Laboratory 3
week 3 (11-15 October 2021)

TASKS:
1. Please submit the assignment A1.
2. Please solve the following assignment A2. Please start to work in the class and complete it at home until the deadline.
DEADLINE of A2 is week 5(25.10.2021 – 29.10.2021)

Assignment A2

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)*/
       | Type Id                                /* (VarDecl) */
       |  Id = Exp                              /* (AssignStmt)*/
       |  Print(Exp)                            /* (PrintStmt)*/
       |  If  Expr Then Stmt1  Else Stmt2       /*   (IfStmt)*/
       |   nop                          /* No operation */
```

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

A type Type can be :

```
Type ::= int
       | bool
```

A value Value in our language can be either an integer number (ConstNumber), or boolean values like (ConstTrue) or (ConstFalse):

```
Value ::= Number    /*(ConstNumber)*/
       | True       /* (ConstTrue)*/
       | False      /* (ConstFalse)*/
```

An expression (Exp) can be either a value (Value), or a variable name (Var), or an arithmetic expression (ArithExp), or a logical expression (LogicExp), as follows:

```
Exp ::= Value                       /*Value*/
      |  Id                                 /*(Var)*/
```

| Exp1 + Exp2 /*(ArithExp)*/
| Exp1 - Exp2
| Exp1 * Exp2
| Exp1 / Exp2
| Exp1 and Exp2 /*(LogicExp)*/
| Exp1 or Exp2

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

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

Example3:
bool a;
a=false;
int v;
If a Then v=2 Else v=3;
Print(v)

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 variable declarations 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->false} denotes the table containing only two variables v and a, v has assigned the integer value 2, while a has assigned the boolean value flase.
For example Out {1,2} denotes the printed values, the order of printing is 1 followed by 2.

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 execution: 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 execution: an assignment statement is on top of the stack
ExeStack1={Id=Exp| Stmt1|....}
SymTable1
Out1
==>
ExeStack2={Stmt1|...}
SymTable2=if IsVarDef(SymTable1,Id) then
Val1=Eval(Exp)
If Type(Val1) == GetType(SymTable1,Id) then
Update(SymTable1,Id,Val1)
else Error: "Type of expression and type of variable do not match"
else Erorr: "Variable Id is not declared"
Out2 = Out1

where Eval(Exp) denotes the expression evaluation and the rules are explained a bit later;
IsVarDef(SymTable,Id) looks for the entry of Id in the table, returns False when it cannot be found and True otherwise;
Update(SymTable1,Id,Val1) update the value to which Id is mapped in SymTable1, for example Update({Id->8},Id,10) --> {Id->10)};
GetType(Eval(exp)) -> get the type of an expression;
GetType(SymTable1,Id) -> get the type of ID from the table

S3. VarDecl execution: a variable declaration statement is on top of the stack
ExeStack1={Type Id | Stmt1|....}

SymTable1
Out1
==>
ExeStack1={Stmt1|....}
if IsVarDef(SymTable1,Id) == True) then Error: "variable is already declared"
else SymTable2 = SymTable1 U {Id-> default value}
Out1

where default value of the variable is determined based on the variable declared type, as follows:
defaultValue(int) =0
defaultValue(bool) = false

S4. PrintStmt execution:
ExeStack1={Print(Exp)| Stmt1|....}
SymTable1
Out1
==>
ExeStack2={Stmt1|...}
SymTable2=SymTable1
Out2 = Out1 U {Eval(Exp)}

S5. IfStmt execution:
ExeStack1={If Exp Then Stmt1 Else Stmt2 | Stmt3|....}
SymTable1
Out1
==>
Cond= Eval(Exp
if getType(Cond)!=bool) error "conditional expr is not a boolean"
else
** if Cond ==True ExeStack2={Stmt1|Stmt3|...} else ExeStack2={Stmt2|Stmt3|...}**
SymTable2=SymTable1
Out2 = Out1

S6. NOP execution:
ExeStack1={Nop | Stmt1|....}
SymTable1
Out1
==>
ExeStack2={Stmt1|...}
SymTable2=SymTable1
Out2 = Out1

S7. Program termination:
ExeStack1 ={}
SymTable1
Out1
==>

end of the program execution

Expression evaluation rules are presented using recursive rules. These rules do not change the program state. During the evaluation, we also have to check the operands types for some specific operations. Therefore we also introduce the rules for the type computation, as follows:

E1. Value Evaluation:
Eval(Number) = Number
Eval(True) = True
Eval(False) = False
the constant is returned by evaluation

getType(Number)=int
getType(True)=bool
getType(False)=bool

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

where LookUp(SymTable,Id) returns the value to which is mapped 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"

getType(Id)=getType(Eval(Id))

E3. ArithExp evaluation:

```
Eval(Exp1 + Exp2)=
nr1= Eval(Exp1)
if getType(nr1)== int  then
      nr2=Eval(Exp2)
     if getType(nr2)== int  then
                (nr1+ nr2)
     else Error: "Operand2 is not an integer"
else Error: "Operand1 is not an integer"
```

The same for -,*.

```
Eval(Exp1 / Exp2)=
nr1= Eval(Exp1)
if getType(nr1)== int  then
   nr2=Eval(Exp2)
   if getType(nr2)== int  then
        if nr2 !=0 then
                (nr1/nr2)
         else Error:"Division by 0"
   else Error: "Operand2 is not an integer"
```

else Error: "Operand1 is not an integer"

getType(Exp1+Exp2) =getType(Eval(Exp1+Exp2))
the same for -,*, and /.

E3. LogicExp evaluation:
Eval(Exp1 and Exp2)=
nr1= Eval(Exp1)
if getType(nr1)== bool then
nr2=Eval(Exp2)
if getType(nr2)== bool then
(nr1&& nr2)
else Error: "Operand2 is not a boolean"
else Error: "Operand1 is not a boolean"

getType(Exp1 and Exp2) = getType(Eval(Exp1 and Exp2))
The same for or.

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. You may want to implement the following approach:

```
interface IStmt{
    PrgState execute(PrgState state) throws MyException;
     //which is the execution method for a statement.
      }


class CompStmt implements IStmt {
        IStmt first;
         IStmt snd;
       .....
  String toString() {
       return "("+first.toString() + ";" + snd.toString()+")";
        }
  PrgState execute(PrgState state)  throws MyException{
      MyIStack<IStmt> stk=state.getStk()
      stk.push(snd);
      stk.push(first);
      return state;
  }
```

```
    }
  }
```

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

           class AssignStmt implements IStmt{
               String id;
    Exp exp;
     ....
              String toString(){ return id+"="+ exp.toString();}
              PrgState execute(PrgState state) throws MyException{
          MyIStack<IStmt> stk=state.getStk();
          MyIDictionary<String,Value> symTbl= state.getSymTable();

           if symTbl.isDefined(id){
                    Value val = exp.eval(symTbl);
                    Type typId= (symTbl.lookup(id)).getType();
                    if (val.getType()). equals(typId)
                              symTbl.update(id, val)
                         else throw new MyException("declared type of variable"+id+" and type of
                                                    the assigned expression do not match");
           else throw new MyExeption("the used variable" +id + " was not declared before");
          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 toString(){ return "(IF("+ exp.toString()+") THEN(" +thenS.toString()
                                                    +")ELSE("+elseS.toString()+"))";}
               PrgState execute(PrgState state) throws MyException{
          ......
```

```
        return state;
    }
            ...
        }


        class VarDeclStmt implements IStmt{
            string name;
            Type typ;
            .....
        }


        class NopStmt implements Istmt{
         ..........................
              }
```

2.2. Design and Implement a hierarchy of classes for the toy language types. Note that later we will add more types to the language. You may want to implement the following approach:

```
interface Type{}

        class IntType implements Type{
          boolean equals(Object another){
                if (another instanceof  IntType)
                        return true;

                else
                        return false;
          }
          String toString() { return "int";}
         }

         class BoolType implements Type {....}
```

2.3. Design and Implement a hierarchy of classes for the toy language values. Note that later we will add more values to the language. You may want to implement the following approach:

```
            interface Value {
              Type  getType(); }

         class IntValue implements Value{
                int val;
                IntValue(int v){val=v;}

                int getVal() {return val;}
                String toString() {...}
                Type getType() { return new IntType();}
```

```
        }

        class BoolValue implements Value{ ....}
```

