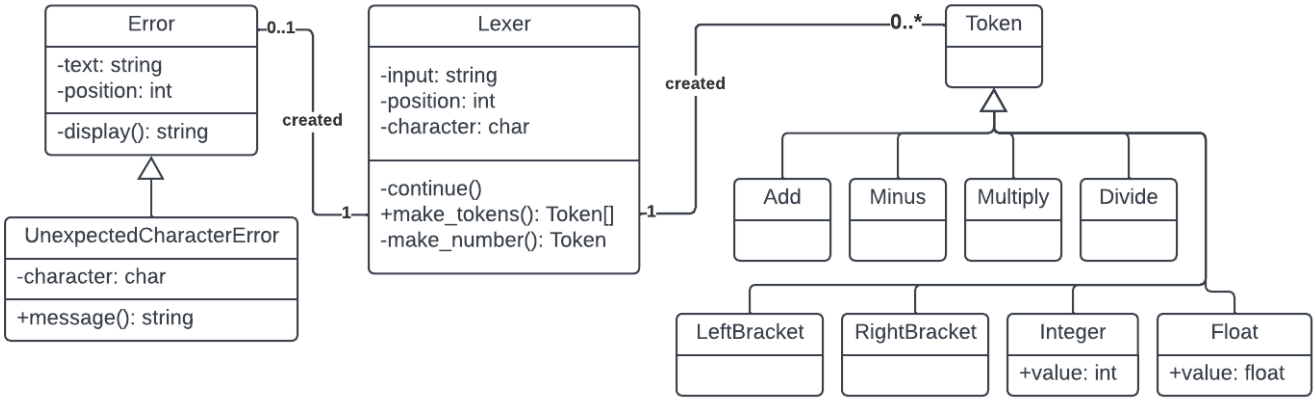
ARITHMETIC

DESIGN: LEXER

As I covered in the Analysis, the point of the Lexer is to convert a plaintext input, which the computer cannot understand, into a list of tokens which hold meaning to the program, which can then be arranged into abstract syntax tree to be executed.

Because I am using an Object-Oriented approach, I have made a UML class diagram which shows the relationship of all the different classes.

I will then go ahead and explain each new section that has been added.



LEXER

This is the main class in this section and will interact and link with all the others. It has an input property, for the plaintext, as well as a position and character property. Because the Lexer will function by iterating character by character through the plaintext input, these will keep track of the current position and character. All these properties will be assigned in the constructor.

➔ continue()

This method will be called to advance through the input string, incrementing the position property and changing the character property to the corresponding value. However, if the end of the string has been reached, then the character will be set to null. This should help to avoid with any errors with accessing a character at an index beyond the length of the string, as this is the centralised method to update the character and ensure any progressions through the string are allowed.

➔ make_tokens()

This is the method that will be called to generate tokens and will comprise of a while loop which repeatedly runs until the character property is null: until the entire input has been cycled through. For each iteration, the current character will undergo a series of checks, which will then, at the end, push the corresponding Token to an array of tokens. For example, if "+" is encountered, then an Add token will be pushed on that cycle. I will go through each of these checks in more detail later.

At the end of each cycle, the continue() method is called to proceed to the next character, or if the current character did not match any of the cases, then an instance of an UnexpectedCharacterError is returned because the program does not recognise it. Once all the characters are cycled through, the list of tokens will be returned, which will then be used by the Parser.

➔ make_number()

This method will be called if the character in make_tokens() encounters a digit, because it may contain multiple different characters within the same Token. Generally, whenever a multiple-character Token needs to be made, make_tokens() will call this method whenever the first character has been detected, and will then push the result of the function to the stream of tokens.

make_number() will then continually call continue() and add the characters to an array until a non-digit is encountered. Full stops will also be added and will increment the value of a fullStops variable. When a non-digit is encountered, it stops the loop and the method will return an instance of an Integer or Float Token, depending on the number of full stops: with an Integer being made if none are present, and a Float with one. The value will be the characters in the array, typecasted into a number.

An additional check will have to be made if many full stops are present, so if fullStops exceeds one, an error will have to be returned, because a number cannot be defined with more than 1 full stop.

TOKENS

There will be a basic Token class, and then a unique child class for each type of token that exists. At this stage, these classes will be empty, because their code will be based on how they should be evaluated, which will come later in this section.

Therefore, a lot of these classes are empty, except Integer and Float, which will have a property called value, which will, obviously, store the value of that number. This will be set in their constructor methods.

Now, I will go through each of the different tokens, specifying how they will be created:

Add ➔ "+" in make_tokens()
Minus ➔ "-" in make_tokens()
Multiply ➔ "*" in make_tokens()
Divide ➔ "/" in make_tokens()
LeftBracket ➔ "(" in make_tokens()
RightBracket ➔ ")" in make_tokens()
Integer ➔ A number with no full stops in make_number()
Float ➔ A number with one full stop in make_number()

For now, these are very basic, because the Lexer is only for arithmetic calculations, but in the future, when keywords, strings and other features will need to be added, the complexity will significantly increase.

ERRORS

This will be the generic, parent class for all the different types of errors in the program. It will accept the position of the error in its constructor, as well as the plaintext of the input with the error, stored in a position and text properties respectively. It will have a single method.

➔ display()

This is a method which will be used by all Error subclasses. It will use the text and position properties and return a two-line string of the line of text with an arrow "^" below. This will indicate exactly where the error occurred so that it will be easier for the user to precisely debug the issue.

UNEXPECTED CHARACTER ERROR

This subclass, alongside the position and text arguments, will also accept the character that called the error in its constructor. Therefore, whenever an error occurs, the Lexer's character property can be passed into the function, and this will be used for displaying the error.

➔ message()

This will return a string with an error message describing the issue for the user to understand, using the character that was passed to it to accurately describe the issue. This will include the result of the display() method at the end of this description, so that the visual position can also be shown.

In the future, all subclasses of Error will have a method called message(), so that whenever an error occurs, message() will be printed, and regardless of the error the issue will be outputted. This is an example of how polymorphism will be useful within the project, by having methods named the same between classes.

Overall, this should be the planning for the first Lexer stage complete. This first implementation should not be extremely complicated, but I need to focus on making the whole system very expandable, because everything else will build off this base lexer, so having the code be clean is very important.

DEVELOPMENT NOTES

Throughout this whole project, I am going to be pasting a lot of different screenshots as proof, and testing, so I will be colour coding the borders of images for reference.

No border ➔ This is code that I have written for the project.
Green border ➔ This is the code or output of a testcase that was successful.
Red border ➔ This is the code or output of a testcase that was a failure.

There may also be other, small fixes aligned over the code screenshots, which will be outlined in white. These are just small changes that originally caused errors, but I then fixed.

These borders will just break up the large number of screenshots and should make it easier to read through the write-up and recognise the purpose of each screenshot.

DEVELOPMENT: LEXER

THE LEXER

BASE CLASSES

```
class Lexer {
    constructor(input){
        this.input = input
        this.position = -1
        this.character = null
        this.continue()
    }

    continue(){
        this.position += 1
        if (this.position == this.input.length){
            this.character = null
        } else {
            this.character = this.input[this.position]
        }
    }
}

// Testing

lexer = new Lexer("this is a test")
while (lexer.character != null){
    console.log(lexer.character)
    lexer.continue()
}
```

I'll start by creating the basic Lexer class, which includes the continue() function to increment the position and change the character property when called. I also made it so it is called once in the constructor, so that the Lexer will begin in the first position with the first character.

```
[R
t
h
i
s

i
s

a

t
e
s
t

[D
```

Initially, this had a small syntax error, which you can see that I corrected. I then wrote a quick test case to check that continue() functioned correctly, and the loop correctly outputted all the characters, showing that the continue() method works as expected.

Then, I added each of the Token classes, which are now just all very empty but will be filled up later. I also gave them each a constructor method, which for now just called the super() constructor, but in the future should have additional code.

```
class Token {
    constructor(){
    }
}

class Add extends Token{
    constructor(){
        super()
    }
}

class Minus extends Token{
    constructor(){
        super()
    }
}

class Multiply extends Token{
    constructor(){
        super()
    }
}

class Divide extends Token{
    constructor(){
        super()
    }
}
```

```
class LeftBracket extends Token{
    constructor(){
        super()
    }
}

class RightBracket extends Token{
    constructor(){
        super()
    }
}

class Integer extends Token{
    constructor(value){
        super()
        this.value = value
    }
}

class Float extends Token{
    constructor(value){
        super()
        this.value = value
    }
}
```

```
// Testing
a = new Add()
console.log(a)
b = new Integer(53)
console.log(b)
console.log(b.value)
c = new Minus(5)
console.log(c)
```

```
[Running] node "/Users/pw/ocr-erl-in
Token Made
Add {}
Token Made
Integer { value: 53 }
53
Token Made
Minus {}

[Done] exited with code=0 in 0.205 seconds
```

The small test case just to ensure that the tokens were properly created, with the values of the Integer also being stored correctly, so now I am ready to generate some tokens.

BASIC TOKEN GENERATION

```
make_tokens(){
    let tokens = []
    while (this.character != null){
        if (this.character == ' '){
        } else if (this.character == '+') {
            tokens.push(new Add())
        } else if (this.character == '-') {
            tokens.push(new Minus())
        } else if (this.character == '*') {
            tokens.push(new Multiply())
        } else if (this.character == '/') {
            tokens.push(new Divide())
        } else if (this.character == '(') {
            tokens.push(new LeftBracket())
        } else if (this.character == ')') {
            tokens.push(new RightBracket())
        } else {
            console.log("ERROR: Unexpected Character: '", this.character, "'")
        }
        this.continue()
    }
    return tokens
}
```

I have implemented each of the cases by comparing the characters to their corresponding tokens. I will implement proper error handling later, but for now, I will just output a message.

```
// Testing
test = new Lexer("(+*-/)")
console.log(test.make_tokens())
test2 = new Lexer(" * +     /  m")
console.log(test2.make_tokens())
```

```
[
  LeftBracket {},
  Add {},
  Multiply {},
  Minus {},
  Divide {},
  RightBracket {}
]
ERROR: Unexpected Character: ' m '
[ Multiply {}, Add {}, Divide {} ]
```

As shown, the test cases for these two worked, with the whitespace being ignored and only the actual characters being registered.

NUMBER TOKENS

DIGITS is declared globally as a constant so it can be re-used.

```
// Constants

const DIGITS = [..."0123456789"]
```

```
    make_number(){
        let number = []
        let fullStops = 0
        while ((DIGITS.includes(this.character) || this.character == '.') && fullStops <=1){
            number.push(this.character)
            if (this.character == '.'){
                fullStops += 1
            }
            this.continue()
        }
        switch (fullStops){
            case 0:
                return new Integer(Number(number.join('')))
            case 1:
                return new Float(Number(number.join('')))
            default:
                console.log("ERROR: Too many decimal places")
        }
    }
```

The make_number() method is added much like in the design description.

```
while (this.character != null){
    if (this.character == ' '){
    } else if (DIGITS.includes(this.character)) {
        tokens.push(this.make_number())
    } else if (this.character == '+') {
        tokens.push(new Add())
```

The check in make_tokens() is also added, which will call the method when a DIGIT is encountered,

```
// Testing
test = new Lexer("2+3*4")
console.log(test.make_tokens())
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
[ Integer { value: 2 }, Integer { value: 3 }, Integer { value: 4 } ]

[Done] exited with code=0 in 0.139 seconds
```

However, my test case failed which led to me doing a series of small tests to debug the error.

```
// Testing
test = new Lexer("2+-3*/4")
console.log(test.make_tokens())
```

The second case showed that the character after a number would be skipped, which therefore made me think that it was an issue with continue(), rather than an issue with the code itself.

```
[
  Integer { value: 2 },
  Minus {},
  Integer { value: 3 },
  Divide {},
  Integer { value: 4 }
]
```

When looking through the code, I realised that the continue() method was called twice after a number was completed, once in the last iteration within the make_number() loop, to check that the next character is not a digit, but then it is also called at the end of a cycle in the make_tokens() loop, which therefore ignores the character directly after the number, which is incorrect.

```
while (this.character != null){
    if (this.character == ' '){
    } else if (DIGITS.includes(this.character)) {
        tokens.push(this.make_number())
        continue
    } else if (this.character == '+') {
        tokens.push(new Add())
```

The solution to this is by adding a continue keyword in make_tokens() at the end of the number creation case. This skips to the next cycle of the while loop without running the rest of the cod, which therefore ignores the second continue() at the end, so will not skip a character.

```
// Testing
test = new Lexer("2+3-4")
console.log(test.make_tokens())
test2 = new Lexer("5.75 * 6.234 + 4 / 79 * (3.14 + 2.17)")
console.log(test2.make_tokens())
```

Now that this has been fixed, the tokens are generated as expected, as shown by my test cases.

```
[
  Integer { value: 2 },
  Add {},
  Integer { value: 3 },
  Minus {},
  Integer { value: 4 }
]
```

```
[
  Float { value: 5.75 },
  Multiply {},
  Float { value: 6.234 },
  Add {},
  Integer { value: 4 },
  Divide {},
  Integer { value: 79 },
  Multiply {},
  LeftBracket {},
  Float { value: 3.14 },
  Add {},
  Float { value: 2.17 },
  RightBracket {}
]
```

Now, this is all the lexing required for this stage of development, so now the proper error handling needs to be added before continuing to parsing, but overall, this implementation has been relatively easy.

ERROR HANDLING

```
// Errors
class Error {
    constructor(text, position){
        this.text = text
        this.position = position
    }
```

I started by creating the Error class, and then the display() method.

The test case I will be using for this is on the right, and I am expecting for the arrow to be pointing at the number 5.

```
test = new Error('0123456789', 5)
test.display()
```

```
display(){
    let spaces = []
    for (let i=0; i<=this.position; i++){
        spaces.push(' ')
    }
    console.log(this.text, "\n", spaces.join(''), '^')
}
```

This was my initial attempt at creating the method, which involved repeatedly adding a space to a list based on the position of the list.

```
0123456789
          ^
```

However, the test case for this was completely wrong.

```
console.log(this.text + "\n" + spaces.join('') + '^')
```

There were two issues with this first attempt.

```
for (let i=0; i<this.position; i++){
    spaces.push(' ')
}
```

Firstly, the outputted message used commas instead of spaces, which would insert a space between each argument, so replacing these with spaces would concatenate them properly.

```
[Running] node
0123456789
          ^
```

Then, the loop needed to be changed to use a < rather than a <= sign, as it still iterated one time too many, but after this change it correctly pointed at the right character.

```
78  class UnexpectedCharacterError extends Error {
79      constructor(text, position, character) {
80          super(text, position)
81          this.character = character
82      }
83
84      message(){
85          return "Unexpected character: '" + this.character + "'\n" + this.display()
86      }
87  }
```

From there, creating the Unexpected Character Error class was very simple, with message() being as described in the plan.

My final two test cases proved that this system is correctly working.

```
89  test = new Error('abcdefghijklmnopqrstuvwxyz 12345', 25)
90  test.display()
```

```
abcdefghijklmnopqrstuvwxyz 12345
                           ^
```

```
89  test = new UnexpectedCharacterError('2 + 5 * 7 p - 2', 10, 'p')
90  console.log(test.message())
```

```
Unexpected character: 'p'
2 + 5 * 7 p - 2
          ^
```

ERRORS IMPLEMENTATION

```
} else {
    return new UnexpectedCharacterError(this.input, this.position, this.character)
}
```

```
default:
    return new UnexpectedCharacterError(this.input, this.position, '.')
```

Firstly, all the temporary Error print statements can be directly replaced by returning a new instance of the Error in make_tokens() and make_number()

```
} else if (DIGITS.includes(this.character)) {
    let number = this.make_number()
    if (number instanceof Error) {
        return number
    }
    tokens.push(number)
    continue
} else if (this.character == '+') {
```

However, when an error is returned in make_number(), it would currently just add the error to the tokens array, so we need to check if what is returned from make_number() is an error, and if it is we return that error in the make_tokens() function, which would stop processing, and otherwise we would just add the result to the end of the list as per usual.

```
160  class Run {
161      constructor(input){
162          this.lexer = new Lexer(input)
163          let tokens = this.lexer.make_tokens()
164          if (tokens instanceof Error){
165              console.log(tokens.message())
166          } else {
167              console.log(tokens)
168          }
169      }
170  }
```

For the medium-term, I have created a new Run class, which will not be kept in the long-term solution. However, for now, it will be useful to link the different stages of lexing, parsing and code execution and will speed up the testing, as the whole sequence will not have to be rewritten, and can all be tested with one line.

Now I need to test each of the different error cases to ensure the message is outputted.

```
172  new Run("1 + 3.5 * 7 - p * 3")
```

```
Unexpected character: 'p'
1 + 3.5 * 7 - p * 3
              ^
```

You can see how useful the new Run class will be by the very short test case on the left, which also shows how unexpected character errors correctly work.

```
new Run("12.3 * 4 - 7.238.943 + 2")
```

```
Unexpected character: '.'
12.3 * 4 - 7.238.943 + 2
           ^
```

However, having too many full stops in make_number() return the error with the wrong position, which is due to an issue in make_number() with how continue() is called before returning the error message.

```
make_number(){
    let number = []
    let fullStops = 0
    while (DIGITS.includes(this.character) || this.character == '.'){
        number.push(this.character)
        if (this.character == '.'){
            fullStops += 1
            if (fullStops == 2){
                return new UnexpectedCharacterError(this.input, this.position, '.')
            }
        }
        this.continue()
    }
    if ((fullStops == 0)){
        return new Integer(Number(number.join('')))
    }
    return new Float(Number(number.join('')))
}
```

I therefore redesigned the make_number() method to check for the full stop errors before calling continue(). I could have just returned the position -1, however I feel this approach makes a lot more logical sense if I have to read it again in the future.

It works properly now, pointing to the full stop.

```
Unexpected character: '.'
12.3 * 4 - 7.238.943 + 2
           ^
```

```
171  new Run("4.5 + 7 - 6.752 + 0")
```

```
[
  Float { value: 4.5 },
  Add {},
  Integer { value: 7 },
  Minus {},
  Float { value: 6.752 },
  Add {},
  Integer { value: 0 }
]
```

I also checked that Run works with normal cases, and it still outputs the correct tokens, which it does so now all the features have been added.

CLEANING UP

Although I have now added all the features that I want for this stage of development, I think it is important to come back and refine previous code, adding commenting and making improvements so that I don't get as confused when I return later in the development process.

```
const prompt = require('prompt-sync')()
```

```
159  class Shell {
160      constructor() {
161          this.main()
162      }
163
164      main(){
165          let input = prompt(" ERL ==> ")
166          while (input != "QUIT()"){
167              this.run(input)
168              input = prompt(" ERL ==> ")
169          }
170      }
171
172      run(input){
173          let tokens = new Lexer(input).make_tokens()
174          if (tokens instanceof Error){
175              console.log(tokens.message())
176          } else {
177              console.log(tokens)
178          }
179      }
180  }
181
182  new Shell()
```

I firstly changed the old Run class into a new Shell class, again, a temporary solution which will make test cases easier, but will take in a user input directly from the user and will then output the result.

I decided to use the prompt-sync module for this, which I installed via node, which just allows me to take in inputs from a terminal, because Node JS does not have an easy way to do this.

```
~/ocr-erl-interpreter on main ↑1 !1 ?4
 node src.js
 ERL ==> 2+2
[ Integer { value: 2 }, Add {}, Integer { value: 2 } ]
 ERL ==> 2.78 - 3.45 / 7.23
[
  Float { value: 2.78 },
  Minus {},
  Float { value: 3.45 },
  Divide {},
  Float { value: 7.23 }
]
 ERL ==> 2 + 3.5 - q
Unexpected character: 'q'
2 + 3.5 - q
        ^
 ERL ==> QUIT()
~/ocr-erl-interpreter on main ↑1 !1 ?4
```

I also made it so the shell would stop executing when QUIT() was entered, another arbitrary decision but this is just a medium-term testing implementation.

BETTER JAVASCRIPT

I'm new to JavaScript, so I tried to research some easier ways to implement features I'd already written. One example is using template literals, which make it easier to format strings.

```
message(){
    return ` ! ERROR\nUnexpected Character: '${this.character}'\n${this.display()}`
}
```

An example of where I used this is in UnexpectedCharacterError's display()

These are equivalent to f-strings in python, which I am familiar with, and make it easier to integrate variables into a string. I also redesigned the error message format slightly.

```
 ERL ==> 2 + 3 - q
 ! ERROR
Unexpected Character: 'q'
2 + 3 - q
        ^
```

Additionally, I found that a repeat() method existed, which I could then use in display() instead of my clunky loop solution, which simplifies the code significantly, and still results in the same result.

```
display(){
    return `${this.text}\n${' '.repeat(this.position)}^`
}
```

The next useful feature is the conditional ternary operator, which can be used within JavaScript with an `expression ? statement : statement` syntax. It is the equivalent of checking if the expression is true, and then returns the first statement if it is true, or the second if it is false, and removes redundant if statements.

For example, I used it within the continue() method, to check if the position is equal to the length, which then sets the current character to null if true, or the corresponding character if false.

```
continue(){
    this.position += 1
    this.character = this.position == this.input.length ? null : this.input[this.position]
}
```

I also changed this in some other places, and it just removes some additional if statements in places. All of these are just minor changes, but I'm hoping to get into a good habit of using them for the rest of the program, because using as many built-in JavaScript functions as possible is great.

Overall, this section was simple but important, and now I can continue onto the parser, which is where the whole process becomes a lot more complicated.

DESIGN: PARSER

```
bnf.txt
<expression> ::= <term> { (Plus | Minus ) <term> }
<term> ::= <exponent> { ( Multiply | Divide ) <exponent> }
<exponent> ::= <factor> { Exponent <factor> }
<factor> ::= Integer | Float | LeftBracket <expression> RightBracket
```

Above is the BNF for this section of implementing the parser. This example looks more complex than the version I used in analysis, but the overall idea is the same. Before I go into the exact details of how I will implement the classes and methods to allow parsing to work, I will explain the process in slightly more depth because my analysis description was very brief.

This will include some basic pseudocode, which is my overall current idea of how I would like the parsing to function, based off my reading from the analysis stage. Note that this pseudocode will be heavily abstracted for the purpose of the explanation, and is not exactly how I will implement it

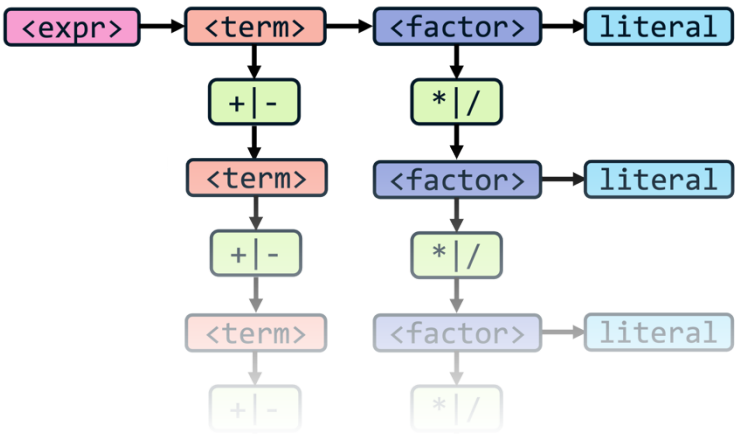
THE COMPLEXITIES OF PARSING

In our previous example, we thought of parsing being constructed from the bottom up: from <term> to <factor> to <expr>. This process occurs the other way around, using recursion, from the top down, with <expr>s constructing the <factor>s, which make the <term>s.

In our code, we will have a different method for parsing the <expr>, the <term>, and the <factor>, each of which will call upon the next. Because this will be done in a class within a single object, the current tokens and positions do not have to be passed between these methods, and instead will be stored as properties, and I will be using the same continue() format as I did in the lexer to progress through these different tokens. This is where having an Object-Oriented approach makes the code significantly simpler.

This means that, after one method has completed parsing, you can call it again and the current position and character are maintained. Our parser will work by each of our different methods repeatedly calling the method that is "below" it, whilst checking that the current character is one of the corresponding characters.

Take the expression method for example, <expr> is defined as <term>s separated by pluses or minuses.



Therefore, the expression method will call the term method when it is called, to parse the term, and then check that the next token is a plus or a minus. If it is, the process will repeat, calling the term method until there are no more pluses or minuses left.

Likewise, the term method will call the factor method, and then check if the next token is a * or a /. If it is, then the term method is called again, and this also repeats.

I think this diagram is useful to visualise this repeated pattern, because that is the best way to thinking about it: a long chain.

It is also now important to visualise the code itself. Diagrams are important to visualise the overall structure but explaining it in pseudocode helps to bridge between the logic into the code itself, and I think in this case, pseudocode makes it a lot easier to understand, but in a very abstract form.

```
<expr>
parse <term>
while token is + or -
        parse <term>
endwhile
```

```
<term>
parse <factor>
while token is * or /
        parse <factor>
endwhile
```

On the left is this idea written in code, repeatedly calling and parsing the next method whilst checking that the token following the previous parsing is one of the permitted tokens. By themselves, these individual methods are very simple, and comprise mainly of this single loop.

The complex part of this is realising what this means on a larger scale. I feel it is best to think of this as "filling in the gaps", and so that each time, for instance, the expression method calls the term method, the next token will either be a plus, minus or the end of the input (so long as there are no syntax errors).

This is because the term method will deal with all the multiplication, division, and literals by itself, so all that is left to the expression method is the multiple steps of addition and subtraction. So again, whenever the term method is called, the expression method anticipates and knows that there will be a plus, minus or the end of the string, and it is this expectation that the parser will be in the right position which is difficult to grasp.

This is a great example of decomposition: each individual method by itself is extremely simple, just iterating over the first few steps, but it's the chaining of these methods that creates a lot of abstraction and solves a very complicated problem in the meantime.

This also ensures the two main principles of parsing are followed. The precedence is maintained by having the first method to be called the most generic and lowest-precedence one, in this case the expression method, because it will immediately call the factor() method, which will arrange any multiplication and division, until there are no more left, at which point it will return to then do any addition and subtraction, which means it has less priority and is of lower precedence.

The associativity is also maintained, because the while loop within each method will iterate through the tokens, therefore the left-most tokens will be reached before the rightmost ones, however this will make more sense when we begin to construct some abstract syntax trees.

CONSTRUCTION

Now that we know in which order build our tree, we must construct it. All the operator tokens will have a left property and a right property, representing the two things they are operating on, which will form the foundation of our tree, and allow for chains of these operators to be made.

```
<expr>
result = parse <term>
while token is + or -
        token.left = result
        result = token
        result.right = parse <term>
endwhile
return result
```

Here, the result variable stores the current contents of the abstract syntax tree. Whenever a new operator is encountered in the loop, it will become the new root of the tree. Therefore, the old tree will become it's left child, because it occurred before it, and when the next method is parsed, its result will be stored in the right property, because it is what follows it. Finally, this tree is then stored in result for the next iteration.

The expression method example is shown on the left, and result will be returned at the end, which will return the abstract syntax tree at the end of that section of parsing.

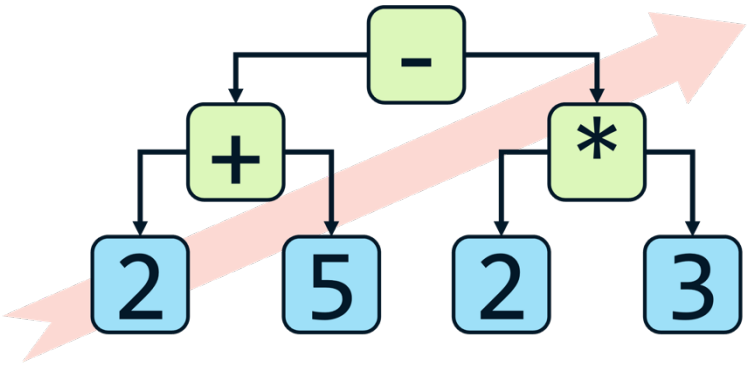
This is the same with all the methods, and again they can be "expected" to be correct. Therefore, when the term method is called, a correctly arranged multiplication and division tree will be returned.

An example should hopefully help illustrate this, so say the expression method is called on this input:

➔ The term method is called and returns an Integer with value 2
➔ The current token is a Plus, so its left property is set to the result: an Integer with value 2
➔ The term method is called again and returns an integer with value 5
➔ This is set as the right property of the Plus token, which now has two children

➔ The new current token is a Minus, so its left property is set to the previous result
    ➔ This means its left property is now a Plus token which has two Integer children

➔ The term method is called again, and now returns a Multiply Token
    ➔ This Multiply Token has a left child of Integer 2 and a right child of Integer 3
➔ This Multiply Token is set as the right child of the Minus token
➔ There is now no new current token so the expression method is complete and returns the Minus.



As you can see, this results in this abstract syntax tree being formed, which is built from the bottom-left to the top-right. Because the lower parts of the tree will be evaluated first, this shows how precedence and associativity are maintained: with the multiplication and first plus being executed before the second minus.

It is good to try and visualise each one of the steps as this tree expanding, growing upwards and to the right, as the tree will always grow one node taller on each iteration of a method, because the new token that is encountered will become the tree for that specific tree.

```
parse_binary_operator (next tokens)
result = parse next
while token is in tokens
        token.left = result
        result = token
        result.right = parse next
endwhile
return result
```

Finally, it is worth thinking about how this solution can be generalised, because these methods follow the same generic format, but just with different tokens that are allowed, and different methods that need to be called next.

The example method on the left takes in two inputs for these unknowns, and now can be called by each of the different methods, vastly reducing code repetition in the interpreter, because the same steps are followed every time. Note that the factor method will have a completely different process, which I will explain soon.

Overall, I hope this explanation has helped, because it took my head a while to wrap around these concepts. In terms of implementation, it should not be that complex, because the methods themselves are very simple, but realising how they work, and why they work is much more important, and has been very interesting for me to discover.

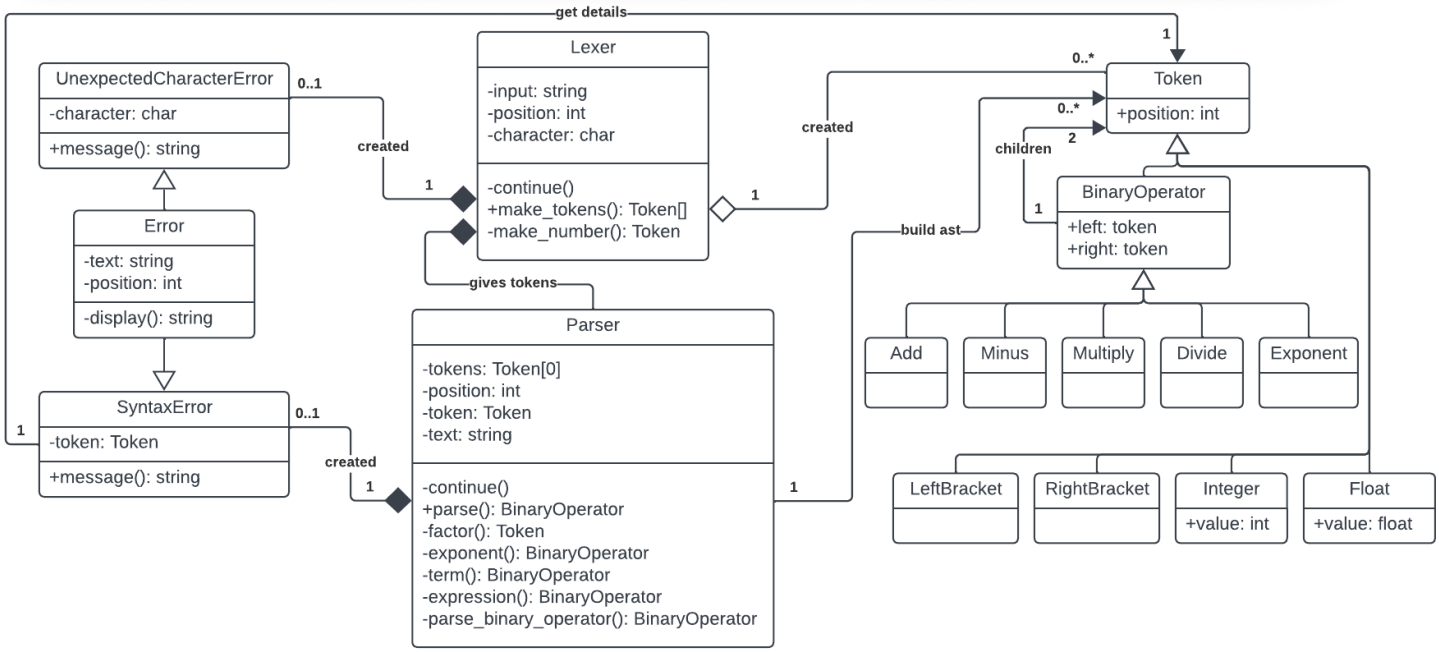
These concepts will also carry into the rest of the interpreter, and essentially the entire parser is different ways of calling other methods with a higher precedence, so this arithmetic example is important to get.

INTEGRATING PARSING

Now let's get back to our implementation and have a look into how our properties will allow us to parse according to our BNF rules, with the expanded UML diagram.

bnf.txt

```
<expression> ::= <term> { (Plus | Minus ) <term> }
<term> ::= <exponent> { ( Multiply | Divide ) <exponent> }
<exponent> ::= <factor> { Exponent <factor> }
<factor> ::= Integer | Float | LeftBracket <expression> RightBracket
```



TOKEN UPDATES

No huge updates in terms of new code, but mainly additions of small features:

➔ Position attribute given to all Token classes.

This allows for error handling to function, as the position for each token must be remembered so that the display() methods will still function and point to the token that has the error.

➔ Binary Operator class is a parent class of all arithmetic classes.
➔ All Binary Operator classes given properties for left and right child nodes.

This allows us to construct abstract syntax trees using the Tokens, but it is the new Parser class which changes these left and right properties to the correct values, and they will be null by default.

➔ New Exponent class is created.

I forgot to include this in the original Lexer, but it's addition should be very straightforward: adding an extra line to make_tokens(), but I will properly show this at the time.

ERRORS

The addition of a new Syntax Error class for the Parser to use. This will take in the parameters of the text and the token which caused the error to occur. In the super() call in the constructor, it will pass the position property of the token that it was given, as this will be the position of the token that cause the error.

Therefore, display() will correctly work, and the message() function will be customised for the error messages that occur based on this token.

PARSER

In its constructor method, it will take in the stream of tokens from the lexer, which will be stored in the tokens property, as well as the original text, stored as text, to be passed to any syntax errors that occur so the correct message will be displayed.

It will also include a position and token property, which are direct equivalents of the position and character properties in the Lexer, allowing us to cycle through the tokens and keep it consistent.

➔ continue()

Again, replicated from the Lexer, incrementing the position on each call, and setting the token property to the corresponding token in the tokens list. This will also set the token to null if the length is exceeded.

➔ parse()

This is the main method to be called, which will return the abstract syntax tree when called. More specifically, it will return the root node, which will have the children that contain the rest of the tree. Additionally, a Syntax Error may also be returned if the stream of tokens was invalid.

In the future, this method will also recognise which "type" of line the code is. For instance, whether it is an assignment, an if statement, or just an expression. It would then call the corresponding method for whatever is required. However, for now, it will call the expression method, as we are only parsing arithmetic expressions, so this is the correct method to call, and will return its result.

Finally, after the expression() method has been called, if the current token is not null it means that the entire stream of tokens has not been parsed, meaning that there was an error in the syntax. In this case, it would mean that there were two literals in a row in the expression, as otherwise the expression would be able to be continually parsed, so therefore a Syntax Error is returned, with the current token being passed into it, as it is this double literal that would have caused the issue to occur.

THE PARSING FUNCTIONS

➔ parse_binary_operator()

This is our generic property which I discussed in the explanation. Therefore, it takes in the next function to be called and an array of the valid tokens to check through.

It also must be able to deal with any errors it receives from methods it calls, so that syntax errors do not get built into the abstract syntax tree. Therefore, whenever it calls a method, it will check if the result is an instance of an Error, and if it is then it will be returned. This will therefore pass the error up the chain of different methods, so it reaches parse() and will be returned there.

➔ expression()

Calls parse_binary_operator() method with the next method being the term() method, and the array of valid tokens being Add and Minus and returns the result.

➔ term()

Calls parse_binary_operator() method with the next method being the exponent() method, and the array of valid tokens being Multiply and Divide and returns the result.

➔ exponent()

Calls parse_binary_operator() method with the next method being the factor() method, and the array of valid tokens being only Exponent and returns the result.

➔ factor()

This stage is more complex than the others, as shown by the BNF definition, and it does not just call parse_binary_operator() because it has a few different cases.

Firstly, we check if the current token Is either an Integer or a Float, and if it is, then we will continue() past it, and return that prior token, because a factor can be either of these.

Secondly, if the current token is a LeftBracket, then we continue() past it and call the expression() method. This will then return the corresponding AST that was between the brackets, and then the following token is checked so that it is a right bracket. If it is, then the expression() result can be returned, and it will be built into the abstract syntax tree. Therefore, the bracket tokens will not end up inside the abstract syntax tree, which is good because they have no actual meaning or operations, and are only used to specify precedence, which they do in arranging the AST.

It the next token is not a RightBracket, then the brackets were not properly closed, and a SyntaxError can be created and returned.

Later on, I will need to also include unary operators into the factor() segment, so that negative numbers can also be created, however I will add this after these features have been added, to not have too much to add in one section.

\

DEVELOPMENT: PARSER

ADDITION

```
148  class Parser {
149      constructor(tokens){
150          this.tokens = tokens
151          this.position = -1
152          this.token = null
153          this.continue()
154      }
155
156      continue(){
157          this.positio
158          this.token =
159      }
160  }
```

```
183  test = new Parser(new Lexer("2 + 3 * 4.5 - 7").make_tokens())
184  while (test.token != null){
185      console.log(test.token)
186      test.continue()
187  }
```

This initial code is copied from the lexer, and the basic test case worked.

```
[Running] node "/Users
Integer { value: 2 }
Add {}
Integer { value: 3 }
Multiply {}
Float { value: 4.5 }
Minus {}
Integer { value: 7 }
```

```
12  class BinaryOperator extends Token{
13      constructor(){
14          super()
15          this.left = null
16          this.right = null
17      }
18  }
19
20  class Add extends BinaryOperator{
21      constructor(){
22          super()
23      }
24  }
```

I added the Binary Operator class which extends Token, and then changed Add, Minus, Multiply and Divide to extend it as well. This just reduces code repeating.

Then I began implementing the expression() method, which will use the following test case. The end goal is to have an Add token, with the left child being an Add token with 2 Integer children of 2 and 3, and the right child being an Integer of 4.

```
220  test = new Parser(new Lexer("2 + 3 + 4").make_tokens()).expression()
221  console.log(test)
```

```
term(doContinue){
    if (doContinue){
        this.continue()
    }
    let result = this.token
    this.continue()
    return result
}

expression(){
    let result = this.term(false)
    while (this.token != null && (this.token instanceof Add || this.token instanceof Minus)){
        if (this.token instanceof Add){
            this.token.left = result
            this.token.right = this.term(true)
            result = this.token
        }
    }
    return result
}
```

I am going to build the different sections one at a time to ensure that they are working, so this first version is moreso a proof of concept, and figuring out if the basics work, before adding the proper methods later on.

The term() method is completely temporary, and there is no generic method yet, but I made this very basic version to test if my basic expression method would work.

And when running the test case, nothing was outputted, which showed how the basic test failed.

```
console.log(this.token)
this.token.left = result
console.log(this.token)
this.token.right = this.term(true)
console.log(this.token)
result = this.token
```

```
Add { left: null, right: null }
Add { left: Integer { value: 2 }, right: null }
Add { left: null, right: null }
Add { left: null, right: null }
<ref *1> Add { left: [Circular *1], right: null }
null
null
```

```
term(doContinue){
    if (doContinue){
        this.continue()
    }
    let result = this.token
    console.log("TERM")
    console.log(result)
    this.continue()
    return result
}
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
TERM
Integer { value: 2 }
Add { left: null, right: null }
Add { left: Integer { value: 2 }, right: null }
TERM
Integer { value: 3 }
Add { left: null, right: null }
Add { left: null, right: null }
<ref *1> Add { left: [Circular *1], right: null }
TERM
Integer { value: 4 }
null
null

[Done] exited with code=0 in 0.057 seconds
```

When adding some output statements to track the tokens, it showed that one of the Add tokens was being set as its own left child, showing that the construction was somehow broken, which I thought was something wrong with how I was using continue() in my method.

With some extra output statements, it shows that the first stage of construction is working correctly, and that the Integer is correctly set to the left child, but after this the code broke, and this weird circular issue was occurring. However, the terms are correct, so continue() must be used correctly.

THE SOLUTION

However, the issue was with my original expression code, because I did not consider that the token property would change after the term() method is called, which is obvious with hindsight, because it had been continued() past, however I did not consider this in the moment, and instead was using the incorrect token, so the old syntax tree was completely discarded and a new, broken one was created.

```
term(){
    let result = this.token
    this.continue()
    return result
}

expression(){
    let result = this.term()
    while (this.token != null && (this.token instanceof Add || this.token instanceof Minus)){
        if (this.token instanceof Add){
            this.token.left = result
            result = this.token
            this.continue()
            result.right = this.term()
        }
    }
    return result
}
```

After I understood that this would change, I rearranged my code slightly so that result would be set to the token before term() was called for a second time, and then I would set the right property of result to the result, which would still hold the old operation node, so it should correctly build the tree.

The new changes to the code can be seen on the left.

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
Add {
  left: Add { left: Integer { value: 2 }, right: Integer { value: 3 } },
  right: Integer { value: 4 }
}

[Done] exited with code=0 in 0.046 seconds
```

Now that I was more familiar with how continue() and the token property worked, I changed term to no longer have the redundant parameter, and to instead call continue() before it, making the code much more readable and consistent.

As shown above, the test functions now, building the correct syntax tree, so this section is complete.

OTHER OPERATIONS

Now I needed to add the three other arithmetic operations, starting with Minus.

```
expression(){
    let result = this.term()
    while (this.token != null && (this.token instanceof Add || this.token instanceof Minus)){
        this.token.left = result
        result = this.token
        this.continue()
        result.right = this.term()
    }
    return result
}
```

```
211  test = new Parser(new Lexer("2 + 3 - 4").make_tokens()).expression()
212  console.log(test)
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
Minus {
  left: Add { left: Integer { value: 2 }, right: Integer { value: 3 } },
  right: Integer { value: 4 }
}
```

This was a lot easier than I thought, and the code ended up being simpler than before, as I removed the redundant check for if the token was an Add, because Add and Minus would have been built in the same way due to their identical precedence.

As shown on the left, this immediately allowed for subtraction to be built into the abstract syntax tree, so this operator is completely implemented.

MULTIPLICATION AND DIVISION

```
factor(){
    let result = this.token
    this.continue()
    return result
}

term(){
    let result = this.factor()
    while (this.token != null && (this.token instanceof Multiply || this.token instanceof Divide)){
        this.token.left = result
        result = this.token
        this.continue()
        result.right = this.factor()
    }
    return result
}
```

Before adding the generic method, I duplicated the code from expression into term, changing the valid tokens to Multiply and Divide, and made it call factor() instead, which is the old term() method.

I then ran the following test case:

```
218  test = new Parser(new Lexer("2 + 3 - 4 * 7 - 3 / 4").make_tokens()).expression()
219  console.log(test)
```

```
Minus {
  left: Minus {
    left: Add { left: [Integer], right: [Integer] },
    right: Multiply { left: [Integer], right: [Integer] }
  },
  right: Divide { left: Integer { value: 3 }, right: Integer { value: 4 } }
}
```

As shown on the left, this produced the correct tree: with multiplication and division having priority from left to right, and then addition and subtraction. This is great, and is really promising to show that the system will work, but now I need to reduce the large amount of code repetition.

GENERALISING.

```
check_instance(check) {
    for (let item of check){
        if (this.token instanceof item){
            return true
        }
    }
    return false
}
```

This was not in the plan, but I created a new check_instance() method, which will accept an array of different classes, and it will check if the current token is an instance of any of them and will return true or false accordingly. This can therefore be used within our generic method, for example, for expressions, it will be passed an array for Plus and Minus, to check if the next iteration must occur.

```
227  test = new Parser(new Lexer("2 + 3 + 4").make_tokens())
228  test.continue()
229  console.log(test.check_instance([Minus, Add]))
```

```
[Runn
true
```

The test case for this proved that it works as expected.

```
parse_binary_operator(nextFunction, tokens){
    let result = nextFunction()
    while (this.token != null && this.check_instance(tokens)){
        this.token.left = result
        result = this.token
        this.continue()
        result.right = nextFunction()
    }
    return result
}
```

```
term(){
    return this.parse_binary_operator(this.factor, [Multiply, Divide])
}

expression(){
    return this.parse_binary_operator(this.term, [Add, Minus])
}
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
/Users/pw/ocr-erl-interpreter/testing.js:185
        return this.parse_binary_operator(this.factor, [Multiply, Divide])
                                          ^

TypeError: Cannot read properties of undefined (reading 'parse_binary_operator')
    at term (/Users/pw/ocr-erl-interpreter/testing.js:185:21)
```

Now, I copied the old expression code into the new parse_binary_operator() method, changing it so that it the nextFunction argument would be called instead of term() being called, and so that it uses check_instance() on the tokens argument.

This naming may be slightly poor, as methods are passed to nextFunction, however the overall functionality is the same.

How this is used is shown on the left, with the first argument being the next function to be called, follow by the array of usable functions.

check_instance() is very useful in this case, as it allows for any number of tokens to be in the tokens array, compared to the hold system which just had a fixed number of or statements, this is more dynamic.

However, instantly an error occurred, with the message saying that parse_binary_operator() was undefined. At first, this confused me, but after some debugging using lots of output statements, but after I researched into the problem, I realised that this was an issue with JavaScript rather than my code.

"this" represents the instance of a class in JavaScript, so it can be used to access all the attributes and methods of the specific object. However, "this" is not specific to only classes, but as JavaScript is an object-oriented language, what "this" references changes based on the context.

In my case, the meaning of "this" was being changed so that it no longer referenced the instance of the Parser class but was referring to one of the methods that was calling it, because the long chain of methods calling each other had mutated the meaning of "this".

This means that I can no longer rely on using "this", because it is not consistent, and instead I need a different way of consistently accessing the current instance. My solution to this is to give every single method a parameter, called "self", which will store the instance of the Parser. Therefore, this argument will be used exactly like "this", however it is passed to the method rather than being a native keyword.

For this to function, every single method in the Parser will need to have this "self" parameter, so for consistency I am going to ensure that it is always the first parameter of every single method.

```
// First parameter = References the instance of the parser
// Second parameter = Method to call that is beneath the current one
// Thid parameter = Array of tokens which are to be checked for
parse_binary_operator(self, nextFunction, tokens){
    let result = nextFunction(self)
    while (self.token != null && self.check_instance(tokens)){
        self.token.left = result
        result = self.token
        self.continue()
        result.right = nextFunction(self)
    }
    return result
}
```

```
term(self){
    return self.parse_binary_operator(self, self.factor, [Multiply, Divide])
}

expression(self){
    return self.parse_binary_operator(self, self.term, [Add, Minus])
}
```

An example of how this is used is on the left, with parse_binary_operator(). The "self" parameter is first, and then the rest of the method uses "self" as if it were "this", accessing the tokens using it, and calling continue() using it.

Additionally, when calling nextFunction(), it passes self as an argument, so that the method can still access its own instance, and this chain of passing will continue throughout the program.

This new syntax will also have to be used in term() and expression() as shown on the left, essentially a direct replacement of "this" with "self" and passing it to any other methods it calls.

