Group Project Report

Gabriel Conte (21094823D)
Jiang Guanlin (21093962D)
Wang Yukai (21094692d)
Ossian Bergström (22015056X)

COMP2021 Object-Oriented Programming (Fall 2022)

Contents

1 Introduction

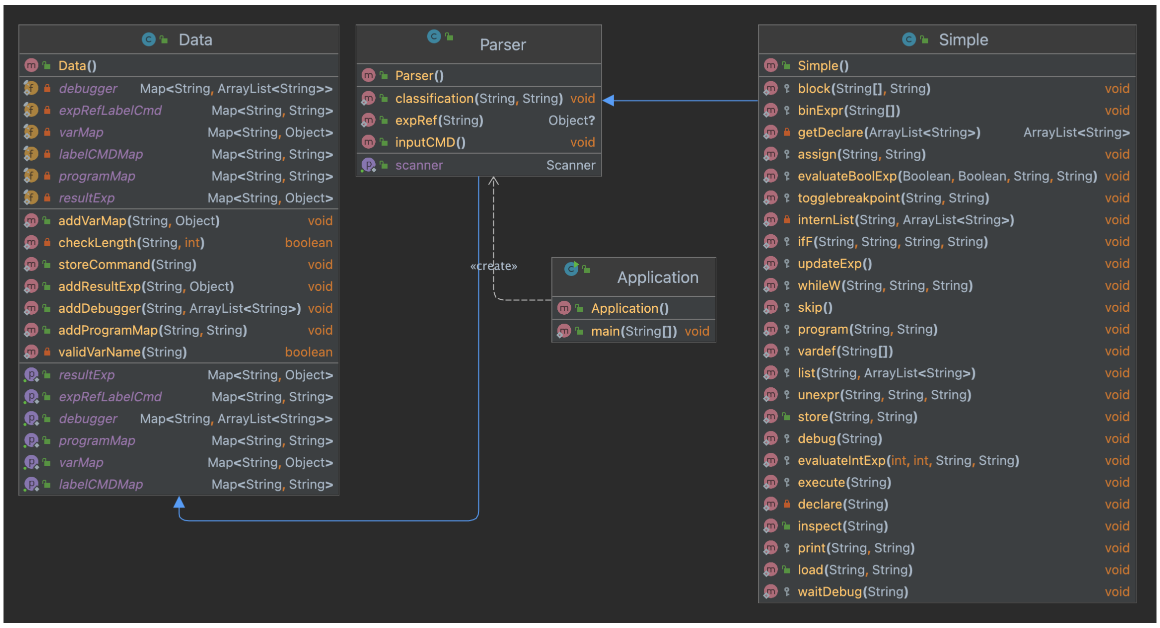
In society, the majority of the population are increasingly becoming more integrated with computer-based utilities, such as mobile phones. It is easy to take the technology behind it for granted, but creating an application and software programs used by these computers demands complex and robust programming languages. A programming language is usually contained by an interpreter and a compiler. The interpreter immediately executes the given instructions written and the compiler translates the high-level programming code to a low-level one, which can be interpreted by the corresponding hardware of the computer. In this project, the main goal is to understand the workload of an interpreter and what it generally does, by developing a command-line based interpreter that handles basic programs written in the *SIMPLE* programming language.

2 A Command-Line-Based Interpreter for Simple Programs

2.1 Design

2.1.1 UML Diagram

As you can see, this is the UML class diagram for Interpreter for Simple Programs.



Data
Data()
debugger Map<String, ArrayList<String>>
expRefLabelCmd Map<String, String>
varMap Map<String, Object>
labelCMDMap Map<String, String>
programMap Map<String, String>
resultExp Map<String, Object>
addVarMap(String, Object) void
checkLength(String, int) boolean
storeCommand(String) void
addResultExp(String, Object) void
addDebugger(String, ArrayList<String>) void
addProgramMap(String, String) void
validVarName(String) boolean
resultExp Map<String, Object>
expRefLabelCmd Map<String, String>
debugger Map<String, ArrayList<String>>
programMap Map<String, String>
varMap Map<String, Object>
labelCMDMap Map<String, String>
Parser
Parser()
classification(String, String) void
expRef(String) Object?
inputCMD() void
scanner Scanner
«create»
Application
Application()
main(String[]) void
Simple
Simple()
block(String[], String) void
binExpr(String[]) void
getDeclare(ArrayList<String>) ArrayList<String>
assign(String, String) void
evaluateBoolExp(Boolean, Boolean, String, String) void
togglebreakpoint(String, String) void
internList(String, ArrayList<String>) void
ifF(String, String, String, String) void
updateExp() void
whileW(String, String, String) void
skip() void
program(String, String) void
vardef(String[]) void
list(String, ArrayList<String>) void
unexpr(String, String, String) void
store(String, String) void
debug(String) void
evaluateIntExp(int, int, String, String) void
execute(String) void
declare(String) void
inspect(String) void
print(String, String) void
load(String, String) void
waitDebug(String) void

