halfway evaluation

This section is going to be my opinion on where the system currently is and doing some larger scale testing. This does not mean final testing, as I will still be able to fix any errors that I come across, however I have only ever tested small parts of the system by themselves, so this will be combining lots of different aspects of the program and ensuring that they work cohesively together, because the whole point of the Interpreter is that it can adapt to all of the different inputs it could receive.

# the current state of the program

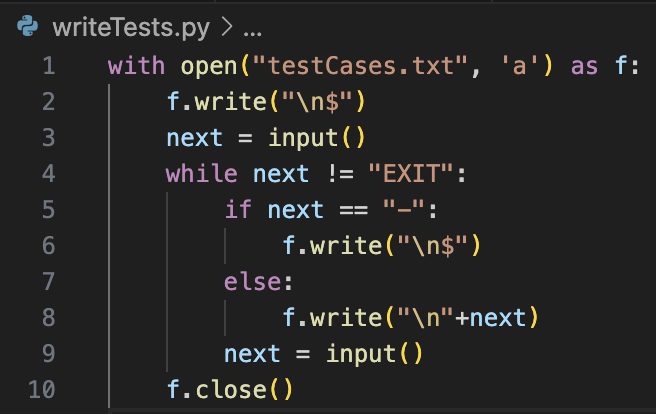
All the success criteria from modules 1 to 6 have now been implemented, and, as far as I am aware, the individual aspects of each section work perfectly as intended.

The system is still not in a very usable place, mainly because input() and print() haven’t been implemented yet so it’s not a very realistic test case. This is because every expression is outputted and they are always predetermined, which is different to normal ERL questions which require lots of inputs and outputs.

Therefore, I feel that beta testing at this stage is not very suitable, because it will cause a lot of confusion due to its incompleteness. However, with some guidance from Tanish, I will be writing a lot of test cases to see how these different aspects are currently working.

### test case evaluation

A screen shot of a computer program

Description automatically generatedI wrote a quick python file which would allow me to quickly write a lot of test cases alongside their intended responses. I would quickly distinguish these by typing a dash in the program, however for the file this would separate each code and answer with a $ symbol on lines between them, as it is never used in the ERL syntax.

Then, back in JavaScript in a copy of the source code, I changed the code so that any outputs would not be console logged and would instead be pushed to an array.

A black background with text

Description automatically generatedMy test case code therefore opened the testCases file, split it based on the dollar signs, and would then run every other section of plaintext and compare it to the intended output. I also changed all error messages to output “error” in this version so that I could write test cases which were meant to result in errors occurring.

The testing program also will count the successes, output any failures with details about them to help them to be fixed, as well as giving an overall success rate at the end of testing.

After doing a basic test of this new system to ensure that it worked, the task then changed to writing a lot of test cases for it to use. I made sure these covered a wide range of features, starting with testing a lot of the basics, but afterwards changing.

A computer screen with white text

Description automatically generatedA black background with white text

Description automatically generatedA screenshot of a computer

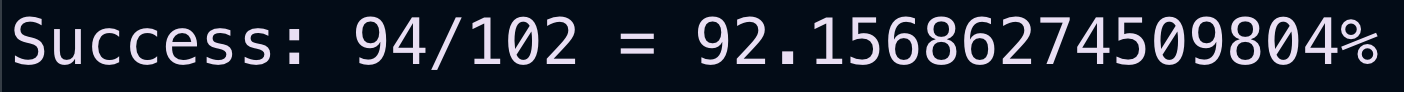
Description automatically generatedA screen shot of a computer

Description automatically generatedA screen shot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generatedBecause the purpose of this testing is to ensure that the separate modules integrate, I tried to mix multiple different modules together in multiple test cases. I also added some strange and very unusual cases to see if the Interpreter could cope with them, even though they would likely never come up.

When trying to run for the first time, one of the cases completely crashed the JavaScript engine completely, so I had to temporarily change the testing algorithm to print out each test first to find the one that was causing the issue.

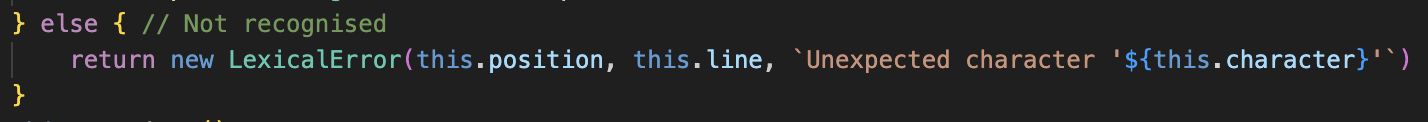
A white text on a black background

Description automatically generatedIt turned out to be the unclosed string, which had not been accounted for, which I temporarily deleted from the test case to check the overall performance.

