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>

The code snippets are in Java, which I am not familiar with and they have constructed their interpreter in a different way, however it is the overall idea which I want to take forwards.

### calling functions

From what I took away, it is best to think as the function as the parenthesis (), rather than the identifier itself. Each one of these calls should call whatever is in front of them and become an instance of this. In this way they almost act like a postfix operator. This will require additional error handling, as calls will be parsed after every <factor> in the BNF because we will be unsure in parsing if they identifier they follow points to a function, or a call could even chain off another call that returns a function.

This focus on the () led me to realise the approach I want to take with this, because the rest of the description for how this person implemented functions does not fit with my system.

## the plan

### call

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.

It 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 constructor method will also create a calle property set to null, a scope property which points to a new instance of a SymbolTable, and an arguments array which will be empty, however because this is a JavaScript keywords, it will have to be named argumentsAsts (this will align a different array that will be used later)

The call() method in the parser will then manage the arguments. First it will check if it is empty, so if the next token is the closing brackets, and if it is then the Call will be returned. Otherwise, it will call statement\_chain(), and will repeatedly push the resulting AST to the argumentsAsts so long as it is followed by a comma, which will have to be added to the Lexer. Then when it detects the closing bracket, it will return the call to factor().

factor() will then set the calee property of the returned Call instance to what it previously parsed in factor(). It will then check if there is another bracket, and therefore if there is another call. If there is, it will repeat the process and will set the previous call method to the calee of the new call method.

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 a lone factor.

### identifier changes

From now, Identifiers will have a scope property which they will use in their evaluate() and assign() method, to refer to which SymbolTable they must search in. By default, this will be the global symbol table, which will now be a property of the Parser, so normal usage will still work as expected.

My idea is then that Identifiers which are parsed when creating a function will have null scope, and then when they are called by a Call method, their scopes will be set to the scope of the Call symbol table, so that they are then treated as local variables, but I will refine how this works later.

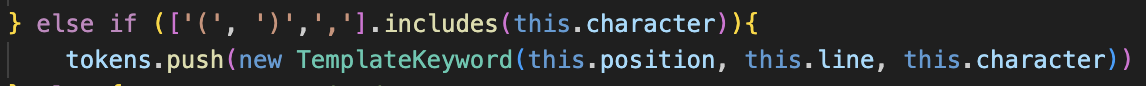
### subprocedure class

These will have position and line properties like all other methods, but their main feature will be the call

# development

### changing brackets to be tagged

A computer screen with colorful text

Description automatically generatedThis is very easy to change in the Lexer.

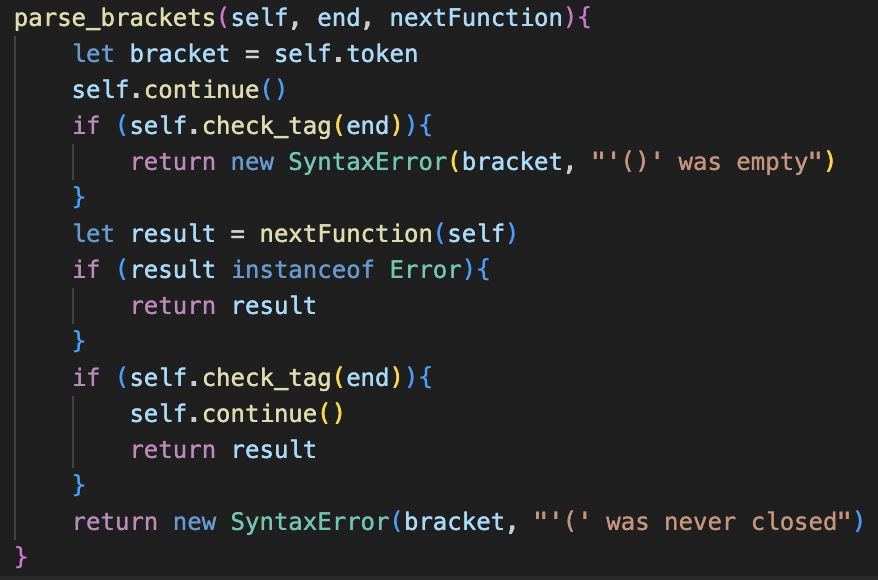
In the long term it is a positive decision, because these classes never contained any code, and as more brackets and punctuation will be added, it is best to use the generic class that exists.

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.

A computer screen with text

Description automatically generatedA screen shot of a computer program

Description automatically generatedSome 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.

A number on a black background

Description automatically generatedJust 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%.

## A screen shot of a computer code Description automatically generatedcall class

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.

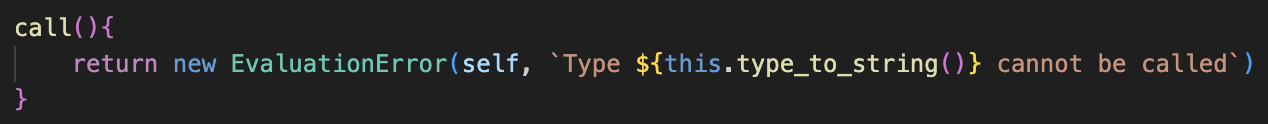
A black background with white text

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generatedThe 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.



### evaluator

**A computer screen with colorful text

Description automatically generated**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.

**A screen shot of a computer code

Description automatically generated**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.

A computer screen shot of text

Description automatically generatedrun(), 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.

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

**A screen shot of a computer screen

Description automatically generated**The new subroutine class is as shown.

**A screen shot of a computer code

Description automatically generated**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.

**A black background with blue and white text

Description automatically generated**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.

**A screen shot of a computer code

Description automatically generatedA black background with blue and yellow text

Description automatically generatedA screen shot of a computer code

Description automatically generated**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.

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

A screen shot of a computer

Description automatically generatedUnfortunately, 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.

**A screen shot of a computer program

Description automatically generatedA black background with colorful text

Description automatically generated**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.

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.

**A black background with blue text

Description automatically generated**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

**A computer screen shot of text

Description automatically generated**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 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.

**A number on a black background

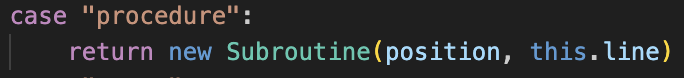
Description automatically generated**Now, the correct value of three is outputted, so therefore the system has worked.

A black background with white text

Description automatically generatedFinally, I changed the call() method in Subroutine to accept the arguments rather than the Call instance because it no longer needs the scope.

From here, this now allows functions to be run, now they need to be parsed.

## parsing subroutines

A computer screen shot of text

Description automatically generatedThe code for implementing the parsing of functions is very similar to a lot of the other structures.

A black background with colorful text

Description automatically generatedThe keyword is also checked for in make\_identifier()

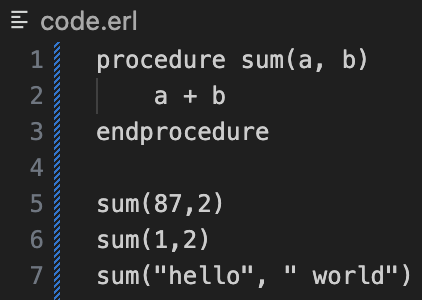
In parse() the check for the token calls the method.

A black background with blue and yellow text

Description automatically generatedThe method itself uses lots of already existing ideas: the loop for current Tokens, and it uses call() to parse the parameters.

Finally, the subroutine token is changed to assign the function to the identifier when it is evaluated.

A black background with white text

Description automatically generatedThe 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.

### A black background with colorful text Description automatically generatedfunctions vs procedure

A computer screen shot of text

Description automatically generatedThe tag class is added to subroutines to distinguish between the two types and make\_identifier() is updated to reflect the changes.

A computer code on a black background

Description automatically generatedA screen shot of a computer code

Description automatically generatedThe 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.

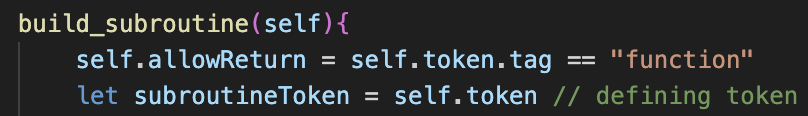
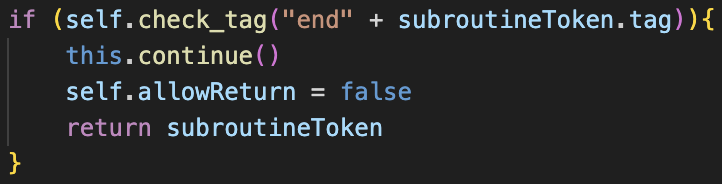
A computer screen shot of code

Description automatically generatedThis will then call the new return() method to parse these statements.

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:



### evaluation

A screen shot of a computer code

Description automatically generatedA screen shot of a computer program

Description automatically generatedA black background with colorful text

Description automatically generatedevaluate\_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.

The Subroutine class now calls the evaluate() method on its child if it receives one and will otherwise return null.

A black screen with white text

Description automatically generatedA screenshot of a computer

Description automatically generatedNow, functions and procedures can be correctly parsed, and the return value of a function can be parsed with other features,

A black background with white text

Description automatically generatedA black background with blue text

Description automatically generatedThe system ensuring that procedures cannot have returns is also correct.

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

A black background with colorful text

Description automatically generatedA screen shot of a computer program

Description automatically generatedFirst, I added a new case in the make\_identifier() method for the “global” keyword.

A screen shot of a computer program

Description automatically generatedIdentifier will have a boolean property for global, which will determine whether it uses the currentScope table or the global symbol table.  
The assignment() property now checks for the global tag as well and will set the corresponding properties of the Identifier according to this.

A computer screen shot of a black screen

Description automatically generatedThis 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.

A screenshot of a computer program

Description automatically generatedA screen shot of a computer code

Description automatically generatedThe 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.

### variable handling redesign

A computer screen shot of text

Description automatically generatedOverall, 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.

A black background with colorful text

Description automatically generatedA screen shot of a computer program

Description automatically generatedThis 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:

A screenshot of a computer program

Description automatically generatedA black background with blue text

Description automatically generatedA screen shot of a computer program

Description automatically generatedThe test cases for the addFive function now works as intended:

A number on a black background

Description automatically generatedMore complex examples also work with more advanced scopes.

## recursion handling

A screen shot of a computer code

Description automatically generatedA screen shot of a computer program

Description automatically generatedMy 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.

A screen shot of a computer program

Description automatically generatedAfter 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.

A screen shot of a computer screen

Description automatically generatedThis 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.

A screen shot of a computer

Description automatically generatedThis 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.

A black screen with white text

Description automatically generatedAs 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.

### the fibonacci problem

A computer screen with white text

Description automatically generatedI 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.

A black background with blue text

Description automatically generatedMy 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

A black background with blue text

Description automatically generatedMy 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.

A black screen with white text

Description automatically generatedHowever, 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.

A black background with white text

Description automatically generatedThe 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.

A close up of a logo

Description automatically generatedFrom here, the function now gave the correct values for any call of fib().

### overflow prevention

A black background with white numbers

Description automatically generatedA screen shot of a computer code

Description automatically generatedMy 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.

A black screen with text

Description automatically generatedA black background with white text

Description automatically generatedUsing 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.

A black background with blue text

Description automatically generatedMy solution is adding a class attribute called callStackSize to the Subroutine class to represent these calls.

A screenshot of a computer screen

Description automatically generatedThis 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().

A screenshot of a computer screen

Description automatically generatedAs 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.

**A screen shot of a computer program

Description automatically generated🡺 Missing identifier and starting bracket ‘(‘**

A black screen with white text

Description automatically generatedThis 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.**

**A screen shot of a computer program

Description automatically generated**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.

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.

**A screen shot of a computer code

Description automatically generatedA screen shot of a computer program

Description automatically generated**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.

**A black background with white text

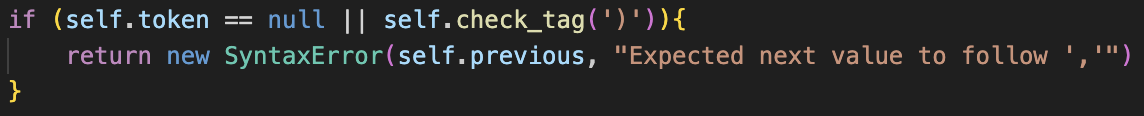
Description automatically generated**

Now any asts will throw an error in the parser.

A black and white screen with white text

Description automatically generatedA black background with white text

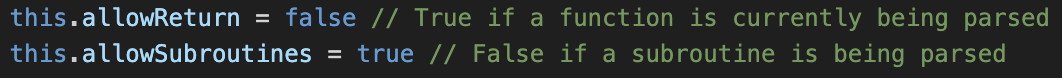
Description automatically generated**🡺 Nothing following commas.**

The error was caught, but I added an extra check for a better message.

**A screen shot of a computer

Description automatically generated🡺 Nested subroutines**

A black background with text

Description automatically generated****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.

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().

A screen shot of a computer code

Description automatically generated🡺 **Calling non-subroutines**

A black background with white text

Description automatically generatedPreviously, 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**

A black background with blue text

Description automatically generatedA black background with colorful text

Description automatically generatedCurrently 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.

A black background with white text

Description automatically generatedSubroutine will now return a custom object with a typeAsString property set to “EmptySubroutineReturn”

Therefore, if it is encountered in calculate(), the user is given an error saying they cannot operate on a subroutine with no return statement.

A black background with blue and yellow text

Description automatically generatedImportantly, 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.

A screenshot of a computer program

Description automatically generated**A screen shot of a computer

Description automatically generated🡺 Other features**

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.