Laboratory 4
(week 24-30 October 2017)

TASKS:

A. Please submit the Lab-Assignment 2 to your lab professor.

B. Please start to work on the Lab-Assignment 3.

The deadline of Lab-Assignment 3 is week 6 (7-13 November 2017).

Lab-Assignment 3

You must work on the JAVA project from **Lab-Assignment 2**. Please implement the followings:

1. Save the repository content into a log text file:

Repository:

Add to the repository interface a method to save the content of the PrgState into a text file:

void logPrgStateExec().

Then implement this method in the Repository class using PrintWriter in append mode (e.g. logFile= new PrintWriter(new BufferedWriter(new FileWriter(logFilePath, true))).

The logFilePath is a new field of the Repository class which contains the path to the log text file. This field is initialized by a string read from the keyboard using Scanner class.

The text file will have the following structure:

ExeStack:
Top of the stack as a string
Top-1 of the stack as a string
.....
Bottom of the stack as a string
SymTable:
var_name1 --> value1
var_name2 --> value2
....
Out:
value1
value2
.....
FileTable:
id1 --> filename1
id2 --> filename2

The above bold names must be present in your file to denote the starting of each section (ExeStack, SymTable, Out, and FileTable). Note that each stack position must be saved as a string that contains the statement which is on that position of the stack. Since internally the statements (and expressions) are represented as binary tree please use the left-root-right binary tree traversal in order to print in infix form the statements and the expressions.

Controller:

Modify the code of the method allStep() of the Controller class such that the content of the repository is saved into a log text file after each oneStep(prg) call, as follows:

```
void allStep(){
        PrgState prg = repo.getCrtPrg();
        try{    while(true){
```

```
                    oneStep(prg);
                    repo. logPrgStateExec(); }
            }
            catch(MyStmtExecException e) {}
           catch (....) ....
    }
```

2. Add new statements to allow file operations in the ToyLanguage:

2.1. Implement a new table FileTable that manages the files opened in our Toy Language. The FileTable must be supported by all of the previous statements. FileTable is part of PrgState and it is a dictionary mapping Toy Language file descriptor (that is an integer) to the following tuple (filename, file descriptor from Java language). The Toy Language file descriptor is managed by your FileTable implementation and must be unique. The filename is a string and denotes the path to the file. The file descriptor from Java language is an instance of the BufferedReader class.
We assume that a file can only be a text file that contains only non-zero positive integers, one integer per line. For example the content of a file can be the following:

1<NL>
2<NL>
3<NL>
<EOF>

The table FileTable will be implemented in the same manner as ExeStack, SymTable and Out (namely the interface and its class implementation).

2.2. Implement the Statement openRFile(var_file_id,filename), where var_file_id is a variable name and filename is a string. Its execution on the ExeStack is the following:

- pop the statement
- check whether the filename is not already in the FileTable. If it exists stopped the execution with an appropriate error message.
- open the file filename in Java using an instance of the BufferedReader class. If the file does not exist or other IO error occurs stopped the execution with an appropriate error message.
- create a new entrance into the FileTable which maps a new unique integer key to the (filename, instance of the BufferedReader class created before).
- set the var_file_id to that new unique integer key (created at the previous step) into the SymTable.

2.3. Implement the Statement: readFile(exp_file_id, var_name), where exp_file_id is an expression and var_name is variable name. Its execution on the ExeStack is the following:

- pop the statement
- evaluate exp_file_id to a value
- using the previous step value we get the BufferedReader object associated in the FileTable. If there is not any entry associated to this value in the FileTable we stop the execution with an appropriate error message.
- Reads a line from the file using readLine method of the BufferedReader object. If line is null create a zero int value. Otherwise translate the returned String into an int value (using Integer.parseInt(String)).
- Add a new mapping (var_name, int value computed at the previous step) into the SymTable. If var_name exists in SymTable update its associated value instead of adding a new mapping.

