Logic

# Design

A white and black rectangular box with black text

Description automatically generated with medium confidenceThis section involves implementing the features that are shown on the right.

This is mainly to build a foundation for the following modules, as they will all require this in order for conditions to be parsed in structures such as loops and if statements, so I need to add logical operations first.

### boolean

A new class, extending token, with position and value attributes. Its value will either be true or false and its evaluate() method will return itself, just like the other data types. It will need to be implemented into make\_identifier() in the Lexer, where “True” and “False” will represent it, with the capitalisation being important, as laid out in the ERL.

### Logical operator

This class will extend Binary Operator. This will be a class which will be inherited by lots of others: Equality (==), GreaterThan, GreaterEqualThan, LessThan, LessEqualThan, NotEqual. These will be made in the make\_tokens() Lexer method by their respective symbol, as laid out in the ERL in the top.

🡺 evaluate()

The evaluate() method will compare the results of the evaluate() methods of the children, checking for errors. If there are no Errors, it will then return a boolean, with the value depending on the type of operator, which should be self-explanatory based on the types.

### and + or

New classes extending Binary Operator, which has two children which must evaluate() to Boolean values or an error will be returned. If they are both Boolean, then they return the corresponding Boolean based of their type. They are represented by capitalised “AND” and “OR” in make\_identifier() for creation.

## Parser upgrades

A screenshot of a computer program

Description automatically generatedI feel that the best way to picture the new additions is a continuation of a diagram I’d used previously.

Essentially, logic is just an extension of the arithmetic system.

### statements

The first step of this builds upon expressions, and I have decided to name it as a statement. This has two possible cases: either being a single expression, or two expressions joined by a single Logical Operator, which is a comparison of the two values. This is the only definition which does not repeat like the others, and I have decided that you will not be able to chain these comparisons.

🡺 statement()

More technically, the BNF for this new method will be:

<statement> ::= Boolean | <expression> | <expression> LogicalOperator <expression>

This is as I described, but an additional case will have to be added so that a statement can also be defined as a boolean. This is because the stage above this will be joining these statements with ANDs and ORs, and booleans can also be joined by these, so they share the same level of precedence as statements.

Hopefully this shouldn’t be too complicated to implement, however it will have to also include lots of different error cases, such as when a LogicalOperator is not followed by anything, however I will run different test cases in development and add errors to any that cause issues.

### statement chains

Then, the top layer of the project is chaining these statements together with ANDs and ORs, and unlike statement(), there is no limit on how many times this can happen. The new method will become the new default one to be called by parse(), instead of expression()

🡺 statement\_chain()

Hopefully, this should just be as simple as calling parse\_binary\_operator(), with the nextFunction being the statement() method, and the valid tokens being And and OR, the new operators.

### brackets

Different logical statements must also be able to be used within brackets, to specify the order of operations with AND and OR.

# development

## statements

To begin with, I will only include the Equality (==) comparison and will add the others later.

### A black background with colorful text Description automatically generatedA screen shot of a computer program Description automatically generatedlexer

A computer screen with text

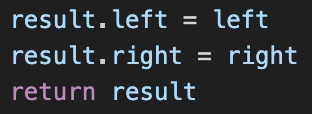
Description automatically generatedI am temporarily using make\_equals() to create Equals and Equality tokens, before I add the method which will cover all of the different Logical Operators.

A screen shot of a computer code

Description automatically generatedA screen shot of a computer code

Description automatically generatedThe test case correctly distinguished the two operators as shown.

### statement method



This was my initial attempt at creating the statement() method. Initially there were a few errors, but I fixed them as shown by the arrows.

A screen shot of a computer

Description automatically generatedA screen shot of a computer code

Description automatically generatedThe results of the tests show that Equalities now work as expected, alongside assignments.

A computer screen shot of white text

Description automatically generatedI also changed parse() to call this new statement() method by default, as expression() is built into statement(), so can still be used.

A more complex case is shown on the left, and it shows how the expressions have been correctly built into the correct abstract syntax tree.

