Project Report

Group 24
COMP2021 Object-Oriented Programming(Fall 2022)
HAN Wenyu 21097519D
HU Wenqing 21094549D
ZHOU Siyu 21094655D

1. Introduction

This document reports the design and implementation of an interpreter for SIMPLE programs by group 24. The SIMPLE program supports two data types, fifteen different operators and seven types of statements. Each variable or expression in the SIMPLE program has a distinct name, and each statement is identifiable by a unique label. The project is part of the course COMP2021 Object-Oriented Programming at PolyU.

2. A Command-Line-Based Interpreter for Simple Programs

In this section, we describe first the overall design and then implementation details of the interpreter.

2.1 Design

Description for UML class diagram:

The user enters the simple code through *Class Simple*, and the system counts each keyword and token in the code(*enumeration token*), uses class *individualToken* and stores it in the *typeMap*, *keywordMap* and *operatorMap* that have implemented interface *HashMaps* (Here we use implements relationship).

As for dependency relationships, because there are so many weak dependency relationships, we only draw three strong dependency relationships from Java class Simple to class Block, While, If and Runner.

For association relationships, we don't have any object reference from another object.

For composition relationships, class CheckExpression was realized with class Expression. Class Block, While, InputOutput, If, Runner are invoked by class Program. Class *Simple* calls the class *Program* to load the program, and then calls class *Execute* to execute the program, and show the output here. Then we have composition relations here from *Simple* to *Program*, and from *Simple* to *Execute*. Again for execute, there's a class *Error* for checking different types of errors, such as checking type define error (*checkTypeDefError(int line)*), check if integer is out of range (*checkIntegerOutofRangeErroe(int line)*), and so on.
(There's few other explanations for other composition relationships in the UML Class Diagram.

UML Class Diagram:

2.2 Requirement

(For each (compulsory and bonus) requirement, describe 1) whether it is implemented and, when yes, 2) how you implemented the requirement as well as 3) how you handled various error conditions.)

[REQ1]

1) *vardef* The requirement is implemented.

2) For defining variables, we use HashMap to achieve capabilities. The first HashMap named *definedBool* stores boolean variables. The second HashMap named *definedInt* is for integer data type. When the variable definition statement is executed, it will be initialized to the value of expression reference *expRef*, where type type is either boolean or integer. To be more specific, class *variableMap* is used to store variables, and it also has relevant methods provided to read or update variables. *addToDefinedBool/addToDefinedInt* methods can add the defined variables to *definedBool/definedInt* HashMaps.

```
public class variableMaps{
    6 usages
    private final Map<String, Boolean> definedBool = new HashMap<>();
    6 usages
    private final Map<String,Integer> definedInt = new HashMap<>();

public void addToDefinedBool(String varName, Boolean value, int line) {  //defining variable
    if(definedBool.containsKey(varName)){
        Error.checkVarAlreadyDefinedError(line);
    }
    definedBool.put(varName,value);
}

   public void addToDefinedInt(String varName, Integer value, int line) {
       if(definedInt.containsKey(varName)){
           Error.checkVarAlreadyDefinedError(line);
       }
       definedInt.put(varName,value);
   }
```

Besides this two methods and HashMaps, another *addToVariables* method could add the name of already defined variables to a HashSet called *variables*, for later use.

```
private final HashSet<String> variables = new HashSet<~>();
```

3)

Error condition 1:

- When the user types a varname that is not allowed into the interpreter. SIMPLE has the following methods to deal with this error:
    - If the length of varname is longer than 8, system throws *checkTooLongVarName* exception.
    - they type keywords of "SIMPLE", the input will throw the *checkKeywords* exception.

