TONBRIDGE
SCHOOL

A-Level Computer Science
NEA Report

An Interpreter

Naphat Buristrakul
Candidate number: 5008
Centre number: 61679

Contents

Chapter 1

Analysis

1.1 Background and problem description

Digital creativity is a subject that is taught exclusively to Year 9 students. One of the most important parts of its curriculum is programming as it teaches students logical thinking as well as preparing them for the GCSE in computer science for those who choose to take it.

Currently, the curriculum teaches students Python. Although Python is one of the most popular languages, it has proved problematic for teaching beginners in practice. In particular, the heavy reliance on built-in functions, such as `range()`, can be intimidating for beginners. Moreover, the unorthodox syntax, which utilises colons and whitespace, can make learning other languages difficult.

Mr Chiu, my client, is a digital creativity teacher. Because of the limitations of Python, **he has asked me to develop a programming language that could be used to teach beginners**. He would like the language to be as minimal as possible, i.e., only containing the basic components of a standard programming language. Ideally, this language should: 1) teach core programming concepts to students which can be applied to other languages, and 2) force students to understand programming more fundamentally, as little would be hidden beneath syntactic sugar.

1.2 Client interview

NAPHAT What are the problems with Python for teaching beginner programmers?

MR CHIU I think the main problem with teaching beginners Python is that there is a lot of functionality that is unnecessary for beginners which can be overwhelming for them. Our GCSE exam board even needs to define a subset of the language that is relevant to them! This makes it difficult to teach them general programming concepts. The quirky syntax, like indents and colons, also makes it confusing for them once they try to learn other languages.

NAPHAT So exactly how could a new language improve the experience for beginner programmers?

MR CHIU I would like to keep the intuitive, natural-language-like feel of Python but lose the unnecessary functionality. Essentially, I want a language that is barebones enough to only teach them programming concepts. They can then apply these concepts when they program in other languages. Some C-like syntax would also be useful since it is considered 'standard' syntax. I also want a language that has easy-to-understand error messages – errors in Python are often cryptic to beginners!

NAPHAT You mentioned 'C-like syntax'. What exactly are you referring to?

MR CHIU Oh, stuff like the semicolons in the for loops.

NAPHAT I see. Moving on, what sort of data types should the language support?

MR CHIU Variables should be able to store numbers, strings, and Booleans. The number type does not need to discriminate between floating point numbers and integers to make it simpler. They should also be able to store arrays and a dictionary. A null-like type would also be useful.

NAPHAT And should the user have to type semicolons to end a line in this language?

MR CHIU No need.

NAPHAT Should variables be statically or dynamically typed?

Mr Chiu I think a dynamically-typed language would be more intuitive for beginners.

Naphat What about functions?

Mr Chiu Definitely—they should also support return values.

Naphat Should there be object-oriented elements? Like classes and methods?

Mr Chiu No need for that I think. They don't learn that until much later.

Naphat I'm guessing you would want errors to be as explicit as possible?

Mr Chiu Yes, and it should be as strict as possible. For example, adding a number to a string should terminate the program and at least force the student to cast one type to the other. It would be a good habit for them to get into.

Naphat You mentioned casting. What about other built-in functions?

Mr Chiu There should be as few as possible, and they should do stuff that the students really cannot do themselves. I want the students to type as much out as possible so they really understand what the programs are doing. So like functions that append and remove from arrays and dictionaries.

Naphat What about array sorting? Should that be done for them?

Mr Chiu Actually, yeah. That might be a little complex for them, and they would definitely need it.

Naphat Finally, would you prefer an interpreted or compiled language?

Mr Chiu I think compiling might get confusing for some beginners. An interpreter would be great. Oh, a REPL interface would also be useful, so students can test stuff out.

Naphat That's all I've got. Thank you.

1.3 Existing solutions

Table 1.1 summarises the advantages and disadvantages of common languages used to teach programming to beginners.

Table 1.1: Advantages and disadvantages of common programming languages used to teach beginners

Language	Advantages	Disadvantages
Scratch	• Block coding makes learning about programming concepts more visual and intuitive for beginners. • Minimal setup required as web-based.	• Functions cannot return values. • Does not teach line coding.
Python	• Natural-language-like syntax makes it easier to remember. • Dynamic typing makes variables easier to use. • It is interpreted, so code execution does not require knowledge of compilers.	• Large library of functions and heavy reliance on them can be overwhelming. • Very different from C-like languages, which may make it harder to learn other languages.
Java	• Teaches object-oriented programming immediately, which may arguably be useful. • C-like syntax makes it easier for students to understand and learn other languages.	• At the same time, object-oriented programming can be overwhelming for beginners. • Amount of boilerplate code needed in source code can be confusing. • Requires some knowledge of compilation to execute source code. • Static typing can be annoying for beginners.
JavaScript	• C-like syntax in many places. • Minimal setup required as it can be run on the browser.	• A lot of behaviour can be unexpected, such as operations between numbers and strings. • Hence, errors are often not strict enough and can be misleading.

1.4 Objectives

Based on the client interview and the advantages and disadvantages of existing solutions, the objectives for the programming language are as follows:

1. Users must be able to print variables and literals.
2. Mathematical, logical, and comparative expressions must be evaluated correctly with precedence.
3. Users must be able to store and modify values in variables with the following types:
   (a) Number (includes integers and floats)
   (b) Boolean
   (c) String
   (d) Array
   (e) Dictionary
   (f) Null
4. Users must be able to access and modify:
   (a) values in arrays.
   (b) values of key-value pairs in dictionaries.
   (c) characters in strings.
5. The language must support the following control structures:
   (a) `if`
   (b) `else if`
   (c) `else`
6. The language must support:
   (a) `while` loops.
   (b) `for` loops.
   (c) `break` statements within loops.
7. The language must support:
   (a) functions with parameters and functions without parameters.
   (b) early `return` statements with return values.
8. Users must be able to use built-in functions to:
   (a) prompt for input.
   (b) append a value to an array.
   (c) remove a value from an array or a key-value pair from a dictionary.
   (d) cast a string or Boolean to a number.
   (e) cast a number or Boolean to a string.
   (f) get the size of an array or dictionary, and the length of a string.
   (g) sort an array of numbers or strings.
9. The language must support variable scoping in control structures, loops, and functions. In other words, names (variables and functions) declared within a scope must not be accessible from outside, whereas names declared outside must be accessible and modifiable from inside.
10. With regard to error reporting,
    (a) error messages should be specific and include the type of the error, details regarding the particular situation (i.e., why it was raised), and the location.
    (b) the interpreter should report as many errors as possible in one execution.
11. The interpreter must provide a REPL interface which prompts for and executes source code line-by-line. To do this, the user should run the executable without any arguments (`nea.exe`).
12. The interpreter must also be able to execute source code stored in files. To do this, the user should give the path to the source code file as an argument to the executable (`nea.exe file_path`).

1.5 Language design

Similarly, the client interview and existing solutions have inspired the **design philosophy** of the language, which can be summarised in the following points:

1. The language should have minimal syntactic sugar.
2. The language should have as few built-in functions as possible.
3. Programs written in the language should not require boilerplate code.
4. Syntax should be simple and intuitive, but should also contain some C-like elements.
5. Errors should be strict and not allow any undefined behaviour (e.g., cross-type operations, truthiness, etc.); error reports should be as descriptive as possible.

This design philosophy has inspired the following specification:

1. Printing uses the `print` statement. *(Objective 1)*
   ```
   print "Hello, world!"
   ```
2. Mathematical and logical operations are as follows in order from lowest to highest precedence: *(Objective 2)*

Operation	Keyword/Symbol
Not	`!`
Negation	`-`
Multiplication	`*`
Division	`/`
Mod	`%`
Addition	`+`
Subtraction	`-`
Less than	`<`
Greater than	`>`
Less than or equal to	`<=`
Greater than or equal to	`>=`
Equal to	`==`
Not equal to	`!=`
And	`and`
Or	`or`

3. String concatenation uses the `+` operator.
4. Comments start with `#`.
5. **Variables**
   (a) Variables support the following types: *(Objective 3)*
      i. Number (includes integers and floats)
      ii. Boolean
      The keywords are `true` and `false`.
      iii. String
      iv. Array
      v. Dictionary

vi. Null
The keyword is `null`.

(b) Variables are declared with `var`.
```
var x = 5
```

(c) Arrays are declared with comma-separated values between square brackets. They are zero-indexed and can contain multiple types. Elements are accessed and modified using square brackets containing the index. *(Objective 4a)*
```
var a  = ['a', 'b', 5, [1, 2, 3]]
# a[3][1] == 2
```

(d) Dictionaries are declared with curly brackets. Key-value pairs are separated by commas; keys and values are separated by colons. Values are accessed and modified using square brackets containing the key. If the key does not already exist, an insertion is performed. *(Objective 4b)*
```
var d = {'one': 1, 'two': 2}
# d['one'] == 1
```

(e) Strings are declared with single- or double-quotes. The opening quote must match the closing quote (whether it is single or double). They are treated as arrays of characters; characters can be accessed and modified using the same syntax as arrays. *(Objective 4c)*
```
var s = 'hello'
var t = "world"
# t[0] == 'w'
```

6. **Control structures**

(a) The `if` keyword is used to begin an if statement. The condition is placed within parentheses and the 'then' clause is placed within curly brackets. *(Objective 5a)*

(b) If statements may be optionally followed by an else if statement which is declared using `else if`, followed by the same structure. *(Objective 5b)*

(c) If and else if statements may be optionally followed by an else statement which is declared using `else`, followed by the body placed within curly brackets. *(Objective 5c)*
```
if (x == 5) {
    print "x is 5"
} else if (x == 4) {
    print "x is 4"
} else {
    print "x is something else!"
}
```

7. **Loops**

(a) A while loop begins with a `while` keyword. The condition is placed within parentheses and the loop body is placed within curly brackets. *(Objective 6a)*
```
while (x <= 5) {
    print x
    x = x + 1
}
```

(b) A for loop begins with a `for` keyword. It uses a C-like three-part statement with the following structure: `(initialiser; condition; increment)`. The loop body is placed within curly brackets. *(Objective 6b)*
```
for (var x = 1; x <= 5; x = x + 1) {
    print x
}
```

(c) The keyword for break is `break` and must be used within a loop. *(Objective 6c)*

8. **Functions**

(a) The keyword `func` is used to define a function, followed by the function name. Parameters are listed in parentheses separated by commas. The function body is placed within curly brackets. *(Objective 7a)*
```
func add_them(a, b) {
    print a + b
}
```

(b) Functions can be called with the function name followed by parentheses containing comma-separated arguments.
```
add_them(5, 6)
```

(c) `return` is a statement and *must* be followed by a value to be returned. `null` can be used for empty returns. *(Objective 7b)*
```
func add_them(a, b) {
    print a + b
    return a + b
}
var x = add_them(5, 6)
# x == 11
```

9. **Built-in functions** *(Objective 8)*

(a) `input(prompt)` will print the string `prompt`, prompt the user for input, then return the user's input string.

(b) `append(array, x)` will append `x` to the array `array` in-place, as well as returning the changed array.

(c) `remove(array_or_dict, x)` will remove the element at index `x` from `array_or_dict` in-place if it is an array; otherwise, it will remove the key-value pair with key `x` in-place. It will also return the changed array or dictionary.

(d) `to_number(x)` will return the numerical representation of `x` (`1` and `0` for true and false Boolean values respectively, or the number represented in the given string).

(e) `to_string(x)` will return the string representation of `x` (`"true"` and `"false"` for true and false Boolean values respectively, or the given number as a string).

(f) `size(array_or_dict)` will return the number of elements in an array or the number of key-value pairs in a dictionary.

(g) `sort(array)` will return the sorted array in ascending order. It will only operate on arrays consisting entirely of numbers or strings.

Mr Chiu, my client, has verbally agreed to this proposed specification.

1.6 Research

1.6.1 Overview

The following materials have formed the basis of my research on interpreter implementation:

- *Crafting Interpreters* by Robert Nystrom
- *Compilers: Principles, Techniques, and Tools* by Aho, Lam, Sethi, and Ullman
- *Programming Language Pragmatics* by Michael L. Scott

Although the last two are compiler textbooks, many of the techniques up to a certain stage can be used in an interpreter.

Generally, the execution of source code by an interpreter can be **decomposed** into three stages: 1) **lexical analysis**, 2) **syntax analysis**, and 3) **execution**. Each stage will be explained in detail in the Design section.

Essentially, lexical analysis (performed by the **tokenizer**[1]) is concerned with converting the raw source code into a sequence of abstract **tokens**, which could be keywords (e.g., an `if` statement) or special characters (e.g., a curly bracket). From this sequence of tokens, syntax analysis (performed by the **parser**) will **recursively** build an **abstract syntax tree**, which represents the structure of the source code. Finally, this tree will be **recursively traversed** in the execution stage and the source code will be 'run'. How the source code is 'executed' differs from implementation to implementation—most modern interpreters will opt to first compile the abstract syntax tree into a lower-level intermediate representation, known as bytecode. This allows for certain optimisations.

[1]The American 'z' is used here as this is the spelling used in textbooks.

1.6.2 Examples

Python

Although Python is commonly referred to as an interpreted language, its standard CPython implementation is much more nuanced. Lexical analysis and syntax analysis are done as normal, converting the source code into a sequence of tokens and then into an abstract syntax tree. However, the nodes in the abstract syntax tree are *not* evaluated or executed directly by the interpreter. Instead, the tree is first recursively compiled into **bytecode**, an **intermediate representation**. During this stage, certain optimisations can be performed. Finally, each bytecode instruction is executed successively by the Python **virtual machine**.[2]

Ruby

In modern versions of Ruby, a similar process to Python is performed—the abstract syntax tree is first compiled to bytecode. Then, the bytecode is executed by YARV (Yet Another Ruby VM), Ruby's virtual machine. However, in older versions, Ruby was interpreted by MRI (Matz's Ruby Interpreter), which was written in C. Here, the abstract syntax tree is **recursively traversed**, and nodes are evaluated and executed **directly** without translation to bytecode or machine code. This approach is known as a **tree-walk interpreter**. Although less optimisable, they are easier to implement.[3]

JavaScript

Implementations of JavaScript greatly vary. Although they were initially **tree-walk interpreters**, modern implementations employ a sophisticated technique known as just-in-time compilation. This technique uses a *monitor* which continuously profiles the running code. Repeatedly-run lines of code will then be (re)compiled if it is determined that the improvement in runtime performance from the added optimisations is worth the time taken to (re)compile.[4]

1.6.3 Evaluation

Modern implementations of interpreters, such as the above, often use advanced techniques such as bytecode optimisations and just-in-time compilation. However, these approaches are beyond the scope of the A-Level; moreover, the language I intend to design will only be used to teach programming concepts for beginners. Hence, performance optimisations using these techniques are not necessary. Therefore, I intend to develop a **tree-walk interpreter** which **recursively walks** the abstract syntax tree immediately, evaluating and executing nodes, without doing further translations.

1.7 Proposed solution

1.7.1 Programming language

I intend to implement my interpreter in **Rust**. Table 1.2 summarises my decision-making process.

Essentially, Python's dynamic typing immediately removed itself from further consideration—in a large project like this, where I could be dealing with many types, it is important that I can keep track of which variables have which types, which types functions return, etc., and catch type-related bugs quickly.

I also considered C++. Although I liked the static typing, the memory safety issues that I have come across in my experience using C++ were too problematic—for example, out-of-bounds indexes when accessing arrays can be hard to spot as no errors are reported and result in undefined behaviour. Memory-related errors like these require the use of *sanitizers* to debug (these, in turn, require a Linux system). Hence, I decided against using C++.

Therefore, Rust seemed to have the best of both worlds—it is statically typed and is notorious for its memory safety features (most notably, the *borrow checker*). I have also read that runtime errors are rare as most bugs are flagged by the compiler before the source code is even allowed to run. These features make it a popular language for systems programming. So, although I was not too familiar with it, I decided to use Rust to implement the interpreter.

[2] https://alumni.media.mit.edu/~tpminka/patterns/python/
[3] https://craftinginterpreters.com/a-map-of-the-territory.html
[4] https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

Table 1.2: Advantages and disadvantages of each programming language for the implementation of the interpreter

Language	Advantages	Disadvantages
Python	• Portable. • Easy setup and execution. • I am familiar with Python.	• Dynamically typed; type safety is very important in large projects. • Slow.
C++	• Very fast. • Statically typed.	• Has memory safety issues, which can lead to memory leaks and undefined behaviour.
Rust	• Popular for systems programming. • Statically typed, and supports various types such as `enum` and `struct`, which may be useful. • Fast. • Has memory safety features (e.g., borrow checker), which are important in large projects. • Strict compiler reduces chance of runtime error, i.e., very **defensive**.	• As Rust is relatively new and changing quickly, there are fewer resources available online. • I am not very familiar with Rust.

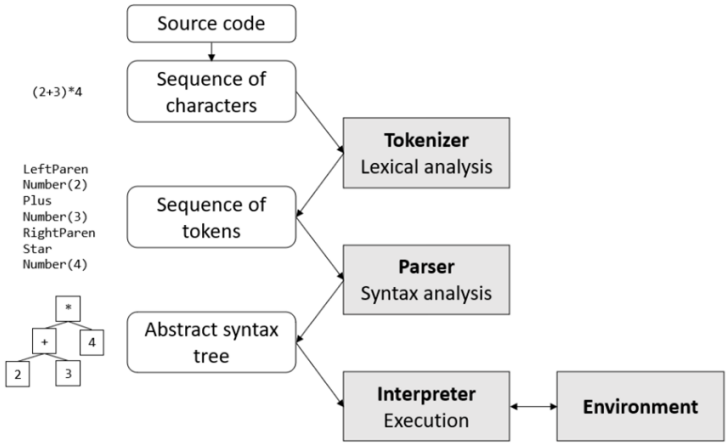

Figure 1.1: Overview of the interpreter

1.7.2 Technical overview

Figure 1.1 provides an overview of the different components of the interpreter, as briefly outlined in Section 1.6.1.

1. **Lexical analysis**
   The **tokenizer** (also known as the lexer or scanner) builds a **sequence of tokens** from the source code. The challenge here could potentially be eliminating one-off errors when scanning the source code string for potential tokens.

2. **Syntax analysis**
   The **parser** constructs an **abstract syntax tree (AST)** from the sequence of tokens. This will possibly present the main algorithmic challenge of the project. The algorithm will have to make use of **grammar rules** to construct a correct tree which takes precedence into account. A potential algorithm that could be used here is **recursive descent parsing**, which makes use of separate **functions** for each grammar rule. Parts of a **production** can be **recursively** constructed by calling the functions responsible for those rules. For example, the rule `<add> ::= <expression> + <expression>` will call the function for the `<expression>` grammar rule, which will return a **subtree** of the AST. Then, both subtrees can be combined into the AST.

3. **Execution**
   The component which performs this is known, confusingly, as the **interpreter**. It will **recursively**

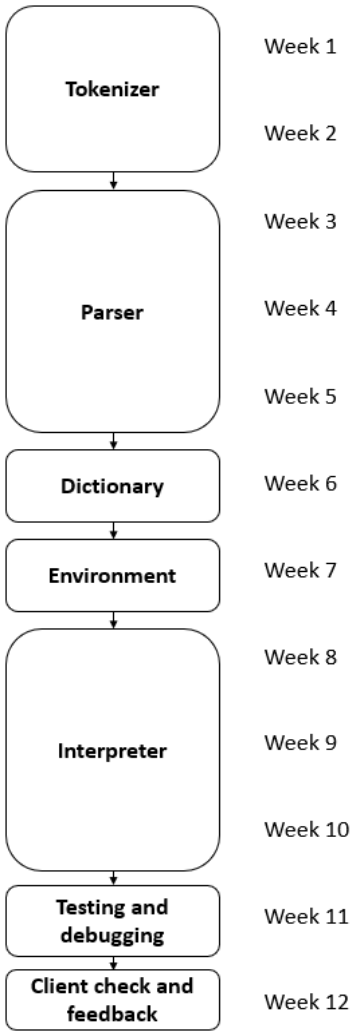

Figure 1.2: Critical path diagram

evaluate or execute nodes in the AST. During execution, names (for variables and functions) will be used to read and write **values** in the **environment**. These names will be stored alongside their associated value in a **hash table**. The implementation of dictionaries in the language will also be challenging, especially in regard to designing a **constant time** hashing algorithm.

This overview makes use of **functional abstraction** to illustrate the process of executing source code, i.e., the tokenizer, parser, and interpreter are *assumed* to work as expected in the diagram. To organise the project, each component can be abstracted into separate **classes** using **object-oriented programming**. For example, a sequence of tokens can be retrieved using a single **public interface method** of the tokenizer's class.

Object-oriented programming enforces **information hiding** and **encapsulation**. This decreases the number of variables in scope, which could make creating a bug less likely.

As part of the objectives, the language will also have to support a dictionary. I plan to implement this myself and use techniques such as **recursive hashing** and two-dimensional arrays, which will store buckets of values.

1.8 Critical path

Time management is essential in large projects such as this. Luckily, the components of the interpreter can be developed separately and successively thanks to functional abstraction. Figure 1.2 illustrates my plan for the project.

Chapter 2

Design

2.1 Error handling

2.1.1 Error types

There are many errors that can occur in each stage of source code execution. In order to achieve Objective 10a, errors have to be detailed and include specific information. To accommodate this, I will use an **enum** `ErrorType`, whose variants are the different types of errors that may occur. Variants in Rust can also store named fields, and I will use these to store specific information regarding the error. Tables 2.1, 2.2, 2.3, and 2.4 lists all variants of `ErrorType`, i.e., all the errors that may happen in lexical analysis, syntax analysis, the environment, and execution respectively.

Table 2.1: Errors that may happen in lexical analysis

Name	Description	Information	Message
Unexpected character	The character in the source code was not a valid character for the tokenizer, e.g., `$`.	• The invalid character • Line number	Line {line number}: unexpected character `{invalid character}`.
Unterminated string	A string was opened but never closed by the end of the program, e.g., `"Hello, world!`.	N/A	A string was never closed by the end of the program.

Table 2.2: Errors that may happen in syntax analysis, i.e., syntax errors

Name	Description	Information	Message
Expected character	Based on the grammar rules, the parser expected a certain character, e.g., a `}` to close a for loop.	• Expected character • Line number	Line {line number}: expected character `{expected character}`.
Expected expression	Based on the grammar rules, the parser expected an expression, e.g., no expression after a `+`.	• Line number	Line {line number}: expected expression.
Expected function name	No (valid) function name was found after a `func` keyword.	• Line number	Line {line number}: expected function name. Make sure it is not a keyword.

Expected parameter name	No parameter was found after a comma in a function declaration, e.g., `func f(a,) {}`	• Line number	Line {line number}: expected parameter name in function declaration.
Expected variable name	No (valid) variable name was found after a `var` keyword.	• Line number	Line {line number}: expected variable name. Make sure it is not a keyword.
Expected semicolon after initialiser	No semicolon was found after the initialising statement in a for loop, e.g., `for (var i = 0 i < 5; i = i + 1) {}`	• Line number	Line {line number}: expected `;` after initialising statement in `for` loop.
Expected semicolon after condition	No semicolon was found after the condition in a for loop, e.g., `for (var i = 0; i < 5 i = i + 1) {}`	• Line number	Line {line number}: expected `;` after condition in `for` loop.
Expected closing parenthesis after increment	There was no `)` after the increment statement in a for loop, e.g., `for (var i = 0; i < 5; i = i + 1 {}`	• Line number	Line {line number}: expected `)` after increment statement in `for` loop.
Expected colon after key	There was no `:` after a key in a dictionary definition e.g., `var d = {'one' 1}`	• Line number	Line {line number}: expected colon after dictionary key.

Table 2.3: Errors that may happen in the environment

Name	Description	Information	Message
Name error	No values (variables or functions) associated with the given name could be found within the current scope and the parent scopes.	• Given name • Line number	Line {line number}: `{name}` is not defined.

Not indexable	Attempted to index a non-indexable value, e.g.,
	`var a = 5`
	`print a[0]`
		• Line number	Line {line number}: the value is not indexable.

Out of bounds index	Attempted to access a element with an index that was out of bounds, e.g.,
	`var a = [1, 2]`
	`print a[47]`
		• Given index
		• Line number
			Line {line number}: index `{index}` is out of bounds.

Insert non-string into string	Attempted to insert a non-string into a string., e.g.,
	`var s = "abc"`
	`s[1] = 5`
		• Line number	Line {line number}: attempted to insert a non-string into a string.

Table 2.4: Errors that may happen during execution, i.e., runtime errors

Name	Description	Information	Message
Invalid assignment target	Attempted to assign a value to a non-variable such as an expression, e.g., `5 = 6`.	• Line number	Line {line number}: invalid assignment target. Make sure you are not assigning to a literal.

Expected type	Certain operations expect a certain type of argument. This error is raised when the given argument does not match the expected type, e.g.,
	`!"abc"  # `!` expects a Boolean value to follow.`
	`size(5)  # This built-in function expects the argument to be either an array or dictionary.`
		• Expected type
		• Given type
		• Line number
			Line {line number}: expected type `{expected type}`; instead got `{given type}`.

Non-natural index	The given index for an array is a number, but it is not a positive integer, e.g., `a[-3]` or `a[3.14]`.	• The given index • Line number	Line {line number}: index evaluated to `{index}`, which is not a positive integer.
Non-number index	The given index for an array is not a number, e.g., `a["abc"]`.	• Type of given index • Line number	Line {line number}: index evaluated to a `{given type}`, which is not a positive integer.
Binary type error	Most binary expressions require both sides' types to match. This error is raised when this is not the case, e.g., `"hello" + 5`.	• Expected type(s) • Type of left-hand side value • Type of right-hand side value • Line number	Line {line number}: this operation expects both sides' types to be `{expected type(s)}`. Instead, got `{LHS type}` and `{RHS type}` respectively.

Divide by zero	The divisor of a division expression is 0.	• Line number	Line {line number}: divisor is 0.
If condition not Boolean	The condition of an if condition did not evaluate to a Boolean value, e.g., `if ("hello") {}`.	• Line number	Line {line number}: the `if` condition did not evaluate to a Boolean value.
Loop condition not Boolean	The condition of a loop did not evaluate to a Boolean value, e.g., `while ("hello") {}`.	• Line number	Line {line number}: the condition of the loop did not evaluate to a Boolean value.

Cannot call name	Attempted to call a non-function name, e.g.,
	`var s = "hello"`
	`s(5)`
		• Line number	Line {line number}: cannot call name as a function.

Argument-parameter number mismatch	The number of arguments given to a function call did not match the number of parameters the function accepts.	• Number of function parameters • Number of arguments given • Line number	Line {line number}: attempted to call function with {# arguments} argument(s), but function accepts {# parameters}.
Cannot convert to number	Attempted to convert something to a number that cannot be converted to a number, e.g., `to_number("hello")`.	• Line number	Line {line number}: could not convert to a number.
Cannot hash function	Attempted to use a function as the key of a dictionary entry.	• Line number	Line {line number}: cannot hash function (functions cannot be used as keys in dictionary entries).
Cannot hash dictionary	Attempted to use a dictionary as the key of a dictionary entry.	• Line number	Line {line number}: cannot hash dictionary (dictionaries cannot be used as keys in dictionary entries).
Key error	Attempted to access a dictionary with a key that does not exist in the dictionary.	• Given key • Line number	Line {line number}: key `{key}` does not exist in the dictionary.
Return outside function	The `return` statement was used outside a function.	• Line number	Line {line number}: `{return}` has to be used within a function.
Break outside loop	The `break` statement was used outside a loop.	• Line number	Line {line number}: `{break}` has to be used within a loop.

Note that the errors **specific**, and provide the location of the error as per Objective 10a. All the errors include the line number, with the exception of 'Unterminated String', which explicitly includes the location ('end of the program') in the error message.

2.1.2 Error reporting

It is good practice to **centralise** the error reporting function, so a single function will be created which will take an **array of errors** as the argument and print the message of each one. This function will be called `report_errors(errors: [ErrorType])`. It will make use of a **helper function** which prints the messages of individual errors, `print_error(error: ErrorType)`. This function will use a `switch` statement to print the

correct message for each error. The pseudocode in Listing 1 illustrates these two functions.

Listing 1 Error reporting procedures

```
function REPORT_ERRORS(errors)
    Print "An error has occurred."
    for error in errors do
        PRINT_REPORT(error)
    end for
end function

function PRINT_REPORT(error)
    switch error do
        case UnexpectedCharacter { character, line }
            Print "Line " + line + ": unexpected character " + character + "."
        etc.
    end switch
end function
```

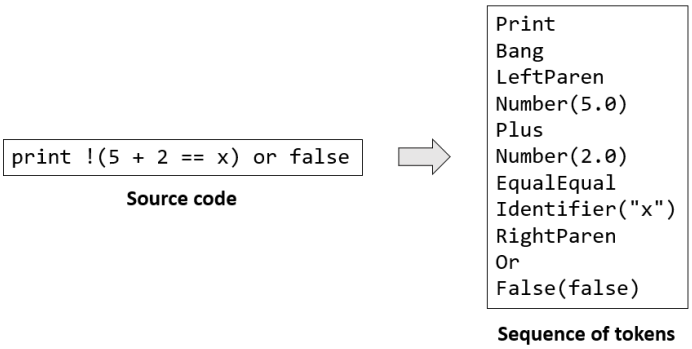

Figure 2.1: Example of lexical analysis

2.2 Lexical analysis

2.2.1 Overview

The tokenizer, which performs lexical analysis, is responsible for scanning the source code for **tokens** and building a sequence of tokens. A token is an abstract representation of a 'meaningful substring' in the source code string (formally known as a *lexeme*), which could be a special character (`{`, `+`, etc.), a literal (numbers, strings, and Boolean values), or a keyword (`and`, `while`, etc.). Figure 2.1 shows an example of this process.

2.2.2 The `Token` class

Enum variants will be used to represent the different *types* of tokens; the enum will be called `TokenType`. The types can be broadly categorised into four categories: single-character tokens, one- or two-character tokens, literals, and keywords. The distinction between the first two categories will be explained later. The different types of tokens and their associated lexemes are listed in Table 2.5. The objective(s) that a token is designed to help complete is also listed in the table.

Table 2.5: The types of tokens

Category	Name	Lexeme	Objective(s)
Single-character tokens	`LeftParen`	`(`	2, 5, 6a, 6b, 7a, 8
	`RightParen`	`)`	2, 5, 6a, 6b, 7a, 8
	`LeftCurly`	`{`	3e, 5, 6a, 6b, 7a, 9
	`RightCurly`	`}`	3e, 5, 6a, 6b, 7a, 9
	`LeftSquare`	`[`	3d, 4
	`RightSquare`	`]`	3d, 4
	`Colon`	`:`	3e, 4b
	`Comma`	`,`	3d, 3e, 7a
	`Minus`	`-`	2
	`Percent`	`%`	2
	`Plus`	`+`	2
	`Semicolon`	`;`	5a, 5b, 6b
	`Slash`	`/`	2
	`Star`	`*`	2
One- or two-character tokens	`Bang`	`!`	2
	`BangEqual`	`!=`	2

	`Equal`	`=`	2
	`EqualEqual`	`==`	2
	`Greater`	`>`	2
	`GreaterEqual`	`>=`	2
	`Less`	`<`	2
	`LessEqual`	`<=`	2
Literals	`True`	`true`	2, 3b
	`False`	`false`	2, 3b

	`Number`	`5`
		`-5`
		`3.14`
		etc.
			2, 3a

	`String_`	`"Hello, world!"`
		`'Hello, world!'`
		etc.
			3c

Keywords	`And`	`and`	2
	`Break`	`break`	6c
	`Else`	`else`	5b, 5c
	`Func`	`func`	7a
	`For`	`for`	6b
	`If`	`if`	5a, 5b
	`Null`	`null`	3f
	`Or`	`or`	2
	`Print`	`print`	1
	`Return`	`return`	7b
	`Var`	`var`	3
	`While`	`while`	6a

Misc.	`Identifier`	Any 'word' that is not a keyword, e.g.,
		`name`
		`x`
		etc.
			3, 7a

	`Eof`	Special token that concludes the sequence of tokens	

In the case of literal tokens (numbers, strings, and Boolean values), another enum, `Literal`, will be used to store the *value* of the literal. Its possible variants are the following:

- `Number`
  Internally, a float will be stored with this variant.
- `String_`
  The string will be stored.
- `Bool`
  The Boolean value will be stored.
- `Null`
  This represents the `Null` type. This variant will also be used when the token does not have an associated literal value.

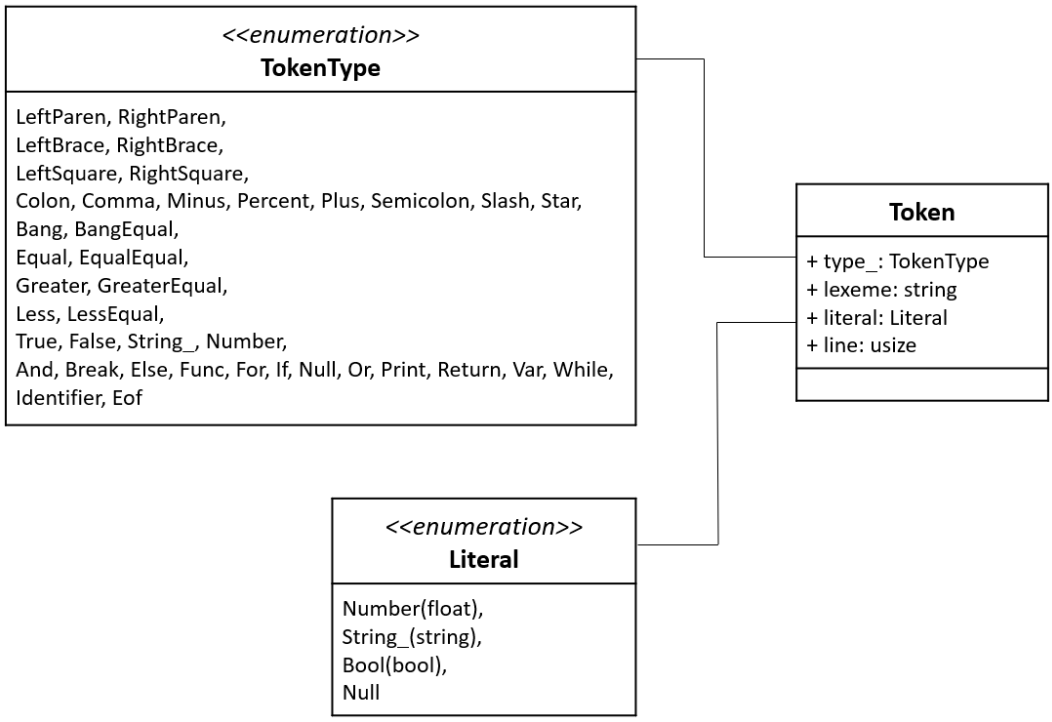

Figure 2.2: Class diagram of the `Token` class and related types

A `Token` class will represent an individual token. It will have the following **public attributes**:

- `type_` (a variant of the `TokenType` enum)
  The type of the token as listed in Table 2.5.
- `lexeme` (string)
  The substring in the source code from which the token was derived. This will be necessary later in the case of the identifier token.
- `literal` (a variant of the `Literal` enum)
  The literal value of the token in the case of `True`, `False`, `Number`, and `String_` token types; otherwise, it will be set to the `Null` variant.
- `line` (usize)
  The line in the source code from which the token was derived. This will be used in error messages.

These relationships are summarised in Figure 2.2.

2.2.3 Algorithm

Overview

One of the key algorithmic challenges of lexical analysis is the characterisation of multi-character tokens. If all tokens were one character long, scanning the source code to build the sequence of tokens would be trivial. However, particularly in the case of one- or two-character tokens, it is impossible to distinguish, say, a `Bang` from a `BangEqual`, purely from the first character. The challenge here is to correctly characterise these tokens while eliminating one-off errors and maintaining memory safety.

First, I will present a naïve approach to the algorithm and explain its disadvantages. I will then explain a preferred approach using a **deterministic finite automaton (DFA)**.

Naïve approach

This approach constructs the sequence of tokens by **iterating** through the source code and **greedily** adding the longest possible tokens[1] by using nested loops. The pseudocode in Listing 2 sketches out the algorithm.

The naïvety in this approach is in the **hardcoding of nested loops**—if a `"` were scanned, we would need a loop to keep scanning until another `"` is scanned to create a `String_` token, and similarly for numbers

[1] Longest possible token means that `314` is recognised as 314, not 3 or 31.

Listing 2 Naïve approach to lexical analysis

```
procedure TOKENIZE(source)
    start ← 0                                          ▷ Index of the first character of current token
    current ← 0                                        ▷ Index of next character to be scanned
    tokens ← [ ]                                       ▷ Result sequence of tokens
    while current not at end do
        start ← current
        switch source[current] do
            case '('                                   ▷ Single-character tokens
                Append LeftParen token to tokens
            etc. (other single-character tokens)
            case '!'                                   ▷ One- or two-character tokens
                if PEEK('=') then
                    Append BangEqual token to tokens
                else
                    Append Bang token to tokens
                end if
            etc. (other one- or two-character tokens)
            case '"'                                   ▷ Strings
                while source[current] ≠ '"' do
                    current ← current + 1
                end while
                s ← source[start..(current + 1)]
                Append String_ token with Literal::String_(s) to tokens
            case digit                                 ▷ Numbers
                while source[current] is a digit do
                    current ← current + 1
                end while
                if source[current] = '.' then          ▷ Capture decimal part
                    while source[current] is a digit do
                        current ← current + 1
                    end while
                end if
                x ← source[start..(current + 1)]
                Append Number token with Literal::Number(x) to tokens
            case alphabetic character or '_'           ▷ Keywords and identifiers
                while source[current] is alphanumeric or '_' do
                    current ← current + 1
                end while
                word ← source[start..(current + 1)]
                if word is a keyword then
                    Append respective keyword to tokens
                else
                    Append Identifier token to tokens  ▷ word will be stored in token.lexeme
                end if
            case '#'                                   ▷ Comments
                while source[current] ≠ newline do
                    current ← current + 1
                end while
            otherwise
                raise UnexpectedCharacter error
        end switch
        current ← current + 1
    end while
    Append Eof token to tokens
end procedure
```

and keywords. Moreover, we need to *peek* at future characters to correctly characterise a token, such as to differentiate a `Bang` from a `BangEqual`. This makes this approach prone to one-off errors and hence memory leaks, which could cause the program to crash.

DFA approach

Alternatively, we can visualise the tokenizer as a **deterministic finite automaton (DFA)**. This will have the benefit of making sure that *only* the **current character** and the **current state** of the DFA are ever checked in an iterative step—this decreases the chance of one-off errors and similar bugs. Figure 2.3 shows the state transition diagram of the DFA. Each accepting state, with the exception of `NoOp`, represents a completed token.

Although it is necessary to hardcode all possible DFA states and transitions, I believe this approach is better than the naïve approach as it eliminates *peeking* and *nested loops*—in each iteration, we are simply making the decision of which state to go to next, based on the current character and the current state.

Now, we are essentially concerned with implementing a simulator for this DFA. All the possible states will be represented by variants of a `State` enum. A function, `scan_token(index)`, will simulate the DFA, starting the scan at the given index and returning a token. This function will, again, make use of `switch` statements to make the decision of which state to go to next, based on the current state and current character. The sequence of tokens can then be called by repeatedly calling this function. Listing 3 sketches out an implementation of this algorithm.

Because this algorithm visits every character at most twice (e.g., if `GotBang` is the current state and the next character is not `'='`, the algorithm must visit the `'='` again in the next call of `scan_token()`), its time complexity is $O(n)$, where n is the length of the source code.

Moreover, the explicit nature of the transitions makes it easy to spot where errors might occur, if there is an invalid combination of current state and character. From the state transition diagram (Figure 2.3), it is clear that there are only two errors that can occur: `UnterminatedString`, when a string is unclosed by the end of the file; and `UnexpectedCharacter`, when there is there is no transition for the current character from the current state.

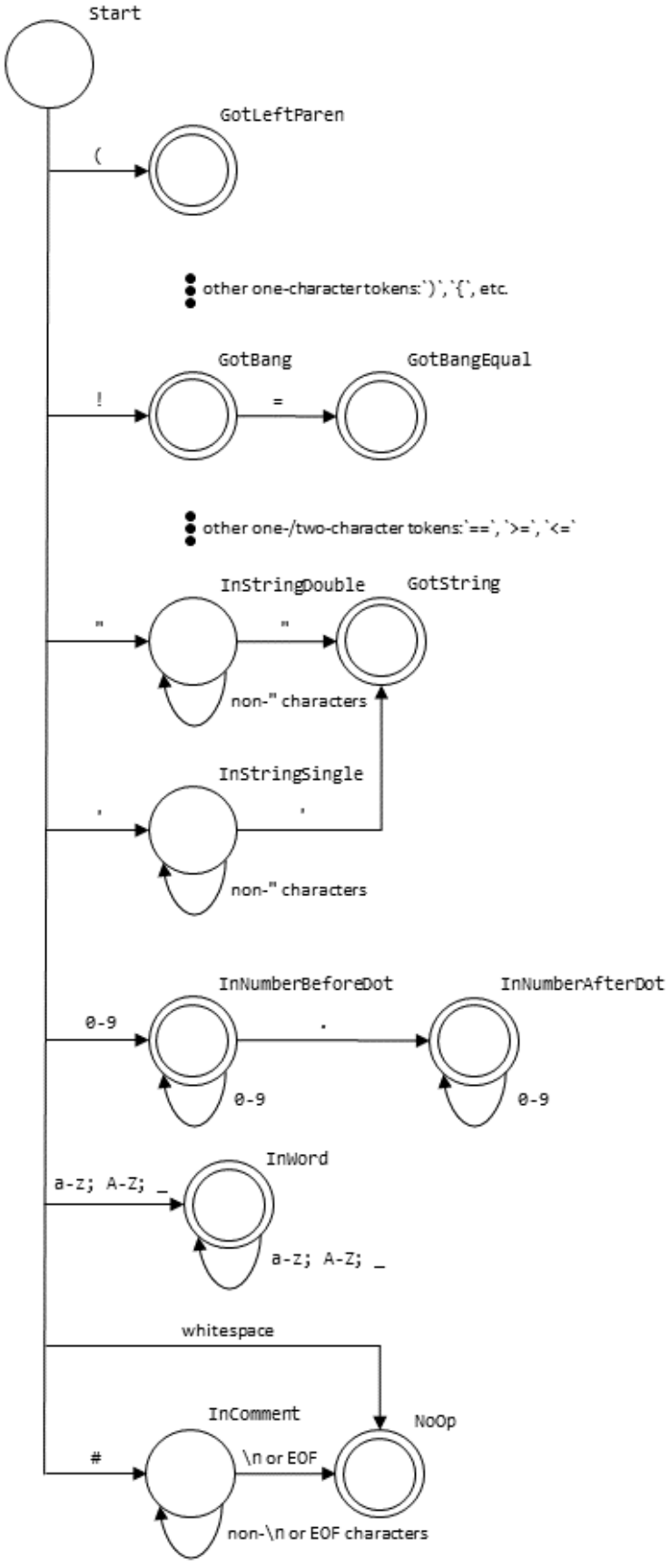

Figure 2.3: State transition diagram of the tokenization DFA

Listing 3 DFA approach to lexical analysis

```
function SCAN_TOKEN(index, source)
    start_index ← index
    current_index ← index
    current_state ← State::Start
    loop                                  ▷ Loop until an early return (i.e., an accepting state) is reached
        current_character ← source[current_index]
        switch current_state do
            case State::Start
                switch current_character do
                    case '('
                        current_state ← State::GotLeftParen
                    etc. (other single-character tokens)
                    case '!'
                        current_state ← State::GotBang
                    etc. (other one- or two-character tokens)
                    case '"'
                        current_state ← State::InStringDouble
                    case '''
                        current_state ← State::InStringSingle
                    case digit
                        current_state ← State::InNumberBeforeDot
                    case alphabetic character or '_'          ▷ Keywords and identifiers
                        current_state ← State::InWord
                    case '#'
                        current_state ← State::InComment
                    case whitespace
                        current_state ← State::NoOp
                    otherwise
                        raise UnexpectedCharacter error
                end switch
            case State::GotLeftParen
                return LeftParen token
            etc. (other single-character tokens)
            case State::GotBang
                if current_character = '=' then
                    current_state ← State::GotBangEqual
                else
                    return Bang token
                end if
            case State::GotBangEqual
                return BangEqual token
            etc. (other one- or two-character tokens)
```

Listing 3 DFA approach to lexical analysis (continued)

```
            case State::InStringDouble
                if current_character = '"' then
                    current_state ← State::GotString
                else if at end then
                    raise UnterminatedString error
                end if
            case State::InStringSingle
                if current_character = ''' then
                    current_state ← State::GotString
                else if at end then
                    raise UnterminatedString error
                end if
            case State::GotString
                s ← source[start_index..current_index]
                return String_ token with Literal::String_(s)
            case State::InNumberBeforeDot
                if current_character = '.' then
                    current_state ← State::InNumberAfterDot
                else if not a digit or at end then
                    x ← source[start_index..current_index]
                    return Number token with Literal::Number(x)
                end if
            case State::InNumberAfterDot
                if current_character not a digit or at end then
                    x ← source[start_index..current_index]
                    return Number token with Literal::Number(x)
                end if
            case State::InWord
                if current_character not (alphabetic character or '_') then
                    if source[start_index..current_index] is a keyword then
                        return respective token
                    else
                        return Identifier token                ▷ Name will be stored in token.lexeme
                    end if
                end if
            case State::InComment
                if current_character = newlineoratend then
                    current_state ← State::NoOp
                end if
            case State::NoOp
                return
        end switch
        current_index ← current_index + 1
    end loop
end function
```

2.2.4 The `Tokenizer` class

To make the tokenizer component independent from the other components, **object-oriented programming** will be used. This component will be **encapsulated** within a `Tokenizer` class, which will have the following **private attributes**:

- `source` (string)
  The source code string.
- `tokens` (an array of `Token` objects)
  The result sequence of tokens.
- `start` (usize)
  An index pointing to the start of the current token. This will be used to set the value of `Token.lexeme` and literals.
- `current_index` (usize)
  An index pointing to the next character to be scanned.
- `current_line` (usize)
  The current line number.

The class will also have the following **methods**:

- `new(source: string)` (public)
  Constructs a `Tokenizer` instance with the given source code string.
- `tokenize() : [Token]` (public)
  The interface method which creates and returns an array of tokens by repeatedly calling `scan_token()`. Handles errors by calling `report_errors` on raised errors (see Subsection 2.1.2).
- `scan_token() : Token` (private)
  As described in Listing 3, it returns the token starting from the current index. However, it will not take any arguments; both the current index and the source code, which were given as arguments in the pseudocode, will be accessed from the object attributes instead (i.e., `self.current_index` and `self.source`).
- `construct_token_with_literal(token_type: TokenType, literal: Literal) : Token` (private)
  A helper function which returns a fully formed token object. It will use the `start` and `current_index` attributes to create the `lexeme` of the token, and the `current_line` attribute as the `line` of the token.
- `construct_token(token_type: TokenType) : Token` (private)
  A helper function which returns a token object without a literal value. This calls `construct_token_with_literal` with the `literal` argument as `Literal::Null`.

The class diagram in Figure 2.4 summarises all the classes and types introduced in this section.

