Document Type	Testing Manual
Name (Student Number)	• Ala Al Din Afana (16395986) • Shanan Lynch(14723071)
Supervisor	David Sinclair

1. Introduction

The goal of this document is to give readers an insight into the different testing techniques used to ensure that the project is correct (no major bugs) whilst incorporating user feedback to ensure high quality and user satisfaction.

For our project testing one of the most critical components is testing as it helped us ensure that the programming language is bug free. The project was developed using a test driven development approach where the tests were written first before the code was developed. We also utilized scripts to automate the process and facilitate regression testing every time we pushed code to our repository.

1.1 Components Tested

There are several components in the project that need to be tested. Essentially what we did was test every single aspect.

- **Lexical analysis** : **Lexical analysis** is the first phase of our compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. It takes a sequence of characters and converts them into a sequence of tokens. We performed the lexical analysis by parsing a sequence of words and checking if the parser parses successfully this will be discussed in more detail below.
- **Semantic analysis:** this phase of the compiler checks if the semantics are correct . We used the abstract syntax tree to derive the nodes and perform semantic analysis on the code. To help us with our semantic checks we created a symbol table . we use store in the symbol table data structure the **identifier** , **type** , **scope** , **description** The semantic checks we perform are:
    - Identifiers declared in scope
    - No multiple declarations of identifiers
    - Correct types for assignments
    - Argument of arithmetic operations of correct type
    - Does every function call have the correct number of arguments?
    - Correct type for lists
    - Identifier declared before used

  We performed the tests on the semantic checks by writing semantically correct and in correct code and checking if our compiler detects if the code is semantically correct or not.

- **IR code generation:** in this section the compiler generates the llvm ir code we then compile and run the llvm ir code , unit tests are run against the output from the llvm ir generation. In this section we test if our compiler is working as it is supposed to and that the ir code we are generating is correct this will be discussed in greater detail below.
- **Python flask web app “please talk about how you achieved this @shanan”**
- User testing this will be discussed later in the document

- **Integration testing: we performed the integration testing after we connected the web app to the compiler . “please talk about how you achieved this @shanan”**

2 Split off work

Ala Al Din Afana

- lexical analysis tests
- ir code generation tests
- Regression testing scripts
- user testing

Shanan lynch

- Semantic check tests
- Flask web app test
- Integration testing

3. Lexical analysis tests - Ala Afana.

For this component of the project the tests were written before any input code that tests the grammar of our compiler and ensures it's parsed correctly by our parser. The grammar of our language was designed by both of us, please see User Manual

For this section a testing mechanism is hard to come up with as there is no unit testing plugin similar to pytest or junit in java cc to facilitate automating the process. To overcome this I wrote a python script that runs our compiler with different mona programing language files as the input and performs a unit test that passes if a file has been parsed successfully.

To cover all aspects of lexical analysis I tested all the terminals and non terminals and wrote and checked if they have been parsed successfully. It took 50 tests to cover the whole grammar. **(please see video walkthrough to see tests running)**

1. **LexerTest1 -** tests main
2. **LexerTest2 -** tests comments
3. **LexerTest3 -** tests function return type void
4. **LexerTest4 -** tests main
5. **LexerTest5 -** tests variable declaration with all types
6. **LexerTest6 -** tests variable declaration with all types and assignment
7. **LexerTest7 -** tests constant declaration with all types
8. **LexerTest8 -** tests constant declaration with value assignments
9. **LexerTest9 -** tests declaration of multiple functions
10. **LexerTest10 -** tests return string function
11. **LexerTest11 -** tests return array string function

