A Compiler for the Verification of a Java-Like Programming Language: User Manual

o   Member:
  ▪   Name: Liu, Monica
  ▪   Student Number: 212 891 644
  ▪   EECS Prism login name: monica01
o   Member:
  ▪   Name: Lai, Chin Yee
  ▪   Student Number: 217 670 993
  ▪   EECS Prism login name: chinyee8
o   Member:
  ▪   Name: Han, Yeseul
  ▪   Student Number: 217847146
  ▪   EECS Prism login name: hany72

Table of Contents

1   Input Languages

1.1   How to Run the Program

- Type in: my-compilers.jar inputProgram.txt testProgram.txt
- There are 2 file prompts
- The first file "inputProgram.txt" is the one where you write your input program in
- The second file "testProgram.txt" is the one where you write your input test in

1.2   Structure of the Program File

```
game Mathclass !

  mymethod MATH DOUBLE [DOUBLE numerator, DOUBLE denominator] !
        #Section 1: Variable Declarations
        input1 << INT
        checker << BOOLEAN
        result << DOUBLE

        #Section 2: Variable Assignments
        input1 <- 2 + 2
        checker <- TRUE
        result <- 1.00

        #Section 3: If Statements

        jackieAsks [checker] !
              result <- numerator

        !
        elseJackie !
              jackieAsks [not checker] !
                    result <- 1.00

              !
              elseJackie !
                    x << INT
                    x <- 10
                    loop (3) !
                          x <- x + 10
                    !
              !
        !
```

```
        #Section 4:  Loops
        loop(3) !
                 result <- INCREMENT[result]
         !

   #Section 5: Return Statement
    jackieReturns result
  !

  mymethod INCREMENT DOUBLE [DOUBLE d] !
        result << DOUBLE
        result <- d + 1.00
        jackieReturns result
  !
 !
```

The program file contains a list of 0 or more methods
The layout of each method body is as follows:
Section 1: Declarations of variables
Section 2: Assignments to variables
Section 3: If Statements
Section 4: Loops
Section 5: Return statement
Comments can be made anywhere in this file with '#' to indicate it
Note: layout must be in this order, i.e. declarations cannot be made after assignments and loops/if -statements cannot be
made before declarations while loop must be made after if statement

1.2.1   Class Declaration and Method Declaration

To start a class declaration, the user must use the keyword 'game' followed by class name
and '!' to specify the opening of the class body and '!' to end the class body.

To declare a method, the user must use keyword 'mymethod' followed by method name
and keyword of **return type** this method expects to return to the method call, followed by '!'
to indicate the beginning of method body, and '!' to indicate the end of the method body.

Return types: STRING, CHAR, BOOLEAN, INT, DOUBLE

1.2.2   Variable Declarations – Section 1

Variable Declarations

The first section of variable declarations allows you to assign data type to each variable
name, the supported data type keywords are: INT, DOUBLE, BOOLEAN, CHAR, STRING
The symbol to declare a variable is '<<'

Variable Naming

Variable names must be in all lowercase, the first letter must be from a-z, any letters after that may be any combination of a-z and 0-9 and '_' underscore. Regular expression as follows: [a-z][a-z0-9_]*

When a variable is declared, it will have default values as follows:

DATA TYPE	DEFAULT VALUE
STRING	''
CHAR	'\u0000'
BOOLEAN	FALSE
DOUBLE	0
INT	0

1.2.3   Variable Assignments – Section 2

Variable Assignments

The assignment symbol is: '<-'

The second section is variable assignments. String assignments must be surrounded by "" and char assignments must be surrounded by ''. The user can assign it values, do mathematical expressions, and method calls. Some examples are below:

STRING	"da234ABCkja"
CHAR	'x' 'A'
BOOLEAN	TRUE FALSE
DOUBLE	7623.00 12.73
INT	10

Note 1: double types need to have 2 decimal places
Note 2: string types allow alphabets, numbers and space only.

Table For Mathematical Expressions:

Parenthesis	(math)
Addition	math + math
Subtraction	math - math
Multiplication	math * math
Division	math / math

Note 3: each math can be replaced by an integer, a double, or a variable name containing any one of those types. The same types must be used for each expression (double works only with double, integer only works with integer).

Note 4: each assignment must be type correct, otherwise it prints a semantic error

Note 5: mathematical expressions may be compounded (i.e. 8 + 9 – (1-4)*9/3)

1.2.4   Variable Assignments with Method Calls – Section 2

Variable Assignments With Method Calls

Method calls can be used with assignments. The method call name and parameter types must match an existing method in the program file (second file prompt) otherwise it will show semantic error. When a method call is made to be assigned to a variable, the arguments need to be surrounded by square brackets and separated by commas (i.e. variablename <- METHODNAME[arg1, arg2])

Method calls cannot be combined with any other expressions in one assignment (i.e. variablename <- METHODNAME[arg1] + 1) this example is not supported. We decided to not support this feature to make sure the programmer can see its return result before doing further computation.

Note 5: Declarations need to be made in the first section, cannot be combined with assignments. All declarations are made first, then assignments can be made, then if-statements then loops can be made. The same order of operation is needed inside if-statements and loops as well.

Example of what is **NOT supported:**

```
game Mathclass !

   mymethod INCREMENT DOUBLE [DOUBLE d] !
         result << DOUBLE
         result <- d + 1

         newvariable << INT #Error: Declaration not allowed after any assignment

         jackieReturns result
   !
!
```

1.2.5   If-Statements and Loops – Section 3

If-Statements

If statements need to start with **'jackieAsks'** along with square brackets around its condition i.e. '[' conditions ']' and '!' to start the if statement body and '!' to end it. For every if-statement, **'elseJackie'** needs to come after it. Similarly, 'elseJackie' needs to be followed by '!' at the beginning of the body, and '!' to end it. Structure of the body of **'jackieAsks/elseJackie'** is similar to the method body: declarations, assignments, if statements, then loops. Any of these statements can be made 0 or more times.

Variable Scoping is similar to Java. The scope of each variable is within each '!' it's surrounded by. The inner of a loop and if-statements can call the variables from its outer body and not vice-versa.

```
jackieAsks [checker] !
      result <- numerator
!
elseJackie !
      jackieAsks [not checker] !
            result <- 1
      !
      elseJackie !
            x << INT
            x <- 10
            jackieAsks [TRUE] !
                  name << MONICA
                  loop (3) !
                        x <- x + 10
                  !
            !
            elseJackie !
            !
            loop (100) !
            !
      !
!
```

Figure 1: Structure of If-Statements inside Program

Loops

Deterministic loops can perform multiple iterations on the loop body with the following syntax: **'loop'** with opening and closing brackets '(10 ')' around a raw integer (does not support variable or any type of expressions) and must be followed by '!' to indicate beginning of loop body and '!' to indicate end of loop body. For more details: read advanced features on loop. Structure of the body inside the loop is similar to an if-statements body.

```
loop(3) !
      result <- INCREMENT[result]

!
```

1.3   List of Advanced Programming Features

1.3.1   Feature 1: Variable scoping

In our language variable scoping works on the scope where you declare it. Each method has its own set of variables and does not share it outside, the only way to retrieve information is through a return statement. This helps with the **security** of our language.