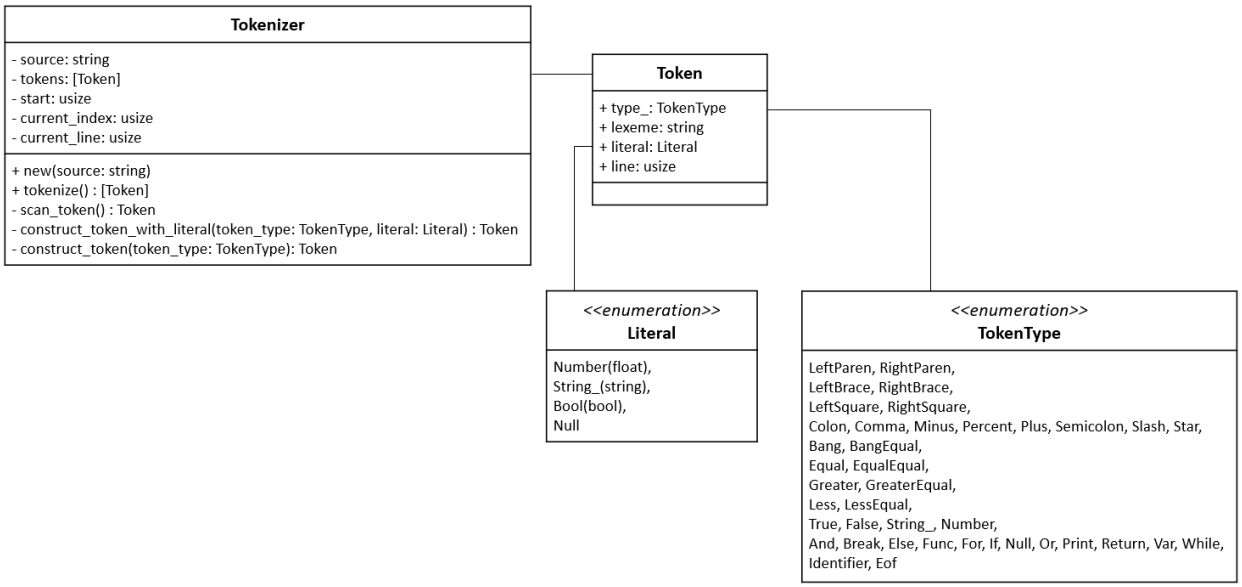

Figure 2.4: Class diagram of all lexical analysis-related classes and types

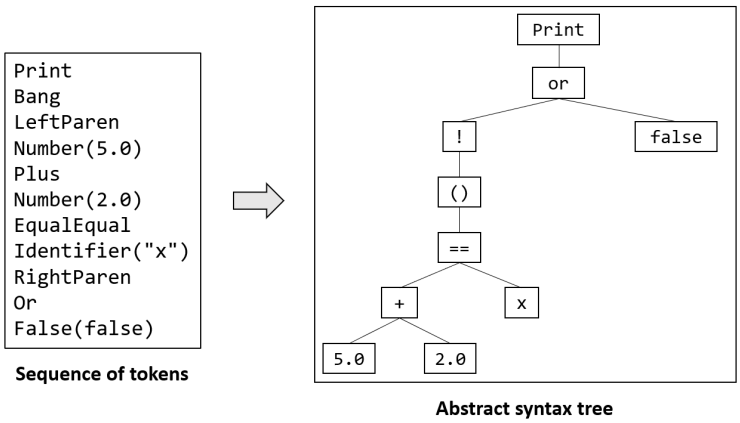

Figure 2.5: Example of syntax analysis

2.3 Syntax analysis

2.3.1 Overview

Syntax analysis is performed by the **parser**. It is concerned with building an **abstract syntax tree (AST)** from the sequence of tokens constructed by the tokenizer. Using the same running example, Figure 2.5 illustrates this process.

There are two types of syntactic 'structures' in this language:

1. **Expressions**
   These will be **evaluated** to **values** and are stored as instances of the `Expr` class.

2. **Statements**
   These will be **executed** to carry out actions and are stored as instances of the `Stmt` class.

2.3.2 Expressions

The `Expr` class

Different expressions have different structures. For example, a binary expression will have a left-hand side expression, an operator token, and a right-hand side expression, whereas an expression for a function call will have the expression for the function and a list of expressions for the arguments.

Variants of an enum, `ExprType`, will be used to represent and store *structural* information about the expressions. As expressions are often recursive, these structures will also be recursive. This recursion is what allows a **tree** to be constructed. Rust's ability to store named fields within enum variants will be used. A special helper class, `KeyValue<T>`, will be used to store key-value entries in dictionaries, and will become more useful later in the dictionary implementation (see Subsection 2.5.2). It stores two public attributes:

- `key` (type `T`)
- `value` (type `T`)

The variants of `ExprType` are listed in Table 2.6 with their names, intended uses, stored information, and objectives they were designed to help complete.

An `Expr` class will essentially wrap the `ExprType` enum. However, it will also store the line number of the expression to use in error messages. Figure 2.6 summarises the `Expr` class in a class diagram.

Table 2.6: Variants of `ExprType`

Name	Use	Information stored	Objective(s)
`Array`	Arrays	• `elements` (an array of `Expr` objects): the expressions for the array elements	3d
`Assignment`	Variable assignments (not variable declarations, which are statements)	• `target` (an `Expr` object): the expression to which the value needs to be assigned (in this case, `a[2]`) • `value` (an `Expr` object): the expression whose evaluated value needs to be assigned to the target (in this case, `5+2`)	3, 4
`Binary`	Binary operations (mathematical, logical, and comparative)	• `left` (an `Expr` object): the LHS expression • `operator` (a `Token` object): the operator (`Or`, `Plus`, etc.) • `right` (an `Expr` object): the RHS expression	2
`Call`	Function calls, e.g., `f(123)(5, "a")`[2]	• `callee` (an `Expr` object): the expression whose evaluated value (which should be a function) is to be called, in this case, `f(123)` • `arguments` (an array of `Expr` objects): the expressions for the arguments to be passed to the function call, in this case, `[5, "a"]`	7a, 8
`Dictionary`	Dictionaries	• `elements` (an array of `KeyValue<Expr>`s): the expressions for the key-value entries	3e
`Element`	Array/dictionary elements and characters of strings, e.g., `a[5][2+3]`[3]	• `array` (an `Expr` object): the expression whose evaluated value is to be accessed using the index (note this can be an array or a dictionary), in this case, `a[5]` • `index` (an `Expr` object): the expression for the index to be used to access the array or dictionary, in this case, `2+3`	4
`Grouping`	Bracketed expressions, e.g., `(1+2)`	• `expression` (an `Expr` object): the expression enclosed within the brackets	2
`Literal`	Literal values (strings, numbers, Boolean values, and null)	• `value` (a `Literal` variant): the literal value	1, 2
`Unary`	Unary operations; in this language, the only unary operators are `!` (logical `NOT`) and `-` (negation)	• `operator` (a `Token` object): the operator (`Bang` or `Minus`) • `right` (an `Expr` object): the RHS expression	2
`Variable`	Name that could refer to a variable or a function in the environment	• `name` (a string): the name of the identifier (this is stored in as lexeme of the token)	1, 2, 3, 4, 7a, 8

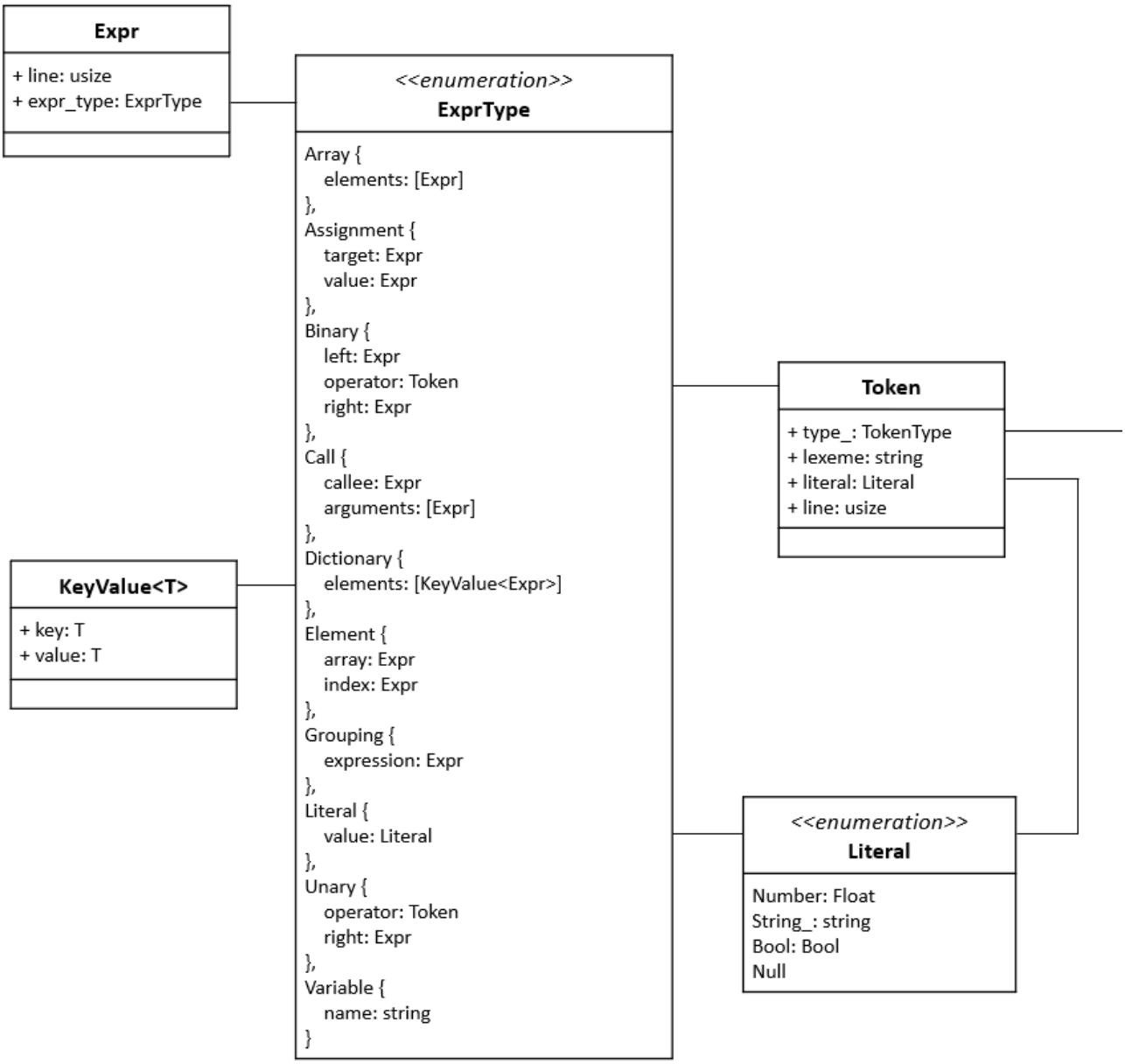

Figure 2.6: Class diagram of the `Expr` class and related types

```
<expression> ::= <assignment>
<assignment> ::= <or> (Equal <assignment>)?    [Assignment]
<or> ::= <and> (Or <and>)*    [Binary]
<and> ::= <equality> (And <equality>)*    [Binary]
<equality> ::= <comparison> ((EqualEqual | BangEqual) <comparison>)*    [Binary]
<comparison> ::= <plus_minus> ((Greater | Less | GreaterEqual | LessEqual) <plus_minus>)*
        [Binary]
<plus_minus> ::= <star_slash_percent> ((Plus | Minus) <star_slash_percent>)*    [Binary]
<star_slash_percent> ::= <unary> (Star | Slash | Percent <unary>)*    [Binary]
<unary> ::= (Bang | Minus) <unary> |    [Unary]
            <element>
<element> ::= <call> (LeftSquare <expression> RightSquare)*    [Element]
<call> ::= <primary> (LeftParen (<expression> (Comma <expression>)*)? RightParen)*
    [Call]
<primary> ::= Literal |    [Literal]
              LeftParen <expression> RightParen |    [Grouping]
              LeftSquare (<expression> (Comma <expression>)*)? RightSquare |    [Array]
              LeftCurly (<expression> Colon <expression> (Comma <expression> Colon <
    expression>)*)? RightCurly |    [Dictionary]
              Identifier    [Variable]
```

Figure 2.7: BNF for expressions

Grammar

Regular languages cannot express recursive structures; hence, we must turn to **context-free grammars**. **Backus-Naur form (BNF)** will, therefore, be used to describe the **grammar** of the language.

Expressions must be evaluated with correct precedence (*Objective 2*). Hence, the constructed AST must also have the correct precedence—in other words, higher precedence operations must be **subtrees** of lower precedence operations so that they can be evaluated first.

With some careful thinking, the BNF for this language's expressions can be written as shown in Figure 2.7 (with the `ExprType` variant that will be used for each production in red). The production rules are listed from lowest to highest precedence.

To test these rules, we can see that the expression

```
a = x or (y == 5) and 1 < 2 == 3 != 4 and true
```

is correctly interpreted as

```
a = (x or (((y == 5) and (((1 < 2) == 3)) != 4) and true)).
```

Note that the productions for `<assignment>` and `<unary>` are described **recursively**. This will make it easier to construct a **right associative** parse tree for these rules, i.e., we want `a = b = c = 5` to be interpreted as `a = (b = (c = 5))` rather than `((a = b) = c) = 5`.

Parsing expressions

The **recursive** nature of the expressions suggests that a recursive algorithm may be best suited to parse the expressions. A popular parsing algorithm is **recursive descent**. Listing 4 outlines a general case of this algorithm.

Essentially, a function will be written for each production. Each production will construct its expression by calling other functions to build the subexpressions, starting with the lowest precedence production (`<expression>` in our case). This will lead to **recursive calls**, either **directly** (in the cases of `<assignment>` and `<unary>`) or **indirectly** (e.g., when `<expression>` is called again for the rule `<primary> ::= LeftParen <expression> RightParen`).

In other words, each function will build a **subtree** of the AST by calling other function(s) to build *its* subtree(s).

To write the parser for this language, we can adapt Listing 4 slightly, using while loops and if statements to handle the `*`s and `?`s respectively. In the case of the `*`, we can **recursively** update the production variable with itself to end up with one 'big' production which contains all the instances.

Listing 4 General recursive descent parsing

```
function A()                                                    ▷ Parse production <A>
    production ← [ ]
    for symbol X_i in rule <A> ::= X_1 X_2 ... X_n do
        if X_i is a non-terminal then
            Append X_i() to production
        else                                                    ▷ X_i is a terminal
            if current token matches X_i then
                Append current token to production
                Advance to next token
            else
                raise error          ▷ Raise specific error to situation, e.g., ExpectedCharacter
            end if
        end if
    end for
    return production
end function
```

Note that the recursive descent in our language will always terminate: our **base case** is in the `<primary>` production, and although it may call `<expression>` again, it will consume one `LeftParen` in the process.

The following listings illustrate how recursive descent can be implemented in practice for this language: Listing 5 shows the handling of the `?` in the `<assignment>` production using **direct recursion**; Listing 6 shows the handling of the `*` in the `<equality>` production using a while loop; Listing 7 shows the **direct recursive** implementation of the `<unary>` production; and Listing 8 show how the structure of the arguments list in the `<call>` production can be handled by **recursively** calling the `<expression>` production. Other productions follow similar implementations.

Listing 5 Handling the `?` in `<assignment>` using **direct recursion** with recursive descent

```
function ASSIGNMENT()                                          ▷ Parse production <assignment>
    expr ← OR()                  ▷ This may or may not end up being the target of the Assignment
    if current token is Equal then                                          ▷ Recursive case
        Advance to next token
        value ← ASSIGNMENT()     ▷ The value of the Assignment could be another Assignment, so recur
        expr ← Assignment {
            target: expr,                ▷ The first expression becomes the target of the Assignment
            value: value
        }
    end if
    return expr wrapped in an Expr object
end function
```

Figures 2.8, 2.9, and 2.10 showcase three example trace diagrams of the parsing algorithm and the resulting abstract syntax trees. Figure 2.8 is a basic case; Figure 2.9 illustrates the **direct recursion** of the `assignment()` and `unary()` functions; and Figure 2.10 illustrates the `expression()` function being recursively called (albeit indirectly) to capture the contents of the parentheses.

Listing 6 Handling the `*` in `<equality>` with recursive descent

```
function EQUALITY()                                          ▷ Parse production <equality>
    expr ← COMPARISON()                                      ▷ May become the left of the Binary
    while current token is EqualEqual or BangEqual do
        operator ← current token
        Advance to next token
        right ← COMPARISON()
        expr ← Binary {
            left:  expr,                                     ▷ The previous expr goes on the left, keeping the left associativity
            operator:  operator,
            right:  right
        }
    end while
    return expr wrapped in an Expr object
end function
```

Listing 7 Implementing `<unary>` using **direct recursion** with recursive descent

```
function UNARY()                                             ▷ Parse production <unary>
    if current token is Bang or Minus then                   ▷ Recursive case
        operator ← current token
        Advance to next token
        right ← UNARY()                                      ▷ As we can have nested unary expressions (e.g., !!true), recur
        expr ← Unary {
            operator:  operator,
            right:  right
        }
    else
        expr ← ELEMENT()                                     ▷ Base case
    end if
    return expr wrapped in an Expr object
end function
```

Listing 8 Handling the arguments list in `<call>` with recursive descent

```
function CALL()                                              ▷ Parse production <call>
    expr ← PRIMARY()                                         ▷ May become the callee of the call
    while current token is LeftParen do                      ▷ Functions may return functions
        Advance to next token
        arguments ← [ ]
        if current token is not RightParen then              ▷ i.e., if there are arguments, not just f()
            loop
                Append EXPRESSION() to arguments             ▷ Indirect recursive call back to <expression>
                if current token is not Comma then           ▷ Reached the end of the arguments
                    break
                end if
            end loop
        end if
    end while
    EXPECT(RightParen)          ▷ Raise an error if the current token is not RightParen; otherwise, advance
    expr ← Call {
        callee:  expr,
        arguments:  arguments
    }
    return expr wrapped in an Expr object
end function
```

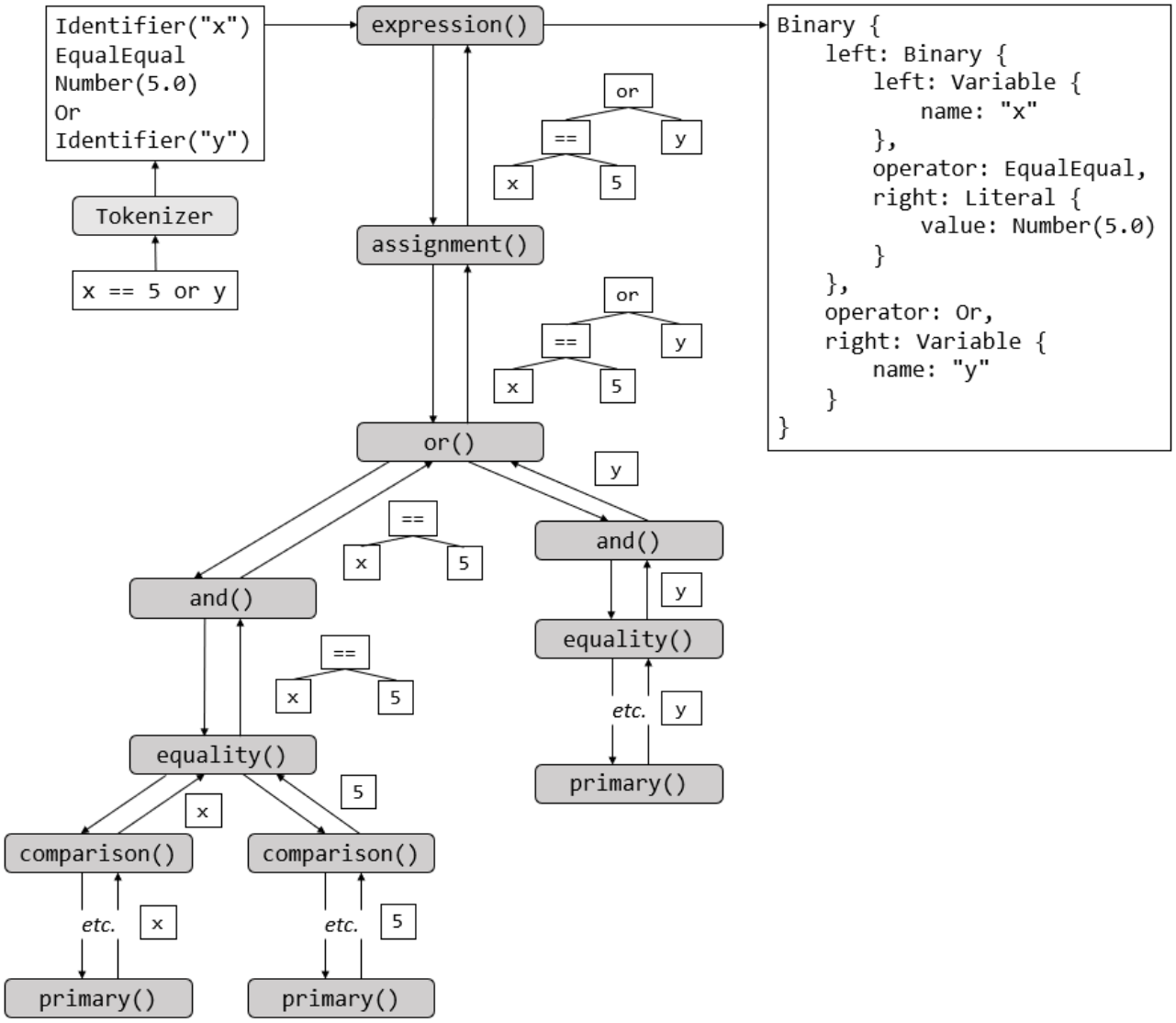

Figure 2.8: Function calls for the parsing of `x == 5 or y`

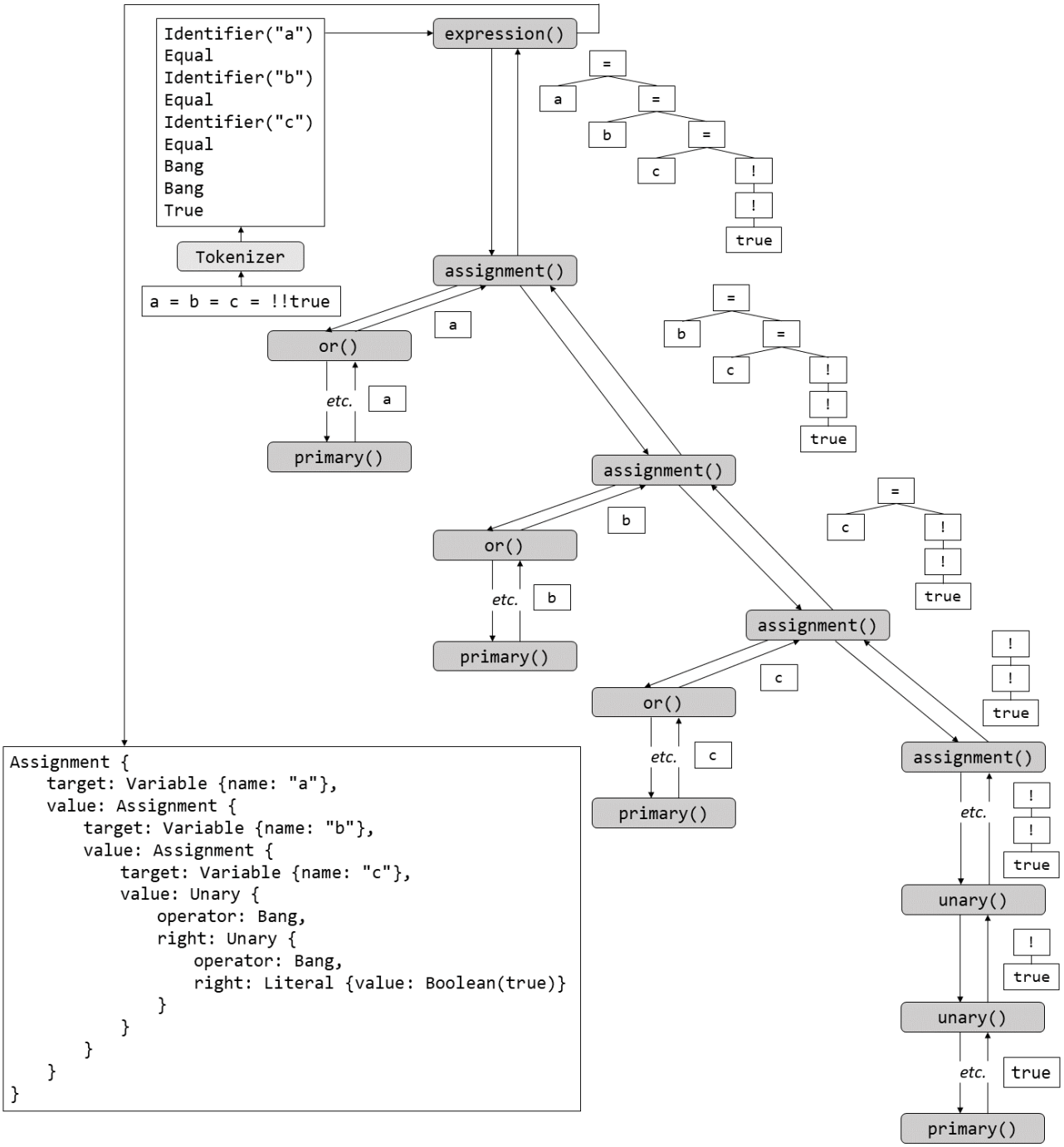

Figure 2.9: **Direct recursive** function calls for the parsing of `a = b = c = !!true`

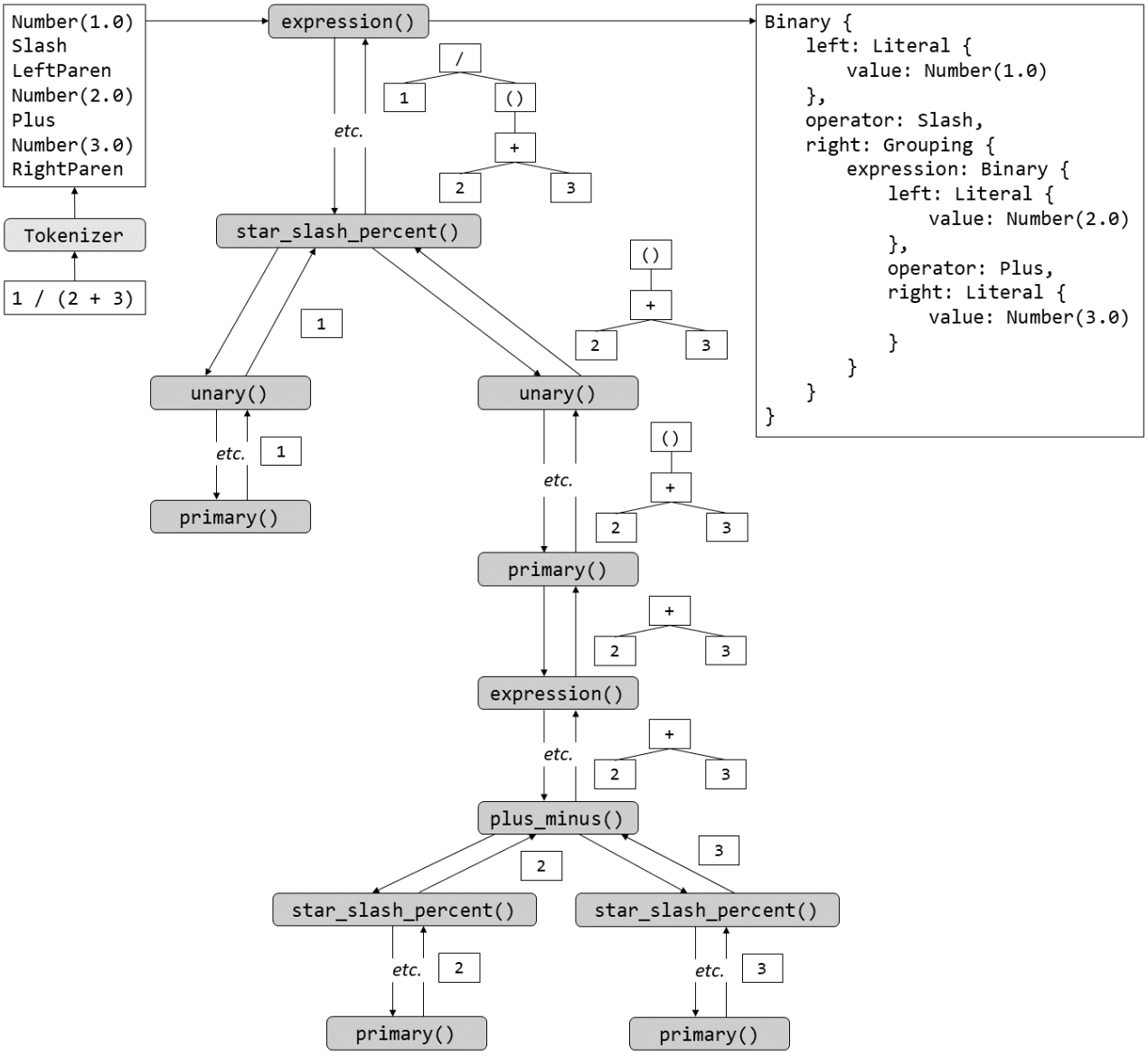

Figure 2.10: **Indirect recursion** in the parsing of `1 / (2 + 3)`

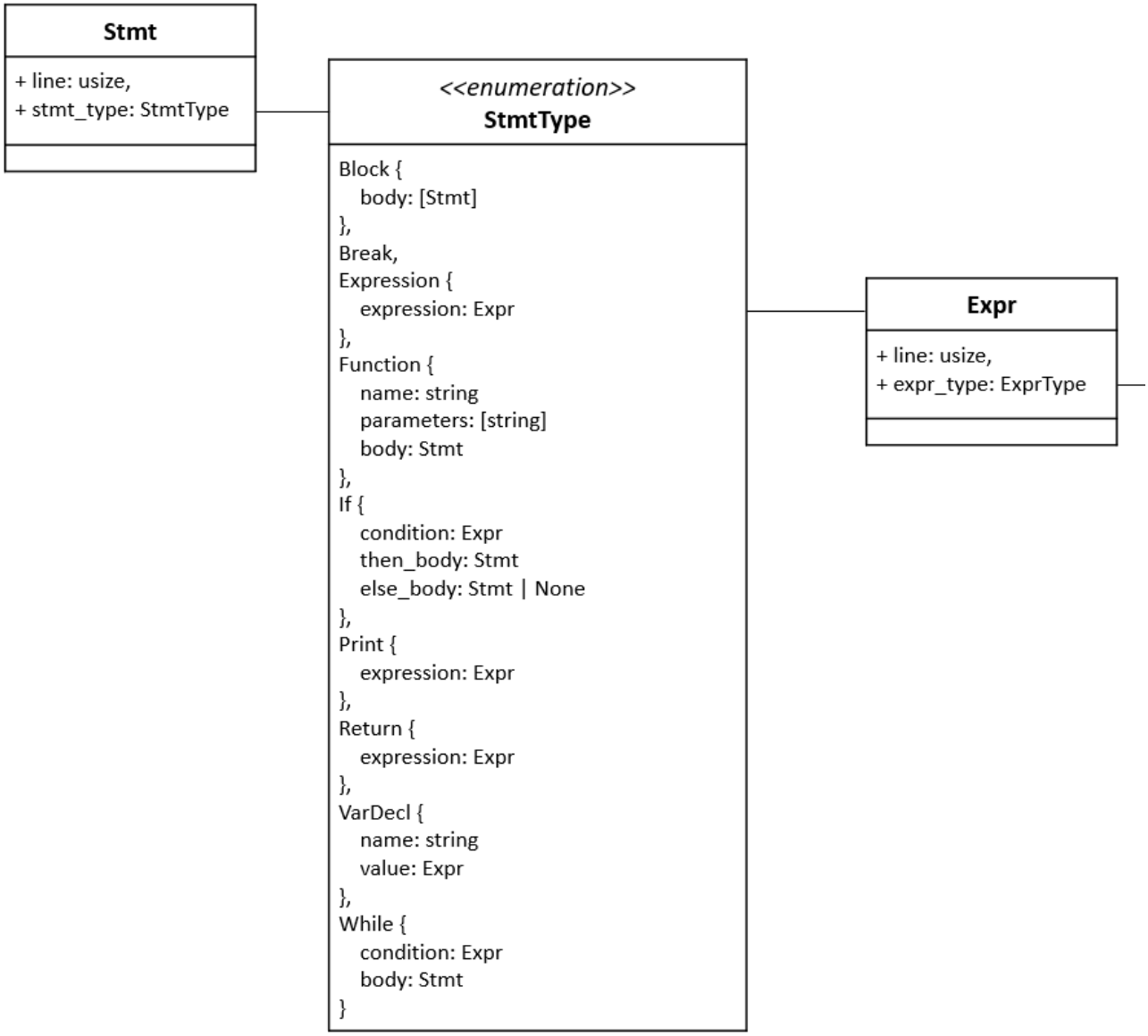

Figure 2.11: Class diagram of `Stmt` class and related types

2.3.3 Statements

The `Stmt` class

Similarly, a `StmtType` enum will be used to represent the different statement structures. Again, Rust's ability to store named fields within enum variants will be used. `StmtType`'s variants are listed in Table 2.7 with their names, intended uses, stored information, and objectives they were designed to help complete.

Note that, as will be explained later, for loops will internally represented as a while loops; hence, there is no `For` variant.

Like we did with the `Expr` class, we will wrap this enum within a `Stmt` class which will store the line number of the statement which will be used in debugging. Figure 2.11 illustrates the `Stmt` class and related types.

Table 2.7: Variants of `StmtType`

Name	Use	Information stored	Objective(s)
`Block`	Blocks of statements enclosed within curly brackets; used in if structures, loops, and functions; represents a variable scope	• `body` (an array of `Stmt` objects): the statements in the block	5, 6a, 6b, 7a, 9
`Break`	Break statements	N/A	6c
`Expression`	Statements can be expressions, e.g., variable assignments and function calls	• `expression` (an `Expr` object): the expression	3, 4, 7a
`Function`	Function declarations	• `name` (a string): the name of the function • `parameters` (an array of strings): the function parameters • `body` (a `Stmt` object; in practice this will be a `Block` variant): the function body	7a, 8
`If`	If statements	• `condition` (an `Expr` object): the expression for the condition of the if statement • `then_body` (a `Stmt` object; in practice this will be a `Block` variant): the 'then' body of the if statement • `else_body` (an optional `Stmt` object; in practice this will be a `Block` variant): the optional 'else' body of the if statement	5
`Print`	Print statements	• `expression` (an `Expr` object): the expression whose evaluated value is to be printed	1
`Return`	Return statements	• `expression` (an `Expr` object): the expression whose evaluated value is to be returned	7b
`VarDecl`	Variable declarations	• `name` (a string): the name of the variable to be declared • `value` (an `Expr` object): the expression whose evaluated value is to be stored in the variable	3
`While`	While and for loops	• `condition` (an `Expr` object): the expression for the condition of the loop • `body` (a `Stmt` object; in practice this will be a `Block` variant): the body of the loop	6a, 6b

```
<program> ::= <statement>* EOF
<statement> ::= <break> |
                <for> |
                <function> |
                <if> |
                <print> |
                <return> |
                <var> |
                <while> |
                <expression>
<block> ::= LeftCurly <statement>* RightCurly    [Block]
<break> ::= Break    [Break]
<for> ::= For LeftParen <statement>? Semicolon <expression>? Semicolon <statement>?
    RightParen <block>    [While]
<function> ::= Func Identifier LeftParen (Identifier (Comma Identifier)*)? RightParen <
    block>    [Function]
<if> ::= If LeftParen <expression> RightParen <block> <else>?    [If]
<else> ::= Else <if> | Else <block>    [If.else_body]
<print> ::= Print <expression>    [Print]
<return> ::= Return <expression>    [Return]
<var> ::= Var Identifier Equal <expression>    [VarDecl]
<while> ::= While LeftParen <expression> RightParen <block>    [While]
```

Figure 2.12: BNF for statements

Grammar

The BNF for this language's statements is shown in Figure 2.12, with the respective `StmtType` variants shown in red.

Although this BNF is technically correct, its implementation can be problematic: in order to correctly choose which production to follow from `<statement>`, there must be some element of backtracking. There is also a similar problem when choosing whether or not to follow the `<else>` production from the `<if>` production—the parser does not know, at that point, whether or not an `Else` token follows.

To avoid these problems, we can note that the first token for each production is unique. Hence, the BNF can be refactored slightly to the one shown in Figure 2.13. This formulation will be the one used when implementing the parser. Note that a 'program' is then just a sequence of `<statement>` productions.

Parsing statements

Recursive descent, as explained in Subsection 2.3.2, will also be used to parse statements.

Possibly the most interesting thing in this process is the conversion of for loops to while loops for internal representation. This is called *syntactic desugaring*—in other words, for loops were just there for ease of use for the user. In general,

```
for (initialiser; condition; increment) {
        body
}
```

can be converted to

```
{
        initialiser
        while (condition) {
                {
                        body
                }
                increment
        }
}
```

```
<program> ::= <statement>* EOF
<statement> ::= Break |    [Break]
                For <for> |
                Func <function> |
                If <if> |
                Print <print> |
                Return <return> |
                Var <var> |
                While <while> |
                <expression>
<block> ::= LeftCurly <statement>* RightCurly    [Block]
<for> ::= LeftParen <statement>? Semicolon <expression>? Semicolon <statement>?
    RightParen <block>    [While]
<function> ::= Identifier LeftParen (Identifier (Comma Identifier)*)? RightParen <block>
       [Function]
<if> ::= LeftParen <expression> RightParen <block> (Else <else>)?    [If]
<else> ::= If <if> | <block>    [If.else_body]
<print> ::= <expression>    [Print]
<return> ::= <expression>    [Return]
<var> ::= Identifier Equal <expression>    [VarDecl]
<while> ::= LeftParen <expression> RightParen <block>    [While]
```

Figure 2.13: Adapted BNF for statements

where curly brackets indicate a new variable scope, or a `Block`. Listing 9 illustrates how this procedure can be implemented.

Using the BNF in Figure 2.13, we can see the implementation for the `<statement>` production simply involves a big switch statement. Listing 10 shows the implementation of this production, and Listing 11 demonstrates how the `<if>` and `<else>` productions can be implemented as examples—the other productions follow a similar pattern.

As the parsing of if, else if, and else statements uses complex **indirect recursive techniques**, the diagram in Figure 2.14 illustrates how the abstract syntax tree is built; note the **recursive** use of the `If` data structure.

Listing 9 Desugaring for loops. EXPECT() is used as short-hand for raising an error if the token does not match; in reality, we can explicitly check with an if statement and issue a **specific** error as per Objective 10a.

```
function FOR()                                                    ▷ Parse production <for>
    EXPECT(LeftParen)          ▷ Raise an error if the current token is not LeftParen; otherwise, advance
    if current token is not Semicolon then                        ▷ If an initialiser was given...
        initialiser ← STATEMENT()
    else
        initialiser ← None
    end if
    EXPECT(Semicolon)
    if current token is not Semicolon then                        ▷ If a condition was given...
        condition ← EXPRESSION()
    else
        condition ← Expr::Literal with value true
    end if
    EXPECT(Semicolon)
    if current token is not RightParen then                       ▷ If an incrementing statement was given...
        increment ← STATEMENT()
    else
        increment ← None
    end if
    EXPECT(RightParen)
    for_body ← BLOCK()
    if increment is not None then          ▷ Add incrementing statement to end of the while loop body
        while_body ← Block {body: for_body + increment}
    else
        while_body ← Block {body: for_body}
    end if
    while_loop ← While {
        condition: condition,                                     ▷ Uses the same condition as the for loop
        body: while_body
    }
    if initialiser is not None then                               ▷ Add initialiser before the while loop
        while_block ← Block {body: initialiser + while_loop}
    else
        while_block ← Block {body: while_loop}
    end if
    return while_block
end function
```

Listing 10 Parsing of the `<statement>` production

```
function STATEMENT()                                  ▷ Parse production <statement>
    switch current token do
        case Break
            Advance token
            return StmtType::Break wrapped in a Stmt object
        case For
            Advance token
            return FOR()
        case Func
            Advance token
            return FUNC()
        case If
            Advance token
            return IF()
        case Print
            Advance token
            return PRINT()
        case Return
            Advance token
            return RETURN()
        case Var
            Advance token
            return VAR()
        case While
            Advance token
            return WHILE()
        otherwise
            expr ← EXPRESSION()
            return expr wrapped in a Stmt object
    end switch
end function
```

Listing 11 Parsing of the `<if>` and `<else>` productions, which is further illustrated in Figure 2.14. Again, EXPECT() is used as short-hand for raising an error if the token does not match; in reality, we can explicitly check the token and issue a **specific** error as per Objective 10a.

```
function IF()                                          ▷ Parse production <if>
    EXPECT(LeftParen)
    condition ← EXPRESSION()                           ▷ The condition of the if statement
    EXPECT(RightParen)
    then_body ← BLOCK()                                ▷ The '{ then }' of the if statement
    if current token is Else then                      ▷ This could be followed by another If token, forming an 'else if'
        Advance token
        else_body ← ELSE()
        return If {
            condition:  condition,
            then_body:  then_body,
            else_body:  else_body
        }
    else
        return If {
            condition:  condition,
            then_body:  then_body,
            else_body: None
        }
    end if
end function

function ELSE()                                        ▷ Parse production <else>
    if current token is If then                        ▷ i.e., forming an 'else if' statement
        Advance token
        return IF()
    else
        return BLOCK()
    end if
end function
```

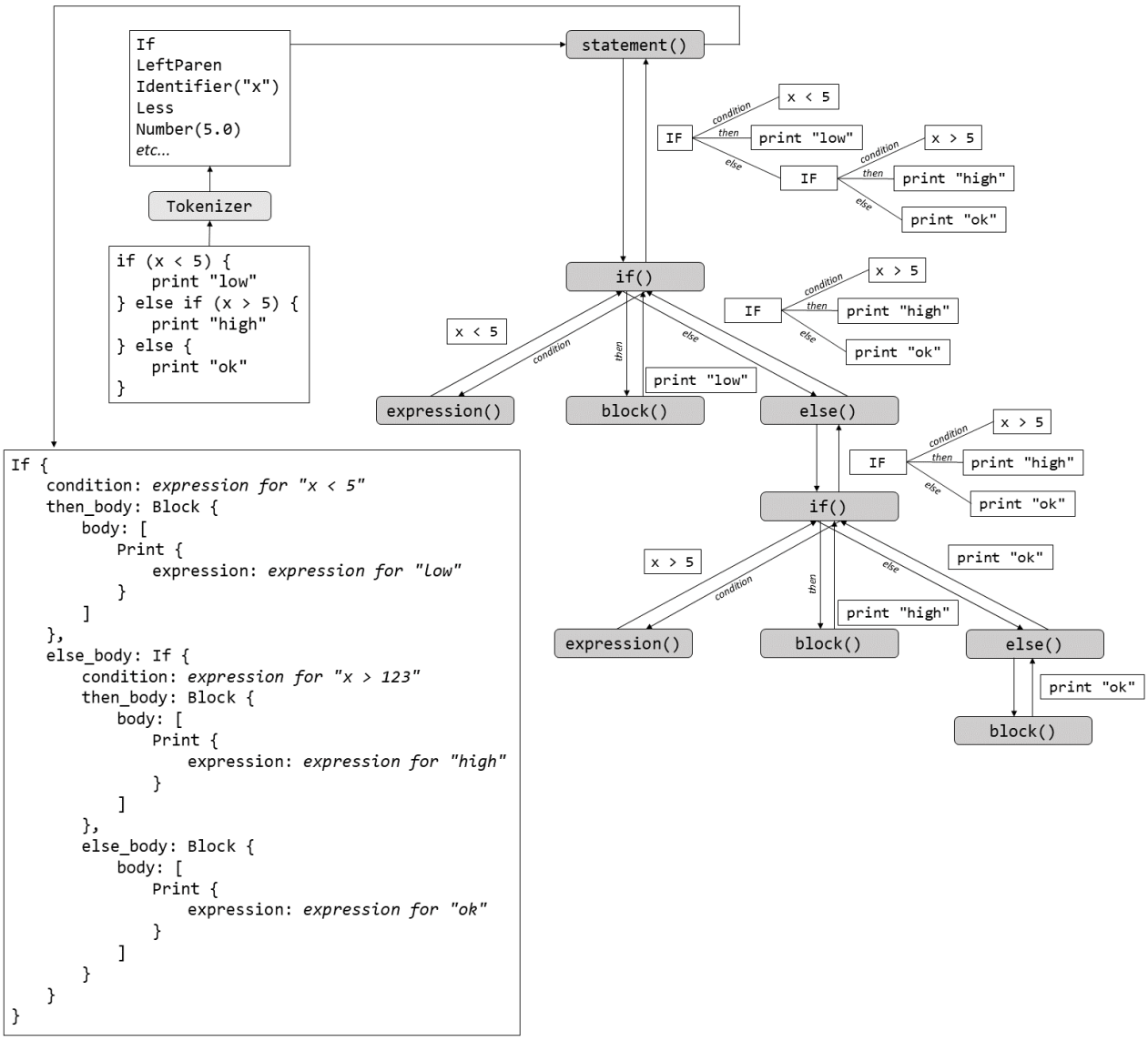

Figure 2.14: Parsing of if, else if, and else statements as described in Listing 11

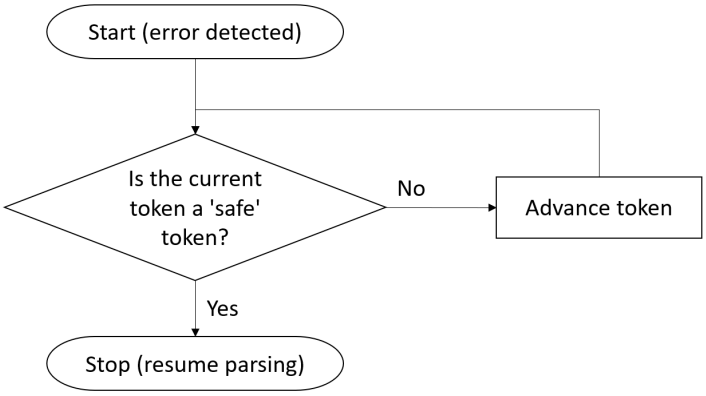

Figure 2.15: Flowchart illustrating the synchronisation process

2.3.4 Synchronisation and error handling

If an error occurred during parsing, the parser must collect as many errors as possible before reporting the errors and terminating the source code execution (*Objective 10b*). Hence, when an error occurs, the parser must advance to the next 'safe' token where parsing can be resumed; this technique is known as **synchronisation**. The tokens which start a new statement will be considered 'safe' tokens, all of which are listed as follows:

- `EOF`
- `For`
- `Func`
- `If`
- `Print`
- `Return`
- `Var`
- `While`

The algorithm is relatively simple and can be summarised in the flowchart in Figure 2.15.

Since recursive descent parsing works with a **call stack**, we can **raise** errors once they happen in the production functions so they can be *bubbled up* the call stack. A single driver function can then repeatedly collect the errors and synchronise to collect more errors. Once as many as possible have been collected, the driver function can then report them all at once by calling `report_errors` (see Subsection 2.1.2). Raised errors will be a variant of the `ErrorType` enum which will allow details of the error (line number, etc.) to be collected (see Subsection 2.1.1).

2.3.5 Driver function

In order to implement synchronisation within the parsing process, a single driver function will need to coordinate the procedure. The flowchart in Figure 2.16 illustrates the operation of the driver function.