12. **LexerTest12 -** tests return int function
13. **LexerTest13-** tests return array int function
14. **LexerTest14 -** tests return float function
15. **LexerTest15 -** tests return array float function
16. **LexerTest16 -** tests return bool function
17. **LexerTest17 -** tests passing argument string to function
18. **LexerTest18 -** tests passing argument array string to function
19. **LexerTest19 -** tests passing argument float to function
20. **LexerTest20 -** tests passing argument array float to function
21. **LexerTest21 -** tests passing argument int to function
22. **LexerTest22 -** tests passing argument array int to function
23. **LexerTest23 -** tests passing argument bool to function
24. **LexerTest24 -** tests passing argument string to function and returning arg
25. **LexerTest25 -** tests passing argument array string to function and returning arg
26. **LexerTest26 -** tests passing argument int to function and returning arg
27. **LexerTest27 -** tests passing argument array int to function and returning arg
28. **LexerTest28 -** tests passing argument float to function and returning arg
29. **LexerTest29 -** tests passing argument array float to function and returning arg
30. **LexerTest30 -** tests passing argument bool to function and returning arg
31. **LexerTest31 -** test function with multiple arguments
32. **LexerTest32 -** test assigning value to declared variables
33. **LexerTest33 -** tests function call
34. **LexerTest34 -** test function call with return , an argument and assigning function call value
35. **LexerTest35 -** tests single if statement
36. **LexerTest36 -** tests multiple if statement and all comparative operators
37. **LexerTest37 -** tests if with else
38. **LexerTest38 -** tests if with multiple else ifs
39. **LexerTest39 - -** tests if with multiple else ifs and else
40. **LexerTest40 -** tests print
41. **LexerTest41 -** Test arithmetic operations int
42. **LexerTest42 -** Test arithmetic operations float
43. **LexerTest43 -** test comparative operators and floats
44. **LexerTest44 -** test array multiple values
45. **LexerTest45 -** test get elements from array
46. **LexerTest46 -** test insert element to array
47. **LexerTest47 -** test length array
48. **LexerTest48 -** test get character from string and length string
49. **LexerTest49 -** tests while loop

4. Intermediate code tests - Ala Afana.

The tests in this section are dependent on all tests in the previous section to be working correctly. In this section I focus on testing if the intermediate code visitor has generated the

correct intermediate code (i.e. the generated IR code compiles) and when compiled, the code returns the expected values.

I wrote a python script to test the return value of the compilation. There are two different use cases that I will be testing.

1. Files that print nothing to the terminal
    a. Ensure the llvm ir code compiled with no issues
2. Files that print something to the terminal e.g HelloWorld

    a. Ensure the LLVM code compiles

    b. Verify the output of the program

Below I will demonstrate how I did the first test, the rest of the tests follow the same format. Everytime new code is added to the project (e.g new function in IR visitor) I ran the python script to ensure that the newly developed code did not break previously working code. This is the approach that I followed to enforce regression testing throughout the development process and ensure that all other unit tests.

Example test format

The mona test file is compiled

IrcodeTest.mona

```
//test print hello world

main{

        print("hello world");

}
```

With a correct Ir Code Generation this code will print hello world in the terminal. IR code generation is done by traversing over the abstract syntax tree.

Generated IR code

```
@ bitcast i8* %.t1 to [20 x i8]*

%.t4 = load [20 x i8] , [20 x i8]* %.t3

store [20 x i8] %.t4,  [20 x i8]* %.t2

%.t5= bitcast [20 x i8]*%.t2 to i8*

call i32 @puts (i8* %.t5)
```

```
.1 = constant [12 x i8] c"hello world\00"

declare i32 @printf(i8*, ...) #1

@.1arg_str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

@.1arg_ = private unnamed_addr constant [3 x i8] c"%f\00", align 1

declare i32 @puts(i8*)

define i32 @main ()

 {

%.t2 = alloca [20 x i8]

%.t1 = getelementptr [12 x i8], [12 x i8]*@.1, i64 0, i64 0

%.t3 =ret i32 0

}
```

The Ir code is then compiled and Run

- `hello world` is printed in the terminal.
- The python script grabs the return value of the compilation of the IR code.
- The script then runs a pyunit test comparing the return value in this case (“hello world”) with our expected return value (“hello world”).
- If they are the same, in this case they are the test passes if not the test fails.