Throughout the design process, we followed some structures:
- A clear structures defined by the documentation and our mind
- Top-to-bottom implementation
- MVC Framework
- Programming in Object-Oriented
- Data Formatting Structures

2.1.2 Data Structures

A. Array

Arrays are used to store the user's command into separate words. The input command will be presented in a string format, which will be splitted into an array and separate each keyword of a instruction.

B. ArrayList

ArrayLists are primarily used for storing the list of instructions a program contains. Since arrayLists are dynamic, they are more useful since programs can have different sizes. Additionally, arrayLists are used for debugging where they store the commands that have a breakpoint.

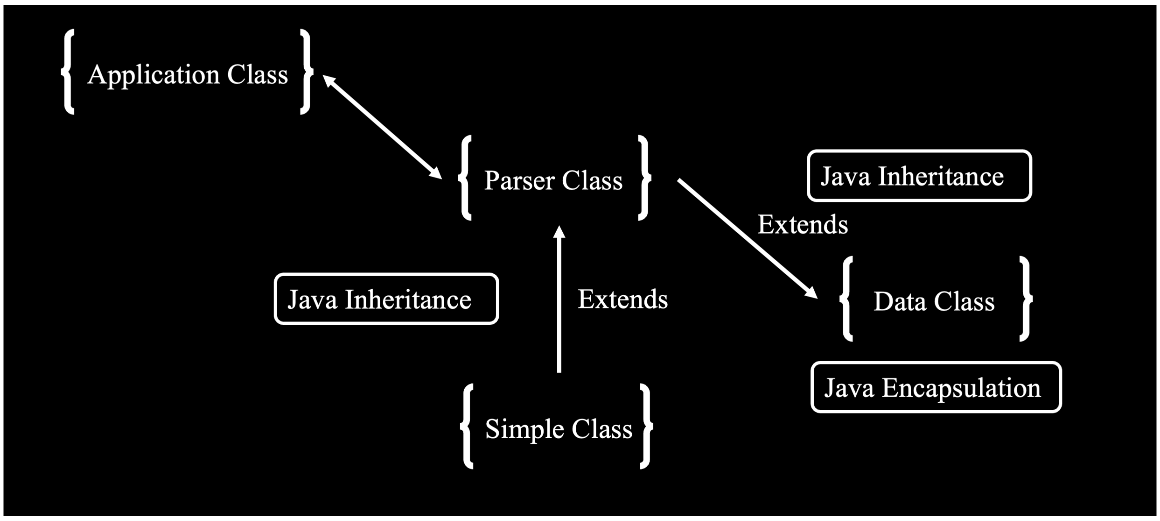
C. HashMap

The chosen method of structuring data is by the usage of *HasMaps*. The data and its corresponding data-types that are being stored are the following:

- expRefLabelCmd $< String, String >$ - Stores the command of expression references and its labels.
- labelCMDMap $< String, String >$ - Labels and its corresponding commands.
- varMap $< String, Object >$ - Variables of type Integer value or Boolean value.
- resultExp $< String, Object >$ - The result of expressions given by objects.
- breakPointMap $< String, String >$ - Stores the programName and the label of breakpoint.
- programMap $< String, String >$ - To store the program label and label in program.
- debugger $< String, ArrayList < String >>$ - Stores the programName and the debug command list.

2.1.3 Object-Oriented Structures

Here is our Object-Oriented and program structures diagram



A. Java Encapsulation

Data encapsulation is used to avoid improper accesses to the data structures in this application. All data structures in the data class are private so they can just be accessed and modified by the get and set methods created to retrieve and add to the specific data structure.

B. Java Inheritance

In this project inheritance is used to reduce redundancy in code, which makes the structure of the program more clear and readable. Additionally, since the super classes just have static fields, the whole program environment uses its parent fields for its own functionality. Inheritance was also used when needing to store and retrieve data objects of sub-classes of the object class. Since simple language just accepts integers and boolean variables, using its superclass Object was helpful for producing the code for the general case.