Overall, this percentage was not as high as I’d have liked it to be, however when looking at the cases which returned errors, it was none of the complicated structures which involved multiple different modules but seemed to be a few small different features which were broken and I had to now go and fix.

# corrections

### unclosed strings crashing the program

A computer screen with text on it

Description automatically generatedFirstly, I had to rename UnexpectedCharacterError to LexicalEror, as I needed a more genereic name, and it needed to be updated to accept a description rather than a character.

A black background with text on it

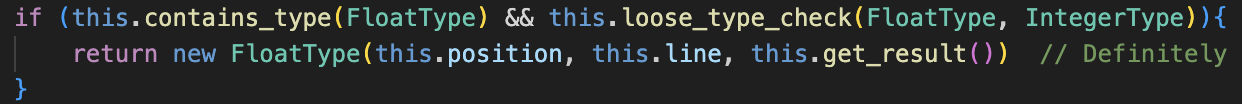
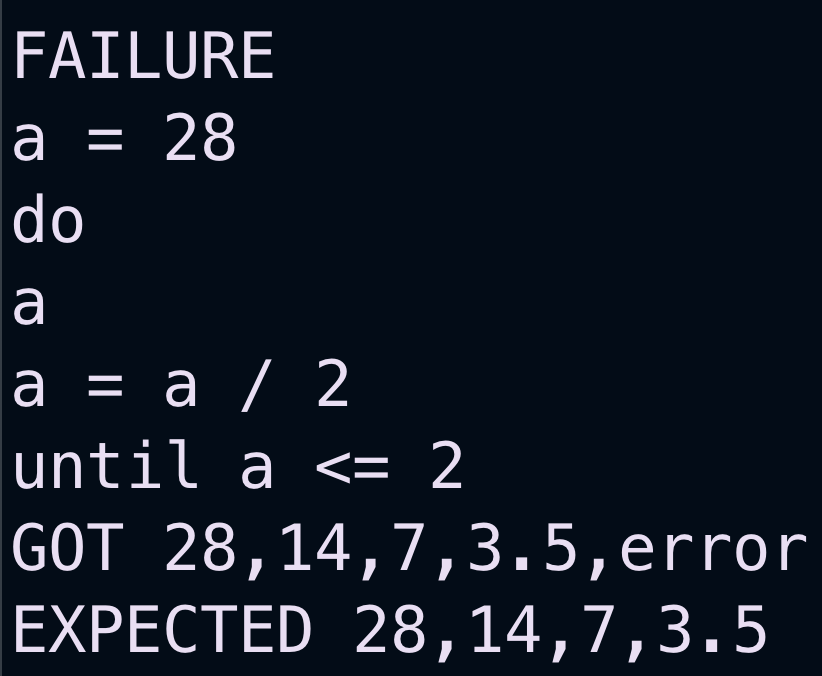
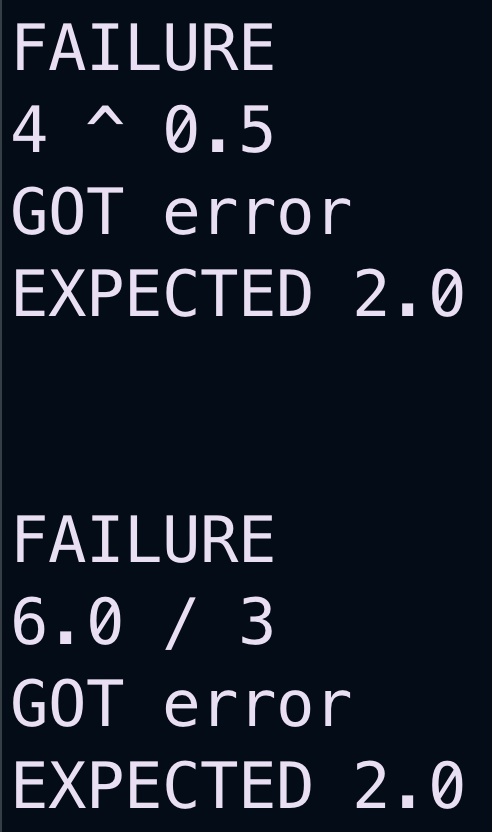
Description automatically generatedA computer screen shot of text

Description automatically generatedA black screen with white text

Description automatically generated

From here, make\_string() is updated to ensure it stops iterating if the current character is null, and that a Lexical error is returned if this is the case. Now when running a test case, it returns an error as expected so it can now be added back to the main testing document as it will no longer crash the program.