IR Code Tests and goal

1. **IrCodeTest1.mona** - test print hello world
2. **IrCodeTest2.mona** - test print integer
3. **IrCodeTest3.mona** - test print double
4. **IrCodeTest4.mona** - test adding two integers and assigning values to variable
5. **IrCodeTest5.mona** - test subtracting two integers and assigning values to variable
6. **IrCodeTest6.mona** - test multiplying two integers and assigning values to variable
7. **IrCodeTest7.mona** - test dividing two integers and assigning values to variable
8. **IrCodeTest8.mona** - test modulus two integers and assigning values to variable
9. **IrCodeTest9.mona** - test integer to the power
10. **IrCodeTest10.mona :**
    a. Assigning a value to an identifier
    b. Performing an arithmetic operation with the id
    c. Storing the value of the arithmetic operation in that id
11. **IrCodeTest11.mona** - test adding two floats and assigning values to variable
12. **IrCodeTest12.mona** - test subtracting two floats and assigning values to variable
13. **IrCodeTest13.mona** - test multiplying two floats and assigning values to variable

14. **IrCodeTest14.mona** - test dividing two floats and assigning values to variable
15. **IrCodeTest15.mona** - test modulus two floats and assigning values to variable
16. **IrCodeTest16.mona** - same as 10 but with floats
17. **IrCodeTest17.mona** - Test an array of integers (“This test tests if ir code compiles)
18. **IrCodeTest18.mona** - Test an array of floats (“This test tests if ir code compiles)
19. **IrCodeTest19.mona** - Test an array of strings (“This test tests if ir code compiles)
20. **IrCodeTest20.mona** - Test array length function
21. **IrCodeTest21.mona** - Test array string get function
22. **IrCodeTest22.mona** - Test array int get function
23. **IrCodeTest23.mona** - Test array float get function
24. **IrCodeTest24.mona** - Test array insert function and get value (string)
25. **IrCodeTest25.mona** - Test array insert function and get value (int)
26. **IrCodeTest26.mona** - Test array insert function and get value (float)
27. **IrCodeTest27.mona** - test if statement with comparative operator ==
28. **IrCodeTest28.mona** - test if statement with comparative operator <
29. **IrCodeTest29.mona** - test if statement with comparative operator >
30. **IrCodeTest30.mona** - test if statement with comparative operator >=
31. **IrCodeTest31.mona** - test if statement with comparative operator <=
32. **IrCodeTest32.mona** - test if statement with else
33. **IrCodeTest33.mona** - test if statement with multiple else_ifs
34. **IrCodeTest34.mona** - test if statement with multiple else_ifs and else
35. **IrCodeTest35.mona** - test if while loop with number one to ten and printing them
36. **IrCodeTest36.mona** - test length string
37. **IrCodeTest37.mona** - test comparing two declared strings
38. **IrCodeTest38.mona** - test comparing declared string with undeclared string
39. **IrCodeTest39.mona** - test comparing two undeclared strings
40. **IrCodeTest40.mona** - test getting a char from string
41. **IrCodeTest41.mona** - test function call hello world. Function prints hello world
42. **IrCodeTest42.mona** - test function call taking in string hello and printing it within function
43. **IrCodeTest43.mona** - test function call taking in integer 10 and printing it within function
44. **IrCodeTest44.mona** - test function call taking in float 1.0 and printing it within function
45. **IrCodeTest45.mona** - test function call taking in array of integers and printing its values
46. **IrCodeTest46.mona** - test function call taking in array of strings and printing its values
47. **IrCodeTest47.mona** - test function call taking in array of floats and printing its values
48. **IrCodeTest48.mona** - test function call taking in a string and returning that string. String is assigned to a variable and printed
49. **IrCodeTest49.mona** - test function call taking in an integer and returning integer. integer is assigned to a variable and printed
50. **IrCodeTest50.mona** - test function call taking in a float and returning float. float is assigned to a variable and printed

51. **IrCodeTest51.mona** - test function takes in a list and returns list . I test the length function on the returned list . this is an important test as im tracking the size of the list and any changes that occurred within that function
52. **IrCodeTest52.mona** - test function call taking in a array of ints and returning array all the values in the returned array are then printed and compared if their the same as the original array
53. **IrCodeTest53.mona** - test function call taking in a array of floats and returning array all the values in the returned array are then printed and compared if their the same as the original array
54. **IrCodeTest54.mona** - test constant declarations

5. User Testing - Ala Afana.

For User testing I created a google form with 9 questions that our test users will answer based on their experience in our platform. Please refer to the user testing problem document in this repository for more details.