From this example, our method is called EXAMPLE, and inside it has 're', 'temp', 'temp2', 'temp3'. These variables only exist within this method and we use jackieReturns to return the value of variable 're'. The variable 'values' is defined in the third jackieAsks so it exists only in that jackieAsks.

```
mymethod EXAMPLE BOOLEAN [INT num] !
        re << BOOLEAN
        temp << BOOLEAN
        temp <- TRUE
        temp2 <- TRUE
        temp3 <- TRUE

        jackieAsks [temp] !
                jackieAsks [temp2] !
                        jackieAsks [temp3] !
                                values << INT
                                values <- values + 1
                                re <- TRUE
                        !
                        elseJackie!
                                re <- FALSE
                        !
                !
                elseJackie!
                        re <- TRUE
                !

        !
        elseJackie!
                re <- FALSE
        !

        jackieReturns re

    !
```

1.3.2 Feature 2: If-else

We also have an equivalent to If-Else in JAVA which is called jackieAsks – elseJackie, named after the Jackie. However, our jackieAsks – elseJackie has a different syntax than JAVA. The body of each jackieAsks / elseJackie need to follow this syntax:
- Variable declaration
- Assignments/method calls
- If-Statements
- Loops

This is to help with readability, all declarations must be on top. No declaration is allowed after assignments and method calls.

All jackieAsks need to have a corresponding elseJackie to maintain completeness.

In jackieAsks [temp], temp can be any statement that can be evaluated to TRUE/FALSE.

jackieAsks – elseJackie, support nested, you can have others jackieAsks – elseJackie or loops.

```
mymethod TEST BOOLEAN [INT num] !
        re << BOOLEAN
        temp << BOOLEAN
        temp <- TRUE
        temp2 <- TRUE
        temp3 <- TRUE

        jackieAsks [1 + 2 > 1 && 4 < 5 || (FALSE && not FALSE) ] !
                jackieAsks [temp2] !
                        jackieAsks [temp3] !
                                values << INT
                                loop(10) !
                                     values <- values + 1
                                !
                        !
                        elseJackie!
                                re <- FALSE
                        !
                !
                elseJackie !
                        re <- TRUE
                !

        !
        elseJackie!
                re <- FALSE
        !

        jackieReturns re
```

1.3.3   Feature 3: Loops

This language also has loops, the syntax of a loop is as follows:

loop(num) !
#body
!

Where num is an integer and cannot be a variable, this is a limitation of our language right now. The body of our loops follow the same structure as an if-statement body and main method body which is:

- Variable declaration
- Assignments/method calls
- If-Statements
- Loops

Similarly, the choice was made to help with readability.

Like in if-statements, loops also have its own variable scoping.

Loops will repeat the commands inside its body the number of times specified. Unlike **jackieAsks** and method calls, loops use '(' ')' to surround its number of iterations.

Loops also support nested structures that contain more loops and **jackieAsks**-statements.

```
loop(10) !
  values <- values + 1
!
```

1.3.4   Feature 4: Comments

This language also includes comments for better readability. The syntax of a comments is:
#write something here. Anything is fine. There is no restriction on comments.

1.4   Structure of Test File (here you make method calls to the Program)

To create a test program: the title needs to start with keywords '**test**' and test name needs to start with keyword '**test_**' and followed by any combination of alphabets' characters, numbers and underscore: [a-zA-Z0-9_]* (i.e. **test test_**abc_123)

It needs to have an opening '!' after the test name and closing '!' at the end of the test case.

```
test test_case !
    #Section: Variable Declarations
    input1 << DOUBLE
    input2 << DOUBLE
    result << DOUBLE
    name << STRING
    x << CHAR

    #Section: Variable Assignments
    input1 <- 2.50
    input2 <- 5.90
```

```
    name <- "TOM"
    x <- 'a'
    result <- MATH[input1, input2]

    x <- GETCHAR[x]
!
```

Declarations

The symbol to declare a variable is '**<<**'.

The first section of variable declarations allows you to assign data type to each variable name, the supported data type keywords are: **INT, DOUBLE, BOOLEAN, CHAR, STRING**

When a variable is declared, it will have default values as follows:

DATA TYPE	DEFAULT VALUE
STRING	''
CHAR	'\u0000'
BOOLEAN	FALSE
DOUBLE	0
INT	0

Assignments

The assignment symbol is: '**<-**'.

The second section is variable assignments. String assignments must be surrounded by double quotes "" and char assignments must be surrounded by single quotes ''. The user can assign it raw values, do mathematical expressions, and method calls. Some examples are below:

STRING	"da234ABCja"
CHAR	'x' 'A'
BOOLEAN	TRUE FALSE
DOUBLE	7623.00 12.73
INT	10

Note 1: double types need to have 2 decimal places
Note 2: string types only accepts alphabet, numbers and space

Table For Mathematical Expressions:

Parenthesis	(math)
Addition	math + math
Subtraction	math - math
Multiplication	math * math
Division	math / math

Note 3: each math can be replaced by an integer, a double, or a variable name containing either one of those types. The same types must be used for each expression (double works only with double, integer only works with integer).

Note 4: each assignment must be type correct, otherwise it prints a semantic error

Note 5: mathematical expressions may be compounded (i.e. 8 + 9 – (1-4)*9/3)

Assignments With Method Calls

Method calls can be used with assignments. The method call name and parameter types must match an existing method in the program file (second file prompt) otherwise it will show semantic error. When a method call is made to be assigned to a variable, the arguments need to be surrounded by square brackets and separated by commas (i.e. variablename <- MATHODNAME[arg1, arg2])

Method calls cannot be combined with any other expressions in one assignment (i.e. variablename <- METHODNAME[arg1] + 1) this example is not supported.

Note5: Declarations need to be made in the first section, cannot be combined with assignments. Assignments are made after declarations if there exists any.

1.5   How a Program and a Test Should be Distinguished

Input test needs to start with keyword 'test' and test name starting with 'test_'
Input program needs to start with keyword 'game'

Similarities between input program and input test are as follows:
- Declarations behave the same
- Assignments behave the same

Differences between the input program and input test are as follows:
- If-statements and loops are only supported in input program
- Return statement is only required in input program's methods

Flow Of Execution:

- Input 'test' starts the flow of execution (like a main class) (second file prompt)

- When assignments are made with method calls existing in input program 'game' (first file prompt), it will transfer execution flow to input program

- Input program executes the method being called, and calls other methods if it's made

- When the input program finishes its task of completing that method call, the execution flow will be transferred back to input 'test' and the return value gets stored in its variable, and the execution flow continues until the closing '!' for the end of the test body.

Body structures of '<u>test</u>' versus '<u>game</u>'

Input 'test' file	Input 'game' file
1. Section for Declarations 2. Section for Assignments	1. Section for Declarations 2. Section for Assignments 3. Section for If-Statements 4. Loops

Note: if-statements and loops have the same body structure as input 'game' file, withoud mymethods

2 Naming Rules

Note: none of these names can contain any of the keywords, refer to keywords table underneath: **Table 1 Keywords and Symbols Table**

- **Class names** in 'game' input : Must start with uppercase alphabet from [A-Z] and continue with any combination of uppercase alphabets [A-Z] and underscore '_'

  - Regular Expression: [A-Z][A-Z0-9_]*
  - Example:

---

game First_class !

    #gameclass body
!

- **Test names** in 'test' input: Must start with 'test_' and continue with any combination of alphabets of uppercase, lowercase, numbers and underscores.