- If a user's input contains the character that is not an English or numeric character, it will throw *checkInvalidVarName*.
- If the varname starts with a number, it will throw the *checkInvalidVarName* exception.

```
public static void varNameValid(String name,int line){
    //not keywords, length <= 8, contains digits and English character only
    if(name.length() > 8){      //check length
        Error.checkTooLongName(line);
    }
    for(Tokens keyword:Tokens.values()){    //check keywords.
        if(keyword.toString().toLowerCase().equals(name.toLowerCase())){
            Error.checkKeyword(line);
        }
    }
    if(Character.isDigit(name.charAt(0))){  //check starting with number
        Error.checkInvalidVarName(line);
    }
    if(!name.matches( regex: "[a-zA-Z0-9]+")){  //check only contains number and English Character
        Error.checkInvalidVarName(line);
    }
}
```

Error Condition 2:

- When the data type of value is neither integer nor boolean, other datatype is not supported in the interpreter,then it will throw the *checkTypeDefError* exception.

Error condition 3:

- When defining variables that have already been defined, SIMPLE will throw the *checkVarAlreadyDefinedError* exception.

Error condition 4:

- The last error condition is that the integer is out of the range between -99999 and 99999, then it will throw the *IntegerOutofRange* exception.

[REQ2]

1) *binexpr* The requirement is implemented.

2) For defining binary expressions, the results will be stored into a *HashMap* of *variablesMaps*. Left operand is an expression reference *expRef1* and the right operand is *expRef2*. *addToTempBool/addToTempInt* will save data into *tempBool* and *tempInt* HashMap, respectively. Then, *tempBool* will store the results of *boolean*, and *tempInt* will store the results of *integers*.

The program also provides *tempBoolContains/tempIntContains, getFromTempBool/getFromTempInt* methods to access the elements stored in the HashMaps. The *tempBoolContains/tempIntContains* will return a boolean value, and *getFromTempBool/getFromTempInt* will return a boolean/int value.

```
private final Map<String,Boolean> tempBool = new HashMap<>();
3 usages
private final Map<String,Integer> tempInt = new HashMap<>();
```

```
public void addToTempBool(String name,Boolean value) { tempBool.put(name,value); }
50 usages
public void addToTempInt(String name, Integer value) {
    tempInt.put(name,value);
}
```

If both sides of the binary operators are integer or boolean, the binary expression can directly do the calculation.
If there are expression or variable names on the right/left side of binary operators, then calculation will be computed after reading from the corresponding HashMap.
If the calculated int result is larger or smaller than the 99999 or -99999, it will be rounded to 99999 and -99999 respectively, by *rounding* method.

```
private int rounding(int numResult){
    if(numResult < NEGATIVE_MAX) {
        numResult = NEGATIVE_MAX;
    }
    else if(numResult > POSITIVE_MAX){
        numResult = POSITIVE_MAX;
    }
    return numResult;
}
```

3)
Error condition 1:

- when the value of an integer is out of the range between -99999 and 99999, then it will throw the *IntegerOutofRange* exception.

Error condition 2:

- Two sides between binary operators are neither integer nor boolean, it will throw the *unsupportedTypeOperation* exception.
- Meanwhile, when the operator and operand in the expression don't match, it will also throw the *unsupportedTypeOperation* exception.

Error condition 3:

- when the operator that user enters is not one of the 13 binary operators supported by the interpreter, it will throw the *checkInvalidOperation* exception.
- When the binary expression does division operations, if the divisor is equal to 0, which is against the arithmetic rules, it will throw the *divisonByZero* exception.

Error condition 4:

- The expression is marked as a binary expression, but if there are less than 5 single words in the statement, it will throw the *checkInvalidExpr* exception.

[REQ3]
1) *unexpr* The requirement is implemented.

2) For unary expression, it is quite similar to binary expression. The results will be stored into a Hashmap of *variableMaps,* according to the different type of operator and result. The boolean results store *tempBool*, and integer results store *tempInt*.
If the data type of operand is integer or boolean, the expression can directly do calculation.
If the data of operand is expression or variables, the calculation will be done after reading the value from *definedBool/definedInt/tempBool/tempInt*.
3)
Error condition 1:

- when the value of integer is out of the range, then it will throw the *IntegerOutofRange* exception.

Error condition 2:

- if the data type of operand is not integer or boolean, then it will throw the *unsupportedTypeOperation* exception.
- Meanwhile, when the operator and operand in the expression don't match, it will also throw the *unsupportedTypeOperation* exception.

Error condition 3:

- if the length of unexpre is less than four, then it will throw the *checkInvalidExpr* exception.