Currently there is no error handling, however I am going to add the different invalid cases later.

Now I need to implement the evaluate() method, so that the Equality token can actually be used to return a boolean value.

A black background with white text

Description automatically generatedA computer screen with colorful text

Description automatically generatedThe new Equality evaluate() method follows the format of the other Binary Operators, returning the correct boolean based on the values on the two sides.

However, it was outputting the JavaScript version, with lowercase true and false, instead of my own with capitalisation. This may not seem like a large issue, but I feel like consistency is very important, and that there are some other issues with the current data types which could be fixed.

## fixing evaluation datatypes

Currently, in the chains of evaluate() calls in the syntax tree, the default JavaScript data types are passed between the different nodes, instead of my custom data types. This therefore loses a lot of information about the type itself, and because JavaScript has no separate Integer or Float data types, the details about whether a number used to be an Integer or a Float are also lost, and it is important to maintain this.

### float addition

A computer screen shot of a black background

Description automatically generatedA computer screen with text

Description automatically generatedA new method check\_for\_float() will take in two parameters, and check if either of them are the Float data type, and if they will return true. This method is in the Token class so will be able to be used by lots of different classes.

A black background with white text

Description automatically generatedAs seen in Add’s evaluate() method, this is then used to determine whether to return a Float or an Integer, so therefore the old types will be maintained, with the Float class taking priority.

After a few corrections, this worked as intended, and now the Data Types are kept consistent. This is important now but also builds the platform for type casting later in the project.

A screen shot of a computer program

Description automatically generatedNow a new method for data types called display() will be required, in order to specify what should be outputted for each type.

A black background with white text

Description automatically generatedFor Integers, this is simple by using the JavaScript typecasting on the value. For Floats, we must ensure they contain a “.0” at the end - if they are a whole number - to distinguish them. The code on the left is how this was implemented, and the temporary code runner now must call display() on the result and output that.

The test cases for this worked for a single operator between two values, and it shows how the different types are maintained, however as soon as more than 1 operator is used in an input, the system breaks.

### multiple operations

The main issue is how .value is treated. Now, the program is checking if the type of the left and right children, however the evaluate() calls are still returning the raw values. This means that Integers return their value, and that binary operators directly use the result of the evaluate() child calls.

A black background with white text

Description automatically generatedA black screen with white text

Description automatically generatedTo fix this, evaluate() must return the data type itself. Therefore, the evaluate() calls of Integer and Float must now return themselves, instead of their value, and the calculations must be performed on left.value and right.value: which are the values of the evaluate() results.

It took me a while to figure out these changes, as there a lot of different things to consider. As said, the calculation occurs on the values of the left and right, however the float checking still occurs on the left and right values, because this is what holds the actual type itself.

I also had some worries about how this would work with the variable system, however it didn’t require any changes to the system, and it now worked perfectly as expected.

### A screen shot of a computer program Description automatically generatedimplementing for all operations

The difference with the other operators is that two Integer values can result in a Float value being returned: for example, 7 divided by 5 does not give a whole number. Therefore, our check\_for\_float() method needs to be expanded to do this additional check.

Now, check\_for\_float() will iterate through the arguments. Alongside continuing to check for instances of Float, if they are a JavaScript Number instances (when the result of the calculations can also be passed), they are type casted to a string, and if they contain a full stop, it means that it is a Float so true must be returned.

A computer screen with white text

Description automatically generatedWhen I ran my tests, at first the float detection did not work, so I had to replace the Number instance check with one that I researched, and afterwards it worked.

