The Lilac Programming Language

Language Design and Interpreter Implementation

Name:	:3
Candidate Number:	3253
Centre Number:	14613

Orleans Park School

Contents

1: Analysis

1.1 Introduction

For my A-level project I am going to design and implement a small programming language, called Lilac. In the world of programming languages, there are the big production languages: Python, JavaScript, C++, C#, etc. While they are the most commonly used languages, they are by no means the only ones. On the outskirts of the field of language design, there are many small languages which seek to either push the boundaries of computation and abstraction, or that have incredibly specific applications (along with, of course, the ones written as jokes). For example, the `Orca` language is a two-dimensional graphical language which is designed for programmatic music production. `Pinecone` was another attempt at creating a light c-style language. And most commonly formats such as `toml` and `yaml` that are used for configuration files lean gently into the most minimal of language. To summarise, wherever there could be an issue with one of the large languages, there are small languages being designed to try to fix it.

I see Lilac in this way. The issue that it is trying to solve is the accessibility of functional programming. Haskell, the giant of the paradigm, is incredibly difficult to learn because it is a huge jump in abstraction and type theory. But often the functional approach to problem solving is the most efficient, and it is an incredibly powerful paradigm. My motivation for this project is to create an intermediary language, a solution that is genuinely functional, but not overwhelming for a first-time functional programmer. As such, the type system should be loose, and I take some liberties with the strictness of functions. But at the core of it, it is a language strongly inspired by lambda calculus, and the aim is that learning Lilac would solidly introduce the concepts necessary for functional programming.

1.2 Translator Design

A translator is a piece of utility software that takes one set of program source code and turns it (translates) into the program source code of another language. They are incredibly versatile pieces of software, and as such there are many different ways of desinging and implementing translators. Generally, they come in three different forms. Most basic are assemblers, which take assembly language code and translate it to executable machine code. Then, we have interpreters and compilers, which translate high level languages. These are significantly more complicated, since they also have to take into account the code semantics and structure. Interpreters and compilers differ generally in their output. And interpreter translates line by line (or construct by construct) and executes as it goes, stopping when it reaches a halting condition; this could be an error or just the end of the code. Compilers translate the entire source code and output an executable machine code file. It always stops when it reaches the end, forming a list of errors as it goes.

Although interpreters and compilers have separate outputs, they follow very similar steps. The way that these steps are connected, so the path that the data takes, is part of what gives each language and translator its individual flavor.

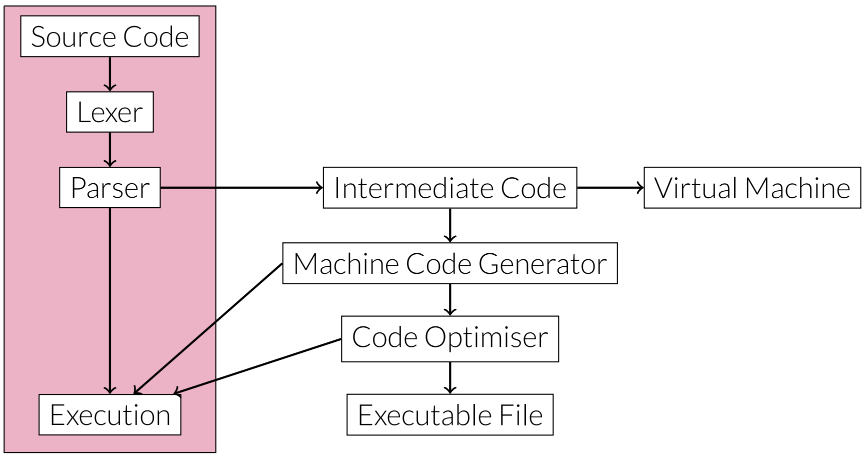

Figure 1.1: Graph showing the possible steps a translator might take.

Figure 1.2 shows the steps a translator could take when executing a piece of program source code. The section in the purple box is what my project is limited to. The first two stages are common to all translators. The lexer performs lexical analysis on the source code. This turns the input text into a list of tokens. At their simplest, tokens are just simple pairs that each represent one semantic component of the source code. In programming, the line of code `var: 3` would be turned into `[(Identifier, 'var'), (Assi gnment, ':'), (Number, 3)]`. At this point, errors can already be caught. For example, if an identifier does not start with a letter or if an illegal symbol is used, the lexer will catch this. If we introduce an example from english, the sentence "The cat likes to sleep" is composed of different types of words. The lexer would be where we notice that "Teh cta ilkse ot eples" is not an allowed sentence in english, since none of the words are allowed. However, the lexer would not notice that "cat to the sleep likes" is not an allowable sentence.

Now, the source code is represented as a flat list of tokens. However this list does not have any grammatical structure. Adding this structure is the next step, called parsing. The result is a parse tree or abstract syntax tree, a tree data structure which represents the grammatical structure of a string. This is where operator precedence and associativity becomes apparent, and differentiates `(3+2)*4` from `3+(2*4)`. Here, grammatical errors are noticed. If I use the assignment operator with nothing on one side, this is noticed as an error. To use the english example, the parser would notice that "cat to the sleep likes" should not be allowed as an english sentence, because the order of word types does not follow the rules of the grammar. For "the cat likes to sleep", the parse tree may look like Figure 1.2. Notice that the syntactic rules of the English grammar are implied as subtrees of the overall parse tree. This same idea applies to programming languages; for a simple line of code in Lilac like `var: 4 * (2 + 3)` the tree would look like Figure 1.3.

Notice that in this in this tree the parentheses are not included. Instead, they are used to specify subtrees of the overall parse tree. If I wrote `4 * 2 + 3`, the parser would realise that `*` is first in the order of operations. It would therefore treat it as `(4 * 2) + 3`, which produces a different tree. There is also another key difference between English and Lilac. Notice that in the programming language, every node of the tree has a token in the string associated, whereas in the english sentence this is not the case. Lilac is explicitly structured, or context-free. There is no way of making a string which can be parsed in more than one way.

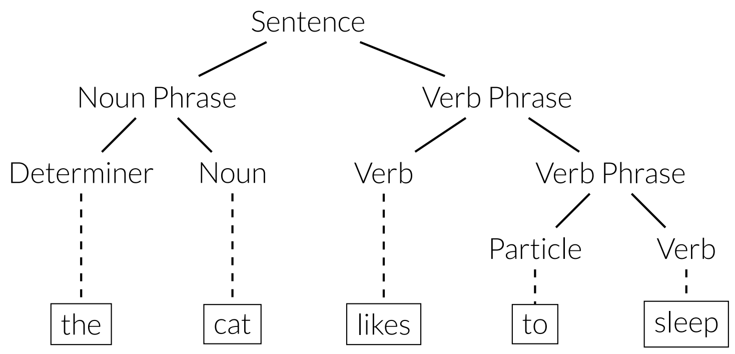

Figure 1.2: English parse tree of the sentence ”the cat likes to sleep”

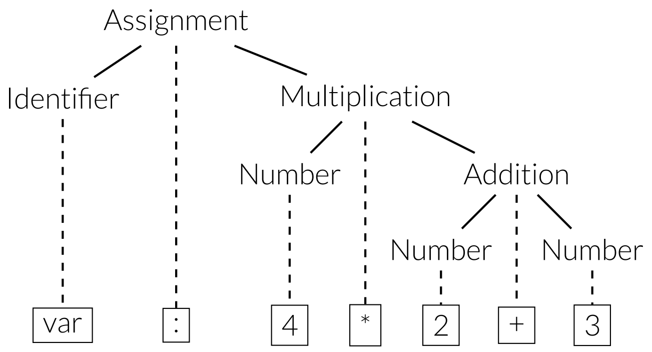

Figure 1.3: An example parse tree for Lilac

There are several ways to define the grammar of a language, and the algorithms to parse it. A common method for defining a grammar is through production rules, usually written in a syntax called Backus-Naur Form. Here is an example, which defines arithmetic between integers and the order of operations:

```
<expr> ::= <expr> + <term>
         | <expr> - <term>
         | <term>

<term> ::= <term> * <factor>
         | <term> / <factor>
         | <factor>

<factor> ::= ( <expr> )
           | - <factor>
           | <int>

<int> ::= <int><digit>
     s   | <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The use of `::=` defines a rule in the production, which is represented with an identifier within the angle brackets. For example, the rule `<expr>` tells us that it is either an `<expr>` followed by a `+` symbol followed by a `<term>`, or an `<expr>` followd by a `-` symbol followed by a `<term>`, or just a `<term>`. The repeated application of these rules can be used to determine whether a string of terminal characters (everything that is not `::=`, \|, or in angle brackets) can be produced from this grammar. There is no way to make the string `3+`, so this is not part of the language defined by the grammar. This sort of definition is used for context-free languages, where the syntax can always be represented by a set of deterministic rules. Parsing algorithms are generally divided in two categories. Top-down parsers start with the wider rules and then narrow down, while bottom-up parsers start with the terminals and build up a parse tree.

Execution is comparatively simpler. Most methods use a walk through the parse tree along with a call stack to execute the code. The call stack keeps track of the order in which sections of code should be executed. The individual pieces are wrapped in ”stack frames” that seperate scope and pointers. For example, recursive function calls are protected by the stack frames so that the inner scopes do not overwrite the outer scope. After something is pushed to the call stack it can then be popped and executed in the correct order. If the interpreter is written in a low-level language the individual memory management of each operation has to be implemented. If it is written in a high-level language then the operations would be implemented in that same high level language.

1.3 Proposed Solution

My proposed solution is called `Lilac`. It is a minimal functional programming language inspired by the syntax of Pinecone and Boa, along with the function logic of Haskell. Here is an example source file:

```
main: ( print "Hello World" )

addToThree: ( fn x → 3 + x )

fooBarBaz:
  let (
    byThree: n % 3,
    byFive: n % 5 )
  in (
    fn n → byThree ? "Foo" | byFive ? "Bar" | "Baz" )
```

1.3.1 Language Specification:

1.3.1.1 Types:

Lilac has built in data types `number`, `string`, and `boolean`, although these are mostly hidden from the user. Lilac is dynamically and weakly typed. The set `number` is defined as all integers and floating point numbers, `string` is the set of all character strings, and `boolean` is the set `{true, false}`.

1.3.1.2 Operators:

All the standard operators for numeric arithmetic, boolean arithmetic, logic and comparison are implemented:

```
Numbers: +, -, *, /, %
Boolean: &&, ||, !
Comparison: =, <, >, ⩽ , ⩾ , ≠ (! =)
```

1.3.1.3 Variables:

Variables are created when a value is assigned to a name using the `:` operator. Lilac evaluates lazily, which means that it does not evaluate any expressions untill the point where the value is needed. This means that you could define the variable `x: 3 + 2`, and it will be kept in memory as the expression $3 + 2$ until x is referenced. This is unlike eager evaluation where the value of `x` is immediately calculated on assignment.

In Lilac everything is also a first-class citizen, which means that practically any expression (other than an assignment) can be assigned to a variable or passed to and from a function.

1.3.1.4 Conditionals:

Lilac uses a ternary operator syntax for an if-then-else statement. The statement "if x then y else z" is `x ? y` z|. Of course, `z` could be another conditional expression; if `z: a ? b` c|, the statement becomes `x ? y` a ? b | c|, or in natural language "if x then y else if a then b else c". From an evaluation point of view, conditional expressions aren't staatements or control structures, but operators that evaluate to values. So, our first example `x ? y` z| is equivalent to `y` if `x` is true. This means a variable can have a conditional value.

1.3.1.5 Functions:

The syntax for function defintion is inspired by lambda expressions in haskell and python:

```
Haskell:
        (\x -> x + 2)
Python:
        (lambda x: x + 2)
Lilac:
        ( fn x -> x + 2 )
```

The label `fn` is explicitly required to show that it is a function (in the future there is the possibility of adding different varieties). A function can be treated as a block which is assigned to a variable, or a lambda which is directly applied to a value, or again as a first class citizen that is returned from another function. Function application uses a blank whitespace as the operator. Crucially, this is left associative. Consider this example:

```
add: ( fn x -> ( fn y -> x + y ))
add 3 2 = (add 3) 2 = 5
```

The second line is true because of the left associativity of the whitespace operator. `(add 3)` returns a function which is applied to `2`. The operator `;` can be used to change the direction of associativity, similarly to parentheses:

```
print add 3 2 = (((print add) 3) 2) (Error because (print add) does not return a function)
print;add 3 2 = (print ((add 3) 2))
```

1.3.1.6 Let-In expression:

This expression is inspired from mathematical literature where the values in an expression are often specified in the following way:

$$\text{let } a = 2 \text{ and } b = 3 \text{ in } \sqrt{a^2 + b^2}$$

In Lilac, it can be helpful to be able to break up a function into temporary definitions. It could be done in the following way:

```
divBy:
  ( fn n → ((fn x → x % 3) n = 0 ? "Three" |
             (fn x → x % 5) n = 0 ? "Five" | "Other" ))

divBySimpler:
  let
    ( byThree: ( fn x → (x % 3) = 0 ),
      byFive:  ( fn x → (x % 5) = 0 ))
  in
    ( fn n → byThree n ? "Three" | byFive n ? "Five" | "Other" )
```

1.4 Objectives

By the end of this project I aim to have written an interpreter for the Lilac programming language which can:

- Execute arithmetic
- Execute boolean arithmetic and comparison
- Define and store variables
- Define and use functions
- Conditional statements
- Recursive function calls and definitions

In addition to this, the interpreter should provide a REPL and the ability to run source files. It should also be able to import files into the REPL or another script. This should also be accessible from a shell script.

2: Design

2.1 Stage One: Framework and Interface

I start with a driver class called `Lilac`. This will contain all the interactive elements of the interpreter, and drive the execution. The class will expose the useful functions `runFile`, `runLine`, both of which take a simple string of text and then execute it as lilac. It also provides a REPL functionality that can provides an interactive output as shown here:

```
Lilac Interactive Mode

<i> str: "hello"
<i> length str
<o> 5
<i> f:
  >   let ( y: length str )
  >   in ( fn n → y + n )
<i> x + f 3
<e> [Line 1] NameUndefinedError: The name 'x' does not exist in this scope
```

2.1.1 Error Reporting

It is crucial that my project implements some kind oferror checking and reporting. There are many errors that a user could make, and each should be differentiated and handled. There needs to be some kind of enumerable which keeps track of all the error types. It will look as follows:

```
enum ErrorType:
        SyntaxError
        NameUndefinedError
        ArgumentError
        TypeError
```

These will be used in conjunction with Lilac's `error` method. This will take an error type, a line number, and a message about the error to print.

2.2 Stage Two: Lexical Analysis

The lexer is represented by a Lexer class:

```
class Lexer:
    tokens :: List[Tokens]
    source :: String
    scan :: method
    add_token :: method
    match_next :: String → bool
    peek :: method → String
    is_alpha :: String → bool
    is_num :: String → Bool
```

The scan method implements the scanning algorithm. This takes the source code as a string and loops through it. The algorithm is as follows:

```
while not at the end of the source:
    character = consume the next character
    match on character:
        if it represents a single charcter token:
            add token(character)
        if it can start a two character token:
            check the next character without consuming
            add token(both characters)
        otherwise:
            add token representing a number or identifier
            or throw an error
```

This algorithm deals with characters that can form multiple tokens (like <and ⩽) by cautiously looking ahead at the next character. If the scanner detects an alphanumeric character, it moves to the state of scanning a number, identifier, or string. It does this by continuing to advance until certain criteria are met. For a number, it keeps going until the next character is not a digit, but one period is allowed in the number. If there is a period, then the number is a float, otherwise it is an int.

2.3 Stage Three: Parsing

In parsing, I seek to translate the list of tokens that the Scanner outputs into a tree data structure. I decide to use a recursively defined tree in order to optimise the simplicity and dynamicity of the structure definition. The tree type is defined as follows:

```
class Tree:
        token :: Token
        action :: Action
        left :: Tree
        right :: Tree

        is_leaf :: Tree → boolean
```

The parser uses a recursive, functional approach:

```
function to_tree(tokens: list of tokens):
        if there is still at least one operator in the source:
                index = find position of the lowest precedence operator
                action = make a new action corresponding to this operator
                left_expr = all tokens to the left of index
                right_expr = all tokens to the right of index
                operator = element of tokens at index

                clean outer parentheses from the left and the right

                left_tree = to_tree(left_expr)
                right_tree = to_tree(right_expr)
                return a Tree with operator, action, left_tree, right_tree

        otherwise:
                action = make a new literal action
                return a Tree with operator, action
```

To find the position of the operator with the lowest precedence, I need a table defining precedence and associativity. Figure 2.1 shows the rules that I make for lilac, inspired by the operator rules for `Haskell` and `C`. When finding the index, the algorithm should ignore expressions in parentheses, as these are guaranteed to be subtrees. Cleaning the parentheses from an expression simply means turning `(2*3)` into `2*3`; parentheses are only important for parsing, and should themselves not be parsed. The associativity of the operator describes the direction in which it should be parsed if there are multiple identicaloperators in a row. For example, addition is given right associativity, which means `a + b + c` is understood as `(a + (b + c))`. Notice, for example, that function application is left associative, but the semi-colon is right associative. Both of these ”do the same thing”, but in opposite directions. Conditional statements also need to be carefully considered. In the complex statement `a ? b | c ? d | e`, we first want `a` to be tested, then to either do `b` or test `c`. The expression shoul be parsed like this: `(((a?b)|(c?d))|e)`, which means pipe has to be left associative so that the first conditional is put prior in the order.

Token Type	Precedence	Associativity
Colon (`:`)	0	Right
Space ( )	10	Left
Arrow (→)	1	Right
Pipe (|)	2	Left
Question (`?`)	2	Right
Semi-colon (`;`)	3	Right
Slash (`/`)	8	Right
Star (`*`)	8	Right
Plus (`+`)	7	Right
Minus (`-`)	7	Right
Equal (`=`)	6	Right
Less (`<`)	6	Right
Greater (`>`)	6	Right
Less or equal (⩽)	6	Right
Greater or equal (⩾)	6	Right
Or (||)	5	Right
And (`&&`)	4	Right

Figure 2.1: Operator table which defines the syntax of Lilac

2.4 Stage Four: Execution

2.4.1 Execution model: The Tree Machine

The parser, thanks to my design of Lilac, outputs a recursive binary parse tree. At this point there are a few ways to proceed with execution. I considered doing a post-order traversal, and then using a simple stack/virtual machine. However, it is difficult to implement code branching and lazy evaluation this way. This is due to the fact that it is a bottom up technique, so at the leaves of the tree it is completely unaware of the context above it. Instead, I decide to manipulate the AST directly, and to execute it recursively in a top down manner. This is handled by my driving class, the `TreeMachine`. It is defined as follows:

```
TreeMachine:
    env_monad :: EnvMonad
    tree :: Tree

    execute :: Tree → (*output)
```

The TreeMachine is the object that is responsible for running the code. In the REPL context it will only do the execute function on one line, but in a script context it will also handle imports, (running the machine on another source file while ignoring main), and running `main`.

2.4.2 Monads: Stack

A central issue in my design is deciding where behaviours go, and who handles what. To solve this, I decide to opt for a monadic model. A monad is an object that contains some value, and handles behaviors and side effects relating to that value. This is helpful for separating behavior from types, and to improve the simplicity of the code. For example, the `Maybe` monad allows for safe computation. The data stored within it can be either `Just x` or `Nothing`. If a computation fails, the monad catches this and makes the value `Nothing` - our computation is saved. The two main components of a monad are the `return` function and the `bind` function. Borrowing the type signatures from haskell, these look like:

```
return :: Monad m ⇒ a → m a
bind (>>) :: Monad m ⇒ m a → (a → m b) → m b
```

Here we see that `return` takes a value of type `a` and returns a monad containing a value of type a. In an object-oriented way, this is the class constructor. We also see that bind, which also has the operator `>>` takes a monad of type `a` and a function that returns a monad of type `b`, then returns a monad of type `b`. Simply, it takes a monad, applies the function to the value inside the monad, then returns the output from that. This is the central benefit of the monad; the behavior is completely separate from the value.

For my call-stack, which is used to evaluate expressions, I use this model. I create three classes: `Stack`, `Node`, and `StackMonad`. The stack is defined in a dynamic linked way using the `Node` class in the following way:

```
class Node:
    value :: Token
    next :: Node
```

```
class Stack:
    top :: Node

    pop :: Stack → Stack
    push :: Stack → Token → Stack
```

Then, I define the `StackMonad` like so:

```
class StackMonad
    stack :: Stack
    out :: List[Tokens]

    (>>) :: StackMonad → function → StackMonad
```

The functions that bind will take are the stack operations, and bind will simply run them. The output of popping the stack is stored in the `out` list of the monad.

2.4.3 Monad: Environment and Actions

I have an `Environment` class, which can also be thought of as a scope. This class looks as follows:

```
class Environment:
    stack_monad :: StackMonad
    table :: Dictionary
    tree :: Tree (optional)
```

We can see here that the environment holds only data, and no behaviors. It is contained within my `EnvMonad` class:

```
class EnvMonad:
    env :: Environment
    trace :: List

    bind (or >>) :: EnvMonad → Action → EnvMonad
    consume :: EnvMonad → Environment → EnvMonad
```

The `bind` function is a feature of monads. It takes an `EnvMonad` (i.e. self) and an `Action`, runs the action on the data of the monad, then returns the result wrapped in a new monad. The `Action` is a function wrapped in a class:

```
class Action:
    left :: Tree
    right :: Tree
    run :: Environment → Environment
    check :: (*args) → (*outputs)
```

Each component of the language gets its own action, and the interface for each action is strictly the same. This allows me to easily extend the language, simply by adding new actions. Each action has a reference to the left and right subtree of its parent node. Execution is done top-down: before an action executes itself, it does the left action and the right action. So, the run algorithm looks like:

```
run(envmonad):
    envmonad >> left action >> right action
    get any output
    check the arguments for the action
    do the action
```

Since actions are kept in classes, I can use inheritance to approximate the idea of typeclasses. I'll make the `Action` class generic, and then have other actions inherit from it. For example, the `ArithmeticAction` will implement the `check` method so that it make sures the left and right operands are numbers. Then thte actions for addition, multiplication, etc. will inherit from `ArithmeticAction` and will be able to use the specific check method. This also means I can do general type checks on actions. The action pushes its result to the stack inside a `Token`, and while it does the calculation. The stack is used as the intermediate structure where data is passed between actions and kept in order.

2.5 Example

To understand how my design works, let's imagine the simple example `var: 3 + 4`. The scanner understands this as `[(IDENTIFIER, var), (COLON, :), (NUMBER, 3), (PLUS, +), (NUMBER, 4)]`. Then, the parser parses this and gives it the structure show in Figure 2.5. This structure is translated into an action tree (which lives, polymorphically, on the same tree rather than on a new instance). This structure is then passed for execution. To begin with, the action of the root node is executed (`AssignAction`). This executes the action of its left subtree and its right subtree, then itself. The left subtree is an `IdentifierAction`, which represents a terminal. It pushes `var` to the stack (Figure 2.3.1), then returns. The

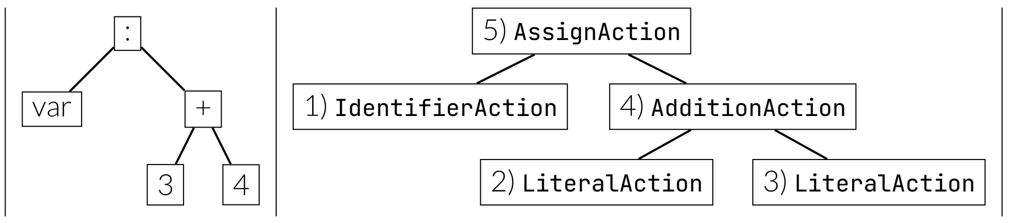

Figure 2.2: The parse tree for `var: 3 + 4`, as a symbol tree and an action tree.

1) `IdentifierAction(var)`

Stack:		Data Table:
top	`var`	
1		
2		

2) `LiteralAction(3)`

Stack:		Data Table:
top	`3`	
1	`var`	
2		

3) `LiteralAction(4)`

Stack:		Data Table:
top	`4`	
1	`3`	
2	`var`	

4) `AdditionAction`

Stack:		Data Table:
top	`7`	
1	`var`	
2		

5) `AssignAction`

Stack:		Data Table:
top	`Assigned`	
1		`var 7`
2		

Figure 2.3: The steps, 1-5, when executing the tree in Figure 2.5

3: Implementation

3.1 Context and Overview

3.1.1 Language Choice

I am using `Python 3.10` and `bash` to implement `Lilac`. The choice of language is crucial, and I considered a few options before settling on Python. In terms of speed, `C++` would be the ideal choice. However, it requires a lot of gritty memory management that is not feasible in the scope of this project. This also eliminated `Rust`, which I briefly considered for its speed and the way Enums and Structs are defined. `Haskell` and `C#`, both of which I know, were considered too. However, each of these is very rigidly in one paradigm, which means that they are not well suited to certain stages of the process. Redesigning the project for one of these would also be unfeasible. So, I turned to Python. Python is well suited to object-oriented programming, but can very easily be used to write in a functional style. This flexibility allows me to adapt my implementation to the specific component I'm writing, while keeping the data I handle in a single form. This does mean I sacrifice on execution speed, but as a proof of concept project this is acceptable.

In addition to this, Python 3.10 has several features that make it particularly useful. It has a powerful pattern-matching 'switch' statement that will be useful when I have large conditionals. Python also lets me easily define decorators that allow me to group behaviours.

3.1.2 Project Strcuture

The project is kept in one folder, called `lilac`, with and `__init__.py` file (Figure **??**). This makes `lilac` a python module, that can be imported in a `main.py` file sitting alongside the lilac folder.

3.2 Implementation Details

3.2.1 House-keeping and Management

3.2.1.1 Lilac driver class

The Lilac class, defined in `driver.py` (Appendix **??**), is the main entry point and management class. It is implemented mostly identically to the design. It is a static class, so it is never instatiated, and does not have a constructor. shows the entry point for the interactive mode.

3.2.1.2 Status and Config

In my original design, I had the Lilac class also handle error checking. However, due to Python's import system, this would cause circular import errors, since the `driver.py` file imports `Scanner` from `scanner.py` (Appendix **??**), which imports `Lilac` from `driver.py`, and so on. To solve this, I split the behavior over two extra files. The first is `config.py` (Appendix **??**), which defines the class `LICONF`, my global configuration container. The key properties are `HAD_ERROR`, which is the global flag that tells the various components of lilac whether there has been an error. It also has information about logging: where to put the log, what the log level is, a table mapping log levels to integers, and a path to the log file. Since `config.py` is the first module loaded by `__init__.py`, it becomes available to all subsequent modules.

The second file is `error_system.py` (Appendix ??). It exports two classes. The first is a simple `ErrorType`, which enumerates the possible types of error. The second is `StatusHandler`, to which I move all the error handling methods originally in `Lilac`. The `throw` method (?? l.47-58) is called whenever another part of the project detects an error, for example when the scanner meets an unrecognised character. The method checks if the program is running in the interactive mode (?? l.51). If it is, the output is handled differently. Most importantly, `LICONF.HAD_ERROR` is set to `True` (?? l.53). This will stop future execution from happening, aand eventually the program will return out to the repl loop with no output. This class also exports two decorators that are used all over my program to do repetitive tasks.

First, an explanation of decorators in Python. Decorators are curried functions that can be placed in front of other method definitions using the `@` symbol. They affect what happens when the function is called. Their basic structure is:

```
def decorator(function):
    def wrapper(*args):
        print('Things happen before...')
        out = function(*args):
        print('...and after.')
        return out
    return wrapper

@decorator
def example(string):
    print(string)

example('Hello World!')
```

We can see that this allows us the option to pad a call to our function in behavior, typically simple repetitive things. In the above example, the output is:

```
Things happen before...
Hello World!
...and after
```

What is really happening when we call `example('Hello World!')` is `decorator(example)('Hello World!')`.

In my project, I use this to handle logging and error checking. StatusHandler implements the `checkerror` decorator (?? l.35-43). I can put this before the functions that should be skipped if there has been an error. If there has, the function is not run and it returns nothing. This cascades so that it returns back out to the REPL loop. I also use this for logging, using the log parameters in `LICONF`. The `logging` decorator (?? l.16-32) is more complex, in that it has three ”levels”. This is so that I can pass arguments to the decorator. There are several log levels, such as `DEBUG`, `INFO`, and `ERROR`, which relate to the significance of an event. This is passed to the logging decorator, so that I can flag the importance of each function that it decorates. In the decorator, I check if the global log threshold is less than or equal to the log level passed to the decorator. If yes, it then checks how to display the log, and outputs it, either to a file or the console. Then, it just does the function and returns the output.

3.2.2 Data Types

In this project, I try as much as possible to separate behavior from data. Instead, I focus on letting data be data and treat behaviors as actions on the data. This means that first, in order to understand my implementation, we have to understand the types I define to hold the information my program uses.

3.2.2.1 Token

The elementary piece of data that my program handles is a Token (Appendix **??**), which represents a single semantic item in source code. It has a type, a lexeme (which is the way that the token appears in source as a string), a line number, and a literal. The literal field is only used when the token represents a number or a string, and stores its actual value. In the case of a string `"string"`, the lexeme is `"string"` but the literal is `string`. We can see that the Token object is very simple, and only exports its constructor and its representation as a string.

The type that a token can have is defined by the `TokenType` object (Appendix **??**). It derives from the `Enum` class exported by the `enum` package, which lets me use the `auto()` method, and treats the items as key-value pairs, rather than identifiers. Each TokenType item represents a possible token in source code.

3.2.2.2 Node and Stack

The call stack uses a dynamic node-based implementation (Appendix **??**), and looks mostly like the design in Section 2.4.2. Important are the functions `push` (**??** l.45-52) and `pop` (**??** l.55-60), the two stack operations. In order to be used by the `StackMonad` I made these curried. This means, when I do `StackMonad >> Stack.push(a) >> Stack.pop()`, each call to the stack operation returns the function that can run on the stack.

A: Code

A.1 Directory Structure

A.2 __init__.py

```python
"""Lilac Interpreter Module.

This module exports all the necessary functions and classes used for execution of the
↪  Lilac language.

Exports:
    Action -- Generic action used by the EnvMonad.
        ArithmeticAction -- Action representing arithmetic.
            AdditionAction -- Action representing addition.
        LiteralAction -- Action representing a literal or identifier.
    EnvMonad -- Monadic wrapper around an Environment.
    Environment -- Data holding class to keep track of program state.
    Error -- Error handling class.
    ErrorType -- Enum type represents
    Grammar -- Class holding all the rules and classifications of the language.
    Lilac -- Driver class.
    Node -- Used to define the Stack.
    Parser -- Handles all parsing behavior.
    PrettyPrinter -- Exports functions to print the contents of the program state in
↪   readable formats.
    Scanner -- Handles all scanning behavior.
    Stack -- Call stack data structure.
    StackMonad -- Mondaic wrapper around Stack.
    Token -- Class representing a lexical token.
    TokenType -- Enum representing the kinds of textual tokens.
    Tree -- Recursively defined generic tree structure for parsing.
"""

from .config import *
from .error_system import *
from .mermaid_printer import *
from .token_type import *
from .grammar import *
from .tokens import *
from .tree import *
from .stack import Stack, StackMonad
from .environment import Environment, EnvMonad

from .tree_actions import *
from .tree_machine import *

from .scanner import *
from .parser import *
from .driver import *
```

A.3 actions.py

```python
from . import *

class LiteralAction:
    def __init__(self, val):
        self.val = val

    def run(self, glob):
        glob.stackmonad >> Stack.push(self.val)
        return glob


class AssignAction():
    def __init__(self):
        pass

    def run(self, glob):

        glob.stackmonad >> Stack.pop() >> Stack.pop()
        val1 = glob.stackmonad.out[-1]
        val2 = glob.stackmonad.out[-2]

        glob.table[val1] = val2
        return glob


class ArithmeticAction:
    def __init__(self):
        pass

    def check(self, glob, op1, op2):
        # if try
        if isinstance(op1, str):
            try:
                val1 = glob.table[op1]
            except:
                print(f'Identifier {op1} does not exist in the data table')
        else:
            val1 = op1

        if isinstance(op2, str):
            val2 = glob.table[op2]
        else:
            val2 = op2

        return val1, val2



class AdditionAction(ArithmeticAction):
    def __init__(self):
        pass

    def run(self, glob):
        glob.stackmonad >> Stack.pop() >> Stack.pop()
        op1 = glob.stackmonad.out[-1]
        op2 = glob.stackmonad.out[-2]

        # check the results
        val1, val2 = self.check(glob, op1, op2)

        # do the operation
        ans = val1 + val2
        glob.stackmonad >> Stack.push(ans)
        return glob

class SubtractionAction(ArithmeticAction):
    def __init__(self):
        pass

    def run(self, glob):

        glob.stackmonad >> Stack.pop() >> Stack.pop()
        op1 = glob.stackmonad.out[-1]
        op2 = glob.stackmonad.out[-2]

        # Check the results
        val1, val2 = self.check(glob, op1, op2)

        # Do the operation
        ans = val1 - val2
        glob.stackmonad >> Stack.push(ans)
        return glob

class MultiplicationAction(ArithmeticAction):
    def __init__(self):
        pass

    def run(self, glob):

        glob.stackmonad >> Stack.pop() >> Stack.pop()
        op1 = glob.stackmonad.out[-1]
        op2 = glob.stackmonad.out[-2]

        # check the results
        val1, val2 = self.check(glob, op1, op2)

        # do the operation
        ans = val1 * val2
        glob.stackmonad >> Stack.push(ans)
        return glob

class DivisionAction(ArithmeticAction):
    def __init__(self):
        pass

    def run(self, glob):

        glob.stackmonad >> Stack.pop() >> Stack.pop()
        op1 = glob.stackmonad.out[-1]
        op2 = glob.stackmonad.out[-2]

        # check the results
        val1, val2 = self.check(glob, op1, op2)

        # do the operation
        ans = val1 / val2
        glob.stackmonad >> Stack.push(ans)
        return glob
```

A.4 config.py

```python
"""Exports the LICONF config object"""

class LICONF:
    """Global configuration container"""
    HAD_ERROR = False
    INTERACTIVE_MODE = True

    LOG_TYPE = 'FILE' # | 'CONFIG'
    LOG_LEVEL = 'DEBUG'
    LOG_TABLE = {'TRACE':0,'DEBUG':1,'INFO':2,'WARN':3,'ERROR':4,'FATAL':5}
    LOG_PATH = '/Users/valerie/Documents/Sixth-Form/Computer-Science/nea-full-repo/lilac-implementation/log/log.txt'


```

A.5 driver.py

```python
from . import *
from sys import exit

class Lilac:
    tree_m = TreeMachine()

    @staticmethod
    def start_prompt():
        open(LICONF.LOG_PATH, 'w').close()
        LICONF.INTERACTIVE_MODE = True
        print(f'{"- "*6}⊣ Lilac Interactive ⊢{" -"*6}')
        print(f'  Type $[q]uit to quit and $[h]elp to get help.\n')
        Lilac.run_prompt()