Error condition 4:

- if the operator is not one of "～, #, ！" for bool or int, then it will throw the *checkInvalidOperation*.

[REQ4]
1) *assign* The requirement is implemented.
2) For defining assignment statements, the value of expression reference *expRef* will be assigned to the variable with name *varName*. If the data type on the right side is int or bool, the value will directly store to the HashMap of *definedBool/definedInt,* by using *updateDefinedBool/updateDefinedInt* methods. If the data type of the right side is an expression or variable, the value will be searched from *tempBool/tempInt/definedBool/definedInt*, then assigned to the corresponding hashmap of *definedInt*, *definedBool*.

```
public void updateDefinedBool(String varName, Boolean value, int line){ //updating variable
    if(!definedBool.containsKey(varName)){
        Error.checkExpressionNotDefinedError(line);
    }
    definedBool.put(varName,value);
}
```

```
public void updateDefinedInt(String varName, Integer value, int line){
    if(!definedInt.containsKey(varName)){
        Error.checkExpressionNotDefinedError(line);
    }
    definedInt.put(varName,value);
}
```

3)
Error condition 1:

- if the varname in the left side is not defined, it will throw the *checkInvalidVarName* exception.

Error condition 2:

- when the value of integer is out of the range, then it will throw the *IntegerOutofRange* exception.
- Also, when the operator and operand in the expression don't match, it will also throw the *unsupportedTypeOperation* exception.

Error condition 3:

- if the length of the assignment statement that user enters is less than four, it will throw the *checkInvalidExpr* exception.

[REQ5]
1) *print* The requirement is implemented.
2) For defining print statements, if the data type of the right side expression is int or bool, the print statement will be directly executed.
If the data type of the right side expression is expression or varname, the print statement will be executed after searching from *definedBool/tempBool/definedInt/tempInt*. Overall, the print statement will call the System.out.print method directly to complete.
3)
Error condition 1:

- if the data type of right side expression is not integer, boolean, expression or variable name, then it will throw the *unsupportedTypeOperation* exception.

Error condition 2:

- When the data type of the right side is an expression or variable name, if they do not be defined program will throw the *checkExpressionNotDefinedError* exception.

Error condition 3:

- Also, if the length of the print statement is less than three, it will throw the *checkInvalidExpr* exception.

[REQ6]
1) *skip* The requirement is implemented.

2) For defining skip statements, when executed,the statement will be skipped after encountering it. Such a statement is designed by the continue loop, the statement that is set to be skipped will be switched, then case will break and do nothing.
3) As for the error handling of skip statements, if the length of expression is less than two, it will throw the *checkInvalidExpr* exception.

[REQ7]
1) *block* The requirement is implemented.
2) For defining block statements, it will execute one by one. The block statement contains a list of statements, so according to the different types of statements, various methods will be called. For example, if the type of statement is *binexpr*, then block statement will call *binexpr*. When this block statement is executed, the statements in the block statement are executed in sequence.

```java
public class Block {

    /**
     * @param expressions instance of expressionMaps, contains the expression already defined
     * @param blockName statement label of the block statement
     * @return return a list of IndividualToken
     * get the methods added to a block
     */
    3 usages
    public List<String> getBasicMethods(expressionMap expressions,String blockName){
        List<String> statements = new ArrayList<>();
        addToStatements(expressions,blockName,statements);
        return statements;
    }
```

3)
Error condition 1:
The first error is related to not defining expression, if the following statement after block statement is not be defined, then it throw the *checkExpressionNotDefinedError* exception.
Error condition 2:
Secondly, if the length of expression is less than three, it will throw the *checkInvalidExpr* exception.

