João Miguel Gago Gonçalves

Bachelor in Computer Science and Engineering

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

Dissertation submitted in partial fulfilment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Artur Dias, Assistant Professor,
NOVA School of Science and Technology

Co-adviser: António Ravara, Associate Professor,
NOVA School of Science and Technology

Examination Committee

Chair: Prof. João Lourenço, FCT-UNL
Rapporteur: Prof. Rogério Reis, FCUP
Members: Prof. Artur Dias, FCT-UNL
Prof. António Ravara, FCT-UNL

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

March, 2020

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

This document was created using the (pdf)LaTeX processor, based on the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

Firstly, I would like to thank the Tezos Foundation for supporting this work through a grant for the project "FACTOR - A Functional Programming Approach to Teaching Portuguese Foundational Computing Courses".

Next, I would like to thank my advisors, Professor Artur Miguel Dias and Professor António Ravara, for their interest and guidance during all stages of this project.

I would like to thank Professor Simão Melo de Sousa, for both his role in this project and his hospitality during my brief visit to the University of Beira Interior.

I would also like to thank all of my friends and colleagues who directly or indirectly helped me during the course of this last academic challenge. In alphabetical order: Bruno Garcia; Duarte Oliveira; Mariana Matias and Rita Macedo.

Lastly, and most importantly, I would like to thank my family, specially my parents, for always putting me first, for always believing in me and for always being there for me. I am forever in your debt.

Abstract

Due to its fairly formal nature, the teaching of the subject of Computation Theory often presents itself as a major obstacle for Computer Science students in general. To combat this, there have been developed many tools for the teaching of Formal Languages and Automata Theory (FLAT). Despite the considerable number of them, occasionally a new tool emerges that contributes with something new.

It was with this objective that we developed a library of OCaml functions that support various concepts of FLAT, namely the definition of finite automata, regular expressions and context-free grammars. Our implementation of these concepts closely follows their classical formalisation. We chose to implement this project using the OCaml language as it would allow us to write code according to the functional programming paradigm, which would better help us reach our defined goals.

In this report, to better contextualize the reader on the challenges and results of our project, we start by giving an overview of the properties of functional languages. Next, we give a small introduction to the FLAT concepts and discuss issues about their implementation. We then review the existing FLAT pedagogical tools.

After establishing our knowledge base, we discuss the architecture of our program. Next, we discuss the various FLAT algorithms and operations we implemented in our program, giving detailed insight on the decisions behind our implementations, as well as on solutions we found for the various technical difficulties such as guaranteeing termination of our algorithms, dealing with the nondeterminism of many concepts and catering to an extensible design. For our last topic, we discuss the top-level functionalities provided in our program.

We conclude that the usage of the functional programming paradigm, in combination with our adherence to the specified formalisms, allowed us to program our tool in a manner that made it viable for use in courses that teach FLAT concepts according to the classical definitions.

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

Resumo

Devido à sua natureza bastante formal, o ensino da disciplina de Teoria da Computação tende a apresentar-se como um obstáculo considerável para a generalidade dos estudantes de Ciências da Computação. Para combater isto, têm sido desenvolvidas várias ferramentas para o ensino de Teoria de Linguagens Formais e Autómatos (FLAT). Apesar do seu considerável número, ocasionalmente surge uma nova ferramenta que contribui com algo novo.

Foi com este objetivo que desenvolvemos uma biblioteca de funções OCaml de suporte a vários conceitos de FLAT, nomeadamente a definição de autómatos finitos, expressões regulares e gramáticas livres de contexto. A nossa implementação destes conceitos é fidedigna à sua formalização clássica. A linguagem OCaml foi escolhida para este projecto pois permitir-nos-ia escrever código no estilo de programação funcional, contribuindo assim a melhor alcançar os nossos objectivos definidos.

Neste relatório, para contextualizar o nosso trabalho, começamos por fazer um resumo das propriedades das linguagens funcionais. Depois, fazemos uma pequena introdução sobre os conceitos de FLAT e discutimos algumas questões sobre a sua implementação. Em seguida revemos as ferramentas pedagógicas de FLAT existentes.

Estabelecida a base de conhecimento, discutimos a arquitetura do nosso programa. Depois, discutimos os vários algoritmos e operações de FLAT que implementámos no nosso projecto, fornecendo uma perspetiva detalhada sobre as decisões para as nossas implementações, bem como para as soluções encontradas para vários problemas técnicos como garantir a terminação dos nossos algoritmos, lidar com o não-determinismo de vários conceitos e escrever código extensível. Para o tópico final, discutimos as funcionalidades do domínio de topo do nosso programa.

