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.

A group of colorful labels

Description automatically generated with medium confidenceThis 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.

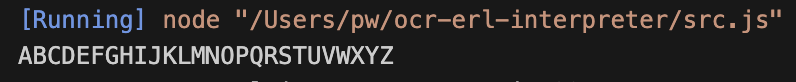
This is the general format. An additional check is needed at the start to see if a TemplateKeyword is present. If it is, and the tag is “const”, this will be remembered, and when the Identifier is reached, its constant property will be set to true.

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

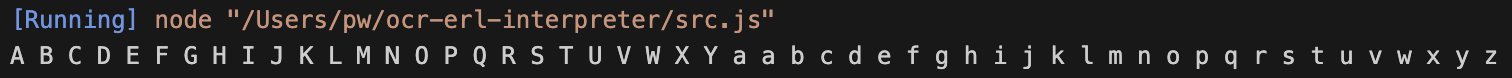
# development

## initial additions

### characters

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>

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

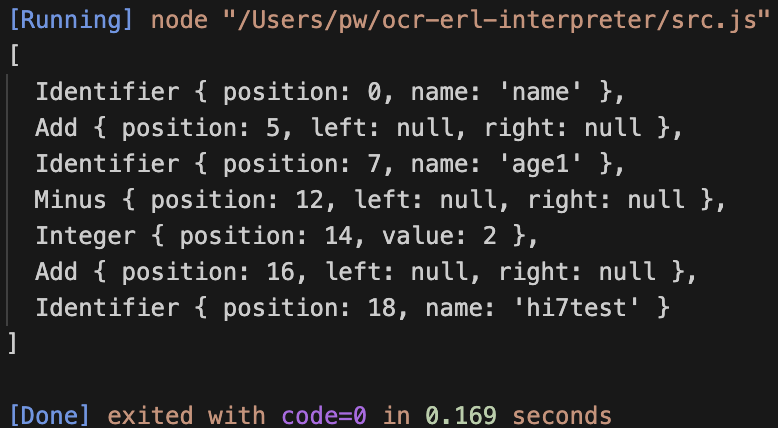
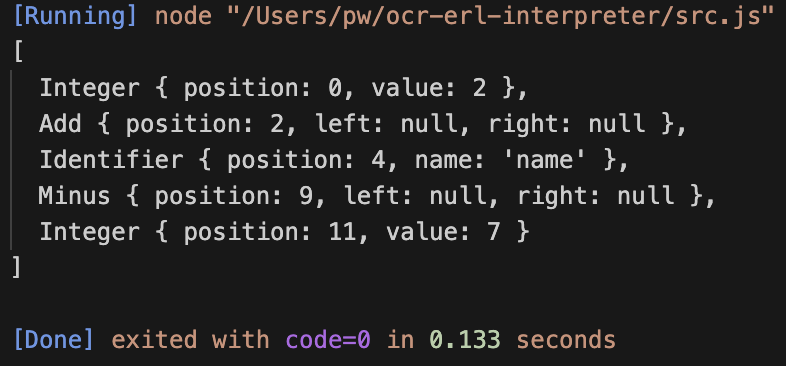
Therefore, I went for the more basic option of unpacking a string of every character into the list.

### A screen shot of a computer Description automatically generatedA computer screen with text on it Description automatically generatedA computer screen with text Description automatically generatedlexer changes

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.

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



### A screen shot of a computer program Description automatically generatedsymbol table and data type

A black background with white text

Description automatically generatedA black screen with text and numbers

Description automatically generatedThis 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.

A black screen with text and numbers

Description automatically generatedA black background with white and blue text

Description automatically generatedThe tests were successful.

A screen shot of a computer code

Description automatically generatedI 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.

A black background with colorful text

Description automatically generatedNow implementing the Symbol table.

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.

### integration with existing code

A screen shot of a computer code

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

A black background with blue and green text

Description automatically generatedThen, the basic Equals instance is created, with its corresponding character check in the Lexer’s make\_tokens()

A computer screen shot of a code

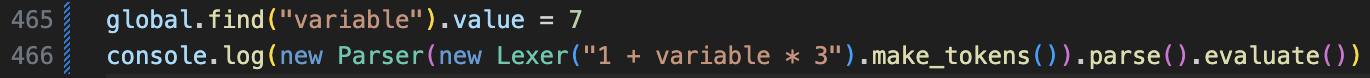
Description automatically generatedA black background with colorful text

Description automatically generatedFinally, 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.

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.

A black background with white text

Description automatically generatedTo let it evaluate, Identifier must be given an evaluate() method.

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

A black screen with text and numbers

Description automatically generated  
  
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/

## A black background with blue text Description automatically generatedA computer screen with text Description automatically generatedassignment

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

A screen shot of a computer code

Description automatically generatedThis was my initial attempt of implementing assignment().

A screen shot of a computer

Description automatically generatedA screen shot of a computer code

Description automatically generatedFor 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.

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

### evaluation

A screen shot of a computer code

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

A black screen with text and numbers

Description automatically generated variable = 7

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

A screen shot of a computer code

Description automatically generatedA screen shot of a computer code

Description automatically generatedThis 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.

A computer code with text and symbols

Description automatically generated with medium confidenceassign() will be the new method called be Equals to get access to the Data Type instance itself.

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

A screen shot of a computer code

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

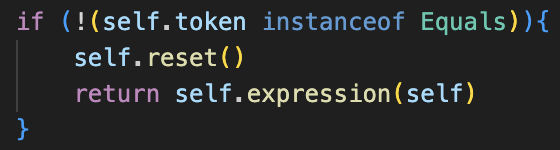
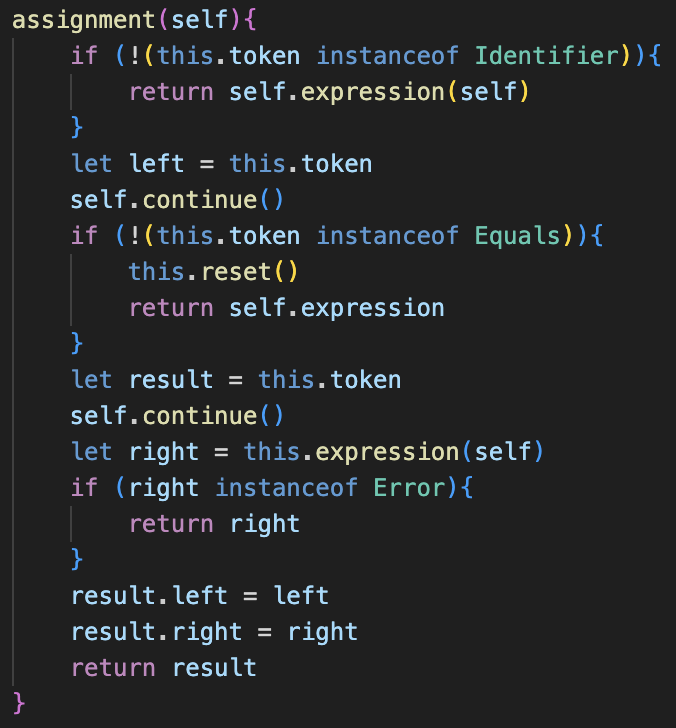
When setting up a very fabricated test case, this worked perfectly, and the value in the Symbol Table was correctly updated with the new value.

A black background with white text

Description automatically generatedHowever, 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

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.

A screen shot of a computer code

Description automatically generatedThe 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.

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

A screenshot of a cell phone

Description automatically generatedA black background with blue text

Description automatically generatedThere are 2 slight changes made.

A black background with text and numbers

Description automatically generatedA black background with white text

Description automatically generatedFirstly, changing Equal’s evaluate() method to return null, so that undefined would not be outputted after an assignment.

A screen shot of a computer code

Description automatically generatedAdditionally, 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.

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

A computer screen with colorful text

Description automatically generatedIdentifier 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.

A screen shot of a computer code

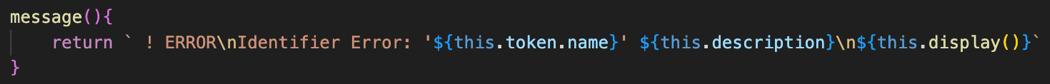
Description automatically generatedNow, find() has been modified for this new system. So, whenever an existing token is accessed, the lastReferenced property will be updated with the new Identifier property.

A screen shot of a computer code

Description automatically generatedA computer screen with text and numbers

Description automatically generatedAdditionally, 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.

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.

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

A black background with white text

Description automatically generatedA computer screen with white text

Description automatically generatedNow, 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.

## constants

### lexer implementation

A computer screen shot of text

Description automatically generatedA screen shot of a computer program

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

A screen shot of a computer

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

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

### evaluation

A screen shot of a computer code

Description automatically generatedI 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.

A screen shot of a computer code

Description automatically generatedA black background with white text

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

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

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.

A screen shot of a computer program

Description automatically generatedThe 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.

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.

A screenshot of a computer program

Description automatically generatedA screenshot of a black background with white text

Description automatically generatedA screenshot of a black background with white text

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

A computer screen with text

Description automatically generatedA computer screen with text

Description automatically generatedIn order to discover the issue, I added some output statements

A computer code with text

Description automatically generated with medium confidenceSomehow 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.

A computer code on a black background

Description automatically generatedA computer screen with text on it

Description automatically generatedMy 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.

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

A screenshot of a computer

Description automatically generatedHowever, 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

A screen shot of a computer code

Description automatically generatedSome 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.

A black screen with white text

Description automatically generatedA computer screen with text

Description automatically generatedMy 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.

A screen shot of a computer code

Description automatically generatedAs 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.

A black background with white text

Description automatically generated**A screenshot of a cell phone

Description automatically generated**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.

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.

A black background with blue and white text

Description automatically generatedA screenshot of a computer

Description automatically generatedIt 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.

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

# evaluation

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

The success criteria for this module are as follows:

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.