2.4. 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 object Value. You may want to implement the following approach:

```
interface Exp {
  Value eval(MyIDictionary<String,Value> tbl) throws MyException;
}

class ArithExp  implements Exp{
    Exp e1;
    Exp e2;
          int op; //1-plus, 2-minus, 3-star, 4-divide
         ....

        Value eval(MyIDictionary<String,Value> tbl)  throws MyException{
            Value v1,v2;
             v1= e1.eval(tbl);
             if (v1.getType().equals(new IntType())) {
                    v2 = e2.eval(tbl);
                     if (v2.getType().equals(new IntType())) {
                           IntValue i1 = (IntValue)v1;
                           IntValue i2 = (IntValue)v2;
                          int n1,n2;
                          n1= i1.getVal();
                          n2 = i2.getVal();
                         if (op==1)  return new IntValue(n1+n2);
                          if (op ==2)  return new IntValue(n1-n2);
                          if(op==3)  return new IntValue(n1*n2);
                         if(op==4)
                               if(n2==0) throw new MyException("division by zero");
                               else  return new IntValue(n1/n2);
                    }else
                       throw new MyException("second operand is not an integer");
             }else
                throw new MyException("first operand is not an integer");
        }
}
```

```
    class ValueExp implements Exp{
          Value e;
         ....
```

```
        Value eval(MyIDictionary<String,Value> tbl)  throws MyException{return e;}
         ....}
```

```
    class LogicExp implements Exp{
          Exp e1;
          Exp e2;
          int op;
          ....
        Value eval(MyIDictionary<String,Value> tbl) throws MyException{....}
          ....}
```

```
    class VarExp implements Exp{
          String id;
          ....
        Value eval(MyIDictionary<String,Value> tbl)  throws MyException
                  {return tbl.lookup(id);}
          ....}
```

2.5. 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>{
      CollectionType<T>  stack; //a field whose type  CollectionType 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, Value>   symTable;
```

```
MyIList<Value>   out;
IStmt   originalProgram; //optional field, but good to have

      PrgState(MyIStack<IStmt> stk, MyIDictionary<String,Value> symtbl,       MyIList<Value>
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 override toString method 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,Value>   symtbl =
new MyDictionary<String,Value>(...);
            MyIList<Value>   out = new MyList<Value>(...);
            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 threads of a program. Perhaps you may want to implement the repository as a 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 maintains 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 execution 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 execution rules) presented above. For example you may want to implement the following approach:

```
        PrgState oneStep(PrgState state) throws MyException{
  MyIStack<IStmt> stk=state.getStk();
  if(stk.isEmpty()) throws  new MyException("prgstate stack is empty");
            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 execution 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
  //here you can display the prg state
 while (!prg.getStk().isEmpty()){
       oneStep(prg);
       //here you can display the prg state
 }}
```

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 and complete execution 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:
int v; v=2;Print(v) is represented as:

```
IStmt ex1= new CompStmt(new VarDeclStmt("v",new IntType()),
new CompStmt(new AssignStmt("v",new ValueExp(new IntValue(2))), new PrintStmt(new
VarExp("v"))))
```

Example2:
int a;int b; a=2+3*5;b=a+1;Print(b) is represented as:

```
IStmt ex2 = new CompStmt( new VarDeclStmt("a",new IntType()),
 new CompStmt(new VarDeclStmt("b",new IntType()),
new CompStmt(new AssignStmt("a", new ArithExp('+',new ValueExp(new IntValue(2)),new
ArithExp('*',new ValueExp(new IntValue(3)), new ValueExp(new IntValue(5))))),
  new CompStmt(new AssignStmt("b",new ArithExp('+',new VarExp("a"), new ValueExp(new
IntValue(1)))), new PrintStmt(new VarExp("b")))))
```

Example3:
bool a; int v; a=true;(If a Then v=2 Else v=3);Print(v) is represented as

```
Stmt ex3 = new CompStmt(new VarDeclStmt("a",new BoolType()),
   new CompStmt(new VarDeclStmt("v", new IntType()),
 new CompStmt(new AssignStmt("a", new ValueExp(new BoolValue(true))),
 new CompStmt(new IfStmt(new VarExp("a"),new AssignStmt("v",new ValueExp(new
 IntValue(2))), new AssignStmt("v", new ValueExp(new IntValue(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 exception 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