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:

A group of text boxes

Description automatically generated🡺 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

A black and white text

Description automatically generatedTherefore, the BNF for <call> can be represented as shown:

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

A close-up of a sign

Description automatically generatedA close-up of black text

Description automatically generatedThis 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.

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

A close up of a logo

Description automatically generatedCreated 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.

A close-up of a sign

Description automatically generatedOverall, 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.

A close-up of a computer screen

Description automatically generatedThis 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

A screenshot of a computer

Description automatically generatedThe 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()**

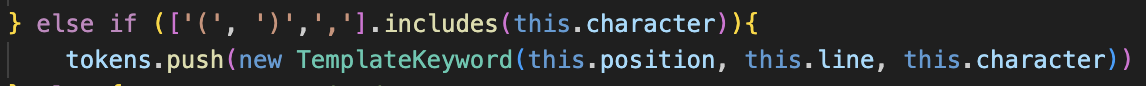
The program already has lots of instances where tags of template keywords must be checked, and it always results in a lot of double-nested if statements, to first check if the token is a TemplateKeyword, and then check the tag, as you cannot just check for the tag as other tokens do not have that property.

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.

# development

### changing brackets to be tagged

A computer screen with colorful text

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

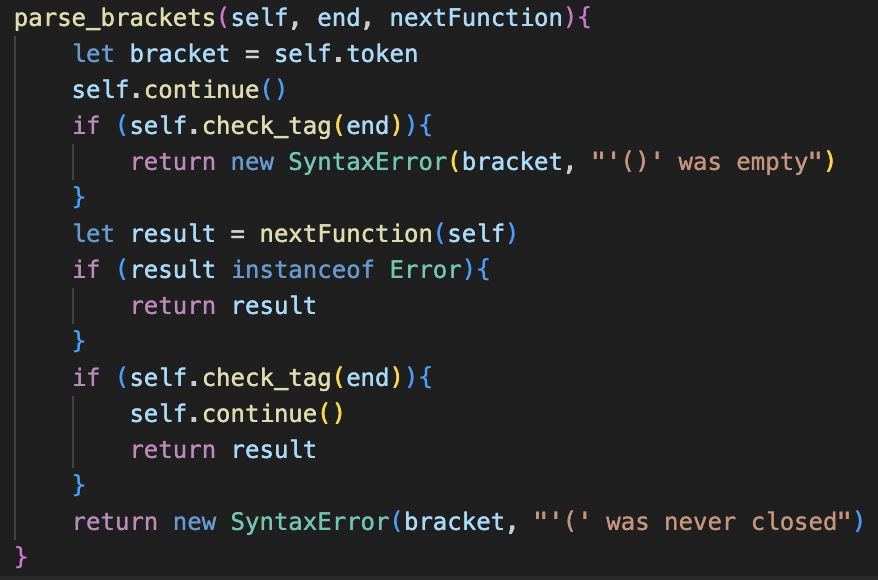
In the long term it is a positive decision, because these classes never contained any code, and as more brackets and punctuation will be added, it is best to use the generic class that exists.

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.

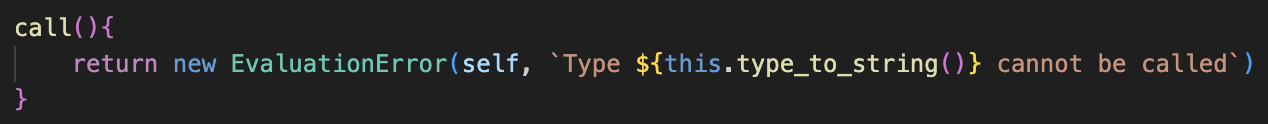
A black background with white text

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

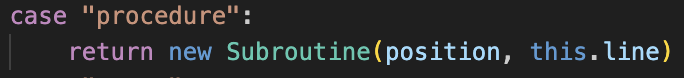
Description automatically generated**Now, the correct value of three is outputted, so therefore the system has worked.

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()

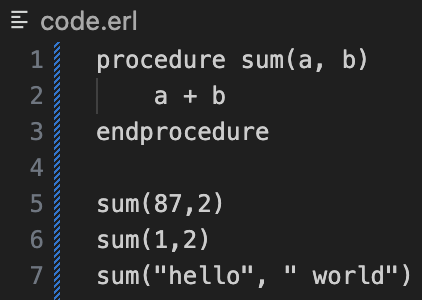
In parse() the check for the token calls the method.

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.

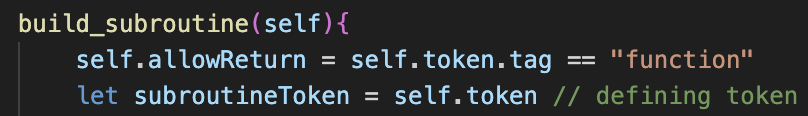
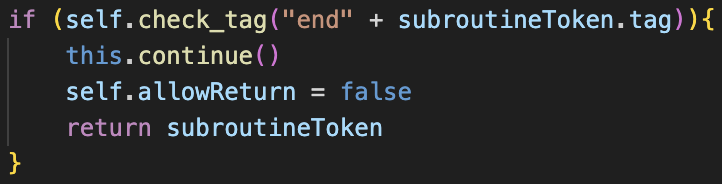
A computer screen shot of code

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

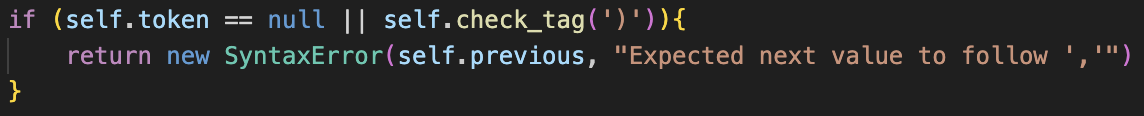
Description automatically generated**

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

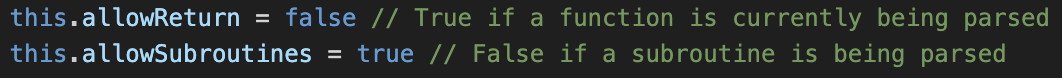
Description automatically generated**🡺 Nothing following commas.**

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.

This is the end of this section, as now all the features I want to add have been implemented.

# evaluation

|  |  |  |
| --- | --- | --- |
| 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** |

After this stage, the state of the success criteria are as follows:

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.