Concluímos que o paradigma de programação funcional, em combinação com a nossa aderência aos formalismos especificados, permitiram-nos programar uma ferramenta viável para uso no ensino de conceitos de FLAT de acordo com as definições clássicas.

Palavras-chave: Teoria das Linguagens Formais e Autómatos, Programação Funcional, Linguagem OCaml, Ferramentas Pedagógicas

Contents

List of Figures

Listings

Acronyms

ADT	Abstract Data Type
AlgDT	Algebraic Data Type
CFG	Context-Free Grammar
DFA	Deterministic Finite Automaton
FA	Finite Automaton
FCT	Faculty of Science Technology
FLAT	Formal Language and Automata Theory
MIT	Massachusetts Institute of Technology
NFA	Non-Deterministic Finite Automaton
UNL	NOVA University of Lisbon

Chapter 1

Introduction

The FACTOR (Functional ApproaCh Teaching pOrtuguese couRses) project is intended to promote the usage of OCaml in the portuguese speaking academic community, particularly through the development of pedagogical tools for the subjects in areas of Computational Logic and Fundamentals of Computing. One of such is the subject of Formal Language and Automata Theory (FLAT).

The teaching of FLAT is a staple of most Computer Science curriculums 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 moulding the minds of students to better critically think on how to solve the Computer Science related problems that may occur during their academic 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 FLAT more accessible to future students.

In the context of the FACTOR project and the teaching of FLAT, this MSc thesis consisted on developing the OCaml-Flat library, a pedagogical tool for support in the teaching of FLAT. The project repository can be accessed through the following url: `https://gitlab.com/releaselab/factor/OCamlFlat`.

1.1 Context

Historically, for a long time, the academic community has realized the utility in developing and using helper tools for teaching FLAT, and during the years it has been shown that students tend to fare 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 has 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 consisted on the development of a pedagogical tool, using the OCaml language, called OCaml-Flat. It is a library of types and functions that can be used as a tool for aiding students in their studies, 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 main objective for this program was for its code to be written in a manner that the students can read it and be able to identify the concepts as defined in the class materials. To achieve this, a certain amount of sophistication was needed for the implementation of computable and legible versions of these definitions. The main challenge was to find ways to deal with non-determinism and guaranteeing termination of our algorithms.

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 Nova School of Science and Technology[4]. As such it will be developed following the formalisms adopted in the discipline as faithfully as possible.

1.2 Contributions

The main expectations for our project are as follows:

- A very clear implementation in OCaml of a set of generators and language recognizers.
- Whenever possible, the code should closely follow the formalization of the concepts as studied by the students. The intent here is for the students to be able to read the code and identify the concepts as they are defined in the source material.
- Cater for an extensible design. 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 FLAT, and discussion of some issues about their implementation.

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

Chapter 5 - Detailed explanation on the chosen architecture for our program.

Chapter 6 - Analysis and discussion of the implementation strategy used in this project.

Chapter 7 - An in-depth discussion of the implementation of our chosen FLAT concepts and algorithms, including an analysis on our solutions for technical problems such as non-determinism and non-termination of some algorithms.

Chapter 8 - A manual for all the top-level functions of our tool.

Chapter 9 - Conclusion and future work.

Chapter 2

Functional Programming

The functional programming style was born from the acknowledgement that it is possible to express computation by only resorting to mathematical functions, applying functions to arguments and evaluating expressions [5]. In this programming style, functions play a central role, being treated as first-class values, as we will explain.

Regardless of their simplicity, functional languages help programmers in expressing ideas with better clarity and certainty than with other programming styles, such as the imperative style for example. This is due to several properties of the functional paradigm that allows us to expresses the logic of how to transform our given arguments into our desired outcome, rather than describing a sequence of instructions.

In this chapter, we give a brief introduction to the history of the functional paradigm, followed by an exposition on some of its main properties, and ending on a discussion between its advantages and disadvantages.

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 [6]. During his research in 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.

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[7],

even though Lambda-Calculus did not actually influence Lisp much, both McCarthy and Hudak 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 two of the most important functional languages of today were born – OCaml and Haskell. The former language uses strict evaluation while the latter uses non-strict evaluation. In the case of OCaml, its origins are found in Robin Miller's LCF test system, which dates from 1962. The language began to be used as an autonomous 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 [8].

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 functional mechanisms.

Even though its beginnings are rooted in the academic, and with the imperative paradigm remaining as the principle way in which most programmers today code [9], 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 distinctive core characteristics, namely:

2.2.1 High-order functions

One useful mechanism that is essential to functional programming is 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.