  - Regular Expression: 'test_'[a-zA-Z0-9_]*

○ Example:

```
test test_hiHowAre10_You !

    #gameclass body
!
```

- **Variable names** must start with a lowercase alphabet in its first letter and continue with any combination of lowercase alphabet and underscore
  - Regular Expression: [a-z][a-z0-9_]*
  - Examples: 'variable_1', 'v2_integer', 'temp_temp_temp'

- **Method names** must start with a capitalized alphabet from [A-Z] and continue with any order of capitalized alphabet [A-Z] with a mixture of numbers and underscore
  - Regular Expression: [A-Z][A-Z0-9_]*
  - Examples:

```
mymethod M_E_T_H_0_DNAME INT [BOOLEAN b]!

    #method body
!
```

- **game**
- **mymethod**
- **test**
- **test_**
- **loop**
- **jackieAsks**
- **elseJackie**
- **jackieReturns**
- **|| , && , not, ==, /= , >=, <=, >, <, (,)**
- **+, - , *, /**
- **INT, DOUBLE, BOOLEAN, STRING, CHAR**
- **TRUE, FALSE**
- **<<, <-**
- **void_**
- **VOID**
- **"**
- **'**
- **#**

Table 1: Keywords and Symbols Table

Users must not name any variables, class names, method names etc. with any of the above key words and symbols. These keywords and symbols should only be used for its intended function.

STRING type syntax: must be surrounded by double quotes with any combination of lowercase, uppercase alphabet, numbers, and spaces. i.e. "hello 6 world"
   ● regular expression for this: '""[a-zA-Z0-9][a-zA-Z0-9]*""'

CHAR type syntax: must be surrounded by single quotes of a single character from the set of alphabets and numbers. i.e. 'x'
   ● regular expression for this: '\''[a-z]'\'' | '\''[A-Z]'\'''

3   Output Structure

First Page: Test Case

- The first page of the html generated should show the test case input. On the left hand side shows the input code in the test file with line numbers and method calls to assignments are highlighted in purple. Users can click on the purple highlighted boxes to take you to coverage results and display. On the right hand side of the test case page, it shows a list of all method calls used in test case, and when clicked on, it will display the computation result from executing method call.

Second Page: Input Program

- When clicked on the purple box method call from the test case, it will take you to the second page of your input program. The top will show the navigation bar which includes the list of coverages available. The left hand side displays your input program code.

Coverage Criteria expected: statement coverage, condition coverage, all-defs coverage, all-c-use coverage, all-p-use coverage

Game Class

Click button below for coverage ↓

| ← TestCase | Statement Coverage | Condition Coverage | All-Defs Coverage | All-C-Uses Coverage | All-P-Uses Coverage |

```
game Aclass !

mymethod MOD_K BOOLEAN [INT num, INT k] !

    re << BOOLEAN
    count << INT

    loop (10000) !
        jackieAsks [ num == 0 ] !
            re <- true
        ! elseJackie !
            jackieAsks [ num < 0 ] !
                re <- false
            ! elseJackie !
                jackieAsks [ num > 0 ] !
                    num <- num - k
                ! elseJackie !
                !
            !
        !
    !

    jackieReturns re
!
mymethod LEAPYEAR BOOLEAN [INT num] !

    re << BOOLEAN
    temp << BOOLEAN
    temp2 << BOOLEAN
    temp3 << BOOLEAN

    temp <- MOD_K [ num , 4 ]
    temp2 <- MOD_K [ num , 100 ]
    temp3 <- MOD_K [ num , 400 ]

    jackieAsks [ temp ] !
        jackieAsks [ temp2 ] !
            jackieAsks [ temp3 ] !
                re <- true
            ! elseJackie !
                re <- false
            !
        ! elseJackie !
            re <- true
        !
    ! elseJackie !
        re <- false
    !

    jackieReturns re
!
!
```

Coverage Results

- When clicked on 'Statement Coverage' from the navigation bar, it will highlight in yellow all the lines that were executed from the method call in the test case that you clicked on from the first page. The right hand side will show your coverage result. The orange highlighted boxes on the left hand side indicate you can click on it to display coverage results of those individual method calls.

Game Class

Click button below for coverage ↓

| ← TestCase | Statement Coverage | Condition Coverage | All-Defs Coverage | All-C-Uses Coverage | All-P-Uses Coverage |

Statement Coverage

```
game Aclass !

mymethod MOD_K BOOLEAN [INT num, INT k] !

    re << BOOLEAN
    count << INT

    loop (10000) !
        jackieAsks [ num == 0 ] !
            re <- true
        ! elseJackie !
            jackieAsks [ num < 0 ] !
                re <- false
            ! elseJackie !
                jackieAsks [ num > 0 ] !
                    num <- num - k
                ! elseJackie !
                !
            !
        !
    !

    jackieReturns re
!
mymethod LEAPYEAR BOOLEAN [INT num] !

    re << BOOLEAN
    temp << BOOLEAN
    temp2 << BOOLEAN
    temp3 << BOOLEAN

    temp <- MOD_K [ num , 4 ]
    temp2 <- MOD_K [ num , 100 ]
    temp3 <- MOD_K [ num , 400 ]

    jackieAsks [ temp ] !
        jackieAsks [ temp2 ] !
            jackieAsks [ temp3 ] !
                re <- true
            ! elseJackie !
                re <- false
            !
        ! elseJackie !
            re <- true
        !
    ! elseJackie !
        re <- false
    !

    jackieReturns re
!
!
```

☐ Click method call for coverage

Percentage = 80%

Note:
☐ statement coverage

When clicked on the orange boxes on the left, on the right hand side will display the method body of that method call you clicked on and display statement coverage for that call. (The right hand side box will always be displayed at the top, so you will need to scroll all the way up to see it)

Game Class

Click button below for coverage ↓

| ← TestCase | Statement Coverage | Condition Coverage | All-Defs Coverage | All-C-Uses Coverage | All-P-Uses Coverage |

Statement Coverage

game Aclass !

  mymethod MOD_K BOOLEAN [INT num, INT k] !

    re << BOOLEAN
    count << INT

    loop (10000) !
      jackieAsks [num == 0] !
        re <- true
      ! elseJackie !
        jackieAsks [num < 0] !
          re <- false
        ! elseJackie !
          jackieAsks [num > 0] !
            num <- num - k
          ! elseJackie !
            !
          !
        !
      !

    jackieReturns re
  !

  mymethod LEAPYEAR BOOLEAN [INT num] !

    re << BOOLEAN
    temp << BOOLEAN
    temp2 << BOOLEAN
    temp3 << BOOLEAN

    temp <- MOD_K [num , 4]
    temp2 <- MOD_K [num , 100]
    temp3 <- MOD_K [num , 400]

    jackieAsks [temp] !
      jackieAsks [temp2] !
        jackieAsks [temp3] !
          re <- true
        ! elseJackie !
          re <- false
        !
      ! elseJackie !
        re <- true
      !
    ! elseJackie !
      re <- false
    !

    jackieReturns re
  !
!

Click method call for coverage

Percentage => 80%

Note:
    statement coverage

  mymethod MOD_K BOOLEAN [INT num, INT k] !

    re << BOOLEAN
    count << INT

    loop (10000) !
      jackieAsks [num == 0] !
        re <- true
      ! elseJackie !
        jackieAsks [num < 0] !
          re <- false
        ! elseJackie !
          jackieAsks [num > 0] !
            num <- num - k
          ! elseJackie !
            !
          !
        !
      !

    jackieReturns re
  !

Condition Coverage

When clicked on condition coverage, the page will display a similar page, but this time you can click a button next to if-statements to see a table of conditions popping up showing condition coverage results.