```
parse(){
    return this.expression(this)
}
```

"self" originates from the parse() method, as this is the main method to be called from the interpreter, and this will pass "this" as an argument to expression(), starting the chain of "self"s with the original instance.

```
test = new Parser(new Lexer("2 + 3 + 4").make_tokens()).parse()
console.log(test)
```

```
Add {
  left: Add { left: Integer { value: 2 }, right: Integer { value: 3 } },
  right: Integer { value: 4 }
}
```

As shown, this fixes the issue and now the generic method works. I do not love this implementation, as it feels kind of messy, and I'd much rather use this if it were possible, because now a lot of additional arguments are required in all the methods.

However, when researching the issue, it sounded like it was just a quirk of JavaScript, and there was no real workaround this issue, so it may be something I will have to deal with for the whole project.

EXPONENTS

```
44  class Exponent extends BinaryOperator{
45      constructor(){
46          super()
47      }
48  }
```

The whole reason for constructing our code in this style is so it is a foundation for the future and can be easily expanded to include new features into the interpreters, such as exponents.

```
} else if (this.character == '^') {
    tokens.push(new Exponent())
} else if (this.character == '(') {
```

To add them, firstly the new class must be defined and extend the Binary Operator class, and the make_tokens() method must be altered to create a new Exponent when "^" is encountered.

```
factor(self){
    let result = self.token
    self.continue()
    return result
}

exponent(self){
    return self.parse_binary_operator(self, self.factor, [Exponent])
}

term(self){
    return self.parse_binary_operator(self, self.exponent, [Multiply, Divide])
}
```

From here, as exponents have a higher precedence than multiplication and divison in bodmas, the exponent() is added, with the nextFunction being factor(), and the valid token being Exponent in an array, so that it works with check_instance().

term() must now be changed to call exponent() instead of factor(), to properly weave it into the system.

```
250 test = new Parser(new Lexer("2 * 5 ^ 3 - 7 ^ 2").make_tokens()).parse()
251 console.log(test)
```

```
Minus {
  left: Multiply {
    left: Integer { value: 2 },
    right: Exponent { left: [Integer], right: [Integer] }
  },
  right: Exponent { left: Integer { value: 7 }, right: Integer { value: 2 } }
}
```

Already, this implementation immedaitely works, without any issues. This really shows how easy it is to expand with this system, and is very positive for the rest of development, because new features can be added so easily.

BRACKETS

Now that the other lines of BNF have been implemented, it leaves us with the factor() method to work on.

```
factor(self){
    if (self.check_instance([Integer, Float])){
        let result = self.token
        self.continue()
        return result
    } else if (self.ch_instance([LeftBracket])){
        self.continue()
        let result = self.expression(self)
        if (self.check_instance([RightBracket])){
            return result
        }
        console.log("ERROR: DIDN'T CLOSE THE BRACKETS")
    }
}
```

My first attempt of implementation matched the description in the design phase. The new Left Bracket check had a temporary outputted error message if the token after expression() was called was not a Right Bracket, to be changed later.

```
254  test = new Parser(new Lexer("2 + (3 + 4)").make_tokens()).parse()
255  console.log(test)
```

```
LeftBracket {}
LeftBracket {}
LeftBracket {}
LeftBracket {}
LeftBracket {}
LeftBracket {}
LeftBracket {}
LeftBracket {}
LeftBracket {}
```

The test case above initially failed, and after outputting the tokens, I realised it was stuck on the LeftBracket, and then that the () were missing after the continue() call, which was causing the issue,

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
Add {
  left: Integer { value: 2 },
  right: Add { left: Integer { value: 3 }, right: Integer { value: 4 } }
}
```

After this was fixed, the abstract syntax tree was in the correct order, and the brackets were working as intended. To ensure it worked perfectly, I had to try a few other test cases.

```
253  test = new Parser(new Lexer("2 * (3 + 4) ^ 7").make_tokens()).parse()
254  console.log(test)
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
Multiply {
  left: Integer { value: 2 },
  right: Add { left: Integer { value: 3 }, right: Integer { value: 4 } }
}
```

The test case for having the brackets in the middle of the input failed, and somehow all of the tokens after the closing bracket were not being built into the abstract syntax tree.

```
if (self.check_instance([RightBracket])){
    self.continue()
    return result
}
```

Again, after some more debugging, it was because continue() was not being called to progress past the RightBracket.

```
Multiply {
  left: Integer { value: 2 },
  right: Exponent {
    left: Add { left: [Integer], right: [Integer] },
    right: Integer { value: 7 }
  }
}
```

This meant that the tokens after it were never reached, which explained why the tokens were being discarded. After adding the continue() call, the correct AST was then produced.

OTHER TEST CASES

```
test = new Parser(new Lexer("2 * (3 + 4 ^ 7").make_tokens()).parse()
console.log(test)
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
ERROR: DIDN'T CLOSE THE BRACKETS
Multiply { left: Integer { value: 2 }, right: undefined }
```

The test cases for unclosed brackets worked as expected, with the temporary error message being outputted. Later I will ensure that this returns a proper error, however for now this is sufficient to show that the check for invalid bracket usage is correct.

```
test = new Parser(new Lexer("1 + (2 + (3 + 4))").make_tokens()).parse()
console.log(test)
```

Finally, I need to test for multiple layers of nested brackets.

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing
Add {
  left: Integer { value: 1 },
  right: Add {
    left: Integer { value: 2 },
    right: Add { left: [Integer], right: [Integer] }
  }
}
```

As shown, the correct AST was produced, and the correct order of operations has been made.

This therefore concludes this section on brackets. Again, reasonably simple, but I did have issues on where I needed to continue(), and I am constantly having to debug different cases to make it function.

Now, I need to add proper error handling, and get rid of the temporary outputted error messages and account for a lot of the cases that completely break the program.

ERROR HANDLING

```
test = new Parser(new Lexer("2 + + ) ^ * 7").make_tokens()).parse()
console.log(test)
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
Add {
  left: Add { left: Integer { value: 2 }, right: undefined },
  right: undefined
}
```

As you can see, currently if the input string has invalid syntax, the Parser attempts to produce an AST which is very incorrect, rather than reporting an error. Now, the program needs to be able to detect when there is an issue and report an error accordingly.

```
class Token {
    constructor(position){
        this.position = position
    }
}

class BinaryOperator extends Token{
    constructor(position){
        super(position)
        this.left = null
        this.right = null
    }
}
```

```
} else if (this.character == '+') {
    tokens.push(new Add(this.position))
} else if (this.character == '-') {
    tokens.push(new Minus(this.position))
} else if (this.character == '*') {
    tokens.push(new Multiply(this.position))
```

The first stage is adding a position property to all token classes, which is needed for the display() property in the Error class to point to the error.

Because the position property needs to be given on creation of the token, this needs to be added to the Lexer. Therefore, in the make_tokens() method, the Lexer's current position is passed into the constructor of the tokens, so therefore the position corresponds correctly with the plaintext, rather than other methods which may cause issues with whitespaces.

For tokens which take up more than one character, I'd like the position property to correspond to the position of the first character. So, for make_number(), we will have to assign the position to a variable at the start of the method, before using continue(), so that the original position is remembered, and then use that variable for the construction of the Integer or Float, as shown in the code below.

```
return fullStops == 0 ? new Integer(position, Number(number.join(''))) : new Float(position, Number(number.join('')))
```

```
Add {
  position: 3,
  left: Add {
    position: 1,
    left: Integer { position: 0, value: 1 },
    right: Integer { position: 2, value: 2 }
  },
  right: Integer { position: 4, value: 3 }
}
```

```
267  test = new Parser(new Lexer("1+2+3").make_tokens()).parse()
268  console.log(test)
```

When quickly testing this, the position attributes corresponded with their actual locations, so it was successful.

Now that this has been implemented, the Syntax Error class can finally be implemented.

SYNTAX ERRORS

```
class SyntaxError extends Error {
    constructor(text, token) {
        super(text, token.position)
        this.token = token
    }

    message(){
        return ` ! ERROR\nInvalid Syntax: '${this.token}'\n${this.display()}`
    }
}
```

The new class will accept the text and a token as its parameters and will pass the position of the token to the super() constructor, which allows the display() method to correctly work.

I also added the temporary error message, which will expand later.

```
factor(self){
    if (self.check_instance([Integer, Float])){
        let result = self.token
        self.continue()
        return result
    } else if (self.check_instance([LeftBracket])){
        self.continue()
        let result = self.expression(self)
        if (self.check_instance([RightBracket])){
            self.continue()
            return result
        }
        return new SyntaxError(self.text, self.token)
    }
    return new SyntaxError(self.text, self.token)
}
```

To implement errors, the Parser's constructor must take in the plaintext input.

```
constructor(tokens, text){
    this.tokens = tokens
    this.text = text
```

Now, I can replace any outputted error messages with new Syntax Error returns, passing the text and the current token as the arguments. On the left, the first case is for unclosed brackets, and the second is for when no valid factor is detected: which is when two operators are used in a row, returning an error with the token that caused the issue.

```
parse_binary_operator(self, nextFunction, tokens){
    let result = nextFunction(self)
    if (result instanceof Error){
        return result
    }
    while (self.token != null && self.check_instance(tokens)){
        self.token.left = result
        result = self.token
        self.continue()
        result.right = nextFunction(self)
        if (result.right instanceof Error){
            return result.right
        }
    }
    return result
}
```

parse_binary_operator() has now also been updated, so that it will check the result of any method it calls, to ensure that any errors do not get build into the abstract syntax tree, and that they will get returned and passed to the previous method.

Now, when running a test case as shown on the right, it is properly detected and returned, instead of a failed ast being produced. However, the error message is very broken, and provides no useful information.

```
[Running] node "/Users/pw/ocr-erl-
 ! ERROR
Invalid Syntax: '[object Object]'
1 * * 3
    ^
```

```
    run(input){
        let tokens = new Lexer(input).make_tokens()
        if (tokens instanceof Error){
            console.log(tokens)
            return
        }
        let parsed = new Parser(tokens, input).parse()
        console.log(parsed instanceof Error ? parsed.message() : parsed)
    }
}
```

Finally, I implemented the system back into our temporary Shell feature, so that our run() method would now correctly also parse the result from the lexer whilst checking for any errors, and if any are detected in the parser, they will be display()ed to show the error more clearly.

COMPLETING ERROR HANDLING

```
└─ node testing
 ERL ==> 2 + (3 - 4
/Users/pw/ocr-erl-interpreter/testing.js:102
        super(text, token.position)
                    ^

TypeError: Cannot read properties of null (reading 'position')
```

One current issue is with unclosed brackets: because null is passed to the Syntax Error, so trying to access its position leads to JavaScript crashing.

```
} else if (self.check_instance([LeftBracket])){
    let errorToken = self.token
    self.continue()
    let result = self.expression(self)
    if (self.check_instance([RightBracket])){
        self.continue()
        return result
    }
    return new SyntaxError(self.text, errorToken)
```

To fix this, I store the original LeftBracket token in a variable before calling expression(), and if there is no RightBracket to close, then the error will be returned, but will pass this stored LeftBracket as the error token instead.

I then also upgraded Syntax Error's message() method, so that if the token is a LeftBracket, a custom error message will be returned.

```
message(){
    if (this.token instanceof LeftBracket) {
        return ` ! ERROR\nInvalid Syntax: '(' was never closed\n${this.display()}`
    }
    return ` ! ERROR\nInvalid Syntax\n${this.display()}`
}
```

This message is much more useful than "Invalid Syntax".

```
└─ node testing
 ERL ==> 2 + / 7
 ! ERROR
Invalid Syntax
2 + / 7
    ^
 ERL ==> 2 - (3 * 9
 ! ERROR
Invalid Syntax: '(' was never closed
2 - (3 * 9
    ^
```

```
 ERL ==> 2 + 7 8 - 3
Add {
  position: 2,
  left: Integer { position: 0, value: 2 },
  right: Integer { position: 4, value: 7 }
}
```

The last error that needs to be implemented is for two literals in a row, where the code still attempts to produce a broken AST.

```
parse(){
    let result = this.expression(this)
    if (this.token == null){
        return result
    }
    return SyntaxError(this.text, this.token)
}
```

The solution to this is at the end of parse(), and if there are still remaining tokens left (so the current token is not null), it means the parser stopped before reaching the end, meaning that there must have been two numbers in a row, so a Syntax error can be returned.

Now, I just added two more error messages for the different cases, depending on the kind of token that was returned. As shown, now all of the different errors return a more descriptive message that matches the issue.

```
message(){
    if (this.token instanceof LeftBracket) {
        return ` ! ERROR\nInvalid Syntax: '(' was never closed\n${this.display()}`
    } else if (this.token instanceof BinaryOperator) {
        // Or identifier in the future
        return ` ! ERROR\nInvalid Syntax: Expected literal\n${this.display()}`
    }   else if (this.token instanceof Integer || this.token instanceof Float) {

        return ` ! ERROR\nInvalid Syntax: Expected operator\n${this.display()}`
    }
    return ` ! ERROR\nInvalid Syntax\n${this.display()}`
}
```

This works for now, but in the future the same token may be involved in different errors, so this system will have to change.

```
└─ node testing
 ERL ==> 7 * / 3
 ! ERROR
Invalid Syntax: Expected literal
7 * / 3
    ^
 ERL ==> 2 + ( 3 - 4
 ! ERROR
Invalid Syntax: '(' was never closed
2 + ( 3 - 4
    ^
 ERL ==> 7 + 3.5 7
 ! ERROR
Invalid Syntax: Expected operator
7 + 3.5 7
      ^
 ERL ==>
```

DESIGN: EXECUTION

In order to get an actual value out of the calculation, we must now add the code in the Token classes.

EVALUATE()

Every single token that can end up in the abstract syntax tree (so excluding the brackets) will have a method called evaluate(), that will return its value.

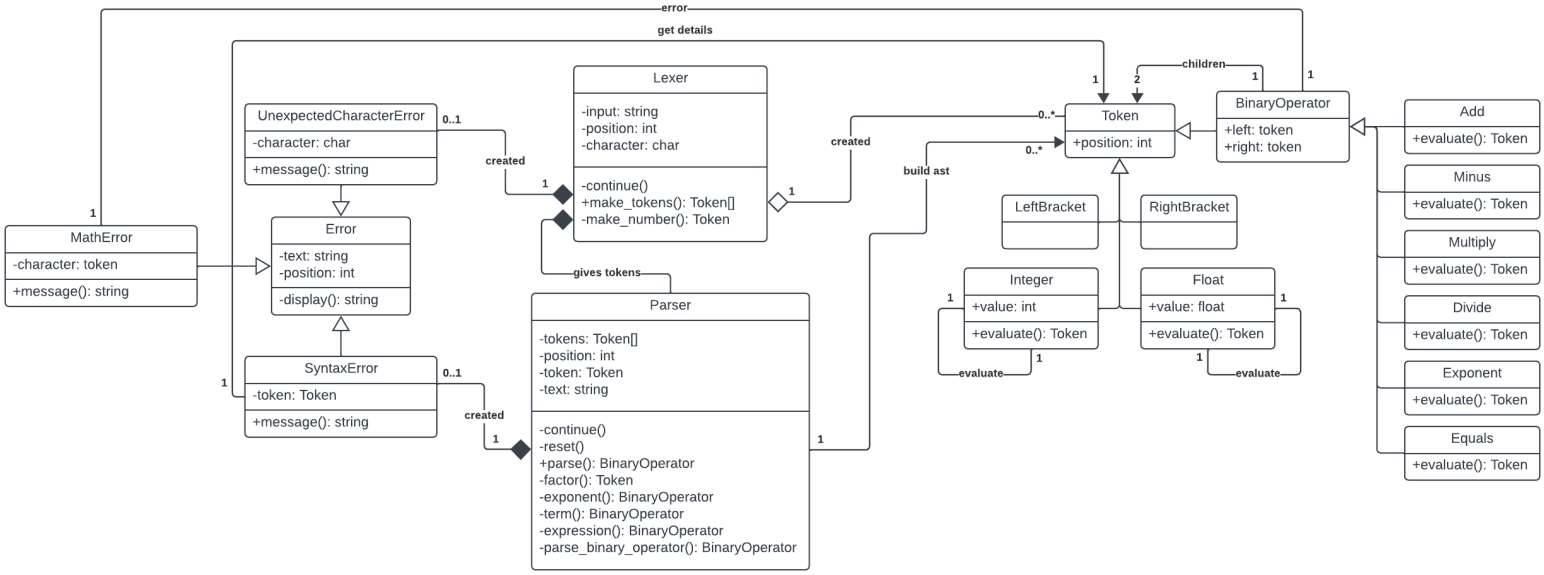
For example, for Integers and Floats, it will return their value property. Likewise, for any other data types that get added in the future, they will also return their own value.

However, for operators, their "value" is based off of the values of their two children. Therefore, all the operators will call evaluate() on both of their children and will use those two values and return the according value. For example, Plus will add together these two values, and return that.

This is use of abstraction allows the whole expression to be calculated. Each token just knows that its child value will return a number. Therefore, it could just be an Integer, or it could be another BinaryOperator or even a huge tree as one of its children, but it doesn't matter. It will form a huge chain of calling evaluate() on all the BinaryOperator nodes, and because all the leaves of the tree will be Integers or Floats, this means that a value will be returned to the root of the tree.

So, after the Parser has finished, evaluate() just needs to be called on the result, as this will be the root node of the tree, and therefore the corresponding value will be returned.

The UML diagram does not massively change for this, as it is mainly just adding evaluate() calls to all of the tokens, however the updates version is as shown:



MATH ERRORS

Some arithmetic operations are illegal. The two cases of this are attempting to divide by 0, as well as raising certain negative numbers to certain powers which create imaginary numbers.

Therefore, in the evaluate() method of Divide and Exponent, we will need check if this occurs, and if it does, we return the new Math Error. For Divide, if the result of evaluating the right child is zero, then it will be returned, and Exponent will just check if the combination of the two values will create a Number.

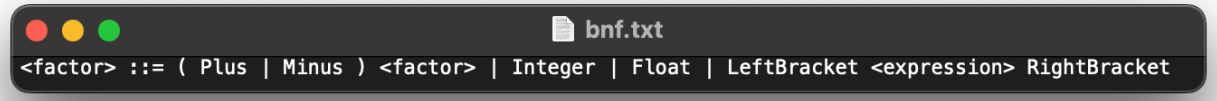
Now, because an Error can be returned from an evaluate() method, all the Binary Operators will have to check that the results of the two evaluate() methods it calls are not errors. If they are, it will have to return it too, much like how the Parser will return an error instead of building it into the AST.

This means that calling evaluate() on the root node will either return a value or an Error, so an additional check in the Shell class must be added so this error can be displayed if needed.

Finally, this new MathError class will essentially be the same as SyntaxError but with a different name, also accepting a token as a parameter, and with different error messages based on the token passed.

UNARY OPERATORS

Because I did not add them in the last section, I need to add unary operators to the factor() method, with the new BNF being as shown:



The changes here are that a <factor> can now be defined as a Plus or Minus followed by a <factor>, so these additional cases must be added.

➔ Plus <factor>

Because a plus in front of a number does not change the value, then whenever a Plus token is detected in factor(), it will continue() past it, and then call factor() again, and return that result.

➔ Minus <factor>

However, a Minus does change the value of the thing after it. So, when a Minus token is detected in factor(), it will continue() past it and call factor(), like for the Plus. However, instead of directly returning the result, it will create a new Minus operator, setting the left child to an Integer with value 0, and setting the right child to the result of calling factor(), and will return the Minus.

Therefore, in the AST, this will return the negative value of the number, so the unary operator will produce the correct value, even if it is using multiple tokens to do so.

I will also need to implement full error handling for both of these tests, but I will do that when developing, trying out different cases and adding fixes for them all.

DEVELOPMENT: EXECUTION

```
class Add extends BinaryOperator{
    constructor(position){
        super(position)
    }

    evaluate(){
        return this.left.evaluate() + this.right.evaluate()
    }
}
```

```
evaluate(){
    return this.value
}
```

Adding the execution itself is extremely simple, just adding each of the evaluate() methods, and you can see a few cases here.

The case just returning the value is for Integers and Floats, as per the Design,

```
evaluate(){
    return this.left.evaluate() - this.right.evaluate()
}
```

```
evaluate(){
    return this.left.evaluate() ** this.right.evaluate()
}
```

```
run(input){
    let tokens = new Lexer(input).make_tokens()
    if (tokens instanceof Error){
        console.log(tokens.message())
        return
    }
    let parsed = new Parser(tokens, input).parse()
    console.log(parsed instanceof Error ? parsed.message() : parsed.evaluate())
}
```

Now, the run() method in Shell just has to be slightly changed to call evaluate() on the result of the Parser (the rood node of the syntax tree), and then to output the result of this for testing purposes.

```
└─ node testing
 ERL ==> 2 + 2
4
 ERL ==> 2 ^ 3 * 7 - 4 + (2 + 3)
57
```

This immediately works, and as you can see it shows that the way the parser arranged the AST is correct, because the precedence and associativity of BODMAS is properly followed.

This is great in showing that the valid cases are now complete!

ERROR HANDLING

```
└─ node testing
 ERL ==> 17/0
Infinity
 ERL ==> (0-1)^0.5
NaN
 ERL ==>
```

Currently, when we try to perform our illegal operations, we just get returned with the default JavaScript values, which is not what I want for this project, as if a student sees Infinity or "NaN" (the Not a Number data type) it would be very confusing. Instead, I need to make these calculations return errors.

```
192  class MathError extends Error {
193      message(){
194          return "Zero Division Error"
195      }
196  }
```

```
run(input){
    let tokens = new Lexer(input).make_tokens()
    if (tokens instanceof Error){
        console.log(tokens.message())
        return
    }
    let parsed = new Parser(tokens, input).parse()
    if (parsed instanceof Error){
        console.log(parsed.message())
        return
    }
    let evaluated = parsed.evaluate()
    console.log(evaluated instanceof Error ? evaluated.message() : evaluated)
}
```

For now, I added a basic version of the MathError, without accepting a token or plaintext, because I have realised a major issue with how errors work.

```
evaluate(){
    let left = this.left.evaluate()
    if (left instanceof Error){
        return left
    }
    let right = this.right.evaluate()
    if (right instanceof Error){
        return right
    }
    if (right == 0){
        return new MathError()
    }
    return left / right
}
```

```
└─ node src.js
 ERL ==> 2/0
Zero Division Error
 ERL ==>
```

However, the check in Division's evaluate() method correctly detects when the program attempts to divide by 0.

ISSUES WITH ERRORS

Currently, the errors have no way of being able to receive the text that they need to use for the display() method. Previously, it was stored in both the lexer and the parser, but for errors in evaluation a different system is needed to have a global place to store the plaintext.

Otherwise, we would have to give every single instance of Token a copy of the plaintext, which would be very inefficient for the memory, so this global place to access the plaintext is needed.

```
// Global
const prompt = require('prompt-sync')()
const DIGITS = ["0123456789"]
let currentText = ""
```

```
run(input){
    currentText = input
```

My solution is to create a global variable called currentText, which will reflect the plaintext of the line that is currently being run.

Now, for all Error classes, rather than having the text passed as a parameter, the Error class sets the text property to the current value of currentText.

```
// Errors
class Error {
    constructor(position){
        this.text = currentText
        this.position = position
    }

    display(){
        return `${this.text}\n${' '.repeat(this.position)}^`
    }
}

class UnexpectedCharacterError extends Error {
    constructor(position, character) {
        super(position)
        this.character = character
    }
```

We therefore also update all subclasses of Error to no longer take in the plaintext and to only pass position to the super() call.

Additionally, I no longer need the Parser class to store a copy of the plaintext as it no longer needs to pass it for errors.

```
class Parser {
    constructor(tokens){
        this.tokens = tokens
        this.position = -1
        this.token = null
        this.continue()
    }
```

This seems like a much more logical approach in general.

```
} else {
    return new UnexpectedCharacterError(this.position, this.character)
}
```

Finally, all the old error messages must be updated to no longer pass in the text.

```
└─ node src.js
 ERL ==> 2 * * 2
 ! ERROR
Invalid Syntax: Expected literal
2 * * 2
    ^
 ERL ==> 2 2
 ! ERROR
Invalid Syntax: Expected operator
2 2
  ^
 ERL ==> 2.7.9
 ! ERROR
Unexpected Character: '.'
2.7.9
   ^
 ERL ==> help
 ! ERROR
Unexpected Character: 'h'
help
^
 ERL ==>
```

As shown on the left, all of the different errors still work properly, with their display() methods still displaying the text.

```
class MathError extends Error {
    constructor(token) {
        super(token.position)
        this.token = token
    }

    message(){
        return ` ! ERROR\nMath Error: Cannot divide by 0\n${this.display()}`
    }
}
```

```
if (right == 0){
    return new MathError(this)
}
```

Now the Math Error class can be properly implemented as shown, with the error token being the Division rather than the number, as I feel it is more logical than pointing to the number.

As shown on the right, this error handling now correctly functions.

```
└─ node src.js
 ERL ==> 2/0
 ! ERROR
Math Error: Cannot divide by 0
2/0
 ^
```

```
if (isNaN(left ** right)){
    return new MathError(this)
}
return left ** right
```

The implementation for preventing imaginary numbers is very similar, adding an case into Exponent's evaluate() method to check if the result of the power would be NaN.

I also added a new error message if the token which raised the Math Error was an Exponent for clarity.

```
message(){
    if (this.token instanceof Divide){
        return ` ! ERROR\nMath Error: Cannot divide by 0\n${this.display()}`
    } else if (this.token instanceof Exponent){
        return ` ! ERROR\nMath Error: Cannot raise negative numbers to this power\n${t
    }
    return ` ! ERROR\nMath Error\n${this.display()}`
}
```

Now the Math Errors are fully implemented.

```
└─ node src.js
 ERL ==> (0-1)^(1/2)
 ! ERROR
Math Error: Cannot raise negative numbers to this power
(0-1)^(1/2)
     ^
```

ERROR ISSUES

```
 ERL ==> 2 +
/Users/pw/ocr-erl-interpreter/src.js:178
        super(token.position)
                    ^

TypeError: Cannot read properties of null (reading 'position')
```

When running some tests, I realised that trying to write an incomplete expression caused the program to completely break, which therefore needed addressing.

```
if (self.token = null){
    return new SyntaxError(this.position)
}
```

The fix was simple enough: in parse_binary_operator(), after continuing past the operator, the current token must be checked if it is null, as otherwise it attempts to parse a non-existent token which causes lots of errors.

This addition is simple; however, it revealed a huge flaw with how the error handling in the program was implemented and led to me writing some very messy and very poor code.

This is because if I were to just pass the operator Token into the Syntax Error call, it would see that it was an operator and therefore would output that it expected a literal when this type of error is a different kind to having two operators in a row in the input. My message() presumed that there could only be one type of error per type of token, which is a huge flaw.

```
176   class SyntaxError extends Error {
177       constructor(token) {
178           if (token instanceof Token){
179               super(token.position)
180               this.token = token
181           } else {
182               super(token)
183               this.token = null
184           }
185       }
```

Instead of deciding to fix this, I instead decided that I would not pass a token but pass an integer if there was an incomplete input error, and this integer would correspond to the position of the operator token which was incomplete.

```
} else if (this.token == null){
    return ` ! ERROR\nInvalid Syntax: Incomplete input\n${this.display()}`
}
```

At first this bad implementation didn't even work, but after realising that I'd used both a single = and this instead of self after a lot of debugging output statements, it then worked.

```
└─ node src.js
 ERL ==> 2 +
 ! ERROR
Invalid Syntax: Incomplete input
2 +
^
```

```
if (self.token == null){
    return new SyntaxError(self.position)
}
```

```
└─ node src.js
 ERL ==> 2 +
 ! ERROR
Invalid Syntax: Incomplete input
2 +
  ^
```

Regardless, this solution was a bad idea – and just led to me being forced to redesign error handling later anyway. This to me felt extremely messy in the moment of writing it, but I procrastinated cleaning it up, which – with hindsight – is really poor development and something that I avoided doing moving forward.

UNARY OPERATORS

```
} else if (self.check_instance([Add])){
    self.continue()
    return self.factor(self)
} else if (self.check_instance([Minus])){
    let result = new Minus(self.position)
    result.left = 0
    self.continue()
    let right = self.factor(self)
    if (right instanceof Error){
        return right
    }
    result.right = right
    return result
}
return new SyntaxError(self.token)
```

Deciding to ignore the state of errors for now, I moved on to implementing Unary operators. My first attempt and implementing this was along the right lines, but with a lot of Errors. Firstly, I had was checking for Plus when my it should have been Add.

After fixing this, I tried to run the code and found that addition worked, but subtraction resulted in an error.

```
└─ node src.js
 ERL ==> 2 + + 2
4
 ERL ==> 2 - - 2
/Users/pw/ocr-erl-interpreter/src.js:47
        let left = this.left.evaluate()
                             ^

TypeError: this.left.evaluate is not a function
```

```
result.left = new Integer(self.position, 0)
```

This is because I was attempting to call evaluate() on just a basic JavaScript Number data type, rather than my custom Integer data type.

```
└─ node src.js
 ERL ==> 2 -- 2
4
 ERL ==> 2 --- 2
0
 ERL ==> 2 - - - - 2
4
```

```
 ERL ==> -7 + 5
-2
 ERL ==> 2*-7
-14
 ERL ==> 3^-6
0.0013717421124828533
 ERL ==> 2*-(3+9)
-24
```

After changing the left child to my own custom 0 and then it worked properly.

Now I had all the arithmetic features of the BODMAS stage implemented into the program, but before finishing I must deal with the absolute mess of error handling, I had left the program with, as it is currently unsustainable.

FIXING ERRORS

UNARY OPERATORS

```
 ERL ==> 2--------
 ! ERROR
Invalid Syntax: Incomplete input
2--------
^
```

When testing the new unary operator for errors, I realised that the error message pointed to the completely wrong position for the incomplete input, as I wanted the last minus token to be the error token.

```
└─ node src.js
 ERL ==> 1 + 2 + 3 +
6
 ! ERROR
Invalid Syntax: Incomplete input
1 + 2 + 3 +
      ^
 ERL ==> 1+2+3+
6
 ! ERROR
Invalid Syntax: Incomplete input
1+2+3+
      ^
```

When adding output messages to check this, I realised that it was not just for unary operators, but all incomplete inputs. As you can see, the pointer pointed to the same position, regardless of if there were spaces between the different tokens or not, which made me realise that the error was being passed the parser's position, rather than the token's position.

This was easy to fix, and adding cases for incomplete unary operators is just checking that the following token is not null.

```
} else if (self.check_instance([Add])){
    let errorToken = self.token
    self.continue()
    if (self.token == null){
        return new SyntaxError(errorToken.position)
    }
    return self.factor(self)
```

```
} else if (self.check_instance([Minus])){
    let errorToken = self.token
    self.continue()
    if (self.token == null){
        return new SyntaxError(errorToken.position)
    }
```

```
 ERL ==> 2+++++
 ! ERROR
Invalid Syntax: Incomplete input
2+++++
      ^
```

This did fix the issues; however this implementation is horrible, and I need consistency with always passing a token, so I am going to redesign all of Errors.

REDESIGN

```
class SyntaxError extends Error {
    constructor(token) {
        if (token instanceof Token){
            super(token.position)
            this.token = token
        } else {
            super(token)
            this.token = null
        }
    }

    message(){
        if (this.token instanceof LeftBracket) {
            return ` ! ERROR\nInvalid Syntax: '(' was never closed\n${this.display()}`
        } else if (this.token instanceof BinaryOperator) {
            // Or identifier in the future
            return ` ! ERROR\nInvalid Syntax: Expected literal\n${this.display()}`
        } else if (this.token instanceof Integer || this.token instanceof Float) {
            return ` ! ERROR\nInvalid Syntax: Expected operator\n${this.display()}`
        } else if (this.token == null){
            return ` ! ERROR\nInvalid Syntax: Incomplete input\n${this.display()}`
        }
        return ` ! ERROR\nInvalid Syntax\n${this.display()}`
    }
}
```

The main issue is with the really badly designed code on the left.

The code will receive the error token, and then from there will guess the type of error based on this token. This worked well initially because each token could only be involved in one type of error.

But now that the same kind of token can be involved in multiple different errors, this is not suitable at all, and I have had to use some weird workaround to pass an Integer instead of a Token if I want a specific kind of error, which is horrible implementation and will not be possible to reuse further into the project.

```
class SyntaxError extends Error {
    constructor(token, description='') {
        super(token.position)
        this.token = token
        this.description = description
    }

    message(){
        return ` ! ERROR\nInvalid Syntax: ${this.description}\n${this.display()}`
    }
}
```

The solution to this is giving the SyntaxError an additional parameter in its constructor, called description, which will take in a string to describe the error alongside the token, so no more guessing needs to take place.

This solution is a lot more expandable, as now the same token type can be in numerous errors.

This also significantly improves the readability of the program, as now there will be lots of string descriptions throughout the project, explaining why an error case could occur, rather than just a random Error being produced with no detail as to why it was an issue.

```
        if (self.check_instance([RightBracket])){
            self.continue()
            return result
        }
        return new SyntaxError(errorToken,  "'(' was never closed")
    } else if (self.check_instance([Add])){
        let errorToken = self.token
        self.continue()
        if (self.token == null){
            return new SyntaxError(errorToken.position, "Incomplete input")
        }
        return self.factor(self)
    } else if (self.check_instance([Minus])){
        let errorToken = self.token
        self.continue()
        if (self.token == null){
            return new SyntaxError(errorToken, "Incomplete input")
        }
```

```
message(){
    return ` ! ERROR\nMath Error: ${this.description}\n${this.display()}`
}
```

```
if (right == 0){
    return new MathError(this, "Cannot divide by 0")
}
```

```
parse(){
    let result = this.expression(this)
    if (result instanceof Error){
        return result
    }
    if (this.token == null){
        return result
    }
    return new SyntaxError(this.token, "Expected operator")
}
```

```
└─ node src.js
 ERL ==> 2 + 2 3 - 7
 ! ERROR
Invalid Syntax: Expected operator
2 + 2 3 - 7
      ^
 ERL ==> 2 * / 7
E
 ! ERROR
Invalid Syntax: Expected literal
2 * / 7
    ^
```

There are a few examples above of these changes, as it had to be done to the entire program in every single place that an error occurred.

I also updated the other error types to also use this system, such as MathError, so that it no longer has to base its message off of the token.

There were quite a lot of errors when initially testing the system, due to some forgotten updates, however I eventually got them all working.

```
if (self.token == null){
    return new SyntaxError(result, "Incomplete Input")
}
```

Now the incomplete input could just be given the token directly, rather than messy solution of the integer position being passed.

```
└─ node src.js
 ERL ==> 2 + 3 - 4 * 7 -
 ! ERROR
Invalid Syntax: Incomplete Input
2 + 3 - 4 * 7 -
              ^
```

This just made the code so much more consistent and logical, and I was finally happy with the way that the errors were being managed, as it made a lot more sense.

This is also positive for the rest of development, as if these changes were not implemented, there would be major issues later on.

LESS REPEATED CODE

My last issue is that there is a lot of repeated code in the error checking for the Binary Operator's evaluate() methods, because every single method includes evaluating the left and right children and checking them both for errors which adds the same set of repeated code in every single class.

```
check_for_errors(left, right){
    if (left instanceof Error){
        return left
    }
    if (right instanceof Error){
        return right
    }
    return null
}
```

My solution for this is creating a new method in the Binary Operator class called check_for_errors(), which takes in two inputs: the result of the left evaluated() method and the result of the right, and will return an error if there is one, and null otherwise.

This will allow every class to easily check for errors with less repeated lines: returning the operation if the result is null, and the result if it isn't.

```
evaluate(){
    let left = this.left.evaluate()
    let right = this.right.evaluate()
    let result = this.check_for_errors(left, right)
    return result == null ? left + right : result
}
```

An example for Plus's evaluate() method is shown on the left, and it is clear how much this reduces the number of lines used.

Now, this just must be replicated for every single binary operator so they all use this system.

```
evaluate(){
    let left = this.left.evaluate()
    let right = this.right.evaluate()
    let result = this.check_for_errors(left, right)
    if (result != null){
        return result
    }
    if (right == 0){
        return new MathError(this, "Cannot divide by 0")
    }
    return left / right
}
```

For operators that require additional checks, they just perform this check after the full error checking has been completed, as shown for Division on the left.

When testing this, it still worked perfectly as expected: this was just a small addition to reduce repeated code and didn't impact the result at all which is perfect. I still feel this could be improved, as there are still a lot of repeated lines, but it is a lot better than before.

Finally, before concluding this stage of development, I just added some comment to some of the less self-explanatory methods I implemented to make it easier for myself in the future to re-read bits of code. My aim with writing code is to make it obvious based on just the code itself, but this extra level of detail is useful

```
// Ensure that if an error has been returned it is not involved in a calculation
check_for_errors(left, right){
```

```
main(){ // Can only be run in terminal as inputs don't work in VScode
    let input = prompt(" ERL ==> ")
    while (input != "QUIT()"){
```

```
// Takes in an array and checks if current token is instance of the items
check_instance(check) {
    for (let item of check){
        if (this.token instanceof item){
            return true
        }
    }
    return false
}
```

This concludes this stage of development, and I am relatively happy with the system after this phase.

```
} else if (self.check_instance([Add])){ // Add unary operator
    let errorToken = self.token
    self.continue()
    if (self.token == null){
        return new SyntaxError(errorToken.position, "Incomplete input")
    }
    return self.factor(self)
} else if (self.check_instance([Minus])){ // Minus unary operator
    let errorToken = self.token
    self.continue()
    if (self.token == null){
        return new SyntaxError(errorToken, "Incomplete input")
    }
    let right = self.factor(self)
    if (right instanceof Error){
        return right
    }
    let result = new Minus(self.position) // Just adds minus node of 0 - value
    result.left = new Integer(self.position, 0)
    result.right = right
    return result
} // Two operators in a row
return new SyntaxError(self.token, "Expected literal")
```

EVALUATION

After each phase of development, I just want to briefly go over the success criteria of that phase, to ensure that I am not ignoring any features, and know what I must come back to afterwards.

No.	*Criteria*	*Implemented?*
1.1	Allow for Addition, Subtraction, Multiplication, Division, Exponentiation, Modulus and Quotient division to be taken of two Integers	**No**
1.2	Allow multiple operations to be chained together, following the BODMAS order of operations.	**Yes**
1.3	Allow for Floats to be used and created, and for any calculation involving Floats to return another Float	**Yes**
1.4	Allow brackets and infix operators + and – to be used in expressions	**Yes**
1.5	Proper error handling is implemented, ensuring division by zero, unexpected characters and invalid syntax errors are managed	**Yes**

The issue with 1.1 is with Modulus and Quotient. These operations, instead of using symbols, use the keywords MOD and DIV to function. Therefore, I have decided to implement them later, after I have implemented keywords into the Lexer, as I feel like a temporary solution, I implement now would not be great.

However, because of the way I have built the project in this phase, I am not at all worried about their implementation, and it should be very straightforward to add them based on the current system: adding them to the array of valid tokens in the corresponding method that calls parse_binary_operator(), and adding their tokens to the Lexer and having the calculation code.

CODE QUALITY

At the end of this module, I am happy with how well the code has been written. Throughout, I had lots of periods of writing poor fixes to code rather than just redesigning the system, which I think is important to take forwards throughout the project: it is better to just do the hard work of re-doing a system than just implementing some poor system. In this section, this was my terrible error handling, however by fix at the end seems very good, and should be usable for the remainder of the project.

STAKEHOLDERS

At the end of some modules, I will show my stakeholder, Tanish Arjaria, my progress to ask for feedback. At this point of the program, although I have done a lot of work, there is not much to show for it.

I did still show him my progress, however because it is just a calculator, there was not really any useful feedback to report on his end because it is currently just the foundation of the project. In the future modules, I will try and report better on what he thinks as it may be useful, however in the nature of the strictly defined ERL code, there will probably not be much feedback.

VARIABLES

DESIGN

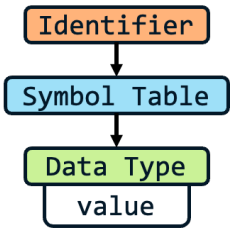
ROUGH DESCRIPTION

This is an initial abstracted version of the plan as there are several ways I could approach this.

The identifier is the name representing a variable. Therefore, I will have a new Identifier Class, which will be created in the Lexer. It will have a name property, which is a string representing it. For example, if a variable called "var" was referenced, an Identifier with the name property as "var" would be produced.

What is important is that an Identifier is not the same as the variable itself: the same variable can be referenced in multiple different locations, but each time it is referenced, it will be by a separate Identifier, but with the same name property. Therefore, we need a way to store these values so that every single Identifier with the same name can access the same location.

This is where the new Data Type class is used, with each instance having a name and value property, with the value being able to be accessed and changed. This is representative of our memory location: with the name aligning with the Identifiers that access it, and the value being self-explanatory.



The Data Types will be stored and created by a class called Symbol Table, which will include an array of all its children Data Types. For now, there will just be a single, global Symbol Table, and all new variables will be added to it. In the future creating multiple tables allows scope for functions and procedures to have their own local variables, so I am ensuring this system can scale for later.

Each Symbol Table will have a method which will take in a string, being the name of the identifier, and will search through their array and return the Data Type with the corresponding name. so that the value can be accessed, otherwise this method will create a new Data Type with the corresponding name that was entered into the Symbol Table method, with null as the initial value for newly created Data Types.

A new Identifier Error will be added if a variable with value null is trying to be accessed, as it means that it has not been declared in an assignment.

ASSIGNMENT

Before the parsing can be added, a new Equals binary operator needs to be added. This will have the expected position property and will have two children like the other operators.

parse() will now call this new assignment() method by default, presuming that any input will be an assignment. assignment() will then include checks to see if it is actually an assignment.

If it doesn't start with an Identifier, or if there is no Equals in the second position, then assignment() will call expression() instead, and return that, as that is now the correct method to be used. A new reset() method will be called before calling expression(), which will revert the parser back to the original token.

If it is an assignment, then the following BNF will be used to create the syntax tree:

```
<assignment> ::= <Identifier><Equals><expression>
```

This is relatively straightforward but it will need to check for every possible invalid case, with lots of opportunities to return different Syntax Errors throughout/

At the end, this will return an Equals token, with the Identifier as its left child and an expression ast as its right child. From here, when the line is executed, the Equal's evaluate() method will be called.

The Equals evaluate() method will call upon the Identifier's evaluate(), which will call the global Symbol Table's finding method, and will return the corresponding Data Type, which the Identifier will return. The Equals will then set the value property of this child to the value from its right child's evaluate() method, which would return the value of the expression, setting the Data Type's.

CONSTANTS

For this, a new TemplateKeyword class will be created. This will be used a lot throughout the program, as there are lots of keywords that will not contain any code themselves, but still provide important feedback on how the abstract syntax tree should be arranged.

These will have a tag property which will represent their type, for example when the Lexer detects a string of characters "const", it will create a TemplateKeyword token with the tag "const".

This will then be recognised by the Parser, and if it is present, then the assignment() method will set the Identifier's new constant method to true (which will be false by default). When evaluate() is called on that Identifier, it will be passed through the Symbol Table, to the Data Type.

If a Data Type ever receives constant being true, it will then record this in its own constant method, and will no longer allow its value to be changed, returning an Identifier Error if there is an attempt to do so.

BASIC IMPLEMENTATION

Now that I have roughly planned out the system logically, I will specify more precisely how this will be implemented.

LEXER CHANGES

Firstly, a new global constant called LETTERS will be needed, like our DIGITS example for the Lexer to use.

➔ New classes

Firstly, the new Identifier class will have to be created, with a name property and a constant property (set to false), alongside the new TemplateKeyword class, which will have a tag property. Both will extend Token.

Equals will be the new class for the '=' operator, extending Binary Operator, so will have a position and two children. It will be made in make_tokens() like the other property.

➔ make_identifier()

This will be a method in the Lexer class, to be called in make_tokens() when the current character is in LETTERS. It will then continue cycling through the characters until a character than is not in DIGITS or LETTERS is encountered, keeping a track of the character that it passes through. It will then return a new identifier with the name property being these characters.

This will be done using an array called name, where the letters will be pushed to before finally being converted to a string. Also, the original position needs to be passed, so will have to be recorded at the start of make_identifier().

Finally, there will be have to be checking on the name, so that if the name is equal to "const", instead of returning an Identifier, it will return a TemplateKeyword with "const" as the tag. This will be expanded massively later in the program, to account for all of the different keywords, but for now it will just be a single check for "const". I will probably use a switch statement for this.

DATA TYPE

This will be a generic class for storing a data type in the symbol table (as later the symbol table will be use. It will have a property for the name as a string and value property. They will also have a constant Boolean property, and where this is true, they will have a declared property so that constants can keep track of whether they have been defined (to ensure they can only be defined once).

➔ value getter

I am going to use getter and setter methods for this class, as they are something I have not experimented with before. The getter will check if value's current value is null, which would represent an undefined variable. If it is null, it will return an Identifier Error, otherwise it will return the value.

➔ value setter

If the constant property is false, then it will assign the new value. Otherwise, it will first check if it has been declared (based on the property). If it has, it will return an Identifier Error stating that it has already been assigned. If not, it then sets declared to true and then assigns the new value.

SYMBOL TABLE

The only property will be an array called table to store all the Data Types.

➔ find()

Find will take in the details of the Identifier and will search through the array of Data Types to find one with a corresponding name and will return it. If one does not exist, it will create a new one, adding it to the array, and creating it with the same details as the Identifier: the same name and details about whether it is a constant. This new Data Type will be returned from the method.

The Identifier's evaluate() method will call upon find(), passing it details, and then Interpreter will return the corresponding Data Type with its details.

PARSER CHANGES

I have described a lot of these changes, but to reiterate:

➔ parse() will now call assignment()
➔ reset() will set the position property to -1, and then call continue(), which will reset the tokens.

➔ assignment()

This will consist of a lot of checks for properties, with the following format:

- Check the first token is an Identifier, if not then call expression() and return it.
- Store the identifier in a temporary variable.
- continue() and check if the next token is an Equals, if it is then store it.
- If it isn't an Equals, call reset() and then call expression() and return it.
- Otherwise continue() and call expression(), returning the error if one is made.
- Set the identifier to the left child of the Equals, and the expression root to the right child.
- Return the Equals token.

This is the general format. An additional check is needed at the start to see if a TemplateKeyword is present. If it is, and the tag is "const", this will be remembered, and when the Identifier is reached, its constant property will be set to true.

I have not covered all of the possible invalid cases here, but during development I will test for all of these.

DEVELOPMENT

INITIAL ADDITIONS

CHARACTERS

```
4  console.log([...Array(26)].reduce(a=>a+String.fromCharCode(i++),'',i=97))
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/src.js"
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

I initially tried to generate the list of LETTERS from a line I found on the URL:

https://codegolf.stackexchange.com/questions/71613/generating-the-alphabet-in-javascript

```
4  console.log(...[...Array(52)].reduce(a=>a+String.fromCharCode(i==90?i=97:i++),'',i=65))
A B C D E F G H I J K L M N O P Q R S T U V W X Y a a b c d e f g h i j k l m n o p q r s t u v w x y z
5  console.log(...[...Array(52)].reduce(a=>a+String.fromCharCode(i==91?i=97:i++),'',i=65))
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a a b c d e f g h i j k l m n o p q r s t u v w x y
4  const LETTERS = [..."qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM"]
```

This worked for just uppercase, but when trying to expand it to include lower I could not alter it to work.