2.1.4 MVC Framework

In this program, we split to three parts to write, which are models, view, and controller. The Simple class and Data class are the project models, and Parser class in this project are controller and view, which will be create the user interface for user to use, and the connection bridge between the models and view, the application file just for initial whole program to start. Also, Data Class in here will be offer the data setting function and data getting function.

2.2 Requirements

2.2.1 Functions within the Parser class

The following functions are used written within the *Parser* class.

StoreCommand function The *storeCommand* function handles the data storage of the Simple Program. It takes the entire command-line as a *String* and splits it up to a *String Array.* The function then stores the inserted command-line to the corresponding *HashMap*, depending on the content of the first element. It then makes a function call to the *Classification* function with the command-line as the input argument, as a *String* type.

Classification function The *Classification* function distributes the inserted command-line in the Simple program and does the corresponding function call to the Simple class. It takes in the command-line as an entire *String* and split the *String* to an *String Array.* Depending on the first element of the array, which is the function name, a switch statement is used in the distribution. This function has the argument *programName* which is used to define if the execution of the program is to be done in execute or debug mode.

expRef function The *expRef* function takes in an expression as its sole input argument. The goal of the function is to evaluate what kind of expression the input argument is and to return the corresponding value of the expression. It does so by evaluating whether the expression is a literal expression (*True*, *False*, or an integer), a variable name or an expression name. It evaluates it with the help of *if* statements and checks if the expression is stored in any of the corresponding *HashMaps.*

internList function The *internList* function takes a program first command and create a list of unique commands that compose the program. In this way, the program will have the composition of a Simple language program.

getDeclare function This function gets the functions that declare variables or expressions in the program.

declare function Executed before the execute function, creates the variables and expressions of a program.

Command Validation for Input The Command Validation for input function is located in the *Simple* class but add into those basic functions, which is more effective to check the commands satisfy the basic rule of the simple programming language. The constraints for the input command are the following.

- The Simple programming language should only support data types of *int* and *bool.* Values of type *bool* only includes *false* or *true* and the values of type *int* are only within the range of $-99999 <$ Integer < 99999.

- Only a total of 15 different operators are allowed in the Simple programming language, namely $+, -, *, /, \sharp, \sim, >, >=, <, <=, ==, != , \&\&, ||, !$. Additionally, to store a program to a *txt* file, the : symbol is also included in the allowed, but not as an operator.

- The Simple programming language only allows variables and expression names to be written with only English letters and digits.

- The Simple programming language only allows a certain amount of command types to be utilized. These commands are the ones included in the project requirements. Furthermore, if a new command is created, the name of that command is not allowed to equal to any of the names of the command types in the Simple programming language.

- The Simple programming language checks the length of each word in the input, not allowing any words with more characters than 8. An exception is used for the *store* and *load* command for the file-path name, which usually contains a greater number of characters.

- The Simple programming language does not allow empty strings as inputs to the program.

In this validation, we will be handle all the input error in here, like user input error, the int value is not in the range, operators error.

2.2.2 Vardef Command

The requirement is implemented.

We use the HashMap *varMap<String, Object>* to store variables value. The variable name is used as the map key and the corresponding to an Object (either integer or boolean for Simple language) value as the map value. A call to this function in the command-line is illustrated below.

vardef vardef1 int x 100

2.2.3 Binexpr Command

The requirement is implemented.

The *Binexpr* function defines a binary expression with a chosen name and does the desired computations according to the inserted input command-line between two given expressions. The function uses a *switch* statement to distinguish which operator is chosen and executes the corresponding calculation depending on the operator given. The function uses two different functions to evaluate if the computation is done between two boolean values or two integer values, namely *evaluateIntExp* and *evaluateBoolExp.* Depending on the data type, a certain set of operators are allowed to be utilized. Afterwards, the result of computation is stored in a *HashMap* named *resultExp.* A call to this function in the command-line is illustrated below.

binexpr exp1 x $*$ 20

2.2.4 Unexpr Command

The requirement is implemented.

The *Unexpr* function takes in one expression and one unary operator (! or $\sim$) together with the name of the new expression, as its input arguments. Depending on the given operator, the function will negate the expression and store it in the *resultExp HashMap* as a new expression and name. The ! operator negates a boolean expression and negates an integer expression. A call to this function in the command-line is illustrated below.

unexpr 20 $\sim$ 10

2.2.5 Assign Command

The requirement is implemented.

The *Assign* function assigns a value from an already existing variable from the *HashMap*, named *varMap*, and stores the value in the new variable with the name given as an input argument. It then updates the stored statements in the *HashMap expRefLabelCmd* that contains this variable, by calling the function *updateExp*. A call to this function in the command-line is illustrated below.

```
assign assign1 x exp2
```

2.2.6 Print Command

The requirement is implemented.

