João Miguel Gago Gonçalves

Bachelor in Computer Science and Engineering

OCaml-Flat - An OCaml Toolkit for experimenting with formal languages theory

Dissertation to obtain the Master's Degree in

Computer Science and Engineering

Advisor: Artur Miguel Dias, Assistant Professor,
Faculty of Sciences and Technology,
NOVA University of Lisbon

Co-advisor: António Ravara, Associate Professor,
Faculty of Sciences and Technology,
NOVA University of Lisbon

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

July, 2019

Acknowledgments

This work was partially supported by the Tezos foundation through a grant for the project “FACTOR - A Functional Programming Approach to Teaching Portuguese Foundational Computing Courses”.

Abstract

Due to its fairly formal nature, the teaching of the subject of computation theory often presents itself as a major obstacle for computer students in general. The academic community has for some time been aware of this situation. Historically there have been developed many tools for the teaching of theory of formal languages and automata (FLAT).

Despite the existence of a considerable number of them, occasionally a new tool emerges that contributes with something new, or some existing tools are extended with new functionalities.

We propose to develop a library of functions in OCaml that support various concepts of FLAT, namely the definition of finite automata, regular expressions, pushdown automata, context-free grammars and Turing machines. We want to provide code that, whenever possible, follows closely the formalization of the concepts studied by the students.

Another goal is to provide support for using this system inside Mooshak.

Finally, an interesting technical problem we will need to handle is the nondeterminism and nontermination in parts of the code.

In this report, first we discuss the properties of functional languages and why they are indicated for our project. Next, we give a small introduction to the FLAT concepts and discuss some issues about their implementation. Finally, there is a small review on the existing FLAT pedagogical tools.

Keywords: Formal Language And Automata Theory, Functional Programming, OCaml Language, Pedagogical Tools

Resumo

Devido à sua natureza bastante formal, o ensino do tema da teoria da computação muitas vezes apresenta-se como um obstáculo para os estudantes de informática em geral. Há algum tempo que a comunidade académica tem conhecimento desta situação. Historicamente, foram desenvolvidas muitas ferramentas para o ensino da teoria das linguagens formais e automatos (FLAT).

Apesar da existência de um número considerável deles, ocasionalmente surge uma nova ferramenta que contribui com algo novo, ou algumas ferramentas existentes são ampliadas com novas funcionalidades.

Propomos desenvolver uma biblioteca de funções em OCaml que apoiam vários conceitos de FLAT, ou seja, a definição de autômatos finitos, expressões regulares, autômatos de pilha, gramáticas livres de contexto e máquinas de Turing. Queremos fornecer código que, sempre que possível, é fidedigno à formalização dos conceitos estudados pelos alunos.

Um outro objetivo é fornecer o apoio para usar este sistema dentro com o Mooshak.

Finalmente, um problema técnico interessante que nós precisaremos de assegurar é o não determinismo e a não terminação em partes do código.

Neste relatório, primeiro discutimos as propriedades das linguagens funcionais e por que elas são indicadas para o nosso projeto. Em seguida, damos uma pequena introdução sobre conceitos FLAT e discutimos algumas questões sobre a sua implementação. Finalmente, é feita uma pequena revisão sobre as ferramentas pedagógicas de FLAT existentes.

Palavras-chave: Teoria Das Linguagens Formais E Automatos, Programação Funcional, Linguagem OCaml, Ferramentas Pedagógicas

Contents

List of figures

1. Introduction

The teaching of formal language and automata theory is a staple of most computer science programs due to both its importance for direct application of the taught concepts in a professional setting that may require using those concepts, and its utility in molding the student's minds to better critically think on how to solve the computer-science related problems that may occur during their academical and professional careers [1]. As such, there exists a continuous need for better methods of teaching these concepts. In response to this need, we can look to both new technologies and old methodologies to better understand how to contribute with new exciting solutions that can make formal language and automata theory more accessible to future students.

1.1 Context

Historically, for a long time, the academical community has realized the utility in developing and using helper tools for teaching formal language and automata theory, and during the years it has been shown that students tend to fair better when they actively use these tools as opposed to not being given the opportunity to use them. As such, a diverse pool of pedagogical tools have been developed over the years [2] in hopes of contributing to the cause, but curiously it has been observed that while some tools offer a very complete range of functionalities, it is when using a variety of different tools with even a few key distinguishing features that students obtain greater insight into the fundamentals of the subject [3].

This MSc thesis will consist of developing, using the OCaml language, a pedagogical tool called OCaml-Flat. It is a library of types and functions that can be used as a tool for students' personal study, for the integration in a testing environment (Mooshak) with various exercises for both in-class and home study, and for integration with a WEB application with interactive graphics.

The functional paradigm was chosen for the implementation of this project because code written in this style is often very legible, concise and easy to understand without much mental fortitude, all properties that are heavily desired if one of the objectives is for the students to read and understand the code.

The tool is planned to be used in future editions of the discipline of computation theory from the Faculty of Sciences and Technology of NOVA University of Lisbon (FCT/UNL). As such it will be developed following the formalisms adopted in the discipline as faithfully as possible.

1.2 Expected Contributions

A very clear implementation in OCaml of a set of generators and language recognizers.

Whenever possible, the code should follow closely the formalization of the concepts as studied by the students.

Cater for an extensible design. In particular, it is important to identify shared features among the mechanisms and to factorize the corresponding code.

Find reasonable ways to deal with the nondeterminism and nontermination of some operations.

The toolkit should present itself as an OCaml module, intended to be used in the context of the OCaml interpreter. Developing a graphical interactive environment is outside the scope of this project.

1.3 Document Structure

This document is organized in the following chapters:

Chapter 1 - Introduction to the problem, its context, the proposed solution and the expected contributions.

Chapter 2 - Detailed presentation of what is functional programming, explaining key aspects such as its history, characteristics, advantages, disadvantages and examples of how it is used to solve problems.

Chapter 3 - Small introduction to the main concepts of formal language and automata theory, and discussion of some issues about their implementation.

Chapter 4 - Survey on the existing pedagogical tools, their utility, their characteristics and history.

Chapter 5 - Work plan for how this project will be approached.

2. Functional Programming

The functional programming style was born form the acknowledgement that it is possible to express computation by only resorting to mathematical functions, applying functions to arguments and evaluating expressions [4]. In functional languages, functions play a central role where they are treated as first-class values, as we will explain further down this paper.

Regardless of its base ingredient's simplicity, functional languages help programmers in expressing their ideas with better clarity and certainty than with other programming styles, such as the imperative style for example.

2.1 History

In the early 1930s, even before the invention of what is widely considered as the first programmable computer, mathematician Alonzo Church introduced what could be considered the first functional language, the lambda-calculus [5]. During his research in the field of foundations of mathematics, Church was investigating a way of defining a different basis for mathematics built on functions, rather than sets, as a way of expressing the computational aspect of functions. Its influence on functional programming has had such impact, it most often represents the bases for modern functional languages [6].

In 1958 John McCarthy, during his work in MIT, gave origin to what is widely considered the first ever functional programming language, Lisp. According to "Conception, Evolution, and Application of Functional Programming Languages" by Paul Hudak, even though lambda-calculus didn't actually influence Lisp much, both made use of Church's lambda notation [6]; beyond that not much similarities are found between the two languages. The project's aim was for programming symbolic data computation.