    @staticmethod
    def run_prompt():
        """Runs an interactive prompt"""
        user_in:str = ''
        while True:
            Lilac.tree_m.empty_stack()
            user_in = input(f'<i> ')

            if user_in[0] == '$':
                Lilac.do_interactive_command(user_in[1:])
                Lilac.run_prompt()

            scanner = Scanner(user_in)
            tokens = scanner.scan()

            if LICONF.HAD_ERROR:
                LICONF.HAD_ERROR = False
                Lilac.run_prompt()

            tokens_string = [f'{t.type}, {t.lexeme}' for t in scanner.scan()]
            # print(tokens)
            print(f'Tokens: {tokens_string}')
            tree = Parser.parse(tokens)

            if LICONF.HAD_ERROR:
                LICONF.HAD_ERROR = False
                Lilac.run_prompt()

            print(f'Tree: {tree.in_order()}\n')
            out = Lilac.tree_m.execute(tree)
            # print(PrettyPrinter.tree_to_mermaid(tree))
            # out = Lilac.tree_m.execute(tree)
            if out ≠ '':
                print(f'<o> {out}')

    @staticmethod
    def do_interactive_command(command: str) → None:
        command_args = command.split()
        cmd = command_args[0]
        args = command_args[1:]
        match cmd:
            case 'q' | 'quit':
                print('Leaving Lilac...')
                exit(0)
            case 'h' | 'help': Lilac.print_help(args)
            case 'i' | 'info': Lilac.print_info(args)

    @staticmethod
    def print_help(args: list[str]) → None:
        if not args:
            print(f'Lilac language repl cool :D')
        else:
            match args[0]:
                case 'author':
                    print(f'Lilac was built and designed by Valérie Thibault.')
                case 'help':
                    print(f'Real smart, nice one...')
                    print(f'You can use the help command followed by any other
                    ↪  command to learn about its usage.')
                case 'info':
                    print(f'Use info with any operator to see informtion about it.')
                case 'quit':
                    print(f'Use this to quit the interactive mode.')
                case _:
                    print(f'Lilac language repl cool :D')

    @staticmethod
    def print_info(args: list[str]) → None:
        if args is []:
            print(f'Well, what do you need info about?')
        else:
            rule = Grammar.bindings.get(args[0])
            if rule is None:
                print(f'Not an operator')
            else:
                print(f'({args[0]}) has a precedence of {rule["prec"]} and is
                ↪  associative in the {rule["assoc"]} direction.')


```

A.6 environment.py

```
from . import *

class Environment:
    def __init__(self):
        self.stackmonad = StackMonad(Stack())
```

```python
        self.table = {}

class EnvMonad:
    def __init__(self, env=None):
        if env is None:
            self.env = Environment()
        else:
            self.env = env
        self.trace = []

    def trace(self):
        string = ''
        for s in self.trace:
            string += s
        return string

    def consume(self, scope: Environment):
        """
        'Consumes' an environment, merging it with the current one.
        The stack of the second is pushed to the top, and the datatables are merged scope ont env.
        """
        newenv = Environment()
        newenv.stack_monad = StackMonad(Stack(Stack.join(scope.stack_monad.stack, self.env.stack_monad.stack)))
        newenv.table = self.env.table | scope.table
        return EnvMonad(newenv)
        # self.env.stack_monad = StackMonad(Stack(Stack.join(scope.stack_monad.stack, self.env.stack_monad.stack)))
        # self.env.table = self.env.table | scope.table

    @StatusHandler.logging('INFO')
    def bind(self, action):
        result = action.run(self.env)
        # Add to the trace the action being run
        self.trace.append(action.name())
        return result

    @StatusHandler.logging('INFO')
    def __rshift__(self, action):
        result = action.run(self.env)
        # Add to the trace the action being run
        self.trace.append(action.name())
        #if result is None:
            #return result
        #else:
            #return EnvMonad(result)
        return result
```

A.7 error_system.py

```python
from . import *
from sys import exit
from datetime import datetime


class ErrorType():
    SyntaxError = 'SyntaxError'
    NameUndefinedError = 'NameUndefinedError'
    NameRedefinitionError = 'NameRedefinitionError'
    OperatorUseError = 'OperatorUseError'
    OperandError = 'OperandError'
    TypeError = 'TypeError'

class StatusHandler:
    @staticmethod
    def logging(target_level):
        """Decorator which handles logging, can be used in front of any function"""
        def decorator(func):
            def wrapper(*args):
                if LICONF.LOG_TABLE[target_level] >=
                ↪  LICONF.LOG_TABLE[LICONF.LOG_LEVEL]:
                    log_msg =
                    ↪  f'[{target_level}][{datetime.now().strftime("%H:%M:%S:%f")}]
                    ↪  {func.__module__}.{func.__name__}'
                    if LICONF.LOG_LEVEL == 'TRACE':
                        log_msg += f' {PrettyPrinter.print_args(*args)}'
                    if LICONF.LOG_TYPE == 'CONSOLE':
                        print(log_msg)
                    elif LICONF.LOG_TYPE == 'FILE':
                        with open(LICONF.LOG_PATH, 'a') as file:
                            file.write(log_msg+'\n')
                output = func(*args)
                return output
            return wrapper
        return decorator

    @staticmethod
    def checkerror(func):
        """Decorator which checks if there has been an error."""
        def wrapper(*args):
            if LICONF.HAD_ERROR:
                return
            else:
                out = func(*args)
                return out
        return wrapper

    @staticmethod
    @logging('ERROR')
    def throw(type: ErrorType, line: int, message: str=''):
        """Called when an error is detected"""
        output = f'[Line {line}] {type}: {message}'

        if LICONF.INTERACTIVE_MODE:
            output = '<e> ' + output
            LICONF.HAD_ERROR = True
            print(output)
            return
        else:
            print(output)
            exit(64)

```

A.8 grammar.py

```
from . import TokenType

class Grammar:
    reserved_ids = {
            # 'fn'
            # 'let'
```

```python
            'in' : TokenType.IN,
            'true' : TokenType.TRUE,
            'false' : TokenType.FALSE
            }
    # operators = [
    #          TokenType.PLUS,
    #          TokenType.MINUS,
    #          TokenType.STAR,
    #          TokenType.SLASH,
    #          TokenType.COLON,
    #          TokenType.SPACE,
    #          TokenType.SEMI_COLON,
    #          TokenType.PIPE,
    #          TokenType.QUESTION,
    #          ]
    parens = [TokenType.LEFT_PAREN, TokenType.RIGHT_PAREN]
    bindings = {
            TokenType.COLON: { 'prec': 0, 'assoc': 'R' },
            TokenType.SPACE: {'prec': 10, 'assoc': 'L'},
            TokenType.ARROW: {'prec': 1, 'assoc': 'R'},
            TokenType.PIPE: {'prec': 2, 'assoc': 'L'},
            TokenType.QUESTION: {'prec': 2, 'assoc': 'R'},
            TokenType.SEMI_COLON: {'prec': 3, 'assoc': 'R'},
            TokenType.SLASH: {'prec': 8, 'assoc': 'R'},
            TokenType.STAR: {'prec': 8, 'assoc': 'R'},
            TokenType.PLUS: {'prec': 7, 'assoc': 'R'},
            TokenType.MINUS: {'prec': 7, 'assoc': 'R'},
            TokenType.EQUAL: {'prec': 6, 'assoc': 'R'},
            TokenType.LESS: {'prec': 6, 'assoc': 'R'},
            TokenType.GREATER: {'prec': 6, 'assoc': 'R'},
            TokenType.LESS_EQUAL: {'prec': 6, 'assoc': 'R'},
            TokenType.GREATER_EQUAL: {'prec': 6, 'assoc': 'R'},
            TokenType.OR: {'prec': 5, 'assoc': 'R'},
            TokenType.AND: {'prec': 4, 'assoc': 'R'},
            }
    operators = bindings.keys()
    unary = [TokenType.NOT, TokenType.MINUS]
    literal = [TokenType.NUMBER, TokenType.STRING, TokenType.TRUE, TokenType.FALSE]
```

A.9 mermaid_printer.py

```python
from . import *

class PrettyPrinter:
    @staticmethod
    def tree_to_mermaid(tree) -> str:
        output = 'graph TB\n'
        output += PrettyPrinter.mermaid_string(tree)
        return output


    @staticmethod
    def mermaid_string(tree) -> str:
        output = ''
        if not tree.is_leaf():
            output += f'{hash(tree.data)}["{tree.data.lexeme}"] --> {hash(tree.right.data)}["{tree.right.data.lexeme}"]\n'
            output += f'{hash(tree.data)}["{tree.data.lexeme}"] --> {hash(tree.left.data)}["{tree.left.data.lexeme}"]\n'
            output += PrettyPrinter.mermaid_string(tree.left)
            output += PrettyPrinter.mermaid_string(tree.right)
        return output

    @staticmethod
    def print_args(*args) -> str:
        output = ''
        for a in args:
            if isinstance(a, list):
                output += f'\t{[str(b) for b in a]}\n'
            else:
                output += f'\t{a}\n'
        return output


```

A.10 parser.py