2.4. Implement the Statement: closeRFile(exp_file_id), where exp_file_id is an expression. Its execution on the ExeStack is the following:

- pop the statement
- evaluate exp_file_id to a value. If any error occurs then terminate the execution with an appropriate error message.
- Use the value (computed at the previous step) to get the entry into the FileTable and get the associated BufferedReader object. If there is not any entry in FileTable for that value we

stop the execution with an appropriate error message.

- call the close method of the BufferedReader object
- delete the entry from the FileTable

2.5. Test your implementation using the following test-examples:

2.5.1. Example 1:

```
openRFile(var_f,"test.in");
readFile(var_f,var_c);print(var_c);
(if var_c then readFile(var_f,var_c);print(var_c)
else print(0));
closeRFile(var_f)
```

2.5.2. Example 2:

```
openRFile(var_f,"test.in");
readFile(var_f+2,var_c);print(var_c);
(if var_c then readFile(var_f,var_c);print(var_c)
else print(0));
closeRFile(var_f)
```

and the **"test.in" file** contains the followings:

```
15<NL>
50<NL>
<EOF>
```

You can also check the above example with an **empty test.in file**.

3. Implement the view part of your MVC architecture:

The view part consists of a text menu from which the user can select the command that is going to be executed by the controller. An instance object of the TextMenu class will be created in the main method. Main method is defined in a main class (let call it Interpreter class). Please implement the following classes and modify your main method as follows:

TextMenu class:

```
public class TextMenu {
   private Map<String, Command> commands;
   public  TextMenu(){ commands=new HashMap<>(); }
   public void addCommand(Command c){ commands.put(c.getKey(),c);}
   private void printMenu(){
      for(Command com : commands.values()){
         String line=String.format("%4s : %s", com.getKey(), com.getDescription());
         System.out.println(line);
      }
   }
 public void show(){
      Scanner scanner=new Scanner(System.in);
      while(true){
         printMenu();
         System.out.printf("Input the option: ");
         String key=scanner.nextLine();
         Command com=commands.get(key);
         if (com==null){
            System.out.println("Invalid Option");
            continue;  }
         com.execute();
         }
     }
}
```

Command class that is an abstract class:

```
public abstract class Command {
   private String key, description;
   public Command(String key, String description) { this.key = key; this.description = description;}
   public abstract void execute();
```

```
    public String getKey(){return key;}
    public String getDescription(){return description;}
}
```

Subclasses of the Command class:

For each option of the menu you must derive a corresponding class.

Exit Command:

```
public class ExitCommand extends Command {
    public ExitCommand(String key, String desc){
        super(key, desc);
    }
    @Override
    public void execute() {
        System.exit(0);
    }
}
```

Run an example coded into the main method:

```
public class RunExample extends Command {
    private Controller ctr;
    public RunExample(String key, String desc,Controller ctr){
        super(key, desc);
        this.ctr=ctr;
    }
    @Override
    public void execute() {
      try{
        ctr.allStep();  }
       catch (...)  {} //here you must treat the exceptions that can not be solved in the controller
    }
}
```

The main method in Interpreter class:

```
class Interpreter {

public static void main(String[] args) {

IStmt ex1=new ....;
PrgState prg1 = new PrgState(..., ex1);
MyIRepository repo1 = new MyRepository(prg1,"log1.txt");
Controller ctr1 = new Controller(repo1);

IStmt ex2=new ....;
PrgState prg2 = new PrgState(..., ex2);
MyIRepository repo2 = new MyRepository(prg2,"log2.txt");
Controller ctr2 = new Controller(repo2);

IStmt ex3= new ...;
PrgState prg3 = new PrgState(..., ex3);
MyIRepository repo3 = new MyRepository(prg3,"log3.txt");
Controller ctr3 = new Controller(repo3);

TextMenu menu = new TextMenu();
menu.addCommand(new ExitCommand("0", "exit"));
menu.addCommand(new RunExample("1",ex1.toString(),ctr1));
menu.addCommand(new RunExample("2",ex2.toString(),ctr2));
menu.addCommand(new RunExample("3",ex3.toString(),ctr3));
menu.show();
}
```

Obs: you can run only once an example.