Listing 2.1: map function in the functional style

```
1 let rec map f l =
2   match l with
3     [] -> []
```

```
4     | x::xs -> ( f x )::map f xs
```

In Listing 2.1, the recursive function *map* receives as arguments a function f and a list l, and applies f to each element of l.

Listing 2.2: map function in the imperative style

```
5  map l =
6    for(int i = 0; i < l.size; i++)
7      l[i] = f (l[i])
```

In Listing 2.2, we have the same *map* function but implemented in an imperative style. Notice that since the function f is not an argument of *map*, f would have to be declared somewhere in the same scope as *map*.

Comparing the imperative implementation with the functional implementation, we observe that the usage of high-order functions gives us more flexibility in how we manage functions in our programs.

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 existenting functions.

Listing 2.3: compose function function

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

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 when applied to a value produces true if the value belongs to the set and false if it does not. It is known as the feature function of the set:

Listing 2.4: Empty set function

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

Listing 2.5: Universal set function

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

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

Listing 2.6: Set function

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

Listing 2.7: Belongs to set function

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

Listing 2.8: Set union function

```
15 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 behaviour 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:

$$f() + f() \tag{2.1}$$

$$2 \times f() \tag{2.2}$$

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, the repeated application of expressions is expressed using recursion [10], 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 technique which involves reducing each problem to a simpler version of the same problem. It is thus an inductive technique and the resulting functions are inductive. Here is a known example of an inductive function:

Listing 2.9: factorial function

```
16  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:

Listing 2.10: optimized factorial function

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

To better contrast the difference between actual iteration and simulated iteration, we can analyse the example in Listing 2.11.

Listing 2.11: imperative factorial function

```
20  int fact n =
21    int res = 1;
22    for(int i = n; i > 0; i--){
23      res = res * i;
24    }
25      return res;
```

Compared to the implementation using recursion, the implementation using iteration requires the reader to mentally execute the written code as to understand the objective of the function, while the recursive implementation can be understood more intuitively.

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 $fact\, n = n \times fact\,(n-1)$ 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. Notice however that 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, etc), with the functional languages category also included in this group. This aspect is widely considered to provide 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

The process of verifying the type safety of a program (that is, if it has any type errors in its code), is called type checking, and it can be classified as either static or dynamic. Static type checking is when the process happens at compile time, while dynamic type checking is when the process happens at run-time. A language has static typing if it performs static type checking and has dynamic typing if it performs dynamic type checking.

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 type of the argument as 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, for 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 Type (AlgDT), which are 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.

Listing 2.12: list function

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

It is normal for 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 AlgDT 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. Human beings are 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". In Listing 2.13 we have 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.

Listing 2.13: contains function

```
27  let containsA list =
28    match list with
29      Nil -> false
30      | Cons(x,xs) -> x = 'a' || containsA xs
```

The example in Listing 2.14 displays the same function *containsA* but implemented in an imperative style.

Notice how the use of pattern matching in Listing 2.13 makes the function more compact and readable compared to its imperative counterpart.

Listing 2.14: contains function

```
31  bool containsA list =
32    bool res = false;
33    for(int i = 0; i < list.size; i++){
34      if (list[i] == 'a'){
35        res = true;
36      }
37    }
38      return res;
```

2.2.9 Imperative mechanisms

It may seem strange, but the generality of functional languages are 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 properties that provide certain advantages [11].

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 techniques such as currying.

Since a functional program doesn't have mutable variables or state, and with the added characteristics of higher-order functions 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 code with less bugs.

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 techniques 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.

Higher order functions capture generic 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 [11].

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 state.

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 create new objects instead of assigning new values to already existing ones. Fortunately, the fact that the data is read-only, provides the implementation with more opportunities to freely share data, instead of copying it.

Chapter 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 at the NOVA School of Science and Technology. 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 FLAT.

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 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 CFGs.

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 (FA) are 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 FA recognizes some language in the sense that the FLAT checks whether a word belongs or does not belong to the language.

FA 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).

FA 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.

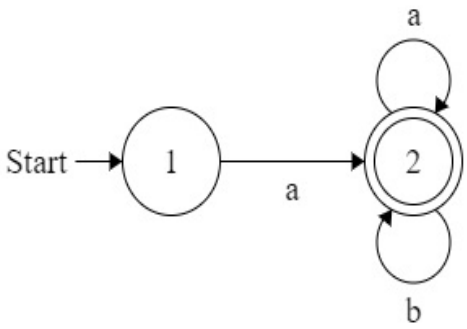


Figure 3.1: Deterministic Finite Automaton

These automata can be either deterministic or non-deterministic, where determinism in this context means that for each state a symbol can only transition to one state, whereas in non-determinism, 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 DFA and NFA.