This function creates a new print statement by taking the label for that statement as an input argument and an expression reference. The function then attains the value of the inserted expression reference and stores it in the *addResultExp HashMap* with the map key name as the new print statement label and the value of the expression reference as the map value. The desired value is then displayed in the terminal within two hard brackets ([*value*]). A call to this function in the command-line is illustrated below.

```
print print1 exp2
```

2.2.7 Skip Command

The requirement is implemented.

A call to this function in the command-line will create an statement that does nothing, basically the way a Simple program is defined, is used to do nothing in an if-else clause. A call to this function in the command-line is illustrated below.

```
skip skip1
```

2.2.8 Block Command

The requirement is implemented.

The *Block* function takes in a *String* array of instructions containing labels of statements to execute. These instructions are stored into an array and will be executed accordingly. This function implements recursion to check if there is another block or while during its call to also execute the commands inside that block. The *Block* function then checks if the map value of the corresponding label is contained by the *HashMaps labelCMDMap* or *exRefLabelCMDMap* and makes a function call to the classification function to execute the corresponding function. A call to this function in the command-line is illustrated below.

```
block block1 assign1 print1
```

2.2.9 If Command

The requirement is implemented.

The *if* function takes in a condition and two statements as input arguments. It checks if the given condition is *TRUE* or *FALSE* and if it is *TRUE*, the function fetches the string command for executing the first function and if its *FALSE*, the function fetches the string command for the second function. The resulting function is then executed. The execution is done by a function call to the classification function in the *Parser* class, which is previously explained. A call to this function in the command-line is illustrated below.

```
if if1 exp5 block1 print1
```

2.2.10 While Command

The requirement is implemented.

The *While* function takes in a condition and one statement as input arguments. It then checks the condition whether it is *TRUE* or *FALSE.* If the condition is *TRUE*, the algorithm will continuously be executing the chosen function until the condition turns *FALSE.* A call to this function in the command-line is illustrated below.

```
while while1 true block1
```

2.2.11 Program Command

The requirement is implemented.

The *Program* function takes the desired name for the new program in Simple to be created and the label of the statement that should be executed when the new program is executed, as input arguments. The newly created program is then stored in the *programMap HashMap* with the name of the new program as the map key and the label of the chosen statement as the map value. A call to this function in the command-line is illustrated below.

```
program program1 while1
```

2.2.12 Execute Command

The requirement is implemented.

The *Execute* function takes in a program as its sole input argument and executes the chosen program. This functions first gets the list of unique instructions using function internList. This is done by extracting the map value from the map key, which is the chosen program name to be executed, from the *programMap HashMap.* When the label is attained, the corresponding command to execute the program is fetched from the *labelCMDMap HashMap* and a function call to the classification function from the *Parser* class is done to execute the chosen program. A call to this function in the command-line is illustrated below.

```
execute program1
```

2.2.13 List Command

The requirement is implemented.

The *List* function is a complimentary function to the *Program* function, by printing out a list of the commands within that specific program that is inserted as an input argument. In this function, we set a new function internList function to get instruction from the HashMap, and printout those commands in that program. A call to this function in the command-line is illustrated below.

list program1

2.2.14 Store Command

The requirement is implemented.

In this project, the *HashMap* help us to store the program code temporally, so we create a file which named by the Program Command to store this program to file. This will create a text file that contains the commands of the stored program in the designated path.

store program1 <File Path>.<Name of txt file to store the program within>

2.2.15 Load Command

The requirement is implemented.

The load command will read a previously stored Simple program and will run its instructions calling the execute function.

load <File Path>.<Name of txt file to store the program within> program1

2.2.16 Quit Command

The requirement is implemented.

When the user inputs a new command-line to the Simple program, the command-line is analysed in the *inputCMD* fuction in the *Parser* class. If the input string equals the *String* "quit", the Simple program will terminate. A call to this function in the command-line is illustrated below.

quit

2.2.17 Debug Command

The requirement is implemented.

The debug command executes the program but stops its execution when there is a breakpoint in certain function. While the program stops, the user can declare/remove breakpoints, see the value of certain variables within the program and continue its execution

debug program1

2.2.18 Togglebreakpoint Command

The requirement is implemented.

The *Togglebreakpoint* command is inserted in the Simple program command-line, the *breakPointMap HashMap* will then catch the breakpoint label and store it. When the user inputs the same breakpoint label together with the same program name, the breakpoint of that specific program will remove it from the *HashMap*. This function's functionalities are correlated with the *Debug* function.