Therefore, I went for the more basic option of unpacking a string of every character into the list.

LEXER CHANGES

```
142  class Identifier extends Token{
143      constructor(position, name){
144          super(position)
145          this.name = name
146      }
147  }
```

For now, Identifier will not include the constant tokens, and neither the rest of the system, as I am going to add all the constant implementation later in this module.

All the lexer change can be seen.

```
} else if (LETTERS.includes(this.character)){
    tokens.push(this.make_identifier())
    continue
} else if (this.character == '+') {
```

```
make_identifier(){
    let name = []
    let position = this.position
    while (LETTERS.includes(this.character) || DIGITS.includes(this.character)){
        name.push(this.character)
        this.continue()
    }
    return new Identifier(position, name.join(''))
}
```

My two test cases correctly worked, showing how variable names are correctly parsing.

```
412  console.log(new Lexer("2 + name - 7").make_tokens())
```

```
[
  Integer { position: 0, value: 2 },
  Add { position: 2, left: null, right: null },
  Identifier { position: 4, name: 'name' },
  Minus { position: 9, left: null, right: null },
  Integer { position: 11, value: 7 }
]
```

```
412  console.log(new Lexer("name + age1 - 2 + hi7test").make_tokens())
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/src.js"
[
  Identifier { position: 0, name: 'name' },
  Add { position: 5, left: null, right: null },
  Identifier { position: 7, name: 'age1' },
  Minus { position: 12, left: null, right: null },
  Integer { position: 14, value: 2 },
  Add { position: 16, left: null, right: null },
  Identifier { position: 18, name: 'hi7test' }
]
```

SYMBOL TABLE AND DATA TYPE

```
156  class DataType {
157      constructor(){
158          this._value = null
159      }
160
161      get value(){
162          if (this._value == null){
163              console.log("Error")
164              return
165          }
166          return this._value
167      }
168
169      set value(newValue){
170          this._value = newValue
171      }
172  }
```

This was my initial experimentation with using getters and setters. I realised that value had to be renamed to _value, as otherwise it clashed with the property names. The "Error" output is temporary, and will be replaced with IdentifierErrors later, when they get implemented.

```
437  variable = new DataType()
438  variable.value = 7
439  console.log(variable.value)
```

```
[Runni
7
```

The tests were successful.

```
437  variable = new DataType()
438  console.log(variable.value)
```

```
Error
undefined
```

I also had to add a name property, which I do not have a screenshot of, but this is declared in the constructor() method based on the argument passed to it, which will be seen in SymbolTable.

```
150  class SymbolTable {
151      constructor(){
152          this.table = []
153      }
154
155      find(name){
156          for (item of this.table){
157              if (item.name == name){
158                  return item
159              }
160          }
161          return new DataType(name)
162      }
163  }
```

Now implementing the Symbol table.

```
447  let test = new SymbolTable()
448  test.find("hello") = 7
449  console.log(test.find("hello"))
```

The initial test did not work, but the issue was that the find() call returns the Data Type, but .value still then needs to be called to change the value, so the correct version is:

```
test.find("hello").value = 7
```

The next issue was that the newly created Data Types were not added to the table list, which was a quick fix: pushing the new DataType to the end of the array, and then returning this value.

```
return this.table[this.table.length - 1]
```

INTEGRATION WITH EXISTING CODE

```
5    let global = new SymbolTable()
```

Firstly, global is declared as a global variable for the whole program to use – this is temporary until functions are implemented.

```
110  class Equals extends BinaryOperator{
111      constructor(position){
112          super(position)
113      }
114  }
```

```
} else if (this.character == '=') {
    tokens.push(new Equals(this.position))
} else if (this.character == '(') {
```

Then, the basic Equals instance is created, with its corresponding character check in the Lexer's make_tokens()

```
factor(self){
    if (self.check_instance([Integer, Float, Identifier])){
        let result = self.token
        self.continue()
        return result
```

Finally, the factor() method must be changed to also include Identifiers alongside Integers and floats, so that they can be parsed and included into expressions. This just needs Identifier to be added to the array of valid tokens.

```
458  console.log(new Parser(new Lexer("1 + variable * 3").make_tokens()).parse())
```

```
Add {
  position: 2,
  left: Integer { position: 0, value: 1 },
  right: Multiply {
    position: 13,
    left: Identifier { position: 4, name: 'variable', value: undefined },
    right: Integer { position: 15, value: 3 }
  }
}
```

When testing, the correct abstract syntax tree was produced. Initially, there were some errors about Symbol Table being referenced before assignment, but this was because I'd defined global at the start of the program, so I moved Symbol Table and Data Type to be at the start of the file.

To let it evaluate, Identifier must be given an evaluate() method.

```
get value(){
    return global.find(this.name).value
}
```

This is done by calling find() on the global symbol table, passing its name, and returning the value of the Data Type.

```
465  global.find("variable").value = 7
466  console.log(new Parser(new Lexer("1 + variable * 3").make_tokens()).parse().evaluate())
```

22

Because assignment had not been implemented yet, I had to manually create the variable using my own code, but after this, the correct value was created, so variables can be used within expressions/

ASSIGNMENT

```
reset(){
    this.position = -1
    this.continue()
}
```

I started by creating the reset() method, and changed parse() to call assignment() by default now, instead of expression().

```
parse(){
    let result = this.assignment(this)
```

```
assignment(self){
    if (!this.token instanceof Identifier){
        return self.expression(self)
    }
    let left = this.token
    self.continue()
    if (!this.token instanceof Equals){
        this.reset()
        return self.expression
    }
    let result = this.token
    self.continue()
    let right = this.expression(self)
    if (right instanceof Error){
        return right
    }
    result.left = left
    result.right = right
    return result
}
```

This was my initial attempt of implementing assignment().

For now, this excludes error handling until they are added, but it is clear to see the expected for how an assignment is laid out. The checks for Identifier and Equals ensure that a normal expression can still be parsed, even if it starts with an Identifier, resetting if need be.

```
console.log(new Parser(new Lexer("variable = 7").make_tokens()).parse())
```

```
Equals {
  position: 9,
  left: Identifier { position: 0, name: 'variable' },
  right: Integer { position: 11, value: 7 }
}
```

```
console.log(new Parser(new Lexer("variable + 7").make_tokens()).parse())
```

```
Add {
  position: 9,
  left: Identifier { position: 0, name: 'variable' },
  right: Integer { position: 11, value: 7 }
}
```

The two test cases confirm that assignments and normal expressions are still correctly produced, and therefore that the reset() method correctly works.

EVALUATION

```
evaluate(){
    let left = this.left.evaluate()
    let right = this.right.evaluate()
    let result = this.check_for_errors(left, right)
    if (result != null){
        return result
    }
    left = right
}
```

The initial attempt at creating an evaluate() method for Equals is shown on the left, but upon trying to run the case:

```
variable = 7
```

It resulted in the outputted "Error" being printed from the Data Type get method.

```
Error
undefined
```

This therefore means that the identifier's evaluate() call is returning the value of the Data Type, rather than the Data Type instance itself. However, I cannot change evaluate() to return just the Data Type, as then Identifiers would not properly evaluate in arithmetic expression, so therefore a new method – similar to evaluate() - will have to be added to Identifier to allow both of these cases.

```
195  class Identifier extends Token{
196      constructor(position, name){
197          super(position)
198          this.name = name
199      }
200
201      evaluate(){
202          return global.find(this.name).value
203      }
204
205      assign(){
206          return global.find(this.name)
207      }
208  }
```

assign() will be the new method called be Equals to get access to the Data Type instance itself.

```
evaluate(){
    let left = this.left.assign()
    let right = this.right.evaluate()
    let result = this.check_for_errors(left, right)
    if (result != null){
        return result
    }
    left.value = right
}
```

Now, Equals will call assign() on its left child, and will then set the value of the resulting Data Type to it's value.

Initially, I had an issue where I just set left to right, but after fixing that it should now work.

```
501  let variable = new Identifier(0, "name")
502  let number = new Integer(3, 5)
503  let equals = new Equals(1)
504  equals.left = variable
505  equals.right = number
506  equals.evaluate()
507  console.log(global.find("name"))
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/src.js"
DataType { name: 'name', _value: 5 }

[Done] exited with code=0 in 0.169 seconds
```

When setting up a very fabricated test case, this worked perfectly, and the value in the Symbol Table was correctly updated with the new value.

However, when trying to run a test case directly from plaintext, an error was returned, which was confusing but meant that something was wrong with the parsing process, because if a correct AST was produced then it should work just like this fabricated test, so now a lot of debugging is going to be needed.

DEBUGGING

```
assignment(self){
    if (!(this.token instanceof Identifier)){
        return self.expression(self)
    }
    let left = this.token
    self.continue()
    if (!(self.token instanceof Equals)){ {
        self.reset()
        return self.expression(self)
    }
    let result = this.token
    self.continue()
    let right = this.expression(self)
    if (right instanceof Error){
        return right
    }
    result.left = left
    result.right = right
    return result
}
```

This debugging process took a while, involving a lot of output statements to track how the parser was progressing.

The problem was with the assignment() method. More specifically, the checks to see whether the stream of tokens was an assignment or an evaluation.

My testing made me realise that I did not understand the order of operations in JavaScript – ironic when building an interpreter. When checking if the current token was an instance of a Class, I had initially but the NOT operator (!) immediately next to this.token, however because the Precedence of ! is higher than that of instanceof, it meant that I was negating the Token, and then comparing that, which obviously was not working as expected, and because JavaScript is so loose with types, no errors were flagged.

The solution is moving the ! outside of brackets which contain the instanceof check. This therefore performs the check first, and then returns the opposite, which was as intended.

```
└─ node src.js
 ERL ==> 2
2
 ERL ==> a=5
undefined
```

This did fix the issue, as now assignments no longer crashed the program, however it revealed more issues with my code which must now be fixed for the solution to be complete.

SMALL CHANGES

```
left.value = right
return null
```

```
if (this.token == null){
    return null
}
```

```
if (parsed == null){
    return
}
```

```
└─ node src.js
 ERL ==> a=5
 ERL ==> a
5
 ERL ==> a + 2
7
 ERL ==> a = 7
 ERL ==> a
7
 ERL ==> a = a + 2
 ERL ==> a
9
```

There are 2 slight changes made.

Firstly, changing Equal's evaluate() method to return null, so that undefined would not be outputted after an assignment.

Additionally, adding extra lines to the parser to return null if an empty stream of tokens are inputted, as previously an empty instruction would cause the program to crash. This also has to be accounted for in the temporary runner.

```
ERL ==>
ERL ==>
ERL ==>
```

Overall, it is now clear that assignment works as intended, and variables can be both defined and referenced to work as intended, as shown by the tests.

ERROR HANDLING

```
257  class IdentifierError extends Error{
258      constructor(token, description='') {
259          super(token.position)
260          this.token = token
261          this.description = description
262      }
263
264      message(){
265          return ` ! ERROR\Identifier Error: ${this.description}\n${this.display()}
266      }
267  }
```

Identifier Error is essentially a copy paste of Syntax Error with the message changed slightly.

This may be inefficient as there is lots of reused code, so this could be later changed to incorporate all the types into one.

The current issue is that Data Types will need access to the last token where they were accessed to create an Identifier Error, but currently they just receive the name. So now I must redesign aspects of the program to allow the Identifier token to be passed through evaluate(), through find() in Symbol Table so that DataType can store the token in a new lastAcessed property, which will therefore allow it to be passed into any errors that occur.

```
find(token){
    for (let item of this.table){
        if (item.name == token.name){
            item.lastReferenced = token
            return item
        }
    }
    this.table.push(new DataType(token))
    return this.table[this.table.length - 1]
}
```

Now, find() has been modified for this new system. So, whenever an existing token is accessed, the lastReferenced property will be updated with the new Identifier property.

Additionally, when a new token is created, the token will be passed as an argument instead of the string name. Therefore, DataType's constructor is changed to use this token: setting the name to the token's name property, and lastReferenced to the argument token.

```
class DataType {
    constructor(token){
        this.name = token.name
        this._value = null
        this.lastReferenced = token
    }

    get value(){
        if (this._value == null){
            return new IdentifierError(this.lastReferenced, "was not declared")
        }
        return this._value
    }
```

```
evaluate(){
    return global.find(this).value
}

assign(){
    return global.find(this)
}
```

This also requires Identifier to be updated, to now pass itself into find(), instead of its name property. However, after these changes, the system should work as it did before.

Now the actual error must be added itself, which is in value(), and is returned if the value is null.

```
message(){
    return ` ! ERROR\nIdentifier Error: '${this.token.name}' ${this.description}\n${this.display()}`
}
```

I slightly update the message() property to output the name of the Identifier.

```
 ERL ==> 2 + a - 3
 ! ERROR
Identifier Error: 'a'  was not declared
2 + a - 3
    ^
 ERL ==> a = 5
 ERL ==> a
5
```

Now, proper errors are returned whenever an undeclared variable is used, as shown.

Additionally, some of the pre-existing error handling covered some other cases, so an error is also returned if multiple Equals are used, requiring no changes to do so.

```
 ERL ==> a = 1 = 3
 ! ERROR
Invalid Syntax: Expected operator
a = 1 = 3
      ^
```

CONSTANTS

LEXER IMPLEMENTATION

```
212  class Keyword extends Token{
213      constructor(position){
214          super(position)
215      }
216  }
217
218  // For generic keywords like const, global which don't need a unique object
219  class TemplateKeyword extends Keyword{
220      constructor(position, tag){
221          super(position)
222          this.tag = tag
223      }
224  }
```

```
name = name.join('')
switch (name) {
    case "const":
        return TemplateKeyword(position, "const")
    default:
        return new Identifier(position, name)
}
```

This is completely following the plan, But I made TemplateKeyword extend a Keyword class, which itself extended token.

This was just because I thought it could be useful within the future to have a shared class, but for now it serves no purpose.

```
549  console.log(new Lexer("const age = 7").make_tokens())
```

```
[
  TemplateKeyword { position: 0, tag: 'const' },
  Identifier { position: 6, name: 'age' },
  Equals { position: 10, left: null, right: null },
  Integer { position: 12, value: 7 }
]
```

After changing the Lexer to include the new keyword, then the case worked as expected.

EVALUATION

```
209  class Identifier extends Token{
210      constructor(position, name, constant=false){
211          super(position)
212          this.name = name
213          this.constant = constant
214      }
```

I decided to add the code into the classes before adding the parser changes.

This started by giving Identifier the new constant property, which by default is set to false.

```
set value(newValue){
    if (!this.constant){
        this._value = newValue
        return
    }
    if (this.declared){
        return new IdentifierError(this.lastReferenced, "is a constant and has already been defined")
    }
    this.declared = true
    this._value = newValue
}
```

```
class DataType {
    constructor(token){
        this.name = token.name
        this._value = null
        this.lastReferenced = token
        this.constant = token.constant
        if (this.constant){
            this.declared = false
        }
    }
```

The additions to the classes themselves should be relatively simple, however I need to add the parser changes before I can properly test the system

PARSING

```
parse(){
    // Check if there are no tokens
    if (this.token == null){
        return null
    }
    // Check if there is a tagged assignment
    if (this.token instanceof TemplateKeyword){
        let result = this.assignment(this, this.token.tag)
        return this.check_result(result) ? result : new SyntaxError(this.token, "Expected operator")
    }
    // Check if it is a normal assignment
    if (this.token instanceof Identifier){
        this.continue()
        let token = this.token
        this.reset()
        if (token instanceof Equals){
            let result = this.assignment(this)
            return this.check_result(result) ? result : new SyntaxError(this.token, "Expected operator")
        }
    }
    // Left over case is just an expression
    let result = this.expression(this)
    return this.check_result(result) ? result : new SyntaxError(this.token, "Expected operator")
}
```

I have decided to move the checking for whether the assignment is a constant or not into the parse() method, as it provides more consistency. Therefore, every time assignment() is called it will be one, and not potentially an expression().

The updated parse() is shown, and now whenever “const” is encountered, assignment() will be called with “const” as an extra argument.

The new check_result() method is just a way to reduce code repetition: if it returns True, then either an error has been returned, or the current token is null: so the result of the previous method can be returned. Otherwise, it means that there is no error and the current token is not null, so therefore the whole stream of tokens has not been parsed, so there are two consecutive operators, so a Syntax Error is returned.

```
assignment(self, tag=null){
    if (tag != null){
        this.continue()
    }
    let left = self.token()
    if (tag = "const"){
        left = new Identifier(left.position, left.name, true)
    }
    self.continue()
    let result = self.token()
    if (!(result instanceof Equals)){
        return new SyntaxError(result, "expected '='")
    }
    self.continue()
    let right = self.expression(self)
    if (right instanceof Error){
        return right
    }
    result.left = left
    result.right = right
    return result
}
```

The updated assignment method() will replace the Identifier with a new one with the constant property set to true. I realise with hindsight that the constant property could have just been modified, but this longer solution still works.

After fixing a small syntax error where I used ‘=’ instead of ‘==’, the program no longer crashed so I could run some test cases.

However, when running some test cases, the system was completely broken, as shown by the very unusual set of test cases that are below.

```
└─ node src.js
 ERL ==> 2 + 2
4
 ERL ==> a = 7
 ERL ==> a
7
 ERL ==> const b = 5
 ERL ==> b
5
```

```
 ERL ==> b = 6
 ERL ==> b
5
 ERL ==> b = 20
 ERL ==> b
5
 ERL ==> a = 8
 ERL ==> a
7
```

```
└─ node src.js
 ERL ==> a = 5
 ERL ==> a
5
 ERL ==> a = 7
 ERL ==> a
7
 ERL ==> const b = 5
 ERL ==> b
 ! ERROR
Identifier Error: 'b' was not declared
b
^
```

In order to discover the issue, I added some output statements

```
set value(newValue){
    console.log("before", this.constant, this.declared)
    if (!(this.constant)){
        this._value = newValue
        return
    }
    if (this.declared){
        return new IdentifierError(this.lastReferenced, "is a constant and has already been defined")
    }
    this.declared = true
    this._value = newValue
    console.log("after", this.constant, this.declared, this.value)
}
```

```
└─ node src.js
 ERL ==> const a = 5
before true false IdentifierError {
  text: 'const a = 5',
  position: 6,
  token: Identifier { position: 6, name: 'a', constant: true },
  description: 'was not declared'
}
 ERL ==>
```

Somehow a "was not declared" Identifier Error occurred, which was only present in the getter function.

Therefore, somehow in the input "const a = 5", a getter was being obtained, which was very confusing

Additionally, the after output never occurred, which meant that somewhere in the setter method it was stopping prematurely, which it shouldn't be.

THE FIX

Eventually I realised this was because I was using a setter system, rather than an actual method. Therefore, when an error occurred in the setter method, it was not properly returned, because setters should not return values, which I think led to the really weird test results that I saw.

My solution to this is replacing the value setter with a method called set(), with the same functionality as the old setter. Instead, it takes in the new value as a parameter, and will have to be called slightly differently, but now it can return errors which can be checked for.

```
set(newValue){
    console.log("before", this.constant, this.declared)
    if (!(this.constant)){
        this._value = newValue
        return null
    }
    if (this.declared){
        return new IdentifierError(this.lastReferenced, "is a constant and has already been def
    }
    this.declared = true
    this._value = newValue
    console.log("after", this.constant, this.declared, this.value)
    return null
}
```

```
evaluate(){
    let left = this.left.assign()
    let right = this.right.evaluate()
    let result = this.check_for_errors(left, right)
    if (result != null){
        return result
    }
    return left.set(right)
}
```

The small changes can be seen above and are not any drastic changes to the code.

```
└─ node src.js
 ERL ==> const a = 5
 ERL ==> a
5
 ERL ==> a = 7
 ! ERROR
Identifier Error: 'a' is a constant and has already been defined
a = 7
^
 ERL ==> const a = 8
 ! ERROR
Identifier Error: 'a' is a constant and has already been defined
const a = 8
      ^
```

However, as shown on the left, the system now works completely as expected: with constants no longer being able to be re-defined after being set.

However, if the constant keyword is used for a second time, an error is still returned. I have not planned how I want this to work, so I will return to this later and decide.

CLEANING UP

```
 ERL ==> a =
/Users/pw/ocr-erl-interpreter/src.js:264
        super(token.position)
                    ^

TypeError: Cannot read properties of null (reading 'position')
```

Some of the cases where an invalid input had been entered causes the program to crash, such as nothing following the equals sign, as well as no Identifier being present after the const keyword.

```
assignment(self, tag=null){
    let errorToken
    if (tag != null){
        errorToken = this.token
        self.continue()
    }
    let left = self.token
    if (left == null){
        return new SyntaxError(errorToken, `epected identifier after '${errorToken.tag}'`)
    }
```

```
errorToken = self.token
self.continue()
if (self.token == null){
    return new SyntaxError(errorToken, "Incomplete input")
}
let right = self.expression(self)
```

My solution was to add additional checks to see if the current token is null throughout assignment() to avoid any of these cases.

These should then hopefully catch these invalid cases and return the errors.

```
└─ node src.js
 ERL ==> a =
 ! ERROR
Invalid Syntax: Incomplete input
a =
  ^
```

As shown, this error is now correctly returned. So now that the incomplete inputs are dealt with, the final decision is whether constants that have been defined should be allowed to have their value changed by another constant keyword.

```
> const a = 4
undefined
> a
4
> const a = 5
undefined
> a
5
```

When attempting to research this, it was difficult to find a standard, so I tested it with JavaScript and it was allowed, which probably means I should include it.

> TANISH
> yeah u should be able to redefine a constant that makes sense

I also asked Tanish, my stakeholder, and he thought it should be allowed based on what I said about JavaScript.

Therefore, I decided to allow it in my version.

```
└─ node src.js
 ERL ==> a = 4
 ERL ==> a
4
 ERL ==> const a = 5
 ERL ==> a
5
 ERL ==> a = 6
 ! ERROR
Identifier Error: 'a' is a constant and has already been defined
a = 6
^
 ERL ==> const a = 7
 ERL ==> a
7
```

It is very bad programming practice to redefine constants anyway, but it is a slight change that I can fully decide upon whether I want to keep or remove later, based on further feedback.

```
set(newValue){
    if (this.lastReferenced.constant == true){
        this.constant = true
        this.declared = false
    }
```

This now concludes this section on variables, and everything has been implemented.

EVALUATION

The success criteria for this module are as follows:

No.	Criteria	Implemented?
2.1	Allow for identifiers to be used to store values, defined with the '=' syntax, which can later be read.	**Yes**
2.2	Ensure the new variable system integrates with the existing arithmetic, for both setting values and inside expressions.	**Yes**
2.3	Let constants be created with the "const" keyword, which are variables that will return errors if the user attempts to redefine them	**Yes**
2.4	Ensure full error handling, so that variables cannot be accessed before assignment, and that all invalid cases are accounted for.	**Yes**

All of the features have been implemented, which is very good, however I have yet to add the criteria that I did not complete from the arithmetic stage.

CODE QUALITY

In terms of the code itself, I am not currently happy that the global symbol table is being stored in a variable, rather than as a property to a class as it does not feel very complete, however this can be changed later in the program.

When subroutines are added, this whole system will probably have to change because different scopes will be required. However, I have borne this in mind, and think I have designed Symbol Tables to work very dynamically, so it should be easy to adapt for the future.

There is still not much stakeholder feedback at this point, although I did ask Tanish to check about how constants should be implemented, so I am involving him in the process.

LOGIC

DESIGN

Comparison operators			
==	Equal to	<=	Less than or equal to
!=	Not equal to	>	Greater than
<	Less than	>=	Greater than or equal to

Boolean operators	
AND	Logical AND
OR	Logical OR
NOT	Logical NOT

This 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.

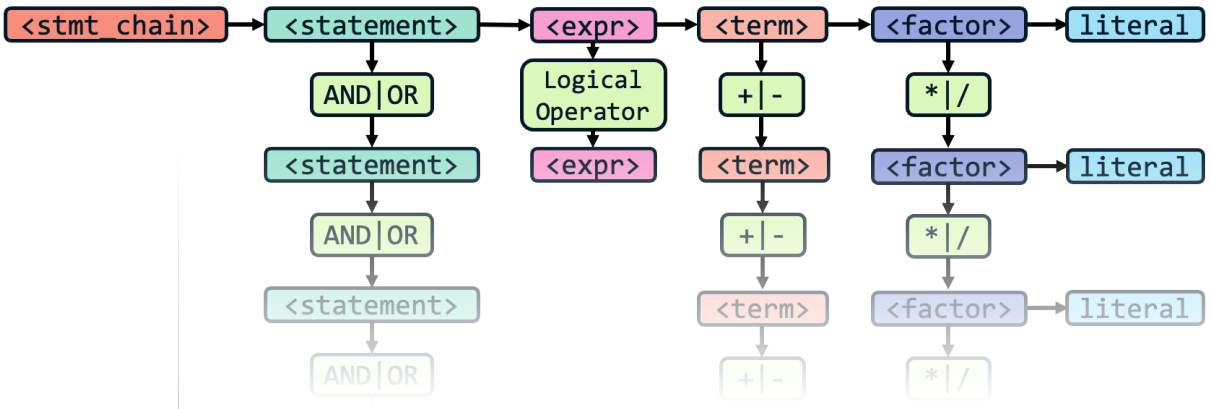
AND + OR

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

I 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.

STATEMENTS

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

LEXER

```
class LogicalOperator extends BinaryOperator{
    constructor(position){
        super(position)
    }
}
// ==
class Equality extends LogicalOperator{
    constructor(position){
        super(position)
    }
}
```

```
} else if (this.character == '=') {
    tokens.push(this.make_equals())
    continue
```

```
make_equals(){
    this.continue()
    if (this.character == '='){
        this.continue()
        return new Equality(this.position-2)
    }
    return new Equals(this.position-1)
}
```

```
Integer { position: 0, value: 1 },
Equals { position: 2, left: null, right: null },
Integer { position: 4, value: 2 }
```

```
Integer { position: 0, value: 3 },
Equality { position: 2, left: null, right: null },
Integer { position: 5, value: 4 }
```

I 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.

The test case correctly distinguished the two operators as shown.

STATEMENT METHOD

```
// Left over case is just an expression or a statement
let result = this.statement(this)
return this.check_result(result) ? result : new SyntaxError(this.token, "Expected operator")
```

```
statement(self){
    let left = self.expression()
    if (left instanceof Error || self.token == null){
        return left
    }
    let result = self.token
    if (result instanceof LogicalOperator){
        self.continue()
        right = self.expression()
        if (right instanceof Error){
            return right
        }
        result.left = left
        result.right = right
        return right
    }
    return this.check_result(result) ? result : new SyntaxError(this.token, "Expected operator")
}
```

self.expression(self)

return result

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.

The results of the tests show that Equalities now work as expected, alongside assignments.

```
636 console.log(new Parser(new Lexer("a = 2").make_tokens()).parse())
```

```
Equals {
  position: 2,
  left: Identifier { position: 0, name: 'a', constant: false },
  right: Integer { position: 4, value: 2 }
}
```

("a == 2")

```
Equality {
  position: 2,
  left: Identifier { position: 0, name: 'a', constant: false },
  right: Integer { position: 5, value: 2 }
}
```

("a + 7 == 2 + b")

```
Equality {
  position: 6,
  left: Add {
    position: 2,
    left: Identifier { position: 0, name: 'a', constant: false },
    right: Integer { position: 4, value: 7 }
  },
  right: Add {
    position: 11,
    left: Integer { position: 9, value: 2 },
    right: Identifier { position: 13, name: 'b', constant: false }
  }
}
```

I 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.

```
185   class Equality extends LogicalOperator{
186       constructor(position){
187           super(position)
188       }
189
190       evaluate(){
191           let left = this.left.evaluate()
192           let right = this.right.evaluate()
193           let result = this.check_for_errors(left, right)
194           if (result != null){
195               return result
196           }
197           return left == right
198       }
199   }
```

```
└─ node src.js
 ERL ==> 2 == 2
true
 ERL ==> 2 + 1 == 3 - 4
false
```

The 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

```
check_for_float(left, right){
    if (left instanceof Float || right instanceof Float){
        return true
    }
    return false
}
```

A 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.

```
evaluate(){
    let left = this.left.evaluate()
    let right = this.right.evaluate()
    let result = this.check_for_errors(left, right)
    if (result != null){
        return result
    }
    return this.check_for_float(left, right) ? new Float(this.position, left + right) : new Integer(this.position, left + right)
}
```

As 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.

```
 ERL ==> 2 + 2
Integer { position: 2, value: 4 }
 ERL ==> 2.0 + 2
Float { position: 4, value: 4 }
```

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.

```
238   class Float extends Token{
239       constructor(position, value){
240           super(position)
241           this.value = value
242       }
243
244       evaluate(){
245           return this.value
246       }
247
248       display(){
249           let result = String(this.evaluate())
250           if (result.includes('.')){
251               return result
252           }
253           return result + ".0"
254       }
255   }
```

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

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 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.

```
└─ node src.js
 ERL ==> 2+2
4
 ERL ==> 2.0 + 2
4.0
 ERL ==> 2.1 + 3
5.1
```

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.

To 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.

```
evaluate(){
    let left = this.left.evaluate()
    let right = this.right.evaluate()
    let result = this.check_for_errors(left, right)
    if (result != null){
        return result
    }
    return this.check_for_float(left, right) ? new Float(this.position, left.value + right.value) : new Integer(this.position, left.value + right.value)
}
```

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.

```
└─ node src.js
 ERL ==> 2 + 2
4
 ERL ==> a = 2 + 2
 ERL ==> a
4
```

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.

IMPLEMENTING FOR ALL OPERATIONS

```
61  class Token {
62      constructor(position){
63          this.position = position
64      }
65
66      check_for_float(){
67          for (let item of arguments){
68              if (item instanceof Float){
69                  return true
70              }
71              if (item instanceof Number){
72                  if (String(item).includes('.')){
73                      return true
74                  }
75              }
76              if (typeof item === 'number'){
77          return false
78      }
79  }
```

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.

When 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.

```
evaluate(){
    let left = this.left.evaluate()
    let right = this.right.evaluate()
    let result = this.check_for_errors(left, right)
    if (result != null){
        return result
    }
    result = left.value + right.value
    return this.check_for_float(left, right, result) ? new Float(this.position, result) : new Integer(this.position, result)
}
```

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.

```
evaluate(){
    let left = this.left.evaluate()
    let right = this.right.evaluate()
    let result = this.check_for_errors(left, right)
    if (result != null){
        return result
    }
    if (right.value == 0){
        return new MathError(this, "Cannot divide by 0")
    }
    result = left.value * right.value
    return this.check_for_float(left, right, result) ? new
}
```

The format is on the left, and the check for the value will occur before the check_for_float() stage.

```
└─ node src.js
 ERL ==> 2 * 2
4
 ERL ==> 5 - 3
2
 ERL ==> 6/0
 ! ERROR
Math Error: Cannot divide by 0
6/0
 ^
 ERL ==> 2.0 * 4
8.0
```

```
└─ node src.js
 ERL ==> a = 2.0
 ERL ==> a
2.0
 ERL ==> a = a * 3
 ERL ==> a
6.0
 ERL ==>
```

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

GENERAL IMPROVEMENTS

Now 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().

```
// Takes in an the classes as arguments and checks if current token is instance of the items
check_instance() {
    for (let item of arguments){
        if (this.token instanceof item){
            return true
        }
    }
    return false
}
```

```
while (self.token != null && self.check_instance(...tokens)){
```

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.

```
factor(self){
    if (self.check_instance(Integer, Float, Identifier)){
        let result = self.token
        self.continue()
        return result
    } else if (self.check_instance(LeftBracket)){
        let errorToken = self.token
        self.continue()
        let result = self.expression(self)
        if (self.check_instance(RightBracket)){
            self.continue()
            return result
        }
```

BOOLEANS

```
class Boolean extends Token{
    constructor(position, value){
        super(position)
        this.value = value
    }

    evaluate(){
        return this
    }

    display(){
        return this.value ? "True" : "False"
    }
}
```

```
switch (name) {
    case "const":
        return new TemplateKeyword(position, "const")
    case "True":
        return new Boolean(position, true)
    case "False":
        return new Boolean(position, false)
    default:
        return new Identifier(position, name)
}
```

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

```
return new Boolean(this.position, left.value == right.value)
```

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.

```
expression(self){
    if (self.token instanceof Boolean){
        let result = self.token
        self.continue()
        return result
    }
    return self.parse_binary_operator(self, self.term, [Add, Minus])
}
```

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.

```
statement(self){
    let left // Left side of statement
    if (this.token instanceof Boolean){
        left = this.token
        this.continue()
    } else {
        left = self.expression(self)
    }
    if (left instanceof Error || self.token == null){ // Check if just a normal expression and not a statement
        return left
    }
    let result = self.token
    if (!(result instanceof LogicalOperator)){ // Middle
        return this.check_result(result) ? result : new SyntaxError(this.token, "Expected operator")
    }
    self.continue()
    let right // Right side of statement
    if (this.token instanceof Boolean){
        right = this.token
        this.continue()
    } else {
        right = self.expression(self)
    }
    if (right instanceof Error){
        return right
    }
    result.left = left
    result.right = right
    return result
}
```

Instead, 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.

```
└─ node src.js
 ERL ==> True == True
True
 ERL ==> 2 + 2 == 4
True
```

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.

AND + OR

```
class Or extends BinaryOperator{
    constructor(position){
        super(position)
    }

    evaluate(){
        let left = this.left.evaluate()
        let right = this.right.evaluate()
        let result = this.check_for_errors(left, right)
        if (result != null){
            return result
        }
        return new Boolean(this.position, left || right)
    }
}
```

```
class And extends BinaryOperator{
    constructor(position){
        super(position)
    }

    evaluate(){
        let left = this.left.evaluate()
        let right = this.right.evaluate()
        let result = this.check_for_errors(left, right)
        if (result != null){
            return result
        }
        return new Boolean(this.position, left && right)
    }
}
```

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.

```
statement_chain(self){
    return self.parse_binary_operator(self, self.statement, [And, Or])
}
```

Then, I added the planned parser methods.

```
let result = this.statement_chain(this)
return this.check_result(result) ? result : new SyntaxError(this.token, "Expected operator")
```

This includes another change to parse(), to ensure that statement_chain() is the default method.

```
└─ node src.js
 ERL ==> 1 == 1 AND 1 == 2
False
 ERL ==> 1 == 1 AND 2 == 2
True
 ERL ==> 1 == 1 OR 1 == 2
True
 ERL ==> True AND True
 ! ERROR
Invalid Syntax: Expected operator
True AND True
     ^
```

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.

```
return new Boolean(this.position, left.value && right.value)
```

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.

```
let result = self.token
if (!(result instanceof LogicalOperator)){ // Middle
    if (self.check_result(result)){
        return result
    }
    if (left instanceof Boolean){
        return left
    }
    return new SyntaxError(self.token, "Expected operator")
}
self.continue()
```

Fixing 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.

```
└─ node src.js
 ERL ==> 1 == 1
True
 ERL ==> True == True
True
 ERL ==> True == True AND False == False
True
 ERL ==> True AND True
True
 ERL ==> True AND False
False
 ERL ==> True OR False
True
 ERL ==> False OR False
False
 ERL ==>
```

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:

```
 ERL ==> *
/Users/pw/ocr-erl-interpreter/src.js:534
            if (this.token instanceof item){
                           ^
```

Entering a lone binary operator would crash the program.

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

```
// Check if the first token is a binary operator
if (this.token instanceof BinaryOperator){
    return new SyntaxError(this.positiom, "Expected literal")
}
```

```
statement(self){
    let left // Left side of statement
    if (self.token instanceof Boolean){
        left = self.token
        self.continue()
    } else if (self.token instanceof BinaryOperator){
        return new SyntaxError(self.token, "Expected literal")
    } else {
        left = self.expression(self)
    }
```

This 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.

```
└─ node src.js
 ERL ==> 1 ==
 ! ERROR
Invalid Syntax: Incomplete input
1 ==
   ^
 ERL ==> 1 == AND
 ! ERROR
Invalid Syntax: Expected literal
1 == AND
     ^
 ERL ==> 1 == 2 AND
 ! ERROR
Invalid Syntax: Incomplete Input
1 == 2 AND
       ^
```

PARSING BRACKETS

```
parse_brackets(self, start, end, nextFunction){
    let bracket = self.token
    if (bracket instanceof start){
        self.continue()
        let result = nextFunction(self)
        if (self.token instanceof end){
            self.continue()
            return result
        }
        return new SyntaxError(errorToken,  "'(' was never closed")
    }
    return null                                  bracket,
}
```

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.

```
let result = self.parse_brackets(self, LeftBracket, RightBracket, self.expression)
if (result != null){
    return result
}
```

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

Checking 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.

```
statement(self){
    let left // Left side of statement
    let bracketCheck = self.parse_brackets(self, LeftBracket, RightBracket, self.statement_chain)
    if (bracketCheck != null){
        left = bracketCheck
    } else if (self.token instanceof Boolean){
```

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.

One 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.

```
└─ node src.js
 ERL ==> 2 + (3 * 1
 ! ERROR
Invalid Syntax: '(' was never closed
2 + (3 * 1
    ^
 ERL ==> 2 ^ (1 * (3 + 4)
 ! ERROR
Invalid Syntax: '(' was never closed
2 ^ (1 * (3 + 4)
    ^
```

```
└─ node src.js
 ERL ==> True AND (True OR False)
True
 ERL ==> 1 == 2 OR (False AND 2 + 2 == 4)
False
```

```
└─ node src.js
 ERL ==> 2 * (1 + 3)
8
```

The 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.

```
 ERL ==> a = True
 ! ERROR
Invalid Syntax: Expected literal
a = True
    ^
```

Initially, 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.

```
let right = self.statement_chain(self)
```

```
└─ node src.js
 ERL ==> a = True AND True
 ERL ==> a
True
 ERL ==> b = a OR False
 ! ERROR
Invalid Syntax: Expected operator
b = a OR False
      ^
```

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()

```
half_statement(self){
    let bracketCheck = self.parse_brackets(self, LeftBracket, RightBracket, self.statement_chain)
    if (bracketCheck != null){
        return bracketCheck
    }
    if (self.token instanceof Identifier){
        if (self.token.evaluate() instanceof Boolean){
            let result = self.token
            self.continue()
            return result
        }
    }
    if (self.token instanceof Boolean){
        let result = self.token
        self.continue()
        return result
    }
    if (self.token instanceof BinaryOperator){
        return new SyntaxError(self.token, "Expected literal")
    }
    return self.expression(self)
}
```

```
if (self.token instanceof Identifier){
    let result = self.token.evaluate()
    if (result instanceof Boolean){
        self.continue()
        return result
    }
}
```

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,

```
statement(self){
    let left = self.half_statement(self)
    if (left instanceof Error || self.token == null){ // Check if just a normal expression and not a statement
        return left
    }
    let result = self.token
    if (!(result instanceof LogicalOperator)){ // Middle
        if (self.check_result(result)){
            return result
        }
        if (left instanceof Boolean){
            return left
        }
        return new SyntaxError(self.token, "Expected operator")
    }
    self.continue()
    if (self.token == null){
        return new SyntaxError(result, "Incomplete input")
    }
    let right = self.half_statement(self)
    if (right instanceof Error){
        return right
    }
    result.left = left
    result.right = right
    return result
}
```

```
└─ node src.js
 ERL ==> a = True
 ERL ==> a OR False
True
```

The 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

```
└─ node src.js
 ERL ==> 2 + (3*2)
8
 ERL ==> (3 * 2)
 ! ERROR
Invalid Syntax: Expected operator
(3 * 2)
     ^

 ERL ==> True AND (Nonsense)
 ! ERROR
Invalid Syntax: Expected operator
True AND (Nonsense)
              ^

 ERL ==> True AND (2 * * 3)
 ! ERROR
Invalid Syntax: '(' was never closed
True AND (2 * * 3)
         ^
```

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

Firstly, 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:

```
expression(self){
    let result = self.parse_binary_operator(self, self.term, [Add, Minus])
    if (self.check_instance(Integer, Float, Identifier)){
        return new SyntaxError(self.token, "Expected operator")
    }
    return result
}
```

The 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.

```
let result = nextFunction(self)
if (result instanceof Error){
    return result
}
```

```
└─ node src.js
 ERL ==> 2 + ( 3 * 4 4)
 ! ERROR
Invalid Syntax: Expected operator
2 + ( 3 * 4 4)
        ^

 ERL ==> 2 + (3 * * 4)
 ! ERROR
Invalid Syntax: Expected literal
2 + (3 * * 4)
        ^
```

```
 ERL ==> True AND (False OR)
 ! ERROR
Invalid Syntax: Expected literal
True AND (False OR)
              ^

 ERL ==> True AND (OR)
 ! ERROR
Invalid Syntax: Expected literal
True AND (OR)
          ^
```

```
 ERL ==> True AND ==
 ! ERROR
Invalid Syntax: Expected literal
True AND ==
         ^

 ERL ==> True AND (()==False)
 ! ERROR
Invalid Syntax: Expected literal
True AND (()==False)
          ^
```

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.

```
class LogicalOperator extends BinaryOperator{
    constructor(position, tag){
        super(position)
        this.tag = tag
    }

    evaluate(){
        let left = this.left.evaluate()
        let right = this.right.evaluate()
        let result = this.check_for_errors(left, right)
        if (result != null){
            return result
        }
        switch (this.tag){
            case "==":
                return new Boolean(this.position, left.value == right.value)
            case ">":
                return new Boolean(this.position, left.value > right.value)
            case ">=":
                return new Boolean(this.position, left.value >= right.value)
            case "<":
                return new Boolean(this.position, left.value < right.value)
            case "<=":
                return new Boolean(this.position, left.value <= right.value)
            case "!=":
                return new Boolean(this.position, left.value != right.value)
        }
```

This 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.

To 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.

```
make_logical_operator(){
    let initialCharacter = this.character
    this.continue()
    if (this.character == '='){
        this.continue()
        return new LogicalOperator(this.position-2, initialCharacter+character)
    }
    switch (initialCharacter){
        case '=':
            return new Equals(this.position-1)
        case '!':
            return new UnexpectedCharacterError(this.position-1, initialCharacter)
        default:
            return new LogicalOperator(this.position-1, initialCharacter)
    }
}
```

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 two-character tag, being the initial character followed by the "=", which will be able to cover all of the two-character comparisons.

This 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 >.

```
└─ node src.js
 ERL ==> 2 <= 2
True
 ERL ==> 1 + 1 > 3
False
 ERL ==> 2 != 3
True
```

```
 ERL ==> a <== 4
 ! ERROR
Invalid Syntax: Expected literal
a <== 4
    ^
```

```
 ERL ==> 4 ! 5
 ! ERROR
Unexpected Character: '!'
4 ! 5
  ^
 ERL ==> a = 5
 ERL ==> a
5
 ERL ==> a >= 2 + 3
True
```

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

I 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

```
evaluate(){
    return this
}
```

I 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

The success criteria for this module are as follows:

No.	Criteria	Implemented?
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

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.

IF STATEMENTS

DESIGN: MANAGING MULTIPLE LINES

This section will be split into two sections, as before we can implement if statements the program must be able to handle multiple lines of inputs. This is because previously, every possible command would only ever span a single line, but to allow for if statements, the Lexer and Parser must be slightly redesigned.

This starts with adding a new class which will be the permanent replacement of the current Run/Shell class, the Interpreter class.

INTERPRETER

As explained, this is the new class to link all of the different aspects of the program together, providing a proper bridge between the different aspects. This will have to be expanded when the interface is added at the end, but it should be permanent all the way for the rest of the development modules.

PLAINTEXT HANDLING

The Interpreter will have a property called plaintext to store the current plaintext. This will be able to be set through a few different methods.

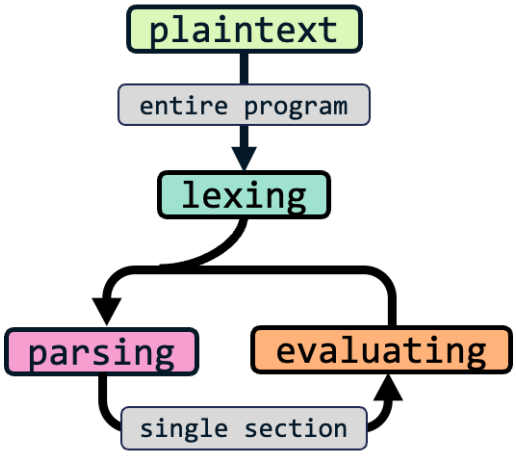
➔ get_plaintext_from_file(filename): will set the plaintext to the contents of a file.
➔ set_plaintext_manually(string): will set the plaintext to the string argument that is passed.

Both will split the string they receive on "\n", to create an array, where each item is a separate line of plaintext in the file. They will also set the have_tokens property to false.

The purpose of have_tokens is because of how the Interpreter will handle the whole program. It will first translate the entire set of plaintext into tokens, and then evaluate it one line/section at a time.

This may not be a pure interpreter, however, as a possible feature to include later will be syntax-highlighting, which will require the plaintext to be Lexed to know how to colour each section.

Therefore, have_tokens and storing the result of the Lexer into a property called tokens reduces the number of times Lexing needs to occur, as if the tokens have already been generated for highlighting, when Run is pressed then parsing and evaluation can immediately begin, without having to inefficiently generate the tokens again.



I may not end up using this implementation in the long run, but I am thinking ahead as this is a possible way to reduce the amount of computation I need to perform later on.

➔ make_tokens()

This will be a method for the Interpreter and will call upon the make_tokens() method of a new Lexer instance with the contents of the plaintext property being passed. If an error is returned, then it will be returned from make_tokens() as well and will otherwise set the tokens property to the 2D array of tokens that is returned, and set have_tokens to true.

LEXER CHANGES

To manage two positions in the Lexer class, new additions of row and line properties will be added, because now the lexer will have to go through each different line.

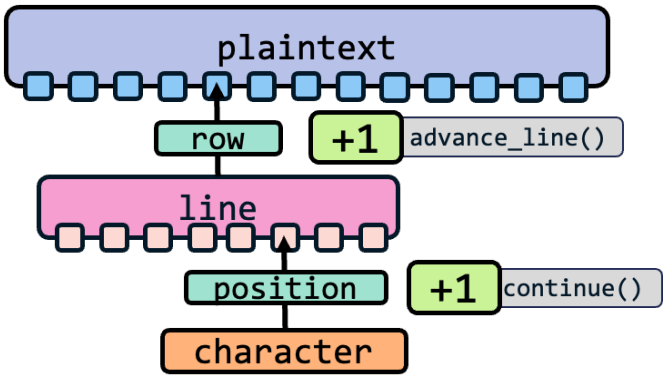
These will be equivalents position and character but will be on the scale of the entire program. Therefore, row is going to be an Integer, representing position of the line. line will be the plaintext at the position of row, being taken from the array which is now plaintext.

Row and position will both start as -1, and plaintext will be assigned in the constructor. continue() will have to be upgraded to now use line instead of the old property.