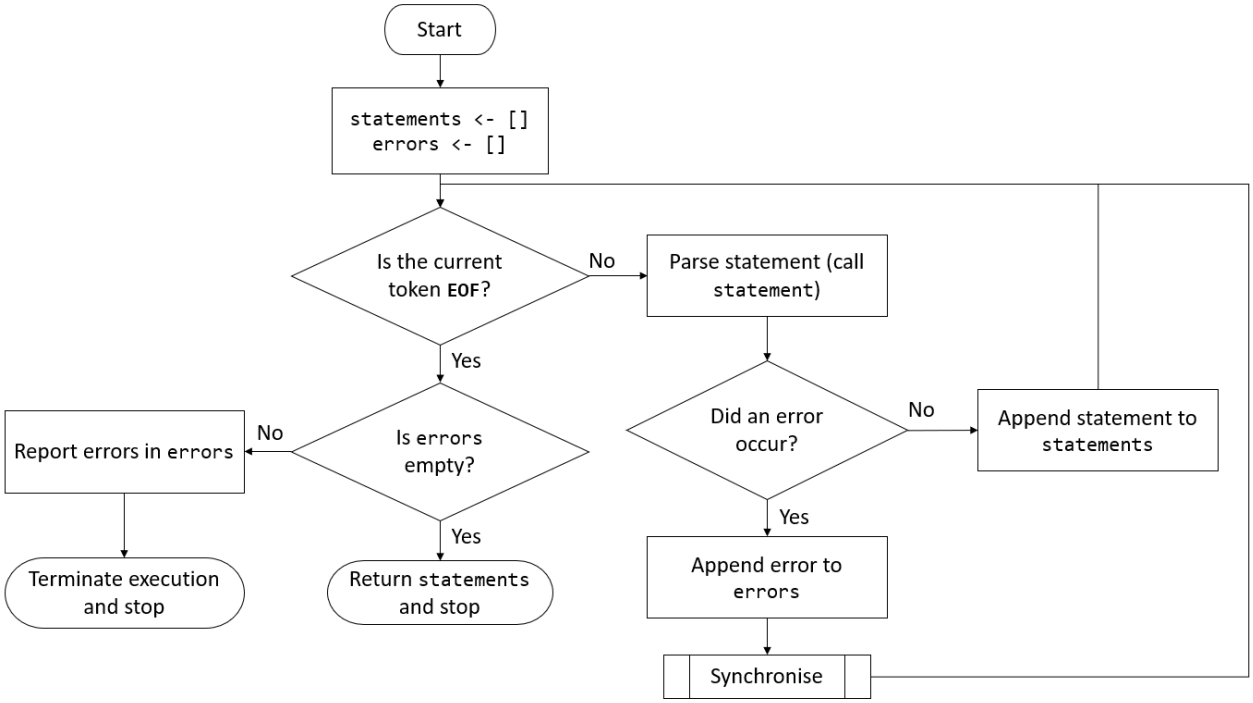

Figure 2.16: Flowchart outlining operation of driver function for parsing

2.3.6 The `Parser` class

The parser will be wrapped within its own class in order to keep it independent from other components. The `Parser` class will have the following **private attributes**:

- `tokens` (an array of `Token` objects)
  The input token sequence.
- `current_index` (usize)
  An index pointing to the current token.
- `current_line` (usize)
  The current line number. This will be used to construct `Expr` and `Stmt` objects.

The class will also have the following **methods**:

- `new(tokens: [Token])` (public)
  Constructs a `Parser` object with the given token sequence.
- `parse() : [Stmt]` (public)
  The interface method is effectively the driver function described in Figure 2.16. It parses the sequence of tokens and returns an abstract syntax tree (i.e., a sequence of statements) and deals with raised errors.
- `sync()` (private)
  Synchronises the parser as described in Figure 2.15.
- `statement() : Stmt, block() : Stmt, etc.` (private)
  These construct a particular productions for **statements** in **recursive descent**.
- `expression() : Expr, assignment() : Expr, etc.` (private)
  These construct a particular productions for **expressions** in recursive descent.
- `check_and_consume(expected_types: [TokenType]) : (Token | None)` (private)
  A helper function: if the current token's type is one of `expected_types`, it returns the token object and advances `current_index`. Otherwise, it returns `None`.
- `check_next(expected_types: [TokenType]) : bool` (private)
  A helper function: if the current token's type is one of `expected_types`, it returns the true and *does not* advance the pointer. Otherwise, it returns false.
- `expect(expected_type: TokenType, expected_char: char) : ErrorType` (private)
  A helper function which **raises** a generic `ExpectedCharacter` error if the current token's type is not `expected_type` to be bubbled up (`expected_char` is used as one of the fields for the `ExpectedCharacter` error). The other specific errors listed in Table 2.2 are issued by using explicit if statements. For example, we can explicitly check if an `Identifier` token follows a `var` keyword in a `<var>` production. If this is not the case, a *specific* `ExpectedVariableName` error can be issued (*Objective 10a*).

Figure 2.17 summarises the class and related types.

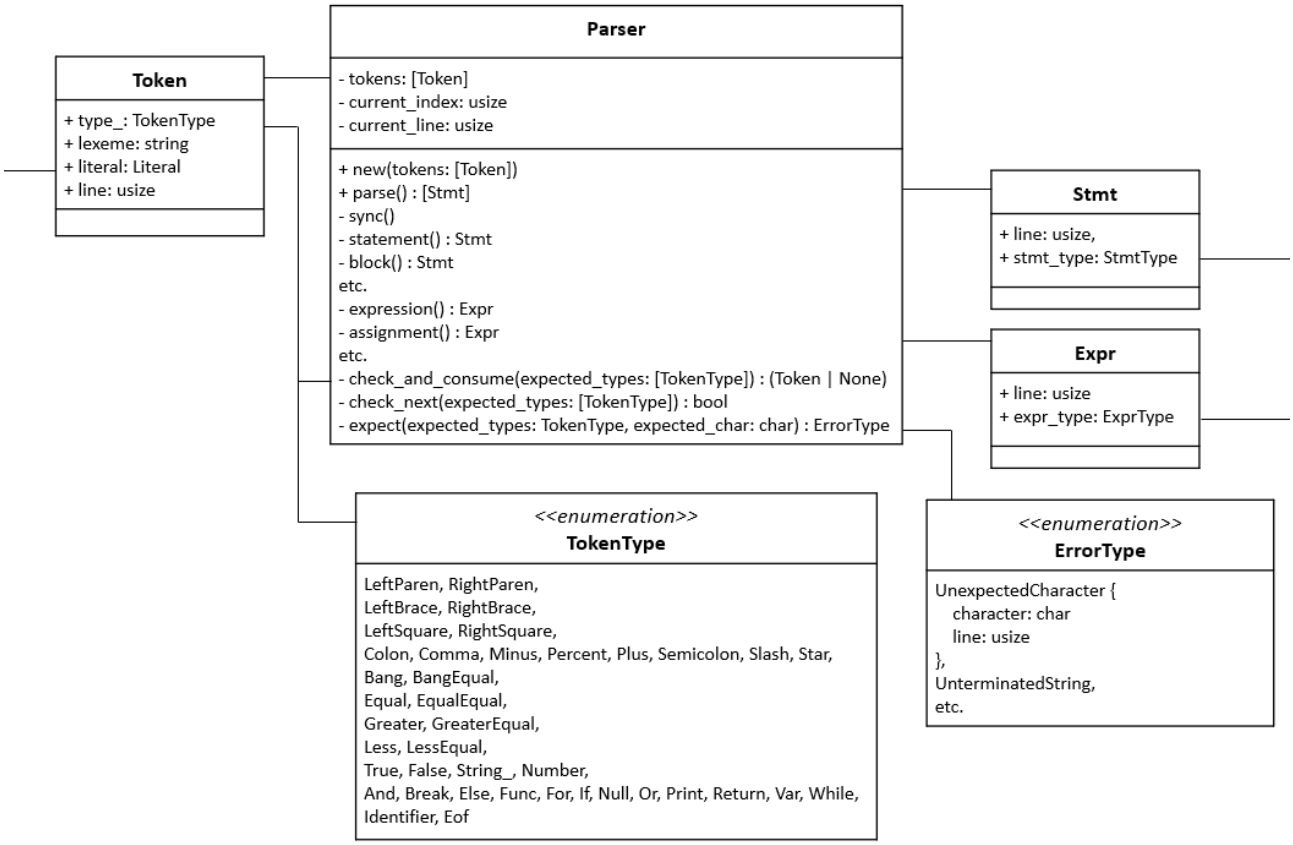

Figure 2.17: Class diagram of the `Parser` class and related types

Table 2.8: Variants of `BuiltinFunction`

Name	Objective
`Input`	8a
`Append`	8b
`Remove`	8c
`ToNumber`	8d
`ToString`	8e
`Size`	8f
`Sort`	8g

2.4 The `Value` enum

2.4.1 Overview

The next three sections concern the *evaluation* of **expressions** into **values**, and the *execution* of **statements**. Values are essentially anything that can be stored in the environment—numbers, strings, functions, etc. Values will be represented and stored using the `Value` enum.

2.4.2 The `BuiltinFunction` enum

In order to achieve Objective 8, the language must be support built-in functions. Most languages do this by implementing built-in functions in the language itself, then importing it. However, as we do not have the ability to do this, the evaluation and execution of built-in functions must be done within the interpreter.

Essentially, the mapping of 'function name' to '`BuiltinFunction` variant' will be pre-stored in the environment's base scope. Once the name is used to call a built-in function, the environment will return the respective `BuiltinFunction` variant (in the same way that the environment will return any user-defined function). The interpreter can then switch on the variant and perform the function's operations.

Further details can be found in Sections 2.6 and 2.7, but for now the variants will be introduced in Table 2.8 along with the objective each was designed to help achieve.

2.4.3 The enum

Table 2.9 lists the variants of the `Value` enum. Note the **recursive** definition of the `Array` variant, which stores the elements in the array using the `Value` enum.

The class diagram in Figure 2.18 summarises the enum and its related types.

2.4.4 Printing values

In order to implement the printing functionality (*Objective 1*), the interpreter must **recursively** print the values in the case of the `Array` and `Dictionary` variants. This is because arrays and dictionaries may contain more arrays and dictionaries. Listing 12 demonstrates this.

Table 2.9: Variants of `Value`

Name	Information stored	Objective(s)
`Number`	• A float: the evaluated numerical value	3a
`String_`	• A string: the evaluated string	3c
`Bool`	• A Boolean value: the evaluated Boolean value	3b
`Array`	• An array of `Value` variants: the evaluated array elements	3d
`Dictionary`	• A `HashTable` object: the internal representation of the dictionary (discussed in Section 2.5)	3e
`Function`	• `parameters` (an array of strings): the parameter names • `body` (a variant of the `Stmt` enum): the body of the function which will be a `Block` variant	7a
`BuiltinFunction`	• A `BuiltinFunction` variant: the type of the built-in function	8
`Null`	N/A	3f

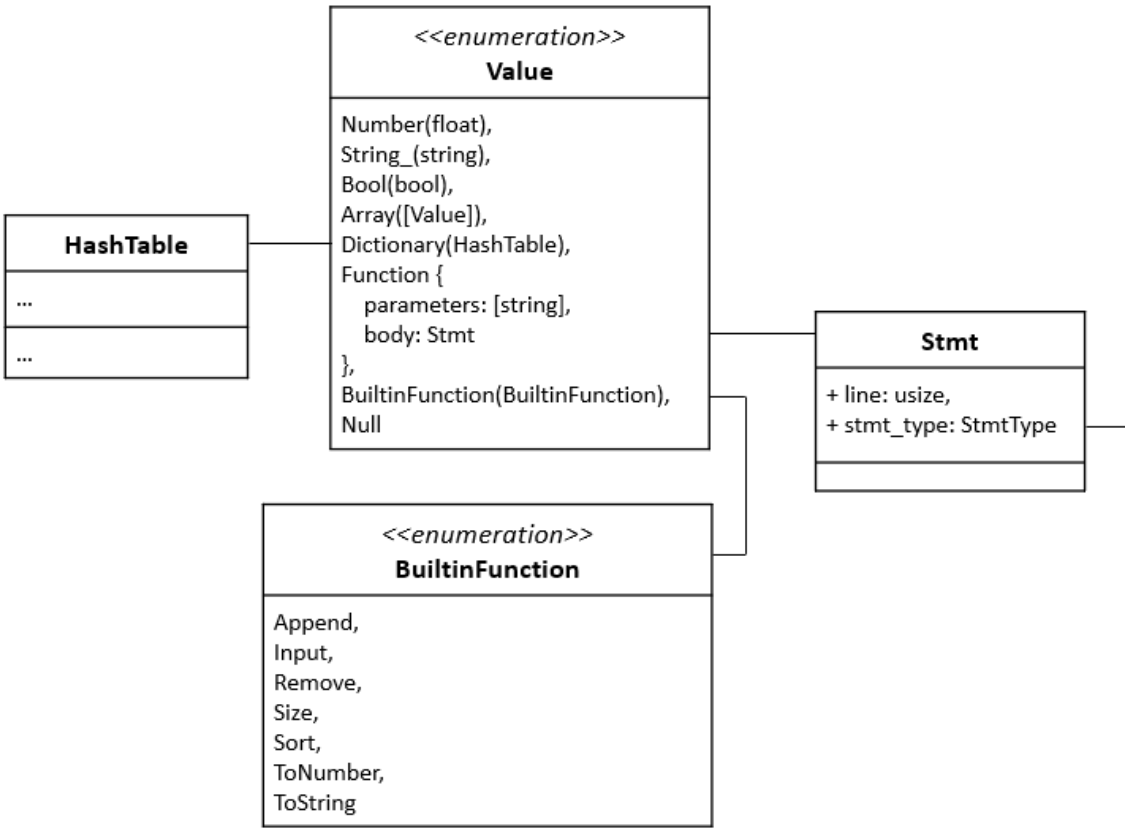

Figure 2.18: The `Value` enum and related types

Listing 12 Printing values using **recursion**

```
function PRINT_VALUE(value)                                   ▷ Prints the given value
    switch value do
        case Number(x)                                        ▷ Base cases
            Print x
        case String_(x)
            Print x
        case Bool(x)
            Print x
        case Array(array)                                     ▷ Recursive cases
            Print "["
            for x in array do
                PRINT_VALUE(x)  ▷ Recursively print the element, which may be another array or dictionary
                if not last then
                    Print ","
                end if
            end for
            Print "]"
        case Dictionary(dict)                                 ▷ dict is a HashTable object (see Section 2.5)
            flattened ← dict.FLATTEN()                        ▷ Returns an array of key-value pairs
            for key, value in flattened do
                PRINT_VALUE(key)                              ▷ Recursive call to print the key
                Print ":"
                PRINT_VALUE(value)                            ▷ Recursive call to print the value
                if not last then
                    Print ","
                end if
            end for
        case Function or BuiltinFunction
            Print "<function>"
        case Null
            Print "Null"
    end switch
end function
```

2.5 Dictionaries

2.5.1 Overview

The language must support the storage of dictionaries and the access and modification of values of their key-value pairs (*Objectives 3e and 4b*). There must also be a built-in function that removes a key-value pair from a dictionary (*Objective 8c*), but the details of this will be explained in Subsection 2.7.6.

Internally, dictionaries will be represented as **hash tables**. The key feature of hash tables is the access and storage of key-value pairs in **constant time**.

2.5.2 The `KeyValue` class

`KeyValue<T>` objects will be used to represent key-value pairs in the hash table. It will have the following **public attributes** which store the key and value respectively:

- `key` (type `T`)
- `value` (type `T`)

The reason for using a **generic type** `T` is that this class can then be used for key-value pairs *before* they are evaluated (i.e., still expressions, in which case `T` would be an `Expr` object) *and after* they are evaluated (i.e., they have been evaluated to a single value, in which case `T` would be a `Value`[4] variant).

2.5.3 Representation

Internally, a **2D array** will be used to store the hash table which will be a private attribute, `array`. In other words, each element in the array will store a **variable-length array**, called a **bucket**. Buckets are needed to implement **separate chaining** which is a way of handling **hash collisions**.

2.5.4 Hash function

Overview

Hash functions should ideally have **constant time operation**. It should also roughly **uniformly distribute** the resulting keys and must give the same hash for the same key. As the derivation of hash functions is beyond the scope of my ability, I will use algorithms found on the Internet.

A hash function is needed for each `Value` variant:

- `Array` and `String`
  The `djb2`[5] algorithm for strings will be adapted for arrays, outlined in the Listing 13. In the case of arrays, we **recursively** hash each element and combine into a total, as an element may be another array. For 1D arrays and strings, the trick to keep it $O(1)$ is to only use the first k elements of the sequence where k is constant. We will set $k = 300$. For recursive structures (arrays in arrays), there is some nuance to this trick that will be discussed later.

Listing 13 The `djb2` algorithm for hashing 1D arrays and strings

```
function DJB2(sequence)
    hash_value ← 5381
    for x in first k elements of sequence do
        hash_value ← hash_value × 33 + HASH(x) ▷ Hash the element, which may result in a recursive call for multi-dimensional arrays
    end for
    return hash_value
end function
```

- `Bool`
  Return 1 for true, 2 for false.
- `Dictionary`
  Raise a `CannotHashDictionary` error for reasons explained in the next part.

[4] The `Value` enum stores *evaluated* values in the interpreter and does not have anything to do with the 'value' of a key-value pair in a hash table; keys *and* values in a hash table will be represented using the `Value` enum.

[5] https://theartincode.stanis.me/008-djb2/

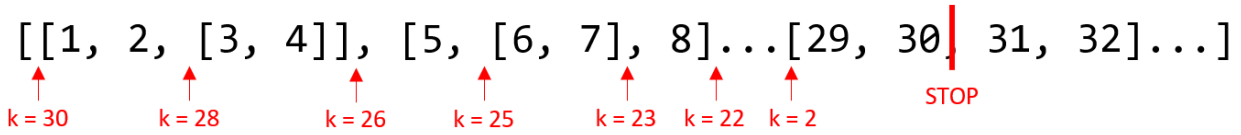

Figure 2.19: Running hash counter with multi-dimensional arrays and $k = 30$

- `Function`
  It is not possible to definitively compare functions, so we will simply raise a `CannotHashFunction` error.
- `Null`
  Return 3.
- `Number`
  Note that we are using floats to represent numbers. Hence, we must discard a number of least-significant bits to account for floating point accuracy. We will discard the 12 least-significant bits by using a **logical right shift**. We will then use the Knuth Variant on Division[6] which is $x(x + 3)$ to obtain the hash.

Limiting the number of elements hashed

In recursive structures, simply hashing the first, say, 300 elements will not work—for example, if the first 300 elements of an array were 300-element arrays, we would end up with $300 \times 300 = 90000$ operations[7]. This clearly is not constant time.

Hence, the hash function must also keep track of a 'global' number of elements left to hash in the recursion. This principle is demonstrated with $k = 30$ in Figure 2.19.

This is also why I have decided against allowing the user to use a dictionary as a key in another dictionary. Consider the dictionaries with the following entries: $d_1 = \{A, B, C\}$, and $d_2 = \{C, B, A\}$ in the order the entries were inserted. Clearly, they should hash to the same value. However, if the hashes of the keys of A, B, and C were equal they would all be stored in the same bucket but in a different order due to separate chaining.

Hence, it is possible that, with $k = 2$, the hash of d_1 would only consider A and B, and the hash of d_2 would only consider C and B, resulting in different hashes. Therefore, more sophisticated techniques possibly involving representing the hash table with a segment tree-like structure would be necessary to maintain hashing in sub-linear time, which is beyond the scope of this NEA.

To implement the 'hash counter' for arrays, an `elements_left` parameter will be part of the hash function. The function will also return an updated counter along with the hash in a tuple of the form `(hash, elements_left)`. The **recursive** hash algorithm is outlined in pseudocode in Listing 14 and Figure 2.20 illustrates the algorithm with an example.

[6] https://www.cs.hmc.edu/~geoff/classes/hmc.cs070.200101/homework10/hashfuncs.html
[7] We will assume Booleans, numbers, and `Null` are hashed in one operation.

Listing 14 Recursive hashing

```
function HASH(key, elements_left)                    ▷ Hash the first elements_left elements of key
    switch key do
        case Bool(b: bool)                                                          ▷ Base cases
            if b is true then
                return (1, elements_left − 1)
            else
                return (2, elements_left − 1)
            end if
        case Null
            return (0, elements_left − 1)
        case Number(x: float)
            x ← x right-shift by 12 bits
            return (x(x + 3), elements_left − 1)
        case String_(s: string)
            hash_value ← 5381
            index ← 0
            while elements_left > 0 and index < length of s do
                hash_value ← hash_value × 33 + ASCII(s[index])
                elements_left ← elements_left − 1
                index ← index + 1
            end while
            return (hash_value, elements_left)
        case Dictionary or Function                  ▷ For above reasons, refuse to hash these
            raise CannotHashDictionary or CannotHashFunction error
        case Array(array: [Value])                                          ▷ Recursive case
            hash_value ← 5381
            index ← 0
            while elements_left > 0 and index < length of array do   ▷ elements_left will change based on the recursive call
                (curr_hash, elements_left) ← HASH(array[index], elements_left)   ▷ Update elements_left using result of recursive call
                hash_value ← hash_value × 33 + curr_hash
                index ← index + 1
            end while
            return (hash_value, elements_left)
    end switch
end function
```

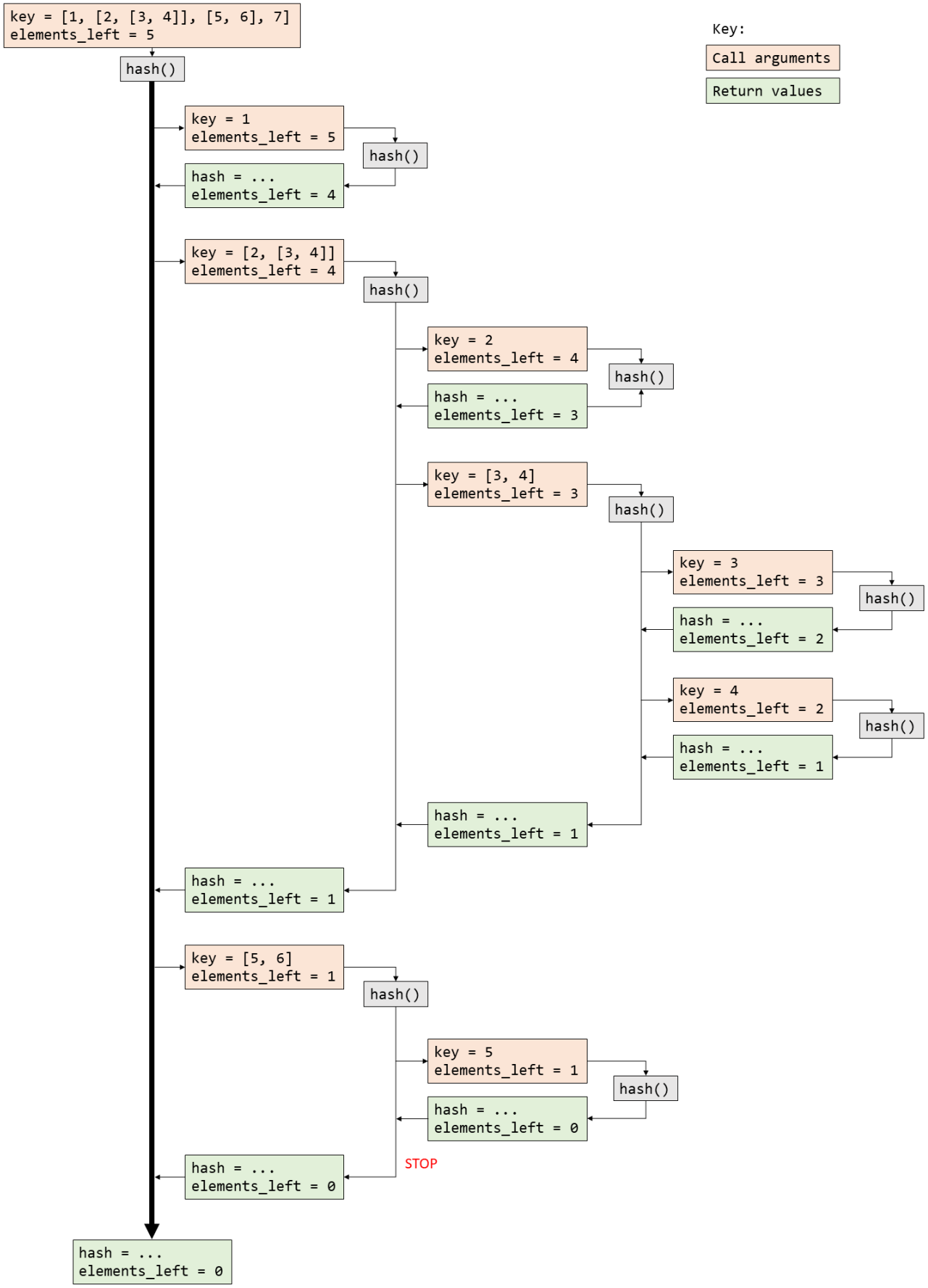

Figure 2.20: Trace diagram of **recursive** hashing strategy

2.5.5 Constants

It is good practice to store hardcoded configuration values in **constants**. The following will be used to store the configuration of the hash table:

- `INITIAL_NUM_BUCKETS = 16` (usize)
  The number of buckets in a newly initialised hash table.
- `MAX_NUM_BUCKETS = 65536` (usize)
  The maximum number of buckets in a hash table (the hash table may grow as more entries are added).
- `MAX_CALC = 65381` (usize)
  This will be used as the **modulus** in *intermediate* hash calculations to prevent overflow. Note that it is **prime** to avoid patterns in the hashes.
- `MAX_LOAD_FACTOR_NUMERATOR = 3`, `MAX_LOAD_FACTOR_DENOMINATOR = 4` (usize)
  The **load factor** of the table is defined as

  $$\text{load factor} = \frac{\text{number of entries}}{\text{number of buckets}}$$

  The maximum load factor is set to $\frac{3}{4}$, and once the table's load factor exceeds this, the table is **rehashed**. In order to keep arithmetic purely in integers, the fraction is split into numerator and denominator.
- `HASH_FIRST_N = 300` (usize)
  This is the number of elements to hash as described previously.

2.5.6 The `HashTable` class

Attributes and methods

The hash table will be implemented within a `HashTable` class with the following **private attributes**:

- `array` (a 2D array of `KeyValue<Value>` objects)
  The internal array of the hash table. Initialised as an array of `INITIAL_NUM_BUCKETS` empty arrays.
- `entries` (usize)
  The number of entries in the table.
- `current_num_buckets` (usize)
  The current number of buckets in the table.

It will also have the following **private methods**:

- `get_bucket_number(key: Value, line: usize) : usize`
  A helper function which returns the bucket number given the value of the key. It simply returns `hash(key) mod self.current_num_buckets`. Note the argument `line` is the line number and will be used as part of an `ErrorType` variant in case an error is raised either within this function or within a call of another—this argument will be common to most methods.
- `check_load(line: usize)`
  Checks whether or not the load factor of the table is greater than the maximum load factor. If so, rehashes the table. Its operation is illustrated in Listing 15.

The class' **public interface methods** are as follows:

- `new()`
  Constructs a new `HashTable` instance. Initialises the internal array to be a 2D array with `INITIAL_NUM_BUCKETS` empty arrays.
- `get(key: Value, line: usize) : Value`
  Returns the value associated with the key. Listing 16 outlines this method.
- `insert(key: Value, value: Value, line: usize) : Value`
  Inserts the key-value pair into the table if it does not exist already; otherwise, updates the existing pair with the new value. This is demonstrated in Listing 17.

- `remove(key: Value, line: usize)`
  Removes the key-value pair from the table, as in Listing 18.

- `size() : usize`
  Returns the number of entries in the table.

- `flatten() : [KeyValue<Value>]`
  Returns the 1D representation of the table, i.e., a 1D array of key-value pairs. This is used when printing the dictionary, rehashing, and comparing hash tables.

Note that, as the hash function is **static**, it will be placed outside the class.

Listing 15 `check_load` method

```
function CHECK_LOAD(line)  ▷ Checks whether load factor is greater than maximum load factor; rehashes table if necessary
    if self.current_num_buckets < MAX_NUM_BUCKETS then  ▷ If the hash table can still grow...
        if self.entries * MAX_LOAD_DENOMINATOR > self.current_num_buckets * MAX_LOAD_NUMERATOR then  ▷ and the load factor exceeds the maximum...
            copy ← self.FLATTEN()  ▷ Returns an array of key-value pairs in the hash table
            self.current_num_buckets ← self.current_num_buckets * 2  ▷ Double number of buckets
            self.array ← array of self.current_num_buckets empty arrays
            for entry in copy do
                self.INSERT(entry.key, entry.value, line)  ▷ Reinsert each entry into the table
            end for
        end if
    end if
end function
```

Listing 16 `get` method

```
function GET(key, line)  ▷ Returns the value associated with the key
    bucket_number ← GET_BUCKET_NUMBER(key)
    for key_value in self.array[bucket_number] do  ▷ Iterate through the bucket...
        if key_value.key = key then  ▷ until the key-value pair with the given key is found
            return key_value.value
        end if
    end for
    raise KeyError with key and line number  ▷ No entry was found with the given key
end function
```

Listing 17 `insert` method

```
function INSERT(key, value, line)  ▷ Inserts key-value pair into the table/updates existing pair
    bucket_number ← GET_BUCKET_NUMBER(key)
    found ← false
    for key_value in self.array[bucket_number] do  ▷ Iterate through the bucket...
        if key_value.key = key then  ▷ until the key-value pair with the given key is found
            key_value.value ← value  ▷ Update the existing pair
            found ← true
            break
        end if
    end for
    if not found then  ▷ Insert the new pair
        Append KeyValue{key: key, value: value} to self.array[bucket_number]
        self.entries ← self.entries + 1
    end if
    self.CHECK_LOAD(line)  ▷ In case the table needs rehashing
end function
```

Listing 18 `remove` method

```
function REMOVE(key, line)                          ▷ Removes the key-value associated with the given key
    bucket_number ← GET_BUCKET_NUMBER(key)
    for key_value in self.array[bucket_number] do    ▷ Iterate through the bucket...
        if key_value.key = key then                  ▷ until the key-value pair with the given key is found
            Remove entry
            return
        end if
    end for
    raise KeyError with key and line number          ▷ No entry was found with the given key
end function
```

Equality

We need to be able to test for the equality of two `HashTable` objects, as it will be used as part of a `Value` enum variant. Listing 19 presents a way to do this.

Listing 19 Equality of hash tables

```
function EQUALITY(h1, h2)                           ▷ Tests for equality between h1 and h2 HashTable objects
    f1 ← h1.FLATTEN()                               ▷ 1D array of entries
    f2 ← h2.FLATTEN()
    if f1 and f2 not same size then
        return false
    end if
    for entry in f1 do
        if an equivalent entry is in f2 then
            Remove that entry
        else
            return false                            ▷ There is an entry in h1 that is not in h2
        end if
    end for
    return true                                     ▷ All entries in h1 have a corresponding entry in h2
end function
```

Note that the algorithm runs in $O(n^2)$ time, where n is the size of the two arrays, because for each entry in f_1, we have to loop through f_2 to see if a corresponding entry exists.

If we were able to sort the entries, then we could do it in $O(n \log n)$, where n is the size of the largest of the two tables—both 'flattened' tables could be sorted and compared pair-wise. However, sorting requires comparison—we would have to be able to tell if a certain `Value` variant is 'larger' than another. This is problematic, as once again we have to compare hash tables (which is part of a `Value` variant) in constant time. Hence, given this setup, there is no easy way out of the $O(n^2)$ solution.

Class diagram

To summarise, Figure 2.21 shows the class diagram for the `HashTable` class and related types.

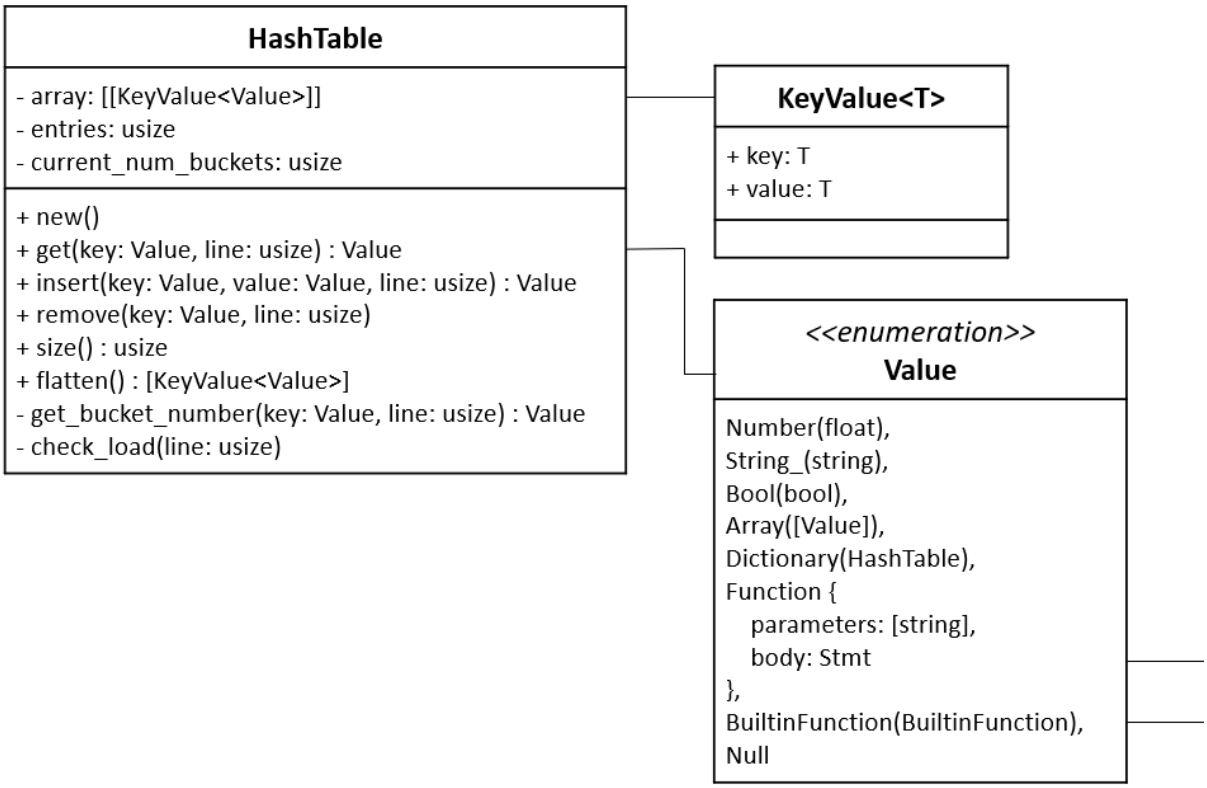

Figure 2.21: Class diagram of the `HashTable` class and related types

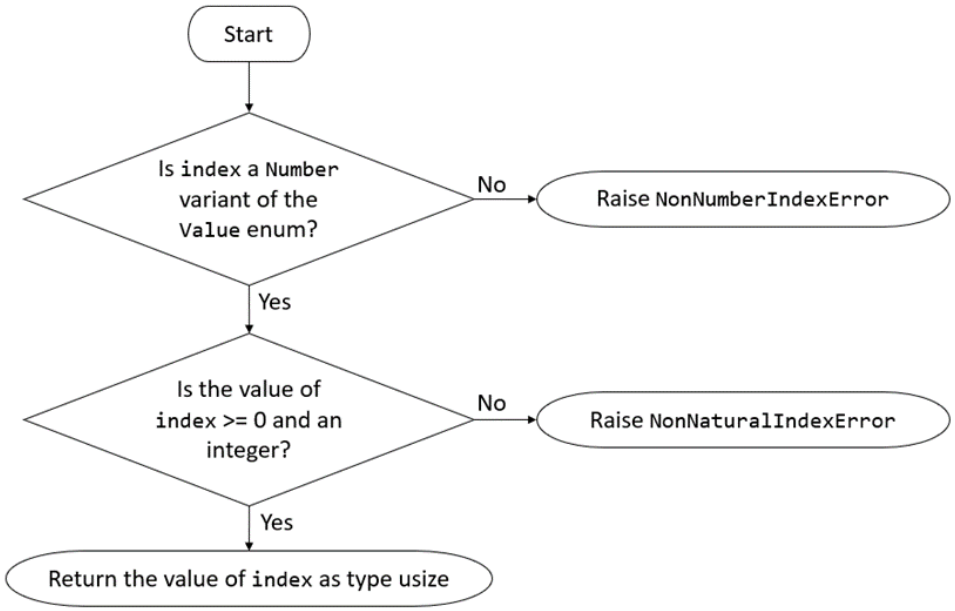

Figure 2.22: Conversion of `Value` types to array indices

2.6 Environment

2.6.1 Overview

During execution, the source code may store, access, and modify variables, and define and call functions (*Objectives 3, 4, 7a, and 8*). These operations must interact with the **environment**, which will be responsible for storing and updating variables and functions, and resolving names.

2.6.2 The `Pointer` class

It is possible that the environment will store multi-dimensional arrays and dictionaries. This is not a problem when accessing elements as it can be recursively evaluated by the interpreter[8]. However, there is a problem when attempting to update values in multi-dimensional structures, as the environment must know *all* the indices in order to update the correct value.

The `Pointer` class will store the following **public attributes**:

- `name` (string)
  The name of the 'base' array or dictionary variable, e.g., `a` in `a[1]["key"]`.[9]
- `indices` (an array of `Value` objects)
  The sequence of indices to the element, e.g., `[1, "key"]` in `a[1]["key"]`. Note that `Value` is used as the indices could also be dictionary keys.

The construction of `Pointer` objects is **recursive** and will be covered in Subsection 2.7.2.

When using `Pointer` objects to access *arrays*, the environment will need to check if each index can be used as an array index, i.e., the index is within $\{0, 1, 2, \ldots\}$. The flowchart in Figure 2.22 illustrates a helper function `index_value_to_usize` which will perform this conversion and the specific errors it may raise (*Objective 10a*).

2.6.3 Name scoping

Objective 9 calls for **name scoping**, in that names declared within a scope must not be accessible from outside, whereas names declared outside must be accessible from inside. Scopes are used in loops, functions, and control structures and are denoted by curly brackets `{ ... }`.

Each scope will be represented as a hash table. A **linked list** of scopes will be used to represent the scopes in the environment. Operations to do with the linked list will be explained in the next part.

[8] e.g., `a[1][2] -> [10,[20,30,40]][1][2] -> [20,30,40][2] -> 40`

[9] The `Pointer` class is only used when *updating* elements of *stored* variables only, in particular multi-dimensional structures. Hence, if the source code uses a *literal* array/dictionary or some other non-variable expression as the 'base' of an assignment target element, we will raise an error in the spirit of being strict with errors, which is part of the design philosophy. For example, we will not allow statements like `[1, 2][0] = 5`. Thus, we can assume the 'base' will always be the string name of a variable.

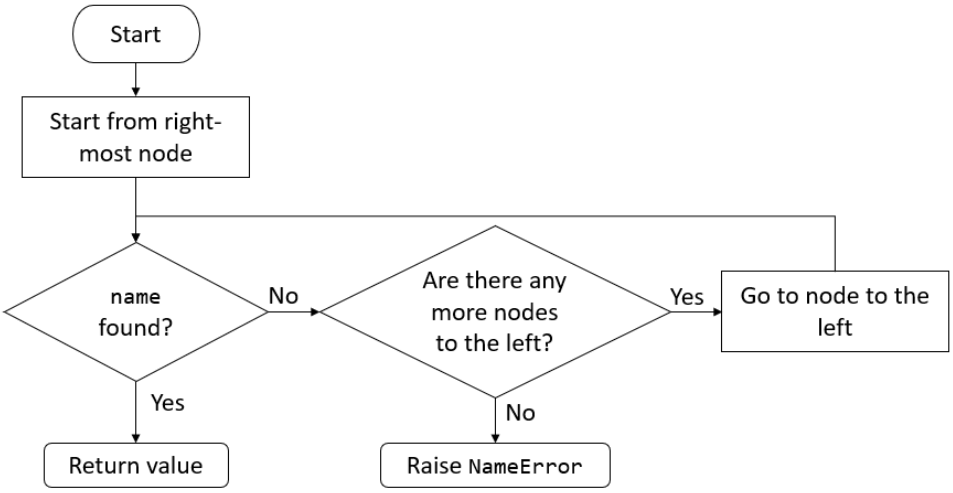

Figure 2.23: `get` method

2.6.4 The `Environment` class

The class will have one **private attribute**:

- `scopes` (an array[10] of Rust `HashMap`s that map strings to `Value` instances)
  The linked list storing the scopes of the environment.

It will also have the following **public methods**:

- `new()`
  Initialises a new `Environment` object. Importantly, it defines the string-to-`Value` mappings for the built-in functions in the base scope (the first node of the linked list), e.g., `"append"` maps to `BuiltinFunction::Append`. The evaluation of the built-in functions is done in the interpreter, so all the environment has to do is resolve the name of the function to the correct `BuiltinFunction` variant.

- `new_scope()`
  Appends a new scope to the end of the linked list with an empty `HashMap`.

- `exit_scope()`
  Removes the 'right-most' scope from the linked list. Raises an error if the linked list is empty.

- `declare(name: String, value: Value)`
  Insert the name-value pair into the right-most scope.

- `get(name: String, line: usize) : Value`
  Returns the value associated with `name`. As there could be multiple values associated with `name` across all the scopes, return the one in the right-most scope. To do this, we iterate from the right-most scope and check if `name` is declared in each scope, as described in Figure 2.23. If no associated value has been found by the time the left-most scope has been checked, then the name does not exist in the environment and we raise a `NameError`. The `line` argument will be used in the error.

- `update(pointer: Pointer, value: Value, line: usize)`
  Updates the value associated with the pointer. Again, update the one in the right-most scope only. However, this method is very nuanced. First, we find the object associated with the pointer's name from the right-most scope as we did in the `get` method. Then, we must **recursively** replace that object (which should be an array or dictionary) with itself using the pointer's indices *but not the last index*. This is because, in the case of dictionaries, the last key-value pair may not exist and should be inserted using the last index if this were the case. Thus, the resulting object should be an array, a dictionary, or a string (*Objective 4*). We can then use the last index to update (or insert, in the case of dictionaries) the correct element. Figure 2.24 outlines this, where `name` and `indices` are taken from the `Pointer` argument.

Figure 2.25 shows an example of the environment's operation and Figure 2.26 summarises the class and related types in a class diagram.

[10] I found out that Rust is notorious for not allowing native pointer-based linked lists, so I will use a variable-length array instead with the same principle of operation as a linked list.

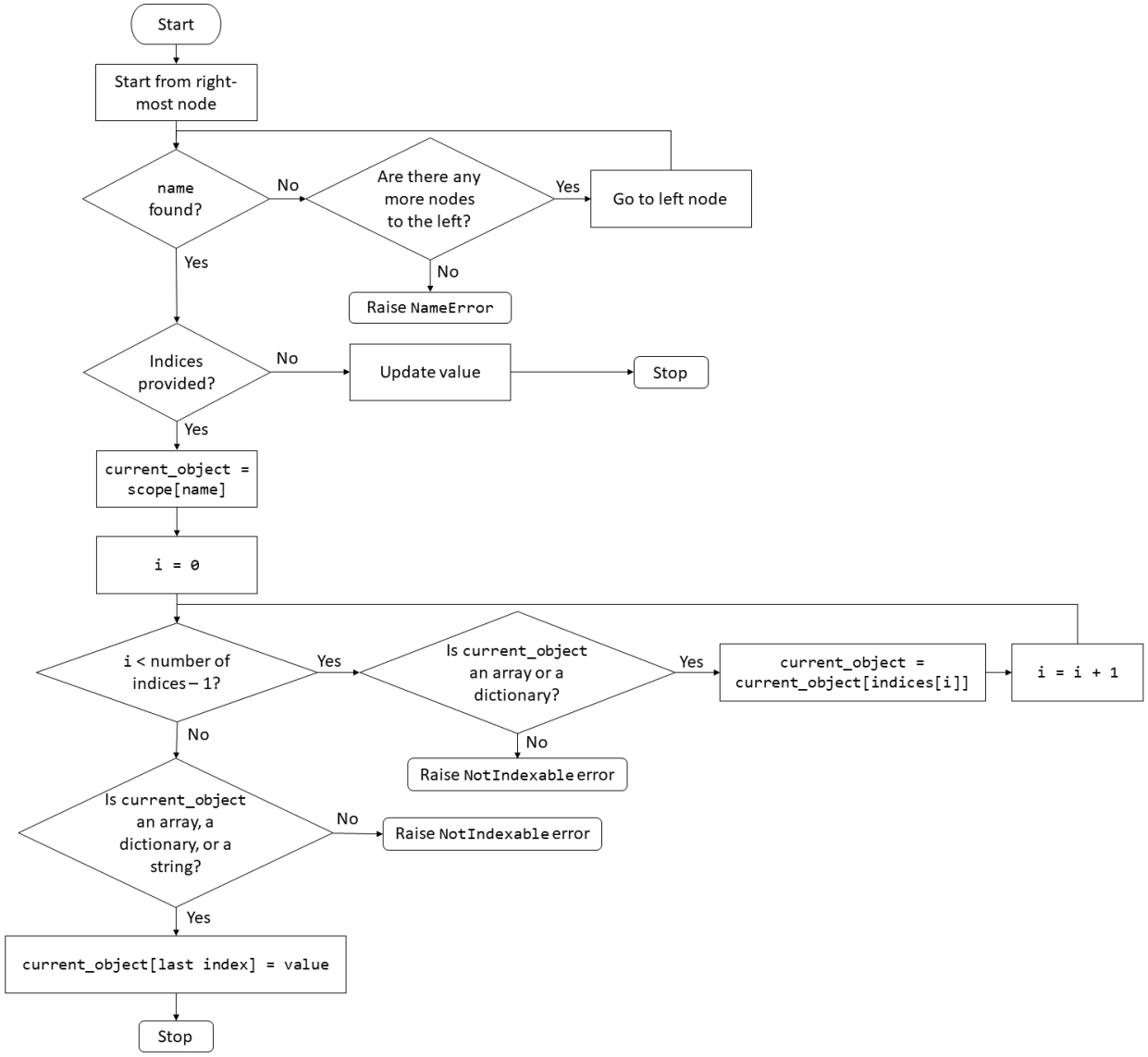

Figure 2.24: `update` method; note `name` and `indices` are taken from the `Pointer` instance

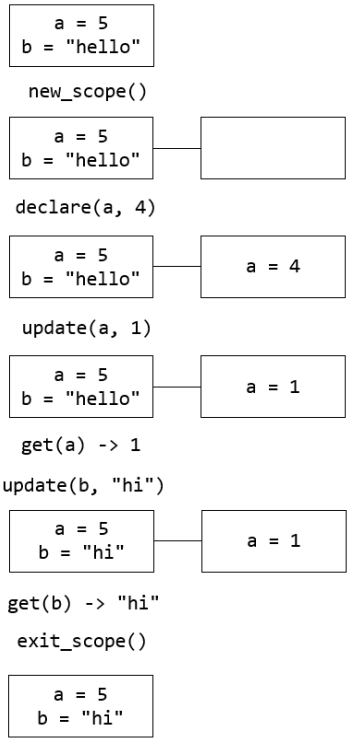

Figure 2.25: Example of environment's operation; note the simplified `Pointer` instances in the `update` calls

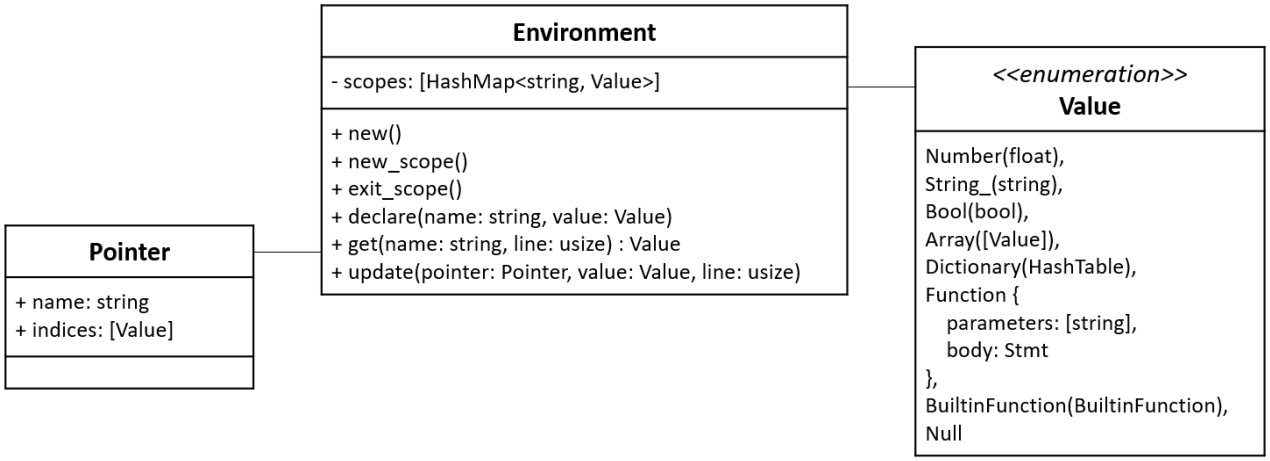

Figure 2.26: Class diagram of the `Environment` class and related types

2.7 Evaluation and execution

2.7.1 Overview

The component of the interpreter that performs the evaluation of expressions into values and the execution of statements is confusingly called the **interpreter**. To do this, the interpreter will **recursively traverse** the **abstract syntax tree (AST)** generated by the parser. Figure 2.27 provides a high-level overview of this process using the running example.

