missing features

Before advancing to the mid-way evaluation, I feel it is important to have completed the success criteria that are not currently complete, which are 1.1 and 3.3. Currently missing are the Modulus and Quotient arithmetic operators, as well as the Logical NOT boolean operators, which all must be added.

# design

### modulus and quotient

A black text on a white background

Description automatically generatedThese will be new classes extending Binary Operator, and they will be created in make\_identifier() by the keywords “MOD” and “DIV”.

A screenshot of a computer

Description automatically generatedPython treats their equivalents as having equal precedence to multiplication and division, and because ERL is typically very similar, I will follow this. Therefore, they just need to also be iterated through in term(), which is very simple and just requires the parse\_binary\_operator() arguments to be altered to include them in the array.

🡺 calculate()

Both operators will accept Integers and Floats, returning a float value if either parameter is a float, and Integer if they are both floats, like the other operators. The get\_value() of Modulus will be using the JavaScript operator on the left and right values, and Quotient will use the floor of the left over the right.

### not

A white rectangular box with black text

Description automatically generatedThe new Not class will only have a single child, so will extend Token instead of Binary Operator. Therefore, it will have a property called child to store the AST of its child. It will also be created in make\_identiifer() with the corresponding string with the keyword “NOT”.

Python treats NOT as a higher precedence than AND + OR, but lower than the comparators, so a method to parse NOT will be required between statement() and statement\_chain()

🡺 not\_statement()

This is a new method, which statement\_chain() will now repeatedly call instead of statement(). It will check if the current token is Not. If it isn’t, then it will return statement(). If it is, then it will call itself, and make this new not\_statement() call the child property of the Not token before returning it.

🡺 evaluate()

The method to evaluate Not will call evaluate() on its child, and check if it is an error. If it is, then it will return it. It will then ensure that its data type is a Boolean. If it is not, then it will return a Type Error. It will then return the not value of the old Boolean value inside a new Boolean data type.

I think this is all the code that will be needed to implement these features, because the foundations have already been built it should be straightforward.

# development

## A screen shot of a computer screen Description automatically generatedmodulus and quotient

A black background with text

Description automatically generatedThe calculate() methods are essentially copies of the Multiply calculate() method because they contain the same checks, but a check that the rightValue is 0 is also required at the start, because – like Division – you cannot take the modulus or quotient of a number by 0. I also changed the Division class because it was checking that the leftValue was not 0, when it is meant to check the rightValue.

A screenshot of a computer

Description automatically generatedFrom here the other changes are very easy to make.

A computer screen with white text

Description automatically generatedAs shown by the test cases, this completely worked as expected. Sometimes with floats some weird results are created, such as by 3.7 MOD 0.6, but this is expected by floats due to how they divide so is okay.

This is all the implementation required so now I can move onto adding NOT.

## not

Even though NOT does not extend Binary Operator, I have placed its class alongside AND + OR in the program as that is where it makes most sense to be.

Like equals(), I have decided to not use a calculate() or get\_value() method for this class and have written all the code in evaluate() as it is a lot simpler than some of the other classes.

A screen shot of a computer program

Description automatically generatedA computer screen with colorful text

Description automatically generatedThe code for NOT is shown on the left and follows the plan. I have also named the result of the child being evaluated childValue, to be consistent with how leftValue and rightValue are used in the other methods, however here it is a variable not an attribute as it is not required.

A black background with blue and yellow text

Description automatically generatedThe addition to the Lexer was an extremely easy change.

A screen shot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedThe parser change was very alike how unary operators are treated in factor(), and I made sure to include a check for null if it occurred. Otherwise, it follows what the plan described, and returns the result of statement() if the token is not Not. However, when running a test case, the program crashed.

A computer screen with colorful text

Description automatically generated

A black background with white text

Description automatically generated The issues ended up being because self was not being passed to the next call of not\_statement(). There were also some small syntax errors in evaluate() which have now been fixed.

Now the test case responds correctly, showing that the new NOT operator works as intended. This should now mean that all of the content from the first six modules is complete, so now the half way evaluation can occur to check I am on the right track with the project.

### quick naming change

A screenshot of a computer program

Description automatically generatedThe current class that consists of the comparators, “==”, “<”, “>=” etc is called Logical Operator, which is a bad name as AND, OR and NOT are the Logical Operators, so I have renamed the class to ComparisonOperator to reflect their purpose more accurately.

This also had to be updated in the Parser and Lexer, but was easy with the replace feature, and after running some tests they still worked as expected.

# evaluation

The success criteria for the two missing features have now been implemented:

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Implemented? |
| 1.1 | Allow for Addition, Subtraction, Multiplication, Division, Exponentiation, Modulus and Quotient division to be taken of two Integers | **Yes** |
| 3.3 | Let NOT be used to negate comparisons and Booleans, with a lower precedence than comparison expressions. | **Yes** |

This means that sections 1 to 6 of the project are now complete in terms of development, and now the half-way evaluation should hopefully help to provide some testing to ensure they are perfect.

Overall, these were both very easy to implement, because there was a lot of existing infrastructure already in place, which reflects positively on how I have coded the Interpreter to be easily adapted.