During the years that would follow, the functional programming community would continue to grow, and many new functional languages would be developed, which would push the understanding of functional concepts. Some of these new languages would include IPL, APL, ML (which would later originate OCaml), SASL, KRC, and Miranda.

It was around the 1980s that 2 of the most important functional languages of today were born – OCaml and Haskall. The first language uses strict evaluation while the second uses non-strict evaluation. In the case of OCaml, its origins are found in the LCF Robin Miller test system, which dates from 1962. The language began to be used as an Autonoma programming language from 1981 onwards, having evolved and gained new implementations. OCaml began to gain popularity and attract many programmers in the late 1990s. In the case of Haskell it was created in a rather deliberate way through a committee with the mission of creating a common language for the non-strict functional programming community [7].

Since then, new functional languages have been developed, many of them, like Scala or F#, support not just the functional paradigm, but also other paradigms, mainly imperative and object-oriented; some languages, like Pearl and PHP, while not designed specifically for the functional paradigm, support some functional mechanisms.

Even though its beginnings are rooted in the academical, and with the imperative paradigm remaining as the principle way in which most programmers today code [8], functional programming has been rising in mainstream popularity in the industry and commercial settings, with many famous applications such as Facebook, WhatsApp and Twitter running functional code, especially in their server side.

2.2 Characteristics

Functional programming displays several core characteristics, namely:

2.2.1 High-order functions

One useful mechanism that is essential to functional programming are the high-order functions. High-order functions can receive functions as arguments and may also return new dynamically generated functions. A great example of this is the map function that applies a function to each element of a list.

```
let rec map f l =
     match l with
          [] -> []
        | x::xs -> ( f x )::map f xs
```

Another example is the `gen` function that receives an integer `n` and generates a new function dependent on `n`.

```
let gen n = fun x -> x + n
```

Being able to program well in the functional style involves knowing how to use high-order functions and in particular, recognizing good opportunities for using library defined high-order functions, such as `map`, `flatMap`, `filter`, `exists`, `partition` and others.

2.2.2 First-class functions

A programming language is said to have first-class functions if the functions have a status as important as the other predefined types, such as integer or real numbers. First-class functions are a requirement for functional languages.

Concretely, in a functional language the functions can: (1) be passed as an argument for other functions; (2) be returned by other functions; (3) be used as constituent elements of data structures; (4) have specific literals for representing anonymous functions. Points (1) and (2) show that without first-class functions we could not have high-order functions, for these pass the others as arguments and results.

Permitting to treat functions as normal data has positive consequences at the level of program sophistication and the ideas that can be expressed in a natural way. In this first simple example, we have a function that implements function composition – a new function is generated using two existent functions.

```
let compose f g = fun x -> f (g x)
compose : ('a -> 'b) -> ('c -> 'a) -> ('c -> 'b)
```

In this second example, we show a classic representation of sets using only functions. It is known that a set is an identity whose main characteristic is the possibility of knowing if a value belongs to it or not. Thus, we can represent each set with a Boolean function: that function, when applied to a value produces true I the value belongs to the set and false if it does not. It is known as the feature function of the set:

Empty set:

```
let set0 = fun x -> false
```

Universal set:

```
let setu = fun x -> true
```

Singular set constructor. Notice that we are representing an infinite set without any problems:

```
let set1 x = fun y -> y = x
```

Belongs to test:

```
let belongs v s = s v
```

Set union:

```
let union s1 s2 = fun x -> s1 x || s2 x
```

2.2.3 **Referential transparency**

Pure functional programs are referentially transparent, meaning that everything which happens during the execution of a program literally depends on the text of the program, and there is no hidden entity (e.g. the imperative state) to influence the execution of the programs.

Referential transparency is a property that allows replacing, in the text of the program, an expression for any other expression that evaluates to the same result, without changing the behavior of the program. Referential transparency is an important principle in mathematics, very much implicitly used in demonstrations.

Without referential transparency, it is very hard to be sure that an expression can be replaced by another. To give an example, in general the following two expressions cannot be considered as equivalent:

```
f() + f()          2 * f()
```

With referential transparency, a function does not produce side-effects and returns always the same result for the same inputs. This property also helps immensely if we need to parallelize our programs.

2.2.4 **Recursion**

In the functional world, repetition is expressed using recursion [12], as opposed to the imperative world where repetition is expressed with iteration.

In functional languages, programmers who pay special attention to code clarity and legibility, use recursion as base for a programming technic which involves reducing each problem to a simpler version of the same problem. It is thus an inductive technic and the resulting functions are inductive. Here is a known example of an inductive function:

```
let rec fact n = if n = 0 then 1 else n * fact (n-1)
```

In any case, functional languages can also use recursion to simulate iteration. Simulated iteration forces the reader of the code to think in a way that is considered less human-like and more machine-like, but which allows for better efficiency, if it is strictly necessary. The gain in efficiency results from the fact that the generality of functional languages being able to optimize the simulated iteration without consuming execution stack (tail recursion optimization). Example:

```
let rec factX n r =
     if n = 0 then r else factX (n-1) (r*n)
let fact n = factX n 1
```

2.2.5 **Declarativity**

The philosophy behind declarative programming is to provide an abstraction with which programmers could write and/or read code and understand it’s goal without the need to “run the algorithm in their heads”, but rather to express the essence of what that code is trying to achieve, almost as if the programmers were “telling the program what they want it to do, without step-by-step instructions”.

The first version of the fact function from the previous is declarative. The function expresses a truth – `fact n = n * fact (n-1)` – that the machine uses to produce the correct results.

The second version of the function is not declarative because it describes with minuteness, step-by-step, a process of calculating the result. Caution, it is possible to mathematically prove that both versions of the function are equivalent and from the mathematical viewpoint it may not be of much relevance distinguishing the two forms. However, for a human, the declarative form is relevant: functions become simpler to invent, to understand, and it becomes simpler to intuitively argument over the correctness of functions.

There is a diverse category of languages that support declarative languages (e.g. logical languages, restriction languages, HTML), with the functional languages category also included in this group. This aspect is widely considered to provide with clear, simple to read code that we think makes functional programming the paradigm of choice for the implementation of our project.

2.2.6 **Static typing**

There are functional languages with dynamic typing (List, Scheme, Lua) and there are functional languages with static typing (OCaml, Scala, Haskell). In the case of languages with static typing, the generality supports type inference, with each declaring the argument's type being optional.

2.2.7 **Evaluation strategy**

Functional languages can be divided into those who use strict evaluation and those who use non-strict evaluation. The difference is that with strict evaluation each function call, the arguments are always evaluated before the call is executed. With non-strict evaluation however, the arguments are always passed unevaluated, and are evaluated inside the function only when their values are needed.

Haskell is an example of a functional language that uses non-strict evaluation. OCaml uses strict evaluation, albeit there are available non-strict mechanism in the data type Stream and the lazy module.

2.2.8 **Algebraic data types and pattern-matching**

Most functional languages, such as OCaml, support the definition of algebraic data types, which are typing made from diverse variants. For example, the next type, which defines lists of values, possesses two variants: `Nil` for empty lists and `Cons` for non-empty lists.

```
type 'a list = Nil | Cons of 'a * 'a list
```

It is normal the existence of operations that only apply to values of certain variants. For example, for lists, the operation for obtaining the tail only applies to non-empty lists.