```python
from . import *

class Parser:
    iterpointer = 0

    @staticmethod
    @StatusHandler.checkerror
    @StatusHandler.logging('DEBUG')
    def find_lowest_bound(expr):
        # returns the index of the operator with the least precedence in the expression
        # skips over any expressions in parentheticals
        in_paren = 0
        min_bind = 20
        min_index = 0

        for i in range(len(expr)):
            # skip parentheses
            if expr[i].type is TokenType.LEFT_PAREN:
                in_paren += 1
            elif expr[i].type is TokenType.RIGHT_PAREN:
                in_paren -= 1
            else:
                in_paren += 0

            if in_paren > 0:
                continue

            if expr[i].type in Grammar.operators:
                rule = Grammar.bindings[expr[i].type]
                if rule['prec'] < min_bind:
                    min_bind = rule['prec']
                    min_index = i
                elif rule['prec'] == min_bind:
                    if rule['assoc'] == 'R':
                        continue
                    elif rule['assoc'] == 'L':
                        min_index = i

        return min_index

    @staticmethod
    @StatusHandler.checkerror
    @StatusHandler.logging('TRACE')
    def clean_expression(expr: list[Token]) -> list[Token]:
        """Cleans an expression before it is parsed and checks for certain conditions."""
        if len(expr) >= 3:
            if expr[0].type is TokenType.LEFT_PAREN and expr[-1].type is TokenType.RIGHT_PAREN:
                expr = expr[1:-1]
        elif len(expr) == 2:
            if expr[0].type in Grammar.unary:
                expr = [Token(None, '', expr[0].line)] + expr
            elif expr[0].type in Grammar.operators:
                StatusHandler.throw(ErrorType.OperatorUseError, expr[0].line,
                            f'Operator {expr[0].lexeme} is missing a left operand.')
            elif expr[1].type in Grammar.operators:
                StatusHandler.throw(ErrorType.OperatorUseError, expr[0].line,
                            f'Operator {expr[1].lexeme} is missing a right operand.')
        return expr

    @staticmethod
    @StatusHandler.checkerror
    @StatusHandler.logging('DEBUG')
    def get_action(token) -> Action:
        match token.type:
            # case TokenType.COLON: return AssignAction()
            case TokenType.PLUS: return AdditionAction()
            case TokenType.MINUS: return SubtractionAction()
            case TokenType.STAR: return MultiplicationAction()
            case TokenType.SLASH: return DivisionAction()
            case TokenType.AND: return AndAction()
            case TokenType.OR: return OrAction()
            case TokenType.COLON: return AssignAction()
            case TokenType.IDENTIFIER: return IdentifierAction(token)
            case _:
                if token.type in Grammar.literal:
                    return LiteralAction(token)
                else:
                    return Action()

    @staticmethod
    @StatusHandler.checkerror
    @StatusHandler.logging('DEBUG')
    def to_tree(expr: list[Token]) -> Tree:
        """Converts a list of tokens into a Tree, which it returns"""
        index = Parser.find_lowest_bound(expr)
        action = Parser.get_action(expr[index])
        if [i for i in expr if i.type in Grammar.operators]:
            lexp = Parser.clean_expression(expr[:index])
            rexp = Parser.clean_expression(expr[index+1:])

            ltree = Parser.to_tree(lexp)
            rtree = Parser.to_tree(rexp)
            # print(Tree(expr[index], action, ltree, rtree))
            return Tree(expr[index], action, ltree, rtree)
        else:
            # print(Tree(expr[index], action))
            return Tree(expr[index], action)

    @staticmethod
    #@printlog
    @StatusHandler.logging('INFO')
    def parse(expr):
        if expr[-1].type is TokenType.EOF:
            return Parser.to_tree(Parser.clean_expression(expr[:-2]))
        return Parser.to_tree(expr)
```

A.11 scanner.py

```python
from . import *

class Scanner:
    """
    Scans the source code
    """

    # for printing the log
    # iterpointer = 0

    def __init__(self, source: str) -> None:
        self.source: str = source
        self.tokens: list[Tokens] = []
        self.start: int = 0
        self.current: int = 0
        self.line: int = 1

    # @printlog
    @StatusHandler.checkerror
    @StatusHandler.logging('INFO')
    def scan(self) -> list[Token]:
        while not self.at_end():
            # new token starts where the last one ended
            self.start = self.current
            self.scan_token()
            if LICONF.HAD_ERROR:
                return []

        # remove non-essential whitespace
        self.tokens += [Token(TokenType.EOF, '', self.line, '')]
        self.clean_tokens()
        return self.tokens

    # @printlog
    @StatusHandler.checkerror
    @StatusHandler.logging('DEBUG')
    def scan_token(self) -> None:
        """Scans a single token by consuming it and checking against combinations"""
        character = self.advance()
        # checks which token the current character is
        match character:
            # single character tokens
            case ':': self.add_token(TokenType.COLON)
            case ';': self.add_token(TokenType.SEMI_COLON)
            case '(': self.add_token(TokenType.LEFT_PAREN)
            case ')': self.add_token(TokenType.RIGHT_PAREN)
            case '+': self.add_token(TokenType.PLUS)
            case '*': self.add_token(TokenType.STAR)
            case '/': self.add_token(TokenType.SLASH)
            case '%': self.add_token(TokenType.MOD)
            case '?': self.add_token(TokenType.QUESTION)
            case '=': self.add_token(TokenType.EQUAL)

            # multi character tokens
            case '-':
                if self.match('>'):
                    self.add_token(TokenType.ARROW)
                else:
                    self.add_token(TokenType.MINUS)
            case '|':
                if self.match('-'):
                    self.add_comment()
                elif self.match('|'):
                    self.add_token(TokenType.OR)
                else:
                    self.add_token(TokenType.PIPE)

            case '&':
                if self.match('&'):
                    self.add_token(TokenType.AND)
                else:
                    StatusHandler.throw(ErrorType.SyntaxError, self.line,
                                  f'Unexpected character {character}')

            case '<':
                if self.match('='):
                    self.add_token(TokenType.LESS_EQUAL)
                else:
                    self.add_token(TokenType.LESS)

            case '>':
                if self.match('='):
                    self.add_token(TokenType.GREATER_EQUAL)
                else:
                    self.add_token(TokenType.GREATER)

            case '!':
                if self.match('='):
                    self.add_token(TokenType.NOT_EQUAL)
                else:
                    self.add_token(TokenType.NOT)

            # white space
            case ' ': self.add_token(TokenType.SPACE)
            case '\t': pass
            case '\r': pass
            case '\n':
                self.add_token(TokenType.NEW_LINE)
                self.line +=1

            # strings
            case '"':
                self.add_string()

            case _:
                if character.isnumeric():
                    self.add_number()
                elif character.isalpha():
                    self.add_identifier()
                else:
                    StatusHandler.throw(ErrorType.SyntaxError, self.line,
                                  f'Unexpected character {character}')
                    return

    @StatusHandler.checkerror
    @StatusHandler.logging('DEBUG')
    def clean_tokens(self):
        """Removes unecessary white space and new lines"""
        t = 0
        paren_count = 0
```

```python
        function_call = False
        while self.tokens[t+1].type is not TokenType.EOF:
            t += 1
            if self.tokens[t].type is TokenType.SEMI_COLON:
                function_call = True

            if self.tokens[t].type is TokenType.SPACE:
                if t == 0:
                    continue
                elif self.tokens[t-1].type is TokenType.IDENTIFIER\
                        and self.tokens[t+1].type in Grammar.literal + Grammar.parens:
                    function_call = True
                    continue
                elif function_call\
                        and self.tokens[t-1].type in Grammar.literal + Grammar.parens\
                        and self.tokens[t+1].type in Grammar.literal + Grammar.parens:
                    continue
                else:
                    del self.tokens[t]

            if self.tokens[t].type is TokenType.LEFT_PAREN:
                paren_count += 1
            elif self.tokens[t].type is TokenType.RIGHT_PAREN:
                paren_count -= 1

            if self.tokens[t].type is TokenType.NEW_LINE:
                function_call = False
                if paren_count == 0:
                    continue
                else:
                    del self.tokens[t]

    # @printlog
    @StatusHandler.logging('TRACE')
    def at_end(self) -> bool:
        """Checks if we are at the end of the source code"""
        if self.current >= len(self.source):
            return True
        else:
            return False

    # @printlog
    @StatusHandler.logging('TRACE')
    def advance(self) -> str:
        """Consumes a character in the string and advances"""
        c = self.source[self.current]
        self.current += 1
        return c

    # @printlog
    @StatusHandler.logging('TRACE')
    def add_token(self, type: TokenType, literal=None) -> None:
        """Adds a token to the list"""
        if type is TokenType.EOF:
            lexeme = ""
        else:
            lexeme = self.source[self.start:self.current]

        self.tokens.append(Token(type, lexeme, self.line, literal))

    # @printlog
    @StatusHandler.logging('TRACE')
    def match(self, character) -> bool:
        """Looks ahead at the next character and consumes it"""
        next = self.peek()
        if next == character:
           self.advance()
           return True
        return False

    # @printlog
    @StatusHandler.logging('TRACE')
    def peek(self) -> str:
        """looks at the next character without consuming it"""
        if self.at_end():
            return ''
        return self.source[self.current]

    @StatusHandler.logging('DEBUG')
    def add_comment(self) -> None:
        while True:
            character = self.advance()
            if (character == '-' and self.match('|')) or self.at_end():
                return
            else:
                continue