Togglebreakpoint program1 block1

2.2.19 Inspect Command

The requirement is implemented.

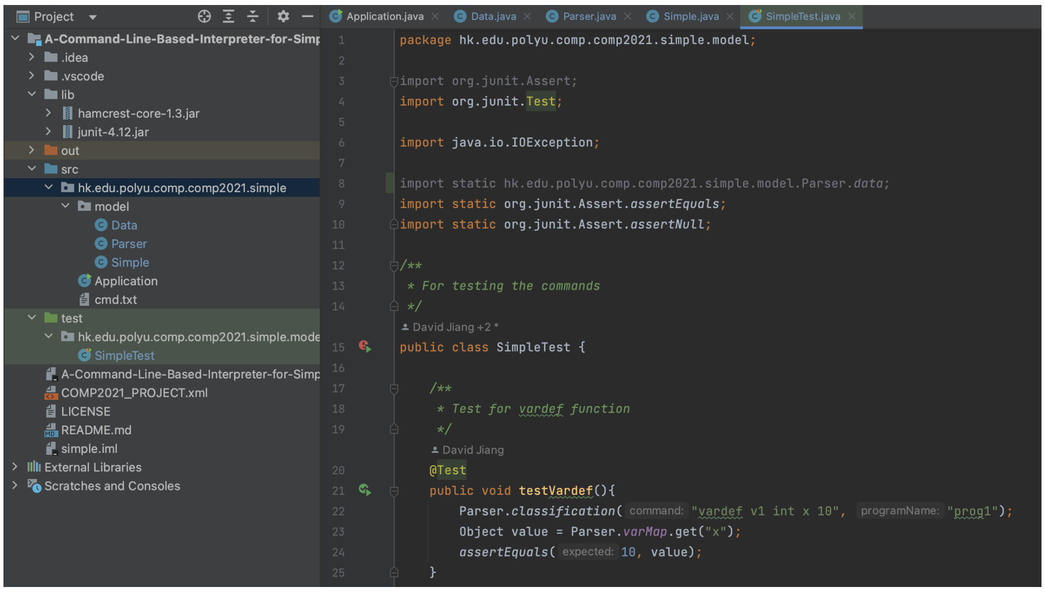
The *Inspect* function allows the user, during debugging, to input the name of a program and that the user is debugging and the corresponding variable within that program. The function will thereafter print out the value of that specific variable.

inspect program1 x

3 Testing

3.1 Java Unit Test

Our Java Unit Test is implemented:



```java
package hk.edu.polyu.comp.comp2021.simple.model;

import org.junit.Assert;
import org.junit.Test;

import java.io.IOException;

import static hk.edu.polyu.comp.comp2021.simple.model.Parser.data;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNull;

/**
 * For testing the commands
 */
public class SimpleTest {

    /**
     * Test for vardef function
     */
    @Test
    public void testVardef(){
        Parser.classification( command: "vardef v1 int x 10",  programName: "prog1");
        Object value = Parser.varMap.get("x");
        assertEquals( expected: 10, value);
    }
```

3.1.1 Test Vardef

In this unitTest, we test the Vardef Command by using a classification function that is in Parser that designed by us and in the second step we use a command that is "vardef v1 int x 10" and set an object then use a HashMap that we name it 'varmap' to get the value of x ,which is 10, and store it to the object. Finally, we use assertEquals to set up a value 10 and test if the object value that we get is equal to the value that we want. If it is equal, the test is successful.

3.1.2 Test BinExpr

In this unitTest, we test the Binexpr Commmand by using classification in Parser twice and we firstly using command to set a value, for example, 10, and secondly we use the command of BinExpr to see if this value is in the range or not. For instance, if the second command is "binexpr exp3 x ¿ 20". Since the value is 10, so the result that come out will be false, in this test will be 0, then we can use assertequals to set the expected value to 0 to see if this test is matched or not. If it is matched, then this test is successful.

3.1.3 Test UnExpr

In this unitTest, we test the Unary command is work our not, and the expression calculation result is right or not. We use example - "unexpr exp2   exp1" to test, and use the assertequals to catch the result and compare with the real result is true or false.

3.1.4 Test Assign

In this unitTest, we test Assign Command by using the classification function in Parser. At first, we write a commmand and we make it int and the name will be x and the value will be 10. After that, we use assign function in simple and make the value become 15 from 10 in the varmap. Finally, we use assertEquals to check if this value is 15 or not. If it is 15, then this test is successful.