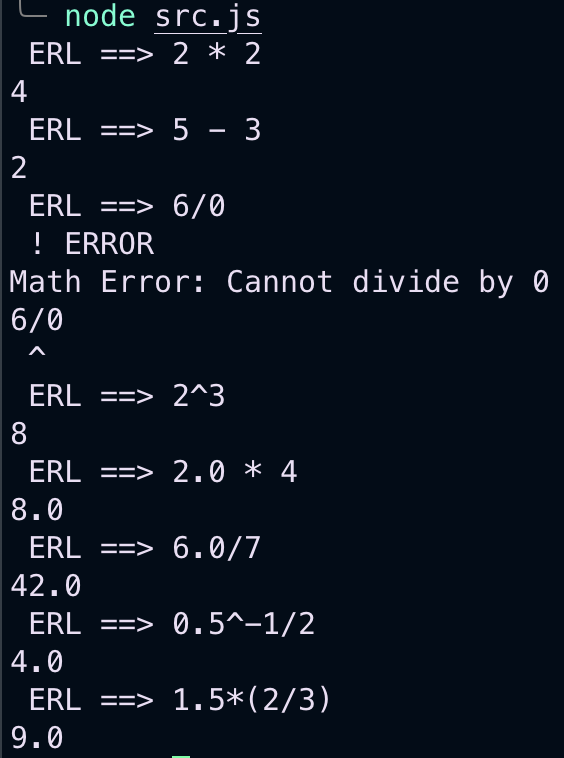
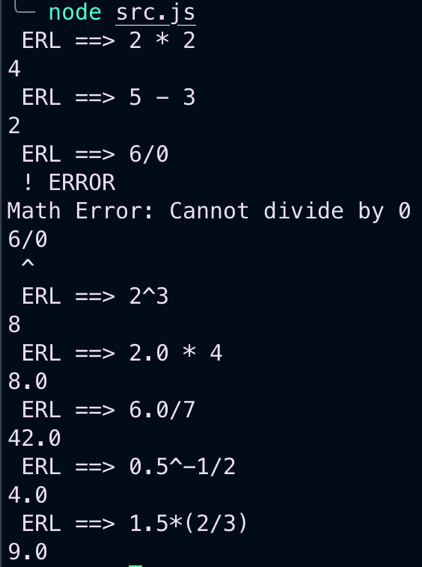
Above is the new implementation of the calculation, reusing the result temporary variable to hold the value of the operation so it does not have to be performed twice.

Now, the additional checks need to be re-added to Division and Exponent to ensure that the Math Errors are still properly implemented.

A screenshot of a computer

Description automatically generatedA screen shot of a computer program

Description automatically generatedThe format is on the left, and the check for the value will occur before the check\_for\_float() stage.



As shown, the series of small tests worked as expected, and the value of the float was always retained.

### general improvements

A screen shot of a computer program

Description automatically generatedA screen shot of a computer

Description automatically generatedNow that I know about the JavaScript arguments feature from using it in check\_for\_floats(), I can refine a few of the pre-existing methods to also use this feature. The main use of this is check\_instance().

This is a much cleaner solution, especially when only one instance being checked, because I did not find that having a single class within an array was a very logical and easy to read solution.

## A screen shot of a computer code Description automatically generatedbooleans

A screen shot of a computer program

Description automatically generatedThe new Boolean Class has a capitalised display() method to keep consistency, unlike the old system which had the lowercase booleans.

A screen shot of a computer code

Description automatically generatedThe Equality evaluate() method is also changed to return the Boolean class, and to compare the values of .left and .right, which is consistent with all the other evaluate() methods.

My initial idea for implementing Booleans into statements was to define them as an alternate definition of expression, however I realised this would not expand well, so I stuck with the initial BNF idea of defining them withing statement().

This now meant that statement() would have to be significantly changed to account for all the cases where booleans are included. The purpose of this is that otherwise the brackets in factor() to call expression() may allow for a Boolean to be included inside the brackets, which is something I want to avoid.

A computer screen shot of a program code

Description automatically generatedA computer screen shot of a program code

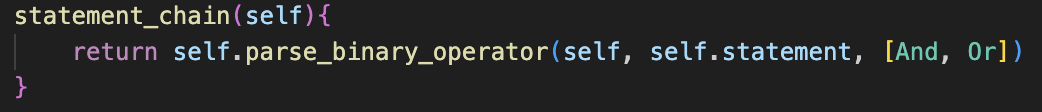
Description automatically generatedInstead, the statement() expression will handle the Booleans as shown on the code on the left, with additional cases on either side of the comparison.