To implement the interpreter, we will define two functions: `evaluate` and `execute`, which will **recursively** call each other.

2.7.2 Pointer construction

A helper function which constructs pointers for use in the environment will be very useful when evaluating `Assignment` expressions. Essentially, we are concerned with deconstructing recursive `Element` expressions (i.e., **trees** of indices) into a name and a list of indices.

As mentioned in Subsection 2.6.2, we are only concerned with the 'base expression' being a variable—since `Pointer` will only be used for updating *stored* variables only, anything else used on the left-hand side of the `=` (literal arrays and dictionaries, results of function calls, etc.) will raise an error.

The function `construct_pointer(element, line)` will perform this process **recursively**:

- The **base case** is when the `element` argument is a `Variable` variant. We return a `Pointer` instance with `Pointer.name` as the variable name and an empty array of indices.
- The **recursive case** is when the `element` argument is an `Element` variant. We will make a recursive call using `Element.array` as the new `element` argument (as it could be another `Element` variant). Then, we will append the index of the `element` argument onto the list of indices returned by the recursive call.
- If the `element` argument is some other expression type, like a literal array or dictionary, then raise an `InvalidAssignmentTarget` error with the `line` number in the spirit of being strict with errors.

Figure 2.28 provides an illustration of this recursive process.

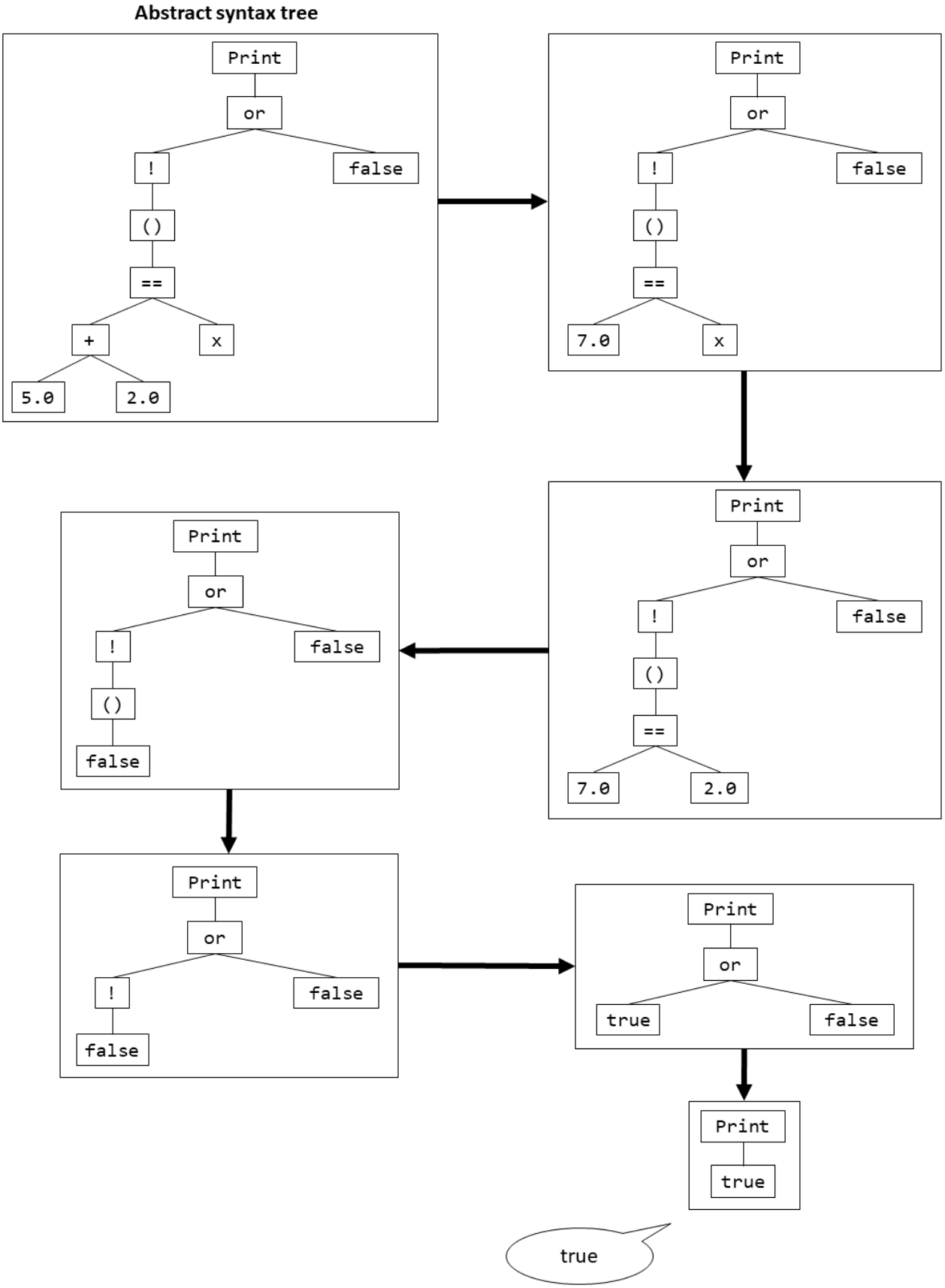

Figure 2.27: Example of evaluation and execution

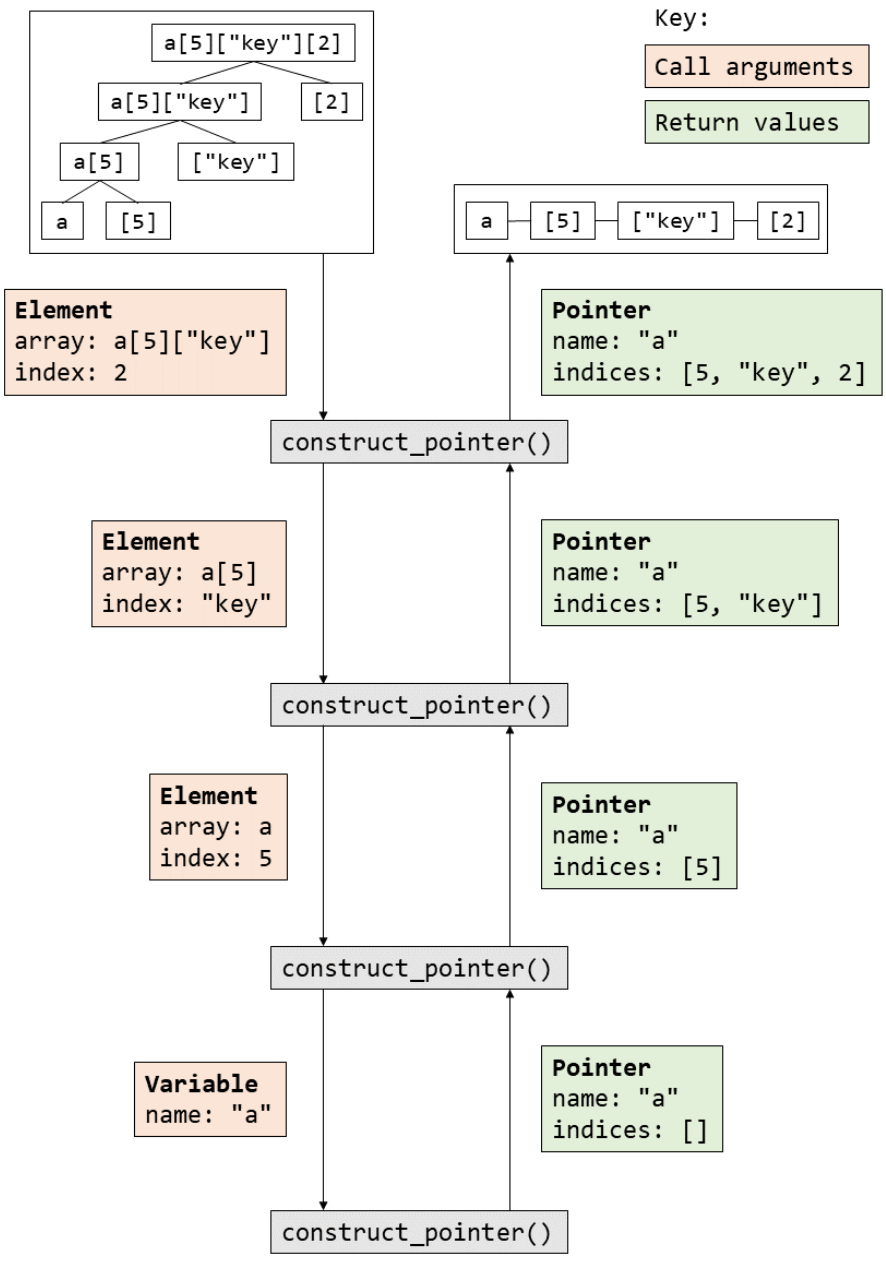

Figure 2.28: Constructing pointers recursively

2.7.3 Evaluating expressions

As expressions can form parts of other expressions, **recursion** will play a key role in evaluation. Because of this recursion, evaluation will start from the **leaves** of the AST. **Subtrees** of the AST will be evaluated to values first. Then, subtrees which uses *these* values will be evaluated, etc. Refer to Figure 2.27 for an example of the recursive nature of evaluation.

The function `evaluate(expr)` will return a `Value` variant. It will switch on the type of the expression and perform the corresponding operations. Note that the environment of the execution is stored in the class attribute `environment`; the complete structure of the class will be discussed later.

Listing 20 shows an excerpt of the `evaluate` function for the cases when the argument is an `Array`, `Assignment`, `Binary`, and `Dictionary`; other variants follow a similar **recursive** method. Function calls will be discussed later.

Listing 20 Evaluating expressions

```
function EVALUATE(expr)                                   ▷ Evaluates expr and returns a Value variant
    switch expr.expr_type do
        case ExprType::Array { elements }                 ▷ elements is an array of expressions
            values ← [ ]
            for element in elements do
                Append EVALUATE(element) to values        ▷ Note the recursion here as element is another expression
            end for
            return Value::Array(values)
        case ExprType::Assignment { target, value }       ▷ target and value are both expressions
            value_eval ← EVALUATE(value)                  ▷ Recursively evaluate the RHS of the assignment
            pointer ← CONSTRUCT_POINTER(target)           ▷ Pointer to the target of the assignment
            self.environment.UPDATE(pointer, value_eval, expr.line)   ▷ Update value in environment
            return value_eval
        case ExprType::Binary { left, operator, right }   ▷ left and right are expressions; operator is a token
            left_eval ← EVALUATE(left)                    ▷ Recursively evaluate the LHS and RHS of the binary expression
            right_eval ← EVALUATE(right)
            switch operator do
                case Or
                    if both LHS and RHS are Boolean then
                        return Value::Bool(left_eval or right_eval)
                    else
                        raise BinaryTypeError with expected = "Boolean", etc.   ▷ Expected both LHS and RHS to be Boolean
                    end if
                etc.
            end switch
        case ExprType::Dictionary { elements }            ▷ elements is an array of KeyValue<Expr> objects
            hash_table ← HashTable.new()                  ▷ The HashTable class as introduced in Section 2.5
            for key_expr, value_expr in elements do
                key_eval ← EVALUATE(key_expr)             ▷ Recursively evaluate the key and value
                value_eval ← EVALUATE(value_expr)
                hash_table.INSERT(key_eval, value_eval)   ▷ Insert the key-value pair into the hash table
            end for
            return Value::Dictionary(hash_table)
        etc.
    end switch
end function
```

2.7.4 Executing statements

The `execute(stmt)` function will execute statements. It will make use of the `evaluate` function to evaluate parts of the statement which are expressions; it will also make **recursive** calls to itself.

Listing 21 shows a part of the `execute` function for the `Block`, `Function`, `If`, and `Print` statements; other types of statements follow similar methods. Handling of break statements in loops and return statements in function calls will be discussed in the next subsection.

Listing 21 Executing statements

```
function EXECUTE(stmt)                                              ▷ Executes stmt
    switch stmt.stmt_type do
        case StmtType::Block { body }                               ▷ body is an array of statements
            self.environment.NEW_SCOPE()
            for block_stmt in body do
                EXECUTE(block_stmt)                                 ▷ Recursively execute each statement in the block
            end for
            self.environment.EXIT_SCOPE()
        case StmtType::Function { name, parameters, body }
            func ← Value::Function {
                parameters: parameters,
                body: body
            }
            self.environment.DECLARE(name, func)                    ▷ Declare function with name name in environment
        case StmtType::If { condition, then_body, else_body }       ▷ condition is an expression,
then_body and else_body are statements
            condition_eval ← EVALUATE(condition)                    ▷ Evaluate the condition
            if condition_eval is a Value::Bool then
                if condition_eval is true then
                    EXECUTE(then_body)                              ▷ Recursively execute then body
                else
                    EXECUTE(else_body)                              ▷ Recursively execute else body
                end if
            else
                raise IfConditionNotBoolean error                   ▷ Specific error as per Objective 10a
            end if
        case StmtType::Print { expression }
            evaluated ← EVALUATE(expression)
            Print evaluated                                         ▷ Simply print the evaluated expression
        etc.
    end switch
end function
```

2.7.5 Breaks and returns

Break and return statements must immediately stop evaluation and execution up to the 'parent process'—while statements for break; call expressions for return (*Objectives 6c and 7b*).

To do this within the **call stack**, we can **raise** 'dummy' errors. We will call these `ThrownBreak` and `ThrownReturn` respectively, and they will both be variants of the `ErrorType` enum; these correspond to the last two rows of Table 2.4. The `ThrownReturn` error will also contain a named `value` field (a `Value` variant) where the return value will be stored.

The idea is to *bubble up* these errors to their respective loop or call, which will immediately unwind the call stack.

Figure 2.29 demonstrates this principle for returns; if the error is not caught and reaches the `interpret` driver function (discussed later), execution will terminate and a suitable error message will be printed as the source code will have used a return statement *outside* a function (the same rule applies to break statements).

Figure 2.30 illustrates how these are handled by the loop and function call. Note that, as the loop and function bodies are forced into being `Block` statements during parsing, the creation and exiting of scopes will be handled by the execution of the `Block`.

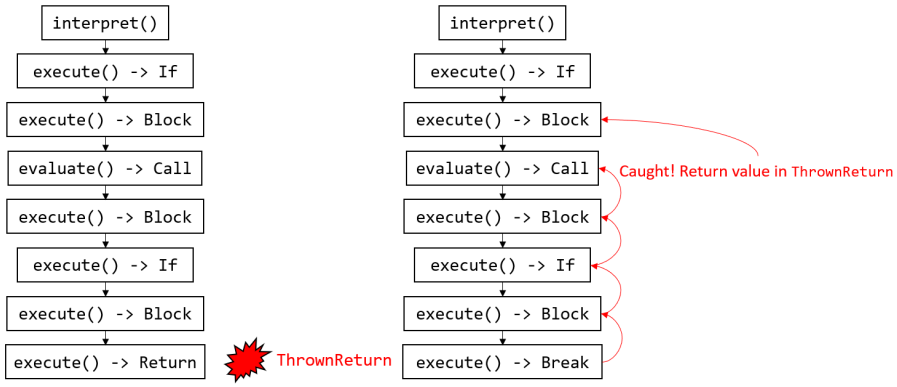

Figure 2.29: Throwing returns

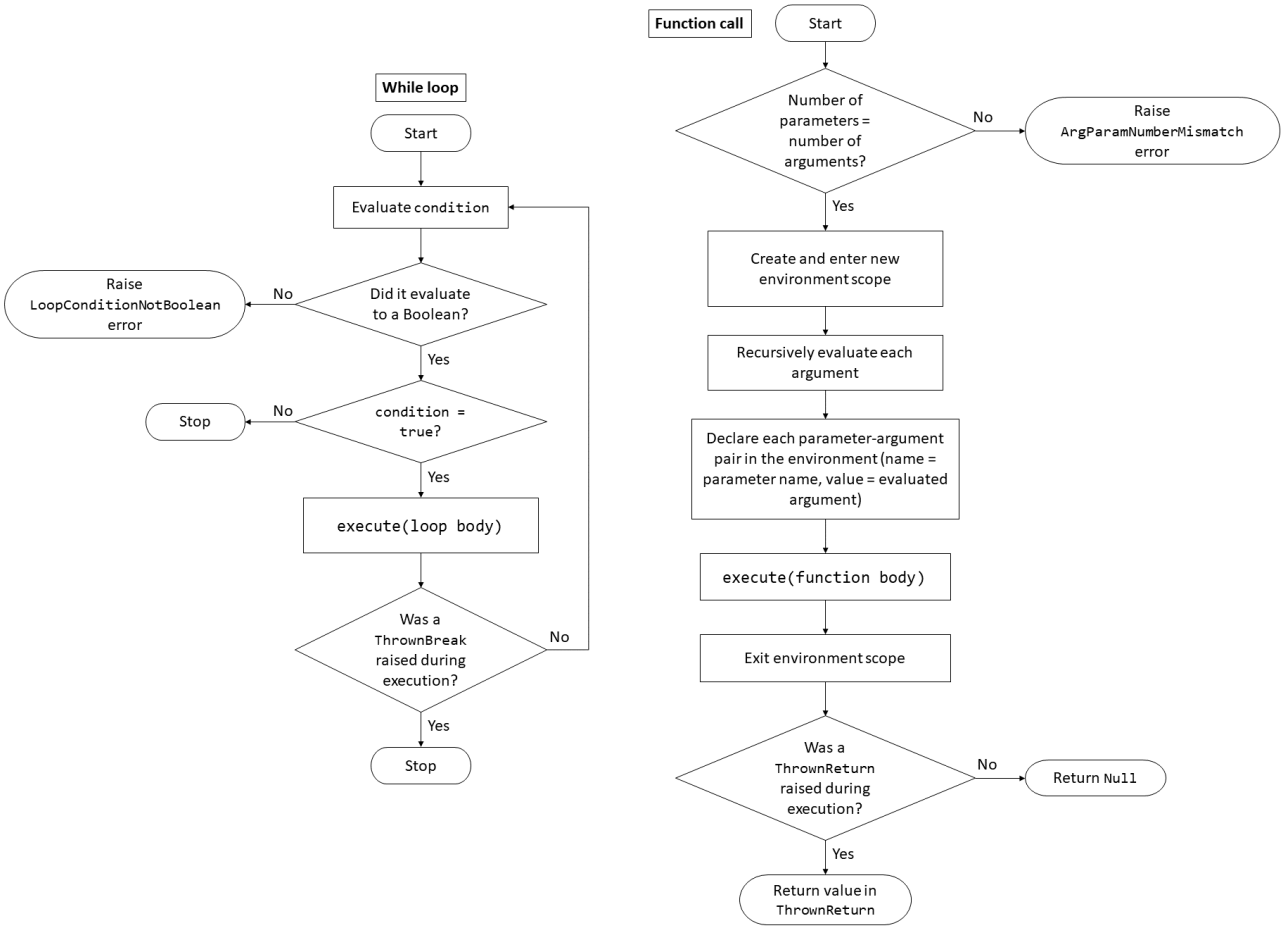

Figure 2.30: Flowchart showing the handling of breaks and returns in while loops and function calls

2.7.6 Built-in functions

Objective 8 requires the language to support built-in functions. The following lists each function and how each will be implemented:

- Name: `input`
  Objective: 8a
  Parameters: the prompt (string)[11]
  Returns: the user's input (string)
  Implementation: will simply use Rust's input functionality.

- Name: `append`
  Objective: 8b
  Parameters: the target array (expression), the value to append (expression)
  Returns: the changed array (`Array` variant of `Value` enum)
  Implementation: as per Listing 22.

Listing 22 `append` built-in function

```
function APPEND(target, value)                          ▷ Appends value to target
    pointer ← CONSTRUCT_POINTER(target)                 ▷ Pointer to target array
    value_eval ← EVALUATE(value)
    if target evaluates to a Value::Array(array) then
        Append value_eval to array
        Update self.environment with new array using pointer
        return Value::Array(array)
    else
        raise ExpectedType error with expected = "Array"   ▷ We can only append to arrays
    end if
end function
```

- Name: `remove`
  Objective: 8c
  Parameters: the target array/dictionary (expression), the index/key at which to remove (expression)
  Returns: the changed array/dictionary (`Array` or `Dictionary` variant of `Value` enum)
  Implementation: as per Listing 23.

Listing 23 `remove` built-in function

```
function REMOVE(target, key)                            ▷ Removes entry at key from target
    pointer ← CONSTRUCT_POINTER(target)                 ▷ Pointer to target array/dictionary
    key_eval ← EVALUATE(key)
    if target evaluates to a Value::Array(array) then
        if key_eval is a valid index and in bounds then
            Remove index key_eval from array
            Update self.environment with new array using pointer
            return Value::Array(array)
        else
            raise appropriate error
        end if
    else if target evaluates to Value::Dictionary(dict) then
        dict.REMOVE(key_eval)
        Update self.environment with new dictionary using pointer
        return Value::Dictionary(dict)
    else
        raise ExpectedType error with expected = "Array or Dictionary"   ▷ We can only remove from arrays and dictionaries
    end if
end function
```

[11] A specific `ArgParamNumberMismatch` error can be raised if the number of arguments given is not equal to the number of parameters; this applies to all built-in functions (*Objective 10a*).

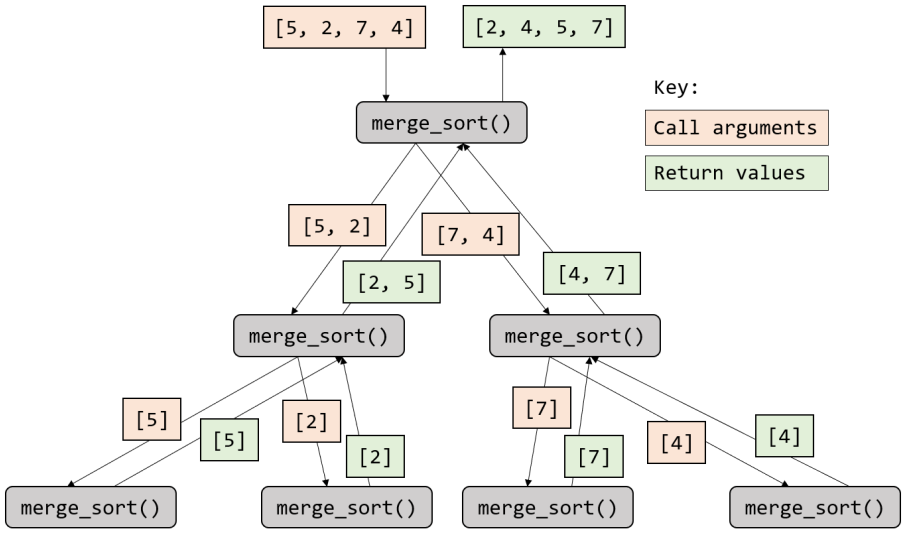

Figure 2.31: **Recursive** merge sort

- Name: `to_number`
  Objective: 8d
  Parameters: the string or Boolean value from which to convert (expression)
  Returns: the corresponding number (`Number` variant of `Value` enum)
  Implementation: will use Rust's casts.

- Name: `to_string`
  Objective: 8e
  Parameters: the number or Boolean value from which to convert (expression)
  Returns: the corresponding string (`String_` variant of `Value` enum)
  Implementation: will use Rust's casts.

- Name: `size`
  Objective: 8f
  Parameters: the target array/dictionary/string (expression)
  Returns: the size/length of the target (`Number` variant of `Value` enum)
  Implementation: will piggyback Rust's size functions for arrays and strings and will use `HashTable.size` method for dictionaries (see Section 2.5).

- Name: `sort`
  Objective: 8g
  Parameters: the array to sort; must consist entirely of numbers or entirely of strings (expression)
  Returns: the sorted array (`Array` variant of `Value` enum)
  Implementation: will use **merge sort**. Rust does not have a native sort function; moreover, we cannot compare `Value` variants by default, so we have to implement our own. Merge sort works **recursively**, first dividing the array into smaller arrays until they contain only one element, then *merging* the subarrays. To merge two *sorted* arrays in $O(n)$ time, we can use a **two pointers** approach, with a pointer for each array: we can compare the values at each pointer and append the 'lower' of the two to the array and advance the respective pointer. The whole algorithm is outlined in Listing 24 and Figure 2.31. Merge sort runs in $O(n \log n)$ time.

Listing 24 Implementing the `sort` built-in function using **merge sort**

```
function SORT(array of length n)                          ▷ Returns array in ascending order
    if array has one element then                                              ▷ Base case
        return array
    end if
    left ← SORT(array[0..n/2])                  ▷ Recursively sort left half of array (recursive case)
    right ← SORT(array[n/2..n])                          ▷ Recursively sort right half of array
    left_index ← 0
    right_index ← 0
    merged_array ← [ ]                                                  ▷ The result array
    while left_index < length of left and right_index < length of right do    ▷ Merge using two pointers
        if left[left_index] = Number(left_num) and right[right_index] = Number(right_num) then
            if left_num < right_num then
                Append left[left_index] to merged_array
                left_index ← left_index + 1
            else
                Append right[right_index] to merged_array
                right_index ← right_index + 1
            end if
        else if left[left_index] = String_(left_str) and right[right_index] = String_(right_str) then
            if left_str < right_str then
                Append left[left_index] to merged_array
                left_index ← left_index + 1
            else
                Append right[right_index] to merged_array
                right_index ← right_index + 1
            end if
        else
            raise BinaryTypeError with expected = "Number or String"    ▷ Can only compare numbers
and strings
        end if
    end while
    Append the remainder of left or right to merged_array
    return merged_array
end function
```

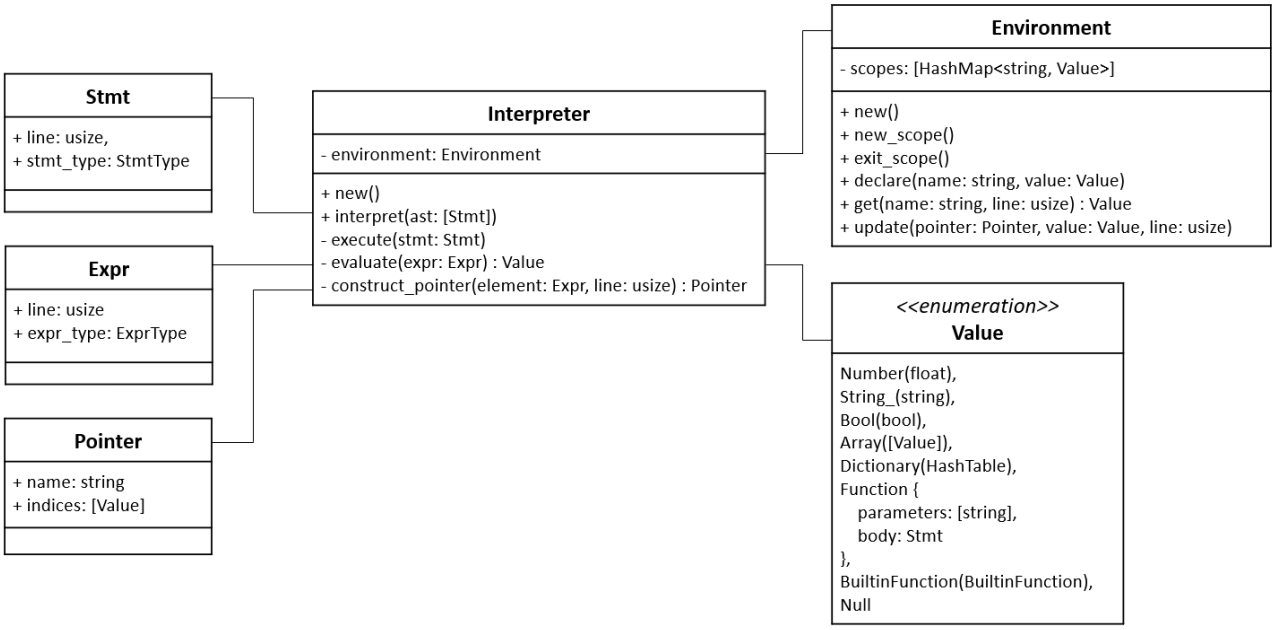

Figure 2.32: Class diagram of the `Interpreter` class and related types

2.7.7 The `Interpreter` class

Evaluation and execution will be wrapped within the `Interpreter` class. It will have one **private attribute**:

- `environment` (an `Environment` object)
  The environment for execution (see Section 2.6)

It will also have the following **methods**:

- `new()` (public)
  Constructs a new instance of the class by initialising a new `environment` object.
- `interpret(ast: [Stmt])` (public)
  This is the interface method and driver function for evaluation and execution. It loops through `ast` and calls `execute` for each statement. Calls `report_errors` on raised errors (see Subsection 2.1.2).
- `execute(stmt: Stmt)` (private)
  Executes the given statement as described in Subsection 2.7.4.
- `evaluate(expr: Expr) : Value` (private)
  Evaluates the given expression and returns the resulting `Value` enum, as described in Subsection 2.7.3.
- `construct_pointer(element: Expr, line: usize) : Pointer` (private)
  Constructs a pointer as per Subsection 2.7.2. The `line` argument will be used to report errors.

Figure 2.32 summarises this class in a class diagram.

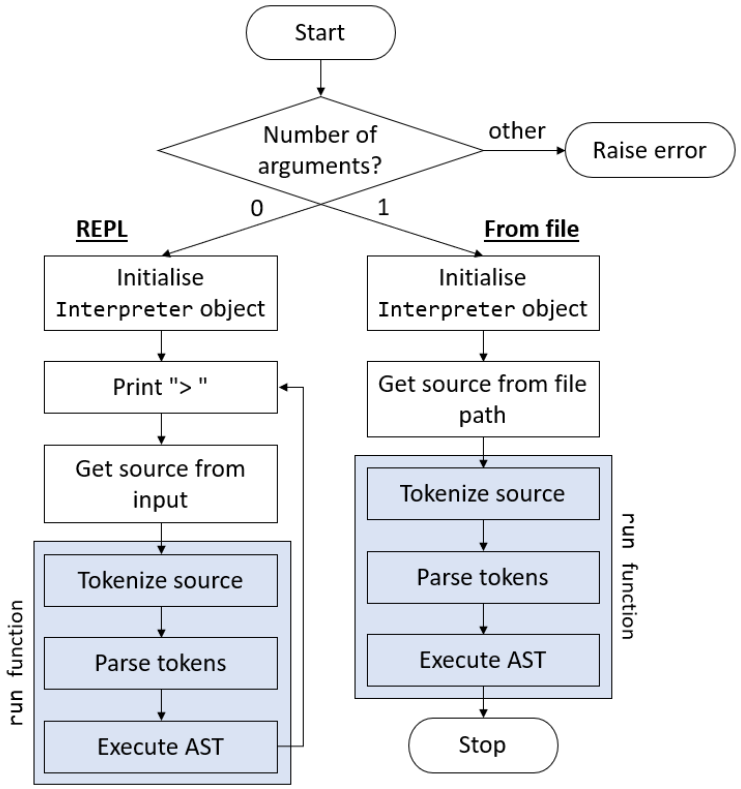

Figure 2.33: Flowchart showing operation of driver function

2.8 Driver code

Objectives 11 and 12 require two different ways of running source code. The driver code will make the decision of which to use based on the number of command-line arguments given to the interpreter binary:

1. If no arguments were given, run the REPL interface.
2. If one argument was given, run the source code in the file at the path in the given argument.

A common function `run` will take care of running the given source code string. Note that the `Interpreter` object has to be initialised beforehand and given to the `run` function as an argument. This is because we want to maintain the environment between each source code input in the REPL interface. The driver function is illustrated in 2.33.

Chapter 3

Technical solution

3.1 Organisation and techniques used

I have separated the implementation into different files to modularise the interpreter. The organisation of this chapter, the contents of each file, and noteworthy techniques used are described in Table 3.1.

3.2 Notes

3.2.1 Notes on Rust

- If the last expression in a function does not end with a semicolon, its evaluated value will be the function's return value. For example,

```
fn add(a: i32, b: i32) -> i32 {
        a + b  // The result of `a + b` will be returned.
}
```

- `Result<T, E>` is a native enum with two variants: `Ok(T)` (i.e., stores a single value of type `T`), and `Err(E)`. This is will be used as the return type for many functions where an error may occur and can be dealt with. The caller can then pattern match on the `Result` and deal with the error accordingly. In most, if not all, cases, the type for `Err(E)` will be the enum `ErrorType` as described in 2.1.1.

- The `?` operator will be used with the `Result` in many places as it allows errors to be *bubbled up* the call stack easily to be dealt with **centrally**. Essentially, the expression `let x = f()?;` is equivalent to

```
let x = match f() {
        Ok(value) => value,
        Err(error) => return Err(error)
};
```

  In other words, `f()?` evaluates to the value contained within the `Ok(T)` variant if the `Result` is indeed an `Ok(T)` variant. Otherwise, it immediately returns the `Err(E)`.

  Figure 3.1 illustrates how the `?` operator will be used in the context of parsing.

3.2.2 Notes on the implementation

In many places in the implementation, I have opted to use *references* in function parameters as opposed to *values* if the function only needs to read the value of the data. This is to avoid potentially copying large values which may cause performance issues.

Table 3.1: Contents of each file

File name	Contents	Section	Page	Techniques used
`main.rs`	• Driver code	3.3	80	• File reading (line 38)
`error.rs`	• `ErrorType` enum • `report_errors` function • `report_error` function	3.4	82	
`token.rs`	• `TokenType` enum • `Literal` enum • `Token` class	3.5	87	• Class (line 39)
`tokenizer.rs`	• `State` enum • `Tokenizer` class	3.6	88	• Class and encapsulation (line 41) • Simulation of a DFA (line 95)
`expr.rs`	• `Expr` class • `ExprType` enum	3.7	95	• Class (line 6) • Recursive definitions (lines 15, 18, 19, etc.)
`stmt.rs`	• `Stmt` class • `StmtType` enum	3.8	96	• Class (line 5) • Recursive definitions (lines 14, 23, 27, 28, 42)
`parser.rs`	• `Parser` class	3.9	97	• Class and encapsulation (line 8) • Direct and indirect recursion (lines 407, 554, etc.) for the **recursive descent algorithm** • Building of an **abstract syntax tree** (lines 98, 302, 442, 44, etc.)
`value.rs`	• `Value` enum • `BuiltinFunction` enum	3.10	112	• Recursion for printing values (lines 48, 60, and 62)
`hash_table.rs`	• `KeyValue` class • `HashTable` class • `hash` function	3.11	114	• Class and encapsulation (line 23) • Functional programming (line 139) • Algorithmic complexity in checking equality of two hash tables (lines 147-171) • Hashing (line 182-241) • Recursion for hashing (line 191)
`environment.rs`	• `Pointer` class • `Environment` class • `index_value_to_usize` function	3.12	119	• Class and encapsulation (lines 8 and 14)
`interpreter.rs`	• `Interpreter` class • `merge_sort` function	3.13	123	• Class and encapsulation (line 12) • Direct and indirect recursion for traversing the **abstract syntax tree** (lines 40, 80, 85, etc.) • Functional programming (line 156) • Recursion for the construction of a `Pointer` object from a tree of indices (line 644) • Recursive merge sort (line 665 onwards)

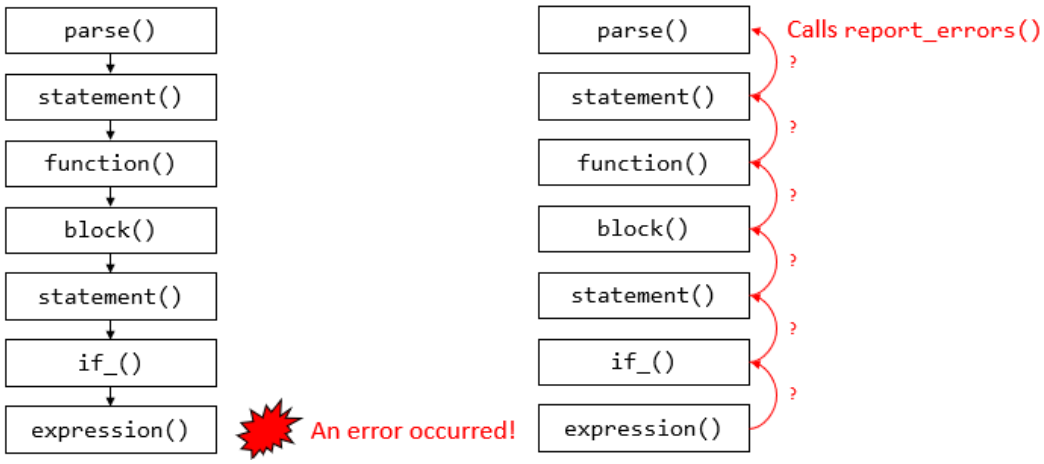

Figure 3.1: `?` operator in action

3.3 Driver code - `main.rs`

As described in Section 2.8, the code in **`main.rs`** coordinates the source code execution pipeline. Note the **file reading** on line 38.

```
1  mod environment;
2  mod error;
3  mod expr;
4  mod hash_table;
5  mod interpreter;
6  mod parser;
7  mod stmt;
8  mod token;
9  mod tokenizer;
10 mod value;
11
12 use std::{env, io, io::Write, fs};
13
14 use parser::Parser;
15 use tokenizer::Tokenizer;
16 use interpreter::Interpreter;
17
18 /// Driver code.
19 fn main() {
20     let args: Vec<String> = env::args().collect();
21
22     // Note that `args[0]` will be the name of the binary.
23     // So to check whether one argument has been passed, we check if `args.len() == 2`.
24     if args.len() > 2 {
25         // Only one given argument is expected.
26         eprintln!("Usage: nea.exe [script]");
27     } else if args.len() == 2 {
28         // `args[1]` will be the given argument, i.e., the file path of the source code.
29         run_file(&args[1]);
30     } else {
31         // No arguments were given. In this case, we run the REPL interface.
32         run_repl();
33     }
34 }
35
36 /// Runs the source code given at the file path.
37 fn run_file(file_path: &str) {
38     // Reading from the file path. If an error occurs, the `expect()` method will print
    ↪  "Failed to read file." and terminate execution.
39     let source = fs::read_to_string(file_path).expect("Failed to read file.");
40
41     // An Interpreter object has to be provided to `run()`, as explained below.
42     let mut interpreter = Interpreter::new();
43
44     run(&source, &mut interpreter);
45 }
46
47 /// Runs the interactive REPL interface in the console.
48 fn run_repl() {
49     // We need the same `Interpreter` instance across all REPL source code inputs to
    ↪  preserve the variables and functions stored in the environment.
50     let mut interpreter = Interpreter::new();
51     loop {
52         print!("> ");
53         io::stdout().flush().expect("Error: flush failed");  // to flush out "> "
54
```

```rust
        // Read user input into `line`.
        let mut line = String::new();
        io::stdin()
        .read_line(&mut line)
        .expect("Failed to read line");

        run(&line, &mut interpreter);
    }
}

/// Executes the source code string with the given interpreter instance.
fn run(source: &str, interpreter: &mut Interpreter) {
    // Lexical analysis.
    let mut tokenizer = Tokenizer::new(source);
    // If the source code was tokenized without errors, assign the token sequence to
    ↪ `tokens`.
    let Ok(tokens) = tokenizer.tokenize() else {
        // If an error occurred, stop trying to execute the current source code string.
        // If the user is using a REPL interface, this does not then end the session but
        ↪ simply prompts the user for a new source code input, as expected.
        return;
    };

    // Syntax analysis.
    let mut parser = Parser::new(tokens);
    // Similarly, if the token sequence was parsed without errors, assign the abstract
    ↪ syntax tree to `ast`.
    let Ok(ast) = parser.parse() else {
        // If an error occurred, stop trying to execute the current source code string.
        return;
    };

    // Evaluation and execution.
    interpreter.interpret(ast);
}
```

3.4 Error handling - `error.rs`

As described in Section 2.1, `error.rs` contains the `ErrorType` enum, and the `report_errors` and `report_error` functions. This **centralises** the error reporting, which is good practice.

```rust
use crate::value::Value;

/// Possible errors that may occur during execution. This type will be used when bubbling up errors.
#[derive(Clone, Debug, PartialEq)]
pub enum ErrorType {
    // Lexical analysis errors, i.e., tokenization errors.
    UnexpectedCharacter {
        character: char,
        line: usize,
    },
    UnterminatedString,

    // Syntax analysis errors, i.e., syntax errors.
    ExpectedCharacter {
        expected: char,
        line: usize,
    },
    ExpectedExpression {
        line: usize,
    },
    ExpectedFunctionName {
        line: usize,
    },
    ExpectedParameterName {
        line: usize,
    },
    ExpectedVariableName {
        line: usize,
    },
    ExpectedSemicolonAfterInit {
        line: usize,
    },
    ExpectedSemicolonAfterCondition {
        line: usize,
    },
    ExpectedParenAfterIncrement {
        line: usize,
    },
    ExpectedColonAfterKey {
        line: usize,
    },

    // Environment errors.
    NameError {
        name: String,
        line: usize,
    },
    NotIndexable {
        line: usize,
    },
    OutOfBoundsIndex {
        index: usize,
        line: usize,
    },
    InsertNonStringIntoString {
```

```
56          line: usize,
57      },
58
59      // Execution errors, i.e., runtime errors.
60      InvalidAssignmentTarget {
61          line: usize,
62      },
63      ExpectedType {
64          expected: String,
65          got: String,
66          line: usize,
67      },
68      NonNaturalIndex {
69          got: Value,
70          line: usize,
71      },
72      NonNumberIndex {
73          got: String,
74          line: usize,
75      },
76      BinaryTypeError {
77          expected: String,
78          got_left: String,
79          got_right: String,
80          line: usize,
81      },
82      DivideByZero {
83          line: usize,
84      },
85      IfConditionNotBoolean {
86          line: usize,
87      },
88      LoopConditionNotBoolean {
89          line: usize,
90      },
91      CannotCallName {
92          line: usize,
93      },
94      ArgParamNumberMismatch {
95          arg_number: usize,
96          param_number: usize,
97          line: usize,
98      },
99      CannotConvertToNumber {
100         line: usize,
101     },
102
103     // Hash table errors.
104     CannotHashFunction {
105         line: usize,
106     },
107     CannotHashDictionary {
108         line: usize,
109     },
110     KeyError {
111         key: Value,
112         line: usize,
113     },
114
115     // Special errors.
```

```
116     // These will be used to unwind the call stack when a break or return statement is
        ↪ used.
117     // If used correctly, these will be caught within the interpreter.
118     // If not, e.g., a return statement was used outside a function, the error will be
        ↪ reported.
119     ThrownBreak {
120         line: usize,
121     },
122     ThrownReturn {
123         value: Value,
124         line: usize,
125     },
126 }
127
128 /// Prints the error message for each error in `errors`.
129 pub fn report_errors(errors: &[ErrorType]) {
130     println!("An error has occurred.");
131     for error in errors {
132         print_report(error);
133     }
134 }
135
136 /// Prints the error message for an individual error.
137 fn print_report(error: &ErrorType) {
138     match error {
139         // Lexical analysis errors, i.e., tokenization errors.
140         ErrorType::UnexpectedCharacter { character, line } => {
141             println!("Line {}: unexpected character `{}`.", line, character);
142         },
143         ErrorType::UnterminatedString => {
144             println!("Unterminated string at end of file.");
145         },
146
147         // Syntax analysis errors, i.e., syntax errors.
148         ErrorType::ExpectedCharacter { expected, line } => {
149             println!("Line {}: expected character `{}`", line, expected);
150         },
151         ErrorType::ExpectedExpression { line } => {
152             println!("Line {}: expected expression.", line);
153         },
154         ErrorType::ExpectedFunctionName { line } => {
155             println!("Line {}: expected function name. Make sure it is not a keyword.",
                ↪ line);
156         },
157         ErrorType::ExpectedParameterName { line } => {
158             println!("Line {}: expected parameter name after a comma in function
                ↪ declaration.", line);
159         },
160         ErrorType::ExpectedVariableName { line } => {
161             println!("Line {}: expected variable name. Make sure it is not a keyword.",
                ↪ line);
162         },
163         ErrorType::ExpectedSemicolonAfterInit { line } => {
164             println!("Line {}: expected `;` after initialising statement in `for` loop.",
                ↪ line);
165         },
166         ErrorType::ExpectedSemicolonAfterCondition { line } => {
167             println!("Line {}: expected `;` after condition in `for` loop.", line);
168         },
169         ErrorType::ExpectedParenAfterIncrement { line } => {
```

```
170             println!("Line {}: expected `)` after increment statement in `for` loop.",
                    ↪ line);
171         },
172         ErrorType::ExpectedColonAfterKey { line } => {
173             println!("Line {}: expected colon after dictionary key.", line);
174         },
175 
176         // Environment errors.
177         ErrorType::NameError { ref name, line } => {
178             println!("Line {}: `{}` is not defined.", line, name);
179         },
180         ErrorType::NotIndexable { line } => {
181             println!("Line {}: the value is not indexable.", line);
182         },
183         ErrorType::OutOfBoundsIndex { index, line } => {
184             println!("Line {}: index `{}` is out of bounds.", line, index);
185         },
186         ErrorType::InsertNonStringIntoString { line } => {
187             println!("Line {}: attempted to insert a non-string into a string.", line);
188         },
189 
190         // Execution errors, i.e., runtime errors.
191         ErrorType::InvalidAssignmentTarget { line } => {
192             println!("Line {}: invalid assignment target. Make sure you are not assigning to
                    ↪ a literal.", line);
193         },
194         ErrorType::ExpectedType { ref expected, ref got, line } => {
195             println!("Line {}: expected type {}; instead got type {}.", line, expected,
                    ↪ got);
196         },
197         ErrorType::NonNaturalIndex { got, line } => {
198             println!("Line {}: index evaluated to {}, which is not a positive integer.",
                    ↪ line, got);
199         },
200         ErrorType::NonNumberIndex { got, line } => {
201             println!("Line {}: index evaluated to a {}, which is not a positive integer.",
                    ↪ line, got);
202         },
203         ErrorType::BinaryTypeError { ref expected, ref got_left, ref got_right, line } => {
204             println!("Line {}: this operation requires both sides' types to be {}. Instead,
                    ↪ got {} and {} respectively.", line, expected, got_left, got_right);
205         },
206         ErrorType::DivideByZero { line } => {
207             println!("Line {}: divisor is 0.", line);
208         },
209         ErrorType::IfConditionNotBoolean { line } => {
210             println!("Line {}: the `if` condition did not evaluate to a Boolean value.",
                    ↪ line);
211         },
212         ErrorType::LoopConditionNotBoolean { line } => {
213             println!("Line {}: the condition of the loop did not evaluate to a Boolean
                    ↪ value.", line);
214         },
215         ErrorType::CannotCallName { line } => {
216             println!("Line {}: cannot call name as a function.", line);
217         },
218         ErrorType::ArgParamNumberMismatch { arg_number, param_number, line } => {
219             println!("Line {}: attempted to call function with {} argument(s), but function
                    ↪ accepts {}.", line, arg_number, param_number);
220         },
221         ErrorType::CannotConvertToNumber { line } => {
```

```
222             println!("Line {}: could not convert to a number.", line);
223         },
224
225         // Hash table errors.
226         ErrorType::CannotHashFunction { line } => {
227             println!("Line {}: cannot hash function (functions cannot be used as keys in
                ↪  dictionary entries).", line);
228         },
229         ErrorType::CannotHashDictionary { line } => {
230             println!("Line {}: cannot hash dictionary (dictionaries cannot be used as keys
                ↪  in dictionary entries).", line);
231         },
232         ErrorType::KeyError { key, line } => {
233             println!("Line {}: key `{}` does not exist in the dictionary.", line, key);
234         },
235
236         // Special errors.
237         ErrorType::ThrownBreak { line } => {
238             println!("Line {}: `break` has to be used within a loop.", line);
239         },
240         ErrorType::ThrownReturn { value: _ , line} => {
241             println!("Line {}: `return` has to be used within a function.", line);
242         },
243     }
244 }
```

3.5 The Token class - token.rs

This file contains the definitions for the `TokenType` and `Literal` enums which are used in the `Token` class (known in Rust as a `struct`). It is good practice to separate the definitions of types which are used in many different modules of the program (the `Token` class will be used in tokenization and parsing).

```rust
/// Possible types of tokens.
#[derive(Clone, Debug, Eq, PartialEq)]
pub enum TokenType {
    // Single-character tokens.
    LeftParen, RightParen,
    LeftCurly, RightCurly,
    LeftSquare, RightSquare,
    Colon, Comma, Minus, Percent,
    Plus, Semicolon, Slash, Star,

    // One- or two-character tokens.
    Bang, BangEqual,
    Equal, EqualEqual,
    Greater, GreaterEqual,
    Less, LessEqual,

    // Literals.
    True, False, String_, Number,

    // Keywords.
    And, Break, Else,
    Func, For, If, Null, Or, Print,
    Return, Var, While,

    Identifier, Eof
}

/// Literal values declared in the source code.
#[derive(Clone, Debug, PartialEq)]
pub enum Literal {
    Number(f64),
    String_(String),
    Bool(bool),
    Null,
}

/// A token.
#[derive(Clone, Debug, PartialEq)]
pub struct Token {
    pub type_: TokenType,  // The type of the token.
    pub lexeme: String,  // The source code substring from which the token was constructed.
    pub literal: Literal,  // The literal value (number/string/Boolean) the token
    ↪ represents; if the token is not a literal, will be set to the `Null` variant.
    pub line: usize,  // The line number of the source code from which the token was
    ↪ constructed.
}
```

3.6 Lexical analysis - `tokenizer.rs`

This implements the tokenizer, as described in Section 2.2. Note that the `scan_token` function returns the type `Result<Option<Token>, ErrorType>`. `Option<T>` is another Rust enum with the variants `Some(T)` and `None`. Here, it is possible that no token was produced, i.e., the tokenizer landed on the `NoOp` state.