    @StatusHandler.logging('DEBUG')
    def add_string(self) -> None:
        """Adds a string literal, continues advancing until the string stops"""
        while self.peek() != '"' and not self.at_end():
            if self.peek() == '\n':
                self.line += 1
            self.advance()

        if self.at_end():
            StatusHandler.throw(ErrorType.SyntaxError, self.line,
                        'Unterminated string, did you forget a "?')
            return

        self.advance()

        value = self.source[self.start + 1: self.current - 1]
        self.add_token(TokenType.STRING, value)

    # @printlog
    @StatusHandler.logging('DEBUG')
    def add_number(self) -> None:
        """Adds a number"""
        is_float = 0
        # in a number the next character is either a number or a period
        while self.peek().isnumeric() or self.peek() == '.':
            if self.peek() == '.':
                is_float += 1
            if is_float > 1:
                StatusHandler.throw(ErrorType.SyntaxError, self.line,
                            'Incorrect number format, too many periods.')
                return
            else:
                self.advance()

        # string representing the number
        lexeme = self.source[self.start:self.current]
        # store as int or float depending on the type
        if is_float == 0:
            self.add_token(TokenType.NUMBER, int(lexeme))
        else:
            self.add_token(TokenType.NUMBER, float(lexeme))

    @StatusHandler.logging('TRACE')
    def is_id_char(self, character) -> bool:
        if character.isalpha() or character.isnumeric() or character == '_':
            return True
        return False

    @StatusHandler.logging('DEBUG')
    def add_identifier(self) -> None:
        """Adds an identifier token"""
        while self.is_id_char(self.peek()):
            self.advance()

        lexeme = self.source[self.start:self.current]
        if lexeme == 'True':
            self.add_token(TokenType.TRUE, True)
        elif lexeme == 'False':
            self.add_token(TokenType.FALSE, True)
        else:
            self.add_token(TokenType.IDENTIFIER)
```

A.12 stack.py

```python
from . import *

class StackMonad:
    """Monadic wrapper around a Stack, used for the pop and push operations"""
    def __init__(self, stack=None, out=[]) -> None:
        if stack is None:
            self.stack = Stack()
        else:
            self.stack = stack
        self.out = out

    def bind(self, f) -> 'StackMonad':
        result, out = f(self.stack)
        if result is None:
            return StackMonad(self.stack)
        elif out is None:
            return StackMonad(result)
        else:
            self.out.append(out)
            return StackMonad(result)

    def __rshift__(self, f) -> 'StackMonad':
        return self.bind(f)


class Node:
    def __init__(self, val=None, n=None):
        self.value = val
        self.next = n


class Stack:
    def __init__(self):
        self.top = Node()

    def __str__(self):
        cur = self.top
        msg = f'top: '
        while cur is not None:
            msg += f'{cur.value}\n     '
            cur = cur.next
        return msg

    @staticmethod
    def push(value):
        def inner_push(stack):
            if stack.top is None:
                stack.top = Node(value)
            else:
                stack.top = Node(value, stack.top)
            return stack, None
        return inner_push

    @staticmethod
    def pop():
        def inner_pop(stack):
            val = stack.top.value
            stack.top = stack.top.next
            return stack, val
        return inner_pop
```

A.13 token_type.py

```python
# from . import *
from enum import Enum, auto

class TokenType(Enum):
    """Enumerates the possible types of tokens"""
    COLON = auto()
    ARROW = auto()
    SPACE = auto()
    SEMI_COLON = auto()
    NEW_LINE = auto()
    LEFT_PAREN = auto()
    RIGHT_PAREN = auto()

    PLUS = auto()
    MINUS = auto()
    STAR = auto()
    SLASH = auto()
    DIV = auto()
    MOD = auto()

    OR = auto()
    AND = auto()
    NOT = auto()
    EQUAL = auto()
    LESS = auto()
    LESS_EQUAL = auto()
    GREATER = auto()
    GREATER_EQUAL = auto()
    NOT_EQUAL = auto()

    PIPE = auto()
    QUESTION = auto()

    STRING = auto()
    NUMBER = auto()
    IDENTIFIER = auto()
    TRUE = auto()
    FALSE = auto()
    IN = auto()

    EOF = auto()

```

A.14 tokens.py

```python
"""Tokens for representing source code"""

from . import *

class Token:
    """Token class which is used to represent a token in source code"""
    def __init__(self,
                 type: TokenType,
                 lexeme: str,
                 line: int,
                 literal=None) -> None:
        self.type = type
        self.lexeme = lexeme
        self.line = line
        self.literal = literal

    def __str__(self) -> str:
        return f'({self.type}: {self.lexeme}, {self.line})'

```

A.15 tree.py

```python
from . import StatusHandler

class Tree:
    def __init__(self, data, action, left=None, right=None) -> None:
        self.data = data
        self.left = left
        self.right = right

        self.action = action
        self.action.left = self.left
        self.action.right = self.right

    def __str__(self) -> str:
        if self.is_leaf():
            return f'{self.data}'
        return f"({self.data}, left:{self.left}, right:{self.right})"

    def post_order(self) -> str:
        string = ''
        if not self.is_leaf():
            string += self.left.post_order()
            string += self.right.post_order()
            string += self.data
        else:
            string = self.data
        return string

    def post_order_list(self):
        out_list = []
        if not self.is_leaf():
            out_list += self.left.post_order_list()
            out_list += self.right.post_order_list()
            out_list += self.data
        else:
            out_list = [self.data]
        return out_list

    def in_order(self):
        string = f''
        if not self.is_leaf():
            string += f'({self.left.in_order()}'
            string += self.data.lexeme
            string += f'{self.right.in_order()})'
        else:
            string = self.data.lexeme
        return string

    def is_leaf(self):
        return (self.left is None and self.right is None)


```

A.16 tree_actions.py

```
from . import *

class Action:
    """
    Default action class, exports 3 methods:
        check :: Action → bool
        run :: Action → Environment → Environment
        name :: Action → str
    """

    def __init__(self) → None:
        self.left: Tree = None
        self.right: Tree = None

    def check(self, *args) → bool:
        return False

    @StatusHandler.checkerror
    def run(self, glob: Environment) → Environment:
        return glob

    def name(self) → str:
        return f'Action'


class AssignAction(Action):
    def __init__(self) → None:
        super().__init__()

    def check(self) -> bool:
        if self.left.data.type is TokenType.IDENTIFIER:
            return True
        else:
            StatusHandler.throw(ErrorType.OperandError, self.left.data.line,
                                f'Cannot assign to something that is not an
                                ↪  identifier.')
            return False

    @StatusHandler.checkerror
    def run(self, glob: Environment) -> Environment:
        if not self.check():
            pass
        else:
            newenv = EnvMonad(glob)
            self.left.action.context = 'DEFINITION'
            newenv >> self.left.action
            glob = newenv.env
            glob.stackmonad >> Stack.pop()
            id = glob.stackmonad.out[-1]
            # if the right is an explicit function, should wrap in a scope
            if self.right.data.type is TokenType.ARROW:
                newenv = EnvMonad(glob)
                newenv >> self.right.action
                glob = newenv.env
                glob.stackmonad >> Stack.pop()
                scope = glob.stackmonad.out[-1]
                glob.table[id] = scope
            else:
                glob.table[id] = self.right
            glob.stackmonad >> Stack.push(Token(TokenType.IDENTIFIER, 'assigned', 1))

        return glob


class LiteralAction(Action):
    def __init__(self, value: Token) -> None:
        self.value = value
        super().__init__()

    @StatusHandler.logging('INFO')
    def run(self, glob: Environment) -> Environment:
        glob.stackmonad >> Stack.push(self.value)
        return EnvMonad(glob)

    def name(self):
        return f'Literal Action, literal is {self.value}'


class IdentifierAction(Action):
    def __init__(self, name: Token) -> None:
        self.id = name
        self.context = 'REFERENCE'
        super().__init__()

    # 'REFERENCE' or 'DEFINITION'
    def run(self, glob):
        if self.context == 'REFERENCE':
            if glob.table.get(self.id.lexeme) is None:
                StatusHandler.throw(ErrorType.NameUndefinedError, self.id.line,
                                    f'The name {self.id.lexeme} does not exist in the
                                    ↪  current scope.')
            else:
```

```python
                # glob.stackmonad >> Stack.push(self.id.lexeme)
                newenv = EnvMonad(glob)
                newenv >> glob.table.get(self.id.lexeme).action
                glob = newenv.env

        elif self.context == 'DEFINITION':
            if glob.table.get(self.id.lexeme) is not None:
                StatusHandler.throw(ErrorType.NameRedefinitionError, self.id.line,
                                    f'Trying to redefine {self.id.lexeme} even though it is already defined.')
            else:
                glob.stackmonad >> Stack.push(self.id.lexeme)

    def name(self):
        return f'Identifier Action, identifier is {self.id}'


class ArithmeticAction(Action):
    def __init__(self) -> None:
        self.left_val = None
        self.right_val = None
        super().__init__()