➔ advance_line()

This is the equivalent of continue() for traversing the whole plaintext, incrementing row by one, setting the line to the corresponding plaintext string, then setting position to -1 and calling continue().

When the end of the plaintext has been reached, then line will be set to null: the same as continue().



➔ make_tokens()

The old make_tokens() will be renamed to make_tokens_line() and make_tokens() will be updated.

It will repeatedly call make_tokens_line() until the current line is null, pushing each result to an array and then calling advance_line(). When it is null, this means there are no more lines, so it will return the array. At any point, if the result from make_tokens_line() is an Error, then this will be returned.

Therefore, for an array of different plaintext lines from the Interpreter, make_tokens() will return a 2D array, where each sub-array contains the stream of tokens for the corresponding line of plaintext.

PARSER CHANGES

The parser will have very similar system to the Lexer, with the new row and line properties cycling through a tokens property. Tokens will be an array that is passed from the Lexer result. The advance_line() method will also be present, identical to in the Lexer.

➔ parse_next()

This is the method that will be called by the Interpreter class. It will call advance_line(), and then check if the current line is null. If it is, it returns null, and will otherwise call the parse() method and return that. This is the equivalent of parsing the next section of code and will be called repeatedly until it returns null to represent the end of the program having been reached.

RUN

The Interpreter will have a run() method, which will first check if have_tokens is true. If it isn't, then it will call make_tokens(), **outputting** any potential errors. The key aspect of this run() method is that it is the final level of interaction, so will be outputting rather than returning values.

Then, using a variable called ast, it will repeatedly call parse_next(), setting ast to the result until it is null. This includes checking for Errors and outputting them if they occur.

Then it will call ast.evaluate() and check the result and print it. If there are any errors, then the loop is stopped by returning out to stop the program continuing to execute.

ERRORS

Errors must also be upgraded to have a line property alongside their position property. This will be defined when they are constructed and passed in as an argument for the lexer errors, as a new line property.

Therefore, every Token will now also have to have a line property, which will have to be given to them in the lexer, so that if a syntax error occurs, detail on their line are still valid. Therefore, the Lexer will have to be upgraded to also pass the line to every single Token.

currentText will also have to be set to match the current plaintext so that Error messages can properly display() the position of the error as expected.

After all this, the program should now be ready for structures that cross multiple lines, as advance_line() will begin to be used within the different methods, allowing for them to span different lines. I will plan all of the if statement implementation after this foundation is working.

DEVELOPMENT: MANAGING MULTIPLE LINES

ADDING LEXING TO THE INTERPRETER

Because spanning multiple lines is not as suited to a shell format, I am going to add implementation for files to be opened and ran, because it is the easiest way for me to run test cases.

OPENING FILES

```
get_plaintext(){
    try {
        return fs.readFileSync(this.fileName, 'utf8').split("\n")
    } catch (err) {
        console.error(err)
    }
}
```

There is no simple way to read the contents of files in JavaScript, so I did some research and ended up using the method at: https://nodejs.dev/en/ledisarn/reading-files-with-nodejs/

Some of the other methods required asynchronous methods, which I thought was a bad idea as I did not understand what it exactly involved. This current method does require a module to be installed by node, however as it is only semi-temporary until the proper interface is introduced later.

code.erl

```
1  a = 5
2  a + 2
3  a >= 3
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/src.js"
[ 'a = 5', 'a + 2', 'a >= 3 ' ]

[Done] exited with code=0 in 0.166 seconds
```

As shown, this solution can read a file and save it into an array of the different lines as different items.

LEXER CHANGES

These changes are very similar to the plan.

```
make_tokens(){
    let file = []
    while (this.line != null){
        result = this.make_tokens_line()
        if (result instanceof Error){
            return result
        }
        file.push(result)
        this.advance_line()
    }
    return result
}

make_tokens_line(){
    let tokens = []
    while (this.character != null){
```

The main difference being that program is used as the property name for the entire plaintext.

```
411  class Lexer {
412      constructor(program){
413          this.program = program
414          this.row = -1
415          this.position = -1
416          this.line = null
417          this.character = null
418          this.advance_line()
419      }
420
421      advance_line(){
422          this.row += 1
423          this.line = this.row == this.program.length ? null : this.program[this.line]
424          this.position = -1
425          this.continue()
426      }
427
428      continue(){
429          this.position += 1
430          this.character = this.position == this.line.length ? null : this.line[this.position]
```

this.program[this.row]

File has also been chosen as the array to store the token results from make_tokens_line(), and it is clear how it ensures no errors are built into this array by constant checking.

As shown, the test case for this worked after a small issue with naming, and all the correct token arrays were produced.

The issue was me getting confused between row and line, so this naming could be changed.

```
863  console.log(new Lexer(["2 + 2 = 4","a = 5","const b = a + 7"]).make_tokens())
```

```
[
  Integer { position: 0, value: 2 },
  Add { position: 2, left: null, right: null },
  Integer { position: 4, value: 2 },
  Equals { position: 6, left: null, right: null },
  Integer { position: 8, value: 4 }
],
[
  Identifier { position: 0, name: 'a', constant: false },
  Equals { position: 2, left: null, right: null },
  Integer { position: 4, value: 5 }
],
```

```
[
  TemplateKeyword { position: 0, tag: 'const' },
  Identifier { position: 6, name: 'b', constant: false },
  Equals { position: 8, left: null, right: null },
  Identifier { position: 10, name: 'a', constant: false },
  Add { position: 12, left: null, right: null },
  Integer { position: 14, value: 7 }
]
]
```

HAVE_TOKENS

```
847  class Interpreter {
848      constructor(){
849          this.plaintext = null
850          this.tokens = []
851          this.have_tokens = false
852      }
853          get_plaintext_from_file(fileName){
854      get_plaintext(fileName){
855          try {
856              this.plaintext = fs.readFileSync(fileName, 'utf8'
857          } catch (err) {
858              console.error(err)
859          }
860          this.have_tokens = false
861      }
862
863      make_tokens(){
864          console.log("MAKING TOKENS")
865          let result = new Lexer(this.plaintext).make_tokens()
866          if (result instanceof Error){
867              return result
868          }
869          this.tokens = result
870          this.have_tokens = true
871      }
872
873      run(){
874          if (!this
875              this.
876          }
877          return th
878      }
879  }
```

```
Name, 'utf8').split("\n")
```

```
run(){
    if (!this.have_tokens){
        let result = this.make_tokens()
        if (result instanceof Error){
            console.log(result)
            return
        }
        this.tokens = result
    }
    return this.tokens
}
```

On the left is the entire new code for the Interpreter class, and I remaned the method to be the full version for extra clarity with the second setting method,

have_tokens() has been added as expected, and the run() method, for now, just returns the tokens, but it calls make_tokens() if have_tokens is false: simulating what the future program would do.

```
882  main = new Interpreter()
883  main.get_plaintext("code.erl")
884  console.log(main.run())
885  console.log(main.run())
```

As a test, we call run() twice, and check if the temporary "MAKING TOKENS" is only printed once. It was, therefore the implementation works, increasing efficiency by not re-lexing the entire program on each call of run so long as no changes are made to the plaintext.

I also had to change my original run() method to include checking for lexical errors.

```
set_plaintext_manually(string){
    this.plaintext = string.split("\n")
    this.have_tokens = false
}
```

Finally, I implemented the second plaintext setter, which I will use for testing at points.

PARSER CHANGES

```
class Parser {
    constructor(tokens){
        this.tokens = tokens
        this.row = -1
        this.position = -1
        this.line = null
        this.character = null
    }

    advance_line(){
        this.row += 1
        this.line = this.row == this.tokens.length ? null : this.tokens[this.row]
        this.position = -1
    }

    continue(){
        this.position += 1
        this.token = this.position == this.line.length ? null : this.line[this.position]
    }
```

```
parse_next(){
    this.advance_line()
    return this.line == null ? null : this.parse()
}
```

The code is a direct copy from the Lexer class, with tokens being the entire set of tokens from the Lexer, which is iterated using advance_line().

parse_next() is also added, to repeatedly call parse(), as described in the plan.

parse_next() is what will be used in the Interpreter code during run() to have an Interpreter style of running a line at a time.

```
889         run(){
890             if (!this.have_tokens){
891                 let result = this.make_tokens()
892                 if (result instanceof Error){
893                     console.log(result.display())
894                     return
895                 }
896                 this.tokens = result
897             }
898             let parser = new Parser(this.tokens)
899             let ast = parser.parse_next()
900             while (ast != null){
901                 if (ast instanceof Error){
902                     console.log(result.display())
903                 }
904                 let result = ast.evaluate()
905                 console.log(result.display())
906                 if (result instanceof Error){
907                     return
908                 }
909             }
910         }
911     }
912
913 main = new Interpreter()
914 main.get_plaintext_from_file("code.erl")
915 console.log(main.run())
```

```
ast = parser.parse_next()
```

The first attempt of creating the run() method is as shown on the left, which had to be edited to repeatedly change ast to the new value of parse_next() at the end of each iteration.

However, there are still several programs with this code when testing and analysing it:

➔ If the result of evaluate() was null, such as in an assignment, it would cause the program to crash as it would attempt to call null's display() method.

➔ Nothing ever gets outputted: the program does not crash but never prints anything.

When debugging by printing out the values of variables, the issue was that continue() is never called after advance_line(), so the token property was always null so nothing was parsed.

A check that line is not null is also required, so that continue() is not called on a null line, which causes an error as shown below.

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
7
True
/Users/pw/ocr-erl-interpreter/testing.js:561
        this.token = this.position == this.line.length ? null : this.line[this.position]
                                                                 ^

TypeError: Cannot read properties of null (reading 'length')
```

The updated code is shown below:

```
let ast = parser.parse_next()
while (ast != null){
    if (ast instanceof Error){
        console.log(result.display())
    }
    if (ast == null){
        return
    }
    let evaluated = ast.evaluate()
    if (evaluated != null){
        console.log(evaluated.display())
    }
    if (evaluated instanceof Error){
        return
    }
    ast = parser.parse_next()
```

I changed the name for the result of evaluation to evaluated to make the code more readable, as well as adding the null check.

```
parse_next(){
    this.advance_line()
    if (this.line == null){
        return null
    }
    this.continue()
    return this.parse()
}
```

When running the test now, the expected result was returned, without the program crashing afterwards.

I ran a few more test cases which were all good too.

```
[Runnir
7
True
```

I slightly rushed through explaining this, but it the same idea as the old Run class, so is not drastically new in theory.

SHELL FUNCTIONALITY

```
shell(){
    this.set_plaintext_manually(prompt(" ERL ==> "))
    while (this.plaintext != "QUIT()"){
        this.run()
        this.set_plaintext_manually(prompt(" ERL ==> "))
    }
}
```

This was not on the plan, but adding a shell will be useful for debugging, as well as being an eventual feature to be added. This code works by repeatedly setting the plaintext to a user input, and then calling run() until the exit keyword was entered: "QUIT()"

```
873 main = new Interpreter()
874 main.shell()
```

The shell is called with the code on the left.

```
└─ node testing.js
 ERL ==> 2 + 2
4
 ERL ==> 7 * 3 - 2 ^ 4
5
 ERL ==> a = 4
 ERL ==> QUIT()
```

As shown on the left, this test case worked as anticipated.

I also added a single run_file() method, which combines the multiple steps into one. I also made it print the result of run(), which now has exits codes 0 and 1, typical of programming languages. In run(), now when it is returned due to an error, 1 is passed instead as the return value, and 0 is only returned after the while loop terminates.

```
872         run_file(fileName){
873             this.get_plaintext_from_file(fileName)
874             let code = this.run()
875             console.log(`\nExited with code ${code}`)
876         }
877     }
878
879     main = new Interpreter()
880     main.run_file("code.erl")
```

This therefore concludes the majority of the multiple-line handling features: now error handling must be reworked as in its current state the program crashes whenever an error occurs.

ERROR HANDLING FOR MULTIPLE LINES

```
set_plaintext_manually(string){
    this.plaintext = string.split("\n")
    currentText = this.plaintext
    this.have_tokens = false
}
```

To allow errors to access their text, every time the plaintext property is changed, the global variable currentText will be altered to reflect it. This may be changed later, as it is inefficient to save the same data twice.

Now the line property must be added to Tokens. For consistency, this will always be the second parameter in any constructors, so position then line and then the other parameters.

```
class Token {
    constructor(position, line){
        this.position = position
        this.line = line
    }
```

```
class Identifier extends Token{
    constructor(position, line, name, constant=false){
        super(position, line)
        this.name = name
        this.constant = constant
    }
```

```
class Error {
    constructor(position, line){
        this.position = position
        this.line = line
        this.text = currentText[line]
    }
```

Errors work similarly and will have line as their second parameter. Therefore, their text property will be current text at the position of their line property. For Errors that accept tokens in the constructor, the token's position and line are passed to the super constructor.

```
class UnexpectedCharacterError extends Error {
    constructor(position, line, character) {
        super(position, line)
```

```
class SyntaxError extends Error {
    constructor(token, description='') {
        super(token.position, token.line)
```

From here, every time a new Token (or Character Error) is created in the program, the parameters must be updated to also take in the line. The error's display() method is also changed to output the line where the error occurred.

```
} else if (this.character == '+') {
    tokens.push(new Add(this.position, this.line))
} else if (this.character == '-') {
    tokens.push(new Minus(this.position, this.line))
} else if (this.character == '*') {
```

```
display(){
    return ` ! ERROR @line${this.line}\nUnexpected Character: '${this.character}'\n${this.location()}`
}
```

ISSUES WITH LINE

```
883 main = new Interpreter()
884 main.set_plaintext_manually("a = 2\na + 3\na * * 7")
885 main.run()
```

The test case on the left was what I used to initially test the changes. The expected result is 5, and then a Syntax Error on line 3.

Initially, the program would not run at all, due to lots of missing line parameters, which was solved by going to the line where the errors occurred in the program and updating them.

```
undefined
 ! ERROR @linea * * 7
Invalid Syntax: Expected literal
undefined
    ^
/Users/pw/ocr-erl-interpreter/testing.js:856
            let evaluated = ast.evaluate()
                                ^
```

From there, the Error message as shown on the left was outputted, and then the program crashed. It is visible from this that the line property of the program, which was meant to be 3, ended up being "a * * 3", which was the contents of the current line. This was confusing at first, but after looking at the source-code it was due to confusing with my naming conventions.

In the parser, the property row was the integer, representing the index of the current line, and line was the corresponding contents of that. However, for the tokens, line was now the integer index of the line, and I had passed the string contents into the expected integer property.

```
class Lexer {
    constructor(program){
        this.allPlaintext = program
        this.line = -1
        this.currentPlaintext = null
        this.position = -1
        this.character = null
        this.advance_line()
    }
```

```
class Parser {
    constructor(tokens){
        this.allTokens = tokens // All of the tokens in the program in an 2D array
        this.line = -1  // The current index of tokens in the array
        this.currentTokens = null // The corresponding line of tokens in an array
        this.position = -1 // The current position in the line
        this.token = null // The corresponding token for the position
        this.previous = null // The previous token
    }
```

The new attribute names are on the left, with line now representing the position value.

The advance_line() and continue() methods also must be updated to reflect these changes, for both the lexer and parser. The make_tokens() and parse_next() methods must also be updated with the new identifiers, which caused numerous errors until the new names were implemented.

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
undefined
 ! ERROR @line2
Invalid Syntax: Expected literal
a * * 7
    ^
/Users/pw/ocr-erl-interpreter/testing.js:856
            let evaluated = ast.evaluate()
                                ^

TypeError: ast.evaluate is not a function
```

The correct line and error message was now displayed, but the values of the previous calculations were not shown. The issues and changes for this were:

➔ The line property was not passed into the new tokens that were created from arithmetic calculations.
➔ If the ast was an error, then it was continued, when it should have been returned with error code 1.

FURTHER ERRORS

code.erl

```
1   A = 5
2   B = 6
3   A > B
4   A + B
5   A ^ B
6   A * B >= A ^ B
7
8   A
```

The new test case is on the left, and experienced lots of problems.

➔ make_logical_operator() had not been updated to include line.
➔ The Error class should be updated to print line + 1, as shown.

```
this.line = line + 1
```

➔ Any empty lines (line 7) would produce a null AST and cause the program to stop prematurely. The fix for this is adding a check in make_tokens(), and if an empty array is attempting to be added to the array of tokens, it is not pushed.

```
False
11
15625
False

Exited with code 0
```

```
if (result.length >= 1){
    file.push(result)
}
this.advance_line()
```

```
False
11
15625
False
5

Exited with code 0
```

The test for Character errors returns correctly.

```
[Running] node "/Users/pw/
 ! ERROR @line 2
Unexpected Character: '%'
P = %^&*
    ^

Exited with code 1
```

```
 ! ERROR @line 2
Invalid Syntax: Expected literal
A + 2 * * 7
      ^
10

Exited with code 0
```

Any lines of code after an error would still be executed, when the program should stop, which is fixed by returning out of run()if ast is an instance of an error.

```
if (ast instanceof Error){
    console.log(ast.display())
    return 1
}
```

```
 ! ERROR @line 2
Invalid Syntax: Expected literal
A + 2 * * 7
      ^

Exited with code 1
```

```
 ! ERROR @line 2
Identifier Error: 'B' was not declared
B + 3
^

Exited with code 1
```

```
 ! ERROR @line 2
Math Error: Cannot divide by 0
5/0
 ^

Exited with code 1
```

The tests that Identifier Errors and any evaluation errors (such as Math Errors) were successful.

ADDITIONAL FEATURES

Now that the debugging is complete, and the multiple lines can be correctly handled, I wanted to add a few extra features to make this section complete.

Firstly, when trying to streamline the command line experience, I researched how to access the command line arguments in JavaScript and came across the following article:
https://www.geeksforgeeks.org/how-to-read-command-line-arguments-in-node-js/

```
node testing
[
  '/Users/pw/.nvm/versions/node/v20.4.0/bin/node',
  '/Users/pw/ocr-erl-interpreter/testing'
]
```

```
888  console.log(process.argv)
```

When checking how this worked, the first two arguments were to run the file, but any arguments continuing from that werte the custom additions, therefore I could write some code to run the file with the name of the third argument, which would make it a lot easier to run a test the code directly from the terminal.

```
~/ocr-erl-interpreter on main !4 ?1
node testing this is a test
[
  '/Users/pw/.nvm/versions/node/v20.4.0/bin/node',
  '/Users/pw/ocr-erl-interpreter/testing',
  'this',
  'is',
  'a',
  'test'
]
```

```
~/ocr-erl-interp
node testing code.erl
1
2
3
4
5

Exited with code 0
 ERL ==> A
6
 ERL ==>
```

I also made it run the shell afterwards to allow for further error checking.

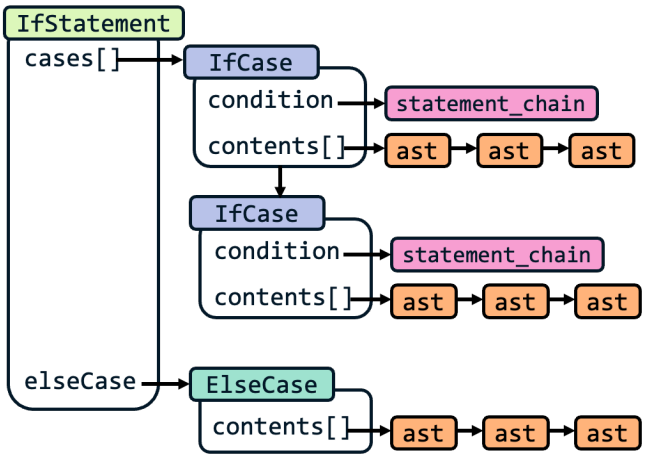
```
886  let commandLineArguments = process.argv
887  main = new Interpreter()
888  if (arguments.length > 2){
889      main.run_file(commandLineArguments[2])
890  }
891  main.shell()
```

DESIGN: IMPLEMENTING IF STATEMENTS

If statements are going to be implemented like a custom data type, so it is important to define their structure before considering the coded aspect.

There will be a class called IfStatement, and any instance of this class will represent an entire if chain. Using the OCR specification example, this encapsulates all the code from `if` to `endif`.

```
if answer == "Yes" then
    print("Correct")
elseif answer == "No" then
    print("Wrong")
else
    print("Error")
endif
```



This will then have two main properties (excluding line and position): an array called cases and elseCase.

The cases array will comprise of a series of objects from a new IfCase class. This will represent each branch of the if statement. In terms of the plaintext, each time `if` or `elseif` is used, it will be a separate IfCase in the cases array, including the code within them.

Each of these IfCases will then contain a condition property: this will be an abstract syntax tree which should evaluate to a Boolean, to represent whether the IfCase should run or not. There will also be a contents array, containing an array of abstract syntax trees for each line of code within the case.

With the OCR specification example, the cases array will contain two IfCases. The first will contain an ast equivalent to `answer == "Yes"` in the condition property, and the contents array will contain a single ast equivalent to `print("Correct")` . Then the second IfCase will have the `"No"` check as its condition, and a contents array with length 1 containing the ast for printing `"Wrong"`.

The elseCase property of IfStatement will either contain null (where an if statement did not include any else statement in the cases), or an instance of ElseCase. ElseCase is very similar to IfCase, but only contains the contents array, and has no condition as it would be automatically run if it is reached.

In this case, its contents would be the ast for `print("Error")`.

Overall, I feel like this structure is a generally clear way to store the different cases, as it is very logical and expandable. Overall, because the main object is the IfStatement class, this is what evaluate() will be called on, however the system is going to have to work slightly differently to the others. I will explain this in more detail after showing how these new classes will be produced and parsed.

PLAINTEXT TO TOKENS

"if" ➔ IfStatement: This will always be the first keyword in the entire if statement, however it will also contain its own IfCase, so therefore an IfCase will also need to be made to represent the case.

"elseif" ➔ IfCase: Each elseif will therefore be one of the cases.

"else" ➔ ElseCase

"then", "endif" ➔ TemplateKeyword: These are still very important parts of ERL, but themselves contain no coded functionality, so the template will be used to represent them, as they are still very important to the parser so cannot be excluded.

These will all be generated in make_identifier() and will be within the switch statement of the name to return each of the different cases.

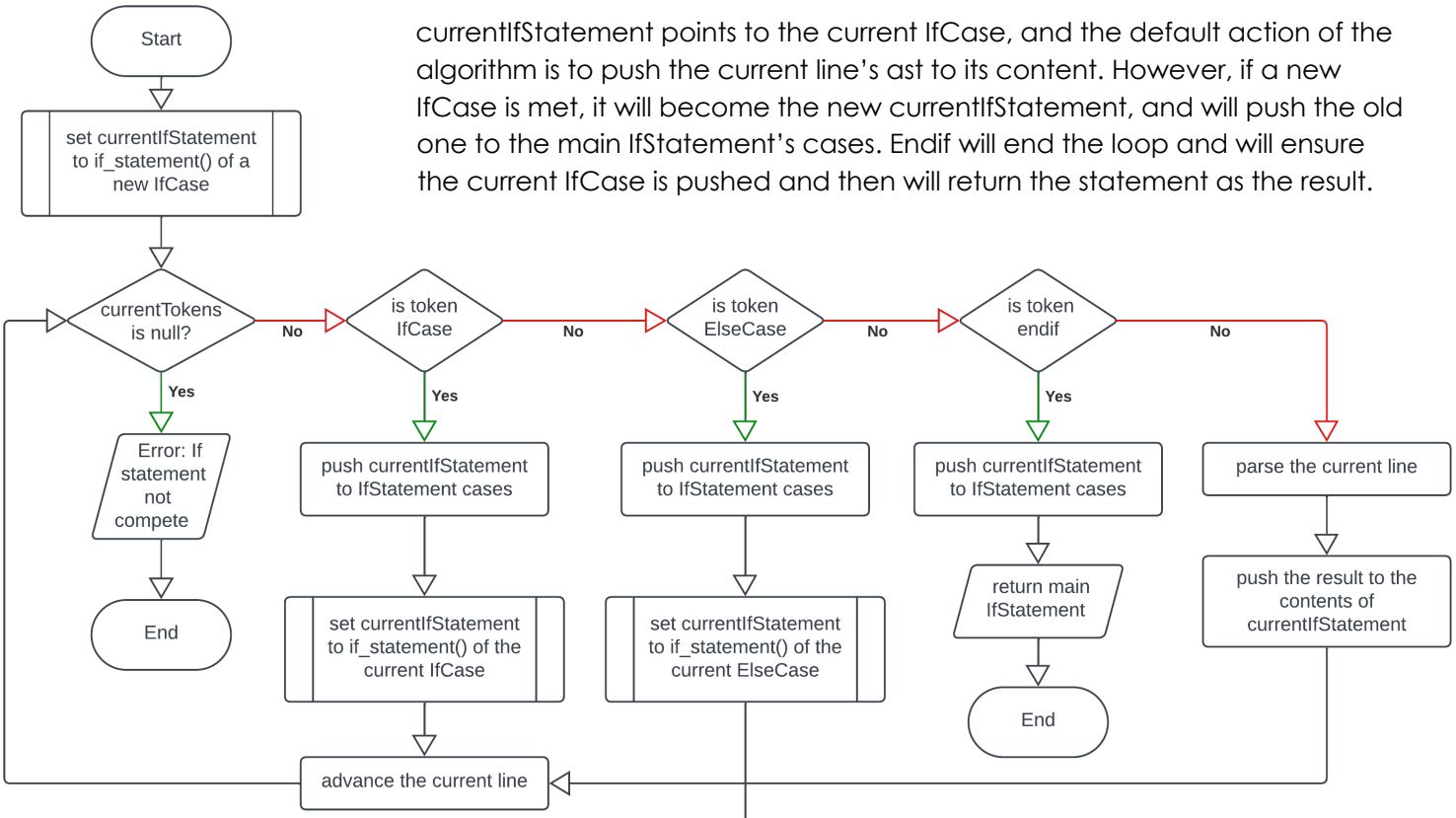
PARSER CHANGES

➔ build_if_chain()

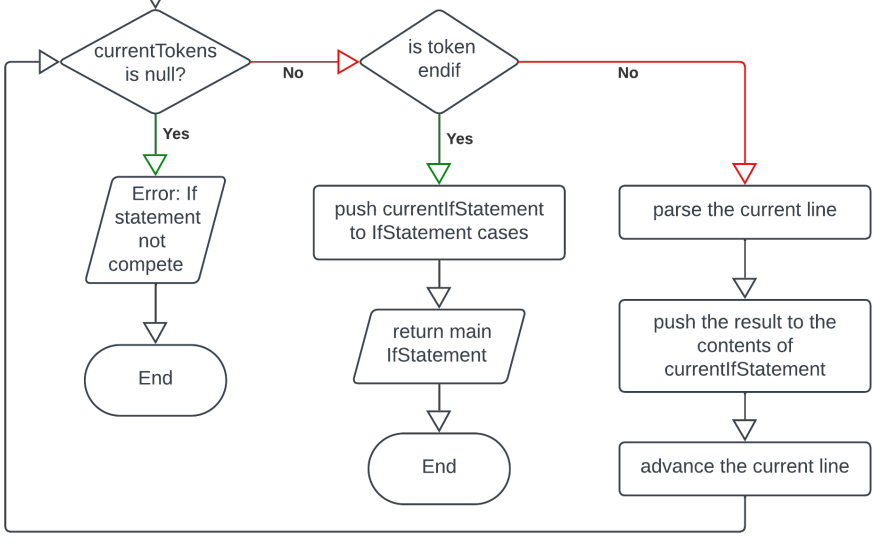
This is the main method for constructing IfStatements. It will be called in parse() if the current token is an IfStatement and will construct the entire custom data structure.

I have included a flowchart, which I feel best shows the order of parsing and how it works, so that it is the best introduction because it is quite complicated, however I have some additional notes surrounding it.

currentIfStatement points to the current IfCase, and the default action of the algorithm is to push the current line's ast to its content. However, if a new IfCase is met, it will become the new currentIfStatement, and will push the old one to the main IfStatement's cases. Endif will end the loop and will ensure the current IfCase is pushed and then will return the statement as the result.



If ElseCase is encountered, it will lead to a very similar loop, but without the options for IfCases or further ElseCases to occur, as else must be the penultimate keyword before endif.



➔ if_statement()

This is constantly referred to in the flowchart, and it is the method that deals with the condition lines.

This will have to be able to deal with all the potential errors from invalid inputs.

The BNF for the condition is as follows:

```
<if_statement> ::= IfCase <statement_chain> TemplateKeyword:"then"
```

The method will accept the IfCase token as a parameter: this could have been done by using self.token, however, we need to replace the initial IfStatement with a new IfCase for this method, so accepting it as a method reduces overall code repetition, whilst allowing self.token to be passed for most cases.

Overall, this seems complex on the surface, but it is just a big while loop with different cases.

ERROR HANDLING

This is a much harder aspect to plan, but error handling is going to be very important in this stage of handling. As the structure becomes more complex, there are lots of different invalid inputs which need to be dealt with, and therefore build_if_statement() and if_statement() are going to have to contain lots of checks to ensure that the input is as expected, and the methods that call these methods will also have to check that no errors have been returned so they do not get built into the structure.

EVALUATION

A lot of the details of new classes have already been laid out. IfStatement, IfCase and ElseCase will all extend token, and still will have position and line parameters in their constructor methods.

For extra code clarity, instead of naming it evaluate(), the method will be called get_case(). This is because it will not execute the if statement itself but will return the array of asts which then must be executed by the Interpreter instead.

➔ get_case()

This will iterate through all the IfCases in the cases array. It will call evaluate() on the condition property for each one. If an error occurs, then it will be returned. If it is true, then the contents[] array for that IfCase will be returned. If it isn't true, then the next IfCase will be iterated to. If every case has been iterated through, the program will check if the elseCase property is null. If it isn't, the contents of the ElseCase will be returned, otherwise get_case() will return null (when no cases were true).

INTERPRETER CHANGES

Now, run() is going to have to be adapted so that it can evaluate an array of ASTs, as that is what calling get_case() will now return.

This means that when parse_next() is called in run(), if the result is an instance of IfStatement, then get_case() will be called on it instead, and the result will be passed into a new method, instead of being displayed like the normal cases.

➔ evaluate_asts()

To reduce the size of run(), this new method will accept the asts from run() as an argument, and will evaluate() all of them, with the same previous checks that run() had for errors. This method can also accept a single ast that is not an array and will still run it. If an error occurs in this method, it will return 1, and then run() will return 1 if it is received.

This means that, as the feature set expands to include iteration and functions, run() becomes the method that chooses how to acquire the asts, and then evaluate_asts() will accept them and run them.

DEVELOPMENT: IMPLEMENTING IF STATEMENTS

BUILDING THE IF STATEMENTS

LEXER CHANGES

```
348   class IfStatement extends Token{
349       constructor(position, line){
350           super(position, line)
351           this.cases = []
352           this.elseCase = null
353       }
354   }
355
356   class IfCase extends Token{
357       constructor(position, line){
358           super(position, line)
359           this.statement = null
360           this.contents = []
361       }
362   }
363
364   class ElseCase extends Token{
365       constructor(position, line){
366           super(position, line)
367           this.contents = []
368       }
369   }
```

Implemented as planned, with changes on the left to make_identifier()

Condition is called statement to match statement_chain()

```
    case "if":
        return new IfStatement(position, this.line)
    case "elif":
        return new IfCase(position, this.line)
    case "else":
        return new ElseCase(position, this.line)
    case "const":
    case "then":
    case "endif":
        return new TemplateKeyword(position, this.line, name)
    default:
        return new Identifier(position, this.line, name)
```

```
918   let main = new Interpreter()
919   main.set_plaintext_manually("if elif else then endif")
920   main.make_tokens()
921   console.log(main.tokens)
```

```
[
  IfStatement { position: 0, line: 0, cases: [], elseCase: null },
  IfCase { position: 3, line: 0, statement: null, contents: [] },
  ElseCase { position: 8, line: 0, contents: [] },
  TemplateKeyword { position: 13, line: 0, tag: 'then' },
  TemplateKeyword { position: 18, line: 0, tag: 'endif' }
]
```

IF_STATEMENT() CONDITION METHOD

```
if_statement(self, ifToken){
    self.continue() // Continues past initial token
    let result = self.statement_chain() // Gets statement to check
    if (result instanceof Error){
        return result
    }
    if (self.token instanceof TemplateKeyword){
        if (self.token.tag == "then"){ // Checks if line finishes witt then
            ifToken.statement = result
            self.continue()
            if (self.token == null){
                return ifToken
            }
            return new SyntaxError(self.token, "Unexpected token after 'then'")
        }                                                    // Tokens after then
    }
    if (self.token == null){ // No then is included, points to if statement
        return new SyntaxError(ifToken, "If statement must end with 'then")
    } // No then and extra tokens, points to extra tokens
    return new SyntaxError(self.token, "Expected 'then")
}

build_if_chain(self){
    let mainStatement = self.token // Stores the entire chain
    let result = self.if_statement(self, new IfCase(self.token.position, self.token.line)) // Creating first if case
    if (result instanceof Error){
        return result
    }
    // Temporary for checking
    return result
}
```

let result = self.statement_chain(self)

```
953   let main = new Interpreter()
954   main.set_plaintext_manually("if 1 + 1 == 2 then")
955   main.make_tokens()
956   console.log(new Parser(main.tokens).parse_next())
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
IfCase {
  position: 0,
  line: 0,
  statement: LogicalOperator {
    position: 9,
    line: 0,
    left: Add { position: 5, line: 0, left: [Integer], right: [Integer] },
    right: Integer { position: 12, line: 0, value: 2 },
    tag: '=='
  },
  contents: []
}

[Done] exited with code=0 in 0.037 seconds
```

build_if_chain() serves only to test if_statement() here, which successfully passes the testcase, creating an IfCase statement that corresponds with the plaintext input.

ifToken is the parameter that it accepts, and the program calls continue(), then calls statement_chain() to save into ifToken's statement parameter and ensures that the line ends with a Template Keyword with the tag "then" as specified in ERL. Throughout, I have included multiple Error checks to ensure this layout is followed, with some examples of catches shown below:

```
[Running] node "/Users/pw/ocr-erl-interpreter/testin
SyntaxError {
  position: 7,
  line: 1,
  text: 'if 1 * * 2 == 3 then',
  token: Multiply { position: 7, line: 0, left: null
  description: 'Expected literal'
}
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
SyntaxError {
  position: 0,
  line: 1,
  text: 'if 1 + 1 == 2',
  token: IfCase { position: 0, line: 0, statement: null, contents: [] },
  description: "If statement must end with 'then"
}
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/testir
SyntaxError {
  position: 19,
  line: 1,
  text: 'if 1 + 1 == 2 then 5',
  token: Integer { position: 19, line: 0, value: 5 }
  description: "Unexpected token after 'then"
}
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
SyntaxError {
  position: 14,
  line: 1,
  text: 'if 1 + 1 == 2 const',
  token: TemplateKeyword { position: 14, line: 0, tag: 'const' },
  description: "Expected 'then"
}
```

ELIF AND ENDIF CASES

I am now going to code build_if_chain(), but only with the default, IfCase and endif branches as they are a lot simpler to implement compared to the elseCase branch.

```
build_if_chain(self){
    let mainStatement = self.token // Stores the entire chain
    let currentIfStatement = self.if_statement(self, new IfCase(self.token.position, self.token.line)) // Creating first if case
    if (currentIfStatement instanceof Error){
        return currentIfStatement
    }
    self.advance_line() // Advances line
    while(self.currentTokens != null){
        self.continue()
        if (self.token instanceof IfCase){ // Checkes for elif
            mainStatement.cases.push(currentIfStatement) // Pushes old if case
            let currentIfStatement = self.if_statement(self, self.token) // Creates new if case
            if (currentIfStatement instanceof Error){
                return currentIfStatement
            }
        } else if (self.token instanceof TemplateKeyword){ // Check for endif
            if (self.token.tag instanceof "endif"){
                mainStatement.cases.push(currentIfStatement) // Pushes old if case
                return mainStatement
            } // Otherwise do defauly
            let result = self.parse()
            if (result instanceof Error){
                return result
            }
            currentIfStatement.contents.push(result)
        } else {
            let result = self.parse() // Dafault
            if (result instanceof Error){
                return result
            }
            currentIfStatement.contents.push(result) //Add AST to the currents contents
        }
        self.advance_line()
    }
    return new SyntaxError(mainStatement, "Expected endif at end") // Not complete
    return mainStatement // Return whole if statement
}
```

```
[Running] node "/Users/pw/ocr-erl-interpreter/testing.js"
/Users/pw/ocr-erl-interpreter/testing.js:828
                mainStatement.cases.push(currentIfStatement) // Pushes old if case
                                                ^

ReferenceError: Cannot access 'currentIfStatement' before initialization
```

```
if (self.token.tag == "endif"){
```

```
/("if 1 + 1 == 2 then\n2 + 2\nelif 2 + 2 == 5 then\n3 + 4\nendif")
```

```
[Running] node "/Users/pw/ocr-erl-ir
IfStatement {
  position: 0,
  line: 0,
  cases: [
    IfCase {
      position: 0,
      line: 0,
      statement: [LogicalOperator],
      contents: [Array]
    },
    IfCase {
      position: 0,
      line: 2,
      statement: [LogicalOperator],
      contents: [Array]
    }
  ],
  elseCase: null
}
```

There were a few errors in my initial attempt, which have been fixed and produce the expected result: I have excluded the fully printed out version but the statement and contents are correct, with each IfCase in the contents array having correctly defined attributes.

Currently, the code has some repeated sections, such as the default case being included inside of the Template Keyword check as well in case a Template Keyword was present that was not endif, which is code repetition, but it can be eliminated for a more efficient solution later.

ELSE CASE

```
self.advance_line()
while(self.currentTokens != null){ // Loops through remaining tokens
    self.continue()
    if (self.token instanceof TemplateKeyword){ // Check for endif
        if (self.token.tag == "endif"){
            mainStatement.cases.push(currentIfStatement) // Pushes else case
            return mainStatement
        }
    }
    let result = self.parse() // Adds the asts
    if (result instanceof Error){
        return result
    }
    currentIfStatement.contents.push(result)
    self.advance_line()
}
```

This code is after the IfCase check and follows the earlier plan for this stage, however the test case failed.

```
{"if 1 + 1 == 2 then\n2 + 2\nelse\n3 + 4\nendif")
  cases: [
   IfCase {
    position: 0,
    line: 0,
    statement: [LogicalOperator],
    contents: [Array]
   },
   ElseCase { position: 0, line: 2, contents: [Array] }
  ],
  elseCase: null
}
```

This is because the ElseCase was appended to the cases array, rather than to the elseCase property.

This is an easy fix to make, and afterwards the correct arrangement occurred.

```
if (self.token.tag == "endif"){
    mainStatement.elseCase = currentIfStatement // Adds else case
    return mainStatement
}
```

This current implementation is good but is missing a lot of error handling for invalid inputs, however I have decided to implement execution before focusing on this.

EXECUTION

```
get_case(){
    for (let ifCase of this.cases){
        let result = ifCase.statement.evaluate()
        if (result instanceof Error){
            return result
        }
        if (result.value){
            return result.contents
        }
    }
    if (elseCase != null){
        return this.elseCase.contents
    }
    return null
}
```

```
execute_asts(asts){
    if (!asts.isArray()){
        asts = [asts]
    }
    for (let ast of asts){
        if (ast instanceof Error){
            console.log(ast.display())
            return 1
        }
        if (ast == null){
            continue
        }
        let evaluated = ast.evaluate()
        if (evaluated != null){
            console.log(evaluated.display(
        }
        if (evaluated instanceof Error){
            return 1
        }
    }
    return 0
}
```

```
run(){
    if (!this.have_tokens){
        let result = this.make_tokens()
        if (result instanceof Error){
            console.log(result.display())
            return 1
        }
    }
    let parser = new Parser(this.tokens)
    let ast = parser.parse_next()
    while (ast != null){
        if (ast instanceof IfCase){
            ast = ast.get_case()
        }
        result = this.execute_asts(ast)
        if (result == 1){
            return 1
        }
        ast = parser.parse_next()
    }
    return 0
}
```

code.erl

```
1  if 1 + 1 == 3 then
2  1
3  elif 1 + 1 == 2 then
4  2
5  else
6  3
7  endif
```

This code follows the very loose plan.

```
└─ node testing code.erl
/Users/pw/ocr-erl-interpreter/testing.js:979
                console.log(evaluated.display())
                                      ^

TypeError: evaluated.display is not a function
```

The initial error was due to not knowing how to properly check for array, which was easily researched to be fixed.

```
if (!Array.isArray(asts)){
    asts = [asts]
}
```

The second error was because it was checking for IfCase, instead of IfStatement.

```
while (ast != null){
    if (ast instanceof IfStatement){
        ast = ast.get_case()
    }
    let result = this.execute_asts(ast)
```

Now the program no longer crashed but did not output any result. After further debugging of asts, it was because get_case() returned result.contents, when it should be ifCase.contents, so after this fix, it worked.

```
if (result.value){
    return ifCase.contents
}
```

```
└─ node testing code.erl
2

Exited with code 0
 ERL ==>
```

QUALITY OF LIFE IMPROVEMENT

Just because my test if statement looked very unclear, I thought it was important to add tabbing and comments quickly. This was changed in make_tokens_line(). Tabs were added by extending the blank space check to include checking for tabs.

```
if (this.character == ' ' || this.character == "\t"){
```

Comments were included by extending the divide check, and if the next comment was another slash, then the current contents of tokens would be returned up to this point, as no more code can be written after a comment in ERL.

```
} else if (this.character == '/') {
    this.continue()
    if (this.character = '/'){ // Comment
        return tokens
    }
    tokens.push(new Divide(this.position-1, this.line))
    continue
} else if (this.character == '^') {
```

This now meant that ERL code could become a lot more readable with indentation: however, it is not a required feature and the program still works without it, as many textbooks do not include it and it is important to have consistency with these resources.

code.erl

```
1  if 1 + 1 == 3 then
2      1 // This is a test
3  elif 1 + 1 == 2 then
4      2
5  else // So is this
6      3
7  endif
```

```
[Running] node src.
2

Exited with code 0
```

ERROR HANDLING

This section is not as planned but involves me attempting to list every possible invalid input and creating a corresponding error check. I will most likely miss a few, but hopefully when I do more formal testing later, I should be able to catch them and they should be relatively straightforward additions.

Examples will be highlighted in red beneath the subtitle, as they would currently crash the program or produce a false error, and the implemented fix will be unhighlighted.

➔ A lone "if"

code.erl

```
1  if
```

```
if_statement(self, ifToken){
    self.continue() // Continues past initial token
    if (self.token == null){
        return new SyntaxError(ifToken, "Expected expression after 'if")
    }
```

Ensures that the next token is not null after the first if token, returning a custom error message

➔ Using an undeclared Identifier in the condition

```
1  if value then
2      5
3  endif
```

```
! ERROR @line 1
Invalid Syntax: Expected operator
if value then
```

This currently does not crash the program, but returns the incorrect error message, and this is due to how half_statement() operates with Identifier.

In the logic chapter, I commented that I did not like the evaluate() check to ensure that Identifiers are Booleans to allow a half_statement to be defined. This now causes problems further down the line, and the solution is to reduce the strictness of half_statement().

This involves removing the check for Identifiers, and that if no Logical Operator is present after the left-hand side, then it is returned. This does therefore mean, that statement() may potentially not always return a Boolean value. This does mean that extra checking is required elsewhere in the program to ensure the datatypes are as expected. This will be added more in depth later on, in the strings section for watertight type handling, however for now there may be some edge cases which break the program.

```
half_statement(self){
    let bracketCheck = self.parse_brackets(self, LeftBracket, RightBracket, self.statement_chain)
    if (bracketCheck != null){
        return bracketCheck
    }
    if (self.token instanceof Boolean){
        let result = self.token
        self.continue()
        return result
    }
    if (self.token instanceof BinaryOperator){
        return new SyntaxError(self.token, "Expected literal")
    }
    return self.expression(self)
}
```

The new methods:

```
statement(self){
    let left = self.half_statement(self)
    if (left instanceof Error || self.token == null){ // Check if
        return left
    }
    let result = self.token
    if (!(result instanceof LogicalOperator)){ // Middle
        if (self.check_result(result)){
            return result
        }
        return left
    }
```

```
[Running] node src.js "/Users/pw/ocr-erl-in
 ! ERROR @line 1
Identifier Error: 'value' was not declared
if value then
   ^

Exited with code 1
```

The correct error message is now returned by the program; that the Identifier has not been defined.

➔ Non-Boolean condition

```
code.erl
1  if 3 then
2      2
3  endif
```

```
if (result.value === true){
    return ifCase.contents
}
```

Now that non-Booleans can be returned from statement_chain(), the strong comparison operator must be used in get_case() to ensure the value is a boolean, as JavaScript evaluates `true == 3` to `true`.

A small syntax error was corrected, as elseCase was used rather than this.elseCase, and then this fix worked.

➔ No endif at end of if statement

```
code.erl
1  if True then
2      2
3  else
4      4
```

```
return new SyntaxError(mainStatement, "Expected endif at end of if statement") // Not complete
```

The fix for this was already implemented, but it contained a small syntax error which has now been fixed, alongside an improved error message.

I do not love that the error points to the if, as the issue is that the if statement is not closed and the start seems very far away from this, however it seems to be the most logical option so I chose it for the error token.

```
[Running] node src.js "/Users/pw/ocr-erl-interpreter/c
 ! ERROR @line 1
Invalid Syntax: Expected endif at end of if statement
if True then
^

Exited with code 1
```

➔ Random elif or else statements.

```
code.erl
1  elif True then
```

The program technically can catch this error; however, the error message is not very representative of the issue.

```
if (this.check_instance(IfCase, ElseCase)){
    return new SyntaxError(this.token, "Needs to follow 'if' statement")
}
```

The fix is to add a check in parse(), and if the first token in a line is an instance of IfCase or ElseCase, then a custom Syntax Error is returned.

```
[Running] node src.js "/Users/pw/o
 ! ERROR @line 1
Invalid Syntax: Expected literal
elif True then
^

Exited with code 1

[Done] exited with code=0 in 0.047
```

```
// Check if there is a tagged assignment
if (this.token instanceof TemplateKeyword){
    switch(this.token.tag){
        case "const":
            let result = this.assignment(this, this.token.tag)
            return this.check_result(result) ? result : new SyntaxError(this.token, "Expected operator")
        case "endif":
            return new SyntaxError(this.token, "Needs to follow 'if' statement")
    }
}
```

I also did the same check for "endif" tagged Template Keywords in parse(),

The resulting error message on the right is now much clearer and more representative of the issue than "Expected literal"

```
[Running] node src.js "/Users/pw/ocr-erl-interp
 ! ERROR @line 1
Invalid Syntax: Needs to follow 'if' statement
elif True then
^

Exited with code 1
```

➔ Nested if statements

code.erl

```
if False then
    2
elif True then
    if False then
        3
    elif True then
        4
    endif
else
    5
endif
```

Although this is a completely valid input, the program currently crashes when attempting to run a nested if statement.