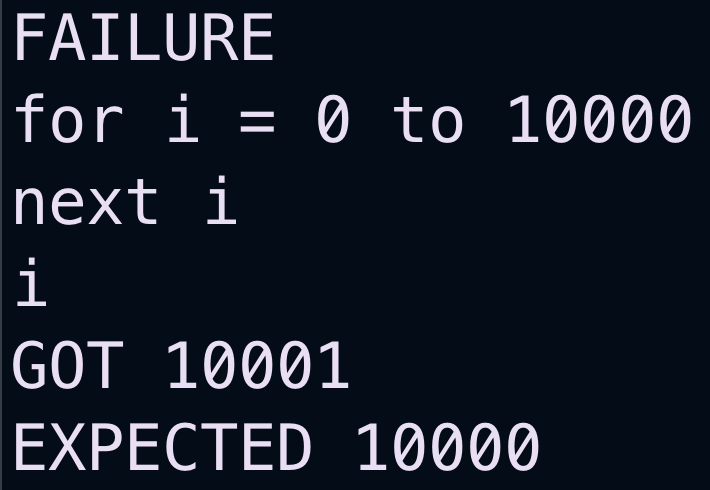
### float type handling

The next error was that using floats in either division or raising to powers returned an error for some reason. This ended up being completely down to a missing return keyword:

Now that this has been fixed, they work as anticipated, increasing the success rate of the testing to above 95%.

### for loop issues

A black background with text

Description automatically generatedA black background with white text

Description automatically generatedThe two issues that occurred with for loops is that the incrementing variable could be declared as a constant, and no error would occur, and that the variable would leave the for loop as one greater than its expected value after the loop had completed.

A screen shot of a computer code

Description automatically generatedThe first issue was fixed in build\_for\_loop() by adding a check if the current token is a Template Keyword before calling assignment, and if it is returning an error.

The second issue required some rearranging of the evaluate\_condition() method to only increment the variables value if another loop was occurring. I used a new variable to represent this, as changing variableValue would change the variable because it points to it, but this works as intended.  
Now the success rate was 97% and there was only a single type of error remaining.

### atypical bracket usage

A black screen with white text

Description automatically generatedA computer code with text

Description automatically generated with medium confidenceA black background with white text

Description automatically generatedThe first and last of these bracket issues are failures, however I feel like the empty set of brackets should return an error and the test case is incorrect. However, currently there is no specialised error message for an empty set of brackets, so I added a check into parse\_brackets() to ensure that this case was properly caught and an accurate error message was shown.

When doing some further debugging, the issues with the other brackets was not just in those cases, but for any arithmetic expression that began with a bracket. This is because in half\_statement(), it will parse the brackets presuming that the contents inside will be logical, however if it is arithmetic, like it is in these test cases, then it will not parse correctly.

Therefore, half\_statement () will need to be changed to account for this.

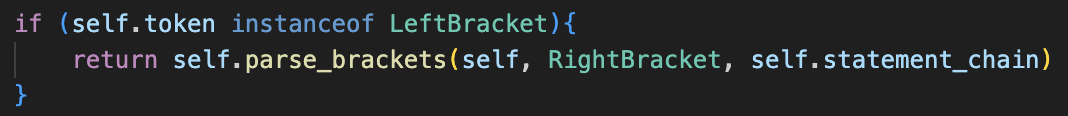
A computer screen shot of text

Description automatically generatedFor a while, I didn’t know how to solve this but I decided to look at the BNF for a different programming language at<https://craftinginterpreters.com/appendix-i.html>

This made me realise that, in my equivalent, brackets should only ever be parsed in factor(), and instead of calling expression() they should call statement\_chain(). The reason that I couldn’t do this previously was because it would lead to a lot of data type clashes, but now that I have proper handling this is no longer the case. Therefore, I will have to redesign statement() to not check for brackets any more.

This ended up significantly simplifying the code, and I completely removed half\_statement() as it can all be contained within a few lines. Previously I had too many checks, and when thinking about the BNF of statement it helps to simplify what to focus on.

<statement> ::= <expression> | <expression> Comparison <expression>

A screen shot of a computer program

Description automatically generatedInstead of all the complicated error handling that was present previously, I just had to call expression(). If a comparison didn’t follow, then I return that and otherwise I call expression() again and return the Comparison with the two expressions as its children. This much simpler solution works.

From here, factor() must be changed to not call expression(), and to instead call statement\_chain(), and all the type handling is now managed in evaluation so no errors can occur.

I also changed parse\_brackets() to no longer check for the starting bracket, and instead did that in factor() as I would like to minimse the number of arguments that need to be parsed.

From here, the cases worked as expected.

### testing complete

Now, all of the test cases are complete, so I am very content that the system is in a good place for the second half.