Game Class

Click button below for coverage ↓

| ← TestCase | Statement Coverage | Condition Coverage | All-Defs Coverage | All-C-Uses Coverage | All-P-Uses Coverage |

Condition Coverage

game Aclass !

  mymethod DIGIT_CHECKER INT [INT num] !

    re << INT

    jackieAsks [(num >= 0 && num <= 9)] !☞ See Coverage
      re <- 1
    ! elseJackie !
      jackieAsks [(num >= 10 && num <= 99)] !☞ See Coverage
        re <- 2
      ! elseJackie !
        jackieAsks [(num >= 100 && num <= 999)] !☞ See Coverage
          re <- 3
        ! elseJackie !
          jackieAsks [(num >= 1000 && num <= 9999)] !☞ See Coverage
            re <- 4
          ! elseJackie !
            jackieAsks [(num >= 10000 && num <= 99999)] !☞ See Coverage
              re <- 5
            ! elseJackie !
              jackieAsks [(num >= 100000)] !☞ See Coverage
                re <- 100
              ! elseJackie !
                re <- -1
              !
            !
          !
        !
      !

    jackieReturns re
  !
!

DIGIT_CHECKER.jackieAsks[(num>=1000&&num<=9999)]

possible case #	num>=1000	num<=9999	is tested?
1	T	T	✔
2	T	F	✔
3	F	T	✔
4	F	F	

75.00% covered!

The condition coverage displays how many conditions in conditional statements of source code were covered during the test case. If a condition statement contains variable

components that affect the condition, a clickable button appears next to the condition statement. When it is clicked, a table for that conditional statement pops up. At the top of the table, it shows which method the conditional statement belongs to. If statement[condition]. First column of the table demonstrates the number of possible cases for the condition that can be done. The number of rows under the first row will be 2 to the power of the component(variable) number. From the second column, each component will be shown. In that column, you can see a combination of the conditions that each of the components has in the actual test case, and how those conditions will affect each other. The number of the combination is the number of rows which was described above. The last column is named 'is tested?', and it shows a check mark symbol for rows when the combination of the component in the condition has been tested. If any of the condition combinations were not covered or a conditional statement was not reached, it will be blank. Finally below the table, it shows the percentage of the condition coverage for the specific conditional statement. You can click the button again to close the table that popped up.

All Defs Coverage/All-C-Uses Coverage/ All-P-Uses Coverage

- All Defs Coverage will display your input program on the left of the page. On the right of the page, there is an all def coverage calculation result along with a table of declared variables for the user to click on to see the path of usage from definition to c-use/p-use. There is a colour legend on the right of that to indicate what different colours mean. All-C-Uses and All-P-Uses Coverage display similar results for its respective coverage.

Game Class

4 Justification of Output

4.1 Justification for the Statement Coverage Criterion

The statement coverage indicates how many statements are executed out of the total number of statements for each test case method call. Statement includes declarations, assignments, if statements (condition evaluation), does not count 'else !' but instead counts statements inside the body of else, and includes a return statement.

If you look at output HTML files generated from Program3.txt and Test3_Program3.txt and click on the statement coverage. When you click on any of the method calls from the test case html page, it will take you to statement coverage and highlight the lines executed based on the argument of 'num' and take you to the range it belongs in. You can count the number of highlighted lines and divide it over the total number of lines and the percentage result matches that.

In the screenshot below, we selected DIGIT_CHECKER [52356], if you follow the flow of execution, you can see that the yellow lines are highlighted correctly. There are 15 statements total, and 7 highlighted. Which gives you ~46% statement coverage.

Game Class

Click button below for coverage ↓

| ← TestCase | Statement Coverage | Condition Coverage | All-Defs Coverage | All-C-Uses Coverage | All-P-Uses Coverage |

Statement Coverage

game Aclass !

mymethod DIGIT_CHECKER INT [INT num] !

re << INT

jackieAsks [(num >= 0 && num <= 9)] !
 re <- 1
! elseJackie !
 jackieAsks [(num >= 10 && num <= 99)] !
  re <- 2
 ! elseJackie !
  jackieAsks [(num >= 100 && num <= 999)] !
   re <- 3
  ! elseJackie !
   jackieAsks [(num >= 1000 && num <= 9999)] !
    re <- 4
   ! elseJackie !
    jackieAsks [(num >= 10000 && num <= 99999)] !
     re <- 5
    ! elseJackie !
     jackieAsks [(num >= 100000)] !
      re <- 100
     ! elseJackie !
      re <- -1
     !
    !
   !
  !
 !

jackieReturns re
!
!

→ Click method call for coverage

Percentage = 53%

Note: statement coverage

4.2 Justification for the Condition Coverage Criterion

The condition coverage means how many conditions in each conditional statement in the source code were covered out of all the possible cases during the actual test. The condition coverage percentage is calculated as Covered cases number / 2^ component number (which is possible cases number) * 100.

TestCase

```
1 test test_api02 !
2   input1 << DOUBLE
3   input2 << DOUBLE
4   bool1 << BOOLEAN
5   result << DOUBLE

7   input1 <- 1.23
8   input2 <- 2.37
9   bool1 <- false
10  result <- MATH [ bool1 , input1, input2 ]
!
```

Result

Line 10 : result <- MATH [bool1, input1, input2]

```
dd << DOUBLE

    dd <- 0.0
    tester1 <- 1 + 2
    tester2 <- 2 + tester1
    input2 <- input1 + input2
    flag <- true
    input2 <- input1 + input2

    jackieAsks [ b ] !☞ See Coverage
        result <- input2
    ! elseJackie !
        result <- 0.0
    !

    loop (5) !
        temp << INT

        jackieAsks [ 1 < 2 ] !☞ See Coverage
            temp <- temp + 1
            loresult <- temp
        ! elseJackie !
            loresult <- temp
        !
    !

    jackieReturns loresult
!
!
```

MATH.jackieAsks[b]

possible case #	b	is tested?
1	T	
2	F	✔

50.00% covered!

MATH.jackieAsks[1<2]

possible case #	1<2	is tested?
1	T	✔
2	F	

50.00% covered!

[Example1: Program1 & Test1_Program1]

In this test, MATH.jackieAsks[b] contains a condition 'b' which is a boolean variable that is determined by the user input bool1. In this conditional statement, there is one component, 'b'. Since b is a single boolean variable, there can be 2 (2^number of component) possible cases: 1. b is true 2. b is false. In the testcase, bool1 was false. So the 3rd row of MATH.jackieAsks[b] table has been checked. Since only one condition has passed, 50 % of the possible case number has been covered in MATH.jackieAsks[b]. Also in MATH.jackieAsks[1<2], the component is '1<2'. This condition won't be affected by the user input, and this condition is considered as true as 1<2 is always true. So it can't reach the elseJackie statement and shows 50% coverage for MATH.jackieAsks[1<2].