The `scan_token` function which simulates the DFA starts at line 95. Note that the use of `map_or` on line 287 is considered idiomatic in Rust as it simplifies pattern matching involving an `Option`.

```
1  use crate::token::{Token, TokenType, Literal};
2  use crate::error::{self, ErrorType};
3
4  /// The states of the DFA.
5  #[derive(Debug)]
6  enum State {
7      Start,
8      GotLeftParen,
9      GotRightParen,
10     GotLeftCurly,
11     GotRightCurly,
12     GotLeftSquare,
13     GotRightSquare,
14     GotColon,
15     GotComma,
16     GotMinus,
17     GotPercent,
18     GotPlus,
19     GotSemicolon,
20     GotSlash,
21     GotStar,
22     InComment,
23     GotBang,
24     GotBangEqual,
25     GotEqual,
26     GotEqualEqual,
27     GotGreater,
28     GotGreaterEqual,
29     GotLess,
30     GotLessEqual,
31     InStringDouble,  // Double quote strings.
32     InStringSingle,  // Single quote strings.
33     GotString,
34     InNumberBeforeDot,
35     InNumberAfterDot,
36     InWord,  // Identifiers and keywords.
37     NoOp,  // No operation.
38 }
39
40 /// Performs lexical analysis.
41 pub struct Tokenizer<'a> {
42     source: &'a str,  // The source code string.
43     tokens: Vec<Token>,  // The result sequence of tokens.
44     start: usize,  // An index pointing to the start of the current token. This will be used
       ↪  to set the value of lexemes and literals.
45     current_index: usize,  // An index pointing to the next character to be scanned.
46     current_line: usize,  // The current line number.
47 }
48
49 impl<'a> Tokenizer<'a> {
50     /// Constructs a `Tokenizer` instance with the given source code string.
51     pub fn new(source: &'a str) -> Self {
52         Self {
```

```
53              source,  // Equivalent to `souce: source`.
54              tokens: Vec::new(),
55              start: 0,
56              current_index: 0,
57              current_line: 1,
58          }
59      }
60
61      /// The interface method which creates and returns an array of tokens.
62      pub fn tokenize(&mut self) -> Result<Vec<Token>, ErrorType> {
63          while self.source.chars().nth(self.current_index).is_some() {
64              // If `current_index` has not reached the end of the source code, scan the next
                ↪  token.
65              match self.scan_token() {
66                  Ok(token_opt) => {
67                      // If no error occurred...
68                      if let Some(token) = token_opt {
69                          // and `scan_token()` returned a token, append it to the sequence of
                            ↪  tokens.
70                          // It is possible that `scan_token()` returns `Ok(None)` if the DFA
                            ↪  lands on the `NoOp` state.
71                          self.tokens.push(token);
72                      }
73                  },
74                  Err(error) => {
75                      // If an error has occurred during the `scan_token()` call, report the
                        ↪  error.
76                      error::report_errors(&[error.clone()]);
77                      // Return an `Err` variant so that the driver code knows to end
                        ↪  execution.
78                      return Err(error);
79                  }
80              }
81          }
82
83          // Push an EOF token to end the token sequence.
84          self.tokens.push(Token {
85              type_: TokenType::Eof,
86              lexeme: String::from(""),
87              literal: Literal::Null,
88              line: self.current_line
89          });
90
91          Ok(self.tokens.clone())
92      }
93
94      /// Scans the token starting from `current_index` by simulating the DFA.
95      fn scan_token(&mut self) -> Result<Option<Token>, ErrorType> {
96          let mut current_state = State::Start;  // The current state of the finite
            ↪  automaton.
97
98          loop {
99              // It is possible that the tokenizer reaches the end of the source code before
                ↪  `scan_token()` returns.
100             // So, we account for `current_char_opt` being None in all possible current
                ↪  states.
101             let current_char_opt = self.source.chars().nth(self.current_index);
102
103             match current_state {
104                 State::Start => {
105                     self.start = self.current_index;  // The next token starts here.
```

```
106                 if let Some(current_char) = current_char_opt {
107                     match current_char {
108                         '(' => current_state = State::GotLeftParen,
109                         ')' => current_state = State::GotRightParen,
110                         '{' => current_state = State::GotLeftCurly,
111                             '}' => current_state = State::GotRightCurly,
112                         '[' => current_state = State::GotLeftSquare,
113                         ']' => current_state = State::GotRightSquare,
114                         ':' => current_state = State::GotColon,
115                         ',' => current_state = State::GotComma,
116                         '-' => current_state = State::GotMinus,
117                         '%' => current_state = State::GotPercent,
118                         '+' => current_state = State::GotPlus,
119                         ';' => current_state = State::GotSemicolon,
120                         '/' => current_state = State::GotSlash,
121                         '*' => current_state = State::GotStar,
122
123                         // For these tokens, we have to see the next character to be
                            ↪  able to correctly identify the token.
124                         '!' => current_state = State::GotBang,
125                         '=' => current_state = State::GotEqual,
126                         '>' => current_state = State::GotGreater,
127                         '<' => current_state = State::GotLess,
128
129                         // Literals.
130                         '"' => current_state = State::InStringDouble,
131                         '\'' => current_state = State::InStringSingle,
132
133                         '0'..='9' => current_state = State::InNumberBeforeDot,
134
135                         // Identifiers and keywords.
136                         'a'..='z' | 'A'..='Z' | '_' => current_state = State::InWord,
137
138                         // Comments.
139                         '#' => current_state = State::InComment,
140
141                         // Whitespace.
142                         ' ' | '\r' | '\t' => current_state = State::NoOp,
143
144                         '\n' => {
145                             self.current_line += 1;
146                             current_state = State::NoOp;
147                         },
148
149                         other => {
150                             // If the character does not match any of the above rules,
                                ↪  raise an `UnexpectedCharacter` error.
151                             return Err(ErrorType::UnexpectedCharacter {
152                                 character: other,
153                                 line: self.current_line,
154                             });
155                         },
156                     }
157                 } else {
158                     // Note that this should be unreachable: the `Start` state is only
                        ↪  ever accessed at the first iteration of the loop,
159                     // and we have already checked we are not at the end in
                        ↪  `tokenize()`.
160                     return Ok(None);
161                 }
162             },
```

```
163
164                 State::GotLeftParen => return
                    ↪  Ok(Some(self.construct_token(TokenType::LeftParen))),
165                 State::GotRightParen => return
                    ↪  Ok(Some(self.construct_token(TokenType::RightParen))),
166                 State::GotLeftCurly => return
                    ↪  Ok(Some(self.construct_token(TokenType::LeftCurly))),
167                 State::GotRightCurly => return
                    ↪  Ok(Some(self.construct_token(TokenType::RightCurly))),
168                 State::GotLeftSquare => return
                    ↪  Ok(Some(self.construct_token(TokenType::LeftSquare))),
169                 State::GotRightSquare => return
                    ↪  Ok(Some(self.construct_token(TokenType::RightSquare))),
170                 State::GotColon => return Ok(Some(self.construct_token(TokenType::Colon))),
171                 State::GotComma => return Ok(Some(self.construct_token(TokenType::Comma))),
172                 State::GotMinus => return Ok(Some(self.construct_token(TokenType::Minus))),
173                 State::GotPercent => return
                    ↪  Ok(Some(self.construct_token(TokenType::Percent))),
174                 State::GotPlus => return Ok(Some(self.construct_token(TokenType::Plus))),
175                 State::GotSemicolon => return
                    ↪  Ok(Some(self.construct_token(TokenType::Semicolon))),
176                 State::GotSlash => return Ok(Some(self.construct_token(TokenType::Slash))),
177                 State::GotStar => return Ok(Some(self.construct_token(TokenType::Star))),
178
179                 State::GotBang => {
180                     if current_char_opt == Some('=') {
181                         current_state = State::GotBangEqual;
182                     } else {
183                         // If the character isn't `=` or we are at the end, just make a
                            ↪  `Bang` token.
184                         return Ok(Some(self.construct_token(TokenType::Bang)));
185                     }
186                 },
187                 State::GotEqual => {
188                     if current_char_opt == Some('=') {
189                         current_state = State::GotEqualEqual;
190                     } else {
191                         // If the character isn't `=` or we are at the end, just make an
                            ↪  `Equal` token.
192                         return Ok(Some(self.construct_token(TokenType::Equal)));
193                     }
194                 },
195                 State::GotGreater => {
196                     if current_char_opt == Some('=') {
197                         current_state = State::GotGreaterEqual;
198                     } else {
199                         // If the character isn't `=` or we are at the end, just make a
                            ↪  `Greater` token.
200                         return Ok(Some(self.construct_token(TokenType::Greater)));
201                     }
202                 },
203                 State::GotLess => {
204                     if current_char_opt == Some('=') {
205                         current_state = State::GotLessEqual;
206                     } else {
207                         // If the character isn't `=` or we are at the end, just make a
                            ↪  `Less` token.
208                         return Ok(Some(self.construct_token(TokenType::Less)));
209                     }
210                 },
211
```

```
212                 State::GotBangEqual => return
                    ↪  Ok(Some(self.construct_token(TokenType::BangEqual))),
213                 State::GotEqualEqual => return
                    ↪  Ok(Some(self.construct_token(TokenType::EqualEqual))),
214                 State::GotGreaterEqual => return
                    ↪  Ok(Some(self.construct_token(TokenType::GreaterEqual))),
215                 State::GotLessEqual => return
                    ↪  Ok(Some(self.construct_token(TokenType::LessEqual))),
216
217                 State::InStringDouble => {
218                     if current_char_opt == Some('"') {
219                         current_state = State::GotString;
220                     } else if current_char_opt.is_none() {
221                         // We have reached the end and there was no closing `"`.
222                         return Err(ErrorType::UnterminatedString);
223                     }
224                 },
225                 State::InStringSingle => {
226                     if current_char_opt == Some('\'') {
227                         current_state = State::GotString;
228                     } else if current_char_opt.is_none() {
229                         // We have reached the end and there was no closing `'`.
230                         return Err(ErrorType::UnterminatedString);
231                     }
232                 },
233                 State::GotString => {
234                     return Ok(Some(self.construct_token_with_literal(
235                     TokenType::String_,
236                     Literal::String_(self.source[self.start+1..self.current_index-1]⌋
                        ↪  .to_owned())
237                     )));
238                 },
239
240                 State::InNumberBeforeDot => {
241                     match current_char_opt {
242                         Some(current_char) => {
243                             if current_char == '.' {
244                                 current_state = State::InNumberAfterDot;
245                             } else if !current_char.is_ascii_digit() {
246                                 // If it is not '0'-'9' (or a '.'), we have reached the end
                                    ↪  of the number.
247                                 return Ok(Some(self.construct_token_with_literal(
248                                 TokenType::Number,
249                                 Literal::Number(self.source[self.start..self.current_index]⌋
                                    ↪  .parse().unwrap())
250                                 )));
251                             }
252                             // If it is a digit, we stay in this state and keep consuming
                                ↪  digits.
253                         },
254                         None => {
255                             // If we have reached the end of the source code, then we can
                                ↪  return with the number we constructed so far.
256                             return Ok(Some(self.construct_token_with_literal(
257                             TokenType::Number,
258                             Literal::Number(self.source[self.start..self.current_index]⌋
                                ↪  .parse().unwrap())
259                             )));
260                         }
261                     }
262                 },
```

```
263                 State::InNumberAfterDot => {
264                     // Similar to above, but do not allow for '.' as we already have one in
                        ↪ the number.
265                     match current_char_opt {
266                         Some(current_char) => {
267                             if !current_char.is_ascii_digit() {
268                                 // We have reached the end of the number.
269                                 return Ok(Some(self.construct_token_with_literal(
270                                 TokenType::Number,
271                                 Literal::Number(self.source[self.start..self.current_index]
                                    ↪ .parse().unwrap())
272                                 )));
273                             }
274                             // If it is a digit, we stay in this state and keep consuming
                                ↪ digits.
275                         },
276                         None => {
277                             // Again, if we have reached the end of the source code, then we
                                ↪ can return with the number we constructed so far.
278                             return Ok(Some(self.construct_token_with_literal(
279                             TokenType::Number,
280                             Literal::Number(self.source[self.start..self.current_index]
                                ↪ .parse().unwrap())
281                             )));
282                         }
283                     }
284                 },
285
286                 State::InWord => {
287                     if current_char_opt.map_or(true, |current_char|
                        ↪ !(current_char.is_ascii_alphanumeric() || current_char == '_')) {
288                         // Construct the token now if:
289                         // we are at the end of the source code, or
290                         // if the current character is not alphanumeric or an `_` (i.e., we
                            ↪ have now scanned through the complete word).
291                         let lexeme = &self.source[self.start..self.current_index];
292                         return Ok(Some(match lexeme {
293                             "and" => self.construct_token(TokenType::And),
294                             "break" => self.construct_token(TokenType::Break),
295                             "else" => self.construct_token(TokenType::Else),
296                             "false" => self.construct_token_with_literal(TokenType::False,
                                ↪ Literal::Bool(false)),
297                             "func" => self.construct_token(TokenType::Func),
298                             "for" => self.construct_token(TokenType::For),
299                             "if" => self.construct_token(TokenType::If),
300                             "null" => self.construct_token_with_literal(TokenType::Null,
                                ↪ Literal::Null),
301                             "or" => self.construct_token(TokenType::Or),
302                             "print" => self.construct_token(TokenType::Print),
303                             "return" => self.construct_token(TokenType::Return),
304                             "true" => self.construct_token_with_literal(TokenType::True,
                                ↪ Literal::Bool(true)),
305                             "var" => self.construct_token(TokenType::Var),
306                             "while" => self.construct_token(TokenType::While),
307                             _ => self.construct_token(TokenType::Identifier)
308                         }));
309                     }
310                 },
311
312                 State::InComment => {
```

```
313                     // If we have a new line or we have reached the end of the file, the
                        ↪  comment has ended.
314                     if current_char_opt == Some('\n') {
315                         self.current_line += 1;
316                         current_state = State::NoOp;
317                     } else if current_char_opt.is_none() {
318                         current_state = State::NoOp;
319                     }
320                 },
321 
322                 State::NoOp => return Ok(None),
323             }
324 
325             // Increment the pointer to the next character.
326             self.current_index += 1;
327         }
328     }
329 
330     /// A helper function which returns a fully formed `Token` object.
331     fn construct_token_with_literal(&mut self, token_type: TokenType, literal: Literal) ->
        ↪  Token {
332         Token {
333             type_: token_type,
334             lexeme: String::from(&self.source[self.start..self.current_index]),
335             literal,
336             line: self.current_line,
337         }
338     }
339 
340     /// A helper function which returns the `Token` object for a non-literal token.
341     fn construct_token(&mut self, token_type: TokenType) -> Token {
342         self.construct_token_with_literal(token_type, Literal::Null)
343     }
344 }
```

3.7 The Expr class - expr.rs

Implements the class described in Section 2.3.2. Note the **recursive** definitions create a **tree** structure for the abstract syntax tree (lines 15, 18, 19, etc.).

```
1  use crate::token;
2  use crate::hash_table::KeyValue;
3
4  /// An expression.
5  #[derive(Clone, Debug, PartialEq)]
6  pub struct Expr {
7      pub line: usize,  // The line of the source code from which the expression was derived.
8      pub expr_type: ExprType,  // The type of expression.
9  }
10
11 /// Possible types of expressions.
12 #[derive(Clone, Debug, PartialEq)]
13 pub enum ExprType {
14     Array {
15         elements: Vec<Expr>,
16     },
17     Assignment {
18         target: Box<Expr>,
19         value: Box<Expr>,
20     },
21     Binary {
22         left: Box<Expr>,
23         operator: token::Token,
24         right: Box<Expr>,
25     },
26     Call {
27         callee: Box<Expr>,
28         arguments: Vec<Expr>,
29     },
30     Dictionary {
31         elements: Vec<KeyValue<Expr>>,
32     },
33     Element {
34         array: Box<Expr>,
35         index: Box<Expr>,
36     },
37     Grouping {
38         expression: Box<Expr>,
39     },
40     Literal {
41         value: token::Literal,
42     },
43     Unary {
44         operator: token::Token,
45         right: Box<Expr>,
46     },
47     Variable {
48         name: String,
49     },
50 }
```

3.8 The `Stmt` class - `stmt.rs`

Implements the class described in Section 2.3.3. Note the **recursive** definitions create a **tree** structure for the abstract syntax tree (lines 14, 23, 27, 28, 42).

```
1  use crate::expr::Expr;
2
3  /// A statement.
4  #[derive(Clone, Debug, PartialEq)]
5  pub struct Stmt {
6      pub line: usize,  // The line of the source code from which the statement was derived.
7      pub stmt_type: StmtType,  // The type of the statement.
8  }
9
10 /// Possible types of statements.
11 #[derive(Clone, Debug, PartialEq)]
12 pub enum StmtType {
13     Block {
14         body: Vec<Stmt>,
15     },
16     Break,
17     Expression {
18         expression: Expr,
19     },
20     Function {
21         name: String,
22         parameters: Vec<String>,
23         body: Box<Stmt>,
24     },
25     If {
26         condition: Expr,
27         then_body: Box<Stmt>,
28         else_body: Option<Box<Stmt>>,
29     },
30     Print {
31         expression: Expr,
32     },
33     Return {
34         expression: Expr,
35     },
36     VarDecl {
37         name: String,
38         value: Expr,
39     },
40     While {
41         condition: Expr,
42         body: Box<Stmt>,
43     },
44 }
```

3.9 Syntax analysis - parser.rs

Contains the `Parser` class as described in Section 2.3.6. This houses the **recursive descent parsing** algorithm from lines 73 to 753. Lines 390 to 753 implements recursive descent for expressions as described in Section 2.3.2, and lines 73 to 386 implements recursive descent for statements as described in Section 2.3.3. Note in particular the **direct recursion** on lines 407 and 554, and the **indirect recursion**, for example, on lines 113, 135, 577, 607, etc. Note that, although it is not very clear, an **abstract syntax tree** is being built in the recursion, as statements/expressions are being used as parts of other statements/expressions, e.g., on lines 98, 302, 442, 444, etc.

Errors are bubbled up as illustrated in Figure 3.1. The function starting on line 54 implements the synchronisation process as described in Section 2.3.4.

```
1  use crate::error::{ErrorType, self};
2  use crate::expr::{Expr, ExprType};
3  use crate::hash_table::KeyValue;
4  use crate::stmt::{Stmt, StmtType};
5  use crate::token::{Token, TokenType, Literal};
6
7  /// Performs syntax analysis.
8  pub struct Parser {
9      tokens: Vec<Token>,  // The input sequence of tokens.
10     current_index: usize,  // An index pointing to the current token.
11     current_line: usize,  // The current line number.
12 }
13
14 impl Parser {
15     /// Constructs a new `Parser` object given the sequence of tokens.
16     pub fn new(tokens: Vec<Token>) -> Self {
17         Self {
18             tokens,
19             current_index: 0,
20             current_line: 1,
21         }
22     }
23
24     /// The interface method which returns the abstract syntax tree of the source code as a
       ↪  sequence of statements.
25     pub fn parse(&mut self) -> Result<Vec<Stmt>, Vec<ErrorType>> {
26         let mut statements: Vec<Stmt> = Vec::new();  // The abstract syntax tree.
27
28         // We aim to collect as many errors as possible in one run into a vector and report
           ↪  them all at once.
29         let mut errors: Vec<ErrorType> = Vec::new();
30
31         while !self.check_next(&[TokenType::Eof]) {
32             // While we have not reached the end of the sequence of tokens (EOF), parse the
               ↪  next statement.
33             match self.statement() {
34                 Ok(statement) => statements.push(statement),
35                 Err(error) => {
36                     // If an error occurred during the parse, collect the error,
                       ↪  synchronise, and continue.
37                     errors.push(error);
38                     self.sync();
39                 },
40             }
41         }
42
43         if errors.is_empty() {
44             // If no error occurred, return the sequence of statements.
45             Ok(statements)
```

```
46          } else {
47              // If errors occurred, report all the errors and return an `Err` variant so that
                ↪ the driver code terminates execution.
48              error::report_errors(&errors[..]);
49              Err(errors)
50          }
51      }
52
53      /// Synchronises the parser to the next possible start of a new statement.
54      fn sync(&mut self) {
55          while !self.check_next(&[
56          // These are considered tokens that are 'safe' to synchronise to.
57          TokenType::Eof,
58          TokenType::For,
59          TokenType::Func,
60          TokenType::If,
61          TokenType::Print,
62          TokenType::Return,
63          TokenType::Var,
64          TokenType::While,
65          ]) {
66              self.current_index += 1;  // Increment `current_index` until a 'safe' token is
                    ↪ found.
67              self.current_line = self.tokens[self.current_index].line;  // Update the line
                    ↪ number as we iterate.
68          }
69      }
70
71      /// Parses a statement.
72      /// <statement> ::= Break | For <for> | Func <function> | If <if> | Print <print> |
        ↪ Return <return> | Var <var> | While <while> | <expression>
73      fn statement(&mut self) -> Result<Stmt, ErrorType> {
74          // If the next token is one of these, consume it and call the relevant function,
                ↪ which will parse the rest of the statement.
75          if self.check_and_consume(&[TokenType::Break]).is_some() {
76              Ok(Stmt {
77                  line: self.current_line,
78                  stmt_type: StmtType::Break
79              })
80          } else if self.check_and_consume(&[TokenType::For]).is_some() {
81              self.for_()
82          } else if self.check_and_consume(&[TokenType::Func]).is_some() {
83              self.function()
84          } else if self.check_and_consume(&[TokenType::If]).is_some() {
85              self.if_()
86          } else if self.check_and_consume(&[TokenType::Print]).is_some() {
87              self.print()
88          } else if self.check_and_consume(&[TokenType::Return]).is_some() {
89              self.return_()
90          } else if self.check_and_consume(&[TokenType::Var]).is_some() {
91              self.var()
92          } else if self.check_and_consume(&[TokenType::While]).is_some() {
93              self.while_()
94          } else {
95              Ok(Stmt {
96                  line: self.current_line,
97                  stmt_type: StmtType::Expression {
98                      expression: self.expression()?
99                  }
100             })
101         }
```

```
102     }
103
104     /// <block> ::= LeftCurly <statement>* RightCurly
105     fn block(&mut self) -> Result<Stmt, ErrorType> {
106         // Consume LeftCurly if it follows; otherwise, raise an error.
107         self.expect(TokenType::LeftCurly, '{')?;
108
109             // Parse <statement>*.
110             let mut statements: Vec<Stmt> = Vec::new();
111             while !self.check_next(&[TokenType::RightCurly, TokenType::Eof]) {
112                 // Keep parsing statements until the next token is a RightCurly or we have
                    ↪ reached the end of the sequence of tokens.
113                 statements.push(self.statement()?);
114             }
115
116             // Consume RightCurly.
117             self.expect(TokenType::RightCurly, '}')?;
118         Ok(Stmt {
119             line: self.current_line,
120             stmt_type: StmtType::Block {
121                 body: statements
122             }
123         })
124     }
125
126     /// <for> ::= LeftParen <statement>? Semicolon <expression>? Semicolon <statement>?
        ↪ RightParen <block>
127     fn for_(&mut self) -> Result<Stmt, ErrorType> {
128         // Consume LeftParen.
129         self.expect(TokenType::LeftParen, '(')?;
130
131         // Parse <statement>? as the initialising statement of the `for` loop. As it is
            ↪ optional, an Option<Stmt> is used.
132         let mut initialiser: Option<Stmt> = None;
133         if !self.check_next(&[TokenType::Semicolon]) {
134             // If the next token is not a semicolon, we parse it as the <statement>.
135             initialiser = Some(self.statement()?);
136         }
137
138         // Consume Semicolon if it follows.
139         if self.check_and_consume(&[TokenType::Semicolon]).is_none() {
140             // If there is no Semicolon, raise a specific error to avoid confusion as there
                ↪ are many semicolons in a `for` loop.
141             return Err(ErrorType::ExpectedSemicolonAfterInit { line: self.current_line });
142         }
143
144         // Parse <expression>? as the condition of the `for` loop. Again, an Option<Expr> is
            ↪ used as it is optional.
145         let mut condition = Expr {
146             line: self.current_line,
147             expr_type: ExprType::Literal {
148                 value: Literal::Bool(true)  // If no condition is given, it will be `true`
                        ↪ by default.
149             }
150         };
151         if !self.check_next(&[TokenType::Semicolon]) {
152             // If the next token is not a Semicolon, we parse it as the <expression>.
153             condition = self.expression()?;
154         }
155
156         // Consume Semicolon if it follows.
```

```
157         if self.check_and_consume(&[TokenType::Semicolon]).is_none() {
158             // If there is no Semicolon, again raise a specific error.
159             return Err(ErrorType::ExpectedSemicolonAfterCondition { line: self.current_line
                ↪  });
160         }
161 
162         // Parse <statement>? as the incrementing statement of the `for` loop.
163         let mut increment: Option<Stmt> = None;
164         if !self.check_next(&[TokenType::RightParen]) {
165             increment = Some(self.statement()?);
166         }
167 
168         // Consume RightParen if it follows; otherwise, raise a specific error.
169         if self.check_and_consume(&[TokenType::RightParen]).is_none() {
170             return Err(ErrorType::ExpectedParenAfterIncrement { line: self.current_line });
171         }
172 
173         // Parse <block>, i.e., the body of the `for` loop including the curly brackets.
174         let for_body = self.block()?;
175 
176         // Now, we internally convert the `for` loop into a `while` loop:
177         //  {
178             //      `initialiser`
179             //      while (`condition`) {
180                 //          {
181                     //              `for_body`
182                     //          }
183                 //          `increment`
184                 //      }
185             //  }
186 
187         // The body of the `while` loop includes the `for` loop body.
188         let mut while_body_vec = vec![for_body];
189         if let Some(inc) = increment {
190             // If there is an increment, put it at the end of the `while` loop body.
191             while_body_vec.push(inc);
192         }
193 
194         // Create the `Block` statement for the `while` loop.
195         let while_body = Stmt {
196             line: self.current_line,
197             stmt_type: StmtType::Block {
198                 body: while_body_vec
199             }
200         };
201 
202         // The `while` loop has the same condition as the `for` loop.
203         let while_loop = Stmt {
204             line: self.current_line,
205             stmt_type: StmtType::While {
206                 condition,
207                 body: Box::new(while_body)
208             }
209         };
210 
211         if let Some(init) = initialiser {
212             // If an initialising statement is given, place it before the `while` loop and
                ↪  wrap in a `Block` statement.
213             Ok(Stmt {
214                 line: self.current_line,
215                 stmt_type: StmtType::Block { body: vec![init, while_loop] }
```

```
216             })
217         } else {
218             // Otherwise, just return the `while` loop.
219             Ok(while_loop)
220         }
221     }
222 
223     /// <function> ::= Identifier LeftParen (Identifier (Comma Identifier)*)? RightParen
    ↪  <block>
224     fn function(&mut self) -> Result<Stmt, ErrorType> {
225         if let Some(function_name_token) = self.check_and_consume(&[TokenType::Identifier])
        ↪  {
226             // If an Identifier was given (the name of the function), consume it.
227 
228             // Consume LeftParen.
229             self.expect(TokenType::LeftParen, '(')?;
230 
231             // Parse (Identifier (Comma Identifier)*)?, i.e., collect an array of strings
            ↪  for the parameters.
232             let mut parameters: Vec<String> = Vec::new();
233             if !self.check_next(&[TokenType::RightParen]) {
234                 // If there are parameters, i.e., not just ().
235                 loop {  // Keep looping until there is no Comma following a parameter.
236                     if let Some(parameter) =
                        ↪  self.check_and_consume(&[TokenType::Identifier]) {
237                         // If an Identifier was given (the name of the parameter), consume
                        ↪  it and push it to the array of parameters.
238                         parameters.push(parameter.lexeme);
239                     } else {
240                         // Otherwise, raise a specific error, as a parameter must be given
                        ↪  after a comma.
241                         return Err(ErrorType::ExpectedParameterName { line:
                        ↪  self.current_line });
242                     }
243 
244                     // If a Comma does not follow a parameter, then there should be no more
                    ↪  parameters.
245                     // Otherwise, if a Comma was found, consume it, and parse the next
                    ↪  parameter.
246                     if self.check_and_consume(&[TokenType::Comma]).is_none() {
247                         break;
248                     }
249                 }
250             }
251 
252             // Consume RightParen.
253             self.expect(TokenType::RightParen, ')')?;
254 
255             // Parse <block>, the body of the function.
256             let body = self.block()?;
257 
258             Ok(Stmt {
259                 line: self.current_line,
260                 stmt_type: StmtType::Function {
261                     name: function_name_token.lexeme,
262                     parameters,
263                     body: Box::new(body),
264                 }
265             })
266         } else {
267             // If an Identifier was not given, raise a specific error.
```

```
268             Err(ErrorType::ExpectedFunctionName { line: self.current_line })
269         }
270     }
271
272     /// <if> ::= LeftParen <expression> RightParen <block> (Else <else>)?
273     fn if_(&mut self) -> Result<Stmt, ErrorType> {
274         // Consume LeftParen.
275         self.expect(TokenType::LeftParen, '(')?;
276
277         // Parse <expression>, the condition of the `if` statement.
278         let condition = self.expression()?;
279
280         // Consume RightParen.
281         self.expect(TokenType::RightParen, ')')?;
282
283         // Parse <block>, the `then` body of the `if` statement.
284         let then_body = self.block()?;
285
286         if self.check_and_consume(&[TokenType::Else]).is_some() {
287             // If there is an Else token after the `then` body, consume the Else, then parse
                ↪ <else>.
288             let else_body = self.else_()?;
289             Ok(Stmt {
290                 line: self.current_line,
291                 stmt_type: StmtType::If {
292                     condition,
293                     then_body: Box::new(then_body),
294                     else_body: Some(Box::new(else_body)),
295                 }
296             })
297         } else {
298             // Otherwise, just return the `if` statement with just the `then` body.
299             Ok(Stmt {
300                 line: self.current_line,
301                 stmt_type: StmtType::If {
302                     condition,
303                     then_body: Box::new(then_body),
304                     else_body: None,
305                 }
306             })
307         }
308     }
309
310     /// <else> ::= If <if> | <block>
311     fn else_(&mut self) -> Result<Stmt, ErrorType> {
312         // After an `else`, there can either be another block, which ends the `if` statement
            ↪ and creates the `else` body, or an `if` to make an `else if`.
313         if self.check_and_consume(&[TokenType::If]).is_some() {
314             // If there is an If token, consume it, then parse <if> to create an `else if`.
315             Ok(self.if_()?)
316         } else {
317             // Otherwise, just parse the `else` block.
318             Ok(self.block()?)
319         }
320     }
321
322     /// <print> ::= <expression>
323     fn print(&mut self) -> Result<Stmt, ErrorType> {
324         Ok(Stmt {
325             line: self.current_line,
326             stmt_type: StmtType::Print {
```

```
327                 expression: self.expression()?
328             }
329         })
330     }
331
332     /// <return> ::= <expression>
333     fn return_(&mut self) -> Result<Stmt, ErrorType> {
334         Ok(Stmt {
335             line: self.current_line,
336             stmt_type: StmtType::Return {
337                 expression: self.expression()?
338             }
339         })
340     }
341
342     /// <var> ::= Identifier Equal <expression>
343     fn var(&mut self) -> Result<Stmt, ErrorType> {
344         if let Some(target_variable_token) =
    ↪  self.check_and_consume(&[TokenType::Identifier]) {
345             // If an Identifier was given (the target variable name), consume it.
346
347             // Consume Equal.
348             self.expect(TokenType::Equal, '=')?;
349
350             // Parse <expression>.
351             let value = self.expression()?;
352             Ok(Stmt {
353                 line: self.current_line,
354                 stmt_type: StmtType::VarDecl {
355                     name: target_variable_token.lexeme,
356                     value,
357                 }
358             })
359         } else {
360             // If an Identifier was not given, raise a specific error.
361             Err(ErrorType::ExpectedVariableName { line: self.current_line })
362         }
363     }
364
365     /// <while> ::= LeftParen <expression> RightParen <block>
366     fn while_(&mut self) -> Result<Stmt, ErrorType> {
367         // Consume LeftParen.
368         self.expect(TokenType::LeftParen, '(')?;
369
370         // Parse <expression>, the condition of the `while` loop.
371         let condition = self.expression()?;
372
373         // Consume RightParen.
374         self.expect(TokenType::RightParen, ')')?;
375
376         // Parse <block>, the body of the `while` loop.
377         let body = self.block()?;
378
379         Ok(Stmt {
380             line: self.current_line,
381             stmt_type: StmtType::While {
382                 condition,
383                 body: Box::new(body),
384             }
385         })
386     }
```

```
387
388     /// Parses an expression.
389     /// <expression> ::= <assignment>
390     fn expression(&mut self) -> Result<Expr, ErrorType> {
391         self.assignment()
392     }
393
394     /// <assignment> ::= <or> (Equal <assignment>)?
395     /// Note that it is done recursively instead of <assignment> ::= <or> (Equal <or>)* as
        ↪ it
396     /// is easier to enforce right associativity with recursion on the right-hand side
        ↪ expression.
397     /// In other words, the parse tree should look like this `a = (b = (c = 3))` as opposed
        ↪ to
398     /// `((a = b) = c) = 3`
399     fn assignment(&mut self) -> Result<Expr, ErrorType> {
400         // Parse <or>, i.e., any expression with higher precedence.
401         let expr = self.or()?;
402
403         if self.check_and_consume(&[TokenType::Equal]).is_some() {
404             // If an Equal was given, consume it.
405
406             // Recursively parse <assignment>.
407             let value = self.assignment()?;
408
409             Ok(Expr {
410                 line: self.current_line,
411                 expr_type: ExprType::Assignment {
412                     target: Box::new(expr),  // Use the <or> as the `target` of the
                        ↪ Assignment.
413                     value: Box::new(value),
414                 }
415             })
416         } else {
417             // If an Equal does not follow, just return the expression as is.
418             Ok(expr)
419         }
420     }
421
422     /// <or> ::= <and> (Or <and>)*
423     /// Note that here, iteration is used instead of recursion because we can iteratively
424     /// replace `expr` with another expression, using the previous `expr` as the left-hand
        ↪ side.
425     /// This enforces left associativity and is the natural order of evaluation for binary
        ↪ operations,
426     /// which will become relevant when two operators have the same precedence and the order
        ↪ of evaluation
427     /// depends on the order they come in, e.g., <plus_minus>.
428     /// In other words, the parse tree should look like `((a or b) or c) or d`, as opposed
        ↪ to
429     /// `a or (b or (c or d))`. This also minimises recursion; hence, it is more memory
        ↪ efficient.
430     fn or(&mut self) -> Result<Expr, ErrorType> {
431         // Parse <and>.
432         let mut expr = self.and()?;
433
434         while let Some(operator) = self.check_and_consume(&[TokenType::Or]) {
435             // While the following token is an Or, consume it and store the token object
                ↪ (Or) in `operator`.
436
437             // Parse <and>.
```

```
438             let right = self.and()?;
439             expr = Expr {
440                 line: self.current_line,
441                 expr_type: ExprType::Binary {
442                     left: Box::new(expr),  // Use the previous `expr` as the left-hand side
                        ↪  to enforce left associativity.
443                     operator,  // Store the token object (Or), as this will be used to
                        ↪  determine the operation in runtime.
444                     right: Box::new(right),
445                 }
446             };
447         }
448         Ok(expr)
449     }
450 
451     /// <and> ::= <equality> (And <equality>)*
452     /// As above.
453     fn and(&mut self) -> Result<Expr, ErrorType> {
454         let mut expr = self.equality()?;
455 
456         while let Some(operator) = self.check_and_consume(&[TokenType::And]) {
457             let right = self.equality()?;
458             expr = Expr {
459                 line: self.current_line,
460                 expr_type: ExprType::Binary {
461                     left: Box::new(expr),
462                     operator,
463                     right: Box::new(right),
464                 }
465             };
466         }
467         Ok(expr)
468     }
469 
470     /// <equality> ::= <comparison> ((EqualEqual | BangEqual) <comparison>)*
471     /// As above.
472     fn equality(&mut self) -> Result<Expr, ErrorType> {
473         let mut expr = self.comparison()?;
474 
475         while let Some(operator) = self.check_and_consume(&[TokenType::EqualEqual,
            ↪  TokenType::BangEqual]) {
476             // This time, allow both EqualEqual and BangEqual tokens as they have equal
                ↪  precedence.
477 
478             let right = self.comparison()?;
479             expr = Expr {
480                 line: self.current_line,
481                 expr_type: ExprType::Binary {
482                     left: Box::new(expr),
483                     operator,
484                     right: Box::new(right),
485                 }
486             };
487         }
488         Ok(expr)
489     }
490 
491     /// <comparison> ::= <plus_minus> ((Greater | Less | GreaterEqual | LessEqual)
        ↪  <plus_minus>)*
492     /// As above.
493     fn comparison(&mut self) -> Result<Expr, ErrorType> {
```

```
494         let mut expr = self.plus_minus()?;
495
496         while let Some(operator) = self.check_and_consume(&[TokenType::Greater,
            ↪ TokenType::Less, TokenType::GreaterEqual, TokenType::LessEqual]) {
497             let right = self.plus_minus()?;
498             expr = Expr {
499                 line: self.current_line,
500                 expr_type: ExprType::Binary {
501                     left: Box::new(expr),
502                     operator,
503                     right: Box::new(right),
504                 }
505             };
506         }
507         Ok(expr)
508     }
509
510     /// <plus_minus> ::= <star_slash_percent> ((Plus | Minus) <star_slash_percent>)*
511     /// As above.
512     fn plus_minus(&mut self) -> Result<Expr, ErrorType> {
513         let mut expr = self.star_slash_percent()?;
514
515         while let Some(operator) = self.check_and_consume(&[TokenType::Plus,
            ↪ TokenType::Minus]) {
516             let right = self.star_slash_percent()?;
517             expr = Expr {
518                 line: self.current_line,
519                 expr_type: ExprType::Binary {
520                     left: Box::new(expr),
521                     operator,
522                     right: Box::new(right),
523                 }
524             };
525         }
526         Ok(expr)
527     }
528
529     /// <star_slash_percent> ::= <unary> ((Star | Slash | Percent) <unary>)*
530     /// As above.
531     fn star_slash_percent(&mut self) -> Result<Expr, ErrorType> {
532         let mut expr = self.unary()?;
533
534         while let Some(operator) = self.check_and_consume(&[TokenType::Star,
            ↪ TokenType::Slash, TokenType::Percent]) {
535             let right = self.unary()?;
536             expr = Expr {
537                 line: self.current_line,
538                 expr_type: ExprType::Binary {
539                     left: Box::new(expr),
540                     operator,
541                     right: Box::new(right),
542                 }
543             };
544         }
545         Ok(expr)
546     }
547
548     /// <unary> ::= (Bang | Minus) <unary> | <element>
549     fn unary(&mut self) -> Result<Expr, ErrorType> {
550         if let Some(operator) = self.check_and_consume(&[TokenType::Bang, TokenType::Minus])
            ↪ {
```

```
551             // If the current token is either Bang or Minus, consume it.
552
553             // Recursively parse <unary>.
554             let right = self.unary()?;
555             Ok(Expr {
556                 line: self.current_line,
557                 expr_type: ExprType::Unary {
558                     operator,
559                     right: Box::new(right),  // Use the recursion as the right-hand side
                        ↪  expression, i.e., !(!(!(!true)))
560                 }
561             })
562         } else {
563             // Otherwise, it is of lower precedence; parse <element>.
564             self.element()
565         }
566     }
567
568     /// <element> ::= <call> (LeftSquare <expression> RightSquare)*
569     fn element(&mut self) -> Result<Expr, ErrorType> {
570         // Parse <call>, i.e., the 'array' part of an element (`a` in `a[2][3]`).
571         let mut expr = self.call()?;
572
573         while self.check_and_consume(&[TokenType::LeftSquare]).is_some() {
574             // While the following token is LeftSquare, consume it.
575
576             // Parse <expression>, i.e., the 'index' part of an element (`1+2` in
                ↪  `a[1+2]`).
577             let index = self.expression()?;
578             expr = Expr {
579                 line: self.current_line,
580                 expr_type: ExprType::Element {
581                     array: Box::new(expr),  // Use the previous `expr` as the 'array' part
                        ↪  to keep left associativity.
582                     index: Box::new(index),
583                 }
584             };
585
586             // Consume the closing RightSquare of an index.
587             self.expect(TokenType::RightSquare, ']')?;
588         }
589         Ok(expr)
590     }
591
592     /// <call> ::= <primary> (LeftParen (<expression> (Comma <expression>)*)? RightParen)*
593     fn call(&mut self) -> Result<Expr, ErrorType> {
594         // Parse <primary>, i.e., the callee (`f` in `f(2)(3)`).
595         let mut expr = self.primary()?;
596
597         while self.check_and_consume(&[TokenType::LeftParen]).is_some() {
598             // While the following token is LeftParen, consume it.
599
600             // Collect the arguments of the function call into an array.
601             let mut arguments: Vec<Expr> = Vec::new();
602
603             if !self.check_next(&[TokenType::RightParen]) {
604                 // If there are arguments, i.e., not just f()...
605                 loop {
606                     // keep parsing the argument expressions and pushing them to the array
                        ↪  of arguments...
607                     arguments.push(self.expression()?);
```

```
608                     if self.check_and_consume(&[TokenType::Comma]).is_none() {
609                         // until the next token is not a Comma, in which case, there are no more arguments.
610                         break;
611                     }
612                 }
613             }
614 
615             // Consume the closing RightParen.
616             self.expect(TokenType::RightParen, ')')?;
617 
618             expr = Expr {
619                 line: self.current_line,
620                 expr_type: ExprType::Call {
621                     callee: Box::new(expr),  // Use the previous `expr` as the 'callee' part to keep left associativity.
622                     arguments,
623                 }
624             }
625         }
626         Ok(expr)
627     }
628 
629 
630     /// <primary> ::= Literal |
631     ///               LeftParen <expression> RightParen |
632     ///               LeftSquare (<expression> (Comma <expression>)*)? RightSquare |
633     ///               LeftCurly (<expression> Colon <expression> (Comma <expression> Colon <expression>)*)? RightCurly |
634     ///               Identifier
635     fn primary(&mut self) -> Result<Expr, ErrorType> {
636         if let Some(token) = self.check_and_consume(&[
637         TokenType::String_,
638         TokenType::Number,
639         TokenType::True,
640         TokenType::False,
641         TokenType::Null
642         ]) {
643             // Literal.
644             // If the token is a `String_`, `Number`, `True`, `False`, or `Null`, use its literal value,
645             // which is stored as an attribute in the Token object.
646             Ok(Expr {
647                 line: self.current_line,
648                 expr_type: ExprType::Literal {
649                     value: token.literal
650                 }
651             })
652 
653         } else if self.check_and_consume(&[TokenType::LeftParen]).is_some() {
654             // Grouping.
655 
656             // Parse <expression>.
657             let expr = self.expression()?;
658 
659             // Consume the closing RightParen.
660             self.expect(TokenType::RightParen, ')')?;
661 
662             Ok(Expr {
663                 line: self.current_line,
664                 expr_type: ExprType::Grouping {
```

```
665                     expression: Box::new(expr)
666                 }
667             })
668 
669         } else if self.check_and_consume(&[TokenType::LeftSquare]).is_some() {
670             // Array.
671 
672             // Collect the expressions of the array elements into an array.
673             let mut elements: Vec<Expr> = Vec::new();
674 
675             if !self.check_next(&[TokenType::RightSquare]) {
676                 // If there are elements, i.e., not just []...
677                 loop {
678                     // keep parsing the element expressions and pushing them to the array of
                        ↪  elements...
679                     elements.push(self.expression()?);
680                     if self.check_and_consume(&[TokenType::Comma]).is_none() {
681                         // until the next token is not a Comma, in which case, we assume
                            ↪  there are no more elements in the array.
682                         break;
683                     }
684                 }
685             }
686 
687             // Consume the closing RightSquare.
688             self.expect(TokenType::RightSquare, ']')?;
689             Ok(Expr {
690                 line: self.current_line,
691                 expr_type: ExprType::Array {
692                     elements
693                 }
694             })
695 
696         } else if self.check_and_consume(&[TokenType::LeftCurly]).is_some() {
697             // Dictionary.
698 
699             // Collect the expressions for the key-value pairs of the dictionary into an
                ↪  array.
700             let mut elements: Vec<KeyValue<Expr>> = Vec::new();
701 
702             if !self.check_next(&[TokenType::RightCurly]) {
703                 // If there are elements, i.e., not just {}.
704                 loop {
705                     // Keep parsing the key-value expressions and pushing them to the array
                        ↪  of entries.
706 
707                     // Parse the key expression.
708                     let key = self.expression()?;
709 
710                     // Consume Colon if it follows the key.
711                     if self.check_and_consume(&[TokenType::Colon]).is_none() {
712                         // Otherwise, raise a specific error.
713                         return Err(ErrorType::ExpectedColonAfterKey { line:
                            ↪  self.current_line });
714                     }
715 
716                     // Parse the value expression.
717                     let value = self.expression()?;
718 
719                     // Push them to the array of entries.
720                     elements.push(KeyValue { key, value });
```

```
721
722                     if self.check_and_consume(&[TokenType::Comma]).is_none() {
723                         // Loop until the next token is not a Comma, in which case, we
                            ↪  assume there are no more entries in the dictionary.
724                         break;
725                     }
726                 }
727             }
728
729             // Consume the closing RightCurly.
730             self.expect(TokenType::RightCurly, '}')?;
731             Ok(Expr {
732                 line: self.current_line,
733                 expr_type: ExprType::Dictionary {
734                     elements
735                 }
736             })
737
738         } else if let Some(identifier) = self.check_and_consume(&[TokenType::Identifier]) {
739             // Variable.
740             // If the token is an Identifier, use its stored lexeme which will be the
                ↪  variable name.
741             // Note 'variable' in this case also means function names.
742             Ok(Expr {
743                 line: self.current_line,
744                 expr_type: ExprType::Variable {
745                     name: identifier.lexeme
746                 }
747             })
748
749         } else {
750             // If no rule matches the token, then we expected an expression but was not
                ↪  given one.
751             // So, raise an ExpectedExpression error.
752             Err(ErrorType::ExpectedExpression { line: self.current_line })
753         }
754     }
755
756     /// Returns `Some(token)` and advances the pointer if the type of the next token is one
        ↪  of the `expected_types`.
757     /// Otherwise, or if we are at the end of the sequence of tokens, return `None`.
758     fn check_and_consume(&mut self, expected_types: &[TokenType]) -> Option<Token> {
759         if let Some(token) = self.tokens.get(self.current_index) {
760             // If we are not at the end of the sequence of tokens...
761             if expected_types.contains(&token.type_) {
762                 // If the type of the next token is one of the `expected_types`,
763                 // increment `current_index` and update `current_line`.
764                 self.current_index += 1;
765                 self.current_line = token.line;
766                 Some(token).cloned()
767             } else {
768                 // If it the token does not match, then return `None`.
769                 None
770             }
771         } else {
772             // If we are at the end, return `None`.
773             None
774         }
775     }
776
777     /// Returns `true` if the type of the next token is one of the `expected_types`.
```

```
778     /// Otherwise, or if we are at the end of the sequence of tokens, return `false`.
779     /// The difference between this and `check_and_consume()` is that this does not advance
        ↪ the pointer if the token matches what is expected.
780     fn check_next(&self, expected_types: &[TokenType]) -> bool {
781         if let Some(token) = self.tokens.get(self.current_index) {
782             // If we are not at the end of the sequence of tokens, return whether or not
783             // the token's type is one of the `expected_types`.
784             expected_types.contains(&token.type_)
785         } else {
786             // If we are at the end, return `None`.
787             false
788         }
789     }
790 
791     /// Returns `Ok(())` and advances the pointer if the type of the next token is one of
        ↪ the `expected_types`.
792     /// Otherwise, return `Err(ErrorType::ExpectedCharacter)`.
793     /// The difference between this and `check_and_consume()` is that this does not return
        ↪ the token itself, just an error to be bubbled up.
794     fn expect(&mut self, expected_type: TokenType, expected_char: char) -> Result<(),
        ↪ ErrorType> {
795         if self.check_and_consume(&[expected_type]).is_none() {
796             // If `check_and_consume()` returned `None`, i.e., the token does not match or
                ↪ we are at the end of the sequence of tokens, return an `ExpectedCharacter`
                ↪ error.
797             return Err(ErrorType::ExpectedCharacter {
798                 expected: expected_char,
799                 line: self.current_line,
800             });
801         }
802         Ok(())
803     }
804 }
```

3.10 The `Value` enum - `value.rs`

This file contains the definitions for the `Value` and `BuiltinFunction` enums, as described in Section 2.4. Note in particular the **direct recursion** used to print arrays and dictionaries on lines 48, 60, and 62. This was outlined in Subsection 2.4.4. Note that the `fmt` function will be *implicitly* called when used in a `println!()` call, e.g., on line 101 of `interpreter.rs` (Section 3.13).

```
1  use std::fmt;
2
3  use crate::stmt::Stmt;
4  use crate::hash_table::HashTable;
5
6  /// Represents evaluated/stored values within the interpreter.
7  #[derive(Clone, Debug, PartialEq)]
8  pub enum Value {
9      Number(f64),
10     String_(String),
11     Bool(bool),
12     Array(Vec<Value>),
13     Dictionary(HashTable),
14     Function {
15         parameters: Vec<String>,
16         body: Stmt,
17     },
18     BuiltinFunction(BuiltinFunction),
19     Null,
20 }
21
22 impl Value {
23     /// Returns the string of the `Value`'s type for error reports.
24     pub fn type_to_string(&self) -> String {
25         match self {
26             Self::Number(..) => String::from("Number"),
27             Self::String_(..) => String::from("String"),
28             Self::Bool(..) => String::from("Boolean"),
29             Self::Array(..) => String::from("Array"),
30             Self::Dictionary(..) => String::from("Dictionary"),
31             Self::Function {..} | Self::BuiltinFunction(..) => String::from("Function"),
32             Self::Null => String::from("Null"),
33         }
34     }
35 }
36
37 /// Used when printing `Value`s.
38 impl fmt::Display for Value {
39     fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
40         match self {
41             Self::Number(x) => write!(f, "{}", x),
42             Self::String_(x) => write!(f, "{}", x),
43             Self::Bool(x) => write!(f, "{}", x),
44             Self::Array(array) => {
45                 write!(f, "[")?;
46                 let mut it = array.iter().peekable();
47                 while let Some(x) = it.next() {
48                     x.fmt(f)?;
49                     if it.peek().is_some() {
50                         write!(f, ", ")?;
51                     }
52                 }
53                 write!(f, "]")
54             },
```

```
55              Self::Dictionary(dict) => {
56                  let flattened = dict.flatten();
57                  write!(f, "{{")?;
58                          let mut it = flattened.iter().peekable();
59                          while let Some(key_value) = it.next() {
60                              key_value.key.fmt(f)?;
61                              write!(f, ": ")?;
62                              key_value.value.fmt(f)?;
63                              if it.peek().is_some() {
64                                  write!(f, ", ")?;
65                              }
66                          }
67                          write!(f, "}}")
68              }
69              Self::Function {..} | Self::BuiltinFunction(..) => write!(f, "<function>"),
70              Self::Null => write!(f, "Null"),
71          }
72      }
73  }
74
75  /// Built-in functions.
76  #[derive(Clone, Debug, Eq, PartialEq)]
77  pub enum BuiltinFunction {
78      Append,
79      Input,
80      Remove,
81      Size,
82      Sort,
83      ToNumber,
84      ToString,
85  }
```

3.11 Hash table - `hash_table.rs`

This file contains the `HashTable` class as discussed in Section 2.5. Note in particular the **direct recursive hashing** algorithm implemented by the function starting on line 182, with the recursive call on line 191. The recursion was described in Subsection 2.5.4. Functional programming is also used on line 139. The algorithm for testing equality between two hash tables discussed in Subsection 2.5.6 is implemented in the function starting on line 147. It will be called *implicitly* when any comparison is performed, e.g., with the `==` operator.