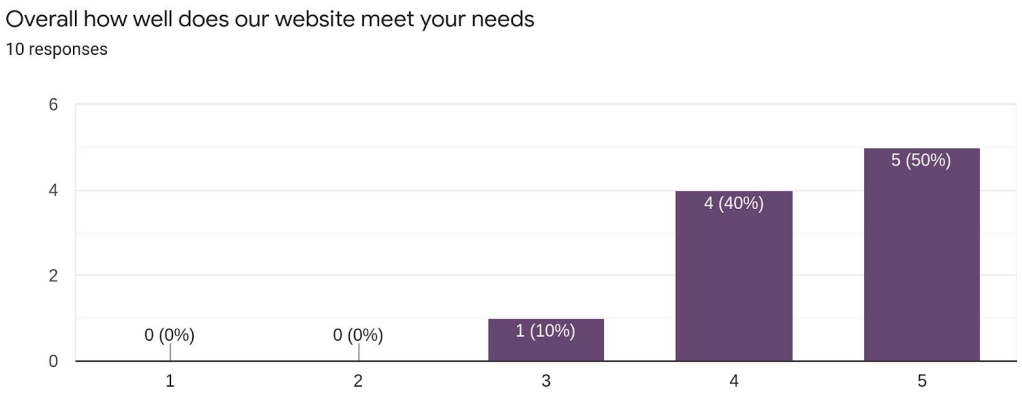
Testing Methodology:

1. The users will solve tasks in programming language they are least familiar with E.g a python developer in java or vice versa
2. Users then watch the tutorials videos I sent them (available in video walkthrough repo) and read the user manual to familiarise them with the Mona programming language.
3. Users will then perform the task1 using Mona
4. Users will be given a second problem (task 2) for them to solve in the same language they used in step 1.
5. Users answer the survey

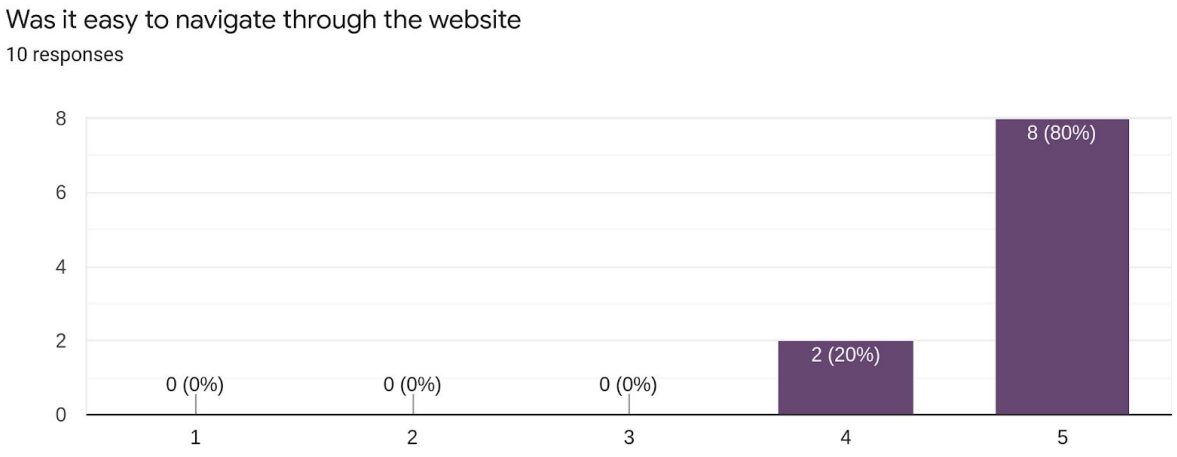
[Continued next page]

5.1 User testing results

Q1.