This will still need to be expanded later to include statements() being able to be defined as just a Boolean, as well as needing brackets to be implemented, however for now it allows two booleans to be compared, as shown by the case below.

A black background with white text

Description automatically generatedAn error with the code on the left is that all the “this” keywords should be replaced with self, which I did after the screenshot to ensure it references token correctly.

## A screen shot of a computer program Description automatically generatedA screen shot of a computer program Description automatically generatedand + or

The two base classes extend Binary Operator and have obvious evaluate() methods: error checking their left and right and returning a Boolean based on their states.

Then, I added the planned parser methods.

A screenshot of a computer code

Description automatically generatedThis includes another change to parse(), to ensure that statement\_chain() is the default method.

There was also an issue in my initial evaluation method due to the recent changes, and the calculation was not being performed on the values, which it now does.

After this addition, the different statements can now be chained together, so long as they are just arithmetic comparisons, however the system breaks down as soon as Booleans are used.

A screenshot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generatedFixing this involved an extra check when the token following the left half was not a Logical Operator. If it wasn’t, and a Boolean was passed then it can be returned, otherwise an error still occurs, because an operator is expected.

This worked as expected, as shown with the test case on the right. I do not love how this is implemented, as it feels messy, with lots of different checks, however for now it works.

### QUICK ERROR FIX:

A screen shot of a computer screen

Description automatically generatedEntering a lone binary operator would crash the program.

A screen shot of a computer program

Description automatically generatedA black background with colorful text

Description automatically generated. Add the check to parse() to ensure it does not crash.

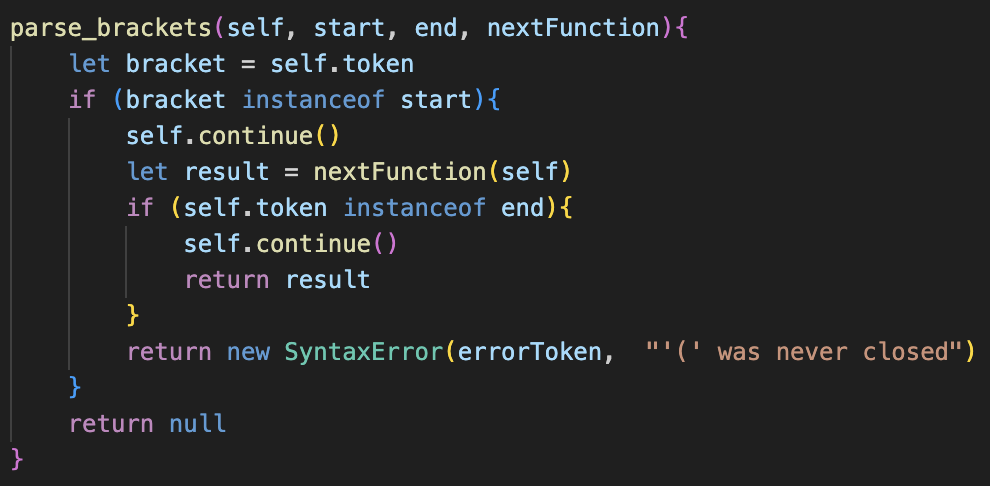
A screen shot of a computer code

Description automatically generatedThis check was also added to the left and right side of statement(), to ensure that  
2 == \* would not cause the program to crash.

Tests show that the error handling works, so now I can continue onto brackets.

## parsing brackets

This method from the Design, as well as parsing brackets, checks if they are there.



If it returns null, this means that brackets are not present. If not, they were present, and it returns the result of the nextFunction method,

If the token was now the same as the end token, then it can be returned properly, otherwise the brackets weren’t closed so an error is returned.

A black screen with green and white text

Description automatically generatedThis can now replace the bracket code in the factor() method as shown, and this element can be repeated.

A black background with white text

Description automatically generatedChecking if the result of the call is not null is the check for whether brackets were present or not. If it was null, then it continues through factor(), otherwise it returns as it knows the brackets have been used.