3.1.5 Test Print

In this unitTest, we test Print Command by using four commands from classification of Parser. In the test, we get the value by using resultExp that we designed previously. Then we get assertEquals to make the value and the expected value matched.If they did match, then the test is successful.

3.1.6 Test skip

In this unitTest, we simply use assertNull since this function is just for skipping so it returns null.

3.1.7 Test List

In this unitTest, we will be compare the list printout and the command we input for a program is same or not.

3.1.8 Test File Operation

In this unitTest, we will test two methods - store and load commands. We will be store the program first, after, we load the program, if the program load is same as the commands that we input for that program, the test will be pass. Also, we test the error of the file operation, like user input error file path, so the Exception will be give to user.

3.1.9 Test Execute

In this unitTest, we will compare the result after the program execute, and the result we already know. If the output is same, the tes will be pass.

3.1.10 Test BreakPoint

In this unitTest, we will call the classifier, and after to get the first value from the HashMap get function - getDebugger() to test is right or not.

3.1.11 Test Debug

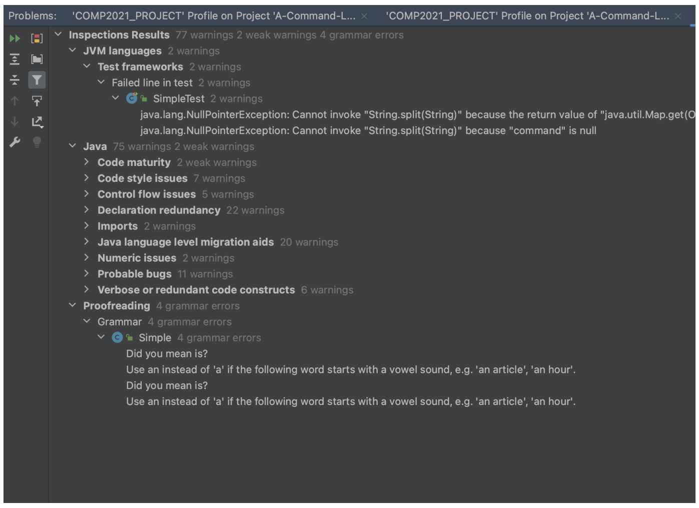
In this unitTest, we will call the classifier, and the test it will give the error or not, if not given error, and the value also right, it will be pass.

3.1.12 Test Inspect

In this unitTest, it will be compare with the value that we given to see it is right or not.

3.2 Rules in Code Inspection

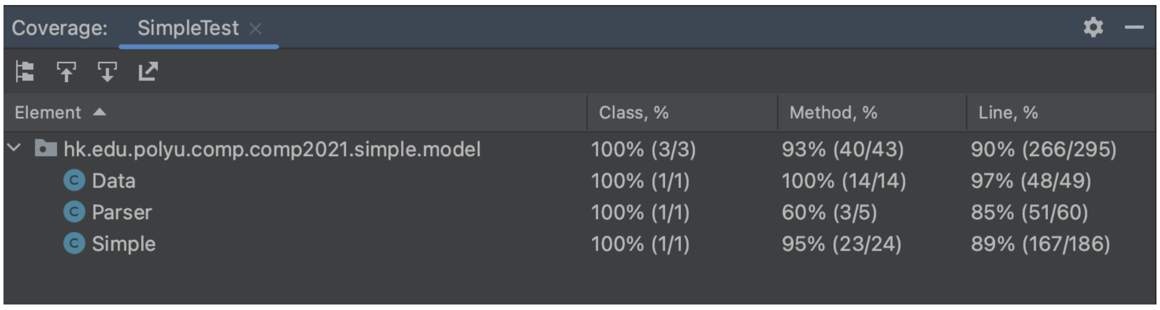
Here is the rules to be used in code inspection, we pass the all, no error in here:



```
Problems:   'COMP2021_PROJECT' Profile on Project 'A-Command-L...   'COMP2021_PROJECT' Profile on Project 'A-Command-L...
Inspections Results 77 warnings 2 weak warnings 4 grammar errors
  JVM languages 2 warnings
    Test frameworks 2 warnings
      Failed line in test 2 warnings
        SimpleTest 2 warnings
          java.lang.NullPointerException: Cannot invoke "String.split(String)" because the return value of "java.util.Map.get(O
          java.lang.NullPointerException: Cannot invoke "String.split(String)" because "command" is null
  Java 75 warnings 2 weak warnings
    Code maturity 2 weak warnings
    Code style issues 7 warnings
    Control flow issues 5 warnings
    Declaration redundancy 22 warnings
    Imports 2 warnings
    Java language level migration aids 20 warnings
    Numeric issues 2 warnings
    Probable bugs 11 warnings
    Verbose or redundant code constructs 6 warnings
  Proofreading 4 grammar errors
    Grammar 4 grammar errors
      Simple 4 grammar errors
        Did you mean is?
        Use an instead of 'a' if the following word starts with a vowel sound, e.g. 'an article', 'an hour'.
        Did you mean is?
        Use an instead of 'a' if the following word starts with a vowel sound, e.g. 'an article', 'an hour'.
```