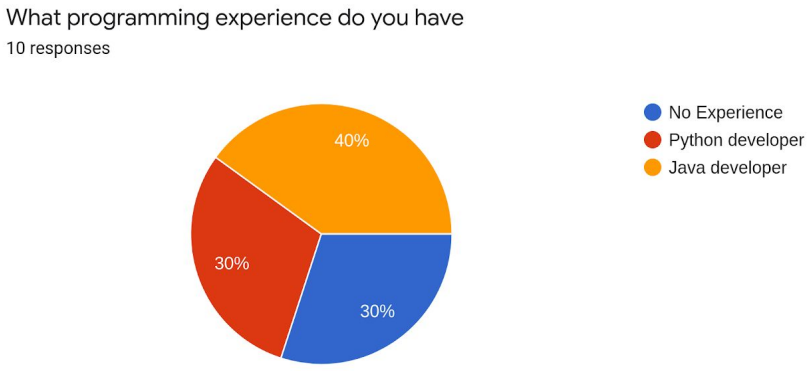


Q2.

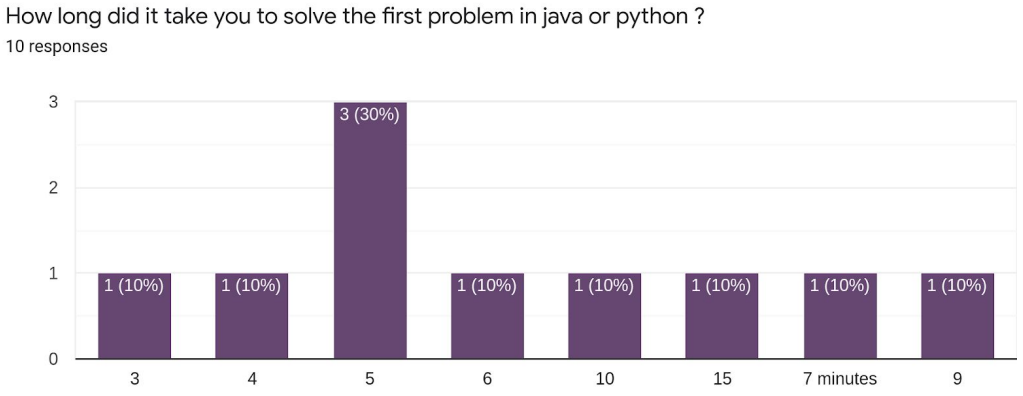


[Continued next page]

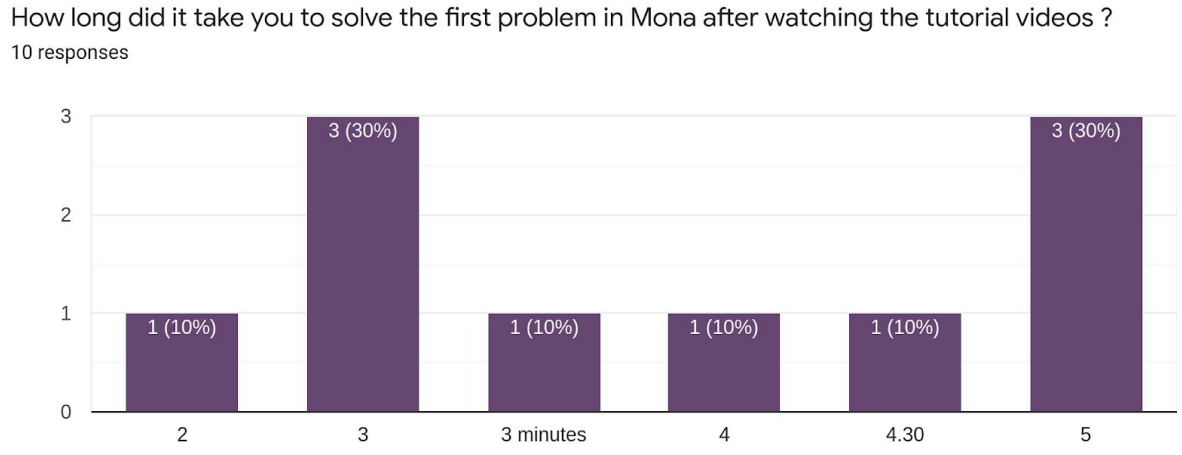
Q3.