Note also that a `get_mut` function was added. This is so that updates to a dictionary by the environment can be carried out—Rust is strict in not allowing changes data pointed to by non-mutable references, so we need another function that returns a mutable reference.

```
1  use std::fmt::Debug;
2
3  use crate::value::Value;
4  use crate::error::ErrorType;
5
6  // Hash table constants.
7  const INITIAL_NUM_BUCKETS: usize = 16;  // Initial number of buckets in the table.
8  const MAX_NUM_BUCKETS: usize = 65536;  // The maximum number of buckets in the table.
9  const MAX_CALC: usize = 65381;  // A prime used to prevent overflow in intermediate
   ↪  calculations.
10 const MAX_LOAD_FACTOR_NUMERATOR: usize = 3;  // Numerator of the maximum load factor before
   ↪  a rehash is required (3/4).
11 const MAX_LOAD_FACTOR_DENOMINATOR: usize = 4;  // Denominator of the maximum load factor
   ↪  before a rehash is required (3/4).
12 const HASH_FIRST_N: usize = 300;  // Number of elements to hash to keep constant time
   ↪  operation.
13
14 // A key-value pair in the hash table.
15 #[derive(Clone, Debug, Eq, PartialEq)]
16 pub struct KeyValue<T> {
17     pub key: T,
18     pub value: T,
19 }
20
21 /// A hash table.
22 #[derive(Clone)]
23 pub struct HashTable {
24     array: Vec<Vec<KeyValue<Value>>>,  // The internal array of the hash table.
25     entries: usize, // The number of entries in the hash table.
26     current_num_buckets: usize,  // The current number of buckets in the table.
27 }
28
29 impl HashTable {
30     /// Initialises a new instance of `HashTable`.
31     pub fn new() -> Self {
32         Self {
33             array: vec![Vec::new(); INITIAL_NUM_BUCKETS],  // Initialise 2D array with
               ↪  `INITIAL_NUM_BUCKETS` number of empty arrays.
34             entries: 0,
35             current_num_buckets: INITIAL_NUM_BUCKETS,
36         }
37     }
38
39     /// Returns the value associated with `key`.
40     pub fn get(&self, key: &Value, line: usize) -> Result<&Value, ErrorType> {
41         // Calculate the bucket number of the key.
42         let bucket_number = self.get_bucket_number(key, line)?;
43
44         // Iterate through the bucket.
```

```
45          if let Some(key_value) = self.array[bucket_number].iter().find(|key_value|
            ↪  key_value.key == key.clone()) {
46              // If a `key_value` is found such that `key_value.key == key`, then return
                ↪  `key_value.value`.
47              Ok(&key_value.value)
48          } else {
49              // Otherwise, the key does not exist in the table. Return a KeyError, providing
                ↪  the `key` for detail.
50              Err(ErrorType::KeyError { key: key.clone(), line })
51          }
52      }
53
54      /// Returns a mutable reference to the value associated with `key`. As above.
55      pub fn get_mut(&mut self, key: &Value, line: usize) -> Result<&mut Value, ErrorType> {
56          let bucket_number = self.get_bucket_number(key, line)?;
57          if let Some(key_value) = self.array[bucket_number].iter_mut().find(|key_value|
            ↪  key_value.key == key.clone()) {
58              Ok(&mut key_value.value)
59          } else {
60              Err(ErrorType::KeyError { key: key.clone(), line })
61          }
62      }
63
64      /// Inserts a key-value pair to the table if the key does not already exist; otherwise,
        ↪  updates the existing pair with the new value.
65      pub fn insert(&mut self, key: &Value, value: &Value, line: usize) -> Result<(),
        ↪  ErrorType> {
66          // Calculate the bucket number of the key.
67          let bucket_number = self.get_bucket_number(key, line)?;
68
69          // Iterate through the bucket.
70          if let Some(key_value) = self.array[bucket_number].iter_mut().find(|key_value|
            ↪  key_value.key == key.clone()) {
71              // If a `key_value` is found such that `key_value.key == key`, then update
                ↪  `key_value.value` to `value`.
72              key_value.value = value.clone();
73          } else {
74              // Otherwise, we are adding a new entry.
75              self.entries += 1;  // Increment the number of entries in the table.
76              self.array[bucket_number].push(KeyValue {  // Push the new key-value pair into
                ↪  the bucket.
77                  key: key.clone(),
78                  value: value.clone()
79              });
80          }
81
82          // Check if the table needs rehashing.
83          self.check_load(line)?;
84
85          Ok(())
86      }
87
88      /// Removes a key-value pair from the table.
89      pub fn remove(&mut self, key: &Value, line: usize) -> Result<(), ErrorType> {
90          // Calculate the bucket number of the key.
91          let bucket_number = self.get_bucket_number(key, line)?;
92
93          // Iterate through the bucket.
94          if let Some(index) = self.array[bucket_number].iter().position(|key_value|
            ↪  key_value.key == key.clone()) {
```

```
95              // If an `index` is found such that `bucket[index].key == key`, then remove the
                ↪  entry at that index.
96              self.array[bucket_number].remove(index);
97              self.entries -= 1;  // Decrement the number of entries in the table.
98              Ok(())
99          } else {
100             // Otherwise, the key does not exist in the table. Return a KeyError, providing
                ↪  the `key` for detail.
101             Err(ErrorType::KeyError { key: key.clone(), line })
102         }
103     }
104
105     /// Returns the number of entries in the table.
106     pub fn size(&self) -> usize {
107         self.entries
108     }
109
110     /// Checks the load factor of the table and performs rehashing if required.
111     fn check_load(&mut self, line: usize) -> Result<(), ErrorType> {
112         if self.current_num_buckets < MAX_NUM_BUCKETS && self.entries *
            ↪  MAX_LOAD_FACTOR_DENOMINATOR > self.current_num_buckets *
            ↪  MAX_LOAD_FACTOR_NUMERATOR {
113             // If `current_capacity` is less than the maximum capacity and greater than the
                ↪  maximum load factor, perform rehashing.
114
115             // Make a copy of the entries in the table.
116             let copy = self.flatten();
117
118             // Double the current capacity of the table.
119             self.current_num_buckets <<= 1;
120
121             // Repopulate the internal array with `current_capacity` number of empty
                ↪  buckets.
122             self.array = vec![Vec::new(); self.current_num_buckets];
123
124             // For each entry in the saved table, re-insert it in the new table.
125             for entry in copy.iter() {
126                 self.insert(&entry.key, &entry.value, line)?;
127             }
128         }
129         Ok(())
130     }
131
132     /// Calculates the bucket number of a key.
133     fn get_bucket_number(&self, key: &Value, line: usize) -> Result<usize, ErrorType> {
134         Ok(hash(key, HASH_FIRST_N, line)?.0 % self.current_num_buckets)
135     }
136
137     /// Returns all the key-value pairs in the table in a one-dimensional array.
138     pub fn flatten(&self) -> Vec<KeyValue<Value>> {
139         self.array.clone().into_iter().flatten().collect()
140     }
141 }
142
143 /// Other parts of the interpreter rely on being able to compare two `Value`s.
144 /// Since `HashTable` will be used as as part of a `Value` variant, it has to be
    ↪  comparable.
145 /// Here, two `HashTable`s are equal if they contain the same set of key-value pairs.
146 impl PartialEq for HashTable {
147     fn eq(&self, other: &Self) -> bool {
148         // One-dimensional array of entries in `self`.
```

```rust
        let self_flattened = self.flatten();

        // Array of entries in `other`.
        let mut other_flattened = other.flatten();

        // If they do not contain the same number of entries, they are not equal.
        if self_flattened.len() != other_flattened.len() {
            return false;
        }

        // Iterate through the entries of `self`. If it exists in `other` as well, remove it from `other`.
        // If it does not, then the two hash tables do not contain the same entries, so we return `false`.
        for self_key_value in self_flattened {
            if let Some(index) = other_flattened.iter().position(|other_key_value| *other_key_value == self_key_value) {
                other_flattened.remove(index);
            } else {
                return false;
            }
        }

        // All entries in `self` correspond to an entry in `other` and they have the same number of entries, so they must be equal.
        true
    }
}

/// Used for printing hash tables.
impl Debug for HashTable {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{:?}", self.flatten())
    }
}

/// Computes and returns the (hash, elements_left) of a key.
fn hash(key: &Value, mut elements_left: usize, line: usize) -> Result<(usize, usize), ErrorType> {
    match key {
        Value::Array(array) => {
            // The `djb2` algorithm is used. (https://theartincode.stanis.me/008-djb2/)
            let mut hash_value: usize = 5381;
            let mut index: usize = 0;

            // Limit the number of elements to `elements_left`, which can change depending on the recursive calls.
            while elements_left > 0 && index < array.len() {
                let result = hash(&array[index], elements_left, line)?;  // (hash, elements_left)
                let curr = result.0 % MAX_CALC;
                hash_value = (((hash_value << 5) + hash_value) + curr) % MAX_CALC; // Equivalent to `* 33 + curr`, but faster

                elements_left = result.1;
                index += 1;
            }

            Ok((hash_value, elements_left))
        },
        Value::Bool(b) => {
```

```
202             if *b {
203                 Ok((1, elements_left - 1))
204             } else {
205                 Ok((2, elements_left - 1))
206             }
207         },
208         Value::Dictionary(..) => {
209             // Hashing dictionaries in constant time will involve more sophisticated
                ↪ techniques.
210             Err(ErrorType::CannotHashDictionary { line })
211         },
212         Value::Function {..} | Value::BuiltinFunction(..) => {
213             // It is tricky to hash functions as the comparison of two functions is not set
                ↪ in stone.
214             // So we raise a descriptive error instead.
215             Err(ErrorType::CannotHashFunction { line })
216         },
217         Value::Null => Ok((3, elements_left - 1)),
218         Value::Number(x) => {
219             // We will discard the 12 least significant bits to mask floating point
                ↪ inaccuracy.
220             let mut binary: usize = (x.to_bits() >> 12).try_into().unwrap();
221             binary %= MAX_CALC;
222 
223             // The 'Knuth Variant on Division'
224             binary = (binary * (binary + 3)) % MAX_CALC;
225             Ok((binary as usize, elements_left - 1))
226         },
227         Value::String_(s) => {
228             // Similar to arrays, we use the `djb2` algorithm.
229             let mut hash_value = 5381;
230             let mut index = 0;
231 
232             while elements_left > 0 && index < s.chars().count() {
233                 hash_value = (((hash_value << 5) + hash_value) +
                    ↪ s.chars().nth(index).unwrap() as usize) % MAX_CALC;
234                 elements_left -= 1;
235                 index += 1;
236             }
237 
238             Ok((hash_value, elements_left))
239         },
240     }
241 }
```

3.12 Environment - environment.rs

Implements the `Environment` and `Pointer` classes as described in Section 2.6. The function starting on line 169 implements the logic outlined in Subsection 2.6.2 for converting a `Value` variant to a `usize` index.

```rust
use std::collections::HashMap;

use crate::value::{Value, BuiltinFunction};
use crate::error::ErrorType;

/// Allows the updating of elements in multi-dimensional arrays and dictionaries.
#[derive(Debug)]
pub struct Pointer {
    pub name: String,  // The name of the 'base' array or dictionary.
    pub indices: Vec<Value>,  // The sequence of indices needed to access the element.
}

/// Stores variables and functions.
pub struct Environment {
    scopes: Vec<HashMap<String, Value>>,  // The 'linked list' of variable scopes. Each scope contains a hash map of name-value pairs.
}

impl Environment {
    /// Initialises a new instance of `Environment`.
    pub fn new() -> Self {
        Self {
            // Initialises the built-in functions in the base scope.
            scopes: vec![HashMap::from([
            (String::from("append"), Value::BuiltinFunction(BuiltinFunction::Append)),
            (String::from("input"), Value::BuiltinFunction(BuiltinFunction::Input)),
            (String::from("remove"), Value::BuiltinFunction(BuiltinFunction::Remove)),
            (String::from("size"), Value::BuiltinFunction(BuiltinFunction::Size)),
            (String::from("sort"), Value::BuiltinFunction(BuiltinFunction::Sort)),
            (String::from("to_number"), Value::BuiltinFunction(BuiltinFunction::ToNumber)),
            (String::from("to_string"), Value::BuiltinFunction(BuiltinFunction::ToString)),
            ])],
        }
    }

    /// Creates and enters a new scope.
    pub fn new_scope(&mut self) {
        self.scopes.push(HashMap::new());
    }

    /// Exits and removes the right-most scope.
    pub fn exit_scope(&mut self) {
        self.scopes.pop();
        if self.scopes.is_empty() {
            panic!("Exited out of base scope.");
        }
    }

    /// Declares a name-value pair in the current scope.
    pub fn declare(&mut self, name: String, value: &Value) {
        if let Some(last_scope) = self.scopes.last_mut() {
            // If there is at least one scope, insert the name-value pair into the right-most scope.
            last_scope.insert(name, value.clone());
        } else {
            // Should be unreachable.
```

```
55              panic!("No scopes to declare to.");
56          }
57      }
58
59      /// Returns the value associated with `name`. As there could be multiple values
        ↪  associated with `name`
60      /// across all the scopes, return the one in the right-most scope.
61      pub fn get(&self, name: String, line: usize) -> Result<Value, ErrorType> {
62          for scope in self.scopes.iter().rev() {
63              // Iterate from the right-most scope.
64              if let Some(object) = scope.get(&name) {
65                  // If there is a value associated with `name`, return the value
                    ↪  immediately.
66                  return Ok(object.clone());
67              }
68          }
69          // We have iterated through all the scopes and no value have been found to be
            ↪  associated with `name`.
70          // So raise a `NameError`, giving the `name` in question to be as detailed as
            ↪  possible.
71          Err(ErrorType::NameError { name, line })
72      }
73
74      /// Updates the value associated with the pointer. Again, update the one in the
        ↪  right-most scope only.
75      pub fn update(&mut self, pointer: &Pointer, value: &Value, line: usize) -> Result<(),
        ↪  ErrorType> {
76          for scope in self.scopes.iter_mut().rev() {
77              // Iterate from the right-most scope.
78              if let Some(object) = scope.get_mut(&pointer.name) {
79                  // If there is a value associated with `pointer.name`...
80                  if !pointer.indices.is_empty() {
81                      // If indices were provided...
82
83                      // This is the array/dictionary associated with `pointer.name`.
84                      let mut current_element = object;
85
86                      // For each index in `pointer.indices` except the last, replace
                        ↪  `current_element` with `current_element[index]`.
87                      for i in pointer.indices.iter().take(pointer.indices.len() - 1) {
88                          match current_element {
89                              Value::Array(array) => {
90                                  // If `current_element` is an array, we have to convert the
                                    ↪  index into `usize` and make sure
91                                  // it is not out-of-bounds.
92                                  let idx = index_value_to_usize(i, line)?;
93                                  if let Some(el) = array.get_mut(idx) {
94                                      current_element = el;
95                                  } else {
96                                      // If the index provided is out-of-bounds, raise an
                                        ↪  `OutOfBoundsIndexError`.
97                                      return Err(ErrorType::OutOfBoundsIndex { index: idx,
                                        ↪  line });
98                                  }
99                              },
100                             Value::Dictionary(dict) => {
101                                 // If `current_element` is a dictionary, we can let
                                    ↪  `HashTable` get `current_element[index]`.
102                                 current_element = dict.get_mut(i, line)?;
103                             },
```

```
104                         // If it is any other variant of `Value`, then we cannot index
                            ↪ it.
105                         // Note: strings can only be indexed with the last index, so it
                            ↪ is not included here.
106                         _ => return Err(ErrorType::NotIndexable { line }),
107                     }
108                 }
109
110                 // Note that the last index is separated so that:
111                 // 1. Dictionaries can insert key-value pairs with the last key if it
                    ↪ does not exist already.
112                 //    For example, `a[1][5] = 1` inserts `5` as a key if it does not
                    ↪ exist already (`a[1]` is a dictionary).
113                 // 2. For strings, you have to do it this way to allow mutations like
                    ↪ `a[2][1] = 'h'`.
114                 let last_index = pointer.indices.last().unwrap();
115                 match current_element {
116                     Value::Array(array) => {
117                         // As above.
118                         let idx = index_value_to_usize(last_index, line)?;
119                         if let Some(el) = array.get_mut(idx) {
120                             current_element = el;
121                         } else {
122                             // If the index provided is out-of-bounds or similar...
123                             return Err(ErrorType::OutOfBoundsIndex { index: idx, line
                                ↪ });
124                         }
125                         *current_element = value.clone();
126                     },
127                     Value::Dictionary(dict) => {
128                         // `HashTable` inserts key-value pairs if the key does not exist
                            ↪ already and updates them otherwise.
129                         dict.insert(last_index, value, line)?;
130                     },
131                     Value::String_(s) => {
132                         // Convert the index value into a `usize`.
133                         let idx = index_value_to_usize(last_index, line)?;
134
135                         // Make sure it is not out-of-bounds.
136                         if s.get(idx..idx+1).is_none() {
137                             return Err(ErrorType::OutOfBoundsIndex { index: idx, line
                                ↪ });
138                         }
139
140                         if let Value::String_(c) = value {
141                             // If `value` is a string, replace `current_element[index]`
                                ↪ with `value`.
142                             s.replace_range(idx..idx+1, c);
143                         } else {
144                             // Otherwise, it cannot be inserted into a string.
145                             return Err(ErrorType::InsertNonStringIntoString { line });
146                         }
147                     },
148                     // Any other variant of `Value` cannot be indexed.
149                     _ => return Err(ErrorType::NotIndexable { line }),
150                 }
151
152                 return Ok(());
153
154             } else {
```

```
155                     // If no indices were provided, simply replace the value associated with
                        ↪  `pointer.name` with `value`.
156                     // Note: `insert()` will update the key-value pair if the key exists
                        ↪  already.
157                     scope.insert(pointer.name.clone(), value.clone());
158                     return Ok(());
159                 }
160             }
161         }
162         // We have iterated through all the scopes and no value have been found to be
            ↪  associated with `name`.
163         // So raise a `NameError`, giving the `name` in question to be as detailed as
            ↪  possible.
164         Err(ErrorType::NameError { name: pointer.name.clone(), line })
165     }
166 }
167
168 /// Converts a variant of `Value` into a usize. If it cannot, raises an appropriate error.
169 pub fn index_value_to_usize(index: &Value, line: usize) -> Result<usize, ErrorType> {
170     match index {
171         Value::Number(index_num) => {
172             // If `index` is a `Number` variant...
173
174             // and the number is non-negative and an integer...
175             if *index_num >= 0.0 && index_num.fract() == 0.0 {
176                 // ...convert it into a `usize`.
177                 Ok(*index_num as usize)
178             } else {
179                 // If it is not non-negative or it is not an integer, then raise an error as
                    ↪  it cannot be used as an index.
180                 Err(ErrorType::NonNaturalIndex { got: index.clone(), line })
181             }
182         },
183         // If it is not a `Number` variant, then it cannot be used as an index, so raise an
            ↪  error.
184         _ => Err(ErrorType::NonNumberIndex { got: index.type_to_string(), line })
185     }
186 }
```

3.13 Evaluation and execution - interpreter.rs

This file contains the interpreter as described in Section 2.7. Note in particular the use of **direct recursion** in many places to traverse, execute, and evaluate the abstract syntax tree generated by the parser, e.g., on lines 46, 85, 156, 178, etc., as well as the **indirect recursion** with the `execute` function calling the `evaluate` function and vice versa (see lines 80, 101, 323, etc.). Note also the use of functional programming to achieve more readable and idiomatic code, e.g., on line 156.

The **recursive** pointer construction algorithm as discussed in Subsection 2.7.2 is implemented in the function starting on line 638 with the recursive call on line 644; the **recursive merge sort** algorithm from Subsection 2.7.6 is implemented in the function starting on line 665 with the recursive calls on lines 676 and 677.

```rust
use std::io::{Write, self};

use crate::environment::{Environment, Pointer, self};
use crate::expr::{Expr, ExprType};
use crate::token::{TokenType, Literal};
use crate::error::{ErrorType, self};
use crate::stmt::{Stmt, StmtType};
use crate::value::{Value, BuiltinFunction};
use crate::hash_table::HashTable;

/// Recursively traverses the abstract syntax tree, executes statements, and evaluates expressions.
pub struct Interpreter {
    environment: Environment,
}

impl Interpreter {
    /// Initialises a new instance of `Interpreter`.
    pub fn new() -> Self {
        Self {
            environment: Environment::new(),
        }
    }

    /// Executes statements in the given abstract syntax tree.
    pub fn interpret(&mut self, ast: Vec<Stmt>) {
        for stmt in &ast {
            // Iterate through each statement.
            if let Err(e) = self.execute(stmt) {
                // If an error occurred in the execution of the statement, report the error and terminate execution.
                error::report_errors(&[e]);
                return;
            }
        }
    }

    /// Executes the given statement.
    fn execute(&mut self, stmt: &Stmt) -> Result<(), ErrorType> {
        match &stmt.stmt_type {
            StmtType::Block { body } => {
                // Create a new variable scope.
                self.environment.new_scope();

                // Recursively execute each statement in the body of the `Block`.
                for block_stmt in body {
                    // We cannot just use `?` here as it will exit this function call right away and not call `exit_scope()`.
                    if let Err(e) = self.execute(block_stmt) {
                        self.environment.exit_scope();
```

```
                    return Err(e);
                }
            }

            // Exit and remove the scope.
            self.environment.exit_scope();
            Ok(())
        },

        StmtType::Break => {
            // Throw a `ThrownBreak` error which can be caught in the `While` statement (see below).
            // This immediately stops execution and unwinds the call stack to the nearest parent `While` statement, which emulates the behaviour of a `break` statement.
            Err(ErrorType::ThrownBreak { line: stmt.line })
        },

        StmtType::Expression { expression } => {
            // Evaluate the expression.
            // This is used for expressions with side effects, e.g., assignments and function calls.
            self.evaluate(expression)?;
            Ok(())
        },

        StmtType::Function { name, parameters, body } => {
            // Declare the function as a new `Value` in the environment.
            self.environment.declare(name.clone(), &Value::Function {
                parameters: parameters.clone(),
                body: *body.clone(),
            });
            Ok(())
        },

        StmtType::If { condition, then_body, else_body } => {
            match self.evaluate(condition)? {
                Value::Bool(condition_bool) => {
                    // If the condition evaluated to a Boolean value...
                    if condition_bool {
                        // and the condition is `true`, execute the `then` body.
                        self.execute(then_body.as_ref())?;
                    } else if let Some(else_) = else_body {
                        // and the condition is `false`, and there is an `else` body, then execute that.
                        self.execute(else_.as_ref())?;
                    }
                    // Otherwise, do nothing.
                    Ok(())
                },
                // If the condition did not evaluate to a Boolean value, we cannot use it as the condition in an `If` statement.
                // Raise a clear and specific error.
                _ => Err(ErrorType::IfConditionNotBoolean { line: condition.line })
            }
        },

        StmtType::Print { expression } => {
            // Print the evaluated expression.
            println!("{}", self.evaluate(expression)?);
            Ok(())
```

```
103             },
104
105             StmtType::Return { expression } => {
106                 // Similar to the `Break` statement, we throw a 'dummy' error.
107                 // We also have to pass the value to be used as the return value of the
                    ↪ function call.
108                 Err(ErrorType::ThrownReturn {
109                     value: self.evaluate(expression)?,
110                     line: stmt.line
111                 })
112             },
113
114             StmtType::VarDecl { name, value } => {
115                 // Evaluate the value.
116                 let value_eval = &self.evaluate(value)?;
117
118                 // Declare the new variable in the environment.
119                 self.environment.declare(name.clone(), value_eval);
120                 Ok(())
121             },
122
123             StmtType::While { condition, body } => {
124                 loop {
125                     let continue_ = match self.evaluate(condition)? {
126                         // If `condition` evaluated to a Boolean value, set `continue_` to
                            ↪ the result of that.
127                         Value::Bool(condition_bool) => condition_bool,
128                         // Otherwise, it cannot be used as the condition for a loop, so
                            ↪ raise a specific error.
129                         _ => return Err(ErrorType::LoopConditionNotBoolean { line: stmt.line
                            ↪ }),
130                     };
131
132                     // If the `condition` evaluated to `false`, stop the loop.
133                     if !continue_ {
134                         break;
135                     }
136
137                     match self.execute(body.as_ref()) {
138                         // If the body executed with no errors, continue as normal.
139                         Ok(()) => (),
140                         // If a `ThrownBreak` error was thrown somewhere in the body, break
                            ↪ the loop.
141                         Err(ErrorType::ThrownBreak {..}) => break,
142                         // If a different error was thrown, continue to bubble up that
                            ↪ error.
143                         Err(e) => return Err(e),
144                     }
145                 }
146                 Ok(())
147             },
148         }
149     }
150
151     /// Evaluates the given expression.
152     fn evaluate(&mut self, expr: &Expr) -> Result<Value, ErrorType> {
153         match &expr.expr_type {
154             ExprType::Array { elements } => {
155                 // Evaluate each expression in the array to a `Value`, and collect those in
                    ↪ an array.
```

```
156                 let values: Result<Vec<Value>, _> = elements.iter().map(|x|
                    ↪ self.evaluate(x)).collect();
157                 Ok(Value::Array(values?))
158             },
159
160             ExprType::Assignment { target, value } => {
161                 // Evaluate the value.
162                 let value_eval = self.evaluate(value.as_ref())?;
163
164                 // Construct the pointer to the target.
165                 match self.construct_pointer(target, expr.line) {
166                     // Use the pointer to update the value in the environment.
167                     Ok(pointer) => self.environment.update(&pointer, &value_eval,
                        ↪ expr.line)?,
168                     // If an error occurred (invalid assignment target), continue to bubble
                        ↪ it up.
169                     Err(e) => return Err(e),
170                 };
171
172                 // Evaluate to the right-hand side value, e.g., a = (b = 5) -> a = 5.
173                 Ok(value_eval)
174             },
175
176             ExprType::Binary { left, operator, right } => {
177                 // Evaluate the left- and right-hand side expressions.
178                 let left_eval = self.evaluate(left.as_ref())?;
179                 let right_eval = self.evaluate(right.as_ref())?;
180
181                 match operator.type_ {
182                     // Perform the appropriate operation based on the type of the `operator`
                        ↪ token.
183                     TokenType::Or |
184                     TokenType::And => {
185                         match (&left_eval, &right_eval) {
186                             (Value::Bool(left_bool), Value::Bool(right_bool)) => {
187                                 match operator.type_ {
188                                     TokenType::Or => Ok(Value::Bool(*left_bool ||
                                        ↪ *right_bool)),
189                                     TokenType::And => Ok(Value::Bool(*left_bool &&
                                        ↪ *right_bool)),
190                                     _ => unreachable!(),
191                                 }
192                             },
193                             (_, _) => {
194                                 // We can only perform logical operations if both sides
                                    ↪ evaluate to Booleans.
195                                 // If this is not the case, raise a descriptive error.
196                                 Err(ErrorType::BinaryTypeError {
197                                     expected: String::from("Boolean"),
198                                     got_left: left_eval.type_to_string(),
199                                     got_right: right_eval.type_to_string(),
200                                     line: left.line,
201                                 })
202                             }
203                         }
204                     },
205
206                     TokenType::EqualEqual => Ok(Value::Bool(left_eval == right_eval)),
207                     TokenType::BangEqual => Ok(Value::Bool(left_eval != right_eval)),
208
209                     TokenType::Greater |
```

```
210                     TokenType::Less |
211                     TokenType::GreaterEqual |
212                     TokenType::LessEqual => {
213                         match (&left_eval, &right_eval) {
214                             (Value::Number(left_num), Value::Number(right_num)) => {
215                                 match operator.type_ {
216                                     TokenType::Greater => Ok(Value::Bool(left_num >
                                        ↪  right_num)),
217                                     TokenType::Less => Ok(Value::Bool(left_num <
                                        ↪  right_num)),
218                                     TokenType::GreaterEqual => Ok(Value::Bool(left_num >=
                                        ↪  right_num)),
219                                     TokenType::LessEqual => Ok(Value::Bool(left_num <=
                                        ↪  right_num)),
220                                     _ => unreachable!(),
221                                 }
222                             },
223                             (Value::String_(left_str), Value::String_(right_str)) => {
224                                 match operator.type_ {
225                                     TokenType::Greater => Ok(Value::Bool(left_str >
                                        ↪  right_str)),
226                                     TokenType::Less => Ok(Value::Bool(left_str <
                                        ↪  right_str)),
227                                     TokenType::GreaterEqual => Ok(Value::Bool(left_str >=
                                        ↪  right_str)),
228                                     TokenType::LessEqual => Ok(Value::Bool(left_str <=
                                        ↪  right_str)),
229                                     _ => unreachable!(),
230                                 }
231                             },
232                             (_, _) => {
233                                 Err(ErrorType::BinaryTypeError {
234                                     expected: String::from("Number or String"),
235                                     got_left: left_eval.type_to_string(),
236                                     got_right: right_eval.type_to_string(),
237                                     line: left.line,
238                                 })
239                             }
240                         }
241                     },
242 
243                     TokenType::Plus => {
244                         match (&left_eval, &right_eval) {
245                             (Value::Number(left_num), Value::Number(right_num)) =>
                                ↪  Ok(Value::Number(left_num + right_num)),
246                             (Value::String_(left_str), Value::String_(right_str)) =>
                                ↪  Ok(Value::String_(format!("{}{}", left_str, right_str))),
247                             (_, _) => {
248                                 Err(ErrorType::BinaryTypeError {
249                                     expected: String::from("Number or String"),
250                                     got_left: left_eval.type_to_string(),
251                                     got_right: right_eval.type_to_string(),
252                                     line: left.line,
253                                 })
254                             }
255                         }
256                     },
257                     TokenType::Minus |
258                     TokenType::Star |
259                     TokenType::Slash |
260                     TokenType::Percent => {
```

```
                    match (&left_eval, &right_eval) {
                        (Value::Number(left_num), Value::Number(right_num)) => {
                            match operator.type_ {
                                TokenType::Minus => Ok(Value::Number(left_num -
                                ↪  right_num)),
                                TokenType::Star => Ok(Value::Number(left_num *
                                ↪  right_num)),
                                TokenType::Slash => {
                                    if *right_num == 0.0 {
                                        Err(ErrorType::DivideByZero { line: right.line
                                        ↪  })
                                    } else {
                                        Ok(Value::Number(left_num / right_num))
                                    }
                                },
                                TokenType::Percent => Ok(Value::Number(left_num %
                                ↪  right_num)),
                                _ => unreachable!(),
                            }
                        },
                        (_, _) => {
                            Err(ErrorType::BinaryTypeError {
                                expected: String::from("Number"),
                                got_left: left_eval.type_to_string(),
                                got_right: right_eval.type_to_string(),
                                line: left.line,
                            })
                        }
                    }
                },
                // This is unreachable because the parser only builds Binary expressions
                ↪  with certain tokens.
                _ => unreachable!(),
            }
        },

        ExprType::Call { callee, arguments } => {
            // Evaluate the callee.
            let function = self.evaluate(callee.as_ref())?;

            match function {
                Value::Function { parameters, body } => {
                    // User-defined functions.
                    if arguments.len() != parameters.len() {
                        // If the number of arguments given does not match the number of
                        ↪  parameters expected, raise a detailed error.
                        return Err(ErrorType::ArgParamNumberMismatch {
                            arg_number: arguments.len(),
                            param_number: parameters.len(),
                            line: expr.line
                        });
                    }

                    // Iterate through the arguments and evaluate each.
                    let mut args_eval = Vec::new();
                    for arg in arguments.iter() {
                        args_eval.push(self.evaluate(arg)?);
                    }

                    // Create a new variable scope for the arguments and function
                    ↪  execution.
```

```
315                     self.environment.new_scope();
316
317                     // Declare the arguments in the new scope.
318                     for i in 0..arguments.len() {
319                         self.environment.declare(parameters[i].clone(), &args_eval[i]);
320                     }
321
322                     // Execute function body.
323                     let exec_result = self.execute(&body);
324
325                     // Exit scope.
326                     self.environment.exit_scope();
327
328                     match exec_result {
329                         // If the function execution did not raise any error, evaluate
                            ↪ the call to `Null` (no return statement used in function).
330                         Ok(()) => Ok(Value::Null),
331                         // If the execution ended because of a raised `ThrownReturn`
                            ↪ error, then evaluate the call to the given return vale.
332                         Err(ErrorType::ThrownReturn { value, line: _ }) => Ok(value),
333                         // If another error occurred, continue to bubble up the error.
334                         Err(e) => Err(e),
335                     }
336                 },
337
338                 Value::BuiltinFunction(function) => {
339                     // Built-in functions.
340                     match function {
341                         BuiltinFunction::Append => {
342                             // We want two arguments: the target array, and the value to
                                ↪ append.
343                             if arguments.len() != 2 {
344                                 // If the number of given arguments was not 2, raise an
                                    ↪ error, providing the number of arguments received.
345                                 return Err(ErrorType::ArgParamNumberMismatch {
                                    ↪ arg_number: arguments.len(), param_number: 2, line:
                                    ↪ expr.line });
346                             }
347
348                             let target = &arguments[0];
349                             let target_eval = self.evaluate(target)?;
350                             let pointer = self.construct_pointer(target, target.line)?;
351
352                             let value_eval = self.evaluate(&arguments[1])?;
353
354                             if let Value::Array(mut array) = target_eval {
355                                 // If `target` is an Array variant of Value, append and
                                    ↪ update the environment using the pointer.
356                                 array.push(value_eval);
357                                 self.environment.update(&pointer,
                                    ↪ &Value::Array(array.clone()), expr.line)?;
358
359                                 // Evaluate to changed array.
360                                 Ok(Value::Array(array))
361                             } else {
362                                 // We can only append to arrays.
363                                 // If `target` is not an Array variant, raise an
                                    ↪ `ExpectedTypeError` and provide the received type.
364                                 Err(ErrorType::ExpectedType { expected:
                                    ↪ String::from("Array"), got:
                                    ↪ target_eval.type_to_string(), line: target.line })
```

```
365                         }
366                     },
367                     BuiltinFunction::Input => {
368                         // We want one argument: the input prompt.
369                         if arguments.len() != 1 {
370                             return Err(ErrorType::ArgParamNumberMismatch {
                                ↪  arg_number: arguments.len(), param_number: 1, line:
                                ↪  expr.line });
371                         }
372
373                         // Print the input prompt.
374                         let prompt = self.evaluate(&arguments[0])?;
375                         print!("{}", prompt);
376                         io::stdout().flush().expect("Error: flush failed");
377
378                         // Read input.
379                         let mut input = String::new();
380                         io::stdin().read_line(&mut input).expect("Error: something
                            ↪  went wrong while reading input");
381                         input = input.trim().to_string();
382
383                         // Evaluate to input string.
384                         Ok(Value::String_(input))
385                     },
386                     BuiltinFunction::Remove => {
387                         // We want two arguments: the target array/dictionary, and
                            ↪  the index/key to remove.
388                         if arguments.len() != 2 {
389                             return Err(ErrorType::ArgParamNumberMismatch {
                                ↪  arg_number: arguments.len(), param_number: 2, line:
                                ↪  expr.line });
390                         }
391
392                         let target = &arguments[0];
393                         let target_eval = self.evaluate(target)?;
394                         let pointer = self.construct_pointer(target, target.line)?;
395
396                         let key_eval = self.evaluate(&arguments[1])?;
397
398                         match target_eval {
399                             Value::Array(mut array) => {
400                                 // If `target` is an Array variant...
401
402                                 // Convert `key` into a `usize` index.
403                                 let index =
                                    ↪  environment::index_value_to_usize(&key_eval,
                                    ↪  arguments[1].line)?;
404
405                                 if index < array.len() {
406                                     // If `index` is not out-of-bounds, perform the
                                        ↪  removal.
407                                     // Note `usize` is guaranteed to be
                                        ↪  non-negative.
408                                     array.remove(index);
409                                 } else {
410                                     // Otherwise, raise an out-of-bounds error.
411                                     return Err(ErrorType::OutOfBoundsIndex { index,
                                        ↪  line: arguments[1].line });
412                                 }
413
414                                 // Update the environment with the new array.
```

```
415                             self.environment.update(&pointer,
                                ↪  &Value::Array(array.clone()), expr.line)?;
416
417                             // Evaluate to the changed array.
418                             Ok(Value::Array(array))
419                         },
420                         Value::Dictionary(mut dict) => {
421                             // If `target` is a Dictionary variant, we can let
                                ↪  `HashTable` take care of the removal.
422                             dict.remove(&key_eval, expr.line)?;
423
424                             // Update the environment with the new dictionary.
425                             self.environment.update(&pointer,
                                ↪  &Value::Dictionary(dict.clone()), expr.line)?;
426
427                             // Evaluate to the changed dictionary.
428                             Ok(Value::Dictionary(dict))
429                         },
430                         // If it is not an Array or a Dictionary variant, then
                            ↪  raise an `ExpectedTypeError`, providing the received
                            ↪  type.
431                         _ => Err(ErrorType::ExpectedType { expected:
                            ↪  String::from("Array or Dictionary"), got:
                            ↪  target_eval.type_to_string(), line: target.line }),
432                     }
433                 },
434                 BuiltinFunction::Size => {
435                     // We want one argument: the target
                        ↪  array/dictionary/string.
436                     if arguments.len() != 1 {
437                         return Err(ErrorType::ArgParamNumberMismatch {
                            ↪  arg_number: arguments.len(), param_number: 1, line:
                            ↪  expr.line });
438                     }
439
440                     let value = self.evaluate(&arguments[0])?;
441                     match value {
442                         Value::Array(array) => Ok(Value::Number(array.len() as
                            ↪  f64)),
443                         Value::Dictionary(dict) => Ok(Value::Number(dict.size()
                            ↪  as f64)),
444                         Value::String_(s) => Ok(Value::Number(s.len() as f64)),
445                         // If `value` did not evaluate to an Array, a
                            ↪  Dictionary, or a String, raise an error.
446                         _ => Err(ErrorType::ExpectedType { expected:
                            ↪  String::from("Array, Dictionary, or String"), got:
                            ↪  value.type_to_string(), line: expr.line }),
447                     }
448                 },
449                 BuiltinFunction::Sort => {
450                     // We want one argument: the array to be sorted.
451                     if arguments.len() != 1 {
452                         return Err(ErrorType::ArgParamNumberMismatch {
                            ↪  arg_number: arguments.len(), param_number: 1, line:
                            ↪  expr.line });
453                     }
454
455                     let value = self.evaluate(&arguments[0])?;
456                     match value {
457                         // If given argument is an array, sort using the
                            ↪  `merge_sort` function defined below.
```

```
458                             Value::Array(array) =>
                                ↪  Ok(Value::Array(merge_sort(&array,
                                ↪  arguments[0].line)?)),
459
460                             // We cannot sort objects which are not arrays, so raise
                                ↪  an error.
461                             _ => Err(ErrorType::ExpectedType { expected:
                                ↪  String::from("Array"), got: value.type_to_string(),
                                ↪  line: expr.line }),
462                         }
463                     },
464                     BuiltinFunction::ToNumber => {
465                         // We want one argument: the Boolean/number/string to be
                            ↪  converted.
466                         if arguments.len() != 1 {
467                             return Err(ErrorType::ArgParamNumberMismatch {
                                ↪  arg_number: arguments.len(), param_number: 1, line:
                                ↪  expr.line });
468                         }
469
470                         let value = self.evaluate(&arguments[0])?;
471                         match value {
472                             Value::Bool(b) => {
473                                 match b {
474                                     true => Ok(Value::Number(1.0)),
475                                     false => Ok(Value::Number(0.0)),
476                                 }
477                             },
478                             Value::Number(..) => Ok(value),
479                             Value::String_(s) => {
480                                 match s.parse::<f64>() {
481                                     Ok(x) => Ok(Value::Number(x)),
482                                     // If something went wrong during Rust's
                                        ↪  conversion, raise an error.
483                                     Err(..) => Err(ErrorType::CannotConvertToNumber
                                        ↪  { line: expr.line }),
484                                 }
485                             },
486
487                             // We can only construct numeric representations of
                                ↪  Booleans, numbers, and strings.
488                             // If not given one of these, raise an error.
489                             _ => Err(ErrorType::ExpectedType { expected:
                                ↪  String::from("Boolean, Number or String"), got:
                                ↪  value.type_to_string(), line: expr.line }),
490                         }
491                     },
492                     BuiltinFunction::ToString => {
493                         // We want one argument: the Boolean/number/string to be
                            ↪  converted.
494                         if arguments.len() != 1 {
495                             return Err(ErrorType::ArgParamNumberMismatch {
                                ↪  arg_number: arguments.len(), param_number: 1, line:
                                ↪  expr.line });
496                         }
497
498                         let value = self.evaluate(&arguments[0])?;
499                         match value {
500                             Value::Bool(b) => {
501                                 match b {
```

```
                                    true =>
                                    ↪  Ok(Value::String_(String::from("true"))),
                                    false =>
                                    ↪  Ok(Value::String_(String::from("false"))),
                                }
                            },
                            Value::Number(x) => Ok(Value::String_(x.to_string())),
                            Value::String_(..) => Ok(value),

                            // We can only construct string representations of
                            ↪  Booleans, numbers, and strings.
                            // If not given one of these, raise an error.
                            _ => Err(ErrorType::ExpectedType { expected:
                            ↪  String::from("Boolean, Number or String"), got:
                            ↪  value.type_to_string(), line: expr.line }),
                        }
                    },
                }
            },

            // If the evaluated `function` was not a `Function` or a
            ↪  `BuiltinFunction` variant, then we cannot 'call' it.
            // So raise an error.
            _ => Err(ErrorType::CannotCallName { line: callee.line })
        }
    },

    ExprType::Dictionary { elements } => {
        // Create a new hash table.
        let mut hash_table = HashTable::new();

        // Iterate through the key-value pairs of the given elements.
        for key_value in elements.iter() {
            // Evaluate each of the keys and values.
            let key_eval = self.evaluate(&key_value.key)?;
            let value_eval = self.evaluate(&key_value.value)?;

            // Insert the evaluated key and value into the table.
            hash_table.insert(&key_eval, &value_eval, expr.line)?;
        }
        Ok(Value::Dictionary(hash_table))
    },

    ExprType::Element { array, index } => {
        // Note that 'array' refers to anything to the left of the index, e.g.,
        // the 'array' in `a[1][2]` is `a[1]` and the index is `2`.

        // Evaluate the index expression.
        let index_eval = self.evaluate(index.as_ref())?;

        match self.evaluate(array.as_ref())? {  // Evaluate `array`.
            Value::Array(array) => {
                // If the evaluated 'array' is an Array variant, convert the
                ↪  evaluated index to a `usize` index.
                let index_num = environment::index_value_to_usize(&index_eval,
                ↪  index.line)?;

                // Try to get the element of `array` at index `index_num`.
                if let Some(element) = array.get(index_num) {
                    Ok(element.clone())
                } else {
```

```
555                         // In this case, `index_num` was out of bounds.
556                         Err(ErrorType::OutOfBoundsIndex { index: index_num, line:
                                ↪ expr.line })
557                     }
558                 },
559                 Value::Dictionary(dict) => {
560                     // If the evaluated 'array' is a Dictionary variant, get value from
                            ↪ the `HashTable` object.
561                     dict.get(&index_eval, expr.line).cloned()
562                 },
563                 Value::String_(s) => {
564                     // If the evaluated 'array' is a String variant, convert the
                            ↪ evaluated index to a `usize` index.
565                     let index_num = environment::index_value_to_usize(&index_eval,
                            ↪ index.line)?;
566
567                     // Try to get the character of `s` at index `index_num`.
568                     if let Some(c) = s.chars().nth(index_num) {
569                         Ok(Value::String_(String::from(c)))
570                     } else {
571                         // In this case, `index_num` was out of bounds.
572                         Err(ErrorType::OutOfBoundsIndex { index: index_num, line:
                                ↪ expr.line })
573                     }
574                 },
575                 // If the 'array' was not an Array, a Dictionary, or a String variant,
                        ↪ it cannot be indexed.
576                 _ => Err(ErrorType::NotIndexable { line: array.line })
577             }
578         },
579
580         ExprType::Grouping { expression } => {
581             self.evaluate(expression.as_ref())
582         },
583
584         ExprType::Literal { value } => {
585             // Convert a `Literal` enum into a `Value` enum.
586             match value {
587                 Literal::Number(x) => Ok(Value::Number(*x)),
588                 Literal::String_(x) => Ok(Value::String_(x.clone())),
589                 Literal::Bool(x) => Ok(Value::Bool(*x)),
590                 Literal::Null => Ok(Value::Null),
591             }
592         },
593
594         ExprType::Unary { operator, right } => {
595             // Evaluate the right-hand side expression.
596             let right_eval = self.evaluate(right.as_ref())?;
597
598             match operator.type_ {
599                 TokenType::Bang => {
600                     // If the operator is `!`...
601                     match right_eval {
602                         Value::Bool(right_bool) => Ok(Value::Bool(!right_bool)),
603                         // This operation only works with Boolean values, so raise an
                                ↪ `ExpectedTypeError` error otherwise.
604                         // Provide the received type for clarity.
605                         _ => Err(ErrorType::ExpectedType {
606                             expected: String::from("Boolean"),
607                             got: right_eval.type_to_string(),
608                             line: right.line,
```

```
609                             })
610                         }
611                     },
612                     TokenType::Minus => {
613                         // If the operator is `-`...
614                         match right_eval {
615                             Value::Number(right_num) => Ok(Value::Number(-right_num)),
616                             // This operation only works with Number variants, so raise an
                                ↪  `ExpectedTypeError` error otherwise.
617                             // Provide the received type for clarity.
618                             _ => Err(ErrorType::ExpectedType {
619                                 expected: String::from("Number"),
620                                 got: right_eval.type_to_string(),
621                                 line: right.line,
622                             })
623                         }
624                     },
625                     // The parser only builds `Unary` expressions with `Bang` or `Minus`, so
                        ↪  this is unreachable.
626                     _ => unreachable!(),
627                 }
628             },
629 
630             ExprType::Variable { name } => {
631                 // Simply retrieve the value of the variable from the environment.
632                 self.environment.get(name.clone(), expr.line)
633             },
634         }
635     }
636 
637     /// Constructs a Pointer object given an expression.
638     fn construct_pointer(&mut self, element: &Expr, line: usize) -> Result<Pointer,
        ↪  ErrorType> {
639         match &element.expr_type {
640             ExprType::Element { array, index } => {
641                 // Recursive case.
642                 // E.g., a[1][2][3] -> Pointer("a", [1, 2]), [3] -> Pointer("a", [1, 2, 3])
643                 // So we simply add the index of the current element to the Pointer
                    ↪  constructed in the recursion.
644                 let Pointer {name, indices} = self.construct_pointer(array.as_ref(), line)?;
645 
646                 // Make a copy of the `indices` array and append the index of the current
                    ↪  element.
647                 let mut indices_copy = indices;
648                 indices_copy.push(self.evaluate(index.as_ref())?);
649 
650                 // Return a `Pointer` with the appended index.
651                 Ok(Pointer { name, indices: indices_copy })
652             },
653             ExprType::Variable { name } => {
654                 // Base case.
655                 // Return an empty `indices` array to be populated in the recursion.
656                 Ok(Pointer {name: name.clone(), indices: Vec::new()})
657             },
658             // Otherwise, the variant does not support assignment, so raise an error (e.g.,
                ↪  a literal array/dictionary, a binary expression, etc.).
659             _ => Err(ErrorType::InvalidAssignmentTarget { line }),
660         }
661     }
662 }
663 
```

```rust
/// Sorts the given array using merge sort.
fn merge_sort(array_to_sort: &Vec<Value>, line: usize) -> Result<Vec<Value>, ErrorType> {
    let n = array_to_sort.len();

    // Base case.
    if n <= 1 {
        return Ok(array_to_sort.to_vec());
    }

    // Recursive case.

    // Recursively sort the left and right halves of the array.
    let left = merge_sort(&array_to_sort[0..n/2].to_vec(), line)?;
    let right = merge_sort(&array_to_sort[n/2..].to_vec(), line)?;

    // Merge the two sorted arrays using two pointers.
    let mut left_index = 0;
    let mut right_index = 0;
    let mut merged = Vec::new();

    while left_index < left.len() && right_index < right.len() {
        match (&left[left_index], &right[right_index]) {
            // Append the 'lower' of the two to the merged array, and advance the respective
            ↪ pointer.
            (Value::Number(left_num), Value::Number(right_num)) => {
                if left_num < right_num {
                    merged.push(left[left_index].clone());
                    left_index += 1;
                } else {
                    merged.push(right[right_index].clone());
                    right_index += 1;
                }
            },
            (Value::String_(left_str), Value::String_(right_str)) => {
                if left_str < right_str {
                    merged.push(left[left_index].clone());
                    left_index += 1;
                } else {
                    merged.push(right[right_index].clone());
                    right_index += 1;
                }
            },

            // We only support comparisons between numbers and between strings.
            (_, _) => {
                return Err(ErrorType::BinaryTypeError {
                    expected: String::from("Number or String"),
                    got_left: left[left_index].type_to_string(),
                    got_right: right[right_index].type_to_string(),
                    line,
                });
            }
        }
    }

    // Only one of `left` and `right` will have any elements left.
    // Append the remainder to the merged array.
    if left_index < left.len() {
        while left_index < left.len() {
            merged.push(left[left_index].clone());
            left_index += 1;
```

```
724             }
725         }
726 
727         if right_index < right.len() {
728             while right_index < right.len() {
729                 merged.push(right[right_index].clone());
730                 right_index += 1;
731             }
732         }
733 
734         Ok(merged)
735     }
```