This requires a redesign of run() and evaluate_asts() in the Interpreter class, to allow evaluate_asts() to recursively call itself. For this, I decided to rename IfStatement's get_case() to evaluate() for consistency and reduction of code repetition. Even thought it was a deliberate decision earlier to uniquely identify the method, it seems that in the long term it should be better, as loops will also have a similar system and each type having a customised evaluate() call avoids using polymorphism which is a very strong and useful aspect of OOP. This change just requires a few naming changes.

```
while (ast != null){
    let result = this.evaluate_asts(ast)
    if (result == 1){
        return 1
    }
    ast = parser.parse_next()
}
return 0
```

```
evaluate_asts(asts){
    if (!Array.isArray(asts)){
        asts = [asts]
    }
    for (let ast of asts){
        if (ast instanceof IfStatement){
            return this.evaluate_asts(ast.evaluate())
        }
        if (ast instanceof Error){
            console.log(ast.display())
            return 1
        }
        if (ast == null){
            continue
        }
        let evaluated = ast.evaluate()
        if (evaluated != null){
            console.log(evaluated.display())
        }
        if (evaluated instanceof Error){
            return 1
        }
    }
    return 0
}
```

The update moves the checks from run() to evaluate_asts(), so that an IfStatement can then continually call further IfStatements by more of these checks. Therefore, each layer of if statement will run a new instance of evaluate_asts(). run() now serves as only the base of the program: creating tokens and calling the initial evaluate_asts(), but all evaluation has now been moved.

This now works as anticipated, and the program produces the correct results. Because of the dynamic implementation, the system will work for any amount of nested if statements as well and allows expansion for other structures in the future.

```
[Running] node src.
4

Exited with code 0
```

FINAL CHANGES

```
if … then
elseif … then
else
endif
```

```
if answer == "Yes" then
    print("Correct")
elseif answer == "No" then
    print("Wrong")
else
    print("Error")
endif
```

Stupidly, I have been using "elif" instead of "elseif" like the specification states, but this is a very easy change, just requiring a string in make_identifier() to be altered.

```
case "elseif":
    return new IfCase(position, this.line)
```

Finally, I did not like that IfCase's condition is called statement, so I decided to change the attribute name. This required changing some of the code in other places, but some testing showed that it still worked.

```
class IfCase extends Token{
    constructor(position, line){
        super(position, line)
        this.condition = null
        this.contents = []
    }
}
```

```
if (self.token instanceof TemplateKeyword){
    if (self.token.tag == "then"){ // Checks if line finishes witt then
        ifToken.condition = result
```

```
let result = ifCase.condition.evaluate()
```

This concludes the code for if statements.

EVALUATION

The result of the success criteria for this section are as follows:

No.	Criteria	Implemented?
4.1	Allow if statements to be created by the "if" keyword, followed by a condition and "then". Then, on a newline, all of the contents, followed by a final, closing "endif" keyword.	**Yes**
4.2	Allow additional cases to be added with the "elseif" keyword, followed by a condition and "then", with the corresponding code, followed by more branches and finally "endif"	**Yes**
4.3	Allow a default case to be made by the "else" keyword alone on a line, then followed by its contents and then "endif" with no other cases, which will be run if no other conditions are.	**Yes**
4.4	Ensure that the cases are always laid out in the correct order, of "if", then optional "elseif", then optional "else" then "endif", to ensure the syntax is maintained throughout	**Yes**
4.5	Ensure that descriptive error messages are returned when the syntax is not followed, with details about the valid conditions, and ensuring that conditions are always valid	**Yes**

I feel that this section has gone very well, and the new Interpreter class is a good new system to merge the different elements and feels a lot more proper than the old solution. However, it will further have to be iterated in design, because loops and other aspects will just further change how things must be evaluated.

There are also still some issues with types: having non-boolean conditions will not be run, but they will not return an error either, which is what I'd prefer. Again, I am delaying this for the types module, where I will add stricter handling to everything, including these conditions.

LOOPS

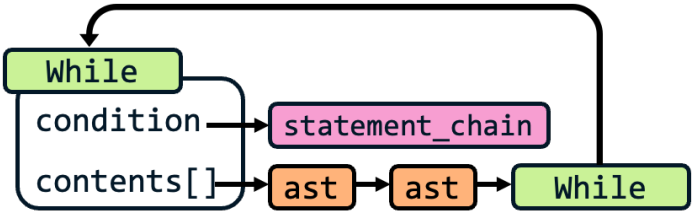
DESIGN: WHILE LOOPS

The plan for the implementation is consistent with IfStatement, and there will be a new While class that extends token. It will also have a condition property, and a contents array. This new While token will be produced by "while" in make_identifier().

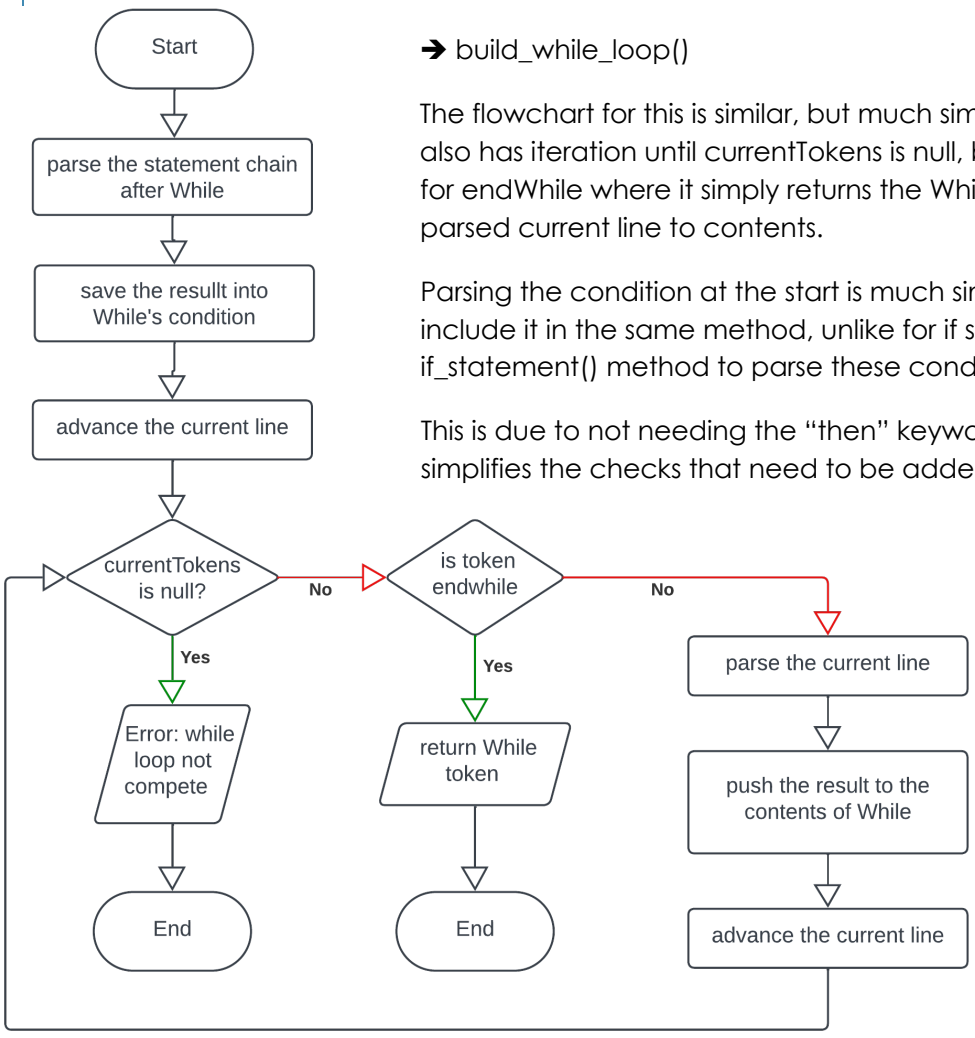
EVALUATION

Even though this is the final step, it most clearly explains how the while loop will operate. When evaluate() is called on the While token, it will evaluate() the condition to check if it is true. If it is true, it will return the value of contents, but the last item in contents will point to itself. This therefore means that when evaluate_asts() is called on the result of evaluate(), it will execute the contents, and then call the While loop again, to check for the condition again and allow another cycle.

This is a recursive way to deal with the issue, and I think is the simplest way to implement it, as separate checks for conditions and contents would complicate the Interpreter's code in run() and evaluate_asts()



PARSING



➔ build_while_loop()

The flowchart for this is similar, but much simpler than build_if_statement(). It also has iteration until currentTokens is null, but with less checks: only checking for endWhile where it simply returns the While token, otherwise it adds the parsed current line to contents.

Parsing the condition at the start is much simpler as well, so I have decided to include it in the same method, unlike for if statements which had the separate if_statement() method to parse these conditions.

This is due to not needing the "then" keyword largely, which significantly simplifies the checks that need to be added.

There will also have to be a lot more error checks included than presented in this flowchart, but they can easily be dealt with, like with if statements, by me trying lots of invalid inputs and implementing as many checks as I can to prevent them.

This plan is not very long, but a lot of the framework for implementing this feature is already in place, so it should be okay to start development.

DEVELOPMENT: WHILE LOOPS

BASICS

```
class While extends Token{
    constructor(position, line){
        super(position, line)
        this.condition = null
        this.contents = []
    }
}
```

A section in parse() also must be added to call the build method.

```
case "while":
    return new While(position, this.line)
```

```
parse(){
    // Check if there are no tokens
    if (this.token == null){
        return null
    }
    // Checks if it a while loop
    if (this.token instanceof While){
        return this.build_while_loop(this)
    }
    // Checks if an if statement is being built
```

BUILD_WHILE_LOOPS()

```
build_while_loop(self){
    let whileToken = self.token
    self.continue()
    if (self.token == null){
        return new SyntaxError(whileToken, "Expected condition after 'while'")
    }
    let condition = self.statement_chain(self)
    if (condition instanceof Error){
        return condition
    }
    whileToken.condition = condition
    self.continue()
    if (self.token != null){
        return new SyntaxError(self.token, "Expected no tokens after condition")
    }
    self.advance_line()
    while (self.currentTokens != null){
        self.continue()
        if (self.token instanceof TemplateKeyword){
            if (self.token.tag == "endwhile"){
                return whileToken
            }
        }
        let result = self.parse()
        if (result instanceof Error){
            return result
        }
        whileToken.contents.push(result)
        self.advance_line()
    }
}
```

code.erl

```
1  while True
2      1
3      2
4  endwhile
```

This was my initial implementation of the method. I included a few error checks – for no condition being present as well as additional tokens after the condition (such as then).

Ignoring the undefined line property (due to a syntax error), the test case was successful and the While loops was built correctly.

```
[Running] node src.js "/Users/pw/ocr-erl-interpreter/code.erl"
While {
  position: 0,
  line: undefined,
  condition: Boolean { position: 6, line: 0, value: true },
  contents: [
    Integer { position: 1, line: 1, value: 1 },
    Integer { position: 1, line: 2, value: 2 }
  ]
}
```

EVALUATION

```
if (self.token.tag == "endwhile"){
    whileToken.contents.push(whileToken)
    return whileToken
}
```

I initially decided to add the While loop to itself in the endWhile check of build_while_loop(), by pushing itself to contents before returning itself.

```
evaluate(){
    if (this.condition.evaluate().value === true){
        return [...this.contents, this]
    }
}
```

However, I then decided to move it to the evaluate() method, as I felt like altering the contents is a potentially bad idea if the system needs to be reworked in the future, as well as the printed ast being more representative of the actual contents if it does not reference itself.

```
if (ast instanceof IfStatement || ast instanceof While){
    return this.evaluate_asts(ast.evaluate())
}
```

Now by changing one line in evaluate_asts(), it meant that While loops were treated the same as IfStatements for the recursive calling.

```
[Ru
1
2
1
2
1
2
1
2
```

The test case correctly worked, and when trying a more complex test involving identifier – like a counter-controlled loop – the expected result was still returned, proving that the implementation was a success.

However, there was one issue with this implementation.

code.erl

```
i = 0
while i <= 5
    i
    i = i + 1
endwhile
```

```
[Running] node src.
0
1
2
3
4
5
```

THE CALL STACK PROBLEM

code.erl

```
i = 0
while i < 5004
    i = i + 1
endwhile
```

Because the solution was implemented recursively, whenever the while loop approached around 5000 iterations or greater, a stack overflow error would occur as the maximum amount of recursion had been met.

```
[Running] node src.js "/Users/pw/ocr-erl-inter
/Users/pw/ocr-erl-interpreter/src.js:73
    check_for_float(){
                    ^

RangeError: Maximum call stack size exceeded
```

```
5473
node:internal/console/constructor:308
        if (isStackOverflowError(e))
            ^

RangeError: Maximum call stack size exceeded
```

Initially, I thought that this may not be a huge problem, due to GCSE questions never requiring this level of iteration. This would then require some sort of iteration cap, which I considered but then decided against.

Additionally, having an enormous call stack uses a lot of memory, and as the target user to for this project uses low-end school devices, optimising performance is important. Therefore, I decided to use a different approach to evaluating the While loops, using iteration rather than recursion.

AN ITERATIVE APPROACH

```
evaluate_condition(){
    return this.condition.evaluate().value
}

evaluate(){
    return this.contents
}
```

The new approach now has two evaluate() methods: evaluate condition() for the condition and evaluate() for the contents.

I do not love having multiple evaluate() methods as it ruins the polymorphism, but because all 3 types of loops will be able to use this same system it is not a major issue.

```
if (ast instanceof While){
    return this.evaluate_loop(ast)
}
```

evaluate_asts() now has a separate check for While (and in future the others) and will call a new method: evaluate_loop(), passing the loop ast to the method.

```
evaluate_loop(loop){
    while (loop.evaluate_condition()){
        return this.evaluate_asts(loop.evaluate())
    }
}
```

This method will then repeatedly call evaluate_asts() on the loop's contents while evaluate_condition() is true, therefore creating an iterative approach.

The naming could be changed in the future, as evaluate() is not a perfect description, however for now it is acceptable.

code.erl
```
i = 0
while i < 10000
    i = i + 1
    i
endwhile
```

```
[Running] node testing
1

Exited with code 0
```

Initially, the test case failed, but this is because I'd used return in evaluate_loops() so the first iteration would only ever occur.

I also realise I'd still need to check for errors and be able to exit the loop if one occurred, so using the 0/1 system, if the result of evaluate_asts() was ever 1, I would then exit out of the loop prematurely.

```
evaluate_loop(loop){
    while (loop.evaluate_condition()){
        console.log(loop.evaluate_condition())
        if (this.evaluate_asts(loop.evaluate()) == 1){
            return 1
        }
    }
    return 0
}
```

From here, we now had no limit on the number of iterations.

code.erl
```
i = 0
while i < 1000000
    i = i + 1
endwhile
i
```

```
[Running] node testing.j
1000000

Exited with code 0

[Done] exited with code=
```

ERROR HANDLING

➔ **Error with condition**

code.erl
```
i = 0
while test
    i = i + 1
endwhile
```

```
[Running] node src.js
0

Exited with code 0
```

Currently, if there is an error in the condition, it is not reported by the interpreter and instead some completely unexpected results are produced.

```
evaluate_condition(){
    let condition = this.condition.evaluate()
    if (condition instanceof Error){
        return condition
    }
    return condition.value
}
```

This requires evaluate_condition() to be changed so that, if an error occurs, it will be returned. Therefore, evaluate_loop() must also be changed so that every time the condition is revaluated, there is a check for an error that can return the error if need be.

```
evaluate_loop(loop){
    let condition = loop.evaluate_condition()
    if (condition instanceof Error){
        return condition
    }
    while (condition){
        if (this.evaluate_asts(loop.evaluate()) == 1){
            return 1
        }
        condition = loop.evaluate_condition()
        if (condition instanceof Error){
            return condition
        }
    }
    return 0
}
```

```
if (ast instanceof Loop){
    let result = this.evaluate_loop(ast)
    if (result instanceof Error){
        console.log(result.display())
        return 1
    }
    continue
}
```

Then, evaluate_asts() must be able to catch these errors from evaluate_loop(), so it does not continue executing instructions after the error occurs, and can return out of the method.

```
[Running] node src.js "/Users/pw/ocr-erl-i
! ERROR @line 2
Identifier Error: 'test' was not declared
while test
      ^

Exited with code 1
```

I do not love this implementation, as sometimes integers are returned when errors occur and sometimes integers of 0/1 are returned, so there is not a lot of consistency which I think may result in issues, however for now this solution is acceptable and can be refined later.

➔ **Tokens after endwhile**

```
i = 0
while i < 10
    i = i + 1
endwhile 2 + 2 == 4
i
```

Even though it doesn't crash the program and simply ignores the tokens, I feel as if there should be an error message for clarity of code.

I also added the change for endif as it did not currently have a check and could also have the same issue

```
// Checks if it a while loop
if (this.token instanceof While){
    let result = this.build_while_loop(this)
    return this.check_result(result) ? result : new SyntaxError(this.token, "Expeceted nothing to follow endwhile")
}
// Checks if an if statement is being built
if (this.token instanceof IfStatement){
    let result = this.build_if_chain(this)
    return this.check_result(result) ? result : new SyntaxError(this.token, "Expeceted nothing to follow endif")
}
```

The solution is by changing parse() to ensure that the current token is null after build_while_loop() or build_if_chain() are used, as that would catch any unexpected tokens.

```
if (self.token.tag == "endwhile"){
    this.continue()
    return whileToken
}
```

The build methods also must be changed to call continue() before they return the token, but from there the error is correctly handled.

```
if (self.token.tag == "endif"){
    mainStatement.elseCase = currentIfStatement // Adds
    this.continue()
    return mainStatement
}
```

```
[Running] node src.js "/Users/pw/ocr-erl-interpreter/
! ERROR @line 4
Invalid Syntax: Expeceted nothing to follow endwhile
endwhile 2 + 2 == 4
        ^

Exited with code 1
```

➔ **No endwhile**

code.erl

```
i = 0
while i < 10
    i = i + 1
```

```
}
return new SyntaxError(whileToken, "Expected 'endwhile' to close loop")
```

A simple addition is needed after the iteration in build_while_loop(), identical to how it is implemented in build_if_chain().

MORE ERROR CASES

These are only the errors involved with parsing, but I have not tested a lot of errors which involve nesting and cases which have errors within these structures.

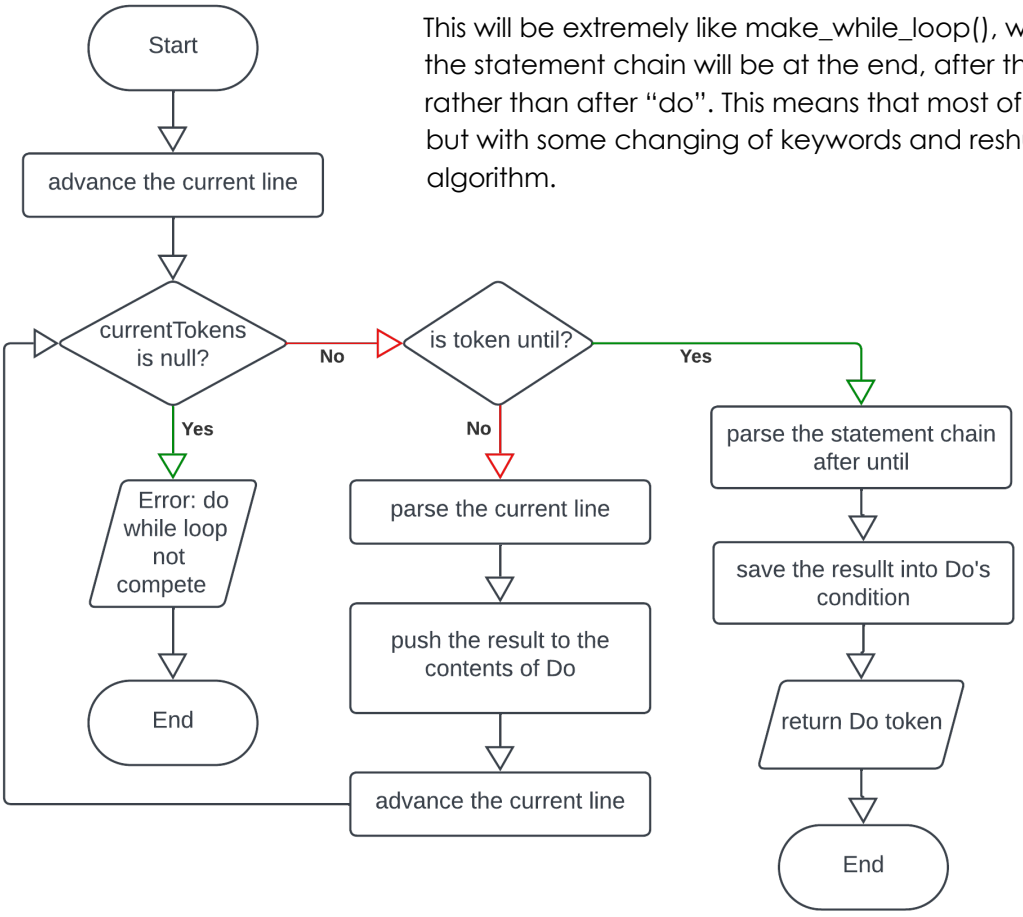
I first, however, will implement the other 2 types of errors, and then return at the end to potentially redesign evaluate_asts() and run() as I am currently not happy with how the system operates, and am anticipating a lot of issues when I run a greater number of tests.

DESIGN: DO UNTIL LOOPS

To reduce code repetition, there will now be a parent class for all Loops. Called Loop, While and the new Do class will extend it. All Loops will have the same attributes and evaluate() method, so these will be in the Loop class, with just a custom evaluate_condition() for each class.

As stated, Do will be the main token for do until loops, and it will be interacted with identically to the While loop in the Interpreter, with only the way it is constructed and its condition being different. It will be produced by "do" in make_identifier(), alongside a new TemplateKeyword: "until".

➔ make_do_loop()



This will be extremely like make_while_loop(), with the difference being that the statement chain will be at the end, after the closing keyword "until", rather than after "do". This means that most of the code should be the same, but with some changing of keywords and reshuffling of the order of the algorithm.

➔ evaluate_condition()

This is where the main difference will be. Because all loops will be run using evaluate_loop() in the Interpreter, I want to avoid using a JavaScript do while loop as this requires extra case checks, and instead want to use a standard while loop for every type of loop the interpreter has.

The solution to this is ensuring that the first time the method is called it always returns true, regardless of the condition.

Therefore, an attribute called firstPass will be added, and will be assigned false in the constructor. Whenever evaluate_condition() is called, it will check if firstPass is false. If it is false, then it will be set to true, and true will be returned by the method. Therefore, every following time that evaluate_condition() is called, it will take the other, default branch and will return the value of evaluating condition.

ERROR HANDLING

The benefit of this system is that is aligns with the pre-existing code, so only the new build_do_loop() will need to have checks for errors, which is very similar to build_while_loop(), and therefore the program will run seamlessly without any changes to the Interpreter.

DEVELOPMENT: DO UNTIL LOOPS

BASICS

```
388  class Loop extends Token{
389      constructor(position, line){
390          super(position, line)
391          this.condition = null
392          this.contents = []
393      }
394
395      evaluate(){
396          return this.contents
397      }
398  }
399
400  class While extends Loop{
401      constructor(position, line){
402          super(position, line)
403      }
404
405      evaluate_condition(){
406          let condition = this.condition.evaluate()
407          if (condition instanceof Error){
408              return condition
409          }
410          return condition.value
411      }
412  }
```

```
414  class Do extends Loop{
415      constructor(position, line){
416          super(position, line)
417          firstPass = false
418      }
419
420      evaluate_condition(){
421          if (!firstPass){
422              firstPass = true
423              return true
424          }
425          let condition = this.condition.evaluate()
426          if (condition instanceof Error){
427              return condition
428          }
429          return condition.value
430      }
431  }
```

The changes to while are shown on the left: the overall code is the same, but it is split across Loop as well .

Note that firstPass is not assigned as a property here, as it does not have the "this" keyword, so this was changed as otherwise it does not work.

```
case "do":
    return new Do(position, this.line)
case "const":
case "then":
case "endif":
case "endwhile":
case "until":
    return new TemplateKeyword(position, this.line, name)
```

BUILD_DO_LOOP()

```
build_do_loop(self){
    let doToken = self.token //do
    self.continue()
    if (self.token != null){
        return new SyntaxError(this.token, "Expeceted nothing to follow 'do'")
    }
    self.advance_line()
    while (self.currentTokens != null){ // adding contents
        self.continue()
        if (self.token instanceof TemplateKeyword){
            if (self.token.tag == "until"){ // finishing loop
                self.continue()
                if (self.token == null){
                    return new SyntaxError(self.previous, "Expected condition after 'until'")
                }
                let condition = self.statement_chain(self) // get condition
                if (condition instanceof Error){
                    return condition
                }
                doToken.condition = condition
                if (self.token != null){
                    return new SyntaxError(self.token, "Expected no tokens after condition")
                }
                return doToken // returned
            }
        }
        let result = self.parse() // default case
        if (result instanceof Error){
            return result
        }
        doToken.contents.push(result)
        self.advance_line()
    }
    return new SyntaxError(doToken, "Expected 'until' to close loop")
}
```

The code from build_while_loop() was copy pasted and rearranged to form this method, as they are very similar.

This led to a lot of errors due to not renaming elements properly, such as not changing the tag check from "endwhile" to "until", as well as still referencing whileToken in a lot of places or incorrectly named error messages.

The final version is shown on the left for simplification, as there were no major changes in its development.

EVALUATION

```
// Checks if a do until loop
if (this.token instanceof While){
    return this.build_do_loop(this)
}
```

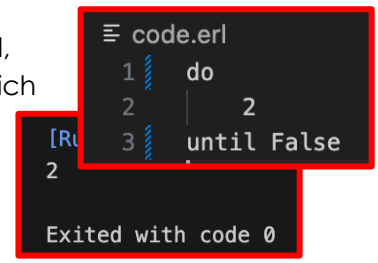
```
if (ast instanceof Loop){
    let result = this.evaluate_loop(ast)
    if (result instanceof Error){
        console.log(result.display())
        return 1
    }
    return result
}
```

```
return !condition.value
```

Another line has to be added in parse() in order to call build_do_loop() when a Do token is encountered.

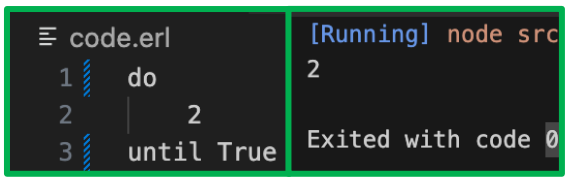
Finally, the line in evaluate_asts() must be altered to call evaluate_loop() for any instance of Loop, not just While.

From here, the test was semi-successful, running the loop only a single time, which would be the correct output it was until True, but it was until False.



This is because I have treated it like a do…while loop, rather than a do…until, where do until will run the contents when the condition is False, and stop when it is True, which is the opposite of what the current implementation did.

However, this fix was easy: changing evaluate_condition() to return the NOT of the value, therefore a false evaluation would return true, so it would continue to run, and when it evaluated to True, false would be returned and the until would be fulfilled, and the loop would stop running.

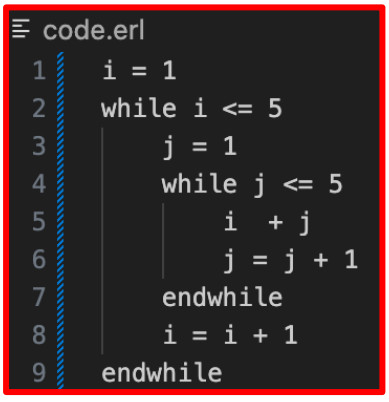


Now the do until loop is successful and will run the until True loop a single time before exiting.

ERROR CHECKING

When trying the possible cases, because the code had been based off the while loop code, all of the error cases from that had carried over, and therefore I could not find any occasions where Syntax Errors should be returned when they weren't.

The main issues were with the evaluation in the Interpreter code, so I trialled a few cases and the main issue was with nesting different loops.



The nested loops on the left continually printed 2,3,4,5,6; with the i value never incrementing. After a lot of debugging, it was due to the way return had been used within evaluate_asts(), and if the first ast was an instance of loop, then it would return out of the method before running the other lines, so the i = I + 1 line was fully ignored.

It's a simple change, but makes me dislike the current Interpreter system even more.

```
if (ast instanceof Loop){
    let result = this.evaluate_loop(ast)
    if (result instanceof Error){
        console.log(result.display())
        return 1
    }
    continue
}
```

FINAL CHANGES

```
class Do extends Loop{
    constructor(position, line){
        super(position, line)
        this.firstPassComplete = false
    }
```

Finally, I renamed the firstPass attribute to firstPassComplete, as I feel it is more descriptive and firstPass could potentially be confusing. I also updated this in evaluate_condition()

This concludes the section on Do loops, and now the final of the 3 loops can be implemented/

DESIGN: FOR LOOPS

This is the most complex of the three loops, due to the much larger number of potential inputs required and the structure having multiple different cases it can hold. There will be 3 new tokens for this module: the For token that extends Loop, and 3 new Template Keywords: "to", "step" and "next".

FOR

As it extends the Loop class, it will adopt the previous attributes and evaluate() method, but will also have many other new properties:

assignment ➔ holds the assignment ast to be evaluated for the first time
variable ➔ holds the identifier for the variable that is being iterated through and is set in assignment
finish ➔ holds the ast for the ending value of the for loop
step ➔ holds the step of the for loop with the default value being 1 if not stated
firstPassComplete ➔ a boolean, like used in Do to outline if the loop has iterated before

➔ evaluate_condition()

If firstPassComplete is false, then it will be set to true and the assignment property will be evaluated(). If it is false then the variable property's assign() method will be called to get the Data Type, and then set() will be called on that to increment the value by the value of step.

Then, the method will call evaluate() on the variable, and if the value is less than or equal to finish, it will return true, and will otherwise return false. It is important that it is less than or equal to because in ERL, for loops are inclusive with the finish value.

PARSING

As shown by the OCR specification on the right, the parsing for this stage will be more complex than the other loops due to the length of the first line in a for loop and the amount of detail it contains.

Overall, the BNF for a for loop is:

```
<for_loop> ::= For <assignment> "to" <expression>
               ("step" <expression>)
```

Where the brackets () represent an optional element, so therefore step does not have to be specified, and defaults to 1.

Furthermore, the closing line of the for loop is defined as:

```
<for_end> ::= "next" Identifier
```

```
for i=0 to 9
    print("Loop")
next i
```
This will print the word "Loop" 10 times, i.e. 0-9 **inclusive**.

```
for i=2 to 10 step 2
    print(i)
next i
```
This will print the even numbers from 2 to 10 inclusive.

```
for i=10 to 0 step -1
    print(i)
next i
```
This will print the numbers from 10 to 0 inclusive, i.e. 10, 9, 8,..., 2, 1, 0.

Note that the `'step'` command can be used to increment or decrement the loop by any positive or negative integer value.

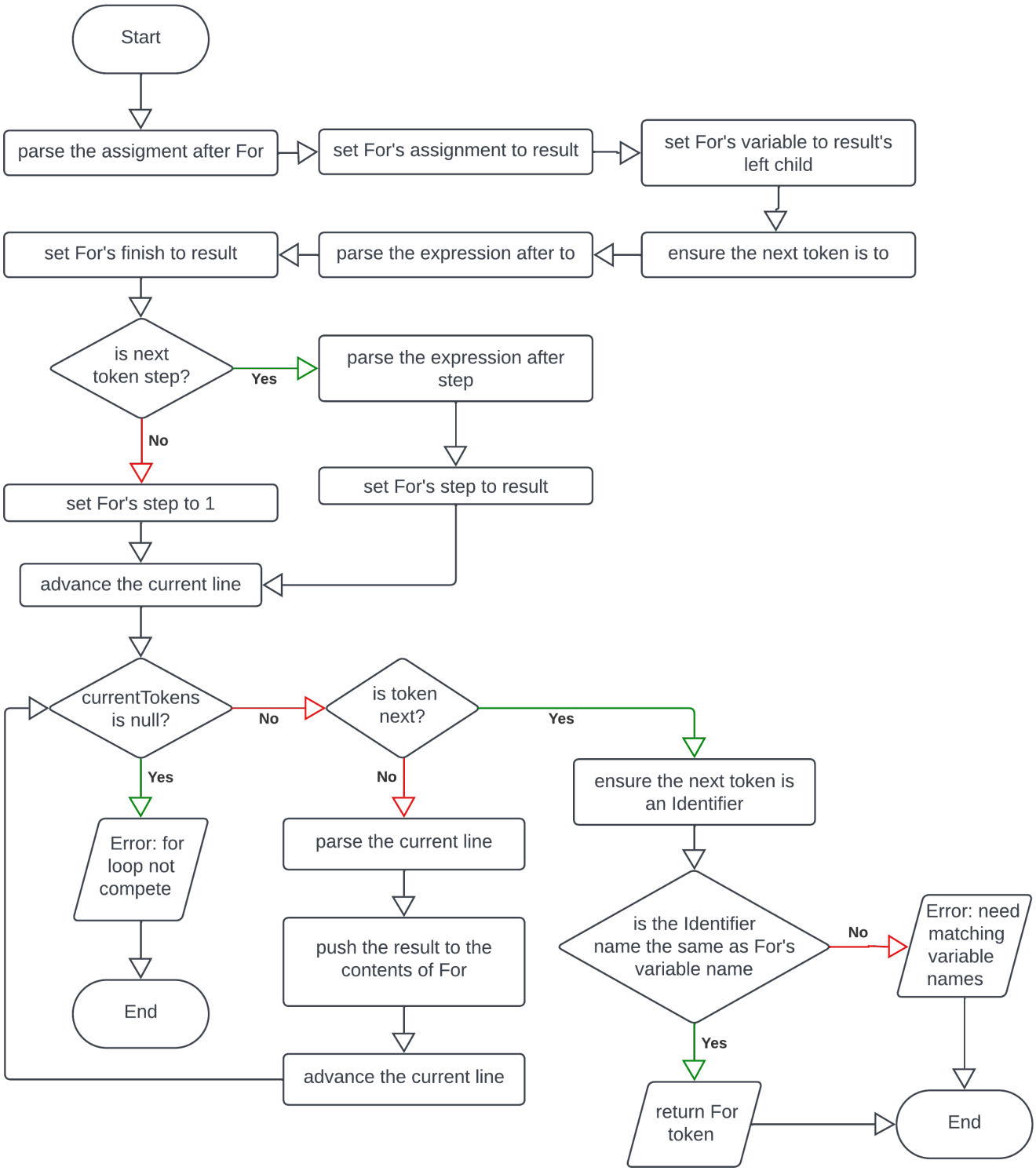
However, the name of the Identifier in `<for_end>` must match the name of the Identifier that is the left-hand child of `<assignment>` in `<for_loop>`, as it represents the variable incrementing.

When saved in the For instance, the `<assignment>` is stored in the assignment property, the left hand of that is stored in the variable property, the first `<expression>` is stored in finish, and the final, optional `<expression>` is stored in the step attribute.

➔ build_for_loop()

This is the method that will build the for loops based on the BNF and will be called in parse() when the For token is the first token in a line. This will follow the same general structure as the others, consisting of the big `while currentTokens != null` loop which repeatedly adds the lines to contents.

The flowchart for this method is as shown:



This shows how long the algorithm is going to be, mainly due to the parsing of the first line. This is because every possible invalid input will have to be considered, so there will have to be lots of checking if the current token is null and if the current token is what it is meant to be: therefore, significantly increasing the size of the method, however it won't be very technically complicated.

INTEGER CHECKS

Finally, due to the way floats round weirdly in computer science, I would like to avoid using float values within the For loops. This is good programming practice and is important for the students to understand, as floats can lead to some weird and unexpected cases, Therefore, a check when evaluate_condition() is called for the first time should take place, and an Error should be returned. I'm planning to introduce a new error type called TypeError, however I will plan in more detail when I approach the issue.

BASIC CHANGES

```
class For extends Loop{
    constructor(position, line){
        super(position, line)
        this.firstPassComplete = false
        this.variable = null
        this.assignment = null
        this.finish = null
        this.step = null
    }

    evaluate_condition(){
        if (this.firstPassComplete){
            this.variable.assign().set(this.variable.evaluate() + this.step)
        } else {
            this.firstPassComplete = true
            this.assignent.evaluate(
        }
        return this.variable.evaluate() <= this.finish ? true : false
    }
}
```

```
case "to":
case "step":
case "next":
    return new TemplateKeyword(position, this.line, name)
```

```
this.assignment.evaluate()
```

This is my first attempt at implementing the For class

I will return to it later, as I did not realise the evaluate_condition() does not work at all, but I cannot debug it until the parsing is completed and I can run some actual test cases.

The issue is mainly that I forget I am using my custom data types rather than JavaScript ones, so I need to use the value property.

BUILD_FOR_LOOP()

```
    build_for_loop(self){
        let forToken = self.token // for
        self.continue()
        if (self.token == null){ // ensures tokena fter for
            return new SyntaxError(forToken, "Expected assignment after 'for'")
        }
        let result = self.assignment(self) //assignment
        if (result instanceof Error){
            return result
        }
        forToken.assignment = result
        forToken.variable = result.left
        if (self.token == null){
            return new SyntaxError(forToken, "Expected 'to'")
        }
        if (!self.token instanceof TemplateKeyword){ // ensures next token is to
            if (self.token.tag != "to"){
                return new SyntaxError(self.token, "Expected 'to'")
            }
        }
        let errorToken = self.token
        self.continue()
        if (self.token == null){
            return new SyntaxError(errorToken, "Expected expression after 'to'")
        }
        result = self.expression(self)
        if (result instanceof Error){
            return result
        }
        forToken.finish = result
        if (self.token == null){ // no step keyword
            forToken.step = 1
        } else {
            if (!self.token instanceof TemplateKeyword){ // ensures next token is step
                if (self.token.tag != "step"){
                    return new SyntaxError(self.token, "Expected 'step' or end of line")
                }
            }
            errorToken = self.token
            self.continue()
            if (self.token == null){
```

In my initial attempt, I tried to implement as much error handling as I could, based off what I had learnt from building the previous loops.

The size of the method is so huge that it spans across two screens. This could potentially mean that it needs to be subdivided further, but I do not feel that Is necessary as it is just a long sequence of checks so it doesn't make it much simpler from that approach.

I've ensured that I comment the important elements to distinguish the order that the first line is parsed.

```
1099                    return new SyntaxError(errorToken, "Expected expression after 'step'")
1100                }
1101                result = self.expression(self)
1102                if (result instanceof Error){
1103                    return result
1104                }
1105                forToken.step = result
1106                if (self.token != null){
1107                    return new SyntaxError(self.token, "Expected end of line")
1108                }
1109            }
1110            self.advance_line()
1111            while (self.currentTokens != null){
1112                self.continue()
1113                if (self.token instanceof TemplateKeyword){
1114                    if (self.token.tag == "next"){
1115                        errorToken = self.token
1116                        self.continue()
1117                        if (self.token == null){
1118                            return new SyntaxError(errorToken, `Expected '${forToken.variable.name}' after 'next'`)
1119                        }
1120                        if (!self.token instanceof Identifier){
1121                            if (self.token.name != forToken.variable.name){
1122                                return new SyntaxError(self.token, `Expected '${forToken.variable.name}'`)
1123                            }
1124                        }
1125                        self.continue()
1126                        if (self.token != null){
1127                            return new SyntaxError(self.token, `Expected no tokens after '${forToken.variable.name}'`)
1128                        }
1129                        return forToken
1130                    }
1131                }
1132                let result = self.parse()
1133                if (result instanceof Error){
1134                    return result
1135                }
1136                forToken.contents.push(result)
1137                self.advance_line()
1138            }
1139            return new SyntaxError(forToken, `Expected 'next ${forToken.variable.name}' to close loop`)
1140        }
```

```
// Check if a for loop
if (this.token instanceof For){
    return this.build_for_loop(this)
}
```

After adding the check in parse() to call the method it is then complete.

```
code.erl
1  for i = 0 to 9
2      i
3  next i
```

```
[Running] node src.j
0

Exited with code 0
```

However, when attempting to run this code, an incorrect error occurred. However, because the program did not crash, it was a good sign that the parsing had worked correctly, and that the issue was with the evaluation.

EVALUATION FIX

```
evaluate_condition(){
    if (this.firstPassComplete){
        this.variable.assign().set(new Integer(this.variable.position, this.variable.line, this.variable.evaluate().value + this.step))
    } else {
        this.firstPassComplete = true
        this.assignment.evaluate()
    }
    return this.variable.evaluate().value <= this.finish ? true : false
}
```

```
[Running] node src.js

Exited with code 0

[Done] exited with co
```

When I realise that I needed to use my custom data types, I changed the value that the variable is set to be an Integer, and for its value to be the value of its old evaluate method, plus the step value. However, this still did not work.

```
return this.variable.evaluate().value <= this.finish.value ? true : false
```

```
[Running] node s
Integer { position: 8, line: 0, value: 0 }
Integer { position: 13, line: 0, value: 9 }
false

Exited with code 0
```

After some debugging, the return statement also must be changed, to compare the value property of finish, and the value of evaluate(), and then the intended result was outputted, as shown.

```
ing] node src.js
0
1
2
3
4
5
6
7
8
9

Exited with code 0
```

```
code.erl
1  for i = 10 to 9 step 2
2      i
3  next i
```

```
[Running] node src.j

Exited with code 0
```

When testing step for the first time it was also broken, however this was due to the same issue: the variable was being incremented by `this.step`, not `this.step.value`.

```
riable.line, this.variable.evaluate().value + this.step.value))
```

```
[Running] node src.j
0
2
4
6
8
10

Exited with code 0
```

Finally, the default step value set in build_for_loop() needs to be changed to a custom Integer data type with value one in order for this to work.

```
if (self.token == null){ // no step keyword
    forToken.step = new Integer(self.previous.position, self.previous.line, 1)
} else {
```

Potentially setting position and null might be a better idea, but largely doesn't matter as does not affect the program in any meaningful way.

ERROR HANDLING

There is a lot to test here, so I have split it into different sections.

INCOMPLETE INPUTS IN THE FIRST LINE

```
 ! ERROR @line 1
Invalid Syntax: Expected assignment after 'for'
for
^
```

➔ Just for keyword

Correctly handles this.

➔ Incomplete assignment

```
code.erl
1  for i
```

This causes the program to crash.

```
[Running] node src.js "/Users/pw/ocr-erl-interpreter/code.er
/Users/pw/ocr-erl-interpreter/src.js:478
        super(token.position, token.line)
                    ^

TypeError: Cannot read properties of null (reading 'positior
```

This is due to the way assignment() is defined, as it currently has no error handling for the first stages, because prior to this it was only ever called when parse() detected an Equals as the second token in the stream, so therefore checking if an Equals was present was redundant.

I therefore redefined assignment() to include error handling for not having an Identifier or Equals or Expression, as well as more null token checks, so that any incomplete assignment errors are returned to build_for_loop()

```
assignment(self){
    let tag = null
    if (self.token instanceof TemplateKeyword){
        if (self.tokem.tag == "const"){
            tag = "const"
            self.continue()
        }
    }
    let variable = self.token
    if (!variable instanceof Identifier){
        return new SyntaxError(variable, "Expected identifier")
    }
    if (tag == "const"){
        variable = new Identifier(variable.position, variable.line, variable.name, true)
    }
    self.continue()
    let equals = self.token
    if (equals == null){
        return new SyntaxError(variable, "Expected '=' to follow identifier")
    }
    if (!equals instanceof Equals){
        return new SyntaxError(equals, "Expected equals")
    }
    self.continue()
    if (self.token == null){
        return new SyntaxError(variable, "Expected expression to follow '='")
    }
    equals.right = self.statement_chain(self)
    if (equals.right instanceof Error){
        return equals.right
    }
    equals.left = variable
    return equals
}
```

```
if (self.token.tag == "const"){
```

I also removed the tag parameter and instead checked for a tag in the method itself, as it makes it fully self-contained, and it is a more logical place for it to occur.

```
// Check if there is a tagged assignment
if (this.token instanceof TemplateKeyword){
    switch(this.token.tag){
        case "const": // Constant
            let result = this.assignment(this)
            return this.check_result(result) ? result : new SyntaxError(this.token, "Expected operator")
```

Therefore, parse() also needs to be updated for the "const" call, so that it no longer passes the tag as a parameter which significantly simplifies and increases how easy it is to read the code.

```
code.erl
1  i = 2
2  i
3  const a = 5
4  a
```

```
[Running] node src.
2
5

Exited with code 0
```

Just to ensure that it still functions correctly, I ran a few tests with them used as normal assignments (with an example on the left) and they still worked as intended.

However, this now also expands error handling for For loops:

```
 ! ERROR @line 1
Invalid Syntax: Expected '=' to follow identifier
for i
    ^

Exited with code 1
```

```
 ! ERROR @line 1
Invalid Syntax: Expected expression to follow '='
for i =
      ^

Exited with code 1
```

➔ **No "to" after assignment.**

```
[Running] node src.js "/Users/p
 ! ERROR @line 1
Invalid Syntax: Expected 'to'
for i = 5
^

Exited with code 1
```

Technically, this error is caught, however the pointer for location is completely unrepresentative of the issue. This has also been a problem in other Error messages: where they point to the initial Token of the structure rather than the point that is relevant to the Error.

Therefore, I feel like a new feature is needed to fix this.

```
class Parser {
    constructor(to
        this.allTo
        this.line
        this.curre
        this.posit
        this.token = null // The corresponding token for the position
        this.previous = null // The previous token
    }
```

```
continue(){
    this.position += 1
    this.previous = this.token
    this.token = this.position == this.currentTokens.length ? null : this.currentTokens[this.position]
}
```

The solution is to add a new property to the parser, called previous,

This will be updated in continue() and will always hold the value of the previous token. It can therefore be passed as the token in Error messages and much more accurately points to the error.

```
if (self.token == null){
    return new SyntaxError(self.previous, "Expected 'to'")
}
```

```
 ! ERROR @line 1
Invalid Syntax: Expected 'to' to follow assignment
for i = 5
      ^

Exited with code 1
```

As show, this error message is now much more useful for the user, as it more precisely points to the issue. This can also be used in the other build methods:

```
return new SyntaxError(self.previous, "Expected 'then")
```

```
if (self.token == null){
    return new SyntaxError(self.previous, `Expected '${forToken.variable.name}' after 'next'`)
}
```

```
if (self.token == null){
    return new SyntaxError(self.previous, "Expected condition after 'until'")
}
```

➔ **“step” related errors**

```
 ! ERROR @line 1
Invalid Syntax: Expected expression after 'step'
for i = 5 to 7 step
                   ^
```

```
 ! ERROR @line 1
Invalid Syntax: Expected end of line
for i = 5 to 7 step 5 then
                     ^
```

```
 ! ERROR @line 1
Invalid Syntax: Expected expression after 'step'
for i = 5 to 7 then
              ^
```

The code for checking if a token other than step was present was incorrect: this new updated version has some repeated code but it is the only way as otherwise the tag property may be called for a token that does not have the tag property.

```
if (!(self.token instanceof TemplateKeyword)){ // ensures next token is step
    return new SyntaxError(self.token, "Expected 'step' or end of line")
} else if (self.token.tag != "step"){
    return new SyntaxError(self.token, "Expected 'step' or end of line")
}
```

This now correctly functions:

```
 ! ERROR @line 1
Invalid Syntax: Expected 'step' or end of line
for i = 5 to 7 then
              ^
```

Now every possible case for the first line of code has correct error handling.

