Variables

# design

## rough Description

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

Firstly, the new Identifier Class will be created, and will be produced in the Lexing stage, containing a name method which is the string of characters it was created with.

However, as we may have multiple unique Instances with the same name, all instances with the same name must access their value from the same location.

This is where the Data Type class is used, with each instance having a name and value property, with the value being able to be accessed and changed.

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 instance, and all new variables will be added to it, however in the future creating multiple tables allows scope for functions and procedures to have their own local variables.

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 starting value.

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, and this Class will extend Error.

### assignment

Before the parsing can be added, a new Equals binary operator needs to be added. This will have the expected position property.

There will be a new method in the parser which will be called in parse() if the second token in the list of tokens is an Equals. The parser will therefore need a reset() method, as we will have to revert the continue() if the second token is not an Equals. From there this assignment() method will be called.

This will then construct the abstract syntax tree with the following BNF:

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

This is relatively straightforward but will need to check for every other possible error in how it is typed, with lots of checks for potential Syntax Errors.

At the end, this will return an Equals token, with the Identifier as its left child and expression as its right. From here, when 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 be a whole expression as the value.

### constants

A new class will be created called Template, which won’t be used by a single keyword, but for any keywords which do not contain any code, and will never be evaluated, but are placeholders.

These will have a tag property which will represent their type, for example when the Lexer detects a string of characters “const”, it will add a Template token with the tag “const”.

This will then be recognised by the Parser, and if it is present, then the assignment() method will give the Identifier a property that constant is true, which will then in evaluation be passed to the Symbol Table, which will then be passed to the Data Type.

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

## basic implementation

### lexer changes

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

🡺 Identifier Class

We will need to create a new class, extending Token with a name 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 CHARACTERS is encountered. It will then return a new identifier with all of the characters that it had cycled through passed as a string.

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.

### data type

This will be a generic class for all basic Data Types. It will have a property for the name as a string and value pointing to an instance of a Data Type class, so far being Integer and Float

🡺 value

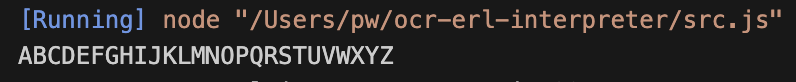
🡺 value getter

I am going to use getter and setter methods for this class, as they are something I have not experimented with before.

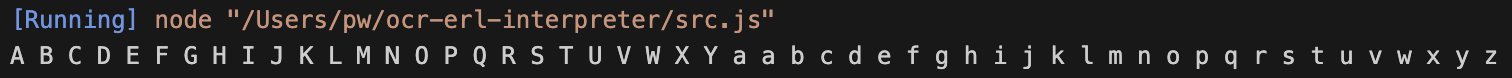
# development

## 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

This initial introduction followed the design stage exactly, ensuring to add the additional check to make\_tokens() to then call the make\_identifier() method.

As seen below, the two test cases functioned.

A screen shot of a computer code

Description automatically generatedA screen shot of a computer code

Description automatically generated



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

A black screen with text and numbers

Description automatically generatedA black background with white text

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 also temporary as Identifier Errors have not yet been implemented.

A black screen with text and numbers

Description automatically generatedA black background with white and blue text

Description automatically generated

The two test cases then correctly function.

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 and takes its value from the parameter name, so is defined upon creation of a Data Type instance.

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 in order to change the value, so the correct version is:

test.find(“hello”).value = 7

A computer screen with colorful text

Description automatically generatedThe next issue was that the newly created Data Types were not added to the table list, which was a quick fix, after realising that using [-1] was a python feature so had to be replaced.

### integration with existing code

A screen shot of a computer code

Description automatically generatedFirstly, global is declared as a variable for the whole program to use – this is temporary until functions are implemented. Identifier is also given the previously used find call as a value property.

A black background with blue and green text

Description automatically generatedA black background with colorful text

Description automatically generatedA screen shot of a computer code

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

Finally, the factor() method must be changed to also include Identifiers alongside Integers and floats, so that they can be parsed and included into expressions.

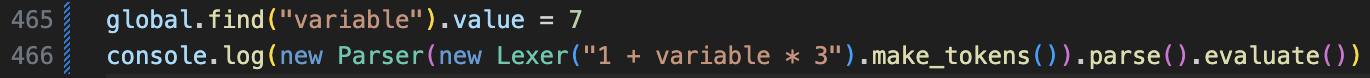
This is the test expression to be parsed, and initial problem occurred with Symbol Table being accessed before initialisation, so the Data Type and Symbol Table classes were moved to the top of the document in order to fix this.

A computer screen shot of a code

Description automatically generatedAfter this, the correct abstract syntax tree was produced, which only outputs Error due to variable not being defined yet, so it works as expected.

A computer screen with text on it

Description automatically generatedIn order to let it evaluate, the value of the Identifier must be returned.



A black screen with text and numbers

Description automatically generatedA black background with white text

Description automatically generatedHowever, this did not work, with the test case, as the old definition of value in Identifier was static and never changed, so a getter must be used so it calls find() on each access.

Changing this produced the expected result of 22.

## assignment

A black background with blue text

Description automatically generatedA computer screen with text

Description automatically generatedA screen shot of a computer code

Description automatically generatedThis was my initial attempt of implementing assignment.

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