What programming experience do you have
10 responses
40%
30%
30%
No Experience
Python developer
Java developer

Q4.


How long did it take you to solve the first problem in java or python ?
10 responses
3
2
1
0
1 (10%)
1 (10%)
3 (30%)
1 (10%)
1 (10%)
1 (10%)
1 (10%)
1 (10%)
3
4
5
6
10
15
7 minutes
9

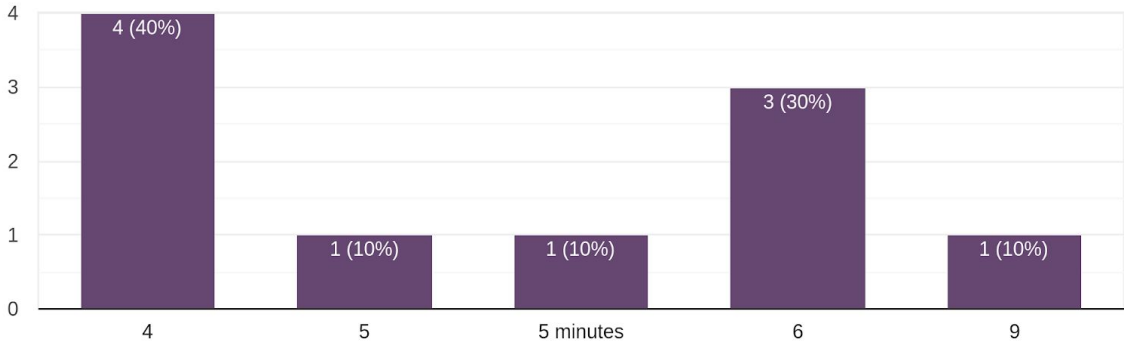
Q5.


How long did it take you to solve the first problem in Mona after watching the tutorial videos ?
10 responses
3
2
1
0
1 (10%)
3 (30%)
1 (10%)
1 (10%)
1 (10%)
3 (30%)
2
3
3 minutes
4
4.30
5

Q6

How long did it take to solve the second problem in python or java

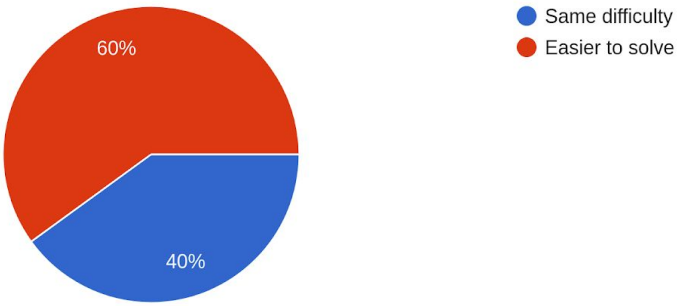
10 responses



Q7.

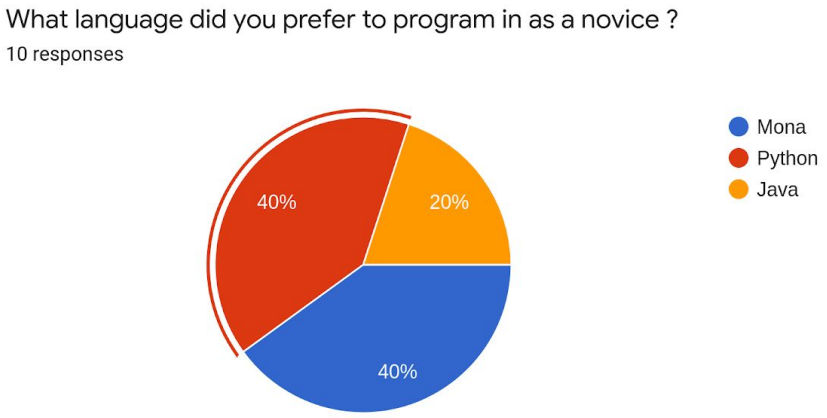
How would you compare your experience with the second problem compared to the first ?