Chapter 4

Testing

4.1 Unit tests

During development, I have also made use of **unit tests** to test each part of the interpreter separately. This is thanks to object-oriented programming which allows each component to be developed and tested independently. There are tests for:

- The `Tokenizer` class
- The `Parser` class
- The `HashTable` class
- The `Environment` class

I have tried to make these tests as thorough as possible—they should check that methods do as they should, in normal and in edge cases, and that suitable errors are raised when they should be. These tests have been appended onto the end of their respective files, which is the idiom in Rust.

As these tests are quite extensive, they can be found in Appendix A along with evidence of all of them passing.

4.2 Objective tests

4.2.1 Video

I demonstrate a representative sample of these tests on video which can be found at the following link:

`https://bit.ly/nea-video-naphat-interpreter`

From 17:10 onwards, I also demonstrate some programs which show the different parts of the language working together, including:

1. A program which prints the factors of a number (17:10).
2. A program which prints the prime factors of a number (18:03).
3. A program which uses recursion to find the factorial of a number (18:50).
4. A program which uses recursion to print the moves needed to solve the Tower of Hanoi problem (19:17).

For each objective below I have included timestamps to the relevant section in the video.

4.2.2 Objective 1

Statement

Users must be able to print variables and literals.

Tests

1:00 in the video.

Source code	Expected output	Output	Passed?
`print 5`	`5`	`5`	✓
`print "abcd"`	`abcd`	`abcd`	✓
`print true`	`true`	`true`	✓
`print false`	`false`	`false`	✓
`print null`	`null`	`null`	✓
`var a = 5` `print a`	`5`	`5`	✓
`var a = "abcd"` `print a`	`abcd`	`abcd`	✓
`var a = true` `print a`	`true`	`true`	✓
`var a = false` `print a`	`false`	`false`	✓
`var a = null` `print a`	`null`	`null`	✓

Result

Objective achieved.

4.2.3 Objective 2

Statement

Mathematical, logical, and comparative expressions must be evaluated correctly with precedence.

Tests

1:22 in the video

Source code	Expected output	Output	Passed?
`print 5 + 3`	`8`	`8`	✓
`print 5 * 2 + 3`	`13`	`13`	✓
`print 5 + 2 * 3`	`11`	`11`	✓
`print (5 + 2) * 3`	`21`	`21`	✓
`print 12 / (4 - 2)`	`6`	`6`	✓
`print 12 + 25 % 3 - 2`	`11`	`11`	✓
`print ((3 + 3 * 5) / (26 / 2 - ( 3 + 2 ) * 2)) % 4`	`2`	`2`	✓
`print true or false`	`true`	`true`	✓
`print true or true and false`	`true`	`true`	✓
`print (true or true) and false`	`false`	`false`	✓
`print !(!false or true) or !false`	`true`	`true`	✓
`print 2 + 2 == 4`	`true`	`true`	✓
`print 2 * 3 != 7 - 1`	`false`	`false`	✓
`print 2 * 3 > 7 - 3`	`true`	`true`	✓
`print 2 * 3 > 7 - -9`	`false`	`false`	✓
`print 2 * 3 > 7 - -9 or 2 * 3 > 7 - 3`	`true`	`true`	✓

Result

Objective achieved.

4.2.4 Objective 3

Statement

Users must be able to store and modify values in variables with the following types:

(a) Number (includes integers and floats)

(b) Boolean

(c) String

(d) Array

(e) Dictionary

(f) Null

Tests

2:30 in the video.

Source code	Expected output	Output	Passed?

`var a = 3.14`
`print a`
`a = true`
`print a`
`a = 'abc'`
`print a`
`a = [1, 2, 3]`
`print a`
`a = {"key1": 5, "key2": true}`
`print a`
`a = null`
`print a`
	`3.14`
	`true`
	`abc`
	`[1, 2, 3]`
	`{key1: 5, key2: true}`
	`null`
		`3.14`
		`true`
		`abc`
		`[1, 2, 3]`
		`{key2: true, key1: 5}`
		`null`
			✓

Result

Objective achieved.

4.2.5 Objective 4

Statement

Users must be able to access and modify:

(a) values in arrays.

(b) values of key-value pairs in dictionaries.

(c) characters in strings.

Tests

3:44 in the video.

Source code	Expected output	Output	Passed?

`var a = [1, 5, "hello", true]`
`print a[0]`
`print a[1]`
`print a[2]`
`print a[3]`
`a[0] = 500`
`print a[0]`
`a[0] = a[0] + a[1]`
`print a[0]`
	`1`
	`5`
	`hello`
	`true`
	`500`
	`505`
		`1`
		`5`
		`hello`
		`true`
		`500`
		`505`
			✓

`var a = {"key1": 1, "key2": true, "key3": 5, "`
`  key4": "hello"}`
`print a['key1']`
`print a['key2']`
`print a['key3']`
`print a['key4']`
`a['key1'] = 123`
`print a['key1']`
`a['key1'] = a['key1'] + a['key3']`
`print a['key1']`
	`1`
	`true`
	`5`
	`hello`
	`123`
	`128`
		`1`
		`true`
		`5`
		`hello`
		`123`
		`128`
			✓

`var s = "a string"`
`print s`
`print s[5]`
`s[5] = 'p'`
`print s[5]`
`print s`
	`a string`
	`i`
	`p`
	`a strpng`
		`a string`
		`i`
		`p`
		`a strpng`
			✓

Result

Objective achieved.

4.2.6 Objective 5

Statement

The language must support the following control structures:

(a) `if`

(b) `else if`

(c) `else`

Tests

5:18 in the video.

Source code	Expected output	Output	Passed?

`if (1 + 3 == 4) {`
`  print "hello"`
`}`
	`hello`	`hello`	✓

`if (1 + 3 != 4) {`
`  print "hello"`
`}`
			✓

`if (1 + 3 != 4) {`
`  print "hello"`
`} else {`
`  print "hi"`
`}`
	`hi`	`hi`	✓

`if (1 + 3 != 4) {`
`  print "hello"`
`} else if (5 + 5 == 10) {`
`  print "ok"`
`} else {`
`  print "hi"`
`}`
	`ok`	`ok`	✓

`if (1 + 3 != 4) {`
`  print "hello"`
`} else if (5 + 5 != 10) {`
`  print "ok"`
`} else if (1 == 2 - 1) {`
`  print "here"`
`} else {`
`  print "hi"`
`}`
	`here`	`here`	✓

`if (1 + 3 != 4) {`
`  print "hello"`
`} else if (5 + 5 != 10) {`
`  print "ok"`
`} else if (1 != 2 - 1) {`
`  print "here"`
`} else {`
`  print "hi"`
`}`
	`hi`	`hi`	✓

Result

Objective achieved.

4.2.7 Objective 6

Statement

The language must support:

(a) `while` loops.

(b) `for` loops.

(c) `break` statements within loops.

Tests

6:24 in the video.

Source code	Expected output	Output	Passed?

`for (var x = 1; x <= 5; x = x + 1) {`
`  print x`
`}`
	1
	2
	3
	4
	5
		1
		2
		3
		4
		5
			✓

`var x = 1`
`while (x <= 7) {`
`  print x`
`  x = x + 2`
`}`
	1
	3
	5
	7
		1
		3
		5
		7
			✓

`for (var x = 100; x <= 200; x = x + 2) {`
`  print x`
`  if (x % 7 == 0) {`
`  break`
`  }`
`}`
	100
	102
	104
	106
	108
	110
	112
		100
		102
		104
		106
		108
		110
		112
			✓

`for (var x = 2; x < 5; x = x + 1) {`
`  var y = 0`
`  while (x + y <= 8) {`
`  print x + y`
`  y = y + 2`
`  }`
`}`
	2
	4
	6
	8
	3
	5
	7
	4
	6
	8
		2
		4
		6
		8
		3
		5
		7
		4
		6
		8
			✓

Result

Objective achieved.

4.2.8 Objective 7

Statement

The language must support:

(a) functions with parameters and functions without parameters.

(b) early `return` statements with return values.

Tests

8:03 in the video.

Source code	Expected output	Output	Passed?

`func f() {`
`  print "hello"`
`}`
`f()`
	`hello`	`hello`	✓

`func add_print(a, b) {`
`  print a + b`
`}`
`add_print(10, 20)`
	`30`	`30`	✓

`func add(a, b) {`
`  print a + b`
`}`
`print add(10, 20)`
	`30`	`30`	✓

`func max(a, b) {`
`  if (a > b) {`
`  return a`
`  } else if (a <= b) {`
`  return b`
`  }`
`  print "not returned"`
`}`
`print max(1, 5)`
`print max(6, 4)`
	`5`
	`6`
		`5`
		`6`
			✓

Result

Objective achieved.

4.2.9 Objective 8

Statement

Users must be able to use built-in functions to:

(a) prompt for input.

(b) append a value to an array.

(c) remove a value from an array or a key-value pair from a dictionary.

(d) cast a string or Boolean to a number.

(e) cast a number or Boolean to a string.

(f) get the size of an array or dictionary, and the length of a string.

(g) sort an array of numbers or strings.

Tests

9:32 in the video.

Source code	Expected output	Output	Passed?
`var a = input("Enter your name: ")` `print "Hello, " + a`	`Enter your name: ( INPUT 'John')` `Hello, John`	`Enter your name: ( INPUT 'John')` `Hello, John`	✓

`var a = [1, 2, 3]`
`print a`
`append(a, 5)`
`print a`
	`[1, 2, 3]`
	`[1, 2, 3, 5]`
		`[1, 2, 3]`
		`[1, 2, 3, 5]`
			✓

`var a = [100, 200, 300]`
`print a`
`remove(a, 1)`
`print a`
	`[100, 200, 300]`
	`[100, 300]`
		`[100, 200, 300]`
		`[100, 300]`
			✓

`var a = {"key1": 5, "key2": 6, "key3 ": 7}`
`print a`
`remove(a, "key1")`
`print a`
	`{key1: 5, key2: 6, key3: 7}`
	`{key2: 6, key3: 7}`
		`{key2: 6, key3: 7, key1: 5}`
		`{key2: 6, key3: 7}`
			✓

`var a = "123"`
`a = to_number(a)`
`print a + 2`
	`125`	`125`	✓

`var a = true`
`a = to_number(a)`
`print a + 2`
`print to_number(false)`
	`3`
	`0`
		`3`
		`0`
			✓

`var a = [1, 2, 3]` `print size(a)`	`3`	`3`	✓
`var a = {'key1': 5, 'key2': 6, 'key3 ': 7}` `print size(a)`	`3`	`3`	✓
`var a = "some really long string"` `print size(a)`	`23`	`23`	✓
`var a = [5, 2, 6, 7, 3, 7, 10, 1]` `print sort(a)`	`[1, 2, 3, 5, 6, 7, 7, 10]`	`[1, 2, 3, 5, 6, 7, 7, 10]`	✓

`var a = ['bee', 'ant', 'helicopter',`
`  'zebra', 'monkey', 'book']`
`print sort(a)`
	`[ant, bee, book,`
	`  helicopter,`
	`  monkey, zebra]`
		`[ant, bee, book,`
		`  helicopter,`
		`  monkey, zebra]`
			✓

Result

Objective achieved.

4.2.10 Objective 9

Statement

The language must support variable scoping in control structures, loops, and functions. In other words, names (variables and functions) declared within a scope must not be accessible from outside, whereas names declared outside must be accessible and modifiable from inside.

Tests

13:39 in the video.

Source code	Expected output	Output	Passed?

`var a = 10`
`print a`
`func f() {`
`  var a = 5`
`  print a`
`}`
`f()`
`print a`
	10
	5
	10
		10
		5
		10
			✓

`var a = 10`
`print a`
`func f() {`
`  a = 5`
`  print a`
`}`
`f()`
`print a`
	10
	5
	5
		10
		5
		5
			✓

`var i = 1000`
`for (var i = 0; i < 4; i = i + 1) {`
`  print i`
`}`
	0
	1
	2
	3
		0
		1
		2
		3
			✓

`var x = 10`
`func f(x) {`
`  return x + 5`
`}`
`print f(123)`
	128	128	✓

Result

Objective achieved.

4.2.11 Objective 10a

Statement

With regard to error reporting, error messages should be specific and include the type of the error, details regarding the particular situation (i.e., why it was raised), and the location.

Tests

15:18 in the video.

In the following tests, I demonstrate that all errors include specific information detailing why it was raised in that particular situation as well as the location.

Source code	Expected output	Output	Passed?
`&`	Unexpected character	`An error has occurred.` `Line 1: unexpected character `&`.`	✓
`'abc`	Unterminated string	`An error has occurred.` `A string was never closed by the end of the program.`	✓
`[1, 2`	Expected character	`An error has occurred.` `Line 1: expected character `]``	✓
`if () {}`	Expected expression	`An error has occurred.` `Line 1: expected expression.`	✓
`func`	Expected function name	`An error has occurred.` `Line 1: expected function name. Make sure it is not a keyword.`	✓
`func f(+)`	Expected parameter name	`An error has occurred.` `Line 1: expected parameter name in function declaration.`	✓
`var`	Expected variable name	`An error has occurred.` `Line 1: expected variable name. Make sure it is not a keyword.`	✓
`for (var i = 0 i < 5; i = i + 1)`	Expected semicolon after initialiser	`An error has occurred.` `Line 1: expected `;` after initialising statement in `for` loop.`	✓
`for (var i = 0; i < 5 i = i + 1)`	Expected semicolon after condition	`An error has occurred.` `Line 1: expected `;` after condition in `for` loop.`	✓
`for (var i = 0; i < 5; i = i + 1`	Expected right parenthesis after increment	`An error has occurred.` `Line 1: expected `)` after increment statement in `for` loop.`	✓
`{'key1': 5, 'key2' 6}`	Expected colon after key	`An error has occurred.` `Line 1: expected colon after dictionary key.`	✓
`print a`	Name error	`An error has occurred.` `Line 1: `a` is not defined.`	✓
`var a = 5` `a[0]`	Not indexable	`An error has occurred.` `Line 2: the value is not indexable.`	✓
`var a = [1, 2]` `a[2]`	Index out of bounds	`An error has occurred.` `Line 2: index `2` is out of bounds.`	✓

`var s = 'hello'` `s[0] = 5`	String insertion error	`An error has occurred.` `Line 2: attempted to insert a non-string into a string.`	✓
`2 = 3`	Invalid assignment target	`An error has occurred.` `Line 1: invalid assignment target. Make sure you are not assigning to a literal.`	✓
`!5`	Expected type	`An error has occurred.` `Line 1: expected type Boolean; instead got type Number.`	✓
`var a = [1, 2]` `a[-5]`	Index is not positive integer error	`An error has occurred.` `Line 2: index evaluated to -5, which is not a positive integer.`	✓
`var a = [1, 2]` `a[2.3]`	Index is not positive integer error	`An error has occurred.` `Line 2: index evaluated to 2.3, which is not a positive integer.`	✓
`var a = [1, 2]` `a['hello']`	Index is not a number error	`An error has occurred.` `Line 2: index evaluated to a String, which is not a positive integer.`	✓
`5 + 'a'`	Binary type error	`An error has occurred.` `Line 1: this operation requires both sides' types to be Number or String. Instead, got Number and String respectively.`	✓
`5 / 0`	Divide by zero error	`An error has occurred.` `Line 1: divisor is 0.`	✓
`if ('hello') {}`	If condition not Boolean value	`An error has occurred.` `Line 1: the `if` condition did not evaluate to a Boolean value.`	✓
`while ('hello') {}`	Loop condition not Boolean value	`An error has occurred.` `Line 1: the condition of the loop did not evaluate to a Boolean value.`	✓

`for (var i = 0; '`
`  hello'; i = i +`
`  1) {}`
	Loop condition not Boolean value	`An error has occurred.`
		`Line 1: the condition of the loop did not evaluate to a Boolean value.`
			✓

`var a = 'abc'` `a(1)`	Cannot call name	`An error has occurred.` `Line 1: cannot call name as a function.`	✓

`func f(a) {`
`  print a`
`}`
`f(1, 2)`
	Different number of arguments and parameters	`An error has occurred.`
		`Line 4: attempted to call function with 2 argument(s), but function accepts 1.`
			✓

`to_number('a')`	Cannot convert to number	`An error has occurred.` `Line 1: could not convert to a number.`	✓

`func f(a) {`
`  print a`
`}`
`{f: 2}`
	Cannot hash function	`An error has occurred.`
		`Line 4: cannot hash function (`
		`functions cannot be used as`
		`keys in dictionary entries).`
			✓

`var d = {'key1': 5}`
`{d: 2}`
	Cannot hash function	`An error has occurred.`
		`Line 2: cannot hash dictionary (`
		`dictionaries cannot be used as`
		`keys in dictionary entries).`
			✓

`var d = {'key1': 5}`
`d['key2']`
	Key error	`An error has occurred.`
		`Line 2: key` `` `key2` `` `does not exist`
		`in the dictionary.`
			✓

`break`	Break used outside loop	`An error has occurred.`
		`Line 1:` `` `break` `` `has to be used`
		`within a loop.`
			✓

`return 5`	Return used outside function	`An error has occurred.`
		`Line 1:` `` `return` `` `has to be used`
		`within a function.`
			✓

Result

Objective achieved.

4.2.12 Objective 10b

Statement

With regard to errors, the interpreter should report as many errors as possible in one execution.

Tests

16:25 in the video.

Source code	Expected output	Output	Passed?

`if () {}`
`func g(or) {`
`  if (true {`
`  var = 1`
`  }`
`  return`
`}`
`var d = {5 6}`
`for (var i = 0 i < 5; i = i`
`  + 1) {}`
	Line 1: expected expression
	Line 2: expected parameter name
	Line 3: expected ‘)‘
	Line 4: expected variable name
	Line 6: expected expression
	Line 8: expected colon after key
	Line 9: expected semicolon after initialiser
		`An error has occurred.`
		`Line 1: expected expression.`
		`Line 2: expected parameter`
		`name in function`
		`declaration.`
		`Line 3: expected character`
		`` `)` ``
		`Line 4: expected variable`
		`name. Make sure it is`
		`not a keyword.`
		`Line 6: expected expression.`
		`Line 8: expected colon after`
		`dictionary key.`
		`Line 9: expected` `` `;` `` `after`
		`initialising statement`
		`in` `` `for` `` `loop.`
			✓

Result

Objective achieved.

4.2.13 Objective 11

Statement

The interpreter must provide a REPL interface which prompts for and executes source code line-by-line. To do this, the user should run the executable without any arguments (`nea.exe`).

Tests

Demonstrated throughout the video.

```
C:\Users\tkbur\Desktop>nea.exe
> var a = 5
> print a
5
> a = "hello"
> print a
hello
> var b = input("Enter name: ")
Enter name: John
> print a + " " + b
hello John
>
```

Result

Objective achieved.

4.2.14 Objective 12

Statement

The interpreter must also be able to execute source code stored in files. To do this, the user should give the path to the source code file as an argument to the executable (`nea.exe file_path`).

Tests

Demonstrated throughout the video.

```
C:\Users\tkbur\Desktop>type test.txt
var a = 5
print a
a = "hello"
print a
var b = input("Enter name: ")
print a + " " + b
C:\Users\tkbur\Desktop>nea.exe test.txt
5
hello
Enter name: John
hello John
```

Note the `type` command was used to display the contents of `test.txt`.

Result

Objective achieved.

4.2.15 Summary

The above tests indicate that all objectives were achieved.

4.3 Advent of Code

Actually using the language is a good way to make sure everything works. I have used this language to solve the first 10 days of problems in the 2023 Advent of Code challenge[1].

As an example, the code below is my solution to the first part of day 4.

```
func slice(str, i, j) {
    var ret = ""
    for (; i < j; i = i + 1) {
        ret = ret + str[i]
    }
    return ret
}

func split(str, delim) {
    var a = []
    var n = size(str)
    var start = 0

    var curr = 0
    for (; curr < n; curr = curr + 1) {
        if (curr + size(delim) <= n) {
            if (slice(str, curr, curr + size(delim)) == delim) {
                append(a, slice(str, start, curr))
                curr = curr + size(delim) - 1
                start = curr + 1
            }
        }
    }
    append(a, slice(str, start, curr))
    return a
}

var num_lines = to_number(input(""))
var ans = 0
for (var _ = 0; _ < num_lines; _ = _ + 1) {
    var nums = split(split(input(""), ": ")[1], " | ")
    var have_nums_with_empties = split(nums[0], " ")
    var win_nums_with_empties = split(nums[1], " ")

    var have_nums = []
    for (var i = 0; i < size(have_nums_with_empties); i=i+1) {
        if (have_nums_with_empties[i] != "") {
            append(have_nums, to_number(have_nums_with_empties[i]))
        }
    }
    var win_nums = []
    for (var i = 0; i < size(win_nums_with_empties); i=i+1) {
        if (win_nums_with_empties[i] != "") {
            append(win_nums, to_number(win_nums_with_empties[i]))
        }
    }

    var have_nums_sorted = sort(have_nums)
    var win_nums_sorted = sort(win_nums)

    var pnts = 0
    var j = 0
    var n = size(have_nums_sorted)
    var m = size(win_nums_sorted)
```

[1] https://adventofcode.com/2023

```
55
56      for (var i = 0; i < n; i=i+1) {
57          while (j < m) {
58              if (win_nums_sorted[j] < have_nums_sorted[i]) {
59                  j = j + 1
60              } else if (win_nums_sorted[j] == have_nums_sorted[i]) {
61                  if (pnts == 0) {
62                      pnts = 1
63                  } else {
64                      pnts = pnts * 2
65                  }
66                  j = j + 1
67                  break
68              } else {
69                  break
70              }
71          }
72      }
73
74      ans = ans + pnts
75  }
76  print "Answer: " + to_string(ans)
```

Note the use of the built-in `sort` function on lines 48 and 49 and the use of user-defined functions. Although it is necessary to know the number of lines of input beforehand, it otherwise works as expected and produces a correct answer of `26443` (for me), as is evidenced below.

```
Card 211: 67 27 31 84 61 64 58 68 21 7
Card 212: 61 14 31  4  8 43 52 37 56 4
Card 213: 51 69 54 42 23 68 27 98 47
Card 214: 38  6 78 12 88 14 51 82 29 7
Card 215: 67 60 86 35 17 62 55 27 54 7
Card 216: 18 71  4 89 17 31 63 28 25 2
Card 217: 54 65 75 13 46  8 37 25 95 8
Card 218: 68 97 66 41 88 16 65 31 23 6
Answer: 26443
```

```
Your puzzle answer was 26443.
```

Chapter 5

Evaluation

5.1 Independent feedback - Mr Chiu

5.1.1 Introduction

After completing development, I sat down with Mr Chiu and let him try out the interpreter (see Appendix B for photographic evidence).

He tried out all of the features, including:

- Storage of single-dimensional and multi-dimensional arrays and dictionaries, and access and modification of elements in these structures (*Objectives 3d, 3e, 4a, and 4b*)
- If, else if, and else structures (*Objective 5*)
- While loops, for loops, and break statements (*Objective 6*)
- Functions and returns (*Objective 7*)
- Built-in functions, in particular `input`, `to_number`, and `sort` (*Objective 8*)
- REPL interface and source code file execution (*Objectives 11 and 12*)

Mr Chiu confirmed everything worked as expected. Throughout the session, he remarked that errors were descriptive and appropriate for beginners (*Objective 10a*).

5.1.2 Interview

NAPHAT Do you think this interpreter meets the requirements for a programming language for beginners?

MR CHIU I do, yes.

NAPHAT Can you expand on that a bit more?

MR CHIU Well, I think the way you designed the syntax makes it really easy to understand and learn. The errors were also very descriptive and helpful.

NAPHAT Great, thanks. Can you think of any way this language could be extended further?

MR CHIU I think it's fine as a language for beginners. If you wanted to make this an actually applicable language, then you would obviously need to add object-oriented programming. Also, it's a bit hard to input multi-line stuff like loops and functions in the REPL because you are forced to input it all in one line.

NAPHAT OK, thanks for the feedback. Do you think there are sufficiently many or too many built-in functions?

MR CHIU I think this is completely fine.

NAPHAT One last thing—do you think the executable is simple enough for beginners to run programs?

MR CHIU Oh yeah. It's perfect to just type the name of the binary followed by an optional file name.

NAPHAT That's all, thanks again.

5.1.3 Discussion

This interview confirms that the language works great in regards to the original problem specification as a language for beginners.

Mr Chiu's point about adding object-oriented programming is certainly a good way of extending the project further, especially as many modern languages have built-in *methods* as opposed to *functions*, e.g., in Python, you would do `a.sort()` rather than `sort(a)`. Implementing object-oriented programming would be another challenge altogether. This may make it easier for beginners to transition to other languages, but it's a minor point—I believe the current syntax works fine as a first language.

His second point about wanting the ability to input multi-line structures into the REPL is very reasonable. Ideally, if the user inputs the first line of a function definition, say, the interpreter should prompt for a new line with an indent for the function body. To solve this, there would need to be some communication between the parser and the driver code. If the parser knows that the current 'line' of source code may not be complete, it should somehow communicate with the driver code, perhaps via another 'dummy error' (like `ThrownBreak` and `ThrownReturn`), to prompt for another line. But for now, I believe the REPL is fine for what it should do—to test simple code line-by-line.

5.2 Independent feedback - a beginner programmer

5.2.1 Introduction

Moncef Slimani is a sixth former in my school. Although he knows basic programming principles (variables, loops, etc.), he is not too confident in any particular language. I set him the task of trying to learn this language from some basic examples I had written for him (again, see Appendix B for photographic evidence).

He primarily executed from a source code file, which worked fine (*Objective 12*). He had a lot of fun messing around with loops (*Objective 6*) and functions (*Objective 7*), both of which worked as expected.

5.2.2 Interview

NAPHAT You seem to have caught on quite quickly! Do you find the syntax intuitive as a beginner programmer?

MONCEF Yes, the syntax if very beginner-friendly. I found the functions like `input` and `sort` intuitively named as well.

NAPHAT Were the errors detailed enough to get you back on track?

MONCEF Yeah, they really help pinpoint the source of the error. (*Objective 10a*)

NAPHAT Do you see anything that could be improved?

MONCEF Hmm... yes. Two things: I think the syntax for 'for' loops aren't as intuitive as in Python. I don't like the stuff with the semicolons. And I don't like how I am forced to add a return value when I just want to return, like `return null`.

NAPHAT Thanks for these points. Anything else?

MONCEF Also, in Python you can loop through the elements of a list, not just through the index. That'd be really useful.

NAPHAT Oh I didn't think of that, thanks. That's all I have for you. Thanks again.

5.2.3 Discussion

From the interview, it is clear that the language is effective as an intuitive first language, especially in regard to the syntactical design and error reporting.

Moncef raised three really interesting concerns. First, he didn't like the C-like three-part for statements. These were added because of Mr Chiu's request to have some C-like syntax in the language for students to get familiar with as they transition onto other languages. However, I admit these are not very intuitive for beginners and that they do feel a bit out of place in terms of the syntax of the language.

Essentially, we have two opposing ways of designing for statements: on one hand, Python, Rust, Swift, etc. use the `for ... in ...` pattern which is very intuitive; on the other, we have the C-like three-part statements which are very common (C, C++, Java, JavaScript, PHP, Go, etc.). I would argue that these two philosophies are polar opposites and cannot be 'met in the middle'. Therefore, it is a matter of choosing one or the other. Perhaps it could be said that the correct choice was actually to choose the intuitive design—students

can then go on to learn the more common one later as it is so common anyway; after all, this language is all about teaching beginners programming *principles* rather than actual syntax.

As an example, if I were to change for loops to look like `for i in 0..n` (like in Rust and Swift), new tokens for `in` and `.` would first have to be added. Then, a new grammar rule would have to be considered by the parser:

`<for> ::= For <expression> In <expression> Dot Dot <expression>.`

In terms of the representation in the abstract syntax tree, I believe you can still get away with internally converting these into while loops of the form

```
var i = 0
while (i < n) {
        ...
        i = i + 1
}
```

Execution would then follow as usual after evaluating the expressions in the 'range' of the loop. Obviously you lose the flexibility of the three-part for loops (we are now only restricted to going from one number to another, as opposed to begin completely free in choosing our loop condition and increment), but I believe this is probably sufficient for beginners.

Secondly, Moncef raised the issue about return statements requiring an expression to follow. If the language were to be changed to only capture an *optional* expression after a return statement, it would either have to take whitespace into account or use line delimiters like semicolons—otherwise, it would be impossible to tell the difference between, say, `return f()` and `return` *followed by* `f()`.

If semicolons were used, then it would be simple to check for a semicolon after a return statement—in the case that there is a semicolon there, it would mean that there is no return value; otherwise, we could capture an expression after the `return` using recursive descent. If whitespace were used, then it would perform the same check but with a newline character instead of a semicolon.

Lastly, Moncef raised the point about iterating through the elements of arrays and dictionaries, rather than just, say, the index. This would not be as simple as desugaring for loops into while loops. In fact, I think it would actually require storing for loop structures in the abstract syntax tree. In the execution stage, the interpreter can simply iterate through the elements after checking that the target value is an array or dictionary.

5.3 Further potential points of improvement

A point raised by neither Mr Chiu nor Moncef was the speed of the interpreter. When I was solving Advent of Code problems with this language, I found that programs that should take less than a second to run on any modern interpreter (e.g., Python) ended up taking several seconds to run. This clearly would not scale well with real-world applications.

However, optimisations will necessitate sophisticated techniques that are beyond the scope of the A-Level. For example, whether or not the condition of a loop is dependent on variables can be analysed; the interpreter may not need to evaluate it repeatedly. Moreover, the fact that neither Mr Chiu nor Moncef noticed this implies that it should not be a problem in the classroom (Mr Chiu even remarked that the REPL was 'nice and fast'). So, for what it was designed for, the speed of the interpreter should not be an issue.

An algorithmic issue I ran into during the design process was the design of a constant time hashing algorithm for dictionaries (i.e., when dictionaries are used as keys for other dictionaries) (see Subsection 2.5.4). I decided simply to not allow dictionaries to be used as keys because I guessed that the algorithmic complexity to implement this would be much beyond the scope of the A-Level.

If I had to come up with a solution to this problem, I can think of two potential ones. The first solution might be to maintain a 'running hash' as the dictionary is modified, i.e., adding, removing, and modifying entries would immediately change the hash, which would be kept as an attribute in the `HashTable` class. This would probably involve either some sophisticated mathematics to come up with an 'invertible' function to be used with entries are added and removed, or the representation of the entries in the table as a binary segment tree-like structure. The tree structure would then update the 'total' hash by propagating hashes calculated from each node using its children nodes from the leaves up, which can be done in $O(\log n)$ time (the height of the tree). This obviously slows down dictionary updates. Adding/removing entire branches of the tree may also be really expensive.

Another solution might be to maintain a 'ordered set' of entries—after all, the problem is that the buckets may not store the entries in the same order. A priority queue could be used with a Fibonacci heap implementation to maintain this order in sub-linear time. Note that now it is possible to compare two dictionaries to maintain this order, as both dictionaries will have its own 'ordered set' of elements, of which we can compare

the first k to do this in constant time. This solution will use more memory, and may not be practical for large dictionaries (we are essentially storing two copies of the same dictionary).

5.4 Evaluation against original problem description

To quote the original problem description (Subsection 1.1), Mr Chiu

> has asked me to develop a programming language that could be used to teach beginners. He would like the language to be as minimal as possible, i.e., only containing the basic components of a standard programming language. Ideally, this language should: 1) teach core programming concepts to students which can be applied to other languages, and 2) force students to understand programming more fundamentally, as little would be hidden beneath syntactic sugar.

The problem description along with other sources had inspired the following design philosophy for the language (Subsection 1.5), which I will evaluate point-by-point:

1. **The language should have minimal syntactic sugar.**

   I believe this is achieved. The only real syntactic sugar in the language is for loops, but even that is standard across most, if not all, modern programming languages. Other than that, users are forced to write out everything manually.

2. **The language should have as few built-in functions as possible.**

   As supported by Mr Chiu's, this is achieved. Moreover, when I was solving Advent of Code problems, I found that I had to implement functions myself that I would normally take for granted, most notably the string `split` and `slice` functions. It was weirdly satisfying to implement everything myself.

3. **Programs written in the language should not require boilerplate code.**

   Source code written in this language requires 0 boilerplate code to run.

4. **Syntax should be simple and intuitive, but should also contain some C-like elements.**

   In terms of the simplicity and intuitiveness, I believe I have achieved this, as supported by Mr Chiu's and Moncef's interviews. I also achieved the requirement of having some C-like elements in the form of for loops. However, as discussed extensively in Subsection 5.2.3, I believe it can be argued that using C-like for loops contradicted with making the language as intuitive as possible, and that perhaps a more intuitive for loop syntax would be more in-line with the general philosophy of the language.

   However, this discussion is to do with the setting of the language requirements; in terms of whether or not I achieved the set requirements, I certainly did.

5. **Errors should be strict and not allow any undefined behaviour (e.g., cross-type operations, truthiness, etc.); error reports should be as descriptive as possible.**

   Throughout the design process, I have made errors strict (e.g., not allowing *any* implicit conversions in binary expressions like `5 + true`) and error messages descriptive (e.g., line numbers; errors for each potential mistake in for statements: no semicolon after initialiser, no semicolon after condition, etc.).

   Also taking Moncef's interview into account, I would say this has been achieved.

5.5 Evaluation against objectives

In the Testing chapter, I thoroughly tested *all* objectives and showed that they were all achieved. Furthermore, the Advent of Code solutions have used many, if not all, of the functionality set by the objectives (loops, dictionaries, functions, etc.), which further suggests that the interpreter works as expected.

Moreover, variables, loops, functions, and built-in functions were tested extensively by Mr Chiu and Moncef, which covers Objectives 3, 4, 5, 6, 7, 8, and 9. They both used the REPL interface and executed source code from files, which covers Objectives 11 and 12. They also encountered errors in their testing; both Mr Chiu and Moncef provided positive feedback regarding error messages. This covers Objective 10.

Therefore, it is safe to say *all* objectives have been achieved.

5.6 Conclusion

In conclusion, I would say the project was a success: all objectives were met and the product is something that satisfactorily solves the original problem.