[REQ8]
1) *if* The requirement is implemented.
2) For defining conditional statements, the first step is to determine the conditions. If it is a true value in a bool expression, it will directly execute the following corresponding true statements. By contrast, it will directly execute the following corresponding false statements. As for data type of int, if the value of integer is greater than 0. it will execute the true statements, if the value of integer is less or equal to 0, it will execute the false statements.
When it comes to the type of expression or varname, the block statement will determine if the expression is executed or not at first. After making sure it is executed,

the statements will read data from *tempBool/tempInt*. According to the determination from conditional statements, variable names read data from *definedBool/definedInt* at first, then calling the corresponding function for various types of statements.

```
public List<String> getExecuteOrder(expressionMap programExpressions, variableMaps variables, String programBody) {
    IndividualToken conditions = programExpressions.getContents(programBody).get(2);
    IndividualToken body1 = programExpressions.getContents(programBody).get(3);
    IndividualToken body2 = programExpressions.getContents(programBody).get(4);
    List<String> bodyStmt = new ArrayList<>();
    bodyStmt.add(conditions.getInputString());
    Tokens condiType = conditions.getType();
    switch (condiType) {
        case INTVALUE:
```

```
            else if(programExpressions.containsExpr(conditions.getInputString())) {
                bodyStmt = sendConditions(conditions.getInputString());
            }
            break;
```

```
public List<String> checkConditions(expressionMap programExpressions,variableMaps variables,String programName) {
    int line = programExpressions.getContents(programName).get(0).getLine();
    String condition = programExpressions.getContents(programName).get(2).getInputString();
    String trueBody = programExpressions.getContents(programName).get(3).getInputString();
    String falseBody = programExpressions.getContents(programName).get(4).getInputString();
    List<String> bodyStatement = new ArrayList<>();
```

3)

Error condition 1:

First of all, if conditional, true and false statements are not defined, it will throw the *checkExpressionNotDefinedError* exception.

Error condition 2:

Secondly, if the length of expression is not equal to five, it will throw the *checkInvalidExpr* exception.

[REQ9]

1) *while* The requirement is implemented.

2) Details:

For defining loop statements, when this loop statement is executed, the value of the expression reference *expRef* is evaluated repeatedly. It is similar to conditional statements, the first step is to determine the conditions, the method of determination is the same as conditional statements.

After executing the body part, then back to judgement of conditions. If the condition is true, then continue executing the body part, until the condition becomes false.

```
public List<String> getExecuteOrder(expressionMap programExpressions, variableMaps variables, String programBody) {
    IndividualToken conditions = programExpressions.getContents(programBody).get(2);
    IndividualToken body = programExpressions.getContents(programBody).get(3);
    int line = body.getLine();
    //Tokens bodyType = programExpressions.getContents(body.getInputString()).get(0).getType();
    List<String> bodyStmt = new ArrayList<>();
    checkConditions(programExpressions,variables,conditions,bodyStmt,body.getInputString(),programBody,line);
    return bodyStmt;
}
```

```
if(variables.definedBoolContains(condition.getInputString())){
    if(variables.getFromDefinedBool(condition.getInputString())){
        bodyStatement.add(bodyName);
        bodyStatement.add(programName);
    }
}
else if(variables.definedIntContains(condition.getInputString())){
    if(variables.getFromDefinedInt(condition.getInputString()) > 0){
        bodyStatement.add(bodyName);
        bodyStatement.add(programName);
    }
}
else if(programExpressions.containsExpr(condition.getInputString())) {
    //sendConditions(programExpressions,variables,condition);
    if(variables.tempBoolContains( name: "temp" + condition.getInputString())){
        if(variables.getFromTempBool( name: "temp" + condition.getInputString())){
            bodyStatement.add(condition.getInputString());
            bodyStatement.add(bodyName);
            bodyStatement.add(programName);
        }
    }
    else{
        bodyStatement.add(condition.getInputString());
        bodyStatement.add(programName);
    }
```

3)
Error condition 1:
First of all, if the conditional and body part statement is not defined, it will throw the *checkExpressionNotDefinedError* exception.
Error condition 2:
Secondly, if the length of expression is not equal to four, it will throw the *checkInvalidExpr* exception.

[REQ10]
1) *program* The requirement is implemented.
2) For defining SIMPLE programs, save the name of program SIMPLE and the statement labeled statementLab as programbody.
3)
Error condition 1:
Similarly, if the length of expression is less than three, it will throw the *checkInvalidExpr* exception.
Error condition 2:
Also, if the body program does not be defined, it will throw the *checkExpressionNotDefinedError* exception.