As contrast to the previous example which was deterministic, here is an example of aNFA:

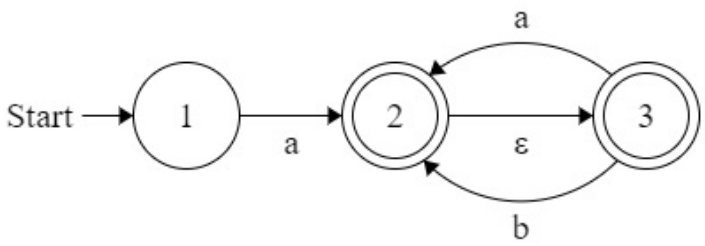


Figure 3.2: Non-Deterministic Finite Automaton

Notice how the automaton can transition from state 2 to 3 without consuming any symbol, thus making it a NFA.

FA 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 (CFGs) 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 non-terminal symbols, 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 CFG 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":

$$P \rightarrow \epsilon \mid aPb \tag{3.1}$$

Notice how P calls itself, indeed recursion plays an important role in defining a CFG.

CFGs 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.

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

Chapter 4

Pedagogical Tools

The study of FLAT 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 behaviour 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" [12], 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 tested.

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" [13], 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" [14] 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" [15], "Tutor – A Turing Machine Simulator" [2] and "Turing's World" [16]; only in 1992 with "Hypercard Automata Simulation" [17] by Hannai et al did a tool support all 3 types, and in 1993, the c++ "Formal Language and Automata Package" [18] (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" [2], 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" [19] by Vieira et al, "Automata en Java" [2] by Dominguez, "Turing Building Blocks" [20] by Luce and Rodger, a Java computability toolkit by Robinson et al [21], "PetC" by Bergström [22], "Thoth" by García-Osorio

4.2 Noteworthy tools

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

For each tool we will also analyse, in the context of the defined goals for this project, their advantages and disadvantages compared to our own tool OCaml-Flat.

4.2.1 Automata Tutor v2.0

Automata Tutor [23] 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 input 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" [24], 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.

Possibly the greatest advantage of this tool is its simplicity, as any student can easily apply their theoretical knowledge of FLAT concepts to the resolution of various practical exercises. However, some of its disadvantages to other tools include the lack of options for adding more exercises, which can be quite negative considering it offers a small compendium of problems to solve.

4.2.2 Racso

Racso [25] 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 CFGs, 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 DFA, CFGs, 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 CFGs 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.

When analysing the advantages and disadvantages of RACSO, we can observe that it is very similar to Automata Tutor. Both are very easy to use, and both suffer from a lack of extensibility. RACSO however has an advantage over Automata Tutor in the form of a wider selection of available exercises.

4.2.3 JFLAT

Perhaps the most popular and well-documented tool, JFLAP [26] 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 CFG, 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 FA, 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 behaviour 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 NFA 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.

JFLAT is possibly the most complete out of all our noteworthy tools. It has many advantages such as the implementation of various algorithms for a vast number of different FLAT mechanisms, a compendium of textual and video tutorials on how to correctly use this tool, etc. Its biggest disadvantage however lies in the initial difficulty some students may experience in both installing and using JFLAP.

4.2.4 PFLAT

PFLAT [27] 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 namings 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.

It is interesting to note that the advantages of PFLAT are similar to those of our own tool, with its biggest disadvantage being that it has since been discontinued.

4.2.5 FAdo

The FAdo [28] system is an open source software library for manipulation of finite automata, regular expressions and other FLAT mechanisms.

While its focus lies mainly in its usage for theoretical and experimental research, it is also foreseen to be used as a pedagogical tool.

It features a plethora of standard operations for the manipulation of regular languages, which are mainly represented as FA and regular expressions. These representations are implemented as Python classes.

Besides the elementary operations over regular languages such as union, intersection and concatenation, the library also contains the implementation of various conversions between FLAT models. Some operations such as the conversion from regular expression to NFA have multiple implemented versions using different approaches such as the Thompson methods, the Glushkov method, the Brzozowski methods, amongst others. Other operations, such as DFA minimization, are implemented using various algorithms, some of which are written in C.

FAdo has been used by its developer team to produce over ten pieces of scientific literature in the area of FLAT. Some of their research includes the implementation of code properties via transducers[29], enumeration and generation of initially connected DFA[30], etc.