    @StatusHandler.checkerror
    @StatusHandler.logging('INFO')
    def check(self, glob: Environment, left: Token, right: Token) -> bool:
        if left.type is TokenType.IDENTIFIER:
            if glob.table.get(left.lexeme) is None:
                StatusHandler.throw(ErrorType.NameUndefinedError, left.line,
                                    f'The name {left.lexeme} does not exist in the current scope.')
                return False

            elif not isinstance(glob.table.get(left.lexeme), int):
                StatusHandler.throw(ErrorType.TypeError, left.line,
                                    f'The identifier {left.lexeme} does not return.')
                return False

            else:
                self.left_val = glob.table.get(left.lexeme)
        elif left.type is TokenType.NUMBER:
            self.left_val = left.literal

        if right.type is TokenType.IDENTIFIER:
            if glob.table.get(right.literal) is None:
                print(f'Name Undefined Error')
                return False

            elif not isinstance(glob.table.get(right.lexeme), int):
                print(f'Wrong type')
                return False
            else:
                self.right_val = glob.table.get(right.lexeme)
        elif right.type is TokenType.NUMBER:
            self.right_val = right.literal

        return True


class AdditionAction(ArithmeticAction):
    def __init__(self) -> None:
        super().__init__()

    @StatusHandler.checkerror
    @StatusHandler.logging('INFO')
    def run(self, glob: Environment) -> Environment:
        # Do the left and right actions
        newenv = EnvMonad(glob)
        newenv >> self.left.action
        newenv >> self.right.action

        # Extract the environment, then get the arguments
        glob = newenv.env
        glob.stackmonad >> Stack.pop() >> Stack.pop()
        left_op = glob.stackmonad.out[-1]
        right_op = glob.stackmonad.out[-2]

        is_allowed = self.check(glob, left_op, right_op)

        if is_allowed:
            result = self.left_val + self.right_val
            glob.stackmonad >> Stack.push(Token(TokenType.NUMBER, str(result),
            ↪  left_op.line, result))
        else:
            print(f'type error')

    def name(self) -> str:
        return f'AdditionAction with {self.left_val} and {self.right_val}'

class SubtractionAction(ArithmeticAction):
    def __init__(self) -> None:
        super().__init__()

    @StatusHandler.checkerror
    @StatusHandler.logging('INFO')
    def run(self, glob: Environment) -> Environment:
        # Do the left and right actions
        newenv = EnvMonad(glob)
        newenv >> self.left.action
        newenv >> self.right.action

        # Extract the environment, then get the arguments
        glob = newenv.env
        glob.stackmonad >> Stack.pop() >> Stack.pop()
        left_op = glob.stackmonad.out[-1]
        right_op = glob.stackmonad.out[-2]

        is_allowed = self.check(glob, left_op, right_op)

        if is_allowed:
            result = self.left_val - self.right_val
            glob.stackmonad >> Stack.push(Token(TokenType.NUMBER, str(result),
            ↪  left_op.line, result))
        else:
            print(f'type error')

    def name(self) -> str:
        return f'SubtractionAction with {self.left_val} and {self.right_val}'

class MultiplicationAction(ArithmeticAction):
    def __init__(self) -> None:
        super().__init__()

    @StatusHandler.checkerror
    @StatusHandler.logging('INFO')
    def run(self, glob: Environment) -> Environment:
        # Do the left and right actions
        newenv = EnvMonad(glob)
        newenv >> self.left.action
        newenv >> self.right.action

        # Extract the environment, then get the arguments
        glob = newenv.env
        glob.stackmonad >> Stack.pop() >> Stack.pop()
        left_op = glob.stackmonad.out[-1]
        right_op = glob.stackmonad.out[-2]

        is_allowed = self.check(glob, left_op, right_op)

        if is_allowed:
            result = self.left_val * self.right_val
            glob.stackmonad >> Stack.push(Token(TokenType.NUMBER, str(result),
            ↪  left_op.line, result))
        else:
            print(f'type error')

    def name(self) -> str:
        return f'MultiplicationAction with {self.left_val} and {self.right_val}'

class DivisionAction(ArithmeticAction):
    def __init__(self) -> None:
        super().__init__()

    @StatusHandler.checkerror
    @StatusHandler.logging('INFO')
    def run(self, glob: Environment) -> Environment:
        # Do the left and right actions
        newenv = EnvMonad(glob)
        newenv >> self.left.action
        newenv >> self.right.action

        # Extract the environment, then get the arguments
        glob = newenv.env
        glob.stackmonad >> Stack.pop() >> Stack.pop()
        left_op = glob.stackmonad.out[-1]
        right_op = glob.stackmonad.out[-2]

        is_allowed = self.check(glob, left_op, right_op)

        if is_allowed:
            result = self.left_val / self.right_val
            glob.stackmonad >> Stack.push(Token(TokenType.NUMBER, str(result),
            ↪  left_op.line, result))
        else:
            print(f'type error')

    def name(self) -> str:
        return f'DivisionAction with {self.left_val} and {self.right_val}'


class BooleanAction(Action):
    def __init__(self) -> None:
        self.left_val = None
        self.right_val = None
        super().__init__()

    @StatusHandler.checkerror
    @StatusHandler.logging('INFO')
    def check(self, glob: Environment, left: Token, right: Token) -> bool:
```

```python
        if left.type is TokenType.IDENTIFIER:
            if glob.table.get(left.lexeme) is None:
                StatusHandler.throw(ErrorType.NameUndefinedError, left.line,
                                    f'The name {left.lexeme} does not exist in the
                                    ↪  current scope.')
                return False

            elif not isinstance(glob.table.get(left.lexeme), int):
                StatusHandler.throw(ErrorType.TypeError, left.line,
                                    f'The identifier {left.lexeme} does not return.')
                return False

            else:
                self.left_val = glob.table.get(left.lexeme)
        elif left.type in [TokenType.TRUE, TokenType.FALSE]:
            self.left_val = left.literal

        if right.type is TokenType.IDENTIFIER:
            if glob.table.get(right.literal) is None:
                print(f'Name Undefined Error')
                return False

            elif not isinstance(glob.table.get(right.lexeme), int):
                print(f'Wrong type')
                return False
            else:
                self.right_val = glob.table.get(right.lexeme)
        elif right.type in [TokenType.TRUE, TokenType.FALSE]:
            self.right_val = right.literal

        return True

class AndAction(BooleanAction):
    def __init__(self):
        super().__init__()

    @StatusHandler.checkerror
    @StatusHandler.logging('INFO')
    def run(self, glob: Environment) -> Environment:
        # Do the left and right actions
        newenv = EnvMonad(glob)
        newenv >> self.left.action
        newenv >> self.right.action

        # Extract the environment, then get the arguments
        glob = newenv.env
        glob.stackmonad >> Stack.pop() >> Stack.pop()
        left_op = glob.stackmonad.out[-1]
        right_op = glob.stackmonad.out[-2]

        is_allowed = self.check(glob, left_op, right_op)

        if is_allowed:
            result = self.left_val and self.right_val
            if result == True:
                glob.stackmonad >> Stack.push(Token(TokenType.TRUE, str(result),
                ↪  left_op.line, result))
            elif result == False:
                glob.stackmonad >> Stack.push(Token(TokenType.FALSE, str(result),
                ↪  left_op.line, result))
        else:
            StatusHandler.throw(ErrorType.TypeError, left_op.line)

    def name(self) -> str:
        return f'AndAction with {self.left_val} and {self.right_val}'

class OrAction(BooleanAction):
    def __init__(self):
        super().__init__()

    @StatusHandler.checkerror
    @StatusHandler.logging('INFO')
    def run(self, glob: Environment) -> Environment:
        # Do the left and right actions
        newenv = EnvMonad(glob)
        newenv >> self.left.action
        newenv >> self.right.action

        # Extract the environment, then get the arguments
        glob = newenv.env
        glob.stackmonad >> Stack.pop() >> Stack.pop()
        left_op = glob.stackmonad.out[-1]
        right_op = glob.stackmonad.out[-2]

        is_allowed = self.check(glob, left_op, right_op)

        if is_allowed:
            print(self.left_val, self.right_val)
            result = self.left_val or self.right_val
            if result == True:
                glob.stackmonad >> Stack.push(Token(TokenType.TRUE, str(result),
                ↪  left_op.line, result))
            elif result == False:
                glob.stackmonad >> Stack.push(Token(TokenType.FALSE, str(result),
                ↪  left_op.line, result))
        else:
            StatusHandler.throw(ErrorType.TypeError, left_op.line)

    def name(self) -> str:
        return f'OrAction with {self.left_val} and {self.right_val}'
```

A.17 `tree_machine.py`

```python
from . import *

class TreeMachine:
    def __init__(self, tree=None):
        self.env_monad = EnvMonad()
        self.tree = tree

    @StatusHandler.logging('INFO')
    def execute(self, tree):
        self.env_monad >> tree.action
        # print(self.env_monad.trace)
        top = self.env_monad.env.stackmonad.stack.top.value
        if top.literal is not None:
            return top.literal
        else:
            return ''

    def empty_stack(self):
        self.env_monad.env.stackmonad = StackMonad()

```