Logic

# Design

### boolean

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

### 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 and will have two expressions as children.

🡺 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 object depending on the type of Logical Operator.

### 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

🡺 statement()

The BNF for this statement will be:

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

The case for just <expression > is when there is just a Boolean or an Identifier referencing a Boolean,

# development

## equality statements

To begin with, I will only include the Equality (==) comparison

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

A screen shot of a computer code

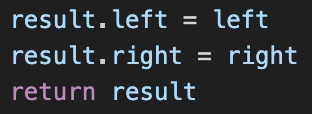
Description automatically generatedA screen shot of a computer code

Description automatically generatedA computer screen with text

Description automatically generatedBefore implementing a more complex method for all the Logical Operators, I have decided to use make\_equals() to distinguish between = and == in the plaintext string.

The test case correctly distinguished the two operators as shown.

### statement method



This was my initial attempt at creating the statement() method. I then ran a series of tests which resulted in a few small errors being corrected as shown.

A screen shot of a computer

Description automatically generatedThe results of the tests after the fixes show the logic works.

A computer screen shot of white text

Description automatically generatedA screen shot of a computer code

Description automatically generated

As expression() is built into statement(), this means that statement() is now the default method to call in parse() and it accounts for both expressions and statements.

This instance of statement() is not fully handled to deal with all errors, but is more of an initial design which definitely works for valid inputs so that error handling can properly be implemented later.

A black background with white text

Description automatically generatedA computer screen with colorful text

Description automatically generatedWhen implementing the relatively simple evaluate() method, the correct Boolean value was returned as shown.

However, this is using JavaScript’s data types rather than instances of the ERL classes, which is a problem for all evaluate() methods which return anything.

## fixing evaluation datatypes

The main issue with not returning the custom data types is with Integers and Floats. JavaScript only contains a generic Number data type, so there is no management which ensures that Floats remain as Floats, so this needs to be implemented into the Arithmetic Operators.

### A computer screen with text Description automatically generatedstarting with addition

A computer screen shot of a black background

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 and uses a separate method to reduce code repetition when used in different classes.

A screen shot of a computer program

Description automatically generatedAfter 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 black background with white text

Description automatically generatedNow a new method for data types called display() will be required, to print the string equivalent. For 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 and now data types are maintained for addition, but currently this would only function with a single operator due to how these data types are handled.

### changes

The main change is how .value is treated. Currently, in the abstract syntax tree, the raw value is passed along the chain of evaluate() calls. This means that Integers return their value, and that binary operators directly use the result of the evaluate() child calls.

I have changed the return of the evaluate() methods to pack the calculation result into the Data Type, but they must also expect a data type in. This means that Floats and Integers must return themselves in their evaluate() method, and that Arithmetic Operators must perform calculations on this.left.evaluate().value (for example), instead of just the evaluate() method.

A black background with white text

Description automatically generatedA black screen with white text

Description automatically generatedA black background with white text

Description automatically generatedBecause I hadn’t fully understood how this change would work yet, I spent a long time trying to change the code in the Data Type class to get it to work before realising the change had to be in the arithmetic operators as explained.

The changes are mainly in the last line. We still check if left and right for their Datatype, as those are the instances of Integer or Float. Then, the actual arithmetic is performed on left.value and right.value and is returned inside the corresponding Data Type. Then, the final result’s display() method will be called, returning its string value so it can be represented.

### 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 (so the result of the calculations can also be passed), they 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.

A screen shot of a computer program

Description automatically generatedAbove 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.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedThis general format is repeated throughout all the different Arithmetic operations. For ones with extra error handling, such as division, they work as shown.

A screenshot of a computer

Description automatically generatedAs 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 instances of this is check\_instance().]

This is a much cleaner solution, especially when only one instance being checked, as having a single Class within an array is not as readable or logical as this solution.

The test cases shows that this still works.

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



A screen shot of a computer code

Description automatically generatedThe new Boolean Class has a capitalised display() method to keep consistency. The 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 decided this was a poor approach as it would lead to the mixing or arithmetic and logic which should be kept separate.

A black background with white text

Description automatically generatedA computer screen shot of a program code

Description automatically generatedA computer screen shot of a program code

Description automatically generatedThis would also cause issue with brackets calling on expression, so is not viable.

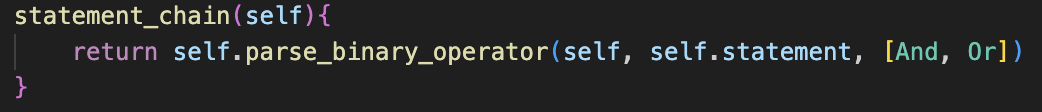
Instead, the statement() expression will handle the Booleans as shown on the code on the left.

This is split into three parts, the left, operator, and the right. 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.

An 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.



A screenshot of a computer code

Description automatically generatedThe statement\_chain() method is the new default case in parse(), and calls parse\_binary\_operator() on the statement() method as next method, based on AND + OR. Therefore, statement\_chain() also represents single statement() and expression(), by building onto each one of them.

Initially, it did not work, due to not using the value of left and right in the calculations.

This was an easy fix, and as shown on the right, this worked as expected.

A screen shot of a computer program

Description automatically generatedHowever, single Booleans could not yet be used directly alongside AND as they are not defined as statements by themselves.

A screenshot of a computer program

Description automatically generatedThis change involved an extra check if 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.

This worked as expected, as shown on the left.

A screen shot of a computer screen

Description automatically generated**QUICK ERROR FIX:**

A black background with colorful text

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

A screen shot of a computer program

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.

## A screen shot of a computer program Description automatically generatedparsing brackets

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

If it returns null, this means that brackets were not present, and if they were it would call 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

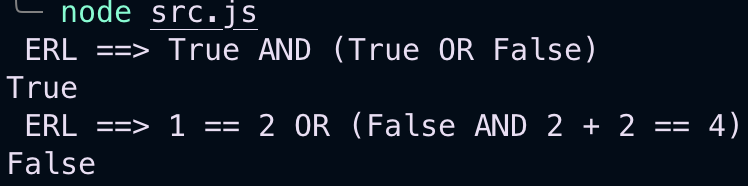
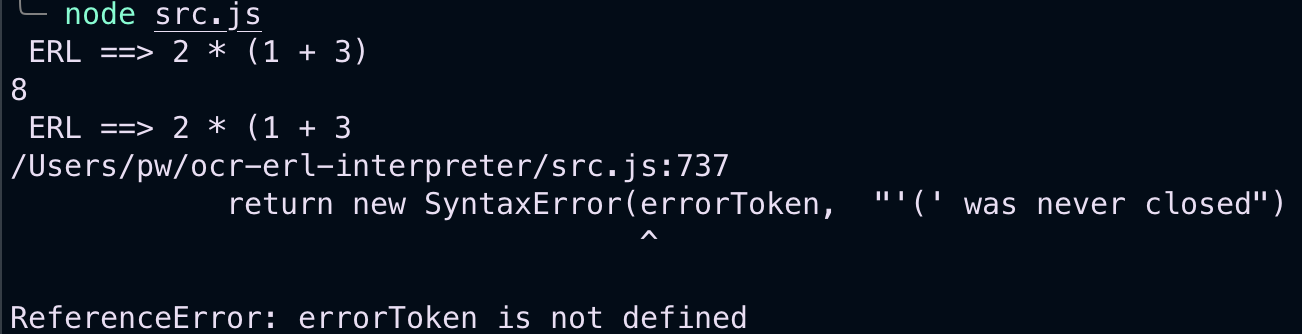
Description automatically generatedChecking if the result of the call is not null is the check for whether brackets were present or not. If it was null, then it continues through factor(), otherwise it returns as it knows the brackets have been used.

A screen shot of a computer program

Description automatically generatedThis has been implemented into the left and right side of the statement, and the first check is now to check for these brackets.

A black background with white text

Description automatically generatedStatement() is now getting long and repetitive with these further additions.



A black background with white text

Description automatically generatedThe test cases all worked as expected, apart from attempting to define a variable as a Boolean – or any other logical expression.

A screen shot of a computer code

Description automatically generatedChanging the assignment() method to evaluate the right-hand side of Equals for statement\_chain() fixed this issue.

A screenshot of a computer program

Description automatically generatedThis worked, and now variables could have Boolean values, however when trying to use them in certain ways the statement() code broke. This is because in our definition of a statement being a single Boolean, it did not account for Identifiers with Boolean values.

This therefore requires a redesign of statement() to account for this.

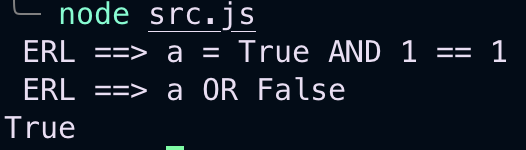
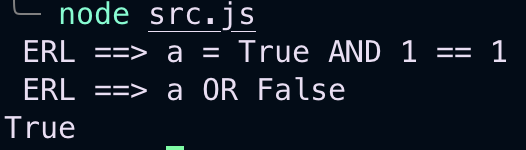
## statement update

A screen shot of a computer program

Description automatically generatedA screen shot of a computer code

Description automatically generatedA screen shot of a computer screen

Description automatically generatedA 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, so could be reimplemented later when more strict type handling is implemented, as currently I am worried that Identifier errors could lead to issues when trying to run certain invalid inputs.

### 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 wrapped in brackets, it will always incorrectly give 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

A computer screen shot of a program code

Description automatically generatedA black background with blue and white text

Description automatically generatedA screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generatedThe other issue is that actual Errors within brackets are never properly reported, and just give the unclosed bracket error message regardless. To fix this, there just needs to be a check to see if an error occurred in the parse\_bracket() method before checking if the current Token is the same as the end Token.

As shown, the test cases now produce the intended result.

## A screen shot of a computer program Description automatically generatedother comparisons

The 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.

This new approach gives Logical Operator a tag property, similar to 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.

A screen shot of a computer program

Description automatically generatedA black background with colorful text

Description automatically generatedAfter a few stages of changes, this was the final method design for making logical operators.

This 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 tag so far.

A black background with white text

Description automatically generatedA screenshot of a computer

Description automatically generatedA 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 >.

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

### cleaning up

A screen shot of a computer code

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

# evaluation

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Implemented? |
| 3.1 | Introduce a Boolean data type represented by “True” and “False” | Badge Tick with solid fillCompleted |
| 3.2 | Allow comparison operators ==, !=, <, <=, > and >= to be used to compare two arithmetic expressions, values, or Booleans | Completed Badge Tick with solid fill |
| 3.3 | Allow NOT to be used in front of statements or single booleans to alter their value, with a lower precedence than comparators | Badge Cross with solid fillNot implemented |
| 3.4 | Allow AND and OR to be used with a lower precedence than NOT to chain these statements together | Completed Badge Tick with solid fill |
| 3.5 | Ensure complete error handling, with incomplete statements and invalid use of operators resulting in errors | Completed Badge Tick with solid fill |

Overall, this section has gone well, with the success criteria shown below:

I forgot to implement NOT, which was a 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.

In terms of the state of the code, I am not happy that there is a lot of repeated code in the evaluate() method for each one of the operators and might rework them later on – must likely in module 6 where stricter type handling gets added. Overall though I am happy.