IE- Laboratory 3
week 3 (15-19 October 2018)

Please solve the following assignment A2:
DEADLINE of A2 is week 5 (29.10.2018 – 02.11.2018)

Starting with this laboratory you are going to implement an interpreter for a toy language. You have to use the model-view-controller architectural 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 language 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 program state (PrgState). Our interpreter can execute multiple programs but for each of them use a different PrgState structures (that means different ExeStack, SymTable and Out structures).

At the beginning, ExeStack contains the original 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, followed by 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 assigned the value 2, while a has 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. Termination means that there is no any evaluation rule that can be further applied.

S1. CompStmt evaluation: when a compound statement is the top of the ExeStack

ExeStack1={Stmt1;Stmt2 | Stmt3|....}
SymTable1
Out1
==>
ExeStack2={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
==>
ExeStack2={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
==>
ExeStack2={Stmt1|...}
SymTable2=SymTable1
Out2 = Out1 U {Eval(Exp)}

S4. IfStmt evaluation:
ExeStack1={If Exp Then Stmt1 Else Stmt2 | Stmt3|....}
SymTable1
Out1
==>
if Eval(Exp)!=0 ExeStack2={Stmt1|Stmt3|...} else ExeStack2={Stmt2|Stmt3|...}
SymTable2=SymTable1
Out2 = Out1

S5. Program termination:
ExeStack1 ={}
SymTable1
Out1
==>
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.

Please use JAVA to implement the following tasks:

1. Use the Model-View-Controler architectural pattern to implement the toy language interpreter.

2. Model (or Domain):
   2.1. Design and Implement the classes for the toy language statements. Note that later we will add more statements to the language. Each class has a method that prints the corresponding statement into a String in a readable manner (using the language grammar described above) and a method to execute the statement. You may want to implement the following approach:

```
interface  IStmt{ ....
     String toStr(); //optional, instead you can override toString inherited from class Object
     PrgState execute(PrgState state); //which is the execution method for a statement.
       ....}
class CompStmt implements IStmt {
         IStmt first;
          IStmt snd;
        ......
   String toStr() {
        return "("+first.toStr() + ";" + snd.toStr()+")";
         }
   PrgState execute(PrgState state){
        MyIStack<IStmt> stk=state.getStk()
        stk.push(snd);
        stk.push(first);
        return state;
   }
   }
}
```

```
class PrintStmt implements IStmt{
    Exp exp;
     ....
    String toStr(){ return "print(" +exp.toStr()+")");}
      PrgState execute(PrgState state){
          ......
           return state;
    }
    ...
 }

class AssignStmt implements IStmt{
     String id;
    Exp exp;
     ....
    String toStr(){ return id"="+ exp.toStr();}
    PrgState execute(PrgState state){
         MyIStack<IStmt> stk=state.getStk()
         MyIDictionary<String,Integer> symTbl= state.getSymTable();
         int val = exp.eval(symTbl);
          if symTbl.isDefined(id) symTbl.update(id, val)
          else symTbl.add(id,val)
         return state;
    }
    ...
 }

class IfStmt implements IStmt{
    Exp exp;
    IStmt thenS;
    IStmt elseS;
     ....
    IfStmt(Exp e, IStmt t, IStmt el) {exp=e; thenS=t;elseS=el;}
    String toStr(){ return "IF("+ exp.toStr()+") THEN(" +thenS.toStr()
+")ELSE("+elseS.toStr()+")";}
     PrgState execute(PrgState state){
          ......
           return state;
    }
    ...
 }
```

2.2. Design and Implement a hierarchy of classes for the toy language expressions. Note that later we will add more expressions to the language. Each expression class has an overridden method eval which takes as argument a reference to the SymTable and returns an integer. You may want to implement the following approach:

```
abstract class Exp {
  abstract int eval(MyIDictionary<String,Integer> tbl);
 abstract String toStr();//again other option is to use toString instead of this toStr
}
```

```
class ArithExp  extends Exp{
    Exp e1;
    Exp e2;
    int op; //1 stands for +, 2 for -, etc
   ....
   //override
   int eval(MyIDictionary<String,Integer> tbl) {
        if (op==1) return (e1.eval(tbl)+e2.eval(tbl));
        .............
    }
}

class ConstExp extends Exp{
    int number;
   ....
   int eval(MyIDictionary<String,Integer> tbl) {return number;}
    ....}

class VarExp extends Exp{
    String id;
   ....
   int eval(MyIDictionary<String,Integer> tbl) {return tbl.lookup(id);}
    ....}
```

2.3. Design and Implement the ExeStack, Out and SymTable for a program state. ExeStack must be designed as a generic ADT Stack, Out must be designed as a generic ADT List, and SymTable must be designed as a generic ADT Dictionary. The implementation of each ADT must consist of a generic interface and a generic class that implements the interface.
For example ADT Stack is implemented as follows:

```
interface   MyIStack<T>{
      .....
      T pop();
      void push(T v);

}
```

The generic class which implements the interface is a wrapper for a Java generic library class such that:

```
class MyStack<T> implements MyIStack<T>{
    Type<T>  stack; //a field whose type Type is an appropriate generic java library
                    //collection
    .....................

}
```

You must to implement ADT List and ADT Dictionary in the same manner. Please use the appropriate generic collections from Java generic libraries in order to implement those 3 ADTs.
Your class PrgState must have the following fields:

```
class PrgState{
   MyIStack<IStmt>  exeStack;
   MyIDictionary<String, Integer>   symTable;
   MyIList<Integer>   out;
   IStmt   originalProgram; //optional field, but good to have
```

```
PrgState(MyIStack<IStmt> stk, MyIDictionary<String,Integer> symtbl,
MyIList<Integer> ot, Istmt prg){
                    exeStack=stk;
                    symTable=symtbl;
                    out = ot;
                    originalProgram=deepCopy(prg);//recreate the entire original prg
                     stk.push(prg);
                }
                .....
            }
```

Note that PrgState class must have a method to print into a String its structure and also must have getters and setters for all fields.

