Computer Science NEA

Lilac Programming Language

Name: Valerie Thibault
Candidate Number: 3253
Centre Number: 14613

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 minimal 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.

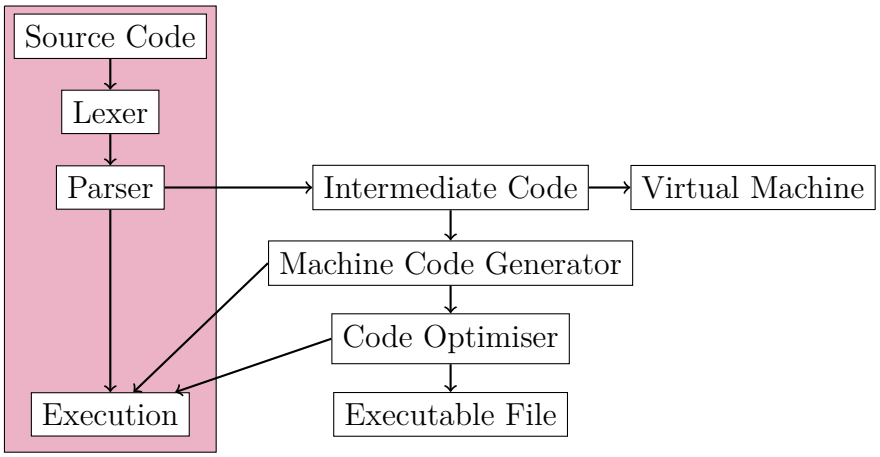

Figure 1: Graph showing the possible steps a translator might take.

Figure ?? 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'), (Assignment, ':'), (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:

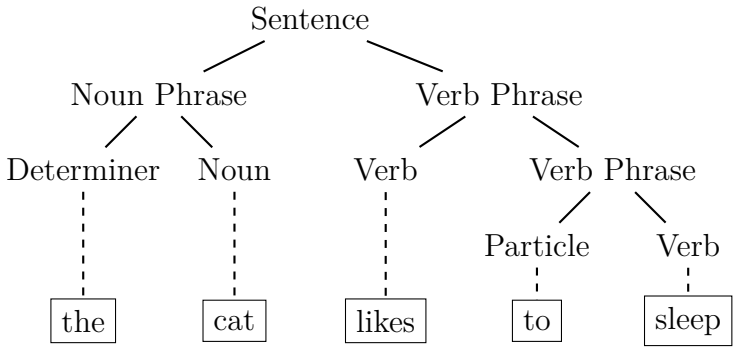

Figure 2: English parse tree of the sentence "the cat likes to sleep"

For a simple line of code in Lilac like `var: 4 * (2 + 3)` it would look like:

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 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:

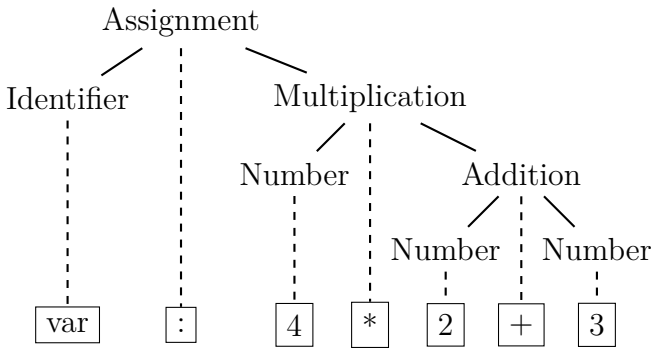

Figure 3: An example parse tree for Lilac

```
<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 ordder. 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:

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}`.

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

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

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.

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.

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 functio
print;add 3 2 = (print ((add 3) 2))
```

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] NameNotDefinedError: 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
```

gitPass

These will be used in conjunctino 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
```

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
```

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 reverse order traversal, and then using a simple stack/virtual machine. However, it is difficult to implement code branching and lay evaluation this way. This is due to the fact that it is a bottom up technique, so at the leaves 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 thhe `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 create 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 the actions for addition, multiplication, etc. will inherit from Arithmetic and will be able to use the specific check method. This also means I can do type checks on actions and