Appendix A

Unit tests

A.1 Tests for the `Tokenizer` class

```
#[cfg(test)]
mod tests {
  use crate::{token::{Token, TokenType, Literal}, error::ErrorType};

  use super::Tokenizer;

  fn tokenize(source: &str) -> Result<Vec<Token>, ErrorType> {
    let mut tokenizer = Tokenizer::new(source);
    tokenizer.tokenize()
  }

  #[test]
  fn one_char_tokens() {
    let source = "( ) { } [ ] : , - % + ; / *";
    assert_eq!(Ok(vec![
    Token { type_: TokenType::LeftParen, lexeme: String::from("("), literal: Literal::Null, line: 1 },
    Token { type_: TokenType::RightParen, lexeme: String::from(")"), literal: Literal::Null, line: 1 },
    Token { type_: TokenType::LeftCurly, lexeme: String::from("{"), literal: Literal::Null, line: 1 },
      Token { type_: TokenType::RightCurly, lexeme: String::from("}"), literal: Literal::Null, line: 1 },
    Token { type_: TokenType::LeftSquare, lexeme: String::from("["), literal: Literal::Null, line: 1 },
    Token { type_: TokenType::RightSquare, lexeme: String::from("]"), literal: Literal::Null, line: 1 },
    Token { type_: TokenType::Colon, lexeme: String::from(":"), literal: Literal::Null, line: 1 },
    Token { type_: TokenType::Comma, lexeme: String::from(","), literal: Literal::Null, line: 1 },
    Token { type_: TokenType::Minus, lexeme: String::from("-"), literal: Literal::Null, line: 1 },
    Token { type_: TokenType::Percent, lexeme: String::from("%"), literal: Literal::Null, line: 1 },
    Token { type_: TokenType::Plus, lexeme: String::from("+"), literal: Literal::Null, line: 1 },
    Token { type_: TokenType::Semicolon, lexeme: String::from(";"), literal: Literal::Null, line: 1 },
    Token { type_: TokenType::Slash, lexeme: String::from("/"), literal: Literal::Null, line: 1 },
    Token { type_: TokenType::Star, lexeme: String::from("*"), literal: Literal::Null, line: 1 },
```

```
30        Token { type_: TokenType::Eof, lexeme: String::from(""), literal: Literal::Null, line:
          ↪  1 },
31        ]), tokenize(source));
32      }
33
34      #[test]
35      fn one_two_char_tokens() {
36        let source = "! != = == > >= < <=";
37        assert_eq!(Ok(vec![
38        Token { type_: TokenType::Bang, lexeme: String::from("!"), literal: Literal::Null,
          ↪  line: 1 },
39        Token { type_: TokenType::BangEqual, lexeme: String::from("!="), literal:
          ↪  Literal::Null, line: 1 },
40        Token { type_: TokenType::Equal, lexeme: String::from("="), literal: Literal::Null,
          ↪  line: 1 },
41        Token { type_: TokenType::EqualEqual, lexeme: String::from("=="), literal:
          ↪  Literal::Null, line: 1 },
42        Token { type_: TokenType::Greater, lexeme: String::from(">"), literal: Literal::Null,
          ↪  line: 1 },
43        Token { type_: TokenType::GreaterEqual, lexeme: String::from(">="), literal:
          ↪  Literal::Null, line: 1 },
44        Token { type_: TokenType::Less, lexeme: String::from("<"), literal: Literal::Null,
          ↪  line: 1 },
45        Token { type_: TokenType::LessEqual, lexeme: String::from("<="), literal:
          ↪  Literal::Null, line: 1 },
46        Token { type_: TokenType::Eof, lexeme: String::from(""), literal: Literal::Null, line:
          ↪  1 },
47        ]), tokenize(source));
48      }
49
50      #[test]
51      fn literals() {
52        let source = "\"abc\" 123 \"abc123\" 123.5 \"\" 123abc 5.5";
53        assert_eq!(Ok(vec![
54        Token { type_: TokenType::String_, lexeme: String::from("\"abc\""), literal:
          ↪  Literal::String_(String::from("abc")), line: 1 },
55        Token { type_: TokenType::Number, lexeme: String::from("123"), literal:
          ↪  Literal::Number(123.0), line: 1 },
56        Token { type_: TokenType::String_, lexeme: String::from("\"abc123\""), literal:
          ↪  Literal::String_(String::from("abc123")), line: 1 },
57        Token { type_: TokenType::Number, lexeme: String::from("123.5"), literal:
          ↪  Literal::Number(123.5), line: 1 },
58        Token { type_: TokenType::String_, lexeme: String::from("\"\""), literal:
          ↪  Literal::String_(String::from("")), line: 1 },
59        Token { type_: TokenType::Number, lexeme: String::from("123"), literal:
          ↪  Literal::Number(123.0), line: 1 },
60        Token { type_: TokenType::Identifier, lexeme: String::from("abc"), literal:
          ↪  Literal::Null, line: 1 },
61        Token { type_: TokenType::Number, lexeme: String::from("5.5"), literal:
          ↪  Literal::Number(5.5), line: 1 },
62        Token { type_: TokenType::Eof, lexeme: String::from(""), literal: Literal::Null, line:
          ↪  1 },
63        ]), tokenize(source));
64      }
65
66      #[test]
67      fn line_count() {
68        let source = "12\n23";
69        assert_eq!(Ok(vec![
70        Token { type_: TokenType::Number, lexeme: String::from("12"), literal:
          ↪  Literal::Number(12.0), line: 1 },
```

```
71        Token { type_: TokenType::Number, lexeme: String::from("23"), literal:
          ↪  Literal::Number(23.0), line: 2 },
72        Token { type_: TokenType::Eof, lexeme: String::from(""), literal: Literal::Null, line:
          ↪  2 },
73        ]), tokenize(source));
74      }
75
76      #[test]
77      fn unterminated_string() {
78        let source = "\"abc\nabc\nabc";
79        assert_eq!(Err(ErrorType::UnterminatedString), tokenize(source));
80      }
81
82      #[test]
83      fn identifiers_and_keywords() {
84        let source = "a a2 if and or ifandor";
85        assert_eq!(Ok(vec![
86        Token { type_: TokenType::Identifier, lexeme: String::from("a"), literal:
          ↪  Literal::Null, line: 1 },
87        Token { type_: TokenType::Identifier, lexeme: String::from("a2"), literal:
          ↪  Literal::Null, line: 1 },
88        Token { type_: TokenType::If, lexeme: String::from("if"), literal: Literal::Null,
          ↪  line: 1 },
89        Token { type_: TokenType::And, lexeme: String::from("and"), literal: Literal::Null,
          ↪  line: 1 },
90        Token { type_: TokenType::Or, lexeme: String::from("or"), literal: Literal::Null,
          ↪  line: 1 },
91        Token { type_: TokenType::Identifier, lexeme: String::from("ifandor"), literal:
          ↪  Literal::Null, line: 1 },
92        Token { type_: TokenType::Eof, lexeme: String::from(""), literal: Literal::Null, line:
          ↪  1 },
93        ]), tokenize(source));
94      }
95
96      #[test]
97      fn comments() {
98        let source = "1\n#abc\n#abc\n1";
99        assert_eq!(Ok(vec![
100       Token { type_: TokenType::Number, lexeme: String::from("1"), literal:
          ↪  Literal::Number(1.0), line: 1 },
101       Token { type_: TokenType::Number, lexeme: String::from("1"), literal:
          ↪  Literal::Number(1.0), line: 4 },
102       Token { type_: TokenType::Eof, lexeme: String::from(""), literal: Literal::Null, line:
          ↪  4 },
103       ]), tokenize(source));
104     }
105   }
```

A.2 Tests for the `Parser` class

```
1   #[cfg(test)]
2   mod tests {
3     use crate::{token, expr::{Expr, ExprType}, error::ErrorType, tokenizer::Tokenizer,
      ↪  stmt::Stmt, stmt::StmtType};
4
5     use super::Parser;
6
7     fn parse(source: &str) -> Result<Vec<Stmt>, Vec<ErrorType>> {
8       let mut tokenizer = Tokenizer::new(source);
9       let tokens = tokenizer.tokenize().expect("Tokenizer returned error.");
```

```
10      let mut parser = Parser::new(tokens);
11      parser.parse()
12    }
13
14    fn errors_in_result(result: Result<Vec<Stmt>, Vec<ErrorType>>, errors: Vec<ErrorType>) ->
      ↪  bool {
15      let Err(result_errors) = result else {
16        return false;
17      };
18      for error in errors {
19        if !result_errors.contains(&error) {
20          return false;
21        }
22      }
23      true
24    }
25
26    #[test]
27    fn for_() {
28      let source = "for (var x = 5; x < 10; x = x + 1) {var y = x}";
29      assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Block {
30          body: vec![
31          Stmt { line: 1, stmt_type: StmtType::VarDecl {
32              name: String::from("x"),
33              value: Expr { line: 1, expr_type: ExprType::Literal { value:
                ↪  token::Literal::Number(5.0) }},
34          }},
35          Stmt { line: 1, stmt_type: StmtType::While {
36              condition: Expr { line: 1, expr_type: ExprType::Binary {
37                  left: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                    ↪  String::from("x") }}),
38                  operator: token::Token { type_: token::TokenType::Less, lexeme:
                    ↪  String::from("<"), literal: token::Literal::Null, line: 1 },
39                  right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪  token::Literal::Number(10.0) }}),
40              }},
41              body: Box::new(Stmt { line: 1, stmt_type: StmtType::Block {
42                  body: vec![
43                  Stmt { line: 1, stmt_type: StmtType::Block {
44                      body: vec![
45                      Stmt { line: 1, stmt_type: StmtType::VarDecl {
46                          name: String::from("y"),
47                          value: Expr { line: 1, expr_type: ExprType::Variable { name:
                            ↪  String::from("x") }},
48                      }},
49                      ],
50                  }},
51                  Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr { line:
                    ↪  1, expr_type: ExprType::Assignment {
52                          target: Box::new(Expr { line: 1, expr_type: ExprType::Variable {
                            ↪  name: String::from("x") }}),
53                          value: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
54                              left: Box::new(Expr { line: 1, expr_type: ExprType::Variable {
                                ↪  name: String::from("x") }}),
55                              operator: token::Token { type_: token::TokenType::Plus, lexeme:
                                ↪  String::from("+"), literal: token::Literal::Null, line: 1 },
56                              right: Box::new(Expr { line: 1, expr_type: ExprType::Literal {
                                ↪  value: token::Literal::Number(1.0) }}),
57                          }}),
58                  }}}},
59                  ],
```

```
60              }}),
61          }},
62          ]
63      }}]), parse(source));
64  }
65
66  #[test]
67  fn for_no_init() {
68      let source = "for (; x < 10; x = x + 1) {var y = x}";
69      assert_eq!(Ok(vec![
70      Stmt { line: 1, stmt_type: StmtType::While {
71          condition: Expr { line: 1, expr_type: ExprType::Binary {
72              left: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                ↪ String::from("x") }}),
73              operator: token::Token { type_: token::TokenType::Less, lexeme:
                ↪ String::from("<"), literal: token::Literal::Null, line: 1 },
74              right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                ↪ token::Literal::Number(10.0) }}),
75          }},
76          body: Box::new(Stmt { line: 1, stmt_type: StmtType::Block {
77              body: vec![
78              Stmt { line: 1, stmt_type: StmtType::Block {
79                  body: vec![
80                  Stmt { line: 1, stmt_type: StmtType::VarDecl {
81                      name: String::from("y"),
82                      value: Expr { line: 1, expr_type: ExprType::Variable { name:
                        ↪ String::from("x") }},
83                  }},
84                  ],
85              }},
86              Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr { line: 1,
                ↪ expr_type: ExprType::Assignment {
87                      target: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                        ↪ String::from("x") }}),
88                      value: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
89                          left: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                            ↪ String::from("x") }}),
90                          operator: token::Token { type_: token::TokenType::Plus, lexeme:
                            ↪ String::from("+"), literal: token::Literal::Null, line: 1 },
91                          right: Box::new(Expr { line: 1, expr_type: ExprType::Literal {
                            ↪ value: token::Literal::Number(1.0) }}),
92                      }}),
93              }}}},
94              ],
95          }}),
96      }},
97      ]), parse(source));
98  }
99
100 #[test]
101 fn for_no_cond() {
102     let source = "for (var x = 5;; x = x + 1) {var y = x}";
103     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Block {
104         body: vec![
105         Stmt { line: 1, stmt_type: StmtType::VarDecl {
106             name: String::from("x"),
107             value: Expr { line: 1, expr_type: ExprType::Literal { value:
                ↪ token::Literal::Number(5.0) }},
108         }},
109         Stmt { line: 1, stmt_type: StmtType::While {
```

```
110             condition: Expr { line: 1, expr_type: ExprType::Literal { value:
                ↪ token::Literal::Bool(true) }},
111             body: Box::new(Stmt { line: 1, stmt_type: StmtType::Block {
112                 body: vec![
113                 Stmt { line: 1, stmt_type: StmtType::Block {
114                     body: vec![
115                     Stmt { line: 1, stmt_type: StmtType::VarDecl {
116                         name: String::from("y"),
117                         value: Expr { line: 1, expr_type: ExprType::Variable { name:
                        ↪ String::from("x") }},
118                     }},
119                     ],
120                 }},
121                 Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr { line:
                ↪ 1, expr_type: ExprType::Assignment {
122                         target: Box::new(Expr { line: 1, expr_type: ExprType::Variable {
                        ↪ name: String::from("x") }}),
123                         value: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
124                             left: Box::new(Expr { line: 1, expr_type: ExprType::Variable {
                            ↪ name: String::from("x") }}),
125                             operator: token::Token { type_: token::TokenType::Plus, lexeme:
                            ↪ String::from("+"), literal: token::Literal::Null, line: 1 },
126                             right: Box::new(Expr { line: 1, expr_type: ExprType::Literal {
                            ↪ value: token::Literal::Number(1.0) }}),
127                         }}),
128                 }}}},
129                 ],
130             }}),
131         }},
132         ]
133     }}]), parse(source));
134 }
135
136 #[test]
137 fn for_no_inc() {
138     let source = "for (var x = 5; x < 10;) {var y = x}";
139     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Block {
140         body: vec![
141         Stmt { line: 1, stmt_type: StmtType::VarDecl {
142             name: String::from("x"),
143             value: Expr { line: 1, expr_type: ExprType::Literal { value:
                ↪ token::Literal::Number(5.0) }},
144         }},
145         Stmt { line: 1, stmt_type: StmtType::While {
146             condition: Expr { line: 1, expr_type: ExprType::Binary {
147                 left: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                ↪ String::from("x") }}),
148                 operator: token::Token { type_: token::TokenType::Less, lexeme:
                ↪ String::from("<"), literal: token::Literal::Null, line: 1 },
149                 right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                ↪ token::Literal::Number(10.0) }}),
150             }},
151             body: Box::new(Stmt { line: 1, stmt_type: StmtType::Block {
152                 body: vec![
153                 Stmt { line: 1, stmt_type: StmtType::Block {
154                     body: vec![
155                     Stmt { line: 1, stmt_type: StmtType::VarDecl {
156                         name: String::from("y"),
157                         value: Expr { line: 1, expr_type: ExprType::Variable { name:
                        ↪ String::from("x") }},
158                     }},
```

```
                    ],
                }},
                ],
            }}),
        }},
        ]
    }}]), parse(source));
}

#[test]
fn for_no_init_semicolon() {
    let source = "for (var x = 5 x < 10; x = x + 1) {var y = x}";
    assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedSemicolonAfterInit { line: 1 }]));
}

#[test]
fn for_no_cond_semicolon() {
    let source = "for (var x = 5; x < 10 x = x + 1) {var y = x}";
    assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedSemicolonAfterCondition { line: 1 }]));
}

#[test]
fn unclosed_for() {
    let source = "for (var x = 5; x < 10; x = x + 1 {var y = x}";
    assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedParenAfterIncrement { line: 1 }]));
}

#[test]
fn unopened_block() {
    let source = "for (var x = 5; x < 10; x = x + 1) var y = x}";
assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedCharacter { expected: '{', line: 1 }]));
}

#[test]
fn unclosed_block() {
    let source = "for (var x = 5; x < 10; x = x + 1) {var y = x";
    assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedCharacter { expected: '}', line: 1 }]));
}

#[test]
fn func() {
    let source = "func hello(a, b) {print a print b}";
    assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Function {
        name: String::from("hello"),
        parameters: vec![String::from("a"), String::from("b")],
        body: Box::new(Stmt { line: 1, stmt_type: StmtType::Block { body: vec![
            Stmt { line: 1, stmt_type: StmtType::Print { expression: Expr { line: 1, expr_type: ExprType::Variable { name: String::from("a") }}}},
            Stmt { line: 1, stmt_type: StmtType::Print { expression: Expr { line: 1, expr_type: ExprType::Variable { name: String::from("b") }}}},
            ]}}),
    }}]), parse(source));
}

#[test]
fn func_keyword_name() {
```

```
213     let source = "func print(a, b) {print a print b}";
214     assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedFunctionName { line: 1
        ↪ }]));
215   }
216
217   #[test]
218   fn if_() {
219     let source = "if (a == 2) {print a}";
220     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::If {
221         condition: Expr { line: 1, expr_type: ExprType::Binary {
222             left: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                ↪ String::from("a") }}),
223             operator: token::Token { type_: token::TokenType::EqualEqual, lexeme:
                ↪ String::from("=="), literal: token::Literal::Null, line: 1 },
224             right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                ↪ token::Literal::Number(2.0) }}),
225         }},
226         then_body: Box::new(Stmt { line: 1, stmt_type: StmtType::Block { body: vec![Stmt {
            ↪ line: 1, stmt_type: StmtType::Print { expression: Expr { line: 1, expr_type:
            ↪ ExprType::Variable { name: String::from("a") } }}}] }}),
227         else_body: None,
228     }}]), parse(source));
229   }
230
231   #[test]
232   fn else_if() {
233     let source = "if (a == 2) {print a} else if (a == 3) {print b} else if (a == 4) {print
        ↪ c}";
234     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::If {
235         condition: Expr { line: 1, expr_type: ExprType::Binary {
236             left: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                ↪ String::from("a") }}),
237             operator: token::Token { type_: token::TokenType::EqualEqual, lexeme:
                ↪ String::from("=="), literal: token::Literal::Null, line: 1 },
238             right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                ↪ token::Literal::Number(2.0) }}),
239         }},
240         then_body: Box::new(Stmt { line: 1, stmt_type: StmtType::Block { body: vec![Stmt {
            ↪ line: 1, stmt_type: StmtType::Print { expression: Expr { line: 1, expr_type:
            ↪ ExprType::Variable { name: String::from("a") } }}}] }}),
241         else_body: Some(Box::new(
242         Stmt { line: 1, stmt_type: StmtType::If {
243             condition: Expr { line: 1, expr_type: ExprType::Binary {
244                 left: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                    ↪ String::from("a") }}),
245                 operator: token::Token { type_: token::TokenType::EqualEqual, lexeme:
                    ↪ String::from("=="), literal: token::Literal::Null, line: 1 },
246                 right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪ token::Literal::Number(3.0) }}),
247             }},
248             then_body: Box::new(Stmt { line: 1, stmt_type: StmtType::Block { body: vec![Stmt
                ↪ { line: 1, stmt_type: StmtType::Print { expression: Expr { line: 1,
                ↪ expr_type: ExprType::Variable { name: String::from("b") } }}}]} }),
249             else_body: Some(Box::new(
250             Stmt { line: 1, stmt_type: StmtType::If {
251                 condition: Expr { line: 1, expr_type: ExprType::Binary {
252                     left: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                        ↪ String::from("a") }}),
253                     operator: token::Token { type_: token::TokenType::EqualEqual, lexeme:
                        ↪ String::from("=="), literal: token::Literal::Null, line: 1 },
```

```
254                     right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                            ↪ token::Literal::Number(4.0) }}),
255                 }},
256                 then_body: Box::new(Stmt { line: 1, stmt_type: StmtType::Block { body:
                    ↪ vec![Stmt { line: 1, stmt_type: StmtType::Print { expression: Expr {
                    ↪ line: 1, expr_type: ExprType::Variable { name: String::from("c") } }}}]}
                    ↪ }),
257                 else_body: None,
258             }}
259             )),
260         }}
261         )),
262     }}]), parse(source));
263 }
264
265 #[test]
266 fn else_() {
267     let source = "if (a == 2) {print a} else {print b}";
268     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::If {
269         condition: Expr { line: 1, expr_type: ExprType::Binary {
270             left: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                    ↪ String::from("a") }}),
271             operator: token::Token { type_: token::TokenType::EqualEqual, lexeme:
                    ↪ String::from("=="), literal: token::Literal::Null, line: 1 },
272             right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪ token::Literal::Number(2.0) }}),
273         }},
274         then_body: Box::new(Stmt { line: 1, stmt_type: StmtType::Block { body: vec![Stmt {
            ↪ line: 1, stmt_type: StmtType::Print { expression: Expr { line: 1, expr_type:
            ↪ ExprType::Variable { name: String::from("a") } }}}]} }),
275         else_body: Some(Box::new(Stmt { line: 1, stmt_type: StmtType::Block { body:
            ↪ vec![Stmt { line: 1, stmt_type: StmtType::Print { expression: Expr { line: 1,
            ↪ expr_type: ExprType::Variable { name: String::from("b") } }}}]} })),
276     }}]), parse(source));
277 }
278
279 #[test]
280 fn print() {
281     let source = "print 5*1+2*(3-4/a)";
282     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Print { expression: Expr { line:
        ↪ 1, expr_type: ExprType::Binary {
283             left: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
284                 left: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                        ↪ token::Literal::Number(5.0) }}),
285                 operator: token::Token { type_: token::TokenType::Star, lexeme:
                        ↪ String::from("*"), literal: token::Literal::Null, line: 1 },
286                 right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                        ↪ token::Literal::Number(1.0) }}),
287             }}),
288             operator: token::Token { type_: token::TokenType::Plus, lexeme:
                    ↪ String::from("+"), literal: token::Literal::Null, line: 1 },
289             right: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
290                 left: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                        ↪ token::Literal::Number(2.0) }}),
291                 operator: token::Token { type_: token::TokenType::Star, lexeme:
                        ↪ String::from("*"), literal: token::Literal::Null, line: 1 },
292                 right: Box::new(Expr { line: 1, expr_type: ExprType::Grouping {
293                     expression: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
294                         left: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                                ↪ token::Literal::Number(3.0) }}),
```

```
295                         operator: token::Token { type_: token::TokenType::Minus, lexeme:
                            ↪ String::from("-"), literal: token::Literal::Null, line: 1 },
296                         right: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
297                             left: Box::new(Expr { line: 1, expr_type: ExprType::Literal {
                                ↪ value: token::Literal::Number(4.0) }}),
298                             operator: token::Token { type_: token::TokenType::Slash, lexeme:
                                ↪ String::from("/"), literal: token::Literal::Null, line: 1 },
299                             right: Box::new(Expr { line: 1, expr_type: ExprType::Variable {
                                ↪ name: String::from("a") }}),
300                         }}),
301                     }}),
302                 }}),
303             }}),
304       }}}}]), parse(source));
305   }
306
307   #[test]
308   fn var() {
309       let source = "var a = 5";
310       assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::VarDecl { name:
          ↪ String::from("a"), value: Expr { line: 1, expr_type: ExprType::Literal { value:
          ↪ token::Literal::Number(5.0) } }}}]), parse(source));
311   }
312
313   #[test]
314   fn invalid_var_name() {
315       let source = "var 123 = 5";
316       assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedVariableName { line: 1
          ↪ }]));
317   }
318
319   #[test]
320   fn while_() {
321       let source = "while (a == 2) {print b}";
322       assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::While {
323           condition: Expr { line: 1, expr_type: ExprType::Binary {
324               left: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                  ↪ String::from("a") }}),
325               operator: token::Token { type_: token::TokenType::EqualEqual, lexeme:
                  ↪ String::from("=="), literal: token::Literal::Null, line: 1 },
326               right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                  ↪ token::Literal::Number(2.0) }}),
327           }},
328           body: Box::new(Stmt { line: 1, stmt_type: StmtType::Block { body: vec![Stmt { line:
              ↪ 1, stmt_type: StmtType::Print { expression: Expr { line: 1, expr_type:
              ↪ ExprType::Variable { name: String::from("b") } }}}]} }),
329       }}]), parse(source));
330   }
331
332   #[test]
333   fn multiple_statements() {
334       let source = "print a if (a == 2) {print a} else {print b} var c = 3";
335       assert_eq!(Ok(vec![
336       Stmt { line: 1, stmt_type: StmtType::Print { expression: Expr { line: 1, expr_type:
          ↪ ExprType::Variable { name: String::from("a") } } } },
337       Stmt { line: 1, stmt_type: StmtType::If {
338           condition: Expr { line: 1, expr_type: ExprType::Binary {
339               left: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                  ↪ String::from("a") }}),
340               operator: token::Token { type_: token::TokenType::EqualEqual, lexeme:
                  ↪ String::from("=="), literal: token::Literal::Null, line: 1 },
```

```
341             right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                ↪  token::Literal::Number(2.0) }}),
342         }},
343         then_body: Box::new(Stmt { line: 1, stmt_type: StmtType::Block { body: vec![Stmt {
                ↪  line: 1, stmt_type: StmtType::Print { expression: Expr { line: 1, expr_type:
                ↪  ExprType::Variable { name: String::from("a") } }}}]} }),
344         else_body: Some(Box::new(Stmt { line: 1, stmt_type: StmtType::Block { body:
                ↪  vec![Stmt { line: 1, stmt_type: StmtType::Print { expression: Expr { line: 1,
                ↪  expr_type: ExprType::Variable { name: String::from("b") } }}}]} })),
345     }},
346     Stmt { line: 1, stmt_type: StmtType::VarDecl { name: String::from("c"), value: Expr {
        ↪  line: 1, expr_type: ExprType::Literal { value: token::Literal::Number(3.0) } } } },
347     ]), parse(source));
348 }
349
350 #[test]
351 fn bidmas() {
352     let source = "5*1+2*(3-4/a)";
353     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr {
        ↪  line: 1, expr_type: ExprType::Binary {
354             left: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
355                 left: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪  token::Literal::Number(5.0) }}),
356                 operator: token::Token { type_: token::TokenType::Star, lexeme:
                    ↪  String::from("*"), literal: token::Literal::Null, line: 1 },
357                 right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪  token::Literal::Number(1.0) }}),
358             }}),
359             operator: token::Token { type_: token::TokenType::Plus, lexeme:
                ↪  String::from("+"), literal: token::Literal::Null, line: 1 },
360             right: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
361                 left: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪  token::Literal::Number(2.0) }}),
362                 operator: token::Token { type_: token::TokenType::Star, lexeme:
                    ↪  String::from("*"), literal: token::Literal::Null, line: 1 },
363                 right: Box::new(Expr { line: 1, expr_type: ExprType::Grouping {
364                     expression: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
365                         left: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                            ↪  token::Literal::Number(3.0) }}),
366                         operator: token::Token { type_: token::TokenType::Minus, lexeme:
                            ↪  String::from("-"), literal: token::Literal::Null, line: 1 },
367                         right: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
368                             left: Box::new(Expr { line: 1, expr_type: ExprType::Literal {
                                ↪  value: token::Literal::Number(4.0) }}),
369                             operator: token::Token { type_: token::TokenType::Slash, lexeme:
                                ↪  String::from("/"), literal: token::Literal::Null, line: 1 },
370                             right: Box::new(Expr { line: 1, expr_type: ExprType::Variable {
                                ↪  name: String::from("a") }}),
371                         }}),
372                     }}),
373                 }}),
374             }}),
375     }}}}]), parse(source));
376 }
377
378 #[test]
379 fn logic() {
380     let source = "true and true or false and true or false";
381     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr {
        ↪  line: 1, expr_type: ExprType::Binary {
382             left: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
```

```
383                 left: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
384                     left: Box::new(Expr { line: 1, expr_type: ExprType::Literal {value:
                        ↪ token::Literal::Bool(true) }}),
385                     operator: token::Token { type_: token::TokenType::And, lexeme:
                        ↪ String::from("and"), literal: token::Literal::Null, line: 1 },
386                     right: Box::new(Expr { line: 1, expr_type: ExprType::Literal {value:
                        ↪ token::Literal::Bool(true) }}),
387                 }}),
388                 operator: token::Token { type_: token::TokenType::Or, lexeme:
                    ↪ String::from("or"), literal: token::Literal::Null, line: 1 },
389                 right: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
390                     left: Box::new(Expr { line: 1, expr_type: ExprType::Literal {value:
                        ↪ token::Literal::Bool(false) }}),
391                     operator: token::Token { type_: token::TokenType::And, lexeme:
                        ↪ String::from("and"), literal: token::Literal::Null, line: 1 },
392                     right: Box::new(Expr { line: 1, expr_type: ExprType::Literal {value:
                        ↪ token::Literal::Bool(true) }}),
393                 }}),
394             }}),
395             operator: token::Token { type_: token::TokenType::Or, lexeme:
                ↪ String::from("or"), literal: token::Literal::Null, line: 1 },
396             right: Box::new(Expr { line: 1, expr_type: ExprType::Literal {value:
                ↪ token::Literal::Bool(false) }}),
397     }}}}]), parse(source));
398 }
399
400 #[test]
401 fn array() {
402     let source = "[[5, a, b], 3+1, \"g\"]";
403     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr {
        ↪ line: 1, expr_type: ExprType::Array {
404             elements: vec![
405             Expr { line: 1, expr_type: ExprType::Array {
406                 elements: vec![
407                 Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪ token::Literal::Number(5.0) }},
408                 Expr { line: 1, expr_type: ExprType::Variable { name: String::from("a") }},
409                 Expr { line: 1, expr_type: ExprType::Variable { name: String::from("b") }},
410                 ]
411             }},
412             Expr { line: 1, expr_type: ExprType::Binary {
413                 left: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪ token::Literal::Number(3.0) }}),
414                 operator: token::Token { type_: token::TokenType::Plus, lexeme:
                    ↪ String::from("+"), literal: token::Literal::Null, line: 1 },
415                 right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪ token::Literal::Number(1.0) }}),
416             }},
417             Expr { line: 1, expr_type: ExprType::Literal { value:
                ↪ token::Literal::String_(String::from("g")) }},
418             ]
419     }}}}]), parse(source));
420 }
421
422 #[test]
423 fn empty_array() {
424     let source = "[]";
425     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr {
        ↪ line: 1, expr_type: ExprType::Array {elements: vec![] }}}}]), parse(source));
426 }
427
```

```
428   #[test]
429   fn unclosed_array() {
430     let source = "[[5, a, b], 3+1, \"g\"";
431     assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedCharacter { expected:
        ↪  ']', line: 1 }]));
432     let source = "[[5, a, b, 3+1, \"g\"]";
433     assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedCharacter { expected:
        ↪  ']', line: 1 }]));
434   }
435
436   #[test]
437   fn error_line_numbers() {
438     let source = "\n[[5, a, b, 3+1, \"g\"]";
439     assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedCharacter { expected:
        ↪  ']', line: 2 }]));
440     let source = "\n\n[[5, a, b, 3+1, \"g\"]";
441     assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedCharacter { expected:
        ↪  ']', line: 3 }]));
442   }
443
444   #[test]
445   fn unclosed_grouping() {
446     let source = "(5 + 5";
447     assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedCharacter { expected:
        ↪  ')', line: 1 }]));
448   }
449
450   #[test]
451   fn element() {
452     let source = "a[5]";
453     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr {
        ↪  line: 1, expr_type: ExprType::Element {
454             array: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                ↪  String::from("a") }}),
455             index: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                ↪  token::Literal::Number(5.0) } }),
456     }}}}]), parse(source));
457   }
458
459   #[test]
460   fn element_2d() {
461     let source = "a[1][2]";
462     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr {
        ↪  line: 1, expr_type: ExprType::Element {
463             array: Box::new(Expr { line: 1, expr_type: ExprType::Element {
464                 array: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                    ↪  String::from("a") }}),
465                 index: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪  token::Literal::Number(1.0) } }),
466             }}),
467             index: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                ↪  token::Literal::Number(2.0) } }),
468     }}}}]), parse(source));
469   }
470
471   #[test]
472   fn comparison() {
473     let source = "1 < 2 == 3 > 4 <= 5 >= 6 != 7";
474     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr {
        ↪  line: 1, expr_type: ExprType::Binary {
475             left: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
```

```
476                 left: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
477                     left: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                        ↪ token::Literal::Number(1.0) }}),
478                     operator: token::Token { type_: token::TokenType::Less, lexeme:
                        ↪ String::from("<"), literal: token::Literal::Null, line: 1 },
479                     right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                        ↪ token::Literal::Number(2.0) }}),
480                 }}),
481                 operator: token::Token { type_: token::TokenType::EqualEqual, lexeme:
                    ↪ String::from("=="), literal: token::Literal::Null, line: 1 },
482                 right: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
483                     left: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
484                         left: Box::new(Expr { line: 1, expr_type: ExprType::Binary {
485                             left: Box::new(Expr { line: 1, expr_type: ExprType::Literal {
                                ↪ value: token::Literal::Number(3.0) }}),
486                             operator: token::Token { type_: token::TokenType::Greater,
                                ↪ lexeme: String::from(">"), literal: token::Literal::Null,
                                ↪ line: 1 },
487                             right: Box::new(Expr { line: 1, expr_type: ExprType::Literal {
                                ↪ value: token::Literal::Number(4.0) }}),
488                         }}),
489                         operator: token::Token { type_: token::TokenType::LessEqual, lexeme:
                            ↪ String::from("<="), literal: token::Literal::Null, line: 1 },
490                         right: Box::new(Expr { line: 1, expr_type: ExprType::Literal {
                            ↪ value: token::Literal::Number(5.0) }}),
491                     }}),
492                     operator: token::Token { type_: token::TokenType::GreaterEqual, lexeme:
                        ↪ String::from(">="), literal: token::Literal::Null, line: 1 },
493                     right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                        ↪ token::Literal::Number(6.0) }}),
494                 }}),
495             }}),
496             operator: token::Token { type_: token::TokenType::BangEqual, lexeme:
                ↪ String::from("!="), literal: token::Literal::Null, line: 1 },
497             right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                ↪ token::Literal::Number(7.0) }}),
498     }}}}]), parse(source));
499 }
500
501 #[test]
502 fn call() {
503     let source = "a(1, \"a\")(bc, 2+3)";
504     assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr {
        ↪ line: 1, expr_type: ExprType::Call {
505             callee: Box::new(Expr { line: 1, expr_type: ExprType::Call {
506                 callee: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                    ↪ String::from("a") }}),
507                 arguments: vec![
508                 Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪ token::Literal::Number(1.0) }},
509                 Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪ token::Literal::String_(String::from("a")) }}
510                 ],
511             }}),
512             arguments: vec![
513             Expr { line: 1, expr_type: ExprType::Variable { name: String::from("bc") }},
514             Expr { line: 1, expr_type: ExprType::Binary {
515                 left: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪ token::Literal::Number(2.0) }}),
516                 operator: token::Token { type_: token::TokenType::Plus, lexeme:
                    ↪ String::from("+"), literal: token::Literal::Null, line: 1 },
```

```
517                 right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                        ↪ token::Literal::Number(3.0) }}),
518             }}
519             ],
520       }]}]}]), parse(source));
521   }
522
523   #[test]
524   fn empty_call() {
525       let source = "a()";
526       assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr {
                ↪ line: 1, expr_type: ExprType::Call {
527             callee: Box::new(Expr { line: 1, expr_type: ExprType::Variable { name:
                    ↪ String::from("a") }}),
528             arguments: vec![],
529       }}}}]), parse(source));
530   }
531
532   #[test]
533   fn unclosed_call() {
534       let source = "a(1, \"a\"(bc, 2+3)";
535       assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedCharacter { expected:
                ↪ ')', line: 1 }]));
536       let source = "a(1, \"a\")(bc, 2+3";
537       assert!(errors_in_result(parse(source), vec![ErrorType::ExpectedCharacter { expected:
                ↪ ')', line: 1 }]));
538   }
539
540   #[test]
541   fn unary() {
542       let source = "!!--5";
543       assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr {
                ↪ line: 1, expr_type: ExprType::Unary {
544             operator: token::Token { type_: token::TokenType::Bang, lexeme:
                    ↪ String::from("!"), literal: token::Literal::Null, line: 1 },
545             right: Box::new(Expr { line: 1, expr_type: ExprType::Unary {
546                 operator: token::Token { type_: token::TokenType::Bang, lexeme:
                        ↪ String::from("!"), literal: token::Literal::Null, line: 1 },
547                 right: Box::new(Expr { line: 1, expr_type: ExprType::Unary {
548                     operator: token::Token { type_: token::TokenType::Minus, lexeme:
                            ↪ String::from("-"), literal: token::Literal::Null, line: 1 },
549                     right: Box::new(Expr { line: 1, expr_type: ExprType::Unary {
550                         operator: token::Token { type_: token::TokenType::Minus, lexeme:
                                ↪ String::from("-"), literal: token::Literal::Null, line: 1 },
551                         right: Box::new(Expr { line: 1, expr_type: ExprType::Literal {
                                ↪ value: token::Literal::Number(5.0) }}),
552                     }}),
553                 }}),
554             }}),
555       }]}]}]), parse(source));
556   }
557
558   #[test]
559   fn etc() {
560       let source = "5--4";
561       assert_eq!(Ok(vec![Stmt { line: 1, stmt_type: StmtType::Expression { expression: Expr {
                ↪ line: 1, expr_type: ExprType::Binary {
562             left: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪ token::Literal::Number(5.0) }}),
563             operator: token::Token { type_: token::TokenType::Minus, lexeme:
                    ↪ String::from("-"), literal: token::Literal::Null, line: 1 },
```

```
564             right: Box::new(Expr { line: 1, expr_type: ExprType::Unary {
565                 operator: token::Token { type_: token::TokenType::Minus, lexeme:
                    ↪ String::from("-"), literal: token::Literal::Null, line: 1 },
566                 right: Box::new(Expr { line: 1, expr_type: ExprType::Literal { value:
                    ↪ token::Literal::Number(4.0) }}),
567             }}),
568       }}}}]), parse(source));
569     }
570
571     #[test]
572     fn sync() {
573       let source = "print {\nfor (x = 5; x < 2; x = x + 1 {print x}";
574         assert!(errors_in_result(parse(source), vec![
575           ErrorType::ExpectedExpression { line: 1 },
576           ErrorType::ExpectedParenAfterIncrement { line: 2 },
577         ]));
578       }
579     }
580 }
```

A.3 Tests for the HashTable class

```
1  #[cfg(test)]
2  mod tests {
3    use crate::{error::ErrorType, stmt::{Stmt, StmtType}, value::Value};
4
5    use super::HashTable;
6
7    #[test]
8    fn insert_and_get() {
9      let mut dict = HashTable::new();
10     assert!(dict.insert(&Value::Number(5.0), &Value::String_("hello".to_string()),
       ↪ 1).is_ok());
11     assert_eq!(dict.get(&Value::Number(5.0), 1), Ok(&Value::String_("hello".to_string())));
12   }
13
14   #[test]
15   fn insert_duplicate_and_get() {
16     let mut dict = HashTable::new();
17     assert!(dict.insert(&Value::Number(5.0), &Value::String_("hello".to_string()),
       ↪ 1).is_ok());
18     assert!(dict.insert(&Value::Number(5.0), &Value::String_("hi".to_string()), 1).is_ok());
19     assert_eq!(dict.get(&Value::Number(5.0), 1), Ok(&Value::String_("hi".to_string())));
20   }
21
22   #[test]
23   fn insert_remove_size() {
24     let mut dict = HashTable::new();
25     assert!(dict.insert(&Value::Number(5.0), &Value::String_("hello".to_string()),
       ↪ 1).is_ok());
26     assert!(dict.insert(&Value::String_("key1".to_string()),
       ↪ &Value::String_("hi".to_string()), 1).is_ok());
27     assert_eq!(dict.size(), 2);
28
29     assert!(dict.remove(&Value::Number(5.0), 1).is_ok());
30     assert_eq!(dict.size(), 1)
31   }
32
33   #[test]
34   fn key_error() {
```

```
35      let dict = HashTable::new();
36      assert_eq!(dict.get(&Value::Number(5.0), 1), Err(ErrorType::KeyError { key:
          ↪ Value::Number(5.0), line: 1 }));
37    }
38
39    #[test]
40    fn cannot_hash_errors() {
41      let dict = HashTable::new();
42      assert_eq!(dict.get(&Value::Function { parameters: vec![], body: Stmt { line: 1,
          ↪ stmt_type: StmtType::Break } }, 1), Err(ErrorType::CannotHashFunction { line: 1 }));
43      assert_eq!(dict.get(&Value::Dictionary(HashTable::new()), 1),
          ↪ Err(ErrorType::CannotHashDictionary { line: 1 }));
44    }
45
46    #[test]
47    fn equality() {
48      let mut dict1 = HashTable::new();
49      let mut dict2 = HashTable::new();
50      assert!(dict1.insert(&Value::Number(5.0), &Value::String_("hello".to_string()),
          ↪ 1).is_ok());
51      assert!(dict1.insert(&Value::Number(6.0), &Value::String_("hello".to_string()),
          ↪ 1).is_ok());
52      assert!(dict1.insert(&Value::Number(7.0), &Value::String_("hello".to_string()),
          ↪ 1).is_ok());
53
54      assert!(dict2.insert(&Value::Number(7.0), &Value::String_("hello".to_string()),
          ↪ 1).is_ok());
55      assert!(dict2.insert(&Value::Number(6.0), &Value::String_("hello".to_string()),
          ↪ 1).is_ok());
56      assert!(dict2.insert(&Value::Number(5.0), &Value::String_("hello".to_string()),
          ↪ 1).is_ok());
57
58      assert_eq!(dict1, dict2);
59    }
60
61    #[test]
62    fn inequality() {
63      let mut dict1 = HashTable::new();
64      let mut dict2 = HashTable::new();
65      assert!(dict1.insert(&Value::Number(5.0), &Value::String_("hello".to_string()),
          ↪ 1).is_ok());
66      assert!(dict1.insert(&Value::Number(6.0), &Value::String_("hello".to_string()),
          ↪ 1).is_ok());
67      assert!(dict1.insert(&Value::Number(7.0), &Value::String_("hello".to_string()),
          ↪ 1).is_ok());
68
69      assert!(dict2.insert(&Value::Number(8.0), &Value::String_("hello".to_string()),
          ↪ 1).is_ok());
70      assert!(dict2.insert(&Value::Number(6.0), &Value::String_("hello".to_string()),
          ↪ 1).is_ok());
71      assert!(dict2.insert(&Value::Number(5.0), &Value::String_("hello".to_string()),
          ↪ 1).is_ok());
72
73      assert_ne!(dict1, dict2);
74    }
75  }
```

A.4 Tests for the Environment class

```rust
#[cfg(test)]
mod tests {
  use crate::{value::Value, error::ErrorType, environment::Pointer};

  use super::Environment;

  #[test]
  fn one_scope() {
    //  var a = 5
    //  var b = [true, "hello world!"]
    //  b = "abc"
    let mut env = Environment::new();
    env.declare(String::from("a"), &Value::Number(5.0));
    env.declare(String::from("b"), &Value::Array(vec![Value::Bool(true),
    ↪  Value::String_(String::from("hello world!"))]));
    assert_eq!(env.get(String::from("a"), 1), Ok(Value::Number(5.0)));
    assert_eq!(env.get(String::from("b"), 1), Ok(Value::Array(vec![Value::Bool(true),
    ↪  Value::String_(String::from("hello world!"))])));

    let _ = env.update(&Pointer { name: String::from("b"), indices: vec![] },
    ↪  &Value::String_(String::from("abc")), 1);
    assert_eq!(env.get(String::from("a"), 1), Ok(Value::Number(5.0)));
    assert_eq!(env.get(String::from("b"), 1), Ok(Value::String_(String::from("abc"))));
  }

  #[test]
  fn many_scopes() {
    //  var a = 1
    //  var b = 2
    //  {
      //      a = 10
      //      "a == 10?"
      //      var b = 20
      //      "b == 20?"
      //      {
        //          b = 30
        //          "b == 30?"
        //      }
      //      "b == 30?"
      //  }
    //  "a == 10?"
    //  "b = 2?"
    let mut env = Environment::new();
    env.declare(String::from("a"), &Value::Number(1.0));
    env.declare(String::from("b"), &Value::Number(2.0));

    env.new_scope();
    let _ = env.update(&Pointer { name: String::from("a"), indices: vec![] },
    ↪  &Value::Number(10.0), 1);
    env.declare(String::from("b"), &Value::Number(20.0));
    assert_eq!(env.get(String::from("a"), 1), Ok(Value::Number(10.0)));
    assert_eq!(env.get(String::from("b"), 1), Ok(Value::Number(20.0)));

    env.new_scope();
    let _ = env.update(&Pointer { name: String::from("b"), indices: vec![] },
    ↪  &Value::Number(30.0), 1);
    assert_eq!(env.get(String::from("b"), 1), Ok(Value::Number(30.0)));

    env.exit_scope();
```

```
55      assert_eq!(env.get(String::from("b"), 1), Ok(Value::Number(30.0)));
56
57      env.exit_scope();
58      assert_eq!(env.get(String::from("a"), 1), Ok(Value::Number(10.0)));
59      assert_eq!(env.get(String::from("b"), 1), Ok(Value::Number(2.0)));
60    }
61
62    #[test]
63    fn name_error_get() {
64      let env = Environment::new();
65      assert_eq!(env.get(String::from("b"), 1), Err(ErrorType::NameError { name:
          ↪ String::from("b"), line: 1 }));
66    }
67
68    #[test]
69    fn name_error_assign() {
70      let mut env = Environment::new();
71      assert_eq!(env.update(&Pointer { name: String::from("b"), indices: vec![] },
          ↪ &Value::Null, 1), Err(ErrorType::NameError { name: String::from("b"), line: 1 }));
72    }
73
74    #[test]
75    fn declare_twice() {
76      let mut env = Environment::new();
77      env.declare(String::from("b"), &Value::Number(123.0));
78      env.declare(String::from("b"), &Value::Number(55.0));
79      assert_eq!(env.get(String::from("b"), 1), Ok(Value::Number(55.0)));
80    }
81  }
```

A.5 Evidence of unit tests passing

```
running 54 tests
test environment::tests::name_error_get ... ok
test environment::tests::name_error_assign ... ok
test environment::tests::declare_twice ... ok
test environment::tests::many_scopes ... ok
test hash_table::tests::cannot_hash_errors ... ok
test environment::tests::one_scope ... ok
test hash_table::tests::inequality ... ok
test hash_table::tests::insert_and_get ... ok
test hash_table::tests::insert_duplicate_and_get ... ok
test hash_table::tests::insert_remove_size ... ok
test hash_table::tests::equality ... ok
test hash_table::tests::key_error ... ok
test parser::tests::array ... ok
test parser::tests::bidmas ... ok
test parser::tests::call ... ok
test parser::tests::comparison ... ok
test parser::tests::element ... ok
test parser::tests::element_2d ... ok
test parser::tests::else_ ... ok
test parser::tests::empty_array ... ok
test parser::tests::else_if ... ok
test parser::tests::empty_call ... ok
test parser::tests::error_line_numbers ... ok
test parser::tests::etc ... ok
test parser::tests::for_ ... ok
test parser::tests::for_no_cond ... ok
test parser::tests::for_no_cond_semicolon ... ok
test parser::tests::for_no_inc ... ok
test parser::tests::for_no_init ... ok
test parser::tests::for_no_init_semicolon ... ok
test parser::tests::func ... ok
test parser::tests::func_keyword_name ... ok
test parser::tests::if_ ... ok
test parser::tests::invalid_var_name ... ok
test parser::tests::logic ... ok
test parser::tests::multiple_statements ... ok
test parser::tests::print ... ok
test parser::tests::unary ... ok
test parser::tests::sync ... ok
test parser::tests::unclosed_array ... ok
test parser::tests::unclosed_block ... ok
test parser::tests::unclosed_call ... ok
test parser::tests::unclosed_for ... ok
test parser::tests::unclosed_grouping ... ok
test parser::tests::unopened_block ... ok
test parser::tests::var ... ok
test parser::tests::while_ ... ok
test tokenizer::tests::comments ... ok
test tokenizer::tests::identifiers_and_keywords ... ok
test tokenizer::tests::line_count ... ok
test tokenizer::tests::literals ... ok
test tokenizer::tests::one_char_tokens ... ok
test tokenizer::tests::one_two_char_tokens ... ok
test tokenizer::tests::unterminated_string ... ok

test result: ok. 54 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.02s
```

Appendix B

Photos

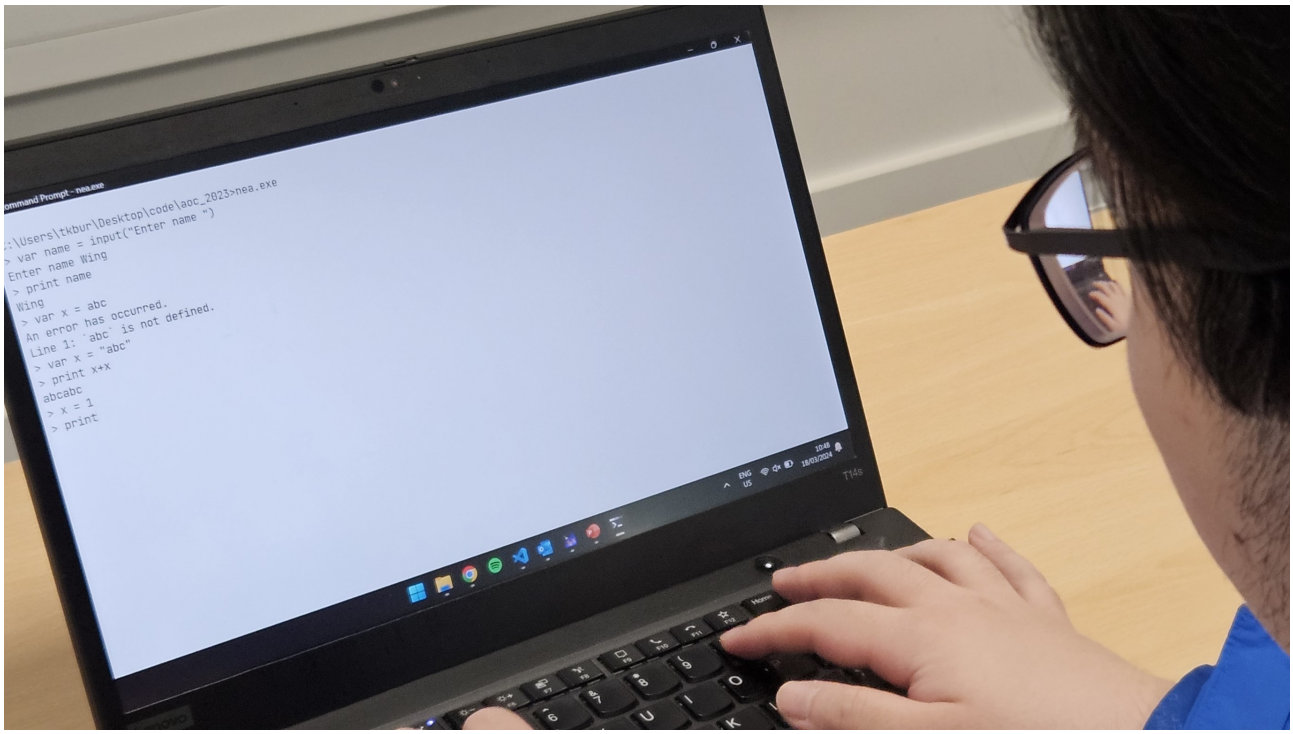

Figure B.1: Mr Chiu using the REPL interface

Figure B.2: Mr Chiu seems happy with the interpreter

Figure B.3: Moncef Slimani using the interpreter

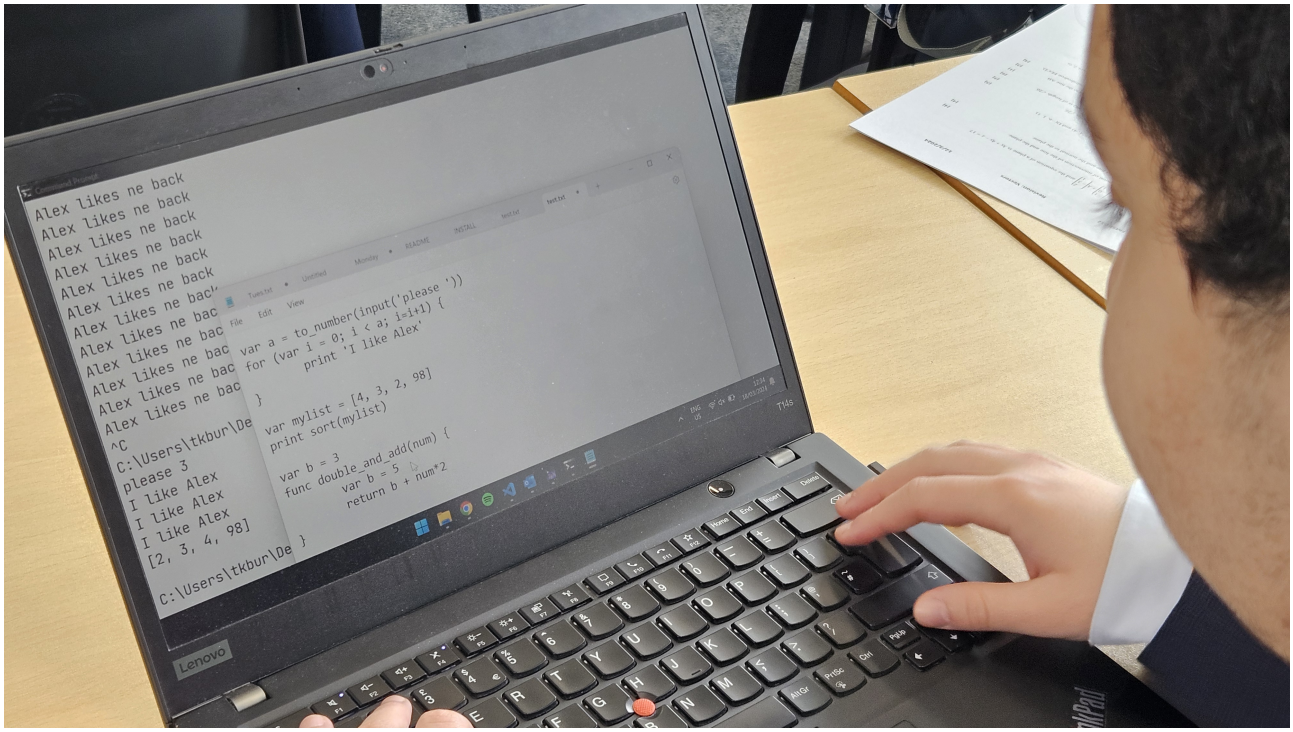

Figure B.4: Moncef Slimani running source code files