CLOSING LINE ERRORS

➔ **No closing statement or lone “next”**

```
 ! ERROR @line 1
Invalid Syntax: Expected 'next i' to close loop
for i = 5 to 7
^
```

```
 ! ERROR @line 3
Invalid Syntax: Expected 'i' after 'next'
next
^
```

➔ **Incorrect variable name**

code.erl

```
1  for i = 5 to 7
2      2
3  next p
```

```
if (!self.token instanceof Identifier){
    return new SyntaxError(self.token, `Expected identifier named '${forToken.variable.name}'`)
}
if (self.token.name != forToken.variable.name){
    return new SyntaxError(self.token, `Expected identifier to be named '${forToken.variable.name}'`)
}
```

On the first test, it outputted as if it was a normal loop, but that was because I had put the name checking if statement within the Identifier checking one, so after separating them out as shown it worked as expected.

```
[Running] node src.js "/Users/pw/ocr-erl-interprete
 ! ERROR @line 3
Invalid Syntax: Expected identifier to be named 'i'
next p
     ^
```

INTEGERS ONLY

code.erl

```
1  for i = 0 to 1 step 0.1
2      i
3  next i
4  i == 1
```

```
[Running
0
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
False

Exited with code 0
```

As briefly described in the design stage, I feel as if it would be good to remove the ability to use floats within for loops, because of the really unexpected results they make, due to the odd behaviour when adding together floats. A for loop should always end with the final value and as shown in this example, using floats does not mean this is always true.

code.erl

```
1  for i = 0 to 10
2      i / 10
3  next i
```

The much better approach, which should be used, is the one on the left to produce more accurate results.

```
[Running] node src.
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

Exited with code 0
```

Note for some reason, the Divide evaluate() method previously returned its two children multiplied, which had to be fixed.

```
result = left.value * right.value
```

TYPE ERROR

```
class TypeError extends Error{
    constructor(token, description='') {
        super(token.position, token.line)
        this.token = token
        this.description = description
    }

    display(){
        return ` ! ERROR @line ${this.line}\nType Error: ${this.description}\n${this.location()}`
    }
}
```

This will be the Error for data-type related issues and will be used significantly in the next module of code, but for now it is a basic copy of the other Errors.

```
} else {
    this.firstPassComplete = true
    this.assignment.evaluate()
    if (!this.variable.evaluate() instanceof Integer){
        return new TypeError(this.variable, "Starting value is not an Integer")
    }
    if (!this.finish instanceof Integer){
        return new TypeError(this.finish, "Final value is not an Integer")
    }
    if (!this.step instanceof Integer){
        return new TypeError(this.variable.value, "Step value is not an Integer")
    }
}
```

From here, when the loop is evaluated for the first time, so when firstPassComplete is false, the checks will run to ensure the starting value, final value and step are all integers, and a TypeError will be returned if this is not the case.

code.erl

```
for i = 0.5 to 10
    i
next i
```

When testing this, it did not catch the error and the loop executed using the float values, which was confusing.

```
if (!(this.variable.evaluate() instanceof Integer)){
    return new TypeError(this.variable, "Starting value is not an Integer")
}
if (!(this.finish instanceof Integer)){
    return new TypeError(this.finish, "Final value is not an Integer")
}
if (!(this.step instanceof Integer)){
    return new TypeError(this.variable.value, "Step value is not an Integer")
}
```

However, it turned out to be due to the order of operations: with brackets being required as ! has higher precedence than instanceof. I'm sure I've had this issue previously in this project, but it still caught me out, and when searching the program for other occasions on this error there were 9 more, which I fixed so they would work as I intended.

ERROR MESSAGES

```
if (!(this.variable.evaluate() instanceof Integer)){
    return new TypeError(this.variable.evaluate(), "Starting value is not an Inte
}
```

Initially the starting value not being an Integer pointed to the Identifier, so I changed it to point at the value, and then the correct error message was returned.

```
[Running] node src.js "/Users/pw/ocr-erl-inte
 ! ERROR @line 1
Type Error: Starting value is not an Integer
for i = 0.5 to 10
        ^

Exited with code 1
```

The step error check also had to be changed to point to the step rather than the variable value (which it was for some reason), but after that all the error messages were correct.

```
[Running] node src.js "/Users/pw/ocr-erl-inte
 ! ERROR @line 1
Type Error: Final value is not an Integer
for i = 1 to 9.5
            ^

Exited with code 1
```

```
if (!(this.step instanceof Integer)){
    return new TypeError(this.step, "Step value is not an Integer")
}
```

```
[Running] node src.js "/Users/pw/ocr-erl-
 ! ERROR @line 1
Type Error: Step value is not an Integer
for i = 1 to 10 step 0.2
                    ^

Exited with code 1
```

NEGATIVE STEP FOR LOOPS

```
code.erl
1   for i = 0 to -10 step -1
2       i
3   next i
```

Currently when attempting to run a for loop with a negative step, the program returns a weird error to do with non-integer values.

```
! ERROR @line NaN
Type Error: Final value is not an Integer
undefined
    ^
```

My initial thought is that this was to do with how my program dealt with negative numbers in factor(): it would return a Minus token, with zero on the left and the right being the value.

Therefore, when checking if the value is an Integer, it sees the Minus token and therefore thinks that it is not, so this system needs to be updated. This needs to be done anyway as it is not a very efficient method.

```
if (self.check_instance(Minus)){ // Minus unary operator
    self.continue()
    if (self.token == null){
        return new SyntaxError(self.previous, "Incomplete input")
    }
    let result = self.factor(self)
    if (result instanceof Error){
        return result
    }
    result.value = -result.value
    return result
} // Two operators in a row
```

```
! ERROR @line 1
Invalid Syntax: Expected literal
-7
^
```

Now, the program directly modifies the value of the token, which also improves memory and therefore performance. I also changed it to use the new previous attribute, rather than saving an errorToken.

When running a test, this worked when the negative numbers were in the middle of an expression, but the expression could not start with a negative number.

FIXING STARTING NEGATIVES

This is due to the check for binary operators in parse() and statement(), to ensure that the current token is not something like a * which would crash the program. However, this should not be the case if it is Add or Minus. Therefore. A new method will be added to check this, to reduce code repetition across methods.

```
check_binary_operator(){
    if (this.token instanceof BinaryOperator){
        if (this.token instanceof Add || this.token instanceof Minus){
            return false
        }
        return true
    }
    return false
}
```

```
// Check if the first token is a binary operator
if (this.check_binary_operator()){
    return new SyntaxError(this.token, "Expected literal")
}
```

```
if (this.check_binary_operator()){
    return new SyntaxError(self.token, "Expected literal")
}
```

This was then called in parse() and half_statement() to replace the old checks for Binary Operators.

```
-7

Exited with code 0
```

```
! ERROR @line 1
Invalid Syntax: Incomplete input
-
^
```

Now the line can start with negative numbers, or numbers beginning with positive signs, and single Plus or Minus operators still cause errors, ensuring that the program does not crash.

UPDATING FOR LOOP CHECKS

```
let result = this.variable.evaluate()
if (!(result instanceof Integer)){
    return new TypeError(result, "Starting value is not an Integer")
}
result = this.finish.evaluate()
if (!(result instanceof Integer)){
    return new TypeError(result, "Final value is not an Integer")
}
result = this.step.evaluate()
if (!(result instanceof Integer)){
    return new TypeError(result, "Step value is not an Integer")
}
```

Even though negative can be used by themselves now, any expression that was in any of the parameters would not pass the Integer check, because I must call evaluate() to get the result, and then check if that is an Integer. However, the test case outputted nothing.

```
code.erl
1   for i = 2 * 3 to 3 ^ 3 step 4 / 2
2       i
3   next i
```

This was because the final return statement and the setting statement both treated variable, finish, and step as if they were Integers and therefore did not call evaluate() as they could potentially be asts.

```
evaluate_condition(){
    let variableResult = this.variable.evaluate()
    if (variableResult instanceof Error){
        return variableResult
    }
    let finishResult = this.finish.evaluate()
    if (finishResult instanceof Error){
        return finishResult
    }
    let stepResult = this.step.evaluate()
    if (stepResult instanceof Error){
        return stepResult
    }
    if (this.firstPassComplete){
        this.variable.assign().set(new Integer(this.variable.position, this.variable.
    } else {
        this.firstPassComplete = true
        this.assignment.evaluate()
        if (!(variableResult instanceof Integer)){
            return new TypeError(variableResult, "Starting value is not an Integer")
        }
        if (!(finishResult instanceof Integer)){
            return new TypeError(finishResult, "Final value is not an Integer")
        }
        if (!(stepResult instanceof Integer)){
            return new TypeError(stepResult, "Step value is not an Integer")
        }
    }
    return variableResult.value <= finishResult.value ? true : false
}
```

```
ble.line, variableResult.value + stepResult.value))
```

In order to minimise calling, evaluate() as much as possible, evaluate_condition() now sets the result of the 3 evaluate() calls to variables and ensures they are not errors. Now, the rest of the method uses those variables instead of the attributes themselves, using their value to change the variable value; using them to check for Integers; comparing their values for the return value of the method.

It is important to not call evaluate() multiple times as in the future there may potentially be input() calls within the evaluate() calls, so if evaluate() is called multiple times it could prompt multiple input calls when there should be a single input which would cause issues.

```
! ERROR @line 1
Identifier Error: 'i' was not declared
for i = 2 * 3 to 3 ^ 3 step 4 / 2
    ^
```

This issue now occurs because variable.evaluate() is called before evaluating the assignment to give it value.

```
evaluate_condition(){
    if (!this.firstPassComplete){
        this.assignment.evaluate()
    }
```

This requires this little check at the start of the program, but the Integer checks and setting firstPassComplete to true still occur later in the program where they used to, only moving the assignment evaluation up.

```
6
8
10
12
14
16
18
20
22
24
26
28
```

Now expressions can be used in For loops, as well as negative numbers.

UPDATING CONDITIONS

code.erl

```
1   end = 5
2   for i = 0 to end
3       i
4       end = end + 1
5   next i
```

The issue with the current system is that a for loop which includes variables in its parameters will have constantly changing finish values. For example, the for loop on the left will continually output increasing numbers, as its finish condition keeps changing. I'm not currently sure if I want this in my program, as the other option is taking the initial values of the finish and step and storing and using them.

When asking my stakeholder Tanish, he said that:

➔ *"I don't think it's good practice to change the loop condition variable within the for loop, so it's best to exclude it as it is for kids to learn coding in school. You could then also have cases like* `for i = 0 to i + 2` *or something, which look weird and would result in this infinite loop thing, so I think you should take the starting values and keep those, and base the program of that, kind of like python's range feature"*

I agree with this idea, so will change evaluate_condition() to reflect this new method, he then also then pointed out a second issue:

➔ *"Should the variable that the for loop is iterating through be able to change? As in, I know you can change it, but when one iteration ends should one be added to that new value, or should it be reverted to the old value and one added to that? I think python would do the second option but the range method is different to traditional for loops so I'm not sure which one in this case would be best."*

Although I think changing the value is bad practice, I think it would be difficult to implement a system to completely block changing the value. My worry then is that infinite loops could be created if something along the lines of `i = i - 1` was included within the loop, and there would be no stop. Therefore, a python like approach where the value of the iterated variable is forcefully set to the expected next value would be best and cause less cases where infinite loops could occur.

➔ *"Yeah, that sounds like the right idea."*

So, with this, a redesign of evaluate_condition() is definitely needed, as these changes are going to need a new system to implement them.

NEW SYSTEM

```
first_pass(){
    let assignment = this.assignment.evaluate()
    if (assignment instanceof Error){
        return assignment
    }
    this.currentVariableValue = this.variable.evaluate()
    if (!(this.currentVariableValue instanceof Integer)){
        return new TypeError(this.currentVariableValue, "Starting value is not an Integer")
    }
    this.finish = this.finish.evaluate()
    if (this.finish instanceof Error){
        return this.finish
    }
    if (!(this.finish instanceof Integer)){
        return new TypeError(this.finish, "Final value is not an Integer")
    }
    this.step = this.step.evaluate()
    if (this.step instanceof Error){
        return this.step
    }
    if (!(this.step instanceof Integer)){
        return new TypeError(this.step, "Final value is not an Integer")
    }
    this.increasing = this.currentVariableValue.value < this.finish.value
    this.firstPassComplete = true
}
```

The new system has a separate method for the first pass, to increase readability. As well as increased error checking, this now sets this.finish and this.step to the result of their initial evaluate() call, and they stay as that for the rest of the program, so any changes in variable do not affect them.

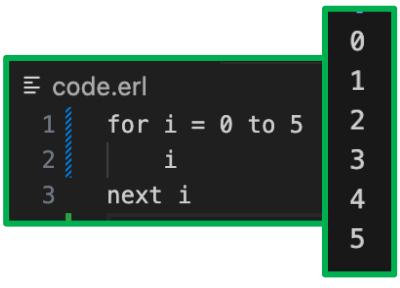
The value of the variable is set as currentVariableValue, a new property. this.variable is kept so its value can be updated at the start of each cycle, but currentVariableValue represents its value for the object, so any changes to the variable don't affect the loop.

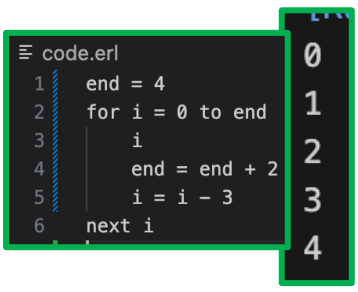
Finally, an increasing property is added, based off whether the loop will be increasing or decreasing, but this will go into more detail later as it currently requires more case handling.

```
evaluate_condition(){
    if (this.firstPassComplete){
        this.currentVariableValue.value += this.step.value
        this.variable.assign().set(this.currentVariableValue)
    } else {
       this.first_pass()
    }
    if (this.increasing){
        return this.currentVariableValue.value <= this.finish.value
    }
    return this.currentVariableValue.value >= this.finish.value
}
```

So, when not the first pass, the value of the currentVariableValue will be incremented by the step value, and then the variable will be assigned this as its new value so it can be used in the program. This means that if the value of variable is changed, it will not affect the loop at all, as it is completely based on current which cannot be changed in the code.

Finally, if increasing is true or false decides which comparator is used.





The test cases show that this new method still allows normal for loops to occur, but the one on the right also shows that the value of i is reset to its actual value after each iteration, and that the initial values of the loop are what affects the condition: the final value is the same

NEGATIVE STEP

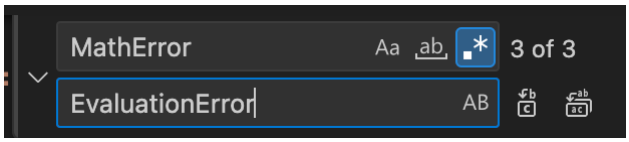
```
code.erl
1  for i = 5 to 1 step -1
2      i
3  next i
```

```
5
4
3
2
1
```

With this new system, negative step now works due to the new increasing property, however some new cases are required when the step is negative but the bounds are not and vice versa, to ensure no crashes occur for.

```
code.erl
1  for i = 0 to 5 step -1
2      i
3  next i
```

```
-11138
-11139
-11140
-11141
```



For this, I have decided to rename MathError to EvaluationError, as MathError was too specific and I need a more generic Error for issues that occur when the program is evaluating, which is different from syntax or type errors. I also updated the error message to describe this type of error more accurately.

```
if (this.currentVariableValue.value < this.finish.value && this.step.value > 0){
    this.increasing = true
} else if (this.currentVariableValue.value > this.finish.value && this.step.value < 0){
    this.increasing = false
} else if (this.step.value == 0){
    return new EvaluationError(this.step, "Step must have non-zero value")
} else {
    return new EvaluationError(this.step, "Step value must align with bounds of for loop")
}
```

Therefore, at the end of first_pass(), this check occurs, and if the step and bounds do not align, an EvaluationError will be returned. Additionally, a check that the step is not zero is added as this should not be permitted.

```
let result = this.first_pass()
if (result instanceof Error){
    return result
}
```

This should have been added earlier, but a check needs to also be added in evaluate_condition(), so that if first_pass() returns an Error, it is returned.

```
! ERROR @line 1
Evaluation Error: Step value must align with bounds of for loop
for i = 0 to 5 step -1
                    ^
```

Now, this is correctly handled, and For loops are decently robust now: more testing will be needed but I am happy with their current state.

LARGE ISSUES

```
code.erl
1  for i = 0 to 3
2      for j = i to 3
3          j
4      next j
5  next i
```

```
0
1
2
3
```

Whenever nested loops are involved, the entire system breaks down. This is very similar to the other loops and if statements, and the entire run() has not been built very robustly so requires a full redesign.

So, work on for loops is complete for now, however I am now adding an extra stage in order to redesign the interpreter.

DESIGN: CHANGING THE RUN SYSTEM

Currently, I feel like there are two main issues with the system. The first is that there are loops being combined with return statements in the same method, which leads to a lot of errors. Secondly, different evaluation methods return different things: sometimes asts, or tokens or 0/1. There is no consistency which leads to a lot of issues, so this needs to be fixed.

➔ evaluate_single_ast()

This is the main method for evaluation and will take a single ast as an input, containing the contents within the loop of the old evaluate_asts(), but with no iteration to avoid conflict between loops and using return. It will have the same checks as always but will be made to only ever return 0 and 1, and when loops and if statements are encountered, their corresponding methods will always return 0 or 1, so it will check any calls, and if the response is 1 then it will return 1.

➔ evaluate_many_asts()

This method contains a loop that will call evaluate_single_ast() until it has iterated through all the asts in its parameter. Each time, it will check if the value it gets back is 1, if it is, it will return 1, and will otherwise return 0 after all the asts have been iterated through.

➔ evaluate_loop()

Very similar to previous, but will only return 0 or 1. Therefore, if the condition is an Error, it will not return the Error and will instead output it in that method and then return 1. Otherwise, it is very similar, also checking each time it calls evaluate_many_asts() to see if it has been returned 1, where it will also return 1.

➔ run()

The same as always, now calling evaluate_single_ast() each time it calls parse_next().

Hopefully, this should deal with a lot of the issues previously had.

DEVELOPMENT: CHANGING THE RUN SYSTEM

```
evaluate_single_ast(ast){
    if (ast instanceof IfStatement){
        return this.evaluate_many_asts(ast.evaluate())
    }
    if (ast instanceof Loop){
        if (this.evaluate_loop(ast) == 1){
            return 1
        }
    }
    if (ast instanceof Error){
        console.log(ast.display())
        return 1
    }
    if (ast == null){
        return 0
    }
    let evaluated = ast.evaluate()
    if (evaluated != null){
        console.log(evaluated.display())
    }
    if (evaluated instanceof Error){
        return 1
    }
    return 0
}
```

```
run(){
    if (!this.have_tokens){
        let result = this.make_tokens()
        if (result instanceof Error){
            console.log(result.display())
            return 1
        }
    }
    let parser = new Parser(this.tokens)
    let ast = parser.parse_next()
    while (ast != null){
        if (this.evaluate_single_ast(ast) == 1){
            return 1
        }
        ast = parser.parse_next()
    }
    return 0
}
```

```
evaluate_many_asts(asts){
    if (!Array.isArray(asts)){
        asts = [asts]
    }
    for (let ast of asts){
        if (this.evaluate_single_ast(ast) == 1){
            return 1
        }
    }
    return 0
}
```

```
evaluate_loop(loop){
    let condition = loop.evaluate_condition()
    if (condition instanceof Error){
        console.log(condition.display())
        return 1
    }
    while (condition){
        if (this.evaluate_many_asts(loop.evaluate()) == 1){
            return 1
        }
        condition = loop.evaluate_condition()
        if (condition instanceof Error){
            console.log(condition.display())
            return 1
        }
    }
    return 0
}
```

This was my first attempt at adding these changes.

code.erl

```
for i = 0 to 3
    for j = i to 3
        j
    next j
next i
```

```
[Running] node sr
0
1
2
3
/Users/pw/ocr-erl-interpreter/src.js:1317
            console.log(evaluated.display())
                                          ^

TypeError: evaluated.display is not a function
```

When running the same script as earlier, instead of just stopping, the program now crashes: which is a good sign as it shows where progress can be made. It seems as if something is being evaluated which should not have been, so some debugging is required.

```
[ Identifier { position: 8, line: 2, name: 'j', constant: false } ]
```

Initially when printing the value of evaluated, it showed an Identifier within an array, so I removed the now redundant array check in evaluate_many_asts().

```
}
console.log("Next item:")
console.log(ast)
let evaluated = ast.evaluate()
console.log(evaluated)
if (evaluated != null){
    console.log(evaluated.displ
```

```
Next item:
For {
  position: 4,
  line: 1,
  condition: null,
```

However, when doing further outputs, somehow a For loop had got to the evaluate() call within evaluate_single_ast(), instead of being caught within the for Loop.

This was because, in the Loop catch, if 0 was returned by evaluate_loop(), it would continue rather than being returned, and would therefore continue to be evaluated and attempted to be displayed.

```
if (ast instanceof Loop){
    return this.evaluate_loop(ast)
}
```

This was a very easy change and shows the benefit of separating the recursion and returns, and now the program no longer crashed, but still prematurely stopped the counter after printing the first 3.

LOOP ISSUES

```
variable: Identifier { position: 4, line: 0, name: 'i', constant: false },
currentVariableValue: Integer { position: 8, line: 0, value: 5 },
assignment: Equals {
```

I then decided to do some debugging in evaluate_loop() as this was likely the source of the error.

When outputting the main For token at the end of the first nested loop, somehow the value of its currentVariableValue had increased when it wasn't supposed to.

```
before 0 4
after 0 5
```

To confirm this, I made evaluate_condition() output the position of the For loop (to distinguish the two) and the before and after values of currentVariableValue.value before and after it was set, and somehow the main For loops had started at 4, the final value of the nested one.

Somehow, whenever the nested For loop was incremented, it also changed the value of the main For loop, even though they should be two completely different properties.

```
let mainLoop = new Parser(new Lexer("for i = 0 to 3\nfor j = i to 3\nj\nnext j\nnext i\n".split("\n")).make_tokens()).parse
let nestedLoop = mainLoop.contents[0]
mainLoop.evaluate_condition()
nestedLoop.evaluate_condition()
console.log("MAIN", mainLoop.currentVariableValue)
console.log("NESTED", nestedLoop.currentVariableValue)
nestedLoop.evaluate_condition()
console.log("MAIN", mainLoop.currentVariableValue)
console.log("NESTED", nestedLoop.currentVariableValue)
```

```
MAIN Integer { position: 8, line: 0, value: 0 }
NESTED Integer { position: 8, line: 0, value: 0 }
MAIN Integer { position: 8, line: 0, value: 1 }
NESTED Integer { position: 8, line: 0, value: 1 }
```

Even when manually calling each method, the same result still occurred.

```
console.log("MAIN", mainLoop.currentVariableValue)
console.log("NESTED", nestedLoop.currentVariableValue)
nestedLoop.currentVariableValue = "test"
console.log("MAIN", mainLoop.currentVariableValue)
console.log("NESTED", nestedLoop.currentVariableValue)
```

```
MAIN Integer { position: 8, line: 0, value: 0 }
NESTED Integer { position: 8, line: 0, value: 0 }
MAIN Integer { position: 8, line: 0, value: 0 }
NESTED HELLO
```

However, manually changing one didn't affect the other.

THE SOLUTION

This meant they did not point to the same memory location, so therefore something was wrong with evaluate_condition() which caused both to be changed. After further debugging, I realised that it was not that currentVariableValue was the same property, it was that they both currently pointed to the same token in memory, so by changing one you change the other.

This is because currentVariableValue of the main loop is set by calling evaluate() on the Identifier i, and for the nested loop it is set to j being set to the evaluate() of i, and that being evaluated. Somehow these two point to the same memory location, which must be JavaScript attempting to optimise memory as much as possible. Therefore, a new Integer instance must be created for currentVariableValue.

```
let result = this.variable.evaluate()
if (!(result instanceof Integer)){
    return new TypeError(result, "Starting value is not an Integer")
}
this.currentVariableValue = new Integer(this.variable.position, this.variable.line, result.value)
```

My new solution is above, however I feel that the program will still not work as there is one more issue.

```
reset(){
    this.firstPassComplete = false
}
```

The nested loop's values must be reset after each iteration, otherwise when the second iteration of the main loop begins the variable will not be back to the starting value. This can be easily added by setting firstPassComplete to false, which I do in a method called reset()

This must be called at the end of evaluate_loop(), before 0 is returned by the method.

```
 ! ERROR @line 2
Evaluation Error: Step value must align with bounds of for loop
    for j = i to 3
              ^
```

When testing, the actual solution was almost returned, however the program returned an Error on the last input due to a bounds issue.

```
0
1
2
3
1
2
3
2
3
3
```

```
if (this.currentVariableValue.value <= this.finish.value && this.step.value > 0){
    this.increasing = true
} else if (this.currentVariableValue.value >= this.finish.value && this.step.value < 0){
    this.increasing = false
```

This just required the bounds check to be changed to allow the values to be equal.

Then, finally, the expected output was returned, which took a lot of attempts but now the new run system works as expected. Although a lot of this was due to issues not in run, the new Interpreter updated has also fixed a lot of other errors, including ones to do with If Statements, but the whole new system is a lot clearer and easier to understand and implement into for the future.

CHANGING CONDITIONS

code.erl

```
1   end = 3
2   for j = 0 to 1
3       for i = 0 to end
4           i
5       next i
6       end = 5
7   next j
```

```
0
1
2
3
0
1
2
3
```

Even though the bounds and step of the loop should not be able to be changed during the loop, after reset() is called any updated parameters should be affected. In this example, the loop should count to 5 on the second iteration, but instead it only counts to 3. This is because the property for end and step are completely replace by their evaluate() result, so their asts are destroyed so any changes are not updated. Therefore, we need separate properties for the asts and the properties when they are used.

```
0
1
2
3
0
1
2
3
4
5
```

```
this.variable = null
this.variableValue = null
this.assignment = null
this.finish = null
this.finishValue = null
this.step = null
this.stepValue = null
```

This needs new properties: finishValue and stepValue, and currentVariableValue has been renamed to variableValue for consistency. finish and step hold the ASTs, and finishValue and stepValue hold the results of their evaluate() methods, which were previously set to themselves. This means a lot of code in first_pass() and evaluate_condition() must be updated to reflect these changes, but it allows the Parser to remain the same.

Now, the correct output is produced due to these new additions.

OTHER CHANGES

Now a reset() method must be added to the other loops. For a Do loop, this is the same as For and it sets firstPassComplete to false. For While it just returns null.

```
reset = () => null
```

Also, because For does not use the condition attribute, I changed it from being in Loop to being defined seperately in While and Do because having it in For is bad practice.

For now, this wraps up the work on Loops, and I will do more in-depth testing during the Half-way evaluation to ensure it is all working as expected.

EVALUATION

Overall, the criteria for this section are fully completed, as shown:

No.	Criteria	Implemented?
5.1	Allow while loops to be created by "while", then a condition, then the contents of the loop on the next line, and closed by the "endwhile" keyword	**Yes**
5.2	Ensure that the contents will be evaluated before the loop is run, and it will iterate while the condition is True	**Yes**
5.3	Allow a do until loop to be created by "do", followed by the contents, and being closed by "until" then the condition of the loop	**Yes**
5.4	Ensure that the loop is run once before the condition is checked, and from there it iterates while the loop is False	**Yes**
5.5	Allow for loops to be created by "for", then an identifier, then "=", then a starting value or expression, then "to", followed by a final value or expression. Then, the contents, closed by "next", followed by the same identifier that the for loop was defined with	**Yes**
5.6	Ensure that, on each iteration, the variable with the identifier name is set to the new value, and that the loop will iterate until the value is greater than the final value, so it is inclusive	**Yes**
5.7	Allow an optional "step" keyword to follow the final value, which must be followed by a value or expression to represent how much the value will increment on each iteration. This can also be a negative value; in which case the loop iterates until the value is lower than the final value.	**Yes**
5.8	Ensure that all values in the for loop are integers, as well as full error handling for all invalid cases for conditions for the other loops, such as non-Booleans in conditions, with descriptive error messages.	**Yes**

I am very happy with the redesign of the Interpreter that took place, as that was my major concern whilst implementing the loops, as it was not a very complete solution before. Now I am happy that the whole system works and the solution is a lot more consistent.

Switching from the original idea that loops would be implemented recursively is also a lot better, as it likely uses a lot less memory and should make the program more efficient, because if I used recursion the memory usage would have been huge, as the memory for the previous iteration would not have been freed up until every .

The only issue is like If Statements, where certain non-Boolean conditions will not return errors when they should, however this will be implemented into the next section so is not a major concern. My only other longstanding issue is with the repetition in evaluate(), but hopefully that can also be reduced in the next section as I am thinking a bit of a redesign will be necessary.

In conclusion, I feel that Loops have been implemented very well.

STRINGS AND TYPES

DESIGN

This module will consist of 3 sections: Implementing strings and concatenation, adding type casting between different types, and stricter type handling.

TEXT STRING

Because the keyword String is reserved by JavaScript, the new class will be called TextString. This will extend Token with position, line, and value properties. It's display() method will just be to return its value.

➔ make_string()

When a quotation mark: " or ' is encountered in make_tokens_line(), this method will be called. It will repeatedly add characters to an array until the same type of quotation that it started with is encountered, stored in a variable called quotationMark. Then a new TextString instance will be returned with the array converted to a string and passed into the value property.

BINARY OPERATORS

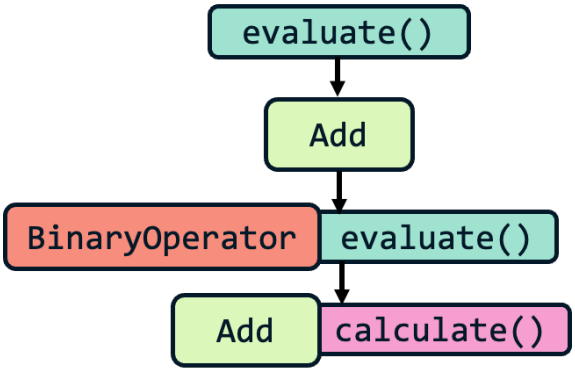
These have always contained a lot of repeated code, and this updated significantly reduces this. It also removes check_for_errors() and check_for_float(), as they become integrated within the new system.

➔ evaluate()

When calling evaluate() on a Binary Operator subclass, there will only be a single evaluate() method defined in Binary Operator, rather than a different one in each subclass.

This method will not return the value itself but will do the generic checks for every evaluate() method, and will evaluate the two children, returning an error if one occurs.

It will set the results of evaluating the children to two new properties: leftValue and rightValue and will then call the new calculate() method, which will use these values.

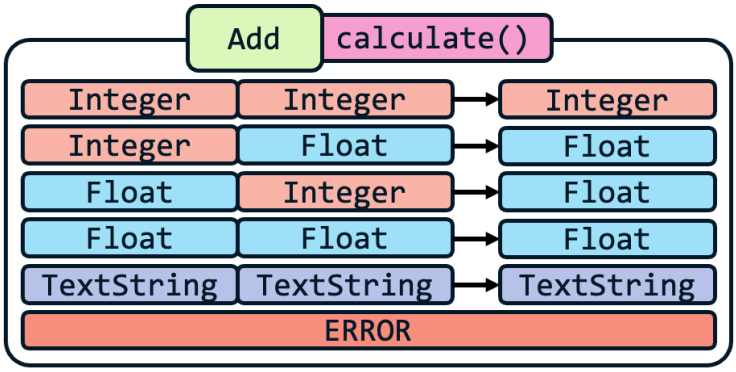


➔ calculate()

This will be a class specific method, so there will be no generic instance in Binary Operator, which will be called by evaluate() when leftValue and rightValue have been set. The purpose of this method is to go through the possible valid combinations of types, returning the corresponding type.

For example, the diagram shown on the right shows all the possible combinations of the leftValue and rightValue and what they should return.

calculate() will check which case the current values are and will return the corresponding type: this is replacing the old check_for_float() as this method will lay out the valid cases for each. If the type of combination is not valid, an error is returned.



Add is the most complex example for this because it can now be used for concatenation and arithmetic, however most calculate() methods should be relatively simple.

Any additional checks that need to occur, such as the leftValue not being zero, will also occur in calculate(). Finally, although this returns the types, to set the value of the types I will be using a method called get_value(), to not have to repeat the same calculation in each type constructor.

This will just return the JavaScript data type of the operation on leftValue and rightValue, with one for each operator, and will be called and passed as an argument to the constructors.

TYPE CHECKING METHODS

To make it easier to undergo all these checks, I am going to introduce the following methods which should make it easier to check each case with less comparisons, such as by grouping all of the cases that will return a Float into a single method that accounts for them all.

➔ strict_type_check()

This will take in two parameters and will check if leftValue is an instance of the first parameter, and if rightValue is an instance of the second. If and only if this is true, it will return true, otherwise false will be returned. This is used to check that the two types are the same as the parameters, in that order.

Using Add, this could be called with the arguments being two Integers, which would contain an Integer return because it is the only case that will return an Integer value.

➔ loose_type_check()

This accepts any number of arguments and ensures that the leftValue and rightValue are both instances of any of the arguments, and there is no limit on how many arguments can be passed. leftValue and rightValue can be any combination, so long as they are both at least one of the types passed.

➔ contains_type()

This also takes a dynamic number of arguments, but in this case if either LeftValue or RightValue are instances of any then true is returned: they do not both have to be, which is what separates it from loose_type_check().

The use case of these last two are checking for when a Float value is returned, so if contains_type(Float) and loose_type_check(Integer, Float) are both True, then a Float value can be returned, because there is at least one Float value and they can also be combined with Integers.

I feel that these 3 type checks are enough to cover all the cases that are required. I will not plan the checks for every single operator, but using these is a very logical approach and it will be easy to use the corresponding checks for each type.

TYPE ERROR

I did slightly introduce this at the end of the last stage of development whilst ensuring all the for-loop values are integers, but the main error type for this section is the TypeError, which accepts a token like SyntaxError. All of the errors returned in calculate() because of invalid types will be this new TypeError, and I aim to make the messages descriptive of the intended types.

Additionally, I will introduce new type checks into the different structures, ensuring that all the conditions for if statements and loops are Booleans, returning a type error if they aren't.

This also includes aspects like comparisons too, so if a greater than is used on two booleans, for example, I would like the program to return an error because Booleans are not "greater" then each other.

TYPE CASTING

`str()`	`str(345)`
`int()`	`int("3")`
`float()`	`float("4.52")`
`real()`	`real("4.52")`
`bool()`	`bool("True")`

The specification shows that there are multiple ways that data can be type casted: between Integers and Floats, between Strings and Integers/Floats, and between Strings that say "True" or "False" to Booleans. However, the specification is not exactly clear on what should be allowed, so I am going to create my own logical set of rules, which can be edited later.

RULES

My final table for how the different types will cast to each other is as shown:

Starting:	To Integer:	To Float:	To Boolean:	To String:
Integer		value is the same	0 ➔ false Else: true	value as String
Float	value is truncated		0.0 ➔ false Else: true	value as String
Boolean	true ➔ 1 false ➔ 0	true ➔ 1.0 false ➔ 0.0		true ➔ "True" false ➔ "False"
String	First check if possible - only digits Yes: value as Number No: TypeError	First check if possible - only digits and "." Yes: value as Number No: TypeError	"True" ➔ true "False" ➔ false Else: TypeError	

Initially, I was intending to now allow for type casting between numbers and booleans, because it could be potentially confusing to students as it is not very intuitive.

However, after discussing with my stakeholder Tanish, we concluded that it was important for children to understand the correlation between 0/1 to False/True, and therefore decided to keep that conversion. I could have also made it so any number that aren't 0 and 1 would cause an error. This could have maybe conveyed the idea that 1 represents True more clearly, but because most existing high-level language convert any non-zero number to True, I decided to keep with convention.

The one main difference between this and typical conversion is from String to Boolean. In typical languages, any non-empty string returned True, and only "" returned False. However, I thought a student attempting to cast `bool("False")` and receiving True would be extremely confusing.

```
bool("True")
```

Furthermore, the specification gives the example of `bool("True")` as a conversion, implying that it is the names of "True" and "False" that should be converted to Booleans, so I have made them do so and every other string will return an error. This is not standard, however the way I implement them should make the solution very easy to change later if I decide to.

➔ cast_to_type()

This method will belong to the data types and will take in an argument for the data type to convert to. It will then return the corresponding result, based on the instructions in the table. For now, this will only be accessible as a method by manual testing, as functions have not yet been implemented, but it should be easy to integrate in the future when they are implemented.

STRING CLASS

```
class TextString extends Token{
    constructor(position, line, value){
        super(position, line)
        this.value = value
    }

    display(){
        return this.value
    }
}
```

```
} else if (this.character == '"' || this.character == "'"){
    tokens.push(this.make_string())
} else if (this.character == '+') {
```

```
make_string(){
    let string = []
    let position = this.position
    let quotationMark = this.character
    this.continue()
    while (this.character != quotationMark){
        string.push(this.character)
        this.continue()
    }
    return new TextString(position, this.line, string.join(''))
}
```

This follows the plan and is very simple to add.

```
1414  console.log(new Lexer(['"hello"+"world!"']).make_tokens())
```

```
[
  TextString { position: 0, line: 0, value: 'hello' },
  Add { position: 7, line: 0, left: null, right: null },
  TextString { position: 8, line: 0, value: 'world!' }
]
```

The test case shows the Lexer correctly handles strings.

NEW BINARY OPERATOR CLASS

```
class BinaryOperator extends Token{
    constructor(position, line){
        super(position, line)
        this.left = null
        this.right = null
        this.leftValue = null
        this.rightValue = null
    }

    evaluate(){
        this.leftValue = this.left.evaluate()
        this.rightValue = this.right.evaluate()
        if (this.leftValue instanceof Error){
            return this.leftValue
        }
        if (this.rightValue instanceof Error){
            return this.rightValue
        }
        return this.calculate()
    }
```

These are replicas from the plan in code form. For now, there is not an easy way to test them, so I will first implement the calculate() method for the Add class, and then test that to ensure that it is working.

```
// Ensures the left data type is left argument, and right data type is right argument
strict_type_check(leftType, rightType){
    if (this.leftValue instanceof leftType && this.rightValue instanceof rightType){
        return true
    }
    return false
}

// Ensures both left and right are instances of one of the arguments each
loose_type_check(){
    let leftDone = false
    let rightDone = false
    for (let type of arguments){
        if (this.leftValue instanceof type){
            leftDone = true
        }
        if (this.rightValue instanceof type){
            rightDone = true
        }
    }
    return leftDone && rightDone
}

// Ensures either left or right are instances of one of the arguments
contains_type(){
    for (let type of arguments){
        if (this.leftValue instanceof type || this.rightValue instanceof type){
            return true
        }
    }
    return false
}
```

ADDITION

I have decided to go with the following calulate() method:

```
calculate(){
    if (this.contains_type(Float) && this.loose_type_check(Float, Integer)){ // Contains at least one float with potential integers
        return new Float(this.position, this.line, this.get_result()) // Definitely a float return
    } else if (this.strict_type_check(Integer, Integer)){ // Contains two integers
        return new Integer(this.position, this.line, this.get_result()) // Definitely an integer return for addition
    } else if (this.strict_type_check(TextString, TextString)){ // Contains two strings
        return new TextString(this.position, this.line, this.get_result()) // Concatenation
    }
}

get_result = () => this.leftValue.value + this.rightValue.value
```

This goes through all the possible cases: returning a Float and Integer and TextString. The 3 different type checking methods have all been used, and it is clear how they will be used for the others.

Because I deleted the check_for_errors() and check_for_floats() methods in this process, the program didn't run because they were called when not existing. After removing these from the other classes, I gave Float a new check for the display() method.

```
display(){
    if (String(this.value).includes('.')){
        return String(this.value)
    }
    return String(this.value) + ".0"
}
```

I also changed factor() in the Parser to allow TextString as a new factor, so that concatenation could therefore be parsed.

```
factor(self){
    if (self.check_instance(Integer, Float, Identifier, TextString)){
        let result = self.token
        self.continue()
        return result
    }
```

code.erl

```
2 + 2
2.7 + 3
"hello " + "world!"
```

```
4
5.7
hello world!

Exited with code 0
```

As shown, the 3 different test cases worked, however currently if an invalid combination occurs then the program will break.

NAMING UPDATE

```
class Symbol {
    constructor(token){
        this.name = token.name
        this._value = null
```

I have decided to rename the old DataType class to Symbol: not only does it make more sense as it is contained within the Symbol Table, it allows for the different actual DataType classes to all extend a new class called DataType.

```
318  ////////////////
319  // DATA TYPES
320  ////////////////
321
322  class DataType extends Token{
323      constructor(position, line, value){
324          super(position, line)
325          this.value = value
326      }
327
328      evaluate() {
329          return this
330      }
331  }
332
333  class Integer extends DataType{
334      constructor(position, line, value){
335          super(position, line, value)
336      }
```

This greatly reduces the amount of repeated code, and allows for easier checking throughout, such as in factor()

```
factor(self){
    if (self.check_instance(DataType)){
        let result = self.token
```

I also added lots of titles, as shown on the left, throughout the source code, as it is getting very large and this makes it easier to manage.

This change to DataType will also allow half_statement() and potentially other methods to be simpler in the future: there will now be no Data Type checking in the Parser due to how Identifiers cause issues, so now all checking will occur in the calculate() methods. This means that Booleans can now be expressions, however if they are used like a number then the error will be thrown in calculate() rather than in the Parser, simplifying the statement code.

INCOMPATABLE DATA TYPES

```
type_to_string = () => "Integer"
```

Firstly, I have given each DataType a method type_to_string(), which returns the string format of their type to be called in an Error.

Then, as the else case of calculate(), I have created different cases for the different initial Data Types. This could all be the same error message, but I have decided to specialise them to provide better error messages. In these cases the left value is treated as the "correct" data type.

```
    } // ERRORS
    if (this.leftValue instanceof TextString){ // String + ?
        return new TypeError(this, `Cannot concatenate type ${this.rightValue.type_to_string()} with String, expected String`)
    }
    if (this.leftValue instanceof Integer || this.rightValue instanceof Float){ // Number + ?
        return new TypeError(this, `Cannot add type ${this.rightValue.type_to_string()} to ${this.leftValue.type_to_string()}, expected Integer or Float`)
    } // ELSE
    return new TypeError(this, `Cannot combine types ${this.rightValue.type_to_string()} and ${this.leftValue.type_to_string()}`)
}
```

```
! ERROR @line 1
Type Error: Cannot add type Boolean to Integer, expected Integer or Float
2 + True
  ^
```

```
! ERROR @line 1
Type Error: Cannot concatenate type Integer with String, expected String
"Hello" + 7
        ^

Exited with code 1
```

Some examples of these error messages are shown, and they are very clear as to what the issue is, which is important when it is Identifiers that are involved in these errors and their data type is not clear. The else case is also very important, as there must be a catch in calculate(), to return an error if two unaccounted datatypes are merged.

OTHER OPERATORS

Now the new calculate() and get_result() methods must be written for all other Binary Operators. The rest of the arithmetic operators are very simple: with less error messages and cases than addition due to concatenation not being a thing. The best example of this is the Minus class:

```
class Minus extends BinaryOperator{
    constructor(position, line){
        super(position, line)
    }

    calculate(){
        if (this.contains_type(Float) && this.loose_type_check(Float, Integer)){ // Contains at least one float with potential integers
            return new Float(this.position, this.line, this.get_result()) // Definitely a float return
        }
        if (this.strict_type_check(Integer, Integer)){ // Contains two integers
            return new Integer(this.position, this.line, this.get_result()) // Definitely an integer return for addition
        } // ERRORS
        return new TypeError(this, `Cannot subtract type ${this.rightValue.type_to_string()} from ${this.leftValue.type_to_string()}, expected Integers or Floats`)
    }

    get_result = () => this.leftValue.value - this.rightValue.value
}
```

The most complex arithmetic example is Divide, requiring the check that the right-hand side is not zero, and because two Integers can produce a Float result, an extra check is needed in the Integer, Integer strict check to ensure a Float is returned if it is a float value by checking for full stops.

```
class Divide extends BinaryOperator{
    constructor(position, line){
        super(position, line)
    }

    calculate(){
        if (this.leftValue.value === 0){
            return new EvaluationError(this.leftValue, "Cannot divide by zero")
        }
        if (this.contains_type(Float) && this.loose_type_check(Float, Integer)){ // Contains at least one float with potential integers
            new Float(this.position, this.line, this.get_result())  // Definitely a float return
        }
        if (this.strict_type_check(Integer, Integer)){ // Contains two integers
            let result = this.get_result() // Must check as can produce a float result
            return String(result).includes('.') ? new Float(this.position, this.line, result) : new Integer(this.position, this.line, result)
        } // ERRORS
        return new TypeError(this, `Cannot divide type ${this.rightValue.type_to_string()} by ${this.leftValue.type_to_string()}, expected Integers or Floats`)
    }

    get_result = () => this.leftValue.value / this.rightValue.value
}
```

The exponent class also requires a check for NaN to ensure no imaginary numbers are created, but apart from that the other Arithmetic operators are very logical.

Equals is the only Binary Operator to completely redefine evaluate(), because it must call assign() on the leftValue instead of the default evaluate(). This leads to a slight bit of repeated code; however, it is acceptable in this context.

Next is the logical operators, which have the simplest checks: requiring that both sides are Booleans, therefore using the strict check. The code for And is on the next page, but Or uses the exact same code, with an updated error message and different get_result() method.

```
class Equals extends BinaryOperator{
    constructor(position, line){
        super(position, line)
    }

    evaluate(){
        this.leftValue = this.left.assign()
        this.rightValue = this.right.evaluate()
        if (this.leftValue instanceof Error){
            return this.leftValue
        }
        if (this.rightValue instanceof Error){
            return this.rightValue
        }
        return this.leftValue.set(this.rightValue)
    }
}
```

```
class And extends BinaryOperator{
    constructor(position, line){
        super(position, line)
    }

    calculate(){
        if (this.strict_type_check(Boolean, Boolean)){ // Contains two Booleans
            return new Boolean(this.position, this.line, this.get_result()) // Expected result
        } // ERRORS
        return new TypeError(this, `Cannot use AND on type ${this.rightValue.type_to_string()} with ${this.leftValue.type_to_string()}, expected Booleans`)
    }

    get_result = () => this.leftValue.value && this.rightValue.value
}
```

Finally, there is the Logical Operator check. The get_result() method is very useful here, as it makes the code a lot easier to understand. I have decided to allow Integers or Floats to be compared in any combination, as well as just two strings, but not combinations of numbers and strings.

For Booleans, I have decided to only allow == and != to be compared. Technically > and < can be used in other languages, however it is a useless feature so I have decided to remove this ability.

```
class LogicalOperator extends BinaryOperator{
    constructor(position, line, tag){
        super(position, line)
        this.tag = tag
    }

    calculate(){
        if (this.loose_type_check(Integer, Float) || this.strict_type_check(TextString, TextString)){ // Contains two Booleans
            return new Boolean(this.position, this.line, this.get_result()) // Expected result
        }
        if (this.strict_type_check(Boolean, Boolean)){
            if (this.tag == "==" || this.tag == "!="){
                return new Boolean(this.position, this.line, this.get_result())
            }
            return new TypeError(this, `Cannot compare two Booleans with comparator '${this.tag}', only '==' or '!='`)
        }
        // ERRORS
        return new TypeError(this, `Cannot compare type ${this.rightValue.type_to_string()} against ${this.leftValue.type_to_string()}`)
    }

    get_result(){
        switch (this.tag){
            case "==":
                return this.leftValue.value == this.rightValue.value
            case ">":
                return this.leftValue.value > this.rightValue.value
            case ">=":
                return this.leftValue.value >= this.rightValue.value
            case "<":
                return this.leftValue.value < this.rightValue.value
            case "<=":
                return this.leftValue.value <= this.rightValue.value
            case "!=":
                return this.leftValue.value != this.rightValue.value
        }
    }
}
```

I ran a few quick test cases for each operator, not testing all the errors but ensuring that the expected results are working as expected, and they seem to be. I will test with more depth later in the process.