As shown, this can also be used within statement(), as shown on the left.

This has been implemented into the left and right side of the statement (I have only shown the left above), and the first check is now to check for these brackets, before checking for the other cases.

A black background with white text

Description automatically generatedOne issue now is that statement() is now getting long and repetitive with these further additions, because all the code is being identically repeated on the left and right side.

A computer screen shot of a error message

Description automatically generatedA black background with white text

Description automatically generatedThe test cases all worked as expected, and old expressions and new logical statements both correctly function with this new generic method for parsing brackets, which is good as it avoids lots of repetition.

A screen shot of a computer code

Description automatically generatedA black background with white text

Description automatically generatedInitially, assigning variables to statements would cause an invalid error to be returned, but after changing assignment() to call statement\_chain() instead of expression(), it then worked.

A screenshot of a computer program

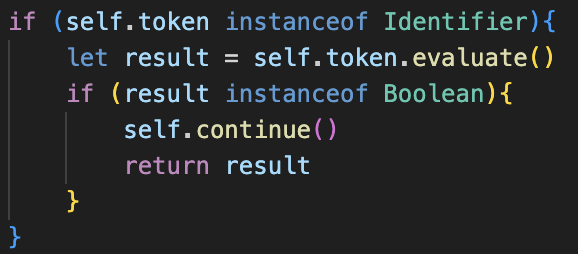
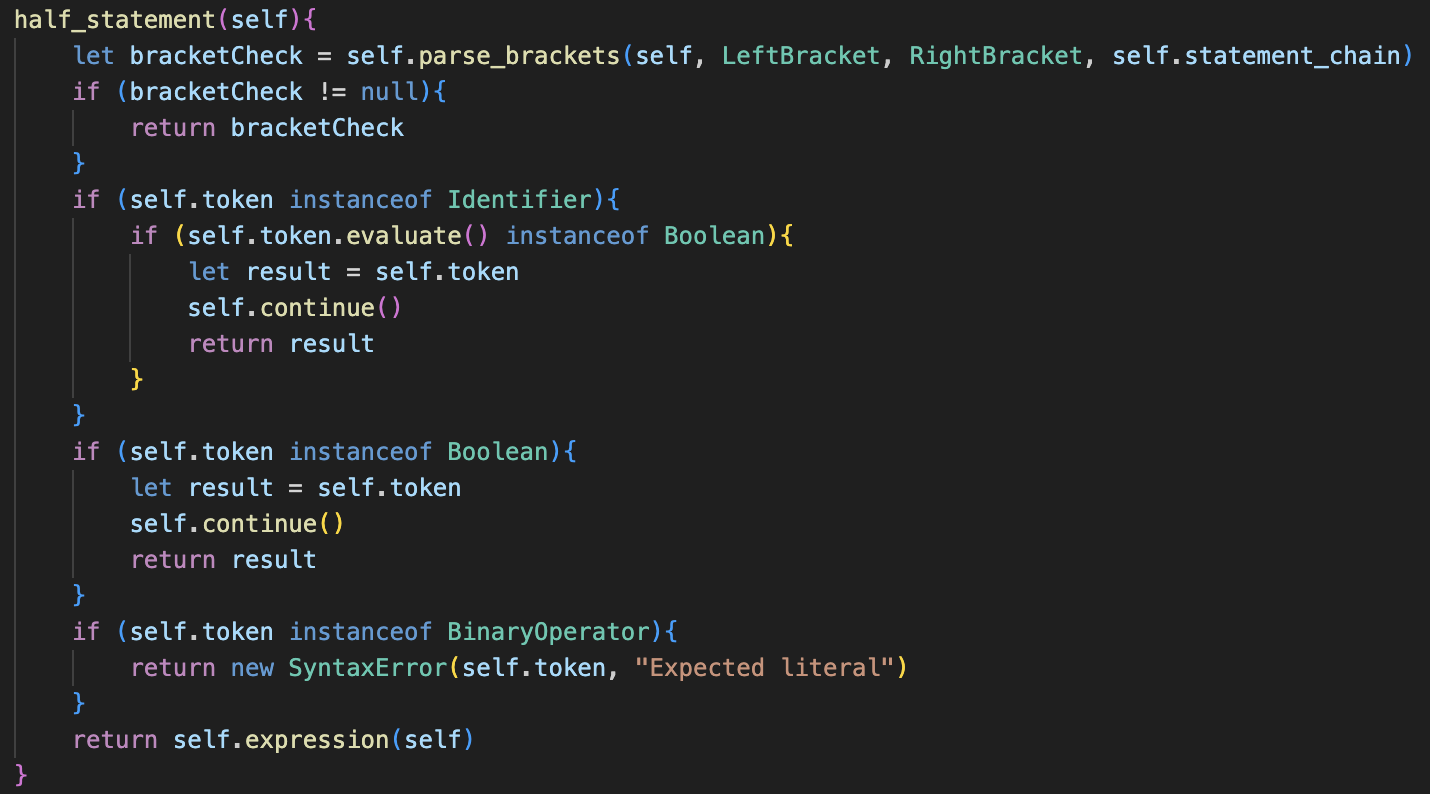
Description automatically generated  
Even though booleans could now be stored into variables, they could not be used in the same way as plainly written booleans could, because in my parser, I did not account for the case for when a statement() could be defined as a single boolean stored in an Identifier.

