PrgState). Our interpreter can execute multiple programs but for each of them use a different PrgState (that means different ExeStack, SymTable and Out structures).

At the beginning ExeStack contains the orginal program, and SymTable and Out are empty. After the evaluation has started ExeStack contains the remaining part of the program that must be evaluated, SymTable contains the variables (from the assignment statements evaluated so far) with their assigned values, and Out contains the values printed so far.

In order to explain the program evaluation rules, we represent ExeStack as a collection of statements separated by the symbol "|", SymTable as a collection of mappings and Out as a collection of messages.
For example the ExeStack {s1 | s2 | s3} denotes a stack that has the statement s1 on top of it, then the statement s2 and at the bottom of the stack is the statement s3.
For example SymTable {v->2,a->4} denotes the table containing only two variables v and a, v has been assigned the value 2 while a has been assigned the value 4.
For example Out {1,2} denotes the printed values, the order of printing is 2 followed by 1.

At the end of a program evaluation ExeStack is empty, SymTable contains all the program variables, and Out contains all the program print outputs.

Statement Evaluation Rules: are described by presenting the program state (ExeStack1,SymTable1,Out1) before applying the evaluation rule, the symbol "==>" for one step evaluation and the program state (ExeStack2,SymTable2,Out2) after applying the evaluation rule. Each evaluation rules shows how the program state is changed. One step evaluation means that only one statement evaluation rule is applied. Complete evaluation means that all possible evaluation rules are applied until the program evaluation terminates.

S1. CompStmt evaluation: when a compound statement is the top of the ExeStack
ExeStack1={Stmt1;Stmt2 | Stmt3|....}
SymTable1
Out1
==>
ExeStack1={Stmt1| Stmt2 | Stmt3|.....}
SymTable2=SymTable1
Out2 = Out1
As you can see the top of the ExeStack is changed while SymTable and Out remain unchanged.

S2. AssignStmt evaluation: an assignment statement is on top of the stack
ExeStack1={Id=Exp| Stmt1|....}
SymTable1
Out1
==>

ExeStack1={Stmt1|...}
SymTable2=SymTable1 U {Id->Eval(Exp)}
Out2 = Out1

where Eval(Exp) denotes the expression evaluation and the rules are explained a bit later.
The symbol "U" denotes the union of two collections.

S3. PrintStmt evaluation
ExeStack1={Print(Exp)| Stmt1|....}
SymTable1
Out1
        ==>
ExeStack1={Stmt1|...}
SymTable2=SymTable1
Out2 = Out1 U {Eval(Exp)}

S4. IfStmt evaluation
ExeStack1={If Exp Then Stmt1 Else Stmt2 | Stmt3|....}
SymTable1
Out1
        ==>
if Eval(Exp)=1 ExeStack1={Stmt1|Stmt3|...} else ExeStack1={Stmt2|Stmt3|...}
SymTable2=SymTable1
Out2 = Out1

S5. Program termination
ExeStack ={}
SymTable
Out
==>
end of the program evaluation

Expression evaluation rules are presented using recursive rules. These rules do not change the program state:

E1. Const Evaluation:
Eval(Number) = Number
the constant is returned by evaluation

E2. Var Evaluation:
Eval(Id) = LookUp(SymTable,Id)

where LookUp(SymTable,Id) returns the value which is mapped to the variable Id. If the variable Id does not exist in SymTable LookUp returns an exception.
Examples:
LookUp({a->2,b->3},a)=2
LookUp({a->2,b->3},x) raised the exception "variable x is not defined"

E3. ArithExp evaluation:
Eval(Exp1 + Exp2)= Eval(Exp1) + Eval(Exp2)

Eval(Exp1 - Exp2)= Eval(Exp1) - Eval(Exp2)
Eval(Exp1 * Exp2)= Eval(Exp1) * Eval(Exp2)
Eval(Exp1 / Exp2)= Eval(Exp1) / Eval(Exp2)

First left operand and right operand are evaluated to values. Then those two values are combined by the arithmethic operator (+,-,* or /) and the final value is computed.

Laboratory 2 Assignment

Final Deadline: Laboratory 5
First Progress Check: Laboratory 3
Second Progress Check: Laboratory 4

1. Use the Model-View-Controler architecture to implement the toy language interpreter.
2. Use Test-Driven Development to develop the interpreter.
3. Each method must be documented with pre and post conditions.
4. The tests for each public method must be presented.

5. Model: Design and Implement a hierarchy of classes for the toy language statements. Note that later we will add more statements to the language.
6. Model: Design and Implement a hierarchy of classes for the toy language expressions. Note that later we will add more expressions to the language.
7. Model: Design and Implement the ExeStack, Out and SymTable for a program state. For this phase of the project design and implement your ADT Stack using arrays in order to represent ExeStack, use an array of strings to represent Out and use an array of pairs (String,int) to represent SymTable.
8. Repository: The repository must contain the state of a program. Note that later the repository may contain the states of multiple programs.
9. Controller: The controller must implement the following functionalities: one step evaluation of a program (a statement) using the program state, complete evaluation of a program (statement) using the program state, set the program state, display the program state, display the complete evaluation steps (including all the intermmediate program state)
10. View: At this phase of the project design and implement a text interface for the following functionalities: input a program, one-step evaluation of a program, complete evaluation of a program, debug evaluation (all the steps and states are displayed). The input of a program must be done through a set of submenus. For example in order to introduce a statement, first the user is asked to introduce the type of the statement and then according to the introduced type the user is asked to introduced the substatements or expressions. The expressions are introduced in the same manner. The statements and expressions are directly stored in their corresponding instance objects (e.g. AssignStmt into an object of the class AssignStm and so on).