When you create object instances of your class PrgState you have to provide object instances of the classes which implement the interfaces MyIStack<T1>, MyIDictionary<T2,T3> and MyIList<T4> respectively:

```
MyIStack<IStmt> stk = new MyStack<IStmt>(....);
MyIDictionary<String,Integer>   symtbl =
                        new MyDictionary<String,Integer>(...);
MyIList<Integer>   out = new MyList<Integer>(...);
PrgState crtPrgState= new PrgState(stk,symtbl,out);
```

where the class MyStack<T1> implements the interface MyIStack<T1>,
the class MyDictionary<T2,T3> implements the interface MyIDictionary<T2,T3>,
and the class MyList<T4> implements the interface MyIList<T4>.

3. Repository: The repository must contain the state of a program. Note that later the repository may contain the states of multiple programs. Perhaps you may want to implement the repository as List of PrgState objects. Repository must be a class which implements an interface that has at least (for the moment) the method : PrgState getCrtPrg(). Later the repository interface will be extended with more methods.

4. Controller: The controller maintain a reference to the respository. The reference type is an interface such that we can easily modify the repository implementation. The controller must implement the following functionalities:
   4.1. one step evaluation of a program (one program statement) using the program state: The controller contains a method which takes one of the PrgStates from repository, analyse the top of the ExeStack of that PrgState and based on the content of the ExeStack top executes one of the S rules (Statement evaluation rules) presented above. For example you may want to implement the following approach:

```
PrgState oneStep(PrgState state){
  MyIStack<IStmt> stk=state.getStk();
  if(stk.isEmpty()) throws MyStmtExecException;
  IStmt  crtStmt = stk.pop();
  return crtStmt.execute(state);
}
```

The statement classes execute methods and oneStep method return a PrgState. We will use the returned PrgState for our later assignments when our language will create threads.

4.2. complete evaluation of a program (all the program statements) using the program state. For example you can implement as follows:

```
void allStep(){
```

```
PrgState prg = repo.getCrtPrg(); // repo is the controller field of type MyRepoInterface
while (!prg.getStk().isEmpty()){
      oneStep(prg);
      //here you can display the prg state
}}
or using exceptions as follows:
void allStep(){
      PrgState prg = repo.getCrtPrg(); // repo is the controller field of type
                                       //    MyRepoInterface
      try{
            while(true){
                  oneStep(prg);
                  //here you can display the prg state
            }
      }
      catch(MyStmtExecException e) {}
}
```

4.3. display the current program state. You may want to display the program state after each execution step if the display flag is set to on.

5. 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.
For the menu option "input a program" you may allow the user to select a program from your programs already hardcoded in your main method.
For example the following programs written in our toy language are stored in the repository as follows:
Example1:
v=2;Print(v) is represented as:
IStmt ex1= new CompStmt(new AssignStmt("v",new ConstExp(2)), new PrintStmt(new VarExp("v")))

Example2:
a=2+3*5;b=a+1;Print(b) is represented as:
IStmt ex2 = new CompStmt(new AssignStmt("a", new ArithExp('+',new ConstExp(2),new ArithExp('*',new ConstExp(3), new ConstExp(5)))),
new CompStmt(new AssignStmt("b",new ArithExp('+',new VarExp("a"), new ConstExp(1))), new PrintStmt(new VarExp("b")))

Example3:
a=2-2;(If a Then v=2 Else v=3);Print(v) is represented as
Stmt ex3 = new CompStmt(new AssignStmt("a", new ArithExp('-',new ConstExp(2), new ConstExp(2))),
new CompStmt(new IfStmt(new VarExp("a"),new AssignStmt("v",new ConstExp(2)), new AssignStmt("v", new ConstExp(3))), new PrintStmt(new VarExp("v"))))

6. Please extend the exceptions class hierarchy with your exceptions in order to treat the special situations that can occur during the interpreter execution. You must define your exceptions at least for the following situations:

- exceptional situations for your 3 ADTs (Stack, Dictionary and List) operations (e.g. writing into a full collection, reading from an empty collection, etc)

- expressions evaluation: Division by zero, variable not defined in symbol table
- statements execution: trying to execute when the execution stack is empty
- you may want to create exception chains, that means you have special exceptions for each level of your application architecture: domain exceptions, repository exceptions, controller exceptions and view exceptions. When an exception cannot be treated (the situation cannot be recovered) at one level, it is thrown an exception to the upper level: for example in controller you can catch the repository exceptions and then if it necessary you can throw again the controller exceptions to the view/main