3.3 Code Coverage

Here is our project code coverage, which is 90% of the line coverage:


Coverage: SimpleTest

Element	Class, %	Method, %	Line, %
hk.edu.polyu.comp.comp2021.simple.model	100% (3/3)	93% (40/43)	90% (266/295)
Data	100% (1/1)	100% (14/14)	97% (48/49)
Parser	100% (1/1)	60% (3/5)	85% (51/60)
Simple	100% (1/1)	95% (23/24)	89% (167/186)

4 User Manual

4.1 Start up the program

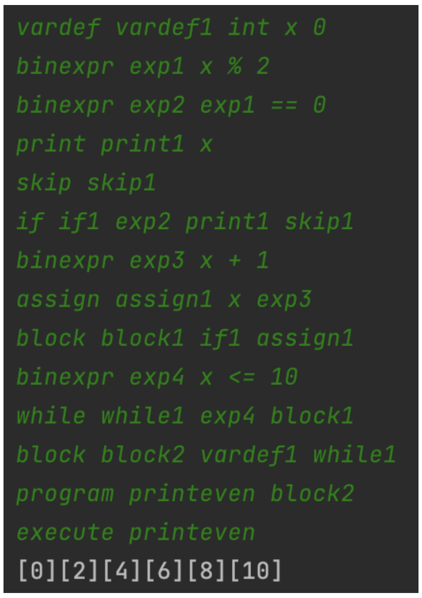
To initialize the Simple program the end-user is required to enter a valid command-line in the terminal after running the *Application* class. The Simple program only allows a correctly written input command-line otherwise a message will be printed out to inform the end-user that the previous command-line was incorrect and incomprehensible for the Simple program to interpret and run. Therefore, if a correctly inserted command-line is written, the program will execute the corresponding instruction given. If the end-user desire to terminate the program, a simple command-line containing the sole word *quit* will close it down. The following are examples of how the command-line for each command can be inserted to the Simple program.

```
vardef vardef1 int x 0
binexpr exp1 x % 2
binexpr exp2 exp1 == 0
print print1 x
skip skip1
if if1 exp2 print1 skip1
binexpr exp3 x + 1
assign assign1 x exp3
block block1 if1 assign1
binexpr exp4 x <= 10
while while1 exp4 block1
block block2 vardef1 while1
program printeven block2
execute printeven
list printeven
store program1 /Users/davidjiang/Desktop/prog1.simple
load /Users/davidjiang/Desktop/prog1.simple program1
togglebreakpoint printeven if1
debug printeven
inspect printeven x
debug printeven
inspect printeven x
togglebreakpoint printeven if1
debug printeven
quit
```

For a more in-depths description of the functionality for each and every command, check section 2.2 of this project report.

For Execute Printout:

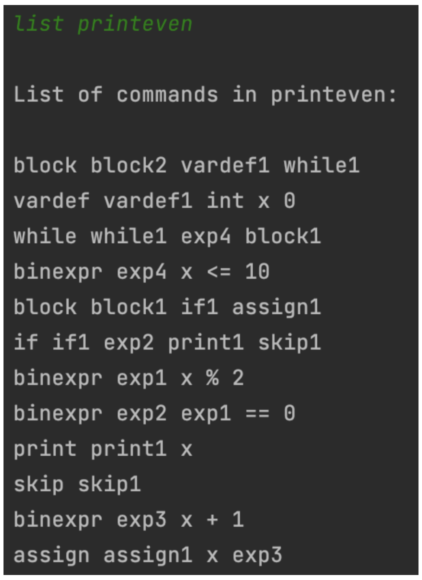
When user input the execute function, the result will be print in the line if this program include the print function. If not, the program will be just running, no result will be printout.



```
vardef vardef1 int x 0
binexpr exp1 x % 2
binexpr exp2 exp1 == 0
print print1 x
skip skip1
if if1 exp2 print1 skip1
binexpr exp3 x + 1
assign assign1 x exp3
block block1 if1 assign1
binexpr exp4 x <= 10
while while1 exp4 block1
block block2 vardef1 while1
program printeven block2
execute printeven
[0][2][4][6][8][10]
```