Click method call below for more coverage ↓

```
1  test test_Aclass !
2    re1 << INT
3    re2 << INT
4    re3 << INT
5    re4 << INT
6    re5 << INT
7    re6 << INT
8    re_1 << INT

10   re1 <- DIGIT_CHECKER [ 1 ]
11   re2 <- DIGIT_CHECKER [ 56 ]
12   re3 <- DIGIT_CHECKER [ 141 ]
13   re4 <- DIGIT_CHECKER [ 3125 ]
14   re5 <- DIGIT_CHECKER [ 52356 ]
15   re6 <- DIGIT_CHECKER [ 562349 ]
16   re_1 <- DIGIT_CHECKER [ -1 ]
!
```

```
game Aclass !

  mymethod DIGIT_CHECKER INT [INT num] !

    re << INT

    jackieAsks [ ( num >= 0 && num <= 9 ) ] !☞ See Coverage
        re <- 1
    ! elseJackie !
        jackieAsks [ ( num >= 10 && num <= 99 ) ] !☞ See Coverage
            re <- 2
        ! elseJackie !
            jackieAsks [ ( num >= 100 && num <= 999 ) ] !☞ See Coverage
                re <- 3
            ! elseJackie !
                jackieAsks [ ( num >= 1000 && num <= 9999 ) ] !☞ See (75.00% covered!
                    re <- 4
                ! elseJackie !
                    jackieAsks [ ( num >= 10000 && num <= 99999 ) ] !☞ See Coverage
                        re <- 5
                    ! elseJackie !
                        jackieAsks [ ( num >= 100000 ) ] !☞ See Coverage
                            re <- 100
                        ! elseJackie !
                            re <- -1
                        !
                    !
                !
```

DIGIT_CHECKER.jackieAsks[(num>=0&&num<=9)]

possible case #	num>=0	num<=9	is tested?
1	T	T	✔
2	T	F	✔
3	F	T	✔
4	F	F	

[Example2: Program3 & Test3_Program3]

In this test, I clicked one of the condition coverage: jackieAsks[(num>=0&&num<=9)] from DIGIT_CHECKER method. num is determined by user input, and there are 7 inputs in the test file. Since there are 2 components in the conditional statement, which are num>=0 and num<=9, the possible cases are 2^2 = 4. When num = 1, num>=0 is true and num<=9 is also true, so row 1 is checked. When num = 56, 141, 3125, 52356, 562349, num>=0 is true but num<=9 is false. So these test cases covered row 2 (checked). When num = -1, num>=0 is false but num<=9 is true, so row 3 is checked. Since 3 of 4 cases were covered, the coverage rate is 75%.

4.3 Justification for the All-Defs Coverage Criterion

All def coverage measures: out of all the definitions (assignments and declarations) , how many of those are used either for predicate use or computational use during runtime. Since we need a variable to be declared before we can assign it a value, if we assign it a value we only count our assignment to be the definition and not count the declaration. If there is a variable declaration without an assignment made to it, then we will count the definition as the declaration.
Our calculations are based on the number of c-use/p-use for every definition(declarations and assignments).

If you look at output HTML files generated from Program2.txt and Test2_Program2.txt and click on SETUP_ACCOUNT[100.0] then All-Def Coverage, you can click on the variables on the right to link each definition to its c-use/p-use with color legend. It shows 100% coverage because each of the variables on the right shows a use for every definition covered during runtime. This method calls another method: CHECK_IF_BALANCE_POSITIVE, and we also count the definitions and compute the total of all methods called within it. In the image below, we highlighted variable 'result' in green for definition and c-uses in yellow.

Game Class

Click button below for coverage ↓

| ← TestCase | Statement Coverage | Condition Coverage | All-Defs Coverage | All-C-Uses Coverage | All-P-Uses Coverage |

All-Defs Coverage

game Firstclass !

  mymethod SETUP_ACCOUNT DOUBLE [DOUBLE balance] !

    account << DOUBLE
    checker << BOOLEAN

    account <- balance
    checker <- CHECK_IF_BALANCE_POSITIVE [account]

    jackieAsks [checker] !
      account <- SET_UP_FEE [account]
    ! elseJackie !
      account <- 0.0
    !

    jackieReturns account
 !

List of Variables - Click to see coverage:

Percentage = 100%

balance

account

checker

result

Note:
green => def
yellow => c-use
purple => p-use
red => no c-use or p-use

If you look at html of Program1.txt and Test1_Program1.txt and click on MATH[bool1, input1, input2], then all def coverage, the coverage is 9/16 ~ 56% if you count all the definitions and uses generated from the table on the right hand side. The image below shows input1 definition highlighted in green, and the c-use of it highlighted in yellow.

Game Class

All-Defs Coverage

game Mathclass !

mymethod MATH INT [BOOLEAN b, DOUBLE input1, DOUBLE input2] !

```
result << DOUBLE
tester1 << INT
tester2 << INT
tester3 << INT
flag << BOOLEAN
loresult << INT
dd << DOUBLE

dd <- 0.0
tester1 <- 1 + 2
tester2 <- 2 + tester1
input2 <- input1 + input2
flag <- true
input2 <- input1 + input2

jackieAsks [ b ] !
    result <- input2
! elseJackie !
    result <- 0.0
!

loop (5) !
    temp << INT

    jackieAsks [ 1 < 2 ] !
        temp <- temp + 1
        loresult <- temp
    ! elseJackie !
        loresult <- temp
    !
!

jackieReturns loresult
!
!
```

List of Variables - Click to see coverage:

Percentage = 56%

b
input1
input2
result
tester3
loresult
dd
tester1
tester2
flag
temp

Note:
- green => def
- yellow => c-use
- purple => p-use
- red => no c-use or p-use

4.4   Justification for the All-C-Uses Coverage Criterion

All-C-Use coverage measures all the definition clear paths between every definition variable to every computational use of that variable from its execution flow.

Similarly to All-Def Coverage, if you look at html of Program1.txt and Test1_Program1.txt and click on MATH[bool1, input1, input2], then all-c-uses coverage, the coverage is 8/16 ~ 50% if you count all the definitions and uses generated from the table on the right hand side. We calculated the number of computational uses of each definition out of the total number of definitions.

The image below shows tester1 underlined in red to indicate its a definition, and tester1 highlighted in yellow to show the corresponding c-use for that definition. The second image shows tester2 definition, but with no c-use so it is highlighted in red.

Game Class

Click button below for coverage ↓

| ← TestCase | Statement Coverage | Condition Coverage | All-Defs Coverage | All-C-Uses Coverage | All-P-Uses Coverage |

All-C-Uses Coverage

game Mathclass !

   mymethod MATH INT [BOOLEAN b, DOUBLE input1, DOUBLE input2] !

      result << DOUBLE
      tester1 << INT
      tester2 << INT
      tester3 << INT
      flag << BOOLEAN
      loresult << INT
      dd << DOUBLE

      dd <- 0.0
      tester1 <- 1 + 2
      tester2 <- 2 + tester1
      input2 <- input1 + input2
      flag <- true
      input2 <- input1 + input2

      jackieAsks [b] !
         result <- input2
      ! elseJackie !
         result <- 0.0
      !

      loop (5) !
         temp << INT

         jackieAsks [1 < 2] !
            temp <- temp + 1
            loresult <- temp
         ! elseJackie !
            loresult <- temp
         !
      !

      jackieReturns loresult
   !
!

List of Variables - Click to see coverage:

Percentage = 50%

| b |
| input1 |
| input2 |
| result |
| tester3 |
| loresult |
| dd |
| tester1 |
| tester2 |
| flag |
| temp |

Note:
   underline => def
   yellow => c-use
   red => no c-use

Game Class

Click button below for coverage ↓

| ← TestCase | Statement Coverage | Condition Coverage | All-Defs Coverage | All-C-Uses Coverage | All-P-Uses Coverage |

All-C-Uses Coverage

game Mathclass !

   mymethod MATH INT [BOOLEAN b, DOUBLE input1, DOUBLE input2] !

      result << DOUBLE
      tester1 << INT
      tester2 << INT
      tester3 << INT
      flag << BOOLEAN
      loresult << INT
      dd << DOUBLE

      dd <- 0.0
      tester1 <- 1 + 2
      tester2 <- 2 + tester1
      input2 <- input1 + input2
      flag <- true
      input2 <- input1 + input2

      jackieAsks [b] !
         result <- input2
      ! elseJackie !
         result <- 0.0
      !

      loop (5) !
         temp << INT

         jackieAsks [1 < 2] !
            temp <- temp + 1
            loresult <- temp
         ! elseJackie !
            loresult <- temp
         !
      !

      jackieReturns loresult
   !
!

