switch and arrays

# design: switch

Overall, the plan with switch statements is to reuse as much code as possible from the existing if statement system because they share very similar structures, with only slightly different syntax and evaluation methods.

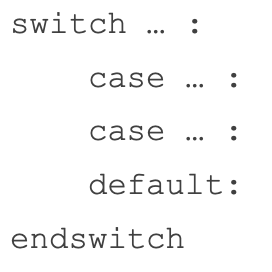
Therefore, I am going to have a redesign of the if statement parsing system as well. However, this manly involves lots of moving of code into different methods, as overall the system will be the same, but it will become more generic to allow switch statements to also be parsed.

### switch statement

This will be the new class and will have an almost identical structure to the IfStatement class: with a cases array and an elseCase property, both with the same defaults. It will also have an additional comparison property, which will store the value of the value that must be matched.

Instead of returning the contents of the Case that evaluates to True, SwitchStatement’s evaluate() method will return the Case where the condition is equal to the comparison. Likewise, it will also return the elseCase if it exists. Overall, these structures are very similar so that’s why I am combining how they are parsed.

## making the system more generic

I would like to re-use the IfCase and ElseCase tokens for the switch statement: because “case” and “if” are equivalents, because they both contain a condition and then are both followed by contents and potentially further cases.

Likewise, “else” and “default” are very similar: neither having a condition, both being followed by contents and both being the final case in the structure.

Therefore, these are very suited to being parsed together: with the only difference being the syntax of the specific lines: with if statements requiring “then”, and case requiring a colon. The other difference is that the first line format is different: with if statements the structure declaration and first case are integrated into the same line, whereas the switch has a declarative line, containing an additional comparison value, and the first case is defined on the following line.

For re-using the classes to work, the case keywords must be replaced with template keyword, instead of being directly converted in the Lexer. This is because otherwise there would be no way to distinguish “elseif” and “case” as they would both be IfCases, and there would be no way to identify them without using a tagged system, at which point just using a template keyword is easier.

### the generic method

generic\_if\_or\_switch() is going to be the generic method for parsing both statements.

This method is going to parse from the contents of the first case to the end of the structure. This means it will take in two parameters: the main statement token, either an IfStatement or SwitchStatement, and the current case. The IfCase class will be renamed to Case now to be more generic.

generic\_if\_or\_switch() will be called by one of two potential functions, which will initialise the different structures in their different syntaxes.

**🡺 build\_if\_statement()**

This is the method that is called whenever the IfStatement token is encountered in parse(). It will store the IfStatement token and will call if\_condition() to parse the first condition. It will then call the generic method, passing the IfStatement and Case as arguments.

**🡺** **build\_switch\_statement()**

This is called when a SwitchStatement is encountered and is slightly more complex. It also stores the original token, and then will call statement\_chain() to parse the value that follows it. It will store the result as the comparison property.

This method needs to parse the first line and the first case. For the first line, the Switch token is stored, and then statement\_chain() will be called to parse the comparator value, which is then stored in the Switch’s comparison property. It will then ensure there is a colon, as this is part of the syntax.

Then, it will use the switch\_condition() method to parse the following line’s initial case (soon to also be explained), and then will also call the generic method, passing the token and the first case.

### the condition methods

There are four, new methods, which serve the purpose of parsing each of the case declaration lines: one for “elseif”, one for “else”, one for “case” and one for “default”. They will ensure they follow the correct syntax for each line is followed and will be used by the generic method.

Overall, the four methods will work as follows:

**if\_condition()** 🡺 Parses a line after an if or elseif keyword, returning a new Case with the condition.  
**else\_condition()** 🡺 Parses the else keyword, ensuring it is alone and returning an ElseCase.

**switch\_condition()** 🡺 Parses each new case declaration, returning a new Case as a result.  
**default\_condition()** 🡺 Parses the default keyword and ensuring it has a colon, returning an ElseCase.

These should make more sense when I explain the generic method in more detail, however the idea is that the method calls these to replace the current method (like the old system), and that the generic method just keeps a track of adding contents to the current method and switching cases.

These methods are the ones that deal with the complexities of switching lines, and will create all of the Cases and ElseCases, ensuring that they strictly follow the syntax for each line.

A screen shot of a computer program

Description automatically generatedThe BNF for each of these methods is as shown:

This lays out the sequence of checks that each method will have to follow, calling the corresponding methods and checking for tokens to create the layout. Overall, though, each method should be very simple, using a lot of sequence and selection to ensure the precise syntax is followed.

The generic method will check to ensure that the first tokens are correct, so the methods themselves will not check for this, but will for everything that follows them. However, continue() will not be called before calling these methods, as the position and line of the first keyword will become the case position and line.

Not checking for the token helps with if\_condition(), as now it can be used for both if and elseif without any issues occurring with the differing token names, as the checking will occur beforehand.

I will now explain how the new method works, and how these methods integrate into the system.

## A diagram of a computer Description automatically generatedthe entire structure

The flowchart for the entire system is as shown. The first section is what we have already discussed, but the later part is familiar:

It is the same format as the old build\_if\_statement(), but instead of having pre-defined comparisons, it is more of a framework that can be filled in with different values and methods and can change itself based off the initial conditions, which are based off the original type of statement. This determines what keywords it is looking for, alongside the methods that correspond to them.

Therefore, if it is an if statement, it will check for “elseif”, “else” and “endif”, and switch statements will just check for “case”, “default” and “endswitch”, and these are equivalents.

Whenever a caseTag (“elseif” or “case”) is encountered, it means that a new case must occur, so it will push the old case to the main token’s cases and call the corresponding method. This will be defined at the start of the method, also depending on if it is an If or Switch statement. This is either if\_condition() or switch\_condition(), and will be stored in conditionFunction, and will then become the new currentCase.

This is the same for when the elseTag is encountered: both “else” and “default” must be the last case in their structures, so therefore the generic method will no longer check for caseTag or elseTag from this point onwards as no more cases can occur.

The elseFunction() method will be called to parse this, either else\_condition() or default\_condition(), and currentCase will be set to this. Previously, no longer checking for additional cases was done with a new nested loop, but instead the allowCases variable will be set to false which makes the system a lot simpler. This will no longer allow new cases to be parsed in the main loop, so achieves the same overall effect.

Finally, the endTag is also checked for, at which point the main token will be returned after pushing the final case. The default for every line is parsing it and adding it to the contents of the current case.

The key idea is that all the different options are chosen at the start of the generic method: the correct conditionFunction, elseFunction, caseTag, elseTag and endTag are all depending on the original token.

### overall

Just to summarise: there is a new SwitchStatement token that has been created, with a comparison property that will return the contents of the case that is equal to that comparison when evaluated.

Now if statements and switch statements will be parsed together, but this will only begin when the first case has been fully parsed. build\_if\_condition() will parse the first line and case, and build\_switch\_statement() parses the first line, comparison value and the first case line that follows it.

Therefore, when generic\_if\_or\_switch() is called, it will begin with the contents of the first case and will then parse the rest of the statement based on the different keywords that it encounters.

# development: switch

## A computer screen shot of a program code Description automatically generatedA screen shot of a computer program Description automatically generatedparser redesigns

As discussed in the plan, each of these methods will be called at each new case to parse it.

For the if statements, this code is very similar to previous: with if\_statement() basically being changed to if\_condition(), as well as now creating its own instance of Case rather than it being passed due to elseif now being a template keyword so that the Case class (the old IfCase class) can be reused.

Previously there was no parsing for the else condition, and I have managed to add some additional error checking, as previously if there were tokens after “else” they would be ignored rather than throwing an error, and now this check has been added so that this is the case.

The new switch equivalents are very similar, but instead of using “then” they use colons, as well as different error message to reflect their issues. The else, default case also requires a colon so that also must be added. When implemented, I will properly test them and add all the needed error handling.

### A screen shot of a computer program Description automatically generatedA screen shot of a computer program Description automatically generatednew build methods

The responsibility of these methods is to correctly parse everything before the contents of the first case. For the if statement, this is simple and involves parsing the if statement on the first line and advancing before calling the new method.

The switch statement has two lines before content, as the switch expression must also be evaluated before the first case, so that is why there is a lot more content. This is not yet properly error handled.

### generic building method

A screen shot of a computer program

Description automatically generatedThis will be called by both of the build() methods and will accept the main if or switch token as well as the initial case.

This is because after this point, both types of statement are parsed with the same structure, just using different keywords, which is why the corresponding keywords are all defined in the first line to reduce the number of checks.

I also managed to make this program a lot more readable. Originally, within the else section there was another loop which was essentially a replica of the main loop but without the if or else cases, which was a lot of repeated code.

Now the solution adds an additional variable that was not on the plan: allowCases, which is true by default, and is set to false when an else or default case are encountered.

This therefore no longer allows further cases to be parsed, which has the same effect of the inner loop but significantly reducing the amount of nesting and increasing how easy the program is to understand.

Overall, having this generic method reduces writing out an almost identical switch-building method so is a positive feature.

### debugging

A screen shot of a computer code

Description automatically generatedA screen shot of a computer program

Description automatically generatedBefore continuing with implementing switch statements, I needed to ensure if statements still parsed correctly, which they didn’t. The first issue was easy to fix, as I forgot to change the tokens to their new Template Keyword versions.

A black screen with text on it

Description automatically generatedA white letter on a black background

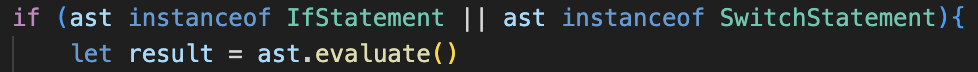
Description automatically generatedA computer screen with text

Description automatically generatedAfter further debugging of printing out the IfStatement class, I realised that when the closing keyword was encountered, the current case was being pushed to cases. However, if the current case is an if statement, it should be assigned to the elseCase property, and therefore the additional check is needed.

Finally, the error handling in parse() needs to be changed to catch the invalid use of keywords.

This is very easy to implement, and I also added the switch equivalents now.

## A screen shot of a computer program Description automatically generatedswitch evaluation

Now the evaluate() method needs to be added. This is similar to the IfStatement method, but instead of checking if there is a true boolean, it compares the condition of each Case to the comparison which is evaluated initially, ensuring that both the type and value are equal, in order to distinguish floats from integers in the switch,

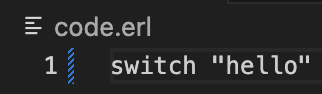
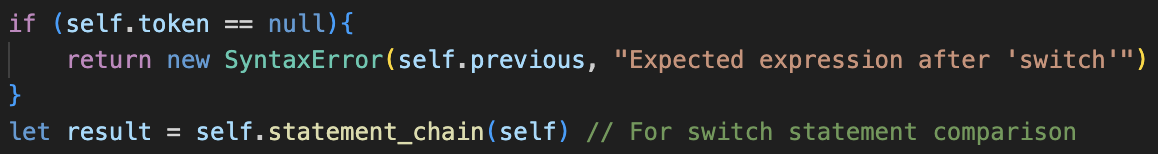
A screen shot of a computer

Description automatically generatedA black background with white text

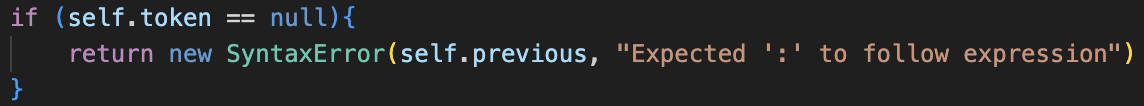
Description automatically generatedEvaluator must also be changed to check for Switch Statements.  
After this, an error occurred, but this was due to an incorrect colon check in build\_switch\_statement(), which after removal then made the program work as intended.

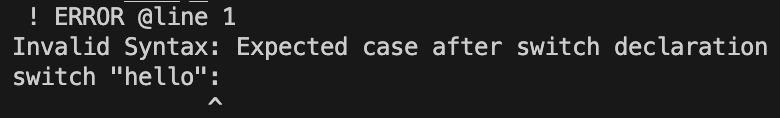
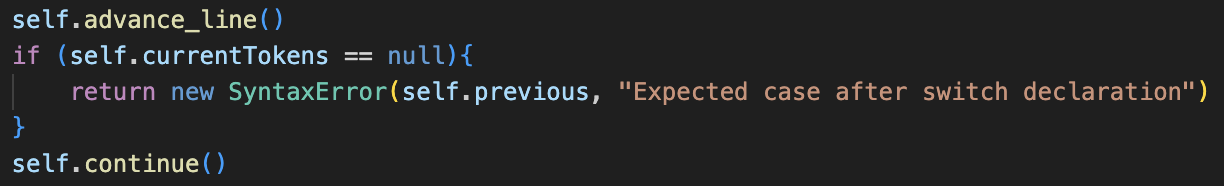
## error handling

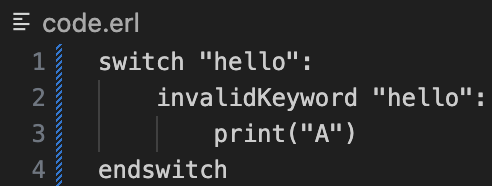
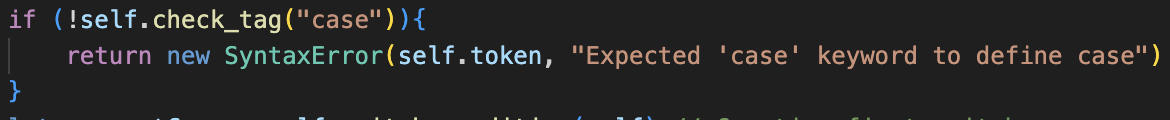
****Again, this will be me trying every single case and ensuring that I add the corresponding checks.

******🡺 Single “switch” keyword**Adding a null check before calling statement\_chain() in build\_switch\_statement() stops this crashing the program.

A black background with white text

Description automatically generated**🡺 No colon**Although the program accounts for tokens that are not colons after the expression, if there are no tokens at all it will crash, so a null check is needed.🡺 **No cases after switch statement**Currently, the program will crash as it tries to call continue() after advance\_line() if the program is empty, so a check that a line follows a switch declaration is added.



****🡺 **No check for “case” keyword**switch\_condition() does not check for the case keyword because the generic build method will already have checked for the keyword. However, this is not true for the first case in build\_switch\_statement(), so therefore an additional check that the tag is “case” must be added as currently the program will run regardless of the token.

**🡺 New error message**This is just to provide more detail about the error.  
**🡺 Default issues**The error handling was all in place, but the wrong identifier was being passed into the error, which has now been fixed.

# design: arrays

A screenshot of a computer program

Description automatically generatedThe main points on the spec show clear rules on how arrays operate, which is very different to python so will be an unfamiliar area for most students.

### in general

In ERL, arrays are completely static, so there is no append() or push() method, and they must be of a pre-defined size. It says clearly that they are 0 indexed, and only store a single data type. Additionally, it mentions multi-dimensional arrays, and these are typically rectangular, so ragged arrays: which are arrays that contain others of different lengths, should be disallowed. Accessing multiple dimensions uses commas in the same set of [], rather than other languages, like python, which uses separate [] for each dimension.

There are also two different syntaxes for creating arrays, one for initialising empty arrays, and one for building pre-defined arrays. Both initialisations will have to follow all the rules. In a summary, these are the main features that will have to be added:

|  |  |  |  |
| --- | --- | --- | --- |
| **Fixed size** | **0 indexed** | **Single data Type** | **Multi-dimensional** |
| **Initialising empty** | **Initialising with values** | **Read and Write** | **Rectangular** |

## creation

A screenshot of a computer

Description automatically generatedOverall, the BNF for the two array definitions are as follows:

These are a lot more complicated than prior BNFs, but when broken down into their methods it makes the process a lot simpler to decompose.

### create\_array()

This is the generic method for both types of array initialisation because they both share the same starting format: the “array” keyword, followed by an Identifier. After parsing this, it will check which token follows the Identifier: if it is a ‘[‘, then it is the format of creating empty arrays, and if it is an Equals token, then it will know to create and check for a pre-defined array.

It will also create the new instance of the ArrayType class to store the values.

### array type

Alongside having the typical position and line properties, ArrayType will also have:

🡺 contents: an array of the actual values that the array contains that is accessed when running  
🡺 astContents: an array of asts for the initial values of the array when it is defined in ERL  
🡺 identifier: an (optional) property for its identifier so that it can assign itself on evaluation  
🡺 arrayLength: an integer representing the length of the array  
🡺 type: the data type of the values in the array: stored in string form using the typeAsString property

🡺 evaluate()

The ArrayType is merging two separate features into one: alongside being a data type, it also assigns itself. The evaluate() method for arrays is not for changing its value or contents, but for initialising the array: representing the defining line of ERL rather than the type itself.

Therefore, it will evaluate() all the astContents, and save them into the contents after checking for errors.  
I will explain this in more detail later, but the initialising methods will all be changing the astContents, which will then be moved into the actual contents when the array is evaluated(). The array will also be assigned to the Identifier at the end of this evaluation so it can be accessed.

### create\_empty\_array()

create\_empty\_array() is the method for creating empty arrays, and it must be designed to work with any number of dimensions. Before being called by create\_array(), parse\_arguments\_or\_parameters() will be called if a ‘[‘ template keyword is detected, with ‘]’ being the closing keyword.

This will then return the array of the indexes. Now, create\_empty\_array() will be called, which will accept two arguments: the ArrayType and the list of arguments. This is one of the few Parser methods to not be passed self, because no further methods will be called from this point, except create\_empty\_array() itself, so self is no longer required.

The method itself will work recursively. If the array of values it receives is one item long, then it will fill the contents of the array it is given with null values, with the number of nulls being the value of the only index in that array.

When the method is called with multiple values for the indexes, then it is the equivalent of making an array with that many dimensions. Whenever the method is called with more values than 1, it will call create\_empty\_array() again, passing a newly created ArrayType, alongside all the values except the first one, and then will append the result to the current array, and will repeat for the value of the current index.

This will therefore construct the multi-dimensional array by recursively adding each dimensions and filling all parent arrays with new arrays, which should correctly function.

### create\_defined\_array()

Whenever

# development: arrays

## A black background with colorful text Description automatically generatedA screen shot of a computer program Description automatically generatedcreating empty arrays

I started by creating the new template keyword for “array”, and then the corresponding creates\_array() method which is called in parse().

A computer screen shot of a program code

Description automatically generatedThis method has the generic start for empty and filled arrays: ensuring that there is an identifier and then checking if it is a full or empty declaration. In this check, the ‘[‘ corresponds to creating an empty array.

This then calls the create\_empty\_array() method. I have changed my idea for the plan slightly and have changed it to fill with null instead of the custom object, which will be why later.

A screen shot of a computer program

Description automatically generatedOverall, though, the method follows the plan, recursively calling itself if it is multi-dimensional. The JavaScript shift() method is the equivalent of popping the first item from an array.

Then I fully implemented the array itself. Even though it is named ArrayType, it does not extend the DataType class as it does not share the same constructor.

The evaluate() method is similar to the one for functions: whenever evaluate() is called it is the equivalent of assigning the identifier itself, so no value is returned.

A black background with white text

Description automatically generatedHere it also sets the contents array to the evaluate() call of every item in astContents before returning, as these are the actual values. Here, any null fields are replaced by the custom nullValue object, because it is a lot easier to check for null than the object.

Initially when outputting the contents of the array, it displaced multiple undefined fields, but after changing evaluate() to return the array, it allowed for multiple dimensions to work with the recursion and therefore the correct structure was built.

It is a very long output so I will not show it, but the main ArrayType contained 3 ArrayTypes in contents, each with 4 nullValue objects.

## A computer screen with text Description automatically generatedcreating defined arrays

A screen shot of a computer program

Description automatically generatedNow the other check must be added to create\_array(), so that if there is an Equals, it will call the new create\_defined\_array() method after ensuring that there is a ‘[‘ tagged token before calling it,

Overall, this method is very similar to the method for parsing arguments, however it is modified to append each statement\_chain() result to the astContents of the array, and so that if a new array, signified by ‘[‘ is encountered, it will recursively call itself, with a new Array, pushing the result of that to the main array to allow for multiple dimensions to be created.

A screen shot of a computer

Description automatically generatedInitially, there were a few syntax errors with the method, but after fixing and printing out the produced array, it worked completely as expected, as shown.  
Full error handling will need to be added later, as now only valid inputs and a few main edge cases are available.

## getting items

A screen shot of a computer program

Description automatically generatedA screen shot of a computer screen

Description automatically generatedThe ArrayCall class is very similar to Call so is parsed in the same way.

Therefore, the end of factor() is changed so that it can produce both Calls and ArrayCalls and will produce the corresponding result depending on the type of bracket used.

Then, instead of calling call(), ArrayCall will call the get\_index() method.

A computer screen shot of text

Description automatically generatedThe overall idea of get\_indexes() is to ensure that all the indexes are Integers, and to return the corresponding item in contents.

If the array of indexes has multiple values, then it will ensure that the item is an array, and then will call get\_index() on that array, with the new indexes without the first value.

This should work, but my initial implementation was incorrect.

### A black background with white text Description automatically generateddebugging

A computer screen shot of text

Description automatically generatedA screen shot of a computer

Description automatically generatedInitially, there was some problems with the call parsing in factor(), but this was due to the differing names for argumentsAsts and indexes. For the sake of polymorphism, I could make these the same, but because they are very different things, I sacrificed the extra if statement for ease of reading.

Due to the completely broken get\_index() method, the program still crashed with tests.

The updates method is on the left, with the same overall structure, but different logic.

The format still follows the plan however, so there are no changes it is just that the first iteration had lots of confusion between what represented indexes and values and now this has been fixed.

### A black screen with white text Description automatically generatedA screen shot of a computer Description automatically generatedlength issues

A screen shot of a computer code

Description automatically generatedThe current issue is that create\_defined\_array() does not set the length property of the array, so therefore when trying to access a valid index this error will be incorrectly thrown.

A black background with white text

Description automatically generatedAfter adding length handling to the method, which will increment a local variable by 1 after each push, and at the end will set the arrays property to the total, the test case works, and the correct values are printed.

### longer test case

A screen shot of a computer

Description automatically generatedA computer screen with white text

Description automatically generatedWhen initially trying to run an example of a longer test case, which is more realistic of the GCSE questions, the program crashed with the more complicated input.

A screen shot of a computer program

Description automatically generatedThis ended up being because of the shift() method, and that when an item was being removed from the indexes parameter, because it pointed to the indexes property of the ArrayCall, it would also remove it from there. Therefore, this would only every work on the first iteration of referencing an array,

The fix is to change the evaluate() method of ArrayCall to evaluate all the items in index before passing them to get\_index(). Therefore, a new version of this evaluated array will be produced on every iteration so there will be no issues when the array is shifted.

A screen shot of a computer code

Description automatically generatedTherefore, get\_index() must be updated to no longer call evaluate() on each item, but overall, it is very similar.

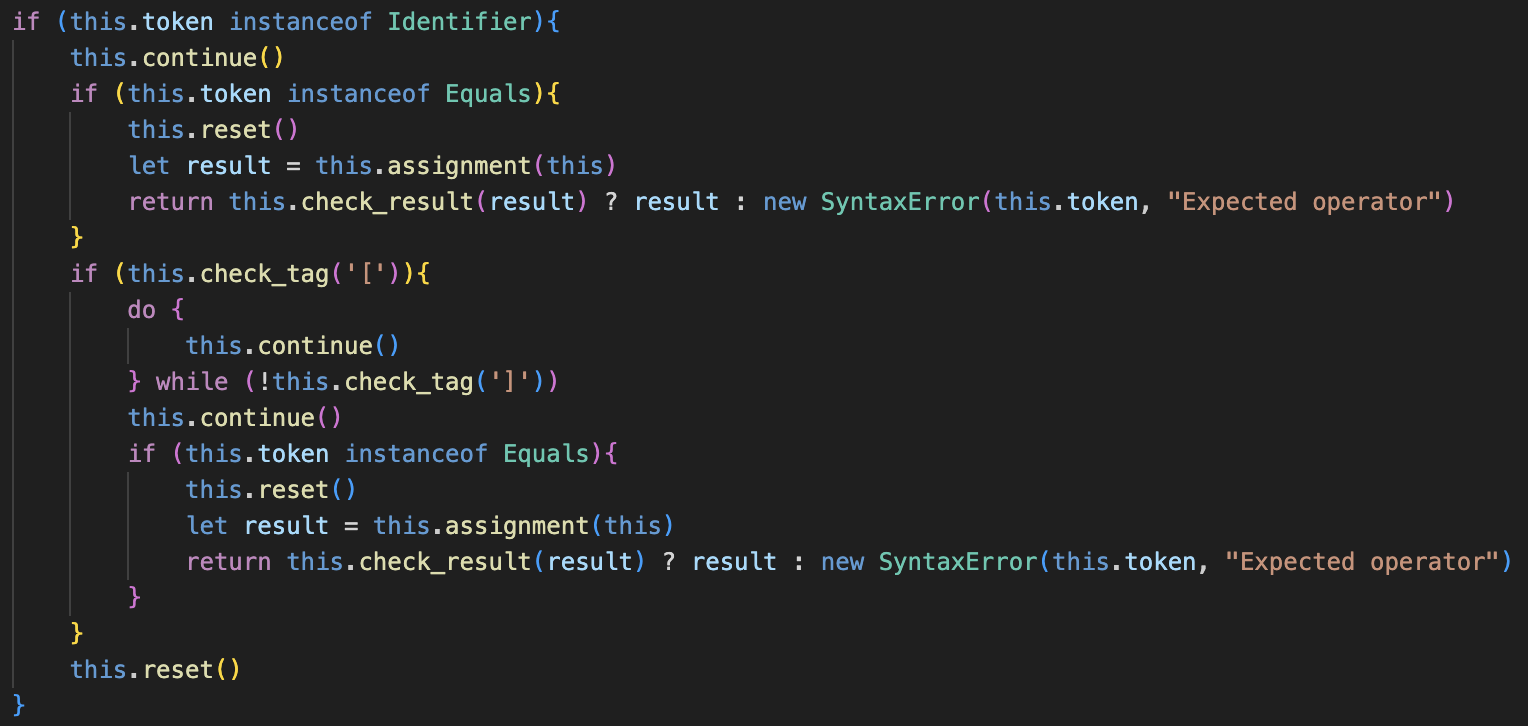
After this change, the correct output is now printed by the program, which shows how cohesive all the elements are.

Now, being able to update elements in an array needs to be added.

## updating array elements

The first change is to assignment(), which now calls factor() instead of checking for an Identifier. This therefore means that ArrayCalls can also be parsed so that their values can be updated.

This will also require the parse() method to be updated, because now the second token of the line will not always be an Equals in an assignment, and so the case for setting the value of array values must also be checked in parse().

This implementation is shown on the left, and if the first token is an Identifier and the token after is not an Equals, it will check if it is an ‘[‘.

If so, it will then continually iterate until the closing ‘]’ is found, and the token after it is checked if it is an Equals. If it is, then it is reset and assignment() called.

### setting values

A computer screen shot of text

Description automatically generatedCurrently, the Equals class will call set() on its left child, passing the value of its right. Because now ArrayCall’s can be their left child, it must be given a set() method to handle this.

As discussed in the plan, it will call evaluate() on itself, with the argument True. This is different to the plan, which said to pass the argument 1, but I feel it makes the functionality clearer in evaluate().

A screen shot of a computer program

Description automatically generatedOriginally, the value passed was meant to represent the number of indexes to ignore at the end, but because the program will only ignore 1 index at maximum, adding a boolean called ignoreLastIndex seems more logical as it more accurately represents the problem.

Now, if it is true, evaluate() will not add the last Index to the array of indexes passed to get\_index(). Therefore, this will return the array that the item belongs to.

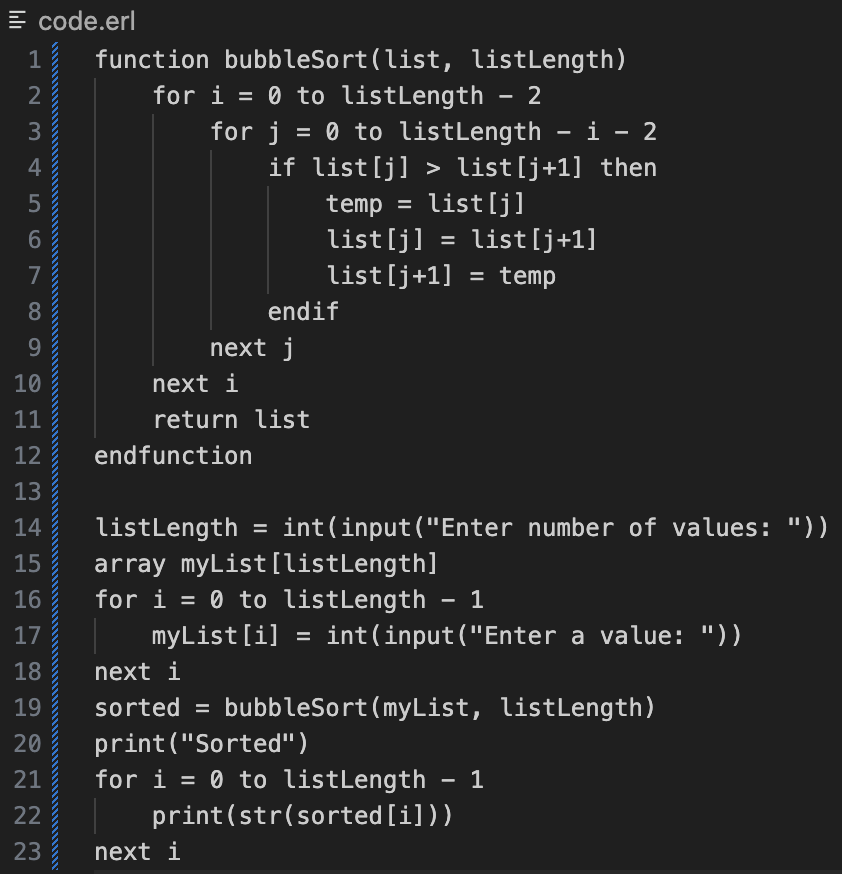
Therefore, the rest of set() will take the returned array, parsing the final index and checking it, and will change the item in the returned array at the last index’s position to the new value.

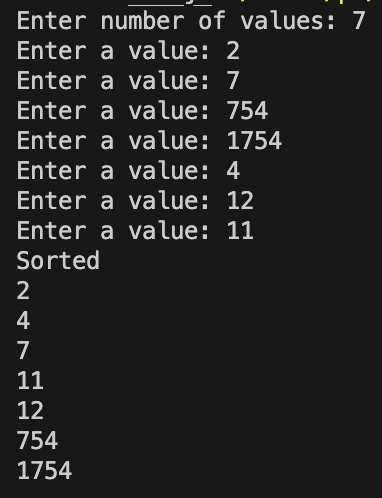
I have ensured that I include some error checking, such as ensuring the indexes are in range, however this is not complete and I will have to do proper testing at the end of this module.

### debugging

A black screen with white text

Description automatically generatedThe main issue is that if a 1D array’s value was being set, get\_index() would not function when it was passed an empty array of indexes, so an additional check to return the entire list when this is true, means that the 1D array is returned so the single index will be parsed and changed in the set() method.

There were also initially some smaller errors, such as me using the wrong values for ignoreLastIndex when using the for loop, but after these small issues were fixed it worked as intended.

The best showcase of this is the example of bubble sort as shown, which takes in the inputs from the user and will output them in the correct order.

This now really shows the functionality of the program, as adding arrays allows lots more complex problems to be implemented.

One main issue is that currently arrays cannot be printed, and this is also an issue with the other data types, which means the items must be iterated through to be outputted.

A black background with white text

Description automatically generatedI think my decision to only allow strings to be printed should potentially be rethought, and I will discuss this with Tanish.

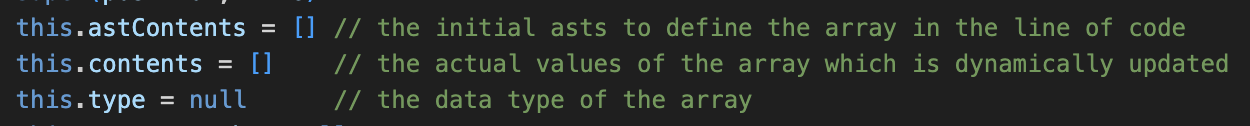
A black screen with white text

Description automatically generatedWhen testing if 2D arrays worked, the first test case completely to updated, which was strange.

  
When looking into the ArrayType’s ast, it showed the new value was being added completely, as a new property rather than changing the contents. This ended up being a simple fix because I forgot to change the contents property of the ArrayType, and instead accessed its properties.

This change fixed the issue; however, I have no idea how the first implementation managed to somehow function with completely incorrect code, but the values are correctly updated.

## strict type handling

Now the type property will be given a use and will be a string that represents the data type of the array. This will be used in the evaluate() method of the array, for when values are initially assigned, as well as when an item is updated in the array.

A screen shot of a computer program

Description automatically generatedThe evaluate() method is very easy to update and so long as the item is not null (for when an empty array is being defined), it will ensure the type aligns with the type property, and if it has not yet been defined it will set the type to the type of the current item. These are all stored as strings as it is easier to use typeAsString.  
The test case correctly returned an error in this case.

A black background with text

Description automatically generatedLikewise, the set() method of ArrayCall is also updated with a very similar syntax and performs the same checks to ensure that the types are maintained.

### debugging and testing

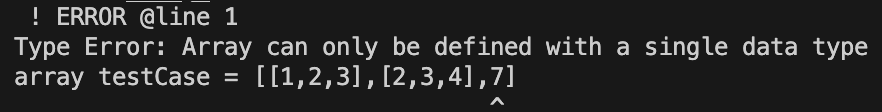
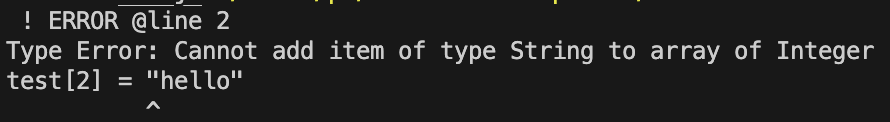
A black screen with white text

Description automatically generatedWhen running a test case, it turned out that one dimensional arrays were now broken, and it was because, in the case of a 1D array, get\_index() would return the contents of the array, rather than the ArrayType itself.

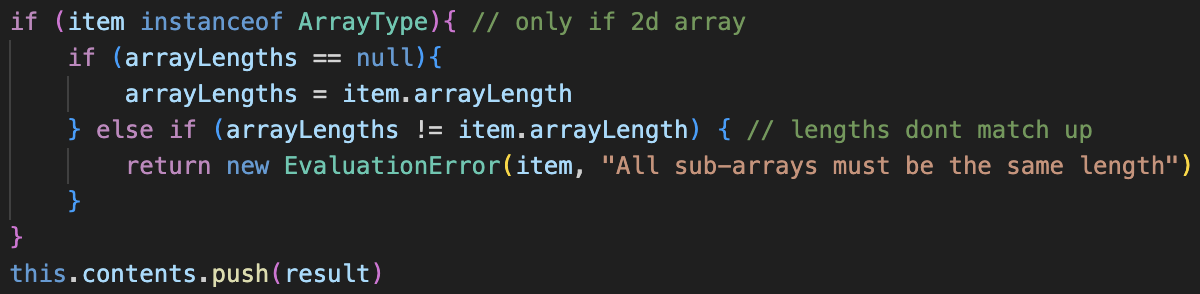
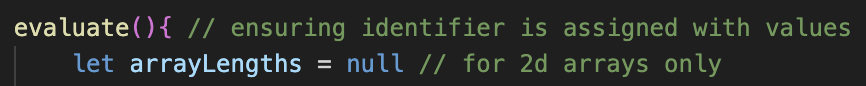
A black background with white text

Description automatically generatedThis is an easy fix, but it explains why the test case for bubble sort worked earlier: as two different errors coincided for the system to correctly work. Now arrays of all dimensions will work as anticipated.

All the different test cases from here worked as expected, including more complex examples with multiple dimensions.



### enforcing rectangular arrays



A black background with white text

Description automatically generatedAn additional check to the evaluate() method for assigning arrays uses the arrayLengths variable to store the length of the first sub-array, and if any of the following do not have the same length then the error will be returned.  
  
The test case for this works, and as the sizes of arrays cannot be changed after declaration, it ensures they will always remain rectangular.

## issues with empty arrays

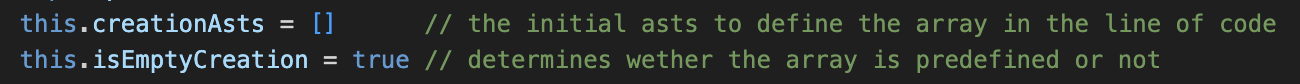
One issue I just considered is to do with my create\_empty\_array() method, and that I have designed it to only work with single Integers, by reading its value and then creating the array of that value. However, this does not correlate with the initial plan for the system:

A black screen with white text

Description automatically generatedCurrently the array is created in the parser, but for it to work with expressions and variables, the code will have to be moved from the parser to the Interpreter. This was a very big oversight; however, a lot of the creation code should remain the same, but with a lot of moving around of components: from out of the Lexer into separate classes.

Currently, when any expression that is not a single Integer is passed into the declaration, there will be no limit on the length of the list, and values at any location can be set, most likely because it is being used as a dictionary instead of an actual array, so a fix is neccecary.

### redesign

To move the empty array creation from the parser to the Lexer, the new isEmptyCreation property will be a boolean to determine which type of creation the array was defined with.

A screen shot of a computer

Description automatically generatedThe plan is to use the same array – previously astContents – for both types of asts, as now the asts for the indexes for empty creations will be stored in the array, so therefore I am renaming the array creationAsts so the naming better corresponds for both types.

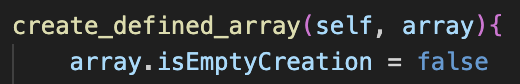
A screen shot of a computer program

Description automatically generatedevaluate() will now call either evaluate\_empty\_array() or evaluate\_defined\_array() depending on the type, as well as checking for errors and assigning an identifier if need be.

evaluate\_defined\_array() is essentially the old evaluate() method, but without the null checks as they are no longer required because there will be no empty definitions in this method.

Apart from that, it is an identical copy and paste, but with some renaming for the new names.

Additionally, create\_defined\_array() in the Parser must also be changed to set the isEmptyCreation property to false.

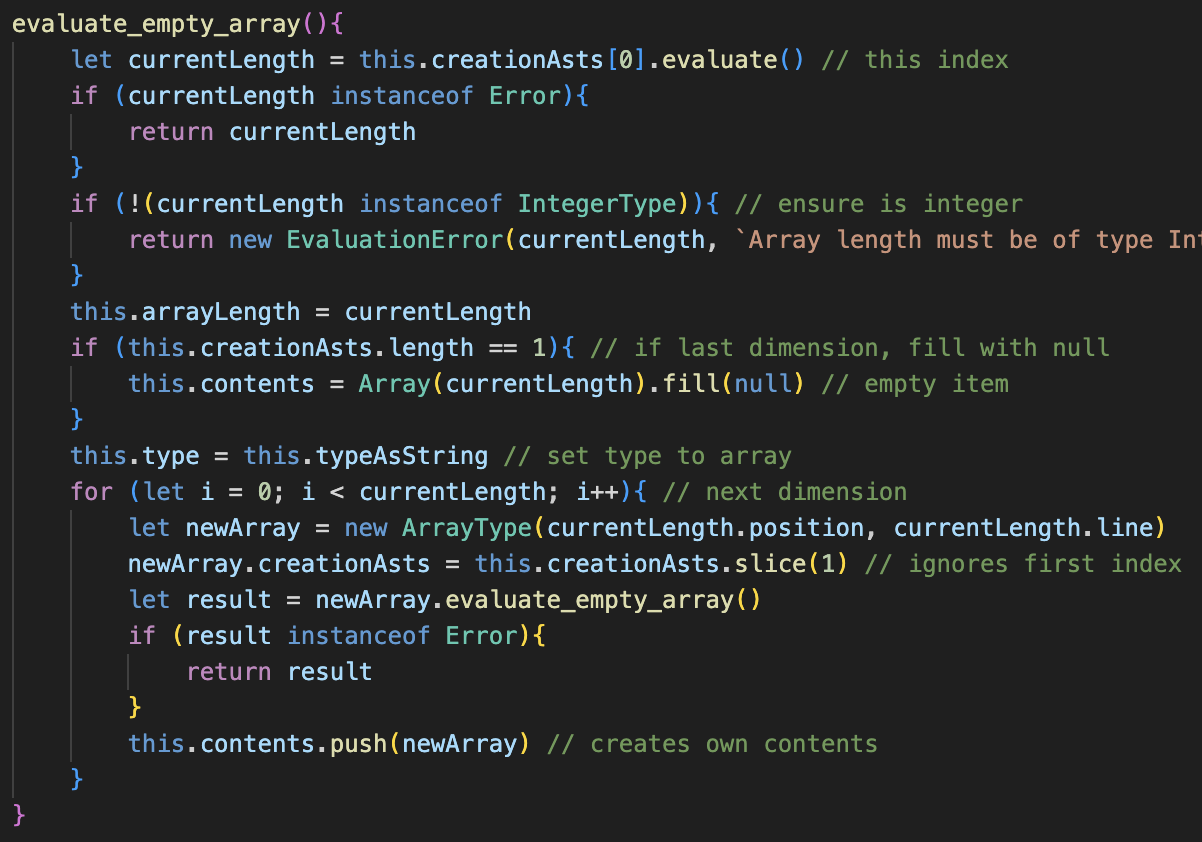


### empty arrays

A computer screen with text

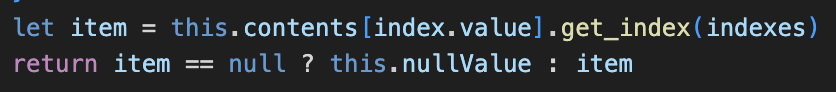
Description automatically generatedNow the create\_empty\_array() method in the parser will be deleted as it will be moved to the ArrayType class instead. However, this requires create\_array() to change.

Now, when an empty array is being created, it will just set the creationAsts array to the array of indexes: so these are not the actual values of the array, but will be the dimensions that will be used to create the empty array.

The new method is very similar to the parser but starts by evaluating the first index. If that is the only index in the array, it will fill it with null values to represent empty locations.

Otherwise, it will fill itself with other Arrays, set their creationAsts to its own but without the first index, then evaluate()s them to repeat the process, and then will full its contents with these evaluated arrays.

That is the main idea, but other aspects include setting the arrayLength property to the value of the index, as well as ensuring that the index is an Integer.

get\_index() also changed, so that it returns the custom nullValue there if there is a null item, instead of filling contents with them.

### testing new system

Initially when running this, it still did not work as intended. After checking through the code this was because arrayLength needed to be a JavaScript number, and I was setting it to my custom IntegerType. So, the value must first be extracted before being used.

Secondly, there was a slight typo in ArrayCall’s set method, which needed to be updated, as it compared the index to be set to the array object instead of its length, so the error would not be flagged, so this needs to be changed.

A black background with white text

Description automatically generatedNow, the proper error message is shown and as shown by the message, the length correctly corresponds with the variable that was passed into it, so this shows that it is being correctly updated.

A black background with white text

Description automatically generatedAdditionally, the filling was slightly broken when testing, as it did not reference the value of currentLength – which I’ve changed, as well as changing it to fill with the empty value instead of null, as the values are not only accessed with the change in get\_index() and this makes it a lot easier.

Now, the error messages are properly updated to show when an empty location is outputted.

A black background with white text

Description automatically generatedA screenshot of a computer program

Description automatically generatedSome further examples of testing show how the system fully works with multiple locations.

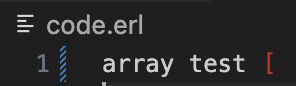
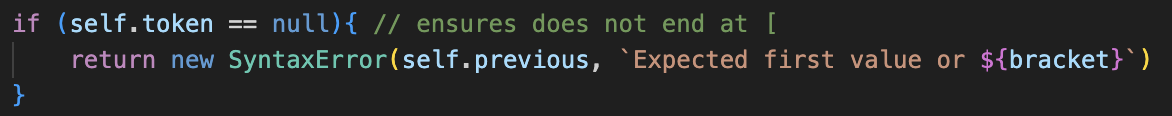
## all invalid cases

Now for the long process of trying every single possible case and adding in a check for each of them to fix them. Because there’s so many, I will only be showing the test case and the fix, and not the error messages.

A black background with blue and white text

Description automatically generatedA black and white text

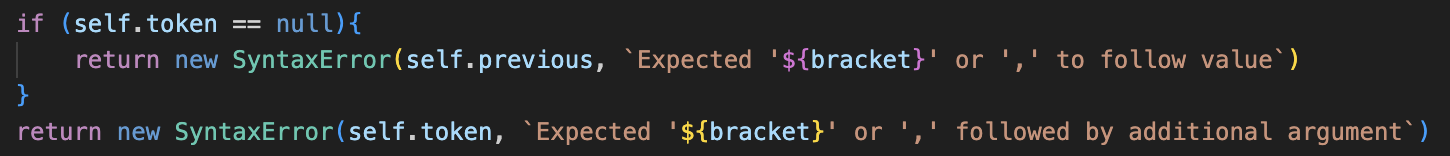
Description automatically generated**🡺 Nothing following identifier.**

**🡺 Single bracket after identifier**

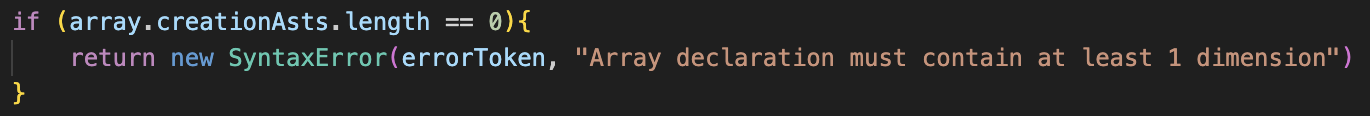
A black background with white text

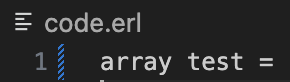
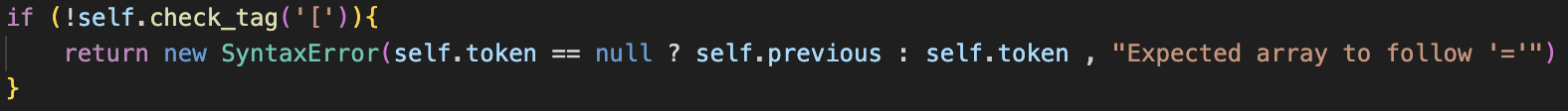
Description automatically generated**🡺 Unclosed, single dimension.**

A black background with white text

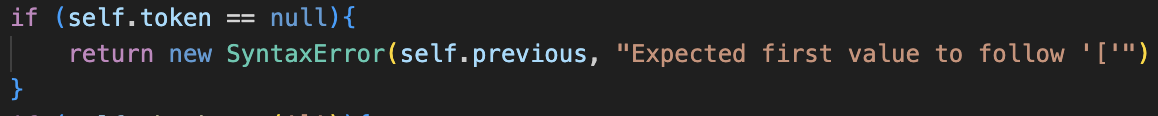
Description automatically generated🡺 **Unclosed, many dimensions.**

A black background with white text

Description automatically generated**🡺 No dimensions**

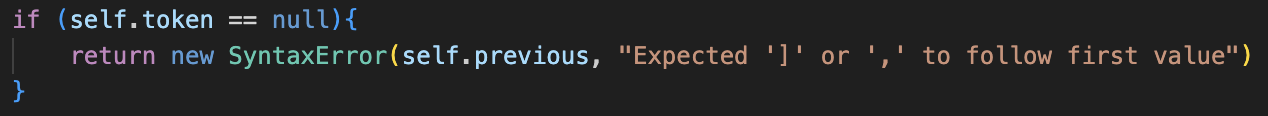


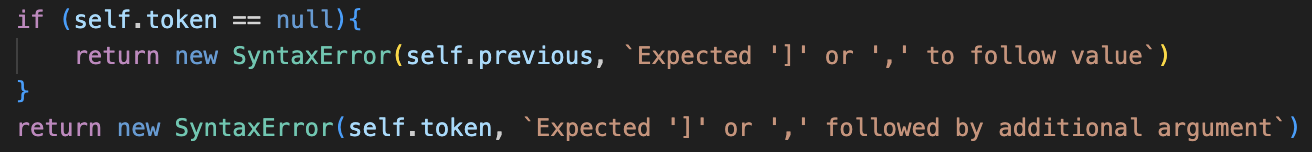
🡺 **Nothing after equals**

A black background with white text

Description automatically generated**🡺 Single bracket after equals**

A black background with white text

Description automatically generated**🡺 Single, unclosed value**



A black background with white text

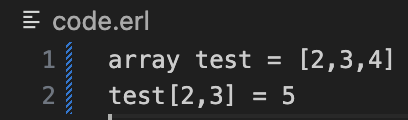
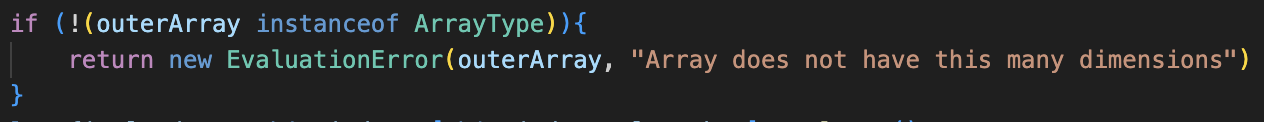
Description automatically generated**🡺 Multiple, unclosed values**

A black background with white text

Description automatically generatedA black background with text

Description automatically generatedA black background with white text

Description automatically generated**🡺 Setting undefined array**This was already implemented for accessing values, but now a check for errors needs to be added to set() in ArrayCall.  
🡺 **Passing too many dimensions.**



**🡺 Updated error message when accessing from a non-array.**

Overall, these checks were very easy to add, as a lot of the checks had been added for when an unexpected token occurs, but not when there were no more tokens are remaining, so it’s a lot of adding checks for null tokens and using the previous property.

## printing update

This is more for quality of life, but when discussing and showing my stakeholder Tanish the system, he found it very annoying that only strings could be printed. Although my original intention was to ensure students stayed aware of type casting, he said it was more of a pain than a help: as it just resulted in having to go back and add lots of str() conversions at every print statement, which just took a lot of time.

Additionally, when checking through some past papers for the exam, they never awarded any marks for type casting an output: only for inputs, so therefore it was not completely needed. Therefore, I am going to add printing for all the other data types to make the overall experience better.

This shouldn’t be that complicated overall, as the system uses display() methods to get the string version of each of the data types, as these were used in the earlier modules when print() did not exist.

A black background with blue and orange text

Description automatically generatedTherefore, the Print classes code can just call display() if any of these types are encountered. Therefore, all DataTypes will be added to the NOT check to return an error in Print’s call statement()

The two other items I would like to print are arrays, and functions.

### arrays

A screen shot of a computer screen

Description automatically generatedArrayType is now also given a display() method, which will call the display() method on all its item, and return a string with them inside square brackets, and separated by commas, like in a list.

**A screenshot of a computer

Description automatically generated**The additions I have added are to represent empty values by <Empty>, in case an empty array is outputted. Strings will also be surrounded by quotations for consistency, and 2D arrays will gain some additional formatting to make them more readable: separating each sub-array on a separate line.

Some examples are shown on the left, and it is a lot clearer than outputting items of an array line by line for checking.

### functions

The only reason I feel being able to print functions is needed is if an array contains functions as values: as currently functions can be stored in them and called, but if the array is outputted, they need some form of representation so that the array can be displayed.

A black background with blue and orange text

Description automatically generatedThe solution is using the same <> format as I did with empty values, as I have decided this is how I will represent values that should not really be outputted. From here, it is as simple as giving each subroutine a display() method with their name to identify them.

A black background with orange text

Description automatically generatedFor user defined subroutines, I am going to use the name of the identifier in the display method for this identification.

A screen shot of a computer program

Description automatically generatedFor native subroutines, as well as declaring them as native, I will just write their name, but this does not work for type casts.

The type casting function is slightly more complex, as it has a tag to represent which type to cast to.

**A screen shot of a computer program

Description automatically generated**This just needs a switch statement within the getter, which will return the corresponding value. This technically means that if the user uses the real() function, it will display as float, but I feel like that is such a minor issue that it is not worth redesigning the system for, and either way it clearly shows that they are the same function.

As shown, now there is at least some form of representation for when functions are printed.

This also means that any errors will be easier to diagnose if a student is outputting values for debugging.