strings and types

# design

This module will consist of 3 sections: Implementing strings and concatenation, adding type casting between different types, and stricter type handling.

### text string

Because the keyword String is reserved by JavaScript, the new class will be called TextString. This will extend Token with position, line, and value properties. It’s display() method will just be to return its value.

🡺 make\_string()

When a quotation mark: “ or ‘ is encountered in make\_tokens\_line(), this method will be called. It will repeatedly add characters to an array until the same type of quotation that it started with is encountered, stored in a variable called quotationMark. Then a new TextString instance will be returned with the array converted to a string and passed into the value property.

## binary operators

These have always contained a lot of repeated code, and this updated significantly reduces this. It also removes check\_for\_errors() and check\_for\_float(), as they become integrated within the new system.

A screenshot of a computer

Description automatically generated🡺 evaluate()

When calling evaluate() on a Binary Operator subclass, there will only be a single evaluate() method defined in Binary Operator, rather than a different one in each subclass.

This method will not return the value itself but will do the generic checks for every evaluate() method, and will evaluate the two children, returning an error if one occurs.

It will set the results of evaluating the children to two new properties: leftValue and rightValue and will then call the new calculate() method, which will use these values.

🡺 calculate()

A diagram of a program

Description automatically generatedThis will be a class specific method, so there will be no generic instance in Binary Operator, which will be called by evaluate() when leftValue and rightValue have been set. The purpose of this method is to go through the possible valid combinations of types, returning the corresponding type.

For example, the diagram shown on the right shows all the possible combinations of the leftValue and rightValue and what they should return.

calculate() will check which case the current values are and will return the corresponding type: this is replacing the old check\_for\_float() as this method will lay out the valid cases for each. If the type of combination is not valid, an error is returned.

Add is the most complex example for this because it can now be used for concatenation and arithmetic, however most calculate() methods should be relatively simple.

Any additional checks that need to occur, such as the leftValue not being zero, will also occur in calculate(). Finally, although this returns the types, to set the value of the types I will be using a method called get\_value(), to not have to repeat the same calculation in each type constructor.

This will just return the JavaScript data type of the operation on leftValue and rightValue, with one for each operator, and will be called and passed as an argument to the constructors.

### type checking methods

To make it easier to undergo all these checks, I am going to introduce the following methods which should make it easier to check each case with less comparisons, such as by grouping all of the cases that will return a Float into a single method that accounts for them all.

🡺 strict\_type\_check()

This will take in two parameters and will check if leftValue is an instance of the first parameter, and if rightValue is an instance of the second. If and only if this is true, it will return true, otherwise false will be returned. This is used to check that the two types are the same as the parameters, in that order.

Using Add, this could be called with the arguments being two Integers, which would contain an Integer return because it is the only case that will return an Integer value.

🡺 loose\_type\_check()

This accepts any number of arguments and ensures that the leftValue and rightValue are both instances of any of the arguments, and there is no limit on how many arguments can be passed. leftValue and rightValue can be any combination, so long as they are both at least one of the types passed.

🡺 contains\_type()

This also takes a dynamic number of arguments, but in this case if either LeftValue or RightValue are instances of any then true is returned: they do not both have to be, which is what separates it from loose\_type\_check().

The use case of these last two are checking for when a Float value is returned, so if contains\_type(Float) and loose\_type\_check(Integer, Float) are both True, then a Float value can be returned, because there is at least one Float value and they can also be combined with Integers.

I feel that these 3 type checks are enough to cover all the cases that are required. I will not plan the checks for every single operator, but using these is a very logical approach and it will be easy to use the corresponding checks for each type.

### type error

I did slightly introduce this at the end of the last stage of development whilst ensuring all the for-loop values are integers, but the main error type for this section is the TypeError, which accepts a token like SyntaxError. All of the errors returned in calculate() because of invalid types will be this new TypeError, and I aim to make the messages descriptive of the intended types.

Additionally, I will introduce new type checks into the different structures, ensuring that all the conditions for if statements and loops are Booleans, returning a type error if they aren’t.

This also includes aspects like comparisons too, so if a greater than is used on two booleans, for example, I would like the program to return an error because Booleans are not “greater” then each other.

## type casting

A white rectangular frame with blue border

Description automatically generatedThe specification shows that there are multiple ways that data can be type casted: between Integers and Floats, between Strings and Integers/Floats, and between Strings that say “True” or “False” to Booleans. However, the specification is not exactly clear on what should be allowed, so I am going to create my own logical set of rules, which can be edited later.

### Rules

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Starting: | To Integer: | To Float: | To Boolean: | To String: |
| Integer |  | value is the same | 0 🡺 false Else: true | value as String |
| Float | value is truncated |  | 0.0 🡺 false Else: true | value as String |
| Boolean | true 🡺 1 false 🡺 0 | true 🡺 1.0 false 🡺 0.0 |  | true 🡺 “True” false 🡺 “False” |
| String | First check if possible - only digits  Yes: value as Number No: TypeError | First check if possible - only digits and “.”  Yes: value as Number No: TypeError | “True” 🡺 true “False” 🡺 false Else: TypeError |  |

My final table for how the different types will cast to each other is as shown:

Initially, I was intending to now allow for type casting between numbers and booleans, because it could be potentially confusing to students as it is not very intuitive.

However, after discussing with my stakeholder Tanish, we concluded that it was important for children to understand the correlation between 0/1 to False/True, and therefore decided to keep that conversion. I could have also made it so any number that aren’t 0 and 1 would cause an error. This could have maybe conveyed the idea that 1 represents True more clearly, but because most existing high-level language convert any non-zero number to True, I decided to keep with convention.

A white rectangular frame with blue border

Description automatically generatedThe one main difference between this and typical conversion is from String to Boolean. In typical languages, any non-empty string returned True, and only “” returned False. However, I thought a student attempting to cast bool(“False”) and receiving True would be extremely confusing.

Furthermore, the specification gives the example of bool(“True”) as a conversion, implying that it is the names of “True” and “False” that should be converted to Booleans, so I have made them do so and every other string will return an error. This is not standard, however the way I implement them should make the solution very easy to change later if I decide to.

🡺 cast\_to\_type()

This method will belong to the data types and will take in an argument for the data type to convert to. It will then return the corresponding result, based on the instructions in the table. For now, this will only be accessible as a method by manual testing, as functions have not yet been implemented, but it should be easy to integrate in the future when they are implemented.

# development

### string class

A black screen with white text

Description automatically generatedA screen shot of a computer code

Description automatically generatedA screen shot of a computer code

Description automatically generatedA black background with text

Description automatically generatedThis follows the plan and is very simple to add.

The test case shows the Lexer correctly handles strings.

### A screen shot of a computer program Description automatically generatedA screen shot of a computer code Description automatically generatednew binary operator class

These are replicas from the plan in code form. For now, there is not an easy way to test them, so I will first implement the calculate() method for the Add class, and then test that to ensure that it is working.

## addition

A screen shot of a computer code

Description automatically generatedI have decided to go with the following calulate() method:

A computer code with colorful text

Description automatically generatedThis goes through all the possible cases: returning a Float and Integer and TextString. The 3 different type checking methods have all been used, and it is clear how they will be used for the others.

A black screen with white text

Description automatically generatedBecause I deleted the check\_for\_errors() and check\_for\_floats() methods in this process, the program didn’t run because they were called when not existing. After removing these from the other classes, I gave Float a new check for the display() method.

A screenshot of a computer

Description automatically generatedA screenshot of a cell phone

Description automatically generatedI also changed factor() in the Parser to allow TextString as a new factor, so that concatenation could therefore be parsed.

As shown, the 3 different test cases worked, however currently if an invalid combination occurs then the program will break.

### naming update

A screen shot of a computer code

Description automatically generatedA computer code with text

Description automatically generated with medium confidenceI have decided to rename the old DataType class to Symbol: not only does it make more sense as it is contained within the Symbol Table, it allows for the different actual DataType classes to all extend a new class called DataType.

A black background with white text

Description automatically generatedThis greatly reduces the amount of repeated code, and allows for easier checking throughout, such as in factor()

I also added lots of titles, as shown on the left, throughout the source code, as it is getting very large and this makes it easier to manage.

This change to DataType will also allow half\_statement() and potentially other methods to be simpler in the future: there will now be no Data Type checking in the Parser due to how Identifiers cause issues, so now all checking will occur in the calculate() methods. This means that Booleans can now be expressions, however if they are used like a number then the error will be thrown in calculate() rather than in the Parser, simplifying the statement code.

### incompatable data types

Firstly, I have given each DataType a method type\_to\_string(), which returns the string format of their type to be called in an Error.

A computer screen shot of text

Description automatically generatedThen, as the else case of calculate(), I have created different cases for the different initial Data Types. This could all be the same error message, but I have decided to specialise them to provide better error messages. In these cases the left value is treated as the “correct” data type.

A black screen with white text

Description automatically generatedA black screen with white text

Description automatically generatedSome examples of these error messages are shown, and they are very clear as to what the issue is, which is important when it is Identifiers that are involved in these errors and their data type is not clear. The else case is also very important, as there must be a catch in calculate(), to return an error if two unaccounted datatypes are merged.

## other operators

A screen shot of a computer program

Description automatically generatedNow the new calculate() and get\_result() methods must be written for all other Binary Operators. The rest of the arithmetic operators are very simple: with less error messages and cases than addition due to concatenation not being a thing. The best example of this is the Minus class:

A screen shot of a computer code

Description automatically generatedThe most complex arithmetic example is Divide, requiring the check that the right-hand side is not zero, and because two Integers can produce a Float result, an extra check is needed in the Integer, Integer strict check to ensure a Float is returned if it is a float value by checking for full stops.

A screen shot of a computer program

Description automatically generatedThe exponent class also requires a check for NaN to ensure no imaginary numbers are created, but apart from that the other Arithmetic operators are very logical.

Equals is the only Binary Operator to completely redefine evaluate(), because it must call assign() on the leftValue instead of the default evaluate(). This leads to a slight bit of repeated code; however, it is acceptable in this context.

Next is the logical operators, which have the simplest checks: requiring that both sides are Booleans, therefore using the strict check. The code for And is on the next page, but Or uses the exact same code, with an updated error message and different get\_result() method.

A screen shot of a computer code

Description automatically generatedFinally, there is the Logical Operator check. The get\_result() method is very useful here, as it makes the code a lot easier to understand. I have decided to allow Integers or Floats to be compared in any combination, as well as just two strings, but not combinations of numbers and strings.

A screen shot of a computer program

Description automatically generatedFor Booleans, I have decided to only allow == and != to be compared. Technically > and < can be used in other languages, however it is a useless feature so I have decided to remove this ability.

I ran a few quick test cases for each operator, not testing all the errors but ensuring that the expected results are working as expected, and they seem to be. I will test with more depth later in the process.

### structure data type cheking

A black background with colorful text

Description automatically generatedA screenshot of a computer

Description automatically generatedA black screen with white text

Description automatically generatedNow If Statement and the Loops need type checking to ensure that their conditions are Booleans. Currently, the program crashes if there is no correct case in an IfStatement, but this was updated by adding a null check in evaluate\_many\_asts(), because IfStatement will return null if there is no valid case.

This fixed the crash; however, we want the Interpreter to return an error if a condition is not a Boolean value.

A screen shot of a computer code

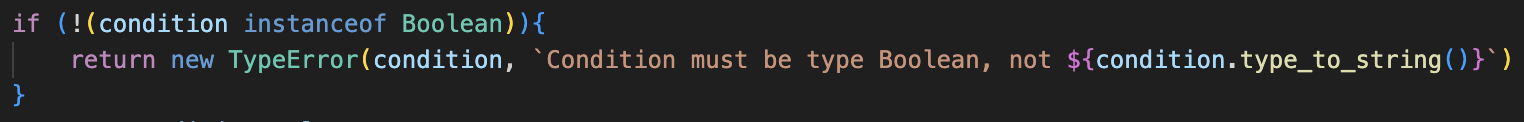
Description automatically generatedA black background with yellow and red text

Description automatically generatedThe code for this is as shown: This is an easy check in IfStatment’s evaluate() method, and if the result is not a Boolean datatype, then a Type Error will be returned.

A black background with white text

Description automatically generatedA black background with white text

Description automatically generatedThe Interpreter also must be updated, because currently it could not deal with errors being returned from IfStatement’s evaluate(), and the simple check added now deals with that.

While and Do are also given the same check in their evaluate\_condition() methods.

The correct error is returned.

That concludes all the Type handling in this program.

### cleaning up

A screen shot of a computer

Description automatically generatedI am not a fan of the name TextString, and it turns out that the Boolean class I have been using is also a reserved keyword by JavaScript, so if I wanted to use any of its methods, I would not be able to. Therefore, I am renaming the datatypes to IntegerType, FoatType, BooleanType and StringType. This provides consistency between them all and is easy to change with the replace feature in my IDE.

A black background with white text

Description automatically generatedA black background with white text

Description automatically generatedBy removing the Boolean check in half\_statement, which was already flawed due to potential Identifiers with a Boolean data type, the code is significantly simpler, as well as the error message being much more representative of the error that has occurred.

## A screen shot of a computer program Description automatically generatedA screen shot of a computer program Description automatically generatedA screen shot of a computer code Description automatically generatedtype casting

These are relatively easy to add, just copying what the table in the design stage was explaining to do.

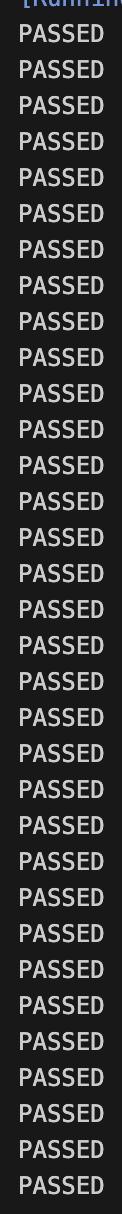
Not fully sure if they work yet, will need to do a lot of test cases afterwards.

A computer screen shot of a program code

Description automatically generatedThe one for strings is the longest as there are a lot more cases to account for.

### testing

A screen shot of a computer program

Description automatically generatedA screen shot of a computer code

Description automatically generatedI wrote a quick algorithm to iterate through a 2D array, which, for each item, would create a new data type from the class at index 0, with the value at index 1, would typecast it to index 2, and the expected result would be at index 3. Then, I made it print pass or fail if the expected result was produced.

The output showed that the program printed PASSED for every single test case, and I tried to include every single possible combination in my array of input data, so therefore I am confident that the implementation has worked.

This therefore concludes this section on strings and typecasting, and overall I feel a lot more confident with the program now that the types are stricter and that a lot of the repeated code has been eliminated.

# evaluation

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Implemented |
| 6.1 | Add a string data type, which can be defined with single or double quotations, and concatenated with + | **Yes** |
| 6.2 | Ensure that the whole system has tighter type handling, as now that operators have different uses across different types, different cases will be required so that invalid cases are rejected | **Yes** |
| 6.3 | Allow for type casting between the 4 primitive datatypes, with errors in some illegal cases | **Yes** |

The success criteria for this section are as follows:

I am very happy with this module, as all the new features have been implemented correctly, but also most of the pre-existing main issues with the program have been fixed: lots of repeated evaluate() code has been removed, issues with conflicting data types have been replaced with checks and errors, and a lot of bad naming has been changed to make the system a lot more clear.

This is a good place to be finishing this half of development with. The main issue currently is still the global variables not being encapsulated, mainly the global symbol table, as well as the currentText system for error messages, but these should be easy to change later in development.