The mechanism of pattern-matching allows for dealing with the various variants of an ADT in a practical, elegant and type-safe matter. The mechanism is quite sophisticated: To start with, it introduces a notion of pattern – a pattern is a special expression with intuitive syntax that represents a set of values. When we verify pairing between a value and a pattern, some of the value's components become immediately available through the pattern's variables. The human being is accustomed in using patterns in its interaction with the real world. The patterns of functional languages help in turning programs more legible and easier to write.

In the case of OCaml, pattern pairing is implemented in the construction of "match". Here is a small example, where only two patterns are used (occurring to the left of the arrow). The function tests if the letter 'a' occurs in a list of characters.

```
let containsA list =
      match list with
```

```
                 Nil -> false
               | Cons(x,xs) -> x = 'a' || containsA xs
```

2.2.9 Imperative mechanisms

It may seem strange, but the generality of functional languages is not pure, which means that they include imperative mechanisms, including state. In the world of functional programming, programs are mostly written in a functional manner, but imperative mechanisms are used in specific, well justified situations.

For example, if the problem involves state, it is best to deal with it using interactive mechanisms, as opposed to trying to simulate them with functional mechanisms. Think of a calculator with registers: it is best to represent the registers using a set of mutable cells. If a language provides appropriate linguistic mechanism to deal directly with the situation, then it is best to use those mechanisms.

Another situation: there are certain operations that are inefficient in their functional way and it may be worthwhile to think of imperative alternatives. For example, adding a value to the end of a list, causes implicit duplication of that list. In the case of an absurdly grand list, it may be advised to think twice.

2.3 Advantages

Compared to imperative programming, the functional paradigm displays certain properties that give it some advantages [9].

Pure functional programming precludes the notion of state as such it only really receives an input and produces an output. This can be of great help in a variety of aspects, including legibility, parallelization and special technics such as currying.

Since a functional program doesn’t have mutable variables or state, and with the added characteristics of higher-order function and declarativity, it is possible to write code that is considerably more succinct and legible compared to other paradigms; because there are no variables or side-effects one must keep track of while mentally thinking of the code’s execution, this allows the programmer to better concentrate on what they want to compute instead of how they will compute it, which will lead to safer, more bug-free code.

Another interesting advantage granted by the lack of states and side-effects, and the irrelevance of the order in which a program executes its functions for the computing of its output, is that it allows for easier program parallelization, since the possibility of a function interfering with the output of another function running concurrently is inexistent.

Thanks to considering all functions as first-class (that is, they can be passed as arguments to other functions, including themselves), functional programming allows for the use of certain coding technics such as currying, a technique that allows programmers to work with functions that take multiple arguments, and use them in settings where functions might only take one argument. This allows not only for better code legibility, but in some cases, code that is more efficient.

Lazy evaluation and data immutability can in some specific cases increase efficiency by allowing the compiler to perform less strict evaluations on its expressions.

Higher order functions capture generical code patterns (e.g. map, filter, etc.), which help program in a concise way, without constant repetition of the same formulas.

2.4 Disadvantages

Although many benefits are to be gained from using the functional paradigm, it isn't without its flaws [9].

The real world is imperative, and the notion of state is useful to better express various real-world problems in code, such as a calculator with memory registers or accessing an external database. While most functional languages (even pure ones) allow for methods to simulate states, these can often-times break legibility and conciseness of the code, thus losing the benefits of not using states.

Another problem is the typically less efficient use of CPU and memory management, in large part due to the lack of mutable data structures whose implementations translates better into various hardware. Not only that, but the lack of state forces us to continually declare and create objects instead of assigning new values to already existing ones, which also leads to the need for more memory in our programs.

In languages such as Haskell that use lazy evaluation, there can be the occurrence of memory leaks. There are some special techniques to deal with them.

2.5 Examples of functional programming

We now show two small examples that illustrate the practical use of some features of these languages. These examples include the use of algebraic data types, pattern-matching and recursion.

This section also prepares to the next chapter where we will discuss the recognition of a word by a non-deterministic finite authenticity and this requires the use of the breath-first strategy.

2.5.1 Binary search tree example using depth-first

In OCaml, we can define a binary tree [10] using the following ADT:

```
Type α tree = Nil | Node of α * α tree * α tree
```

In the example below, `belongs1` is a function that receives a value "val" and a binary tree "tree", and checks if there is a node in the "tree" whose value equals val. This function was written using intuition and aiming for simplicity. Not surprisingly, in the end we verify that the function implements a depth-first algorithm [11].

```
let rec belongs1 val tree =
     match tree with
        Nil -> false
        | Node(x,left,right) -> val = x
        || find val left || find val right
```

Notice how the function is declarative. What this code affirms is the following: (1) the value cannot occur in an empty tree; (2) for the value to occur in a non-empty tree, either it occurs at the root, or at the left sub-tree, or at the right sub-tree. If we wish to analyze the operational effects of this function's execution, we see that first we test if the value occurs at the root; if not then we search for the value in all of the left sub-tree; only if the value was not found until this point do search in the right tree. Thus, we show how the function is depth-first.

2.5.2 **Binary search tree example using breadth-first**

The following example resolves the same problem as "belongs1" but now using a breadth-first strategy. In breadth-first, the search works on the nodes of the tree at each horizontal level at a time, starting on the root, only moving to the next level if the value is not found in the previous level. The presented solution has a certain degree of artificiality because it is necessary for a way of representing the notion of horizontal level. To represent each horizontal level, we use a list of trees, and even in the first call we need to pass the original tree inside a list.

```
let rec belongs2 val Level =
     match level with
          [] -> false
        | Nil::ls -> find val ls
        | Node(x,l,r)::ls ->
              x = val || find val (ls@[l,r])
```

It is pertinent to consider three cases relative to the first argument: (1) if the list is empty, certainly the value does not occur; (2) if the NIL tree is at the head of the list, that tree can be ignored for not having any element; (3) if a non-empty tree is at the head of the list, if the value occurs at the root of that first tree the value is considered found, if it does not occur then it is necessary to check if it occurs in the rest of the list with the two sub-trees added at the end.

Since in the third case the node's successors are added to the end of the list instead of the start, the function evaluates the tree in breadth-first.

3. Formal language and automata theory

For this project, we will be adhering to the formalisms and naming conventions adopted in the theory of computation class of the computer science engineering course in the Faculty of Sciences and Technology, NOVA University of Lisbon. This is important because it will allow us to maintain coherence between what the students learn during the class and what they might learn or revise using our toolkit, minimizing the student's efforts in adapting their knowledge from the classes to their usage of our toolkit in an attempt to advance their understanding of formal language and automata theory.

Theory of computation is a branch of computer science and mathematics that studies the properties of computation. Among other aspects, it also studies which problems can be solved by a computer and among these which can be programmed efficiently.

In his book "Introduction to the theory of computation", Michael Sipser [1] divides the subject into three main branches: automata and languages, computability theory and complexity theory. Due to the objectives of our project, we only wish to elaborate on the automata and languages branch.

3.1 Chomsky Hierarchy

Before diving into explaining the main FLAT concepts we will discuss for our project, we think it is of interest to present the following concept, the Chomsky hierarchy [1].

In FLAT, a formal grammar is a set of rules for producing strings in a formal language. According to Chomsky, we can divide these grammars into four groups based on the type of languages they can generate. Note that the levels with the lowest number identifier represent the languages that require more capable recognition mechanisms and have more general generation rules.

The hierarchy is as follows: Type 0 grammars generate recursively enumerable languages, that is, languages whose words can be generated by a computationally universal machine; Type 1 grammars generate context-sensitive languages; Type 2 generates context-free languages; Type 3 generates regular languages. Every regular language is context-free, every context-free language is context-sensitive, every context-sensitive language is recursively enumerable.

Not all context-free languages are regular, and the same logic could be expressed to the rest of the discussed languages.

The following section will describe the key FLAT concepts we pretend to cover with our project, as well as establish the terminology used throughout the framework.

3.2 FLAT concepts covered in the project

3.2.1 Regular expressions

Regular expressions are special expressions which represent languages whose words have a simple structure. By starting with a finite number of words and then applying regular operations such as union, catenation and iteration closure, we can define regular expressions. They can be seen as language generators but are less expressive than context-free grammars.

An example of a regular expression could be `a(a + b)*,` which denotes the language of all the words over the alphabet {a, b} starting with an “a”.

In the real world, regular expressions are used in search engines, lexical analysis, word processors, text editors and others; many programming languages even provide regular expression capabilities.

3.2.2 Finite automata

Finite Automata is a simple mathematical model of computation that can be used to specify languages. The expressive power of the model is relatively weak and is only capable of describing languages with very regular structure. Nevertheless, it is a useful model with many theoretical and practical applications.

In this model, the specification of each language is accomplished via recognition. Each particular Finite Automaton (FA) recognizes some language in the sense that the FA checks whether a word belongs or do not belongs to the language.

Finite automata are formally defined by the following 6-tuplet: a finite set of states; a finite set of symbols (an alphabet); a list of transitions where each transition describes the progression between two states by consuming a symbol; one initial state; and zero or more accepted states (states that symbolize the success of a computation).

Finite automata are regular language recognizers, as they are able to recognize all languages that can be obtained from a regular expression and are less capable than pushdown automata. For example, the automaton in Figure3.1 recognizes the previously discussed language denoted by all words over {a, b} starting with an a.

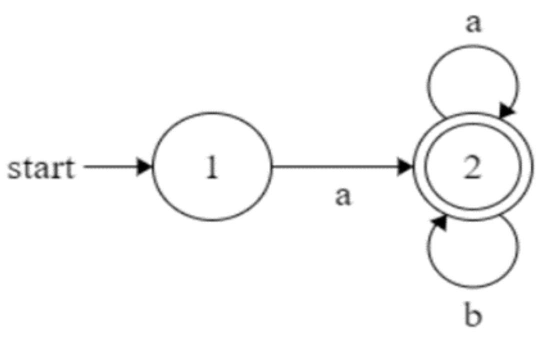

Figure 3.1-Deterministic Finite Automata

These automata can be either deterministic or nondeterministic, where determinism in this context means that for each state a symbol can only transition to one state, whereas in nondeterminism, transitions can happen from a state to one or more states using the same symbol, and some states can also transition to another state without consuming any symbol. In this project we will be working with both deterministic and nondeterministic finite automata.

As contrast to the previous example which was deterministic, here is an example of a non-deterministic finite automaton:

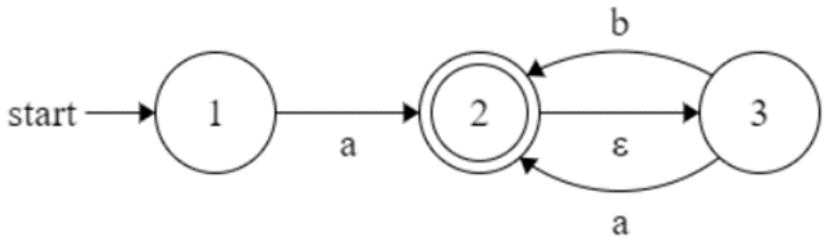

Figure 3.2- Non-Deterministic Finite Automata

Notice how the automata can transition from state 2 to 3 without consuming any symbol, thus making it non-deterministic.

Finite automata are widely used in software engineering, compilers, network protocols, as well as in other areas not strictly to due with computer science, such as philosophy, biology and linguistics.

3.2.3 Context-free grammars

Context-free grammars are a type of formal grammars used to describe all possible strings in a specified formal language, more precisely, they are context-free language generators, and are represented as the following 4-tuplet: a non-empty finite set of nonterminal characters, each representing a part of a sentence; a set of terminals different from the first set, that defines the alphabet of the language; a set of rules composed of a character and a string of characters and terminals; and the start symbol, an element of the first set that represents the entire sentence.

Here is an example of a context-free grammar that generates the language of all words over the alphabet {a, b} that have an equal number of both letters a and b, and where all occurrences of “a” are at the left of all occurrences of “b”:

```
P -> ε | aPb
```

Notice how P calls itself, indeed recursion plays an important role in defining context-free grammar.

Context-free grammars are mainly used for describing structures for programming languages and are also used in linguistics to describe the structure of sentences and words in natural languages.

Context-free grammars correspond to a more powerful mechanism of generating languages than that of regular expressions. There is no regular expression equivalent to the previous grammar.

3.2.4 **Pushdown automata**

A pushdown automaton is a type of automata that uses a stack, can be both deterministic or non-deterministic, and are expressed through the following 7-tuplet: a finite set of states; a finite set for the input alphabet; a finite set for the stack alphabet; a finite set representing the transition relation; a starting state; the initial stack symbol; and a set of accepted states. Compared to finite automata, they can handle theoretically infinite amounts of data. A word is accepted by a push-down automaton only if it leads to both a state of acceptance and the stack being empty.

They are language recognizers. It is interesting to note that non-deterministic pushdown automata are more powerful than the deterministic ones. This means that there are some context-free languages that are only recognized by a non-deterministic automaton. In general pushdown automata are more powerful than finite automaton but less powerful than Turing machines.

Here is the example of a push-down automaton for the language previously given as example in the context-free grammar section.

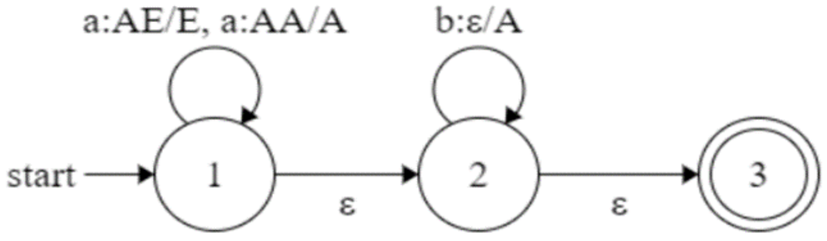

Figure 3.3-Pushdown Automata

Notice that for every “a” consumed we add it to the stack, while for every “b” we remove an “a” from the stack, thus the word needs to have the same number of occurrences of a and b to be accepted.

Their main practical applications involve parser designing, but are also used in other areas such as online transaction systems.

3.2.5 **Turing machines**

A Turing machine is a computation model that consists of a single read and write control unit and an infinite tape where the unit can write or read symbols. The tape is initialized only with the initial input and the rest is blank. In the lectures, the Turing machine presented is stack-based, which is represented by the following 6-tuplet: a finite, non-empty set of control states; an input alphabet which keeps track of process data; a starting state; the initial memory; a transition relation that specifies the rules followed by the computation; and a list of final states.

Due to its relative complexity compared to the other four concepts, we decided to omit an example for a Turing machine.

A Turing machine is a formal language recognizer, that is, it recognizes languages generated by any grammar, including unrestricted grammars. They are used in a variety of computer science fields, including algorithmic complexity theory, machine learning, software engineering, computer networks and evolutionary computations.

3.3 **Implementing FLAT concepts in the functional paradigm**

During the implementation of our project, there will be various properties we wish for our end product to possess. The most important aspect of our code will be for it to, whenever possible, be as legible, intuitive and faithful to the formalisms adopted by the computation theory class, even if it means sacrificing some efficiency.

For some problems the most natural translation of its solution into functional code will produce the desired outcomes and be very easy to understand. But there are other problems where the natural solution enters an infinite cycle, and so we need to write less institutive code using special techniques.

To better demonstrate this point, as well as give a general impression of the challenges that will be faced during this project, we will showcase two examples involving finite automata. The first problem has a solution that is simple and intuitive and will follow the lectures closely; for the second problem the intuitive solution enters an infinite loop, forcing us to resort to less intuitive techniques.

To start we need to introduce an OCaml representation for finite automata. A finite automaton is characterized by an initial state (represented by a simple name), a set of transitions between states, labelled by symbols, and a set of acceptance states. We also need a representation for words, and we will use a simple character list. The following code represents the definition of an automata.

```
type symbol = char
type word = symbol list
type state = string
type states = state list
type transition =
         state
        * symbol
        * state
type transitions = transition list
type fAutomaton = {
        initialState: state;
        transitions: transitions;
        acceptStates: states     }
```

3.3.1 Reachable example

An example of a typical FLAT problem that translates nicely into functional code that is intuitively understood is the reachable problem, where given an initial state and the list of transitions for an automaton, we want to obtain all states reachable from the initial state (including the initial state itself).

Before we write our main function however, we need to define the auxiliary functions `gcut`.

Without bothering too much about the implementation details of said function, the function `gcut` receives as arguments a specified state and list of transitions for an automaton, and returns the pair formed by the list of transitions starting from that state, and the list of all remaining transitions (those that do not start from that state.

Here is the OCaml code for the reachable function as stated before, which receives as arguments a state s and the list of transitions t for an automaton, and computes the list of states reached by the state s through transitions in t:

```
let rec reachable s t  =
     let (a,b) = gcut s t in
           s::flatMap (fun (_,_,n) -> reachable n b) a
```

The reachable function declares the following: The reachable states are s and all the states reachable from all neighbor of s.

3.3.2 Accept example

To give an example of a typical FLAT problem that requires a non-intuitive solution, we will discuss the problem of testing the acceptance of a word in a finite automaton. To resolve the problem in its maximum generality we will consider that the automata can be non-deterministic. Notice that in this situation it will be necessary to use a breadth-first exploration strategy, for only then can we guarantee the detection of the acceptance situation; thankfully we already saw the usage of depth-first and breadth-first strategies in the previous chapter, so hopefully the reader will have a better base understanding of these strategies, which will be important for grasping the full scope of our example.

Before jumping to the main problem, we will need a function that, given a state, a symbol and a set of transitions, will give us all the states for which we can transition to. Here is the code of said function.

```
let nextStates sy st t =
      let n = List.filter
          (fun (a,b,c) -> st = a && sy = b) t in
            List.map (fun (_,_,d) -> d) n
```

3.3.2.1 Deterministic Finite Automata accept example using depth-first

In the first solution we will attempt to use the definitions of the documentation as straightforward as possible, and we reach the following function:

```
let rec accept1 w st t sta =
      match w with
            [] -> List.mem st sta
          | x::xs -> let n = nextStates x st t in
                  List.exists (fun c -> acc xs c t sta) n
```

The function receives as arguments a word, a current state, a list of the automata's transitions and the list of its accepted states. The first call passes the full word, the initial state, the transitions and the accepted states. Notice that it has a similar structure to the `belongs1` function previously presented in chapter 2.

The function analyses a word which we pretend to verify. In the case of the empty word, it is accepted by the automata only if the current state is of acceptance. If the word displays the form x::xs, it is necessary to consider every state to where we can transition with the symbol x and check if from any of those states the sub-word xs can be accepted. This is a typical inductive form of reasoning. The high-order function `List.exist` deals with a variable number of recursive calls and tests if any of those calls guarantee acceptance.

This implementation is depth-first because each call inside `List.exists` is evaluated until de end. This function makes sense only for DFAs because otherwise the analysis could become trapped in an unproductive path.

Note that to simplify we assumed that the automata does not contain any empty transitions.

3.3.2.2 Non-deterministic Finite Automata accept example using breadth-first

The previous solution easily enters in infinite loops because the automaton can contain loops. It is necessary to devise a sophisticated and less intuitive solution using breadth-first.

The function receives the list of transitions and accepted states, but its first argument is a list of pairs (state, word), as explained below. The initial call passes the pair (initial state, word), and both transitions and accepted states lists.

```
let accept2 cf t sta =
     match cf with
          [] -> false
        | (st, [])::ls -> List.mem st sta || acc ls t sta
        | (st,x::xs)::ls -> let n = nextStates x st t in
          let cfn = List.map (fun c -> (c,xs)) n in
              acc (ls@cfn) t sta
```

The function has three branches, just as the function `belongs2`. The hardest part to explain is the fact that the arguments w and st of the function `accept1` now appear in the form of a list of configurations which are ordered pairs containing a state and a word. In reality, the function uses a breadth-first strategy to go through an implicit search tree which may be infinite. Each ramification of that implicit tree is determined by the state from which we want to perform recognition and the word we want to recognize.

Thus, if the list of configurations is empty, then we can decide that the word is not accepted. If the list is not empty and the first configuration of the list has an empty word, then that word is only accepted if the state of the same configuration is of acceptance; otherwise it is necessary to analyse the remaining configurations. If the list is not empty and the first configuration of the list has a non-empty word `x::xs`, it is necessary to consider all the states to where we can transition using the symbol x and create new configurations with the sub-word xs to be analysed in the future.

This is breadth-first because when consuming the symbol x, we add new configurations to the end of the list (as expressed by `ls@cfn`), thus assuring that we first analyse the configurations that are waiting longer.

If the word belongs to the language accepted by the automata, then the function will eventually terminate and produce the result true. If the word does not belong to the language recognized by the automata, then the function can terminate with the result false, or it may never terminate, originating uncertainty over the real result. We are present before a semi-decidable algorithm. If we want to deal the problem of non-termination, we have to limit in some way the depth of the search over the implicit tree where the function `accept2` traverses.

4. Pedagogical Tools

The study of Formal Language and automata theory presents the computer science student with the opportunity to better understand the context and fundamentals behind some problems they might face during their future professional careers, which in turn will allow them to better think of the possible solutions and make the better decisions, thus contributing to improved productivity and work quality.

While the subject proves to be indispensable for any computer science curriculum, its sometimes abstract and mathematical nature presents an added challenge for the student to fully grasp its essence. As such, many teachers have found that the use of both visual and non-visual auxiliary learning tools, which provide a more concrete foundation on the topic, appear to improve the student's understanding of the taught concepts, thus facilitating both the teaching and learning of this subject.

Since the early 1960s, many learning tools have been developed, most of them would distinguish themselves from the competition by focusing on a certain niche or characteristic that the other tools didn't support; eventually, during the years some have become more popular than others.

According to "Fifty Years of Automata Simulation: A Review" [2], these tools could be divided into two main groups: one represents the text-based tools that use a collection of symbols to form a language, which we then use to write the definition of an automata, which is then processed using either compilers or interpreters; the second represents the tools that accept an automata specification (either in a structured or diagrammatic form) and then simulates its behavior in a graphical environment, often with the added implementation of animations.

The following section will aim at describing the state of the current landscape pertaining to these tools.

4.1 Classification and characteristics

Some like "Online Turing Machine Simulator" [13], an online Turing machine simulator, allow the user to define a Turing machine using the tool's syntax, and then write and input for the program to validate. The user can also select from a collection of pre-defined examples, view tutorials and even define the speed at which the simulation runs.

Others, such as "Abstract Machine Simulator" provide a module for generating words accepted by the automata being texted.

Some also allow the user to draw their own automata for all testing purposes, using drag and drop styled interfaces, one criticism these sometimes face is how disorganized and confusing writing bigger, more complex automata can become with these tools.

There are also various tools that allow for the conversion of nondeterministic automata into deterministic automaton, and then into a Turing machine.

Here is a list of most of these tools one might find while exploring the subject.

In 1963, Coffin et all published a paper entitled "Simulation of Turing machine on a digital computer" [14], which was perhaps the first ever automata simulator study, in it the authors describe the tool being text-based and adopting the classical Turing machine representation of quintuplet for each transition.

In 1972, on what would probably be the first graphic-based tool, Gilbert and Cohen published the paper "A simple hardware model of a Turing machine: its educational use" [15] where they describe a Turing machine simulator and its utility for teaching programming fundamentals.

Most tools until 1992 would then only support either Turing machines or Finite and pushdown automata, such as "Turing Machine Simulator" [16], "Tutor – A Turing Machine Simulator" [17] and "Turing's World" [18]; only in 1992 with "Hypercard Automata Simulation" [19] by Hannai et al did a tool support all 3 types, and in 1993, the c++ "Formal Language and Automata Package" [20] (the precursor to JFLAP) not only supported the 3 types mention, but also non-determinism.

In 1997, Head et al developed "A Simple Simulator for State Transitions" [21], which had support for a finite state machine simulator, a nondeterministic pushdown automaton simulator and a Turing machine simulator, all based on notational languages with rigid formats.

New tools have been developed since, other similar to the ones discussed so far include "Language Emulator" [22] by Vieira et al, "Automata en Java" [23] by Dominguez, "Turing Building Blocks" [24] by Luce and Rodger, a Java computability toolkit by Robinson et al [25], "PetC" by Bergström [26], "Thoth" by García-Osorio

4.2. Noteworthy tools

The following tools deserve a more profound introduction due to either their popularity, concept or availability.

4.2.1. Automata Tutor v2.0

Automata Tutor [29] is a web-browser application for users to test their knowledge on DFA, NFA, NFA to DFA conversion and regular expression constructions by providing exercises in which the user must use a drag and drop styled graphics tool to create the requested automata, or imput the correct regular expression in a more text-based window. Once submitted the answer, the site will calculate a score from 0 to 10 based on how closed the input is to the desired solution and provide feedback on how to improve the answer.

In the paper "Automated Grading of DFA Constructions ∗", Rajeev Alur et al explain that the grading of the exercises is done through the conversion of DFAs into a MOSEL formula and vice-versa, which allows for a method of evaluating the answer; the actual grading is achieved with an algorithm that divides all errors into 3 common types, them being an attempt to provide a solution to a different problem, the lack of a transition or final state, and an error on a small part of the answer string.

In the paper, the authors also concluded after testing for comparison between the gradings of the site and those of actual instructors, that the tool was able to provide a quality of grading equivalent to that of human graders.

4.2.2. **Racso**

Racso [30] is a web-browser application created in 2012 by Carles Creus and Guillem Godoy of Polytechnic University of Catalonia, with the objective of providing their students with exercises on FLAT, specifically context-free grammars, it is entirely text-based. According to their paper "Automatic Evaluation of Context-Free Grammars (System Description)", the tool consists of multiple exercises on FLAT concepts such as deterministic finite automata, context-free grammars, push-down automata, reductions between undecidable problems and reductions between NP-complete problems. The idea is that for every exercise describing a specific language, the student as to solve it by providing a (sometimes unambiguous) grammar that generates that language, the correctness of the answer is achieved by comparing the student's grammar with the professor's known-to-be-correct grammar to see if they generate the same language. Since equivalence of context-free-grammars is an undecidable problem, the work-around is to define a length L and test if there is a word with length lesser or equal then L that can be generated by only one of the two grammars, if such word exists, either both grammars are not equivalent (thus the answer is wrong) or the value of L was too low. The reason this works is due to the academic nature of the exercises, both grammars and L will almost always be of sufficiently small size for the comparison to behave adequately. The comparison itself is based on hashing, SAT and automata.

To use this tool as of this writing, the user must visit the site (https://racso.cs.upc.edu/juezwsgi/index) where they may choose from a plethora of representative exercises, divided into various classifications according to their specific topic, most noticeably exercises on DFA, CFG, regular and context-free operations. After selecting the exercise, a window with some instructions and a small text console will display and allow the user to input their solution, once submitted the site will display whether the answer was correct or not.

The site also allows the user to create an account which they can then use to gain access to a collection of exams on the subject of the exercises.

4.2.3. **JFLAT**

Perhaps the most popular and well-documented tool, JFLAP [31] is a Java implemented toolkit that, according to Susan Rodger et al in “A Hands-on Approach to FLA with JFLAP” began in 1990 as a collection of c++ and x windows tools called NPDA, when professor Susan Rodger of at the time Rensselaer Polytechnic Institute began teaching a FLAT course and found that students requested further counselling and feedback on their understanding of the subject. By 1993 the program already had support for simulating non-deterministic push-down automata, deterministic push-down automata and Turing machines with building blocks. In 1996 the tool switched to Java and in 2001 to Swing, suffering a complete rewrite and even saw a change in some of the algorithms. During the initial years, various FLAT concepts and tools have been implemented, such as L-systems in 1993, pumping lemma in 1996, a brute-force parser, LL parser and SLR parser between 1996 and 1997, and regular expressions in 1999. Some of the more recent additions include Moore and Mealy machines, Batch grading, regular pumping lemma proof, context-free lemma proof and various preference settings such as defining the empty string (epsilon or lambda).

To use the tool as of this writing, a user must go to the JFLAT official website (http://www.jflap.org) and fill in a form on why they are interested on the program, as well as country and faculty where they come from. Afterwards, they will be allowed to download an executable jar file which comprises of the JFLAT toolkit. The site also features a plethora of tutorials and even video instructions on how to use most of JFLAP’s functionalities, and how instructors can use them to better explain the subject.

The main features of JFLAT allows the user to create various types of automata and regular languages; convert NFA into regular grammar or expressions; create context-free languages such as push-down automata or context-free grammar, as well as to exert transformations on them; define Turing machines (either multi-tape or building blocks based) and create/render L-systems. The tool provides a drag and drop interface which allows the user to define a finite automata, then by entering a test word (various words can also be tested in simultaneous), the program can either compute whether the string is accepted, generate a diagram showing the behavior of the automata for a word up to a specific symbol, or analyze the consumption of its symbols one by one. One interesting functionality is the ability to show in parallel the possible transitions and state of non-deterministic automata for an input word. The tool also supports the ability to analyze certain properties and identify them for the student to better grasp them, like highlighting lambda-transitions or non-deterministic states. For pumping lemma, the tool implements an interface where you “play a game” against the computer, where each side will decide on an input until the end of the proof is reached, the interface allows the user to click a button that displays an explanation of the problem’s context.

The popularity of JFLAP is widely considered unparalleled compared to other FLAT tools; according to the JFLAP website, from 2004 to 2008 the tool had seen over 64.000 downloads in 161 different countries, and as of this date the site lists over 10 books that mention the usage of JFLAP, and over 30 published papers that reported having used, or even modifying JFLAP. One could argue that these numbers alone would suffice as testament to the importance of the tool’s role in helping teach FLAT, but in "Increasing Engagement in Automata Theory with JFLAP," Susan Rodgers et al conducted a 2 year study to see the responses of students from over 10 faculties when using JFLAP for their FLAT courses, and the results showed that more than half the enquired students admitted that the usage of the tool had either made learning the subject easier or more engaging.

4.2.4. **PFLAT**

PFLAT [32] is a text-based SWI-Prolog implemented tool from 2005, which focuses on providing a library of Prolog predicates that map the concepts of formal language and automata theory as closely as possible to their respective mathematical and formal definition. The tool provides the source code as to better help students grasp the intricacies of the subject. The tool allows for the instructors to adapt its definitions and naming’s to those they prefer and provides both student and teacher with the ability to extend the library with their own implementations of further concepts from the subject. To facilitate its usage, PFLAT also allows for various operators on words, regular languages and automata, such as concatenation, union, closure, and which ever possible operator its user might want to implement.

In “A Prolog Toolkit for Formal Languages and Automata“, the authors describe some of the functionalities and concepts and how they are implemented in PFLAT, and provide as example the definition of all binary words with an even number of 1’s. In PFLAT, an alphabet can be defined and checked against the computation of the set of symbols of a random set expression; a user can check for declaration errors and even have them shown on screen as error messages; words can be represented, with operations for concatenation and N-th power already available; predicates on words for checking if they belong to a specific alphabet, to generate all words over an alphabet, or to compare to another word and conclude if they respect a certain lexical order, for dealing with prefix, suffix and sub-word, and with the possibility to change/add predicates; operands for languages including literal sets of words, names of alphabets and language definitions, as well as operators for set, Kleene star, positive closure, product and power; defining regular language with regular expressions or finite automata; expressions can be build over finite automata, with the latter having support for the union, complement, intersection, closure, minimization and determinization operators; and few more features.

As of its debut, the tool only had support for regular languages and pushdown-automata. Later versions of the system have received support for all the other classic mechanisms.

4.3. **Conclusion**

It is interesting to note that, even though FLAT as been stabilized for some years, the advent of these learning tools has introduced new interesting challenges for the field of computation theory.

An observation one could draw from the analysis of the history of FLAT tools, is that most of them were, in the early years, mostly textual, and as time passed, more graphics-based tools were being made; a possible reason for this shift could be attributed simply to the evolution of more powerful, easy to use frameworks and mechanisms that facilitated the appearance of such tools.

While a great number of different tools already exist for helping teach formal language and automata theory, it is the communities' belief that the welcoming of further tools with different interpretations and concepts will help complement already existing ones and provide both student and teacher with often new functionalities to apply to their learning and teaching respectively.

5. Work plan and solution

5.1 Solution

The objective is to develop, using the OCaml language, a pedagogical tool called OCaml-Flat to support students in learning the concepts of formal languages and automata theory (FLAT). The functionalities of the system will follow the documentation of the discipline of computation theory, organized by Professor António Ravara. In this documentation, several choices are made, especially regarding the selection of specific algorithms.

The tool will be developed into a library of types and functions developed in OCaml. There are three intended uses for this library:

1-using the tool within the textual interface of the OCaml interpreter, students will have the possibility to define, load, test and manipulate FLAT mechanisms. However, it is necessary to realize that for a student to use this system, they will have to spend some time learning the data representation and API of the tool.

2-the tool will be used in the context of the Mooshak automatic evaluation system. The idea is to support the creation of FLAT exercises, ready to use by students. In this case, each student only needs to know the format of the answer, without needing to know the API of the tool. Examples of Exercises: (1) write a finite automata that recognizes the language of binary numbers that are multiples of 3; (2) Minimize the given finite automata; (3) Convert a given regular expression into a regular grammar; (4) Give four examples of words that are recognized by a given stack automata.

3-On top of the tool will be developed a WEB application with interactive graphics. But this point is part of a separate master’s dissertation, which runs in parallel with this one.

5.2. Validation

All examples of the Computation theory documentation, and probably a few more, will be translated into the tool formats, which will allow for some confidence in the implementation correction but also to critically assess the usability of the system. Note that the system will need to implement some semi-decidable procedures and special tests will be created for these cases. Also, regarding usability, the opinion of professors of disciplines related to FLAT will be requested.

5.3. Work Plan

Objectives are better reached when divided into detailed sub-goals with concise yet flexible deadlines. As such we have devised a work plan for how we aim to proceed during the development of this Mc's thesis. Figure5.1 shows an overview of the scheduling for this work plan.

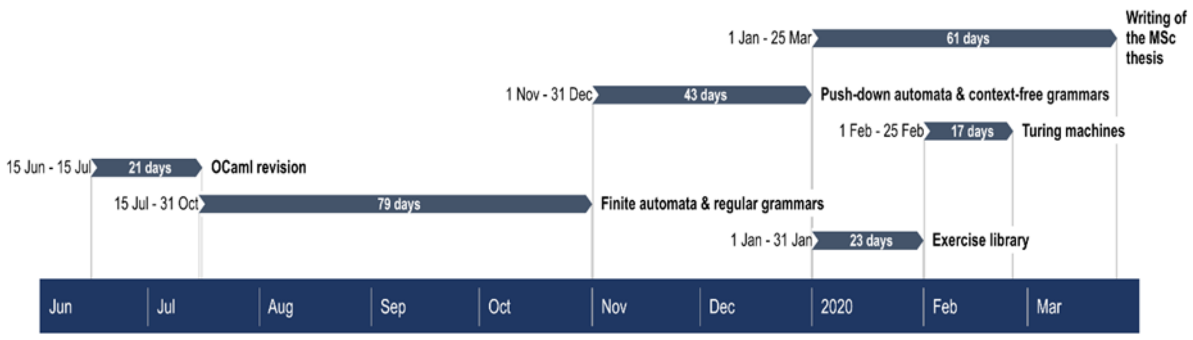

Figure 5.1- MSc's thesis work plan

1-Review of some functional programming techniques in OCaml. Writing some functions on finite automatons to gain some initial experience and to gain a more accurate notion about future work. This part has already been carried out. (15Jun-15Jul).

2-develop the essential of the project, in this first phase limited to finite automata and regular expressions. There are many non-trivial algorithms that need to be developed and they should be expressed in a very clear manner and as close as possible to the TC documentation. In order to be able to support all kinds of exercises for students, some additional less orthodox functions should be added to the API, such as a function of accepting words for regular expressions (although this is a mechanism) and a word generation function for finite automata (despite being a recognizer mechanism). Some of the algorithms involve non-determinism and non-terminations, which makes emerging some challenging practical problems that will be interesting to deal with. (15Jul-31Oct).

3-Repeat what was described in the previous point, but now for push-down automata and for context-independent grammars. Many of the situations to be dealt with will be repeated, but now appear in more complex versions, more difficult to handle, thus demanding greater mental effort due to the significant complexity of the mechanisms. (01Nov-31Dec).

4-Create a rich library of exercises to test the system, both directly in the OCaml interpreter, as well as within Mooshak. In addition, if necessary, develop in partnership with the colleague of the other project, a small number of other functions, which result from the needs of the parallel project that involves a WEB application for FLAT. (1Jan-31Jan).

5-Ideally, features for Turing machines will also be programmed. In the case of scarce revealing time, this will be the omitted part. (1Feb-29Feb).

6-Writing of the master's thesis. (1Jan-25Mar).

Bibliography

[1] Learning, C., Reserved, A. R., & Learning, C. (n.d.). Introduction to the theory of computation_third edition - Michael Sipser.

[2] Chakraborty, P., & Saxena, P. (2011). Fifty years of automata simulation: a review. ACM Inroads, 2(4). Retrieved from http://dl.acm.org/citation.cfm?id=2038893

[3] Chesñevar, Carlos & Cobo, Maria & Yurcik, William. (2003). Using theoretical computer simulators for formal languages and automata theory. SIGCSE Bulletin. 35. 33-37. 10.1145/782941. 782975.

[4] Ctp.di.fct.unl.pt. (2019). [online] Available at: http://ctp.di.fct.unl.pt/miei/lap/teoricas/02.html [Accessed 15 Jul. 2019].

[5] Rojas, R. (2015). A Tutorial Introduction to the Lambda Calculus Rául Rojas∗ FU Berlin, WS-97/98 Abstract. Blackwell, 1–17. https://doi.org/10.1006/anbe.1999.1219

[6] Rojas, R. (2015). A Tutorial Introduction to the Lambda Calculus Rául Rojas∗ FU Berlin, WS-97/98 Abstract. Blackwell, 1–17. https://doi.org/10.1006/anbe.1999.1219

[7] Ocaml.org. (2019). *OCaml – OCaml*. [online] Available at: http://ocaml.org/ [Accessed 15 Jul. 2019].

[8] Vujosevic Janicic, Milena & Tošić, Dušan. (2008). The role of programming paradigms in the first programming courses. The Teaching of Mathematics. 11.

[9] John Hughes, the U. of G. (2004). Why Functional Programming Matters. "Research Topics in Functional Programming," 3(1), 53–67. https://doi.org/10.1163/1574-9347_bnp_e328980

[10] Heineman, G., Pollice, G., & Selkow, S. (2016). Algorithms in a Nutshell. In International immunology. https://doi.org/10.1093/intimm/dxu021

[11] Kozen, D. C. (1990). The design and analysis of algorithms. Statistics, I(December 1990), 346. https://doi.org/10.1007/978-1-4612-4400-4

[12] Felleisen, M., Findler, R., Flatt, M. and Krishnamurthi, S. (n.d.). How to Design Programs.

[13] Turingmachinesimulator.com. (2019). Online Turing Machine Simulator. [online] Available at: https://turingmachinesimulator.com/ [Accessed 15 Jul. 2019].

[14] Coffin, R. W., Goheen, H. E., & Stahl, W. R. (2008). Simulation of a Turing machine on a digital computer. 35. https://doi.org/10.1145/1463822.1463827

[15] Gilbert, I. and Cohen, J. 1972. A simple hardware model of a Turing machine: its educational use. Proceedings of the ACM Annual Conference, pp. 324-329.

[16] Cernansky, M., Nehéz, M., Chudá, D. and Polický, I. 2008. On using of Turing machine simulators in teaching of theoretical computer science. Journal of Applied Mathematics, 1(2): 301-312

[17] Pierce, J. C., Singletary, W. E. and Vander Mey, J. E. 1973. Tutor – a Turing machine simulator. Information Sciences, 5: 265-278.

[18] Barwise, J. and Etchemendy, J. 1986. Turing's World: An Introduction to Computability, Academic Courseware Exchange

[19] Hannay, D. G. 1992. Hypercard automata simulation: finite-state, pushdown and Turing machines. ACM SIGCSE Bulletin, 24(2): 55-58

[20] LoSacco, M. and Rodger, S. H. 1993. FLAP: a tool for drawing and simulating automata. Proceeding of the World Conference on Educational Multimedia and Hypermedia, pp. 310-317

[21] Head, E. F. S. 1997. ASSIST: A Simple SImulator for State Transition. http://www. cs.binghamton.edu/~software/ASSIST.html.

[22] Vieira, L. F. M., Vieira, M. A. M. and Vieira, N. J. 2004. Language emulator, a helpful toolkit in the learning process of computer theory. inroads – ACM SIGCSE Bulletin, 36(1): 135-139.

[23] Dominguez, A. E. O. 2009. Automata. http://torturo.com/wp-content/uploads/Automata.jar.

[24] Luce, E. and Rodger, S. H. 1993. A visual programming environment for Turing machines. Proceedings of the IEEE Symposium on Visual Languages, pp. 231-236.

[25] Robinson, M. B., Hamshar, J. A., Novillo, J. E. and Duchowski, A. T. 1999. A Java-based tool for reasoning about models of computation through simulating finite automata and Turing machines. inroads – ACM SIGCSE Bulletin, 31(1): 105-109.

[26] Bergström, H. 1998. Applications, Minimisation, and Visualisation of Finite State Machines. M.Sc. dissertation, Royal Institute of Technology, Stockholm University

[27] García-Osorio, C., Mediavilla-Sáiz, I., Jimeno-Visitación, J. and García-Pedrajas, N. 2008. Teaching pushdown automata and Turing machines. inroads – ACM SIGCSE Bulletin, 40(3): 316.

[28] Almeida, André & Almeida, Marco & Alves, José & Moreira, Nelma & Reis, Rogério. (2012). FAdo and GUItar: Tools for Automata Manipulation and Visualization.

[29] Automatatutor.com. (2019). App: Home. [online] Available at: http://www.automatatutor.com/ [Accessed 15 Jul. 2019].

[30] Racso.cs.upc.edu. (2019). RACSO. [online] Available at: https://racso.cs.upc.edu/juezwsgi/index [Accessed 15 Jul. 2019].

[31] Jflap.org. (2019). JFLAP. [online] Available at: http://www.jflap.org/ [Accessed 15 Jul. 2019].

[32] Wermelinger, M., & Dias, A. M. (2006). A prolog toolkit for formal languages and automata. ACM SIGCSE Bulletin, 37(3), 330. https://doi.org/10.1145/1151954.1067536.