The biggest advantage of FAdo, besides its extensibility and its implementation of a generous amount of operations and algorithms, is that its source code is available for its users to incorporate in their study of FLAT. Its biggest disadvantage however is that its code does not follow the classic definition (the definition we base our solution on) as closely as we desired for our project.

As a complement to the library, the FAdo team has also developed other tools to enhance the FAdo user experience, namely GUItar[31] and I-LaSer[31]. GUItar is a visualization tool for various types of automata, its features include diagram drawing, animation of algorithms, etc. I-LaSer is a system for answering questions pertaining to regular languages.

As of this date, FAdo is available for download at their official website `http://fado.dcc.fc.up.pt/` as either a tar.gz file or as a pip installation.

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.

Each tool referenced in chapter 4.2 has its own set of advantages and disadvantages that seem to effectively complement each other. However, for this project we wanted to devise an OCaml tool using the functional style as to try and produce code that would follow the intended definitions as closely as possible, while simultaneously being easy to understand by the future users of our tool.

Furthermore, while there already exists a great number of different tools for helping in the teaching of FLAT, it is the belief of the community 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.

Chapter 5

Program Architecture

The program is organized as a set of three libraries, each presented as a coherent logical unit, and will allow for the creation and usage of examples of FLAT mechanisms. It will be accompanied by a collection of examples which we consider of interest to illustrate the properties of their respective mechanisms, most of which are directly referenced in the source material of the course. The examples consist of json files that the program will be able to parse and interpret as FLAT mechanisms, allowing for a user to define their own examples and exercises.

5.1 Overall program organization

The first library "OCamlFlat" consists of all the logic behind the manipulation of our abstract concept models, this includes implementation of both their in-code representation and their various operations.

The second library "OCamlFlatSupport" contains all support functions that are needed for the first library, but whose logic is not directly related to the logic of the FLAT concepts themselves. These include the functions for the parsing of regular expressions and CFGs. By placing the referred functions in this second library instead of the first, we allow for a better organization and readability of both libraries.

The third library "OCamlFlatTop" contains the top-level interface functions to be used in the context of an OCaml interpreter.

The purpose of this organization was to facilitate in the understanding of the program's functionalities, as writing everything in the same library would clearly make the code too convoluted, and thus less accessible to future users.

5.2 Modules

In this project, we will need to implement various abstract concepts, each with their own set of functions and parameters. It would go against our main goal of producing elegant and easy to read declarative code, without a good organization of these concepts and their properties, thus our code is divided into modules, where each module contains only the logic directly pertaining to its associated abstract concept. The following architecture allows for clear logical organization and future extensibility of our program.

5.2.1 Error

The Error module serves as a way of registering all errors that may be detected during the validation of the models.

5.2.2 Util

The Util module contains all functions not directly related to any other module, but that are needed in the rest of the program, such as various type conversions and file loading functions. This eliminates the need of duplicated code and allows for an overall better read of the other functions.

5.2.3 Set

The Set module contains all functions needed for the usage of sets in our program, as set theory plays a vital role in all our FLAT mechanisms.

5.2.4 Json

The Json module contains all functions necessary to manipulate Json files in our program.

5.2.5 RegExpSyntax

The RegExpSyntax module is used to parse regular expressions. Since parsing is not part of the concept itself, we separate these functions from those that directly pertain to regular expressions.

5.2.6 CFGSyntax

The CFGSyntax module is used to parse CFGs. Like with regular expressions, we separate the parsing functions from those directly pertaining to the concept itself.

5.2.7 Entity

The Entity module is used to represent every abstract concept we will manipulate in our program. Various properties such as name and description, which are common in all

our abstractions, are defined in this module, as such all the other modules pertaining to specific concepts will thus inherit from this module.

5.2.8 Exercise

This module allows for the creation of exercises through the partial definition of languages. The user is given a description of a pretended language, as well as a set of words that belong to said language, and a second set of words that do not belong to it. The goal is for the user to try and define an example of a mechanism (depending on what the exercise specifies) where the language it defines is the same as the language defined in the exercises. The two sets of words can then be used to test if the example is correct. If the exercise is defined properly, the example can be considered correct if all words of the first set are accepted by the example, and if no word of the second set is accepted.

5.2.9 Model

The Model module inherits from Entity and is used to represent all FLAT mechanisms. Every function that is common amongst all mechanisms is defined in this module.

5.2.10 FiniteAutomaton

The FiniteAutomaton module is used to define FA. All elements of an automaton, as well as every function where we directly manipulate the mechanism itself, are defined and implemented within this module.

5.2.11 RