native functions

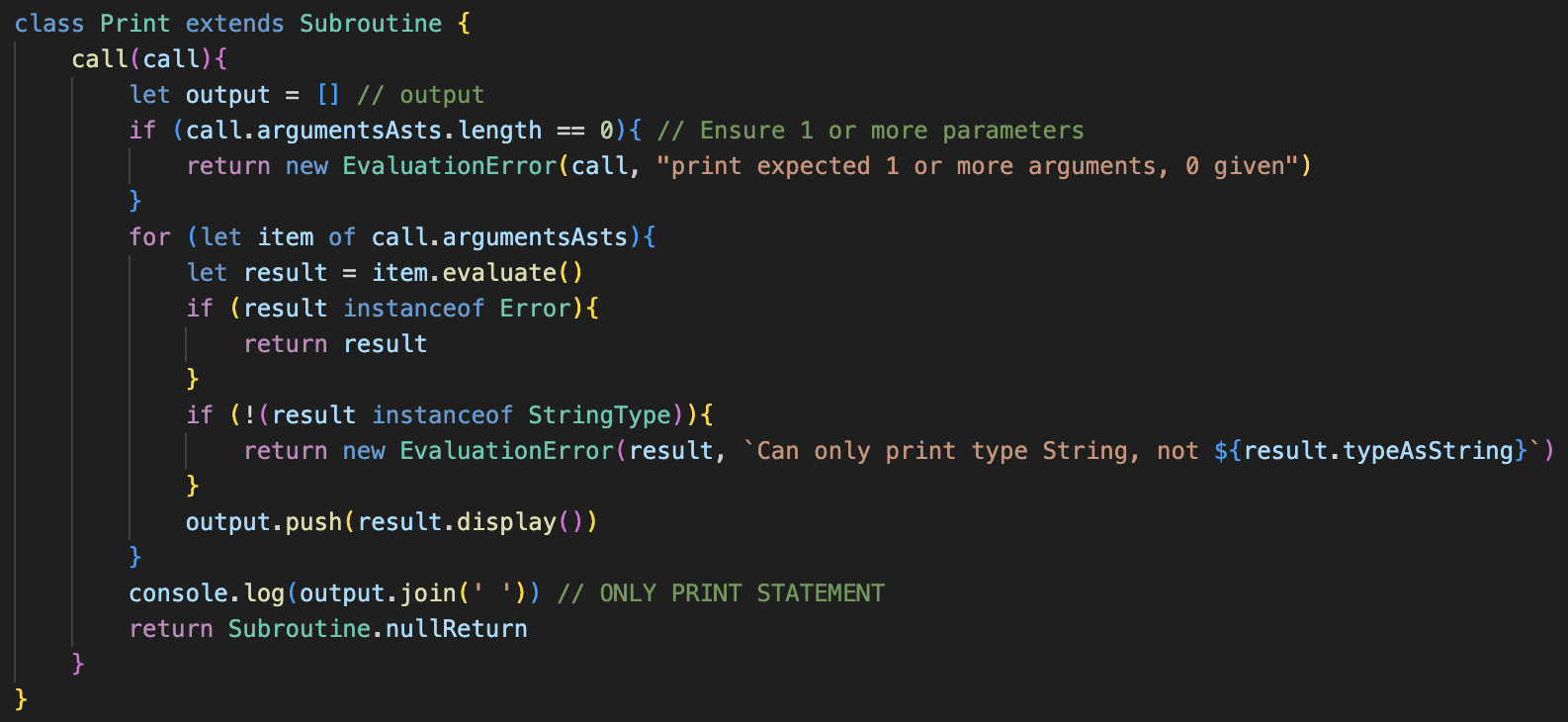
# design

# development

## print

A screen shot of a computer program

Description automatically generatedThe first step was to spit the existing Subroutine class into the basic Subroutine class, with all of the features being transferred into the UserDefinedSubroutine class, which will represent any non-native functions.

A lot of the other code had to be changed, mainly in the parser, to reflect this change, however in evaluation all Subroutines will be treated the same.  
Therefore, calls will check that they are calling any subroutine, regardless of the type.

Now the new Print class extends subroutine, containing a call(argument) to parse the print statement.

It ensures there is at least one argument and will otherwise return an error. Then, for each argument, it will evaluate() it, ensure it is a String, and will join the strings together with spaces and output it.

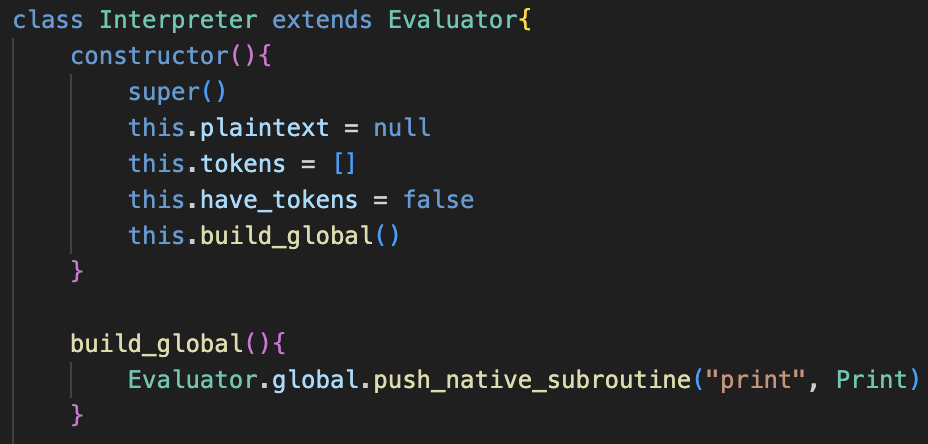
It will then return the same empty subroutine return that was previously used, which is now a class property of subroutine to reduce code repetition.