This will therefore require further changes to the statement() method,

## statement update

A new method called half\_statement() will be implemented, to reduce code repetition on the left- and right-hand sides of statement(). This will therefore contain the bracket check, checking for Booleans, checking for lone Binary Operators, and calling expression()

This implementation is not the most ideal, mainly due to the evaluate() call to check which data type the variable is.



This seems like an unnatural approach, but it is a working solution, as currently, as just shown, having booleans stored in variables does not work.

I however feel like this system may break down later on, when multiple lines are run at once.

However, I am planning to delay any upgrades to this to the types module, where I will be introducing strings. My plan for this module is to reinforce very strict type handling for the whole system, and hopefully I may be able to reduce the complexity in the parser, by adding more type checking throughout.

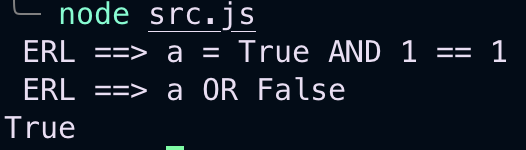
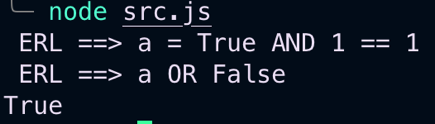
However, for now, I am sticking with slightly worse solution.

It is not all awful, and will work currently for the current example, but for the long-term maintenance, it is definitely not a good approach,

A screen shot of a computer program

Description automatically generatedThe updated statement() method uses this new method on either half, and as shown it significantly simplifies the whole process, and reduces the amount of repetition by a lot.

The test case for this is successful, and now booleans that are stored in variables can interact with the rest of the system.



Therefore this update, although slightly poor, is producing successful results, which is good.