For List Printout:

Here is the list command that printout the whole program commands:



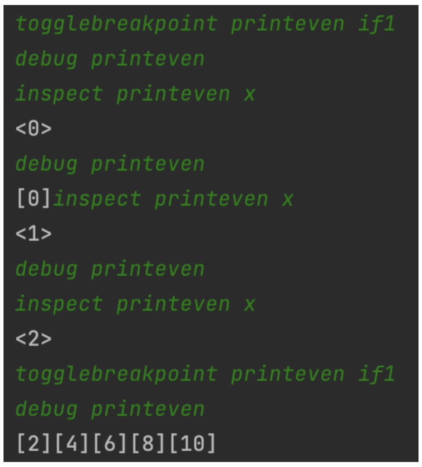
```
list printeven

List of commands in printeven:

block block2 vardef1 while1
vardef vardef1 int x 0
while while1 exp4 block1
binexpr exp4 x <= 10
block block1 if1 assign1
if if1 exp2 print1 skip1
binexpr exp1 x % 2
binexpr exp2 exp1 == 0
print print1 x
skip skip1
binexpr exp3 x + 1
assign assign1 x exp3
```

For Debug & Inspect Printout:

Here is the output in debug mode and the output by inspect the value in debug mode:



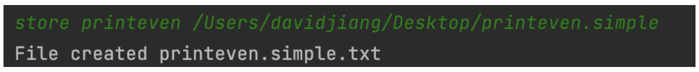
```
togglebreakpoint printeven if1
debug printeven
inspect printeven x
<0>
debug printeven
[0]inspect printeven x
<1>
debug printeven
inspect printeven x
<2>
togglebreakpoint printeven if1
debug printeven
[2][4][6][8][10]
```

4.2 Program File Operation

If the user wants to store a created program to on a later occasion be able to run the same program, the usage of the store and load function is recommended. A side-note in using those functions, the entire file path must be entered and correctly so. If the path is incorrectly written or if the path itself does not exist in the file system, an error will printed out warning the end-user. On the contrary, if the file already exists, a message will be printed out notifying the user that there already exist identical program. The file will be replaced by the new code either way.

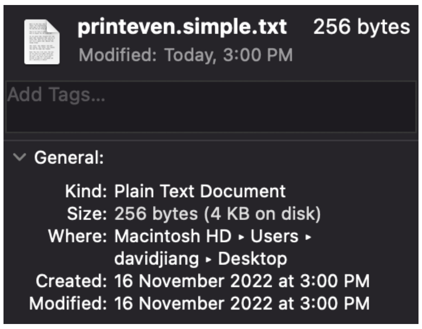
For Program Creation:

Here is the output that tell user the program already save:



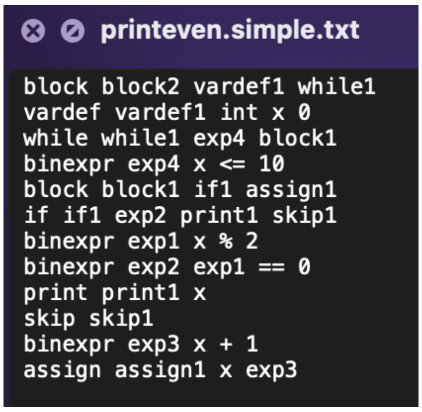
```
store printeven /Users/davidjiang/Desktop/printeven.simple
File created printeven.simple.txt
```

Here is the program file generate by store command:



```
printeven.simple.txt   256 bytes
Modified: Today, 3:00 PM
Add Tags...
General:
    Kind: Plain Text Document
    Size: 256 bytes (4 KB on disk)
   Where: Macintosh HD ▸ Users ▸
          davidjiang ▸ Desktop
 Created: 16 November 2022 at 3:00 PM
Modified: 16 November 2022 at 3:00 PM
```

Here is the commands in program file:



```
printeven.simple.txt
block block2 vardef1 while1
vardef vardef1 int x 0
while while1 exp4 block1
binexpr exp4 x <= 10
block block1 if1 assign1
if if1 exp2 print1 skip1
binexpr exp1 x % 2
binexpr exp2 exp1 == 0
print print1 x
skip skip1
binexpr exp3 x + 1
assign assign1 x exp3
```

End of Object-oriented Programming Group Project Report
COMP2021 Group 1