### changes to pushing system

Instead of the original system, I have decided to use a new push\_native\_subroutine() method to use abstraction, which now accepts the name for the identifier and the class of the subroutine.

A computer screen with text

Description automatically generatedThis is much better than the old system, as it removes a lot of the additional steps that will be added to build\_global() to allow it to seem a lot simpler.

Identifier is also changed to allow the constant property to be added, which is default at false, so therefore in the new method the native functions can be set to constants by default so that they cannot be edited.

build\_global() is now called in the constructor of the Interpreter class, and now the chain of calls will be listed in sequence inside this class, as shown with the arguments being the name “print” with the corresponding class.

A black background with white text

Description automatically generatedA black background with white text

Description automatically generatedA black background with white text

Description automatically generatedFinally, the output statement in evaluate\_single\_ast() is removed so that not every single calculation is outputted (finally)

Examples:

## input

A screen shot of a computer program

Description automatically generatedThis call() method for Input ensures only 0 or 1 inputs are allowed.

A black background with white text

Description automatically generatedIt will output the only argument if it is a String, before using the previous prompt() system to get an input from the user.

A black background with white text

Description automatically generatedA black screen with white text

Description automatically generatedNote that this is still a temporary solution, as when a proper interface is added the prompt() implementation will need to be replaced, but for now this system should work.

Then, after adding this input to build\_global(), it could be tested. It did mainly work; however, I would like the input to be on the same line as the message.

When researching, a solution for outputting without bringing a new line was using process.stdout.write, however when using this, the prompt() method would remove the output when entering an input.

A black background with white text

Description automatically generatedA black background with text

Description automatically generatedTherefore, I decided to pass the string that should have been inputted into the prompt parameter, set to a string that is empty “” by default. I do not love this, however it is acceptable because prompt() is temporary.

Now, this input is on the same line so the input system is complete.

## random

### generating random numbers

A computer screen with numbers and symbols

Description automatically generatedA black background with text

Description automatically generatedBecause ERL using inclusive random number generation, and because JavaScript’s random() only gives a number between 0 and 1, I decided upon using these two algorithms to generate inclusive random numbers.

I used each of these functions 10,000,000 times and took the average value to ensure they aligned with the expected value. In this case, because I am generating numbers from 1 to 9, the expected value is 5, which the integer random number generator was very close to, so therefore it is working as expected.

A black background with white text

Description automatically generatedHowever, the float **value** was significantly off, at 5.5, so the float random was incorrect.

The solution was to remove the +1, which is only needed for generating Integer random numbers, which then produced a value close to 5.

Technically, because Math.random() produces a value x where 0 <= x < 1, the algorithm for generating float numbers is not purely inclusive, because the float that is exactly 10 can never be achieved, which may cause a very slight bias in this random function so that it is not purely fair.

However, this is okay because the chance of getting exactly 10 in a random float generation from 1 to 10 is 1 in 100,000,000,000,000,000, so this will be unnoticeable because the interpreter will not be used for something where exact random precision is not absolutely necessary.

### impelementing

A screen shot of a computer program

Description automatically generatedThe new Random class is as follows:

A black background with white text

Description automatically generatedA computer screen shot of a black background

Description automatically generatedThere were originally a few syntax errors, but they have been removed after testing.

After adding the function to the global table in build\_global(), the small test program seemed to work as random, with YES and NO being printed in a very random arrangement. More formal testing to check the expected value will be implemented after typecasting, as currently non-strings cannot be printed.

## type casting

A computer screen with colorful text

Description automatically generatedA screen shot of a computer program

Description automatically generatedpush\_native\_subroutine() has been upgraded to accept an optional tag. By default, no tag will be pushed to the subroutine, but if one is passed it will be. This initially caused issues because I was trying to use ternary operators, but now works.

Because all the type casting code had already been written, this section was very easy to implement just checking for the right number of arguments and then casting to the tag type.

A screenshot of a phone

Description automatically generatedA screen shot of a computer screen

Description automatically generatedA screenshot of a computer code

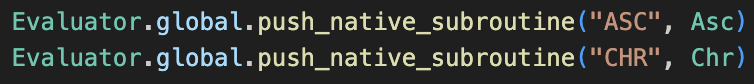
Description automatically generatedThen, build\_global() must be expanded with all the different calls according to the plan: remembering that “float” and “real” both can cast to floats.

As shown, these methods now work, allowing numerical values to be outputted when they are type casted to a string. Also, the string values can be casted to numbers and then is correctly used in a calculation and outputted again.

This is not a thorough test of all the features, and I will formally test them later.

## A screen shot of a computer program Description automatically generatedasc and chr conversions

These methods follow the very similar format to the other methods: checking for the number of arguments, then returning the corresponding value is returned as the correct type.

A computer code on a black background

Description automatically generatedThese also need to be pushed to global, with their capitalised function names.

A black and white background with white text

Description automatically generatedA black background with white text

Description automatically generatedFrom here, the values worked as expected, as shown in the test cases.