10 responses



[Continued next page]

Q8.



Q9. *Would you recommend Mona over python or java to anyone interested in learning programming. If so why ? (*10 responses)

1. I would recommend mona , the tutorial videos and documentation seem easy to learn.
2. Python has the easiest syntax . no strict rules
3. yes Mona was easy to understand
4. no python looks easier
5. I recommend java as java is more complex and allows you to follow strict programming rules . Mona's grammar is similar to java but it is more flexible
6. No, Python is much easier for a novice
7. yes . It looks easy to use and has similar grammar to both python or java
8. It is not as strict as java but gives a good idea on java concepts . it would be a fun challenge to learn how to code in this :)
9. yes , found it easier to code in python after learning Mona
10. No . Java is the best.

5.2 User testing Analysis

We saw some pretty interesting results throughout our user testing some of which we used to derive improvements in our UI and another was a minor bug in the modulus function for an edge case we didn't cover.

5.2.1 User Experience

- Based on question 1 and 2 we can see that the majority of feedback for our UI was positive. Our UI is similar to online programming tools that the users are likely to have encountered before and we kept it very simple with minimal steps in order to write and compile code.

- In question one we received one 3 rating where the student followed up with us and advised us that he was unable to utilize the modulus function on floats. Upon receiving the feedback I tested this and realised it is in fact supported in python and java but not in c / c++. I used this feedback to further improve the platform and added the ability to perform modulo operations on floats.
- We also received 2 ratings in Q2 and the feedback that we got is that users would prefer the output to be displayed under the text editor rather than over it, especially as programs get bigger. We followed up by moving the output box accordingly.

5.2.2 Distribution of user experience

Users were more-less equally distributed with a slightly larger number of java developers. We feel that the fact only 30% of our test sample are python developers may have swayed our results.

5.2.3 Time taken to solve tasks

Looking at graphs for Q4,5 and 6 respectively we see that there was a slight improvement in the time to complete the tasks, however this may have been inhibited by the fact that 30% of our users have no programming experience so they are unfamiliar with programming as a whole (programming concepts aren't picked up in the space of 30 minutes).

5.2.4 Did we meet our goal

- Q7 and 8 focus on evaluating did our programming language meet its goal i.e. providing users with a language where they can find it easier to transition to and from.
- We see that in Q7 it is all positive feedback where users felt the second task was the same difficulty or easier to solve and not harder which means we didn't cause any confusion by the introduction of our grammer.
- Q8 tied our language with java based on which the users preferred to use as a novice programmer. There are many reasons for this such as:
    - Our program is more similar to java than python i.e most java developers preferred something that isn't too different .
    - Some Novice developers preferred our language for the following 2 reasons
        - They didn't have to figure out python's dynamic typing in 30 minutes.
        - They didn't have to establish an understanding of all of java's syntax to write a simple program (code inside a class, public static void main etc.)
    - Some python developers preferred our language as it introduced not too many concepts at once as a novice they had only figure out:
        - variable type
        - { } are used instead of indentation

- ; at the end of each line

Main function is more similar to python than it is in java.

5.2.5 Limitations

- We were restricted to a sample size of only 10 test users
- users were restricted to only 2 problems
- There may have been a slight bias programmers had access to mona tutorial videos and documentation whereas they had to research for languages they don't know.
- Programming is a skill picked up over time i.e. we would have gained a much deeper insight if the users did 2 exercises for a week / month but we didn't have the resources or test sample.

6. Semantic check tests - Shanan Lynch

7. Flask web app test - Shanan Lynch

8. Integration testing - Shanan Lynch