[REQ11]

1) *execute* The requirement is implemented.

2) For executing a defined simple program, the first step is setting a new program object, then transiting the program name and program body into the object. Then, all of the interpreters we designed before will be encapsulated in the program object, such as conditional and block statements. The whole program object will be considered as reference transmitted inwards the method of executeProgram to execute a simple program.

```
else if (list.get(0).getInputString().equals("execute")) {
    if(list.size() == 2){
        programName = list.get(1).getInputString();
        exe.executeProgram(programName, new Program(expressions,myWhile,myIf,interpreter,block,programName));
        System.out.println();
    }
    else{
        System.out.println("no defined program");
        continue;
    }
```

```
public Program(expressionMap expressions,While myWhile, If myIf, Runner interpreter, Block block,String programName){

    setProgramExpressions(expressions);
    setBlock(block);
    setInterpreter(interpreter);
    setMyIf(myIf);
    setMyWhile(myWhile);
    setProgramName(programName);
}
```

3)

Errors condition 1:

Similarly, if the length of expression is not equal to two, it will throw the *checkInvalidExpr* exception.

Error condition 2:

Also, if the program does not be defined, it will throw the *checkExpressionNotDefinedError* exception.

[REQ12]

1) *list* The requirement is implemented.

2) For listing a defined simple program to the console, the first step is calling the method of *outputExecutionOrder* in the execute part, then to print out a *list<String>*. All of the defined functions in the program will be saved in the list, accounting to the expression name in the *list<String>*, the program will search the corresponded expressions in the *expressionMap*. Finally, the results will print out.

```
public List<String> outputExecutionOrder(String programName, Program newProgram){
    Tokens statementType;
    List<String> statements = newProgram.setProgramExpression(programName);
    List<String> temp = new ArrayList<>();
    statements.add( index: 0,programName);
    statements.add( index: 1,newProgram.getProgramExpressions().getContents(programName).get(2).getInputString());
```

3)
Error condition 1:
Same as the error condition of a defined simple program, if the length of expression is not equal to two, it will throw the *checkInvalidExpr* exception.
Error condition 2:
Also, if the program does not be defined, it will throw the *checkExpressionNotDefinedError* exception.

[REQ13]
1) *store* The requirement is implemented.
2) For storing a defined simple program into a file, store the defined program with name *programName* into the file at path. Similar to listing a defined SIMPLE program to the console, it will call the method of *outputExecutionOrder*, then print out a *list<String>*. For storage, it is just a simple store in the appropriate location of *programName.simple*. Static method implemented in Class InputOutput

```
public static void storeExpressions(expressionMap expressions,List<String> statements,String path){
```

3)
Error condition 1:
- When it comes to the storage, we need to consider the error handling of the save path. If there is an error, the whole program will directly exit.

Error condition 2:
- Similarly, if the length of expression is not equal to two, it will throw the *checkInvalidExpr* exception.

Error condition 3:
- Also, if the program does not be defined, it will throw the *checkExpressionNotDefinedError* exception.

[REQ14]
1) *load* The requirement is implemented.
2) For loading a defined SIMPLE program from a file, it will load the defined program from path and name it as programName. The first step is to read the corresponding file from the correct path, then save it into *expressionMap*. Then, to rename the program, change the *programName* from *expressionMap* to the new one.
Static method implemented in *InputOutput* class

```
public static expressionMap loadProgram(String path, String programName){
```

3)
Error condition 1:
- Similar to file storage of file, if the path information gets an error, the whole program will directly exit.

Error condition 2:
- If the file is empty, the whole program will also directly exit.

[REQ15]

1) *quit* The requirement is implemented.

2) Details:

For terminating the current execution of the interpreter, it is quite simple which is just needing to call "*System.exit(0)*" to exit the whole program.

```
if (source.equals("quit")) {
    System.exit( status: 0);
}
```

3) Errors:

Error condition 1:

- The main point to successfully quit the program is to correctly enter the command. If the user's typing gets some errors, they may not exit.

[BON1]

1) The requirement is not implemented.

[BON2]

1) The requirement is not implemented.