### issues with brackets

A screenshot of a computer program

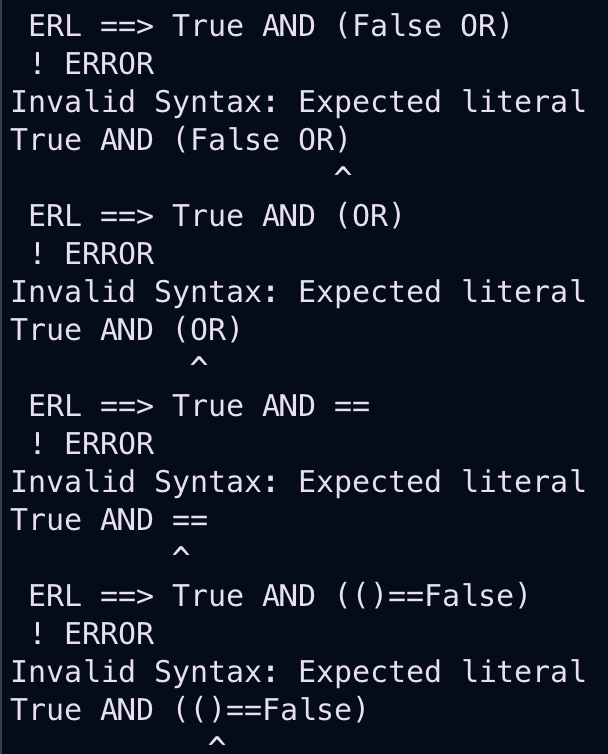
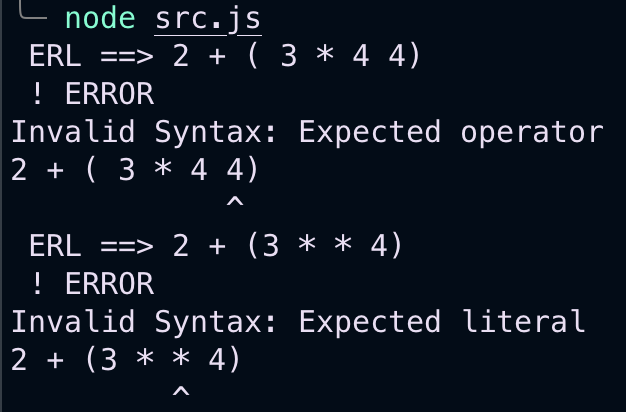
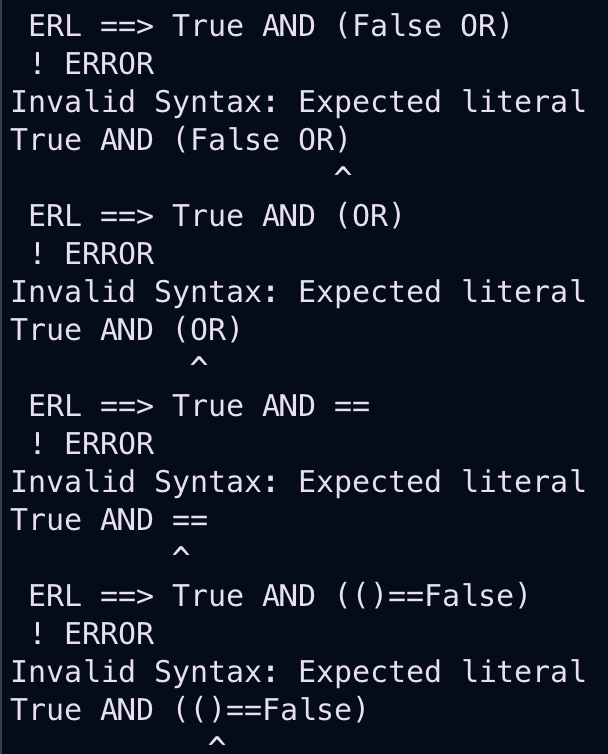
Description automatically generatedWhen running some tests, I notice that the new bracket system had a lot of incorrect Errors. There were two issues.

A screen shot of a computer code

Description automatically generatedFirstly, if an expression is completely wrapped in brackets, it will always incorrectly return a Syntax Error, as the end of expression() currently returns an Error if the next token is not null when it is finished, so this can be easily fixed by amending expression() to:

A black background with blue and white text

Description automatically generatedThe second issue is that actual Errors within brackets are never properly reported, and just return the unclosed bracket error message regardless. To fix this, there just needs to be a check to see if an error occurred when calling nextFunction, before checking for the end token.



As shown, the test cases now produce the intended result, including some of the really strange cases that I tried, and I am fairly sure with how many different cases I tried, so I feel this solution is very robust.

## other comparisons

The original plan was to have a different class for each different Operator, however that would have resulted in a lot of repeated code. Furthermore, all the different comparisons are treated as the same during parsing, so they only need to be distinguished in the evaluation stage. Therefore, I decided to combine everything into the single LogicalOperator class.

A screen shot of a computer program

Description automatically generatedThis new approach gives Logical Operator a tag property, like Template Keyword for consistency, which determines how it is evaluated in a long switch statement.

This approach allows all parsing code to be kept the same, with only updates to the Lexer being required for full implementation.

As shown, there is just a switch statement on the tag whenever it needs to evaluate, which determines the value of the boolean that needs to be returned.

A screen shot of a computer program

Description automatically generatedTo create these tokens in the first place, the lexer is going to be expanded with the new make\_logical\_operator() method, which will deal with creating these tokens.

A black background with colorful text

Description automatically generatedThis is called in make\_tokens() when the current character is >, =, < or !, as they are all the possible starting characters.

Then, if the next character is an =, it will return a Logical Operator with the two-character tag, being the initial character followed by the “=”, which will be able to cover all of the two-character comparisons.

A screenshot of a computer

Description automatically generatedThis is because it must be >=, ==, <= or !=, which are all valid operators. If it is not an = in the second position, we need to still account for all the cases for >, =, < and !. This is where the switch statement comes in, returning an Equals for =, an Error for !, and a Logical Operator for < or >.

A screenshot of a computer

Description automatically generatedThis relatively quick implementation immediately works, proven by the set of tests to cover a few different cases.

A black background with white text

Description automatically generatedI am happy with this method, as I feel like it is a very smart way to combine all of the different comparisons, instead of my original plans of having lots of checks in make\_tokens() and is a lot more logical.

### cleaning up

A screen shot of a computer code

Description automatically generatedI just changed the default evaluate() method in Token to return itself to avoid a lot of code repetition for the Data Types.

Overall, I believe that is this section of implementation complete, so now logical methods should be able to be integrated with the structures that will be added in the following modules.

# evaluation

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Passed? |
| 3.1 | Allow for Booleans to be created with “True” or “False” | **Yes** |
| 3.2 | Let the comparisons ==, !=, >, >=, <, and <= be used between two expressions, Booleans or values and return a Boolean | **Yes** |
| 3.3 | Let NOT be used to negate comparisons and Booleans, with a lower precedence than comparison expressions. | **No** |
| 3.4 | Let AND and OR chain multiple different expressions together, with a lower precedence than NOT. | **Yes** |

The success criteria for this module are as follows:

I forgot to implement NOT, which was a huge flaw in my Design plan as I forgot it was needed. However, I am not super concerned about this, as it should be very easy to add later when I come round to adding some of the missing features, alongside MOD and DIV which have still not been added.

The reason I am not rushing to implement this now is because I feel like a lot of what I’ve done in statement() will likely have to be re-worked later, as it is more of a temporary solution until proper type handling gets added, as I am hoping that could make the solution slightly cleaner.

Hopefully it should just be as simple as adding another layer of parse\_binary\_operator() in the correct place when I come to implementing it, however I will properly plan this implementation later.

### code quality

Being honest, I think this has been a very mixed module. Although most parts have been able to be added very cleanly and simply, the statement() method is absolutely horrific.

The whole method is full of lots of different cases where I am accounting for the different weird edge cases that I found that broke the system, as supposed to the rest of the methods, which all work quite elegantly.

Overall, simple solutions seem to yield the best results in this program, and I feel that statement() is such a mess of different cases, so I will aim to hopefully change it later. This is mainly because currently I am checking for all the types when I am in the parsing stage, which does work, but as soon as variables become involved there becomes an element of uncertainty as to its type, and therefore all types kind of must be treated as an unknown for the system to correctly account for all cases.

Currently, statement() does lots of checking into the cases, even going as far as to checking the value of any Identifiers, which is a poor solution. My aim is to hopefully move all this type checking into the evaluate() methods instead, which should hopefully clean up the solution.

### moving on

I know I have been very negative about this module, but my solution does work and the error handling is very good, and it builds a good foundation into moving into the different structures, which is when the project starts to become a lot more interesting and more like an actual language.