STRUCTURE DATA TYPE CHEKING

```
code.erl
1    if 2 + 2 then
2        4
3    endif
```

Now If Statement and the Loops need type checking to ensure that their conditions are Booleans. Currently, the program crashes if there is no correct case in an IfStatement, but this was updated by adding a null check in evaluate_many_asts(), because IfStatement will return null if there is no valid case.

```
/Users/pw/ocr-erl-interpreter/src.js:1478
        for (let ast of asts){
                        ^

TypeError: asts is not iterable
```

This fixed the crash; however, we want the Interpreter to return an error if a condition is not a Boolean value.

```
evaluate_many_asts(asts){
    if (asts == null){
        return 0
    }
```

The code for this is as shown:

```
evaluate(){
    for (let ifCase of this.cases){
        let result = ifCase.condition.evaluate()
        if (result instanceof Error){
            return result
        }
        if (!(result instanceof Boolean)){
            return new TypeError(result, `Condition must be type Boolean, not ${result.type_to_string()}`)
        }
```

```
evaluate_single_ast(ast){
    if (ast instanceof IfStatement){
        let result = ast.evaluate()
        if (result instanceof Error){
            console.log(result.display())
            return 1
        }
        return this.evaluate_many_asts(result)
    }
```

This is an easy check in IfStatment's evaluate() method, and if the result is not a Boolean datatype, then a Type Error will be returned.

The Interpreter also must be updated, because currently it could not deal with errors being returned from IfStatement's evaluate(), and the simple check added now deals with that.

While and Do are also given the same check in their evaluate_condition() methods.

```
if (!(condition instanceof Boolean)){
    return new TypeError(condition, `Condition must be type Boolean, not ${condition.type_to_string()}`)
}
```

```
! ERROR @line 1
Type Error: Condition must be type Boolean, not Integer
if 2 + 2 then
   ^
```

```
! ERROR @line 1
Type Error: Condition must be type Boolean, not Integer
while 7
      ^
```

The correct error is returned.

That concludes all the Type handling in this program.

CLEANING UP

I am not a fan of the name TextString, and it turns out that the Boolean class I have been using is also a reserved keyword by JavaScript, so if I wanted to use any of its methods, I would not be able to. Therefore, I am renaming the datatypes to IntegerType, FoatType, BooleanType and StringType. This provides consistency between them all and is easy to change with the replace feature in my IDE.

```
355 > class IntegerType extends DataType{…
365   }
366
367 > class FloatType extends DataType{…
380   }
381
382 > class BooleanType extends DataType{…
392   }
393
394 > class StringType extends DataType{…
404   }
```

By removing the Boolean check in half_statement, which was already flawed due to potential Identifiers with a Boolean data type, the code is significantly simpler, as well as the error message being much more representative of the error that has occurred.

```
! ERROR @line 1
Invalid Syntax: Expected operator
True + 5
     ^
```

```
! ERROR @line 1
Type Error: Cannot combine types Boolean and Integer
True + 5
     ^
```

TYPE CASTING

```
cast_to_type(type){ // FROM INTEGERS
    switch (type){
        case IntegerType:
            return this
        case FloatType:
            return new FloatType(this.position, this.line, this.value)
        case BooleanType: // true when not 0, false when 0
            return new BooleanType(this.position, this.line, this.value != 0)
        case StringType:
            return new StringType(this.position, this.line, this.display())
    }
}
```

```
cast_to_type(type){ // FROM BOOLEAN
    switch (type){
        case IntegerType:
        case FloatType: // 1 when true, 0 when false
            return new type(this.position, this.line, this.value ? 1 : 0)
        case BooleanType:
            return this
        case StringType:
            return new StringType(this.position, this.line, this.display())
    }
}
```

```
cast_to_type(type){ // FROM FLOAT
    switch (type){
        case IntegerType:
            return new IntegerType(this.position, this.line, Math.floor(this.value))
        case FloatType:
            return this
        case BooleanType: // true when not 0, false when 0
            return new BooleanType(this.position, this.line, this.value != 0)
        case StringType:
            return new StringType(this.position, this.line, this.display())
    }
}
```

These are relatively easy to add, just copying what the table in the design stage was explaining to do.

Not fully sure if they work yet, will need to do a lot of test cases afterwards.

```
cast_to_type(type){ // FROM STRING
    switch (type){
        case IntegerType: // Can be converted to a number, no full stops
            if (isNaN(Number(this.value)) || this.value.includes('.')){
                return new TypeError(this, `Cannot cast ${this.value} to type Integer`)
            }
            return new IntegerType(this.position, this.line, Number(this.value))
        case FloatType: // Can be converted to a number
            if (isNaN(Number(this.value))){
                return new TypeError(this, `Cannot cast ${this.value} to type Float`)
            }
            return new FloatType(this.position, this.line, Number(this.value))
        case BooleanType: // "True" or "False" are accepted, rest are errors
            switch (this.value){
                case "True":
                    return new BooleanType(this.position, this.line, true)
                case "False":
                    return new BooleanType(this.position, this.line, false)
                default:
                    return new TypeError(this, `Cannot cast ${this.value} to type Float, expected "True" or "False"`)
            }
        case StringType:
            return this
    }
}
```

The one for strings is the longest as there are a lot more cases to account for.

TESTING

I wrote a quick algorithm to iterate through a 2D array, which, for each item, would create a new data type from the class at index 0, with the value at index 1, would typecast it to index 2, and the expected result would be at index 3. Then, I made it print pass or fail if the expected result was produced.

```
// FORMAT = original type, original value, typecasted to what, expected
for (let item of testCases){
    let result = new item[0](0, 0, item[1]).cast_to_type(item[2])
    if (item[3] == Error){
        if (result instanceof Error){
            console.log("PASSED")
        } else {
            console.log(`FAIL`)
        }
    } else if (result.value == item[3]){
        console.log("PASSED")
    } else {
        console.log(`FAIL`)
    }
}
```

```
testCases = [
    [IntegerType, 5, IntegerType, 5],
    [IntegerType, 5, FloatType, 5],
    [IntegerType, 5, BooleanType, true],
    [IntegerType, 0, BooleanType, false],
    [IntegerType, 5, StringType, "5"],
    [FloatType, 2.3, IntegerType, 2],
    [FloatType, 4, IntegerType, 4],
    [FloatType, 7.01, IntegerType, 7],
    [FloatType, 9.8, IntegerType, 9],
    [FloatType, 2.3, FloatType, 2.3],
    [FloatType, 2.3, BooleanType, true],
    [FloatType, 0.0, BooleanType, false],
    [FloatType, 2.3, StringType, "2.3"],
    [FloatType, 1.0, StringType, "1.0"],
    [BooleanType, true, IntegerType, 1],
    [BooleanType, false, IntegerType, 0],
    [BooleanType, true, FloatType, 1],
    [BooleanType, false, FloatType, 0],
    [BooleanType, true, BooleanType, true],
    [BooleanType, true, StringType, "True"],
    [BooleanType, false, StringType, 'False'],
    [StringType, "2", IntegerType, 2],
    [StringType, "1000", IntegerType, 1000],
    [StringType, "hello", IntegerType, Error],
    [StringType, "2.75", IntegerType, Error],
    [StringType, "2", FloatType, 2],
    [StringType, "10.5", FloatType, 10.5],
    [StringType, "hello", FloatType, Error],
    [StringType, "2.75", FloatType, 2.75],
    [StringType, "True", BooleanType, true],
    [StringType, "False", BooleanType, false],
    [StringType, "hello", BooleanType, Error],
    [StringType, "hello", StringType, "hello"],
]
```

```
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
```

The output showed that the program printed PASSED for every single test case, and I tried to include every single possible combination in my array of input data, so therefore I am confident that the implementation has worked.

This therefore concludes this section on strings and typecasting, and overall I feel a lot more confident with the program now that the types are stricter and that a lot of the repeated code has been eliminated.

EVALUATION

The success criteria for this section are as follows:

No.	Criteria	Implemented
6.1	Add a string data type, which can be defined with single or double quotations, and concatenated with +	**Yes**
6.2	Ensure that the whole system has tighter type handling, as now that operators have different uses across different types, different cases will be required so that invalid cases are rejected	**Yes**
6.3	Allow for type casting between the 4 primitive datatypes, with errors in some illegal cases	**Yes**

I am very happy with this module, as all the new features have been implemented correctly, but also most of the pre-existing main issues with the program have been fixed: lots of repeated evaluate() code has been removed, issues with conflicting data types have been replaced with checks and errors, and a lot of bad naming has been changed to make the system a lot more clear.

This is a good place to be finishing this half of development with. The main issue currently is still the global variables not being encapsulated, mainly the global symbol table, as well as the currentText system for error messages, but these should be easy to change later in development.

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.

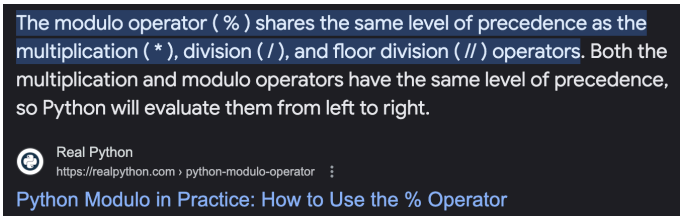
DESIGN

MODULUS AND QUOTIENT

These will be new classes extending Binary Operator, and they will be created in make_identifier() by the keywords "MOD" and "DIV".

MOD	Modulus
DIV	Quotient

Python 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

The 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()

6	< , <=, >, >=, ==, !=	Less than, less than or equal, greater, greater or equal, equal, not equal
7	not	Boolean Not
8	and	Boolean And
9	or	Boolean Or

➔ 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.

MODULUS AND QUOTIENT

```
class Modulus extends BinaryOperator{
    constructor(position, line){
        super(position, line)
    }

    calculate(){
        if (this.rightValue.value === 0){
            return new EvaluationError(this.leftValue, "Cannot take modulu by zero")
        }
        if (this.contains_type(FloatType) && this.loose_type_check(FloatType, IntegerType)){ // Contains at least one float with potential integers
            return new FloatType(this.position, this.line, this.get_result()) // Definitely a float return
        }
        if (this.strict_type_check(IntegerType, IntegerType)){ // Contains two integers
            return new IntegerType(this.position, this.line, this.get_result()) // Definitely an integer return
        } // ERRORS
        return new TypeError(this, `Cannot take modulus of type ${this.leftValue.type_to_string()} modulo ${this.rightValue.type_to_string()}, expected Integers or Floats`)
    }

    get_result = () => this.leftValue.value % this.rightValue.value
}

class Quotient extends BinaryOperator{
    constructor(position, line){
        super(position, line)
    }

    calculate(){
        if (this.rightValue.value === 0){
            return new EvaluationError(this.leftValue, "Cannot do quotient division by zero")
        }
        if (this.contains_type(FloatType) && this.loose_type_check(FloatType, IntegerType)){ // Contains at least one float with potential integers
            return new FloatType(this.position, this.line, this.get_result()) // Definitely a float return
        }
        if (this.strict_type_check(IntegerType, IntegerType)){ // Contains two integers
            return new IntegerType(this.position, this.line, this.get_result()) // Definitely an integer return
        } // ERRORS
        return new TypeError(this, `Cannot take the quotient of type ${this.leftValue.type_to_string()} and ${this.rightValue.type_to_string()}, expected Integers or Floats`)
    }

    get_result = () => Math.floor(this.leftValue.value / this.rightValue.value)
}
```

The 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.

```
switch (name) {
    case "MOD":
        return new Modulus(position, this.line)
    case "DIV":
        return new Quotient(position, this.line)
```

```
term(self){
    return self.parse_binary_operator(self, self.exponent, [Multiply, Divide, Modulus, Quotient])
}
```

From here the other changes are very easy to make.

code.erl

```
5 MOD 2
31 MOD 17
9 DIV 2
11 DIV 5
3.7 MOD 0.6
11.2 DIV 3
10 DIV 0
```

```
1
14
4
2
0.10000000000000031
3.0
 ! ERROR @line 7
Evaluation Error: Cannot do quotient division by zero
10 DIV 0
^

Exited with code 1
```

As 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.

```
class Not extends Token{
    constructor(position, line){
        super(position, line)
        this.child = null
    }

    evaluate(){
        childValue = this.child.evaluate()
        if (childValue instanceof Error){
            return childValue
        }
        if (childValue instanceof Boolean){
            return new Boolean(this.position, this.line, !this.childValue.value)
        }
        return new TypeError(this, `Cannot use NOT on type ${this.childValue.type_to_string()}, expected Boolean`)
    }
}
```

The 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.

```
not_statement(self){
    if (self.token instanceof Not){
        let notToken = self.token
        self.continue()
        if (self.token == null){
            return new SyntaxError(self.previous, "Incomplete input")
        }
        let result = self.not_statement()
        if (result instanceof Error){
            return result
        }
        notToken.child = result
        return notToken
    }
    return self.statement(self)
}

statement_chain(self){
    return self.parse_binary_operator(self, self.not_statement, [And, Or])
}
```

```
case "NOT":
    return new Not(position, this.line)
```

The addition to the Lexer was an extremely easy change.

The 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.

code.erl

```
1  NOT True
2  NOT False
3  NOT True == True
4  NOT False AND True
5  NOT True OR NOT True
```

```
[Running] node src.js "/Users/pw/ocr-erl-interpreter/code.erl"
/Users/pw/ocr-erl-interpreter/src.js:1218
        if (self.token instanceof Not){
                ^

TypeError: Cannot read properties of undefined (reading 'token')
```

```
evaluate(){
    let childValue = this.child.evaluate()
    if (childValue instanceof Error){
        return childValue
    }
    if (childValue instanceof BooleanType){
        return new BooleanType(this.position, this.line, !childValue.value)
    }
    return new TypeError(this, `Cannot use NOT on type ${childValue.type_to_string()}, expected Boolean`)
}
```

```
let result = self.not_statement(self)
```

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.

```
False
True
False
True
False
```

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

The 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.

```
////////////////
// LOGICAL
////////////////

class And extends BinaryOperator{…
}

class Or extends BinaryOperator{…
}

class Not extends Token{…
}

class ComparisonOperator extends BinaryOperator{…
}
```

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.

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

writeTests.py > ...

```python
with open("testCases.txt", 'a') as f:
    f.write("\n$")
    next = input()
    while next != "EXIT":
        if next == "-":
            f.write("\n$")
        else:
            f.write("\n"+next)
        next = input()
    f.close()
```

I 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.

```javascript
function main(){
    let testCases = fs.readFileSync("testCases.txt", 'utf8').split("\n$\n")
    let interpeter = new Interpreter()
    let success = 0
    for (let i = 0; i < testCases.length; i+= 2) {
        output = []
        interpeter.set_plaintext_manually(testCases[i])
        interpeter.run()
        if (output.join("\n") == testCases[i+1]){
            success++
        } else {
            console.log(`FAILURE\n${testCases[i]}\nGOT ${output}\nEXPECTED ${testCases[i+1].split("\n")}\n\n`)
        }
        global.table = []
    }
    console.log(`Success: ${success}/${testCases.length/2} = ${success/testCases.length*200}%`)
}
```

My 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.

```javascript
display(){
    return "error"
}
```

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.

```
for i = 1 to 10
if i MOD 2 == 0 AND i MOD 3 == 0 then
"FizzBuzz"
elseif i MOD 2 == 0 then
"Fizz"
elseif i MOD 3 == 0 then
"Buzz"
else
"None"
endif
next i
-
None
Fizz
Buzz
Fizz
None
FizzBuzz
None
Fizz
Buzz
Fizz
```

```
prime = True
num = 57
for i = 1 to num DIV 2
if num MOD i == 0 then
prime = False
endif
next i
prime
-
False
EXIT
```

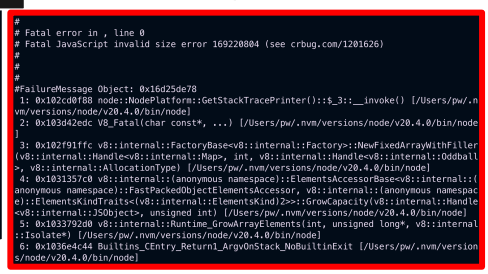
```
python3 writeTest
a = 7
b = a
b = b + 1
a
b
-
7
8
EXIT
```

```
~/ocr-erl-
python3 writeTest
for i = -3 to 3
12 / i
next i
-
-4
-6
-12
error
EXIT
```

```
-
triangle = 0
for i = 1 to 10
triangle = triangle + i
next i
triangle
-
55
```

```
2 ^ 3
-
8
-
for i = 0 to 10
i
next i
-
0
1
2
3
4
5
6
7
8
9
10
EXIT
```

Because 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.

```
"hello
```

It 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.

```
Success: 94/102 = 92.15686274509804%
```

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

```
class LexicalError extends Error {
    constructor(position, line, description='') {
        super(position, line)
        this.description = description
    }

    display(){
        return ` ! ERROR @line ${this.line}\nLexical Error: ${this.description}\n${this.location()}`
    }
}
```

```
} else { // Not recognised
    return new LexicalError(this.position, this.line, `Unexpected character '${this.character}'`)
}
```

Firstly, 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.

```
make_string(){
    let string = []
    let position = this.position
    let quotationMark = this.character
    this.continue()
    while (this.character != quotationMark && this.character != null){
        string.push(this.character)
        this.continue()
    }
    return this.character == null ? new LexicalError(position, this.line, "Unclosed string") :new StringType(position, this.line, string.join(''))
}
```

```
let string = this.make_string()
if (string instanceof Error){
    return string
}
tokens.push(string)
```

```
! ERROR @line 1
Lexical Error: Unclosed string
"test
^
```

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

```
FAILURE
4 ^ 0.5
GOT error
EXPECTED 2.0


FAILURE
6.0 / 3
GOT error
EXPECTED 2.0
```

```
FAILURE
a = 28
do
a
a = a / 2
until a <= 2
GOT 28,14,7,3.5,error
EXPECTED 28,14,7,3.5
```

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:

```
if (this.contains_type(FloatType) && this.loose_type_check(FloatType, IntegerType)){
    return new FloatType(this.position, this.line, this.get_result())  // Definitely
}
```

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

FOR LOOP ISSUES

```
FAILURE
for const a = 0 to 3
a
next a
GOT 0,1,2,3
EXPECTED error
```

```
FAILURE
for i = 0 to 10000
next i
i
GOT 10001
EXPECTED 10000
```

The 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.

```
if (self.token instanceof TemplateKeyword){
    return new SyntaxError(self.token, "Expected identifier")
}
let result = self.assignment(self) //assignment
```

The 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.

```
evaluate_condition(){
    let newValue
    if (this.firstPassComplete){ //increasing value
        newValue = this.variableValue.value + this.stepValue.value
    } else {
        let result = this.first_pass()
        if (result instanceof Error){
            return result
        }
        newValue = this.variableValue.value
    }
    let condition = this.increasing ? newValue <= this.finishValue.value : newValue >= this.finishValue.value
    if (condition){ // only change identifier if another loop will occur
        this.variableValue.value = newValue
        this.variable.assign().set(this.variableValue)
    }
    return condition
}
```

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

```
FAILURE
((((((7)))+3)))
GOT error
EXPECTED 10


FAILURE
()
GOT error
EXPECTED


FAILURE
(3)*(7)-(4)
GOT error
EXPECTED 17
```

The 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.

```
if (bracket instanceof start){
    self.continue()
    if (self.token instanceof end){
        return new SyntaxError(bracket, "'()' was empty")
    }
```

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.

```
! ERROR @line 1
Invalid Syntax: Expected operator
(3*7)-4
   ^

Exited with code 1
```

Therefore, half_statement () will need to be changed to account for this.

```
expression     → assignment ;

assignment     → ( call "." )? IDENTIFIER "=" assignment
               | logic_or ;

logic_or       → logic_and ( "or" logic_and )* ;
logic_and      → equality ( "and" equality )* ;
equality       → comparison ( ( "!=" | "==" ) comparison )* ;
comparison     → term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
term           → factor ( ( "-" | "+" ) factor )* ;
factor         → unary ( ( "/" | "*" ) unary )* ;

unary          → ( "!" | "-" ) unary | call ;
call           → primary ( "(" arguments? ")" | "." IDENTIFIER )* ;
primary        → "true" | "false" | "nil" | "this"
               | NUMBER | STRING | IDENTIFIER | "(" expression ")"
               | "super" "." IDENTIFIER ;
```

For 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.

```
statement(self){
    if (this.check_binary_operator()){ // starts with binary operator
        return new SyntaxError(self.token, "Expected literal")
    }
    let left = self.expression(self) //left hand side
    if (left instanceof Error || !(self.token instanceof ComparisonOperator)){
        return left // returns if error or next is not comparison
    }
    let result = self.token // middle
    self.continue()
    if (self.token == null){ // nothing after comparison
        return new SyntaxError(result, "Incomplete input")
    }
    if (this.check_binary_operator()){ // starts with binary operator
        return new SyntaxError(self.token, "Expected literal")
    }
    let right = self.expression(self) //right hand side
    if (right instanceof Error){
        return right
    }
    result.left = left
    result.right = right
    return result // returns comparison
}
```

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>
```

Instead 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.

```
parse_brackets(self, end, nextFunction){
    let bracket = self.token
    self.continue()
    if (self.token instanceof end){
        return new SyntaxError(bracket, "'()' was empty")
    }
    let result = nextFunction(self)
    if (result instanceof Error){
        return result
    }
    if (self.token instanceof end){
        self.continue()
        return result
    }
    return new SyntaxError(bracket, "'(' was never closed")
}
```

```
if (self.token instanceof LeftBracket){
    return self.parse_brackets(self, RightBracket, self.statement_chain)
}
```

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

```
Success: 103/103 = 100%
```

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.

FUNCTIONS AND PROCEDURES

DESIGN

RESEARCH

Excluding this section and arithmetic, I've gone into all the other modules with a vague idea of how I'd like to implement the solution, and completely invented by own approach. However, I am not completely sure how I am going to approach this module as I feel it is the most complex, so am going to do some research as to how some other interpreters deal with functions, and then adapt their approach so it can merge with how I have already built up by Interpreter.

The site I am going to use, which I also used when initially researching translator design is:
https://craftinginterpreters.com/functions.html

I am not copying any of the code, as the website is intended as a tutorial for creating a language, but I will only be looking at their approach and ideas, rather than looking at any code snippets.

CALLING FUNCTIONS

From what I took away, it is best to think as the function as the parenthesis () that follow it, rather than the identifier itself. These brackets represent a function call, calling whatever is in front of them and become an instance of it. In this way they almost act like a postfix operator.

Because it is like a postfix operator, it will be parsed at the end of factor(), so therefore lots of type checking will have to be included so that only subroutine can be called. However, the parser itself will not do this, so if ever brackets are following a factor, a call will be created.

This focus on the brackets led me to realise the approach I want to take with this. The rest of the website's description did not fit how I have already designed my Interpreter, so from here the rest of the ideas are completely my own planning.

THE CALL CLASS AND HOW TO PARSE IT

At the end of each factor() call, instead of returning the value, the method will check if the next token is a LeftBracket. If it is, then it will call the new call() method to construct this Call.

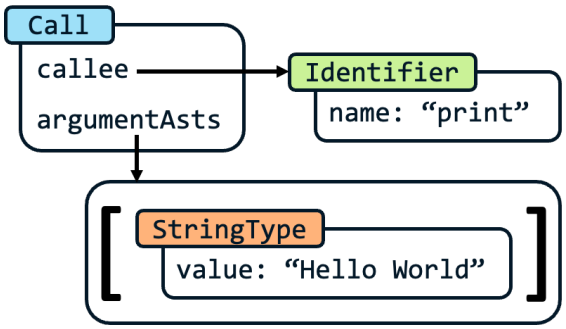
This will then create a new instance of the new Call class, with the bracket's position and line being passed into the constructor method. This call class will have several properties:

➔ callee: an Identifier that represents the subroutine the Call is calling, set to null by default
➔ scope: pointing to its symbol table (more detail in the Identifier section)
➔ argumentsAsts: an array that will contain the arguments in ast form, empty by default.

The array cannot just be named arguments because it is a reserved JavaScript keyword.

The diagram on the right is a good representation of this structure, for the input `print("Hello World")`

It is clear how the Identifier is stored in the callee, with the name of the function name being the Identifier name.



PARSING THE ARGUMENTS

The call() method in the parser will then manage the arguments for the call: i.e. what is contained within the brackets and will fill the argumentsAsts array with the different arguments that have been passed.

There are three different cases that the call() method needs to be able to expect.

➔ No arguments: it must first check If the next token is a closed bracket, if it is it will return the call immediately with the array of arguments staying empty.

Example: input() ➔ Any subroutine that is passed no arguments

➔ One argument: it will call statement_chain(), which will be the call to parse each actual argument and push the result to argumentsAsts after checking for errors. If the following token is a closed bracket, then the Call will be returned with a single item.

Example: print("Hello world") ➔ Any subroutine that is passed one argument

➔ Multiple arguments: from here, because multiple arguments must be separated by commas, it will check for a comma before calling statement_chain() for each successive argument. When there are no more commas, it will ensure that the next token is a closed bracket, then return the call instance.

Example: random(1, 6) ➔ Any subroutine that accepts two or more arguments

Therefore, the BNF for <call> can be represented as shown:

bnf.txt — Edited

```
<call> := '(' ')'
        | '(' <statement_chain> ')'
        | '(' <statement_chain> { ',' <statement_chain> } ')'
```

There is a more condensed way to write this but separating it out into the 3 possible options makes it clearer and will make implementing the code easier.

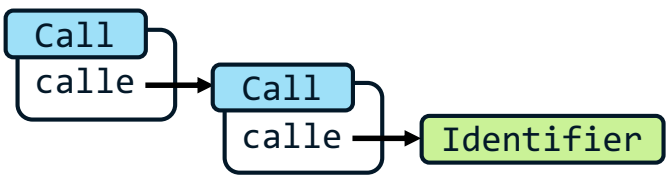
So, call() must be able to distinguish between these types and be able to correctly assign argumentsAsts with these different syntax trees. Lots of error handling will have to be added to counter for all invalid cases, but I will implement this in development. This includes cases such as a comma not being followed by a statement chain, where it does not fall into any of the categories.

RETURNING THE VALUE TO FACTOR()

After receiving the Call instance from call(), factor() will then set the calee property to what it previously parsed in factor(): For example, if an identifier was at the start of factor(), it will now become the callee property of the Call method, which can then be returned.

However, before returning the Call, it will once again check if there is another call by checking for a left bracket. If there is, the process will be repeated, and the old Call will become the callee of the new call.

This allows functions to return functions, which can then be called, and they will be structured as shown in the abstract syntax tree shown on the left.



Finally, when no more instances of call are detected, then factor() will return this chain of calls, or just a single factor if no subroutine was present.

Therefore any <factor> can become a child of the call in the syntax tree, so additional checking will have to occur so that the callee is an Identifier, and that the value the Identifier holds is a Subroutine, and not a different datatype, as Call is only intended for Subroutine.

THE SUBROUTINE CLASS

This class is representative of a custom, user made subroutines. The token itself will be created by either the procedure or function keyword, which will be given a tag of its name to represent its type.

```
procedure name(…)

endprocedure
```

```
function name(…)
    …
    return …
endfunction
```

Additionally, TemplateKeywords for endprocedure and endfunction will be added to make_identifier()

The Subroutine class will have additional properties that extend token:

➔ parameters: an array of identifiers which will be assigned values when the function is called
➔ contents: the asts contained inside the function which will be run on evaluation
➔ scope: will point to the scope currently calling it at runtime (will expand on later)

This will be parsed like a lot of other structures: after the first declarative line, the current line's ast will be continually added to the contents until the closing keyword is detected.

In this case, it will be checking for a template keyword with the tag of "end" + the tag of the subroutine, allowing it to work for both functions and procedures. This is because the same parsing method will be used for both types as they share the same overall structure.

This will be called build_subroutine(), for consistency with the other structures.

PARSING THE DECLARATIVE LINE

The identifier after the keyword will be stored, so that it's value can later be set to the function.

Then the call() method will be reused to parse the brackets containing the parameters: as they will be laid out in the same format as arguments. However, the Call object that will be returned is not used, and instead the parameters array is set to the argumentsAsts array of the Call object, after ensuring that every item is an Identifier and not an ast, because call() can allow asts to be included.

This is not ideal, however writing another, almost identical method would involve a lot of repetition for the sole purpose of parsing parameters, so this is a better solution.

From here, the typical code will be run, to fill up the contents ast with all the asts. However, these arrays need to be regulated, ensuring that no return statements occur in procedures.

THE RETURN TOKEN

Created by "return" in make_identifier(), it will have a single property called child to represent what code follows it. When parse() finds an instance of Return, it will call the return() method, which has two cases: the return statement being followed by nothing, or by a statement chain, as shown by the BNF:

If nothing follows return, the child property will be set to null to represent this.

However, "return" statements are only permitted when functions are being created, therefore a new property for Parser called allowReturn will be created, set to false as default.

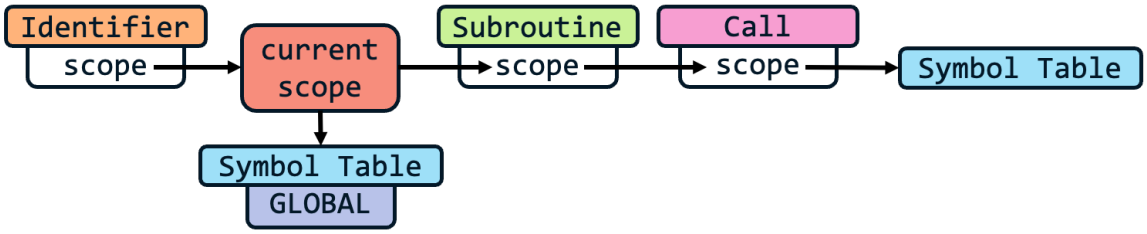
Whenever a return statement is encountered, the program will check allowReturn. If it is false, an error will be returned because return is now permitted. allowReturn will only be set to true when a function is being created: so when make_subroutine() is called with the tag being "function" and not "procedure". This will therefore allow return() to parse, and it will be set back to false at the end of parsing.

IDENTIFIER CHANGES

Now that different scopes are going to be required, the Identifier system will have to be changed. So far, only one instance of the symbol table class has been used, but now with functions this must change.

My initial idea was to give each function its own symbol table: however, this would not allow any recursion to occur, which would be an issue. Instead, each instance of the Call class, represented by (), will have its own symbol table stored in a property called scope.

Overall, the new diagram on the right showcases the general structure of how I'd like to implement the new system.



➔ Identifiers

These will now have a scope property, which they will call find() on in their evaluate() and assign() methods instead of always using the global symbol table. This scope will be assigned during parsing, and will be set to the current value of a new currentScope property of the parser.

This will point to the global symbol table by default, however, whenever a subroutine is being parsed in build_subroutine(), then it will be changed to point to the scope property of the subroutine, so any identifiers will then be parsed to use the symbol table of the subroutine.

➔ Local scopes

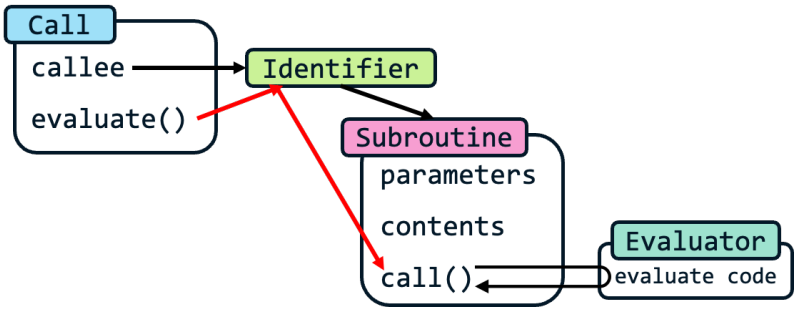
However, the scope property in Subroutine will not be used, and will be a placeholder until the function is called. Then, the subroutine scope property will be set to the scope property of the call instance, which should hopefully then set all the Identifier scope to also point to the call instance.

I am not completely sure if this will work, however I have had a lot of issues in the past of this project with properties pointing to objects and not copying them but referencing their location. This has been annoying previously, but I am hoping to use it here to my advantage to create a chain of references.

EVALUATING SUBROUTINES

We have discussed the structure of the new features, now they need to be evaluated.

This begins with the call instance, which will be the root of the structure, so has an evaluate() method to execute the subroutine.



Call's evaluate method will then call evaluate() on its callee property, which is an Identifier, which will return the Subroutine instance that it corresponds to.

Now, the Subroutine's new call() method will be called, and the instance of Call will be passed as an argument.

From here, this new call() method will deal with executing the function: the Call being parsed so that its argumentsAsts and scope properties can be used by it.

At the end, call() will return the value that the Subroutine returns, which then Call's evaluate() method will also return, so that function calls link into the rest of the program.

THE CALL METHOD

As explained, this method will accept the Call instances as a parameter so it can access its properties. Overall, there are a few steps that need to take place before the contents are run:

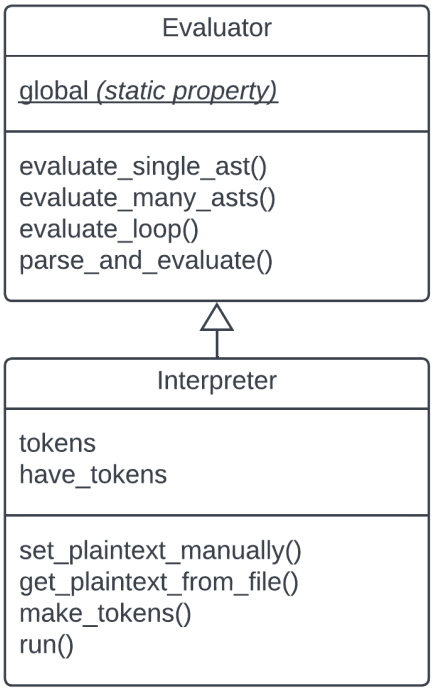
➔ **Checking for arity**
This means that the number of parameters must be equal to the number of arguments, which can be checked by ensuring that the length of the two arrays is equal. If they aren't, an error is returned.

➔ **Setting parameter values**
The argumentsAsts array will be iterated through, evaluated, and then the corresponding Identifier with the same position in the parameters arrays will have its value set to the result. Note that these parameters scope's must be set to the scope of the call instance so that they are accessible when run.

➔ **Changing scope**
The functions scope must be changed to point to the Call's scope, so that all the identifiers in the contents will now use the new scope when run, and the scope must then be changed back to the old scope to ensure that recursion can occur in the system.

THE EVALUATOR CLASS

The Evaluator class will be the new parent class of Interpreter, and a lot of the most basic functions will be given to it: evaluate_single_ast(), evaluate_many_asts() and evaluate_loop(). However, the tokens relevant to parsing and keeping track of inputs and plaintext will be kept in the Interpreter class, as they are not required.



Now, to run the subroutine's code, a new instance of an Evaluator will be created, and it will be passed the contents array to run. From here, the code will be run and if any errors occur, they will instead be returned into the Subroutine method, instead of being displayed.

This therefore means that no error messages will be outputted in these messages, and that 1s and 0s will no longer be returned, which was a system that was inconsistent anyway and caused some issues.

Now when an error occurs in one of the Evaluator methods, it will just be returned, and this chain of returning errors will continue until it reaches the original run() method in the Interpreter. This will now be the single place that error messages are outputted and will make the system overall a lot clearer and easier to follow.

The other change is for any return statements: in evaluate_single_ast(), any Return tokens will be returned, in the same way that errors are, so that as soon as one is encountered, further evaluation will stop and the return method will reach the Subroutine class. This is not an issue for anywhere else in the program, because the parser only allows return statements to be present within functions.

➔ parse_and_evaluate()

This is the method that will be called whenever the contents of a function needs to be evaluated, and the subroutine's call() method will call this on the new Evaluator, passing the contents array.

It is essentially the run() method for the Evaluator, excluding all the Lexing stages because the tokens have already been generated. Apart from this, it is very self-explanatory: parsing each line and then evaluating it, returning any errors that may occur.

This should be very simple to implement: just moving some lines from run() into this new method.

CONCLUDING EVALUATION

The result of the Evaluator evaluating the contents is stored in a temporary variable. If it is an instance of Error, it is returned. If the return value was null, or it is an instance of Return with the child set to null, then null is returned by call(). Otherwise, the child of Return is evaluated, and that is the value that is returned.

This will therefore be returned to the call's evaluate() method and returned from there, so that any values from functions can be integrated within other expressions.

FINAL SCOPE CHANGES

Because I do not like having random global variables in the project, I am going to move the global scope to be a class (static) property of Evaluator, as it seems like a logical place to keep it. Therefore, it will be accessed by `Evaluator.global` and will have to be changed in the program.

ACESSING GLOBAL VARIABLES IN SUBROUTINES

Subroutines should still be able to access global variables when being run, so therefore the find() method needs to be changed for symbol tables.

After find() is called, if no Symbol with the corresponding name is found in the current table, instead of immediately creating a new symbol, it will first check, so long as the symbol table itself is not the global symbol table, through the different symbols in the global table. If an equivalent symbol is present there, it will return it to be accessed. Otherwise, the new symbol will be created in the original scope.

GLOBAL KEYWORD

A new template keyword will be created if the keyword "global" is encountered. In parse(), if this keyword is encountered it will call assignment(). Then, in the same style as the const keyword, if a global keyword is present in assignment then it will change the new global property of an identifier to be true.

When calling assign() on the Identifier, regardless of the scope property it has, it will look itself up in the global symbol table, which is a small change but should achieve the effect.

OTHER SMALL CHANGES

In order to eliminate any classes which are redundant, I am going to remove the LeftBracket and RightBracket classes and instead use template keywords with '(' and ')' tags instead. This is because, as more punctuation will be needed as the program progresses, such as commas, colons, and square brackets, it would create a lot of redundant classes.

One option was to create a Punctuation class; however, it would have been identical to the TemplateKeyword class so, despite the name not being very descriptive, I will use it for punctuation too.

Therefore, make_tokens_line() will be updated to check if '(', ')' or ',' are the current character, and if they are it will create the corresponding template keyword.

➔ check_tag()

The program already has lots of instances where tags of template keywords must be checked, and it always results in a lot of double-nested if statements, to first check if the token is a TemplateKeyword, and then check the tag, as you cannot just check for the tag as other tokens do not have that property.

Instead, a new, simple check_tag() method will 1 or more arguments and will check that the current token is a Template Keyword, and if its tag is one of the arguments. It will then return true or false accordingly, which will significantly simplify the existing code and make it a lot more readable.

CHANGING BRACKETS TO BE TAGGED

```
} else if (['(', ')',',',].includes(this.character)){
    tokens.push(new TemplateKeyword(this.position, this.line, this.character))
```

This is very easy to change in the Lexer.

```
check_tag(){
    if (!(this.token instanceof TemplateKeyword)){
        return false
    }
    for (let item of arguments){
        if (this.token.tag == item){
            return true
        }
    }
    return false
}
```

In the long term it is a positive decision, because these classes never contained any code, and as more brackets and punctuation will be added, it is best to use the generic class that exists.

The check_tag() method was also added, and now the rest of the existing code base must be updated to use this change. Overall, I feel it makes it a lot more readable, as it avoids a lot of double nesting for tag checks.

```
// Check if there is a constant assignment
if (this.check_tag("const")){
    let result = this.assignment(this)
        return this.check_result(result) ? result : new SyntaxError(this.token, "Expected operator")
}
// Check if elseif or endif is used not at end of line
if (this.check_tag("endif", "elseif")){
    return new SyntaxError(this.token, "Needs to follow 'if' statement")
}
```

```
    } else if (self.check_tag("endif")){ // Check for endif
        mainStatement.cases.push(currentIfStatement) // Pushes old if case
        this.continue()
        return mainStatement // complete
    } else {
        let result = self.parse() // Default
```

```
} else {
    if (!self.check_tag("step")){ // ensures next token is step
        return new SyntaxError(self.token, "Expected 'step' or end of line")
```

```
    if (self.check_tag('(')){
        return self.parse_brackets(self, ')', self.statement_chain)
    }
```

```
parse_brackets(self, end, nextFunction){
    let bracket = self.token
    self.continue()
    if (self.check_tag(end)){
        return new SyntaxError(bracket, "'()' was empty")
    }
    let result = nextFunction(self)
    if (result instanceof Error){
        return result
    }
    if (self.check_tag(end)){
        self.continue()
        return result
    }
    return new SyntaxError(bracket, "'(' was never closed")
}
```

Some examples of these changes are shown:

Only one of these changes affected how the code operates: with parse_brackets() now being passed the tag for the ending bracket as a string rather than a class, so no longer checks for the instance.

Just to ensure the changes didn't break anything, I ran the testing program with the new source code and the success rate was still 100%.

```
Success: 103/103 = 100%
```

CALL CLASS

```
class Call extends Token{
    constructor(position, line){
        super(position, line)
        this.callee = null
        this.scope = new SymbolTable()
        this.argumentsAsts = []
    }
}
```

The new call system requires the original code of factor() to be changed so that it no longer returns a value, but instead sets the result to the result variable, which will then be used at the end. However, errors still need to be returned, as shown in the code on the next page.

The new call method(), which is called by factor() is there is a LeftBracket present has cases for any number of arguments.

Therefore, If it is an empty call (), or containing one lone argument, or containing multiple separated by commas then these are all pushed to the argumentsAsts array. There is currently some error handling implemented here, but I will come back and refine this later to ensure that the error messages are descriptive, as they may not currently be very specific to parsing arguments: instead of returning expected comma or argument it likely prints "expected literal" in its current state.

```
call(self){
    let call = new Call(self.token.position, self.token.line)
    self.continue()
    if (self.check_tag(')')){
        return call
    }
    let argument = self.statement_chain(self)
    if (argument instanceof Error){
        return argument
    }
    call.argumentsAsts.push(argument)
    while (self.check_tag(',')){
        self.continue()
        argument = self.statement_chain(self)
        if (argument instanceof Error){
            return argument
        }
        call.argumentsAsts.push(argument)
    }
    if (self.check_tag(')')){
        self.continue()
        return call
    }
    return new SyntaxError(self.token, "Expected ')' or ',' followed b
}
```

```
factor(self){
    let result
    if (self.check_instance(DataType, Identifier)){
        result = self.token
        self.continue()
    } else if (self.check_tag('(')){
        result = self.parse_brackets(self, ')', self.statement_chain)
    } else if (self.check_instance(Add)){ // Add unary operator
        self.continue()
        if (self.token == null){
            return new SyntaxError(self.previous, "Incomplete input")
        }
        result = self.factor(self)
    } else if (self.check_instance(Minus)){ // Minus unary operator
        self.continue()
        if (self.token == null){
            return new SyntaxError(self.previous, "Incomplete input")
        }
        result = self.factor(self)
        if (result instanceof Error){
            return result
        }
        result.value = -result.value
    } else { // Two operators in a row
        return new SyntaxError(self.token, "Expected literal")
    }
    while (self.check_tag('(')){
        let call = self.call(self)
        if (call instanceof Error){
            return call
        }
        call.callee = result
        result = call
    }
    return result
}
```

```
Call {
  position: 19,
  line: 0,
  callee: Call {
    position: 10,
    line: 0,
    callee: Identifier {
      position: 0,
      line: 0,
      name: 'myFunction',
      constant: false
    },
    scope: SymbolTable { table: [] },
    argumentsAsts: [ [StringType] ]
  },
  scope: SymbolTable { table: [] },
  argumentsAsts: [
    Add {
      position: 22,
      line: 0,
      left: [IntegerType],
      right: [IntegerType],
      leftValue: null,
      rightValue: null,
      get_result: [Function: get_result]
    },
    Not { position: 27, line: 0, child: [BooleanType] },
    Identifier { position: 37, line: 0, name: 'var', constant:
  ]
}
SyntaxError {
  position: 15,
  line: 2,
  text: 'procedure(2, 7,)',
  token: TemplateKeyword { position: 15, line: 1, tag: ')' },
  description: 'Expected literal'
}
```

The test case I used shows all these features: how calls can be chained one after each other, how they can accept any number of parameters and parse and store them correctly.

code.erl

```
1  myFunction("hello")(1 + 1, NOT True, var)
```

```
call(){
    return new EvaluationError(self, `Type ${this.type_to_string()} cannot be called`)
}
```

EVALUATOR

```
class Evaluator {
    evaluate_loop(loop){
        let condition = loop.evaluate_condition()
        if (condition instanceof Error){
            return condition
        }
```

Now the new Evaluator class is created, and the Interpreter class is set to extend it. The methods evaluate_loop(), evaluate_single_ast() and evaluate_many_asts() now belong to it. These methods have all been changed to no longer pass 1s and 0s to represent any error messages, but instead return the error without printing it.

This change was simple, requiring lots of small changes to the code as shown above.

```
parse_and_evaluate(tokens){
    let parser = new Parser(tokens)
    let ast = parser.parse_next()
    while (ast != null){
        let result = this.evaluate_single_ast(ast)
        if (result instanceof Error){
            return result
        }
        ast = parser.parse_next()
    }
}
```

Now the new parse_and_evaluate() method is added to the Evaluator class, which takes is essentially the second half of the old run() method. It accepts an array of tokens, and will parse them, returning any errors that occur.

run(), which is in Interpreter is also changed to now call parse_and_evalaute() with the contents of this.tokens. It will check if the result of this is an error. If it is, it will output the display() method and return 0 or 1 accordingly.

```
run(){
    if (!this.have_tokens){
        let result = this.make_tokens()
        if (result instanceof Error){
            console.log(result.display())
            return 1
        }
    }
    let result = this.parse_and_evaluate(this.tokens)
    if (result instanceof Error){ // All error messages printed here
        console.log(result.display())
        return 1
    }
    return 0
}
```

Therefore, every single error message, excluding any Lexical errors, will be outputted with this single output message, including any errors in instances of functions, as they will be returned from any instances of Evaluator, back to the main instance of Interpreter where they will be passed down to this statement.

This makes the program a lot easier to handle because it provides a lot more consistency,

BASIC SUBROUTINES

The new subroutine class is as shown.

```
class Subroutine extends Token{
    constructor(position, line){
        super(position, line)
        this.scope = null
        this.parameters = []
        this.contents = []
    }

    call(call){
        this.scope = call.scope
        if (this.parameters.length != call.argumentsAsts.length){
            return new EvaluationError(call, `Subroutine expected ${this.parameters.length} arguments, ${call.argumentsAsts.length} given`)
        }
        for (let i = 0; i < this.parameters.length; i++){
            let result = call.argumentsAsts[i].evaluate()
            if (result instanceof Error){
                return result
            }
            this.parameters[i].assign().set(result)
        }
        let result = new Evaluator().parse_and_evaluate(this.contents)
        if (result instanceof Error){
            return result
        }
    }
}
```

As a summary, the scope property of subroutine is just a placeholder that all the variables included in its contents will point to. When the subroutine is called, this scope will be changes to the scope of the call, and therefore all the variables will now point to this new scope.