List of Variables - Click to see coverage:

Percentage = 50%

| b |
| input1 |
| input2 |
| result |
| tester3 |
| loresult |
| dd |
| tester1 |
| tester2 |
| flag |
| temp |

Note:
   underline => def
   yellow => c-use
   red => no c-use

4.5   Justification for the All-P-Uses Coverage Criterion

All-P-Use coverage measures all the definition clear paths between every definition variable to every predicate use (in if-statements) of that variable from its execution flow.

Similarly to All-Def Coverage and All-C-Use Coverage, if you look at html of Program1.txt and Test1_Program1.txt and click on MATH[bool1, input1, input2], then all-p-uses coverage, the coverage is 1/16 ~ 6% if you count all the definitions and uses generated from the table on

the right hand side. We calculated the number of predicate uses of each definition out of the total number of definitions. The total definition here is 16 and only definition 'b' is ever used.

The image below shows the variable 'b' definition underlined in red, and p-use highlighted in yellow. In contrast, the second photo shows variable 'result' highlighted in red twice because it is defined twice but there is no p-use for it.
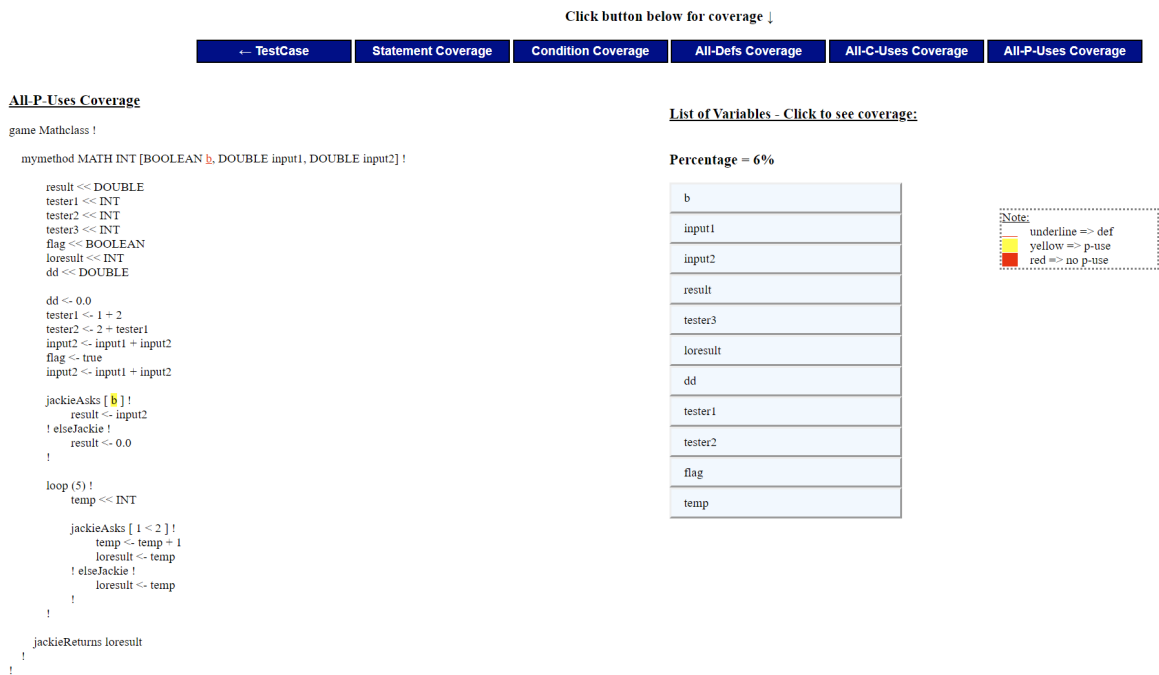
Game Class

Click button below for coverage ↓

| ← TestCase | Statement Coverage | Condition Coverage | All-Defs Coverage | All-C-Uses Coverage | All-P-Uses Coverage |

All-P-Uses Coverage

game Mathclass !

mymethod MATH INT [BOOLEAN b, DOUBLE input1, DOUBLE input2] !

```
        result << DOUBLE
        tester1 << INT
        tester2 << INT
        tester3 << INT
        flag << BOOLEAN
        loresult << INT
        dd << DOUBLE

        dd <- 0.0
        tester1 <- 1 + 2
        tester2 <- 2 + tester1
        input2 <- input1 + input2
        flag <- true
        input2 <- input1 + input2

        jackieAsks [ b ] !
            result <- input2
        ! elseJackie !
            result <- 0.0
        !

        loop (5) !
            temp << INT

            jackieAsks [ 1 < 2 ] !
                temp <- temp + 1
                loresult <- temp
            ! elseJackie !
                loresult <- temp
            !
        !

        jackieReturns loresult
        !
!
```

List of Variables - Click to see coverage:

Percentage = 6%

| b |
| input1 |
| input2 |
| result |
| tester3 |
| loresult |
| dd |
| tester1 |
| tester2 |
| flag |
| temp |

Note:
underline => def
yellow => p-use
red => no p-use

Image 1: variable b

Game Class

Click button below for coverage ↓

| ← TestCase | Statement Coverage | Condition Coverage | All-Defs Coverage | All-C-Uses Coverage | All-P-Uses Coverage |

All-P-Uses Coverage

game Mathclass !

mymethod MATH INT [BOOLEAN b, DOUBLE input1, DOUBLE input2] !

```
        result << DOUBLE
        tester1 << INT
        tester2 << INT
        tester3 << INT
        flag << BOOLEAN
        loresult << INT
        dd << DOUBLE

        dd <- 0.0
        tester1 <- 1 + 2
        tester2 <- 2 + tester1
        input2 <- input1 + input2
        flag <- true
        input2 <- input1 + input2

        jackieAsks [ b ] !
            result <- input2
        ! elseJackie !
            result <- 0.0
        !

        loop (5) !
            temp << INT

            jackieAsks [ 1 < 2 ] !
                temp <- temp + 1
                loresult <- temp
            ! elseJackie !
                loresult <- temp
            !
        !

        jackieReturns loresult
        !
!
```

List of Variables - Click to see coverage:

Percentage = 6%

| b |
| input1 |
| input2 |
| result |
| tester3 |
| loresult |
| dd |
| tester1 |
| tester2 |
| flag |
| temp |

Note:
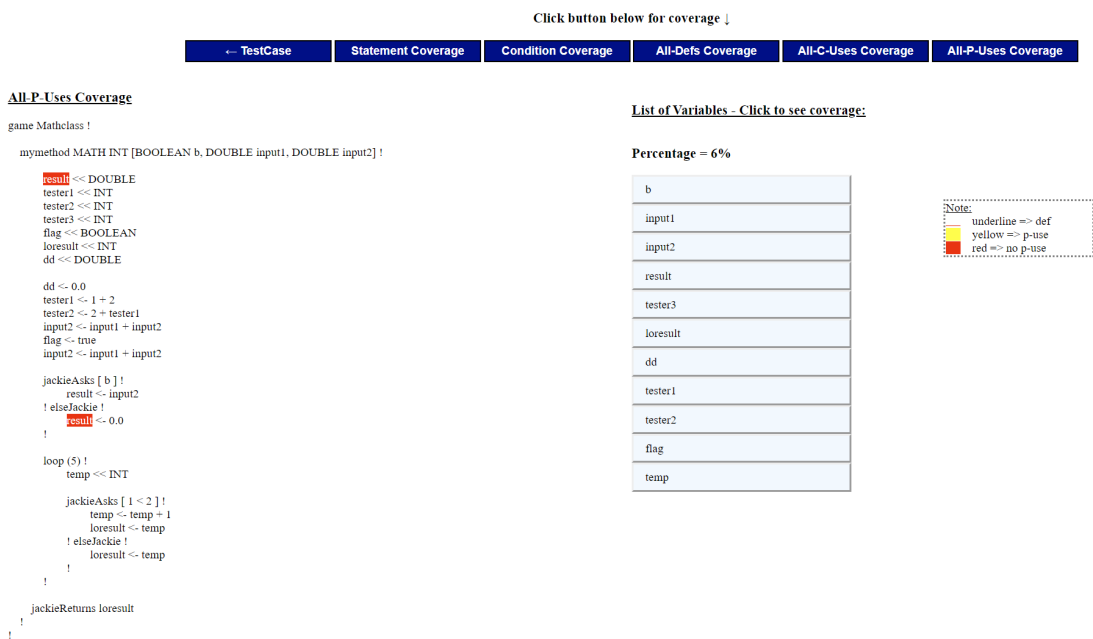underline => def
yellow => p-use
red => no p-use

Image 2: variable result

5   Summary of Submitted Examples

5.1   Highlights of Example Input 1
- Files: Program1 and test1_program1
    - o   This test is a general test to demonstrate a normal standard program.
    - o   It contains different data types and **jackieAsks** and loops
    - o   It also demonstrates variable scoping

5.2   Highlights of Example Input 2
- Files: Program2 and test2_program2
    - o   This program is simulating a bank account and the test tests those functions
    - o   This demonstrates functions calls and variable scope

5.3   Highlights of Example Input 3
- Files: Program3 and test3_program3
    - o   This program demonstrates nested **jackieAsks**

5.4   Highlights of Example Input 4
- Files: Program4 and test4_program4
    - o   This program demonstrates multiple **jackieAsks,** function calls and branching conditions and results by implementing a simple calculator.

5.5   Highlights of Example Input 5
- Files: Program5 and test5_program5
    - o   This program demonstrates chain call

5.6   Highlights of Example Input 6
- Files: Program6 and test5_program6
    - o   This program demonstrates chain call inside loops

5.7   Highlights of Example Input 7
- Files: Program7 and test5_program7
    - o   This program demonstrates the ability to write branching paths with loops and demonstrates its ability to do math problems by implementing power functions and some math functions using power.

5.8   Highlights of Example Input 8
- Files: Program8 and test8_program8
    - o   This program demonstrate the language ability with String (assigning)

5.9   Highlights of Example Input 9
- Files: Program9 and test9_program9
    - o   This program demonstrates further ability for loops and math.
    - o   It implements modular arithmetic
    - o   It also checks for prime and count number of prime from 1 to n
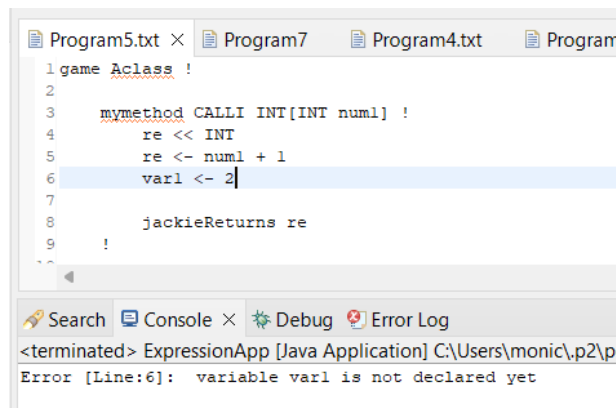
5.10   Highlights of Example Input 10
- Files: Program10 and test10_program10
    - o   This programs implements a leap year checker

6 Miscellaneous Features

6.1 Error reporting

This language is not a scripting language so we have a syntax checker with line number and columns of problem reported, it will only execute if there are no errors. There is also semantic error checking for logic errors, with that we have error reporting to tell if there are problems with the program (reasons for why it can't be compile) here are a list of possible problems:
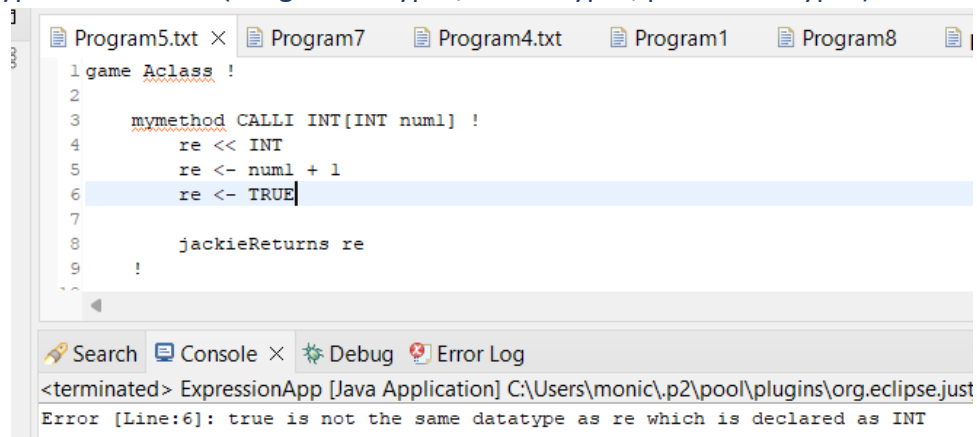
6.1.1 Use or assignment of undeclared variable

```
Program5.txt ×   Program7      Program4.txt      Program
 1 game Aclass !
 2
 3    mymethod CALLI INT[INT num1] !
 4        re << INT
 5        re <- num1 + 1
 6        var1 <- 2
 7
 8        jackieReturns re
 9    !
```

```
Search   Console ×   Debug   Error Log
<terminated> ExpressionApp [Java Application] C:\Users\monic\.p2\p
Error [Line:6]:  variable var1 is not declared yet
```

6.1.2 Type mismatched (assignment types, return types, parameter types)
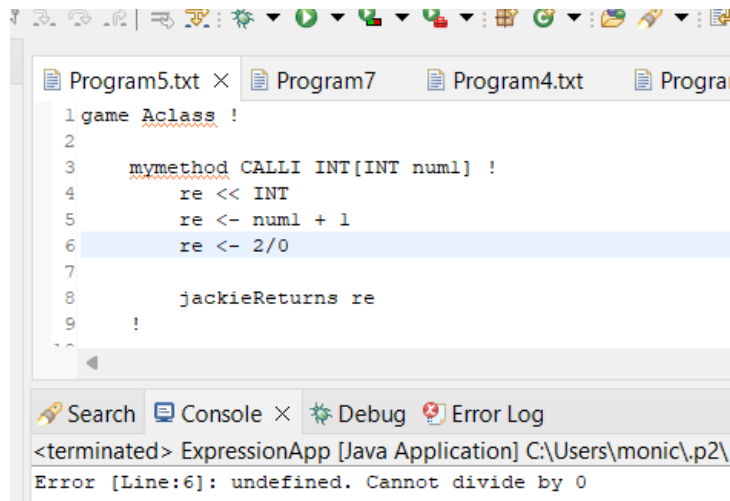
```
Program5.txt ×   Program7      Program4.txt      Program1      Program8
 1 game Aclass !
 2
 3    mymethod CALLI INT[INT num1] !
 4        re << INT
 5        re <- num1 + 1
 6        re <- TRUE
 7
 8        jackieReturns re
 9    !
```

```
Search   Console ×   Debug   Error Log
<terminated> ExpressionApp [Java Application] C:\Users\monic\.p2\pool\plugins\org.eclipse.just
Error [Line:6]: true is not the same datatype as re which is declared as INT
```
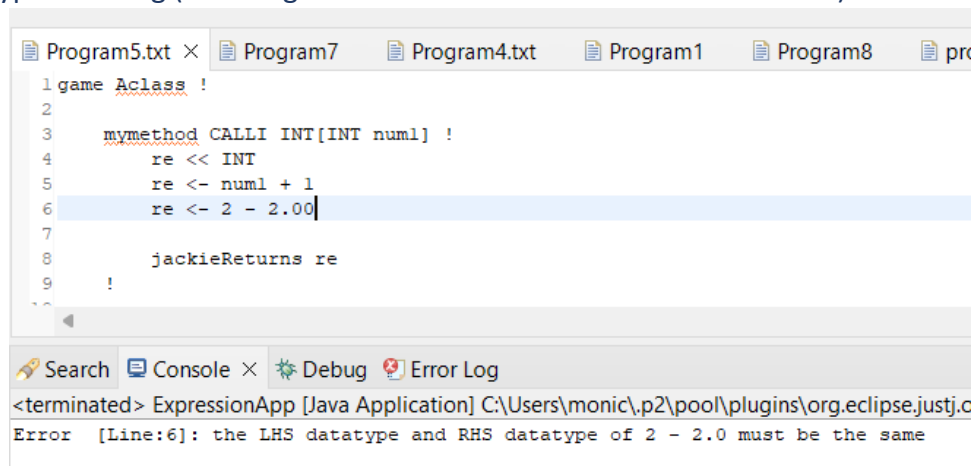
6.1.3 Division by Zero

```
Program5.txt ×    Program7      Program4.txt      Progra

1 game Aclass !
2
3     mymethod CALLI INT[INT num1] !
4         re << INT
5         re <- num1 + 1
6         re <- 2/0
7
8         jackieReturns re
9     !
```

Search Console × Debug Error Log
<terminated> ExpressionApp [Java Application] C:\Users\monic\.p2\
Error [Line:6]: undefined. Cannot divide by 0

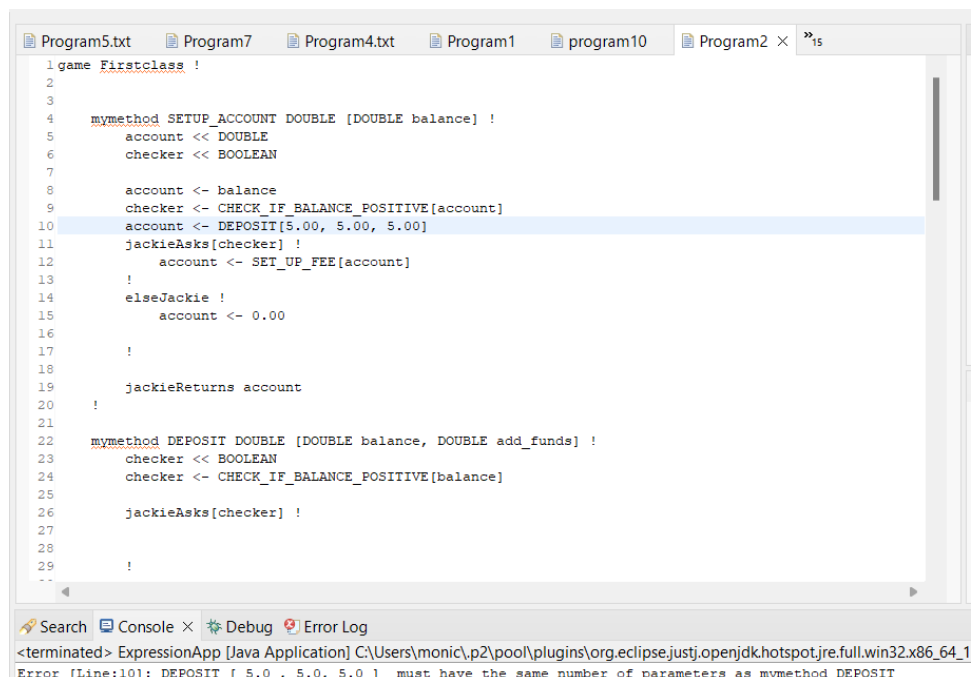### 6.1.4 Type checking (i.e. using a double with an int to calculate addition)

```
Program5.txt ×    Program7      Program4.txt      Program1      Program8      pro

1 game Aclass !
2
3     mymethod CALLI INT[INT num1] !
4         re << INT
5         re <- num1 + 1
6         re <- 2 - 2.00
7
8         jackieReturns re
9     !
```

Search Console × Debug Error Log
<terminated> ExpressionApp [Java Application] C:\Users\monic\.p2\pool\plugins\org.eclipse.justj.c
Error [Line:6]: the LHS datatype and RHS datatype of 2 - 2.0 must be the same

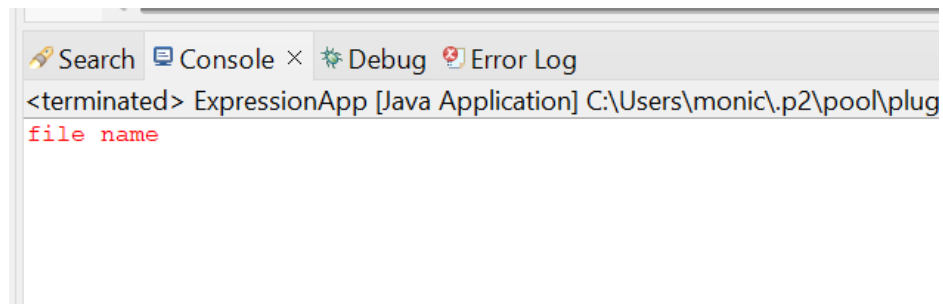### 6.1.5 Parameter checking (i.e. checking number of arguments passed into method calls)

```
Program5.txt    Program7      Program4.txt      Program1      program10      Program2 ×   15

1 game Firstclass !
2
3
4     mymethod SETUP_ACCOUNT DOUBLE [DOUBLE balance] !
5         account << DOUBLE
6         checker << BOOLEAN
7
8         account <- balance
9         checker <- CHECK_IF_BALANCE_POSITIVE[account]
10        account <- DEPOSIT[5.00, 5.00, 5.00]
11        jackieAsks[checker] !
12            account <- SET_UP_FEE[account]
13        !
14        elseJackie !
15            account <- 0.00
16
17        !
18
19        jackieReturns account
20    !
21
22    mymethod DEPOSIT DOUBLE [DOUBLE balance, DOUBLE add_funds] !
23        checker << BOOLEAN
24        checker <- CHECK_IF_BALANCE_POSITIVE[balance]
25
26        jackieAsks[checker] !
27
28
29        !
```

Search Console × Debug Error Log
<terminated> ExpressionApp [Java Application] C:\Users\monic\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_1
Error [Line:10]: DEPOSIT [5.00 , 5.0, 5.0] must have the same number of parameters as mymethod DEPOSIT

6.1.6 File Inputs (if number of files inputted is not 2, you will get an error to console)

```
 Search  Console ×  Debug  Error Log
<terminated> ExpressionApp [Java Application] C:\Users\monic\.p2\pool\plug
file name
```

7   Limitations

- Language supports simple loops, not while loops. Loops only support raw integer values (does not include variable names)
- Language does not have system print line
- String type cannot be concatenated, or checked for equality
- Assignments to Boolean variables cannot be a conditional expression, conditional expressions only allowed in if statement brackets (i.e. jackieAsks [expression]
- Cannot cast double to integer and vice versa