```
    this.scope = null
}

evaluate(){
    return this.scope.find(this).value
}

assign(){
    return this.scope.find(this)
}
```

Identifiers new changes to match this scope now are implemented too, and instead of storing the global symbol table in an instance of the Parser, I have decided to change the plan and store it as a class property of the Evaluator class, as it seems to be a more generalised and widely accessible place for it to be accessed.

```
class Evaluator {
    static global = new SymbolTable()
```

```
factor(self){
    let result
    if (self.token instanceof DataType){
        result = self.token
        self.continue()
    } else if (self.token instanceof Identifier){
        self.token.scope = self.currentScope
        result = self.token
        self.continue()
```

```
this.currentScope = Evaluator.global // points to current scope of variables
```

Therefore, every time an Identifier is parsed, its scope must be changed to be the current scope of the program. This can easily be done in factor(). By default, this scope will be set to the global Symbol Table, however later when parsing functions this will be changed.

```
let procedure = new Subroutine()
procedure.parameters = [new Identifier(0,0,"num1"), new Identifier(0,0,"num2")]
let funcParser = new Parser(new Lexer("num1 + num2").make_tokens())
funcParser.scope = procedure.scope
procedure.contents = [funcParser.parse_next()]
let id = new Identifier(1, 1, "func")
id.scope = Evaluator.global
id.assign().set(procedure)
let main = new Interpreter()
main.set_plaintext_manually("func(1,2)")
console.log(main.run())
```

```
evaluate(){
    return this.callee.evaluate().call(this)
}
```

Now Call must be changed so that its evaluate() method will call the call() method on its callee, and pass itself as the call to be used for the scope.

Now, hopefully this idea should work, so I wrote a test case that is shown on the left to check.

This is very handmade and a lot of additional parsing will need to be added to construct these functions, but this is a test to make sure the system as a whole works.

DEBUGGING

```
/Users/pw/ocr-erl-interpreter/src.js:498
        return this.scope.find(this)
                          ^

TypeError: Cannot read properties of null (reading 'find')
```

Unfortunately, my plan has not worked as expected, and the variables have not pointed to the SymbolTable and they remain as null. This could be fixed in a very complex way, but I have a better idea now.

Since adding the global scope as a class property to Evaluator, I have realised that I could create another class property to reflect the current scope, which would be changed each time that a subroutine was called, and reverted to global after it has completed.

```
class Evaluator {
    static global = new SymbolTable()
    static currentScope = Evaluator.global
```

```
call(call){
    let previousScope = Evaluator.currentScope
    Evaluator.currentScope = new SymbolTable
    if (this.parameters.length != call.argumentsAsts.length){
        return new EvaluationError(call, `Subroutine expected ${this.parameters.length} arguments, ${call.argumentsAsts.length} given`)
    }
    for (let i = 0; i < this.parameters.length; i++){
        let result = call.argumentsAsts[i].evaluate()
        if (result instanceof Error){
            return result
        }
        this.parameters[i].assign().set(result)
    }
    let result = new Evaluator().parse_and_evaluate(this.contents)
    if (result instanceof Error){
        return result
    }
    Evaluator.currentScope = previousScope
}
```

```
class Identifier extends Token{
    constructor(position, line, name, constant=false){
        super(position, line)
        this.name = name
        this.constant = constant
    }

    evaluate(){
        return Evaluator.currentScope.find(this).value
    }

    assign(){
        return Evaluator.currentScope.find(this)
    }
}
```

These changes must be reflected in Identifier as well, and all the redundant scope properties can now be removed due to this new system. This is a better system than the previous, as if the same call is repeated multiple times in a loop, then a new Symbol Table will be created each time, and deleting the used symbol tables means that the memory usage of the program will be lower.

```
[Runn
0
```

However, this did not output anything and just the exit code of 0 was outputted.
This was still the case after removing some of the syntax errors, and turned out to be a flaw with the Evaluator system.

CHANGING EVALUATION

```
run(){
    if (!this.have_tokens){
        let result = this.make_tokens()
        if (result instanceof Error){
            console.log(result.display())
            return 1
        }
    }
    let parser = new Parser(this.tokens)
    let ast = parser.parse_next()
    while (ast != null){
        let result = this.evaluate_single_ast(ast)
        if (result instanceof Error){
            console.log(result.display()) // all er
            return 1
        }
        ast = parser.parse_next()
    }
    return 0
}
```

The issue was that parse_and_evaluate() tried to call parse_next() on the contents array. Because the contents are already parsed, this is redundant and instead evaluate_many_asts() can be called on the

```
let result = new Evaluator().evaluate_many_asts(this.contents)
```

contents from a new Evaluator() instance.

Therefore, run() needs to be reverted to its original state, but adapted to receive any error messages from evaluate_single_ast() and to output them.

```
[Ru
3
0
```

Now, the correct value of three is outputted, so therefore the system has worked.

Finally, I changed the call() method in Subroutine to accept the arguments rather than the Call instance because it no longer needs the scope.

```
evaluate(){
    return this.callee.evaluate().call(this.argumentsAsts)
}
```

From here, this now allows functions to be run, now they need to be parsed.

PARSING SUBROUTINES

The code for implementing the parsing of functions is very similar to a lot of the other structures.

```
build_subroutine(self){
    let subroutineToken = self.token // defining token
    self.continue()
    if (!self.token instanceof Identifier){ // function name
        return new SyntaxError(self.token, "Expected identifier for procedure")
    }
    subroutineToken.identifier = self.token
    self.continue()
    if (self.token == null || !self.check_tag("(")){
        return new SyntaxError(self.token == null ? subroutineToken : self.token, "Expected
    }
    let call = self.call(self) // checking parameters
    if (call instanceof Error){
        return call
    }
    subroutineToken.parameters = call.argumentsAsts
    if (self.token != null){
        return new SyntaxError(self.token, "Expected no tokens following subroutine definition")
    }
    self.advance_line()
    while (self.currentTokens != null){ // adding contents
        self.continue()
        if (self.check_tag("endprocedure")){ // closing clause
            this.continue()
            return subroutineToken
        }
        let result = self.parse()
        if (result instanceof Error){
            return result
        }
        subroutineToken.contents.push(result)
        self.advance_line()
    }
    return new SyntaxError(subroutineToken, "Expected 'endprocedure' to close procedure")
}
```

```
case "procedure":
    return new Subroutine(position, this.line)
```

The keyword is also checked for in make_identifier()

```
// Checks for a subroutine definition
if (this.token instanceof Subroutine){
    return this.build_subroutine(this)
}
```

In parse() the check for the token calls the method.

The method itself uses lots of already existing ideas: the loop for current Tokens, and it uses call() to parse the parameters.

```
evaluate(){
    this.identifier.assign().set(this)
}
```

Finally, the subroutine token is changed to assign the function to the identifier when it is evaluated.

code.erl

```
procedure sum(a, b)
    a + b
endprocedure

sum(87,2)
sum(1,2)
sum("hello", " world")
```

```
89
3
hello world

Exited with code 0
```

The only issue with the original build_subroutine() were some syntax errors, but after correction any valid input correctly outputted the right response.

Currently, build_subroutine() cannot deal with a lot of invalid cases; however, I will error test this later in the process after all the features have been added.

FUNCTIONS VS PROCEDURE

```
class Subroutine extends Token{
    constructor(position, line, tag){
        super(position, line)
```

The tag class is added to subroutines to distinguish between the two types and make_identifier() is updated to reflect the changes.

```
case "procedure":
case "function":
    return new Subroutine(position, this.line, name)
case "return":
    return new Return(position, this.line)
```

The return class is also added, with the Parser's new allow Return property set to return on default. The parser only allows return statements to be parsed if this is true, and otherwise returns an error.

```
class Return extends Token {
    constructor(position, line){
        super(position, line)
        this.child = null
    }

    evaluate(){
        return this.child.evaluate()
    }
}
```

```
this.allowReturn = false // Depends if a function is currently being parsed
```

```
// Check if it is a return statement
if (this.token instanceof Return) {
    if (this.allowReturn){
        return this.return(this)
    }
    return new SyntaxError(this.token, "Can only use 'return' within functions")
}
```

This will then call the new return() method to parse these statements.

```
return(self){
    let returnToken = self.token
    self.continue()
    if (self.token == null){
        return returnToken
    }
    let result = self.statement_chain(self)
    if (result instanceof Error){
        return result
    }
    returnToken.child = result
    return returnToken
}
```

This is overall very simple, allowing for empty return statements, and otherwise calling statement_chain() and returning that.

build_subroutine() also needs small changes: adjusting the check for the closing tag to be based on if the tag is function or procedure, and changing allowReturn to true if the tag is function, and setting allowReturn to false when it is complete.

The changes to build_subroutine() are shown here:

```
build_subroutine(self){
    self.allowReturn = self.token.tag == "function"
    let subroutineToken = self.token // defining token
```

```
if (self.check_tag("end" + subroutineToken.tag)){
    this.continue()
    self.allowReturn = false
    return subroutineToken
}
```

EVALUATION

```
evaluate_single_ast(ast){
    if (ast instanceof Return){
        return ast
    }
```

evaluate_single_ast() is changed so that it returns a Return statement when encountered. Therefore, now not just errors are passed between the different evaluate() methods in Evaluator, but so are Return methods, so the error checks between methods are replace by ensuring the result is not null, and each Evaluator method now returns null on default.

```
let result = this.evaluate_single_ast(ast)
if (result != null){
    return result
}
```

```
let returnValue = null
if (result instanceof Return){
    returnValue = result.child.evaluate()
}
Evaluator.currentScope = previousScope
return returnValue
```

The Subroutine class now calls the evaluate() method on its child if it receives one and will otherwise return null.

```
code.erl
1  function sum(a, b)
2      return a + b
3  endfunction
4
5  sum(2, 3) * 2
```

```
Run
10
```

Now, functions and procedures can be correctly parsed, and the return value of a function can be parsed with other features,

```
code.erl
1  procedure testing(value)
2      return value
3  endprocedure
4
5  testing(5)
```

The system ensuring that procedures cannot have returns is also correct.

```
! ERROR @line 2
Invalid Syntax: Can only use 'return' within functions
    return value
    ^
```

Again, this has very minimal error handling.

PROPER VARIABLE SCOPES

This includes adding the global keyword to assignments and ensuring that the scope of variables is correct: so that global variable's values can be accessed in subroutines but not set without global.

GLOBAL SETTING

```
class Identifier extends Token{
    constructor(position, line, name){
        super(position, line)
        this.name = name
        this.constant = false
        this.global = false
    }

    evaluate(){
        return (this.global ? Evaluator.global : Evaluator.currentScope).find(this).value
    }

    assign(){
        return (this.global ? Evaluator.global : Evaluator.currentScope).find(this)
    }
}
```

```
// Check if there is a tagged variable assignment
if (this.check_tag("const", "global")){
    let result = this.assignment(this)
    return this.check_result(result) ? result : new Syntax
}
```

First, I added a new case in the make_identifier() method for the "global" keyword.

Identifier will have a boolean property for global, which will determine whether it uses the currentScope table or the global symbol table.

```
assignment(self){
    let tag = null
    if (self.check_tag("const", "global")){
        tag = self.token.tag
        self.continue()
    }
    let variable = self.token
    if (!(variable instanceof Identifier)){
        return new SyntaxError(variable, "Expected identifier")
    }
    variable.constant = tag == "const"
    variable.global = tag == "global"
    self.continue()
```

The assignment() property now checks for the global tag as well and will set the corresponding properties of the Identifier according to this.

This also improved on the old system, as previously an entirely new Identifier would be created to make the parser a constant, and a boolean was passed in the constructor. Now this has been removed, and constant is not set through the arguments but the property is changed in the method.

```
code.erl
1  procedure testScope()
2      global globalVar = "i am global"
3      localVar = "i am local"
4  endprocedure
5
6  testScope()
7  globalVar
8  localVar
```

```
i am global
! ERROR @line 8
Identifier Error: 'localVar' was not declared
localVar
^

Exited with code 1
```

The test case on the left shows that global variables can be set and accessed outside, but local variables will not affect the rest of the program. However, global variables can still not be read inside of functions: which should be allowed.

```
code.erl
1  const five = 5
2  function addFive(num)
3      return num + five
4  endfunction
5
6  addFive(6)
```

```
class SymbolTable {
    constructor(){
        this.table = []
    }

    get(token){
        for (let item of this.table){ // Checks if in current table
            if (item.name == token.name){
                return item.value
            }
        }
        if (this != Evaluator.global){ // If current table isnt global, checks global
            for (let item of Evaluator.global.table){
                if (item.name == token.name){
                    return item.value
                }
            }
        }
        return new IdentifierError(token, "was not declared") // Not defined
    }

    set(token, newValue){
        for (let item of this.table){ // Checks own table onlky
            if (item.name == token.name){
                item.lastAssignment = token
                return item.set(newValue)
            }
        }
        this.table.push(new Symbol(token)) // If doesn't exist, will create
        return this.table[this.table.length - 1].set(token, newValue)
    }
}
```

Overall, the old system was implemented a long time ago and was not very clean. This includes some of the syntax, such as when an Identifier was assigned a value, then `Identifier.assign().set(value)` would have to be called, which has poor readability.

Now, this new implementation is much easier to the understand, with the Symbol Table having separate get() and set() methods.

The get() method will search the current table, and global (if it isn't global) for the identifier and will return its value. If it is not there, then it has not been declared, which removes the getter method being needed in Symbol.

Setting is still overall very similar, with some slight naming changes.

```
class Symbol {
    constructor(token){
        this.name = token.name
        this.value = null
        this.constant = token.constant // Is a constant
        this.declared = false // If it is a constant, has it been declared
    }

    set(token, newValue){
        if (token.constant){ // Check if is now a constant
            this.constant = true
            this.declared = false
        }
        if (!this.constant){ // Normal variable
            this.value = newValue
            return null
        }
        if (this.declared){ // Constant and already defined
            return new IdentifierError(token, "is a constant and has already been defined")
        }
        this.declared = true // Defining a cor
        this.value = newValue
        return null
    }
}
```

This also requires changes to Identifier to align with this new system, as well as all of the setting methods, including Equals, which now have much cleaner messages, as shown in the example in For:

```
if (condition){ // only change identifier
    this.variableValue.value = newValue
    this.variable.set(this.variableValue)
}
```

The test cases for the addFive function now works as intended:

```
[Run
11
```

More complex examples also work with more advanced scopes.

```
var = 0
procedure test()
    var
    var = 1
    var
endprocedure

test()
var
```

```
0
1
0
```

```
class Identifier extends Token{
    constructor(position, line, name){
        super(position, line)
        this.name = name
        this.constant = false
        this.global = false
    }

    evaluate(){
        return Evaluator.currentScope.get(this)
    }

    set(newValue){
        return (this.global ? Evaluator.global : Evaluator.currentScope).set(this, newValue)
    }
}
```

RECURSION HANDLING

code.erl
```
procedure recursion(a)
    a
    recursion(a+1)
endprocedure

recursion(1)
```

My main worry was that, after a certain number of recursions, JavaScript would exceed the call stack size and would crash, so I decided to run a test case to see how many calls this would take, however it ended up throwing an error that one of the parameters was not defined.

```
1
 ! ERROR @line 3
Identifier Error: 'a' was not declared
    name (a+1)
         ^

Exited with code 1
```

After some debugging, I found that this was due to the change in scope. This is because the arguments need to be evaluated in the old scope, but the parameters need to be set in the new scope.

```
let argumentsValues = []
for (let i = 0; i < argumentsAsts.length; i++){ // First get argument values in old scope
    let result = argumentsAsts[i].evaluate()
    if (result instanceof Error){
        return result
    }
    argumentsValues.push(result)
}
let previousScope = Evaluator.currentScope // Then change
Evaluator.currentScope = new SymbolTable()
for (let i = 0; i < argumentsValues.length; i++){ // Then sets parmaters in new scope
    this.parameters[i].set(argumentsValues[i])
}
```

This meant the arguments could not be evaluated one by one and assigned to each parameter, and now an intermediate array called argumentsValues stores the results of evaluating the arguments before the scope changes, and then these are set to each of the parameters after.

```
1599
node:internal/console/constructor:308
          if (isStackOverflowError(e))
              ^
```

This specific example shows that the limit for this recursion is at 1599 calls, and my program requires a limit that is lower than this to ensure the program never crashes.

code.erl
```
procedure change()
    global a = 7
endprocedure

a = 3
a
change()
a
```

As an additional note, I spent an extremely long time trying to debug the program on the left, as somehow my global implementation had broken and the value of a would be undefined after change() was called.

This turned out to be that one of the set() calls in SymbolTable had the arguments incorrectly mapped, which cause it to break.

```
if (item.name == token.name){
    item.lastAssignment = token
    return item.set(token, newValue)
}
```

THE FIBONACCI PROBLEM

code.erl
```
function fib(n)
    if n <= 2 then
        return 1
    endif
    return fib(n-1) + fib(n-2)
endfunction

fib(4)
```

I then wanted to try recursion where the same function called itself twice, and a Fibonacci function was the perfect test. However, this did not work at all: the program did not crash but a completely incorrect value was returned, and this resulted in a large amount of debugging with lots of outputting of the symbol tables, calls and so on.

It seemed to be that, even though the value of the lower fib() functions were calculated properly, they would be incorrectly passed to the upper fib() functions so give the illusion of being wrong.

My initial thought was that it was the same issue that I have encountered a lot of times: that setting a variable to another would cause them to both point to the same thing, so if one was changed it caused both to be changed. In the past I solved this by creating entirely new instances

```
duplicate(){
    return new this.constructor(this.position, this.line, this.value)
}
```

```
for (let i = 0; i < argumentsValues.length; i++){ // Then s
    this.parameters[i].set(argumentsValues[i].duplicate())
}
```

My solution was giving the DataType class a duplicate() method, which would return a completely new instance with identical properties so it would not be pointed to be set to the values of parameters.

```
[Running] node src.js "
CALLED n=2 RETURNING 1
CALLED n=1 RETURNING 1
CALLED n=3 RETURNING 2
CALLED n=2 RETURNING 1
CALLED n=4 RETURNING 3
CALLED n=2 RETURNING 1
CALLED n=1 RETURNING 1
CALLED n=3 RETURNING 2
CALLED n=5 RETURNING 3
3

Exited with code 0
```

However, when testing, this did not fix the issue at all, so I deleted it and began to look down a different route for implementation.

When outputting all the calls and results, I came up with a theory that the left value of the addition was somehow being replaced with one, and this was being added to the right value which led to some weird returns that were often like the value of n that the function was originally called with.

My prior focus had all been on the new calling and subroutine code, however I decided to look into the addition itself, and more specifically the evaluate() call in Binary Operator and this is where the problem occurred.

All binary operators currently use properties for the result of the left and right evaluation so that the corresponding calculate() method can then be called to give the correct result. However, in certain cases of recursion, calling evaluate() on the right half can cause the value of this property to change.

This is because the instance of the Add object is for the actual + token in the plaintext, and there is not a new Add token for each call of the function. Therefore, when evaluating the right, it would call the function again, triggering a new addition, and therefore overriding the old value of the leftValue so that the calculation would be incorrect.

The fix for this is very simple and is storing the result of left.evaluate() into a variable that is unique to that instance of the evaluate() call, then evaluating right(), and then setting the leftValue property to the stored value so that calculate() can occur, as evaluating the right side will not override a local variable.

```
evaluate(){
    let leftValueHolder = this.left.evaluate()
    this.rightValue = this.right.evaluate()
    this.leftValue = leftValueHolder
```

From here, the function now gave the correct values for any call of fib().

```
8   fib(25)   75025
```

OVERFLOW PREVENTION

```
code.erl
1   calls = 0
2
3   function fib(n)
4       global calls = calls + 1
5       if n <= 2 then
6           return 1
7       endif
8       return fib(n-1) + fib(n-2)
9   endfunction
10
11  fib(35)
12  calls
```

```
9227465
18454929
```

My original idea for stopping call stack errors was counting each new call() that occurred and ensuring that this never exceeded a certain value: which is now going to be 1500 due to the earlier test that showed that the factorial function crashed at 1599.

However, as shown in the example on the left, the actual value needs to reflect the number of concurrent calls, not overall calls, as the Fibonacci call for 35 results in over 18 million calls occurring.

Using this example, calling the fib function on a huge number, such as 20 thousand, would cause an almost instant crash due to these calls being concurrent.

```
11  fib(20000)
12  calls
```

```
call(call){
    if (Subroutine.callStackSize >= 1500){
        return new EvaluationError(call, "Call stack exceeded maximum size of 1500")
    }
    Subroutine.callStackSize++
```

My solution is adding a class attribute called callStackSize to the Subroutine class to represent these calls.

```
Evaluator.currentScope = previousScope
Subroutine.callStackSize--
return returnValue
```

This will be checked each time a subroutine is called, and if it exceeds the threshold of 1500 an error will be thrown. Otherwise, it will be incremented and then decremented at the end of call().

```
! ERROR @line 5
Evaluation Error: Call stack exceeded maximum size of 1500
    return triangle(n-1) + n
                         ^
```

```
1125750
```

As shown, a function that calls itself 1501 times will return an error, but when testing for 1500 times the program still allows it to execute.

INVALID INPUT HANDLING

Because I have done a lot of similar testing before, and a lot of error handling has already been added, I am not going to show all of the cases that already worked but will only include the failed cases and the changes I made to the code to fix them. This will mainly consist of parser errors.

➔ Missing identifier and starting bracket '('

```
if (self.token == null ){ // function name
    return new SyntaxError(subroutineToken, `Expected identifier to follow ${subroutineToken.tag} decleration`)
}
if (!(self.token instanceof Identifier)){
    return new SyntaxError(self.token, `Expected identifier`)
}
subroutineToken.identifier = self.token
self.continue()
if (self.token == null){
    return new SyntaxError(self.previous, "Expected '(' to follow identifier")
}
if (!self.check_tag("(")){
    return new SyntaxError(self.token, "Expected '(' followed by arguments")
}
```

```
! ERROR @line 1
Invalid Syntax: Expected identifier
function ()
         ^
```

This error handling was not completely needed, as it was already in place, however I have made the messages more descriptive and so that they point to the correct token.

➔ Only Identifiers should be allowed as parameters.

Currently, the system uses call() to parse the parameters, which uses statemenet_chain(), and therefore non-Identifier parameters can be declared and will not result in an error.

```
parse_arguments_or_parameters(self){
    self.continue()
    if (self.check_tag(')')){
        self.continue()
        return []
    }
    let argumentsOrParameters = []
    let argument = self.statement_chain(self)
    if (argument instanceof Error){
        return argument
    }
    argumentsOrParameters.push(argument)
    while (self.check_tag(',')){
        self.continue()
        argument = self.statement_chain(self)
        if (argument instanceof Error){
            return argument
        }
        argumentsOrParameters.push(argument)
    }
    if (self.check_tag(')')){
        self.continue()
        return argumentsOrParameters
    }
    return new SyntaxError(self.token, "Expecte
}
```

As a fix, I am changing call) to parse_arguments_or_paramaters() to return an array of arguments/parameters instead of pushing them to a new instance of call. Therefore, this can be easily reused in parsing for calls and parameters.

Factor will now need to create the new instances of call and set the result to the argumentsAsts property of this call. Similarly, build_subroutine() will call it, ensure that none of the items in it aren't identifiers, and then set the parameters property to it.

```
while (self.check_tag('(')){
    let call = new Call(self.token.position, self.token.line)
    let argumentsAsts = self.parse_arguments_or_parameters(self)
    if (argumentsAsts instanceof Error){
        return argumentsAsts
    }
    call.argumentsAsts = argumentsAsts
    call.callee = result
    result = call
}
```

```
let parameters = self.parse_arguments_or_parameters(self) // checking paramete
if (parameters instanceof Error){
    return parameters
}
for (let parameter of parameters){
    if (!(parameter instanceof Identifier)){
        return new SyntaxError(parameter, "Expected identifier as parameter")
    }
}
subroutineToken.parameters = parameters
```

```
! ERROR @line 1
Invalid Syntax: Expected identifier as parameter
procedure test(value + 1)
               ^
```

Now any asts will throw an error in the parser.

➔ Nothing following commas.

```
! ERROR @line 1
Invalid Syntax: Expected literal
procedure test(value,)
               ^
```

The error was caught, but I added an extra check for a better message.

```
! ERROR @line 1
Invalid Syntax: Expected next value to follow ','
procedure test(value,)
               ^
```

```
if (self.token == null || self.check_tag(')')){
    return new SyntaxError(self.previous, "Expected next value to follow ','")
}
```

➔ Nested subroutines

```
code.erl
1  procedure test(value)
2      procedure test2()
3          "hello"
4      endprocedure
5  endprocedure
```

Currently a subroutine can be defined within another subroutine. Although this feature is present in some programming languages, such as python, I have decided to exclude it. This is because there is no mention of it in the specification, and because it is a complicated and infrequently used feature which will most likely be used incorrectly if added.

```
this.allowReturn = false // True if a function is currently being parsed
this.allowSubroutines = true // False if a subroutine is being parsed
```

```
if (this.token instanceof Subroutine){
    if (this.allowSubroutines){
        return this.build_subroutine(this)
    }
    return new SyntaxError(this.token, "Cannot define a subroutine inside another subroutine")
}
```

When checking with my stakeholder Tanish, he agreed that it should not be added.

The implementation is almost identical to the one for ensuring return statements cannot be defined outside of functions: a new property called allowSubroutines for the parser, which is true by default, and is set to false when a subroutine is being parsed. Therefore, if a new subroutine definition occurs inside of a subroutine, parse() throws the new error instead of calling build_subroutine().

➔ Calling non-subroutines

```
evaluate(){
    let callee = this.callee.evaluate() // Ensures callee is valid datatpye
    if (callee instanceof Subroutine){
        return this.callee.evaluate().call(this)
    }
    if (callee instanceof Error){
        return callee
    }
    if (callee instanceof DataType){
        return new TypeError(this.callee, `Type ${callee.type_to_string()} cannot be called`)
    }
    return new TypeError(this.callee, "Cannot call this")
}
```

```
! ERROR @line 1
Identifier Error: 'notReal' was not declared
notReal()
^
```

Previously, calling undefined functions would crash the program because there was no check to see if evaluate() returns an error. Now this is check present, and the catch for non-functions being called has been moved for consistency.

➔ Operating on subroutines with no return

Currently a subroutine returns null if it has no return, which will then crash the program if it is operated on. My solution also helps change the data type classes, because the type_to_string() should not have been a method and has now been replace by a getter called typeAsString which will be called in all of the calculate() error messages.

```
class StringType extends DataType{
    get typeAsString(){
        return "String"
    }
```

```
} else {
    returnValue = {typeAsString : "EmptySubroutineReturn"}
}
```

Subroutine will now return a custom object with a typeAsString property set to "EmptySubroutineReturn"

```
! ERROR @line 5
Type Error: Cannot combine types EmptySubroutineReturn and Integer
test() + 3
^
```

Therefore, if it is encountered in calculate(), the user is given an error saying they cannot operate on a subroutine with no return statement.

```
if (evaluated instanceof DataType){
    console.log(evaluated.display())
}
```

Importantly, evaluate_single_ast() must be changed to only call display() on any Data Types. Practically this does nothing different because it would only happen anyway, but this stops the program crashing if a subroutine was called normally.

Overall, I do not love this implementation: it is not as smooth as solution as the rest of the system, and the error message is not super descriptive. However, it has a very similar implementation to python, which describes it as "NoneType", so overall I feel that it is acceptable in the short term and can be changed in the future if I feel a better error message is neccecary.

➔ **Other features**

```
code.erl
1  procedure test(value)
2      value * 4
3  endprocedure
4
5  function myFunc()
6      return test
7  endfunction
8
9  myFunc()(2)
```

Neither of these are bugs, but they both showcase some interesting behaviours of the system.

The first shows how subroutines can be returned from functions, and can then be called, and this example correctly outputted others. The second shows how functions can be called within subroutines, and this correctly displays the square numbers from 1 to 144.

```
code.erl
1  function squared(x)
2      return x ^ 2
3  endfunction
4
5  for i = 1 to 12
6      squared(i)
7  next i
```

This is the end of this section, as now all the features I want to add have been implemented.

EVALUATION

After this stage, the state of the success criteria are as follows:

No.	Criteria	Implemented?
7.1	Allow a procedure to be defined by "procedure", followed by the identifier to represent it, followed by brackets () containing an optional list of parameter names. Then, the contents of the procedure, closed by the "endprocedure" keyword	**Yes**
7.2	Allow subroutines to be called by their identifier after definition, followed by brackets containing the arguments	**Yes**
7.3	Ensure that a new variable scope is created whenever a subroutine is run, with the parameters being assigned as variables in this scope. Ensure that this scope is removed after the subroutine is complete, and global variables being able to be read, but not written to.	**Yes**
7.4	Allow the "global" keyword to be used before an assignment inside a subroutine definition, so that global variables can be re-written inside of the local scope.	**Yes**
7.5	Allow functions to be created in the same format as procedures, but with the "function" and "endfunction" keyword instead. Allow the return keyword to be used within functions, at which point the subroutine will immediately stop processing	**Yes**
7.6	Allow the returned value of a function to be stored in values and integrated with other expressions as if a normal value.	**Yes**
7.7	Ensure recursion can occur, with multiple instances of the same subroutine existing at once, each with separate scopes and parameters which cannot interact.	**Yes**
7.8	Ensure complete error handling, with the number of arguments always matching the parameters, the syntax being perfectly followed and so that return statements cannot be used in procedures, all with descriptive error messages.	**Yes**

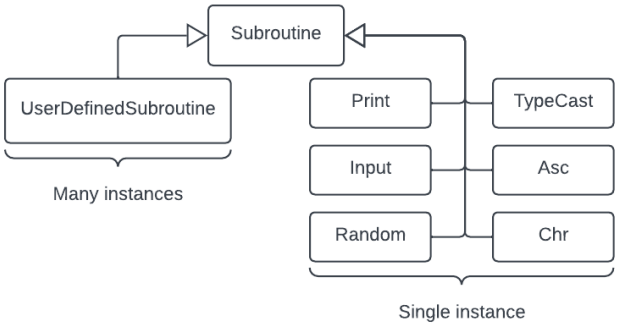
Overall, I am very happy with how this stage ended up. It ended up being quite different to my plan in certain aspects: mainly with how the different scopes were managed.

However, I feel like I adapted very well, and the new approach is a lot easier to understand and use so feel like it is better than my original approach, even if it worked. Therefore, I am happy with how this stage has gone, and am ready to move onto the native functions. This next stage should be very important, as I will finally have print() and input() implemented, so it will become more like an actual programming language, so hopefully I will be able to get some more user evaluation at the end of that stage.

NATIVE FUNCTIONS

DESIGN

The Subroutine code from last section will now largely be moved into the UserDefinedSubroutine class, which will extend Subroutine, to allow for native functions.



Now, there will be a new class for each of the native functions. Only one instance will exist of each in a program, but I feel it is still best to have classes for consistency with the rest.

This is because, when the program is run, the global symbol table will be initialised with some values as default, with the different names corresponding to the different instances of each class. Therefore, whenever an Identifier with that function name is used, it will reference the function so it can be called.

This should be very easy to implement, as it works with the pre-existing code. Each class must have a custom call() method, where it can access the arguments and perform the corresponding action.

My plan will describe these methods, alongside the arity (number of parameters) and the types of values it must be given. For each function, if the arity or types do not align, an error will be returned: an EvaluationError for the arity and a TypeError if the types do not align.

For any additional pre-conditions that are expected, I will explain them in the brief plan for each method. Additionally, even though I am describing some of these as functions, some are procedures, so will return the same nullReturn as the old functions.

PRINT

➔ Arity: 1 or more arguments
➔ Type: expects all arguments to evaluate to be a string.

The value of all the evaluated strings should be concatenated with a space separating them, and then outputted, using console.log() for now. Later, this will have to be changed to work with the interface, because it will need to be displayed into some sort of output window. The custom null value should be returned, which is a property of the Subroutine class so can be accessed.

Note that the Evaluator code must also be changed to no longer output the result of every single calculation in evaluate_single_ast(), so that now print statements are the only time values are outputted.

INPUT

➔ Arity: 0 or 1 argument
➔ Type: evaluate to a string

If there is an argument passed, then it will be outputted as a prompt for the input. Then the program will take an input from the user, for now using the same method as the old shell input, and then return a new StringType with the value being the string that was entered.

Again, this will also have to be later redesigned with the new interface, however I am not planning that at all now and will just use the same Node JS input that I have been using for the shell to function.

RANDOM

➔ Arity: 2 arguments
➔ Type: either both integers or both floats

```
myVariable = random(1,6)
Creates a random integer between 1 and 6 inclusive.

myVariable = random(-1.0,10.0)
Creates a random real number between -1.0 and 10.0 inclusive.
```

The specification clearly describes that the random system here is inclusive. Therefore, depending on whether the values are integers or floats, the corresponding value will be generated and returned as the value of the equivalent data type, using JavaScript's Math.random() generator.

For example, if the 2 and 5 are passed, then an IntegerType with an integer value from 2-5 will be returned.

TYPE CAST

➔ Arity: 1 argument
➔ Type: any

For this native subroutine only, 5 different versions will be created for each of the 5 casting methods. Each instance will have a tag property for the corresponding class which the call() property will use.

For example, when creating the TypeCast instance for the "str" Identifier, it will be given the tag StringType, to represent what it converts to. Likewise, the ones for "float" and "real" will both have FoatType. Therefore, they will all use the same call() method based on the property given without having to create entirely separate classes for each of the 5 different casts.

The call() method itself will be very simple, because the typecasting code was already written in section 6, so it will call cast_to_type() on the argument that it received, passing its tag – which is the type to cast to – as a parameter. This will return the typecasted version, which can then be returned.

ASC

➔ Arity: 1 argument
➔ Type: String with length of 1 only

This will an IntegerType with the ascii value of the character that was passed to it, so this is why there must be an extra check that the string is only 1 character long.

CHR

➔ Arity: 1 argument
➔ Type: integer

This will return a StringType with the value being the corresponding ascii character of the integer argument.

ADDING THESE TO GLOBAL

The new build_global() method in Interpreter will be called in its constructor and will add each of these to the global symbol table. Therefore, it must create a new Identifier with the name of each subroutine and set its value to a new instance of the corresponding class, with the corresponding tag for the TypeCasts.

Overall, this module should be very simple to implement due to the framework already being in place, however implementing them, especially printing and input, will drastically change the program and make it very close to the final product by making it a feasible program.

PRINT

```
class Subroutine extends Token {
    static nullReturn = {typeAsString : "EmptySubroutineReturn"}

    get typeAsString(){
        return "Subroutine"
    }
}

class UserDefinedSubroutine extends Subroutine {
    static callStackSize = 0
```

The first step was to spit the existing Subroutine class into the basic Subroutine class, with all of the features being transferred into the UserDefinedSubroutine class, which will represent any non-native functions.

A lot of the other code had to be changed, mainly in the parser, to reflect this change, however in evaluation all Subroutines will be treated the same.

Therefore, calls will check that they are calling any subroutine, regardless of the type.

```
class Print extends Subroutine {
    call(call){
        let output = [] // output
        if (call.argumentsAsts.length == 0){ // Ensure 1 or more parameters
            return new EvaluationError(call, "print expected 1 or more arguments, 0 given")
        }
        for (let item of call.argumentsAsts){
            let result = item.evaluate()
            if (result instanceof Error){
                return result
            }
            if (!(result instanceof StringType)){
                return new EvaluationError(result, `Can only print type String, not ${result.typeAsString}`)
            }
            output.push(result.display())
        }
        console.log(output.join(' ')) // ONLY PRINT STATEMENT
        return Subroutine.nullReturn
    }
}
```

Now the new Print class extends subroutine, containing a call(argument) to parse the print statement.

It ensures there is at least one argument and will otherwise return an error. Then, for each argument, it will evaluate() it, ensure it is a String, and will join the strings together with spaces and output it.

It will then return the same empty subroutine return that was previously used, which is now a class property of subroutine to reduce code repetition.

CHANGES TO PUSHING SYSTEM

```
push_native_subroutine(name, subroutine){
    let symbol = new Symbol(new Identifier(null, null, name, true))
    symbol.value = new subroutine(null, null)
    this.table.push(symbol)
}
```

Instead of the original system, I have decided to use a new push_native_subroutine() method to use abstraction, which now accepts the name for the identifier and the class of the subroutine.

This is much better than the old system, as it removes a lot of the additional steps that will be added to build_global() to allow it to seem a lot simpler.

```
class Identifier extends Token{
    constructor(position, line, name, constant=false){
```

Identifier is also changed to allow the constant property to be added, which is default at false, so therefore in the new method the native functions can be set to constants by default so that they cannot be edited.

```
class Interpreter extends Evaluator{
    constructor(){
        super()
        this.plaintext = null
        this.tokens = []
        this.have_tokens = false
        this.build_global()
    }

    build_global(){
        Evaluator.global.push_native_subroutine("print", Print)
    }
```

build_global() is now called in the constructor of the Interpreter class, and now the chain of calls will be listed in sequence inside this class, as shown with the arguments being the name "print" with the corresponding class.

Finally, the output statement in evaluate_single_ast() is removed so that not every single calculation is outputted (finally)

Examples:

code.erl

```
print("hello", "world")
```

```
hello world
```

```
! ERROR @line 1
Evaluation Error: print expected 1 or more arguments, 0 given
print()
    ^
```

INPUT

```
class Input extends Subroutine {
    call(call){
        if (call.argumentsAsts.length > 1){ // Ensure 1 parameter
            return new EvaluationError(call, `input expected 0 or 1 arguments, ${call.argumentsAsts.length} given`)
        }
        if (call.argumentsAsts.length == 1){
            let result = call.argumentsAsts[0].evaluate()
            if (result instanceof Error){
                return result
            }
            if (!(result instanceof StringType)){
                return new EvaluationError(result, `Can only output type String, not ${result.typeAsString}`)
            }
            console.log(result.display()) // Outputs input message
        }
        return new StringType(call.position, call.line, prompt())
    }
}
```

This call() method for Input ensures only 0 or 1 inputs are allowed.

It will output the only argument if it is a String, before using the previous prompt() system to get an input from the user.

```
build_global(){
    Evaluator.global.push_native_subroutine("print", Print)
    Evaluator.global.push_native_subroutine("input", Input)
}
```

Note that this is still a temporary solution, as when a proper interface is added the prompt() implementation will need to be replaced, but for now this system should work.

code.erl

```
1  a = input("Enter name: ")
2  print("Hello", a)
```

```
└─ node src code.erl
Enter name:
Pramukhi
Hello Pramukhi
```

Then, after adding this input to build_global(), it could be tested. It did mainly work; however, I would like the input to be on the same line as the message.

When researching, a solution for outputting without bringing a new line was using `process.stdout.write`, however when using this, the prompt() method would remove the output when entering an input.

```
process.stdout.write(result.display()) // Outputs input message
```

Therefore, I decided to pass the string that should have been inputted into the prompt parameter, set to a string that is empty "" by default. I do not love this, however it is acceptable because prompt() is temporary.

```
    message = result.display() // Outputs input message
}
return new StringType(call.position, call.line, prompt(message))
```

Now, this input is on the same line so the input system is complete.

```
└─ node src code.erl
Enter name: Pramukhi
Hello Pramukhi
```

RANDOM

GENERATING RANDOM NUMBERS

```
1   function getIntegerRandom(min, max){
2       return Math.floor(Math.random() * (max - min + 1) + min)
3   }
4
5   function getFloatRandom(min, max){
6       return Math.random() * (max - min + 1) + min
7   }
```

Because ERL using inclusive random number generation, and because JavaScript's random() only gives a number between 0 and 1, I decided upon using these two algorithms to generate inclusive random numbers.

```
9   total = 0
10  trials = 100000
11  for (let i = 0; i < trials; i++){
12      total += getIntegerRandom(1, 9)
13  }
14  console.log(total/trials)
```

I used each of these functions 10,000,000 times and took the average value to ensure they aligned with the expected value. In this case, because I am generating numbers from 1 to 9, the expected value is 5, which the integer random number generator was very close to, so therefore it is working as expected.

```
5.0002778
```

```
5.500471771058012
```

However, the float **value** was significantly off, at 5.5, so the float random was incorrect.

```
function getFloatRandom(min, max){
    return Math.random() * (max - min) + min
}
```

The solution was to remove the +1, which is only needed for generating Integer random numbers, which then produced a value close to 5.

```
4.999412263513929
```

Technically, because Math.random() produces a value x where 0 <= x < 1, the algorithm for generating float numbers is not purely inclusive, because the float that is exactly 10 can never be achieved, which may cause a very slight bias in this random function so that it is not purely fair.

However, this is okay because the chance of getting exactly 10 in a random float generation from 1 to 10 is 1 in 100,000,000,000,000,000, so this will be unnoticeable because the interpreter will not be used for something where exact random precision is not absolutely necessary.

IMPELEMENTING

The new Random class is as follows:

```
class Random extends Subroutine {
    call(call){
        if (call.argumentsAsts.length != 2){
            return new EvaluationError(call, `random expected 2 arguments, ${call.argumentsAsts.length} given`)
        }
        let min = call.argumentsAsts[0].evaluate()
        if (min instanceof Error){
            return min
        }
        let max = call.argumentsAsts[1].evaluate()
        if (max instanceof Error){
            return max
        }
        if (min instanceof IntegerType && max instanceof IntegerType){
            return new IntegerType(call.postition, call.line, Math.floor(Math.random() * (max.value - min.value + 1) + min.value))
        }
        if (start instanceof FloatType && max instanceof FloatType){
            return new FloatType(call.position, call.line, Math.random() * (max.value - min.value) + min.value)
        }
        return new TypeError(call, `Cannot use random on type ${min.typeAsString} with ${max.typeAsString}, expected two Integers or two Float`)
    }
}
```

```
Evaluator.global.push_native_subroutine("random", Random)
```

There were originally a few syntax errors, but they have been removed after testing.

code.erl

```
for i = 1 to 10
    if random(1, 2) == 1 then
        print("YES")
    else
        print("NO")
    endif
next i
```

```
YES
YES
NO
YES
YES
NO
YES
YES
NO
NO
```

After adding the function to the global table in build_global(), the small test program seemed to work as random, with YES and NO being printed in a very random arrangement. More formal testing to check the expected value will be implemented after typecasting, as currently non-strings cannot be printed.

TYPE CASTING

```
push_native_subroutine(name, subroutine, tag=null){
    let symbol = new Symbol(new Identifier(null, null, name, true))
    if (tag == null){
        symbol.value = new subroutine(null, null)
    } else {
        symbol.value = new subroutine(null, null, tag)
    }
    this.table.push(symbol)
}
```

push_native_subroutine() has been upgraded to accept an optional tag. By default, no tag will be pushed to the subroutine, but if one is passed it will be. This initially caused issues because I was trying to use ternary operators, but now works.

Because all the type casting code had already been written, this section was very easy to implement just checking for the right number of arguments and then casting to the tag type.

```
class TypeCast extends Subroutine {
    constructor(position, line, tag){
        super(position,line)
        this.tag = tag
    }

    call(call){
        if (call.argumentsAsts.length != 1){
            return new EvaluationError(call, `${this.tag} expected 1 argument, ${call.argumentsAsts.length} given`)
        }
        let old = call.argumentsAsts[0].evaluate()
        if (old instanceof Error){
            return old
        }
        return old.cast_to_type(this.tag)
    }
}
```

```
Evaluator.global.push_native_subroutine("str", TypeCast, StringType)
Evaluator.global.push_native_subroutine("int", TypeCast, IntegerType)
Evaluator.global.push_native_subroutine("float", TypeCast, FloatType)
Evaluator.global.push_native_subroutine("real", TypeCast, FloatType)
Evaluator.global.push_native_subroutine("bool", TypeCast, BooleanType)
```

Then, build_global() must be expanded with all the different calls according to the plan: remembering that "float" and "real" both can cast to floats.

code.erl

```
print(str(0.57))
print(str(True))
a = int("10") / 2
print(str(a))
```

```
0.57
True
5
```

As shown, these methods now work, allowing numerical values to be outputted when they are type casted to a string. Also, the string values can be casted to numbers and then is correctly used in a calculation and outputted again.

This is not a thorough test of all the features, and I will formally test them later.

ASC AND CHR CONVERSIONS

```
class Asc extends Subroutine {
    call(call){
        if (call.argumentsAsts.length != 1){
            return new EvaluationError(call, `asc expected 1 argument, ${call.argumentsAsts.length} given`)
        }
        let character = call.argumentsAsts[0].evaluate()
        if (character instanceof Error){
            return character
        }
        if (!(character instanceof StringType)){
            return new TypeError(character, `Expected type string, not type ${character.typeAsString}`)
        }
        if (character.value.length != 1){
            return new TypeError(character, "Expexted single character, as string of length 1")
        }
        return new IntegerType(call.position, call.line, character.value.charCodeAt(0))
    }
}
```

These methods follow the very similar format to the other methods: checking for the number of arguments, then returning the corresponding value is returned as the correct type.

These also need to be pushed to global, with their capitalised function names.

```
Evaluator.global.push_native_subroutine("ASC", Asc)
Evaluator.global.push_native_subroutine("CHR", Chr)
```

```
class Chr extends Subroutine {
    call(call){
        if (call.argumentsAsts.length != 1){
            return new EvaluationError(call, `chr expected 1 argument, ${call.argumentsAsts.length} given`)
        }
        let code = call.argumentsAsts[0].evaluate()
        if (!(code instanceof IntegerType)){
            return new TypeError(code, `Expected type Integer, not type ${code.typeAsString}`)
        }
        return new StringType(call.position, call.line, String.fromCharCode(code.value))
    }
}
```

From here, the values worked as expected, as shown in the test cases.

code.erl

```
print(str(ASC("A")))
print(CHR(65))
```

```
65
A
```

EVALUATION

After this stage, the success criteria are as follows:

No.	Criteria	Implemented?
8.1	Allow print() to be used, accepting 1 or more arguments, and outputting each argument separate by a space.	Yes
8.2	Allow input() to be used, accepting an optional parameter to be outputted as an argument. This will prompt for a user input, which will be returned by the function as a string containing the text that the user gave to the program	Yes
8.3	Allow str(), int(), float(), real() and bool() to return the corresponding data type that was defined in module 6.	Yes
8.4	Allow ASC() to be used, accepting a string with a length of 1, and returning the character's corresponding ascii value as an integer	Yes
8.5	Allow CHR() to be used, accepting an integer as an argument, and returning the corresponding ascii value as a single character string	Yes
8.6	Allow random() to accept two Integers as arguments, where it will return a random integer inclusively, or two floats, where it will return a random float inclusively.	Yes
8.7	Ensure all these functions cannot be redefined, receive the correct number of arguments in the correct types, and are accessible globally throughout the entire program	Yes

This module has been very simple to implement, mainly because all the foundations had already been implemented previously. However, I feel this module has been very important because now the program is usable: having inputs and outputs allows it to finally be treated like an actual language.

Therefore, I feel like this has finally reached the level where it could be used by students, which I feel like is a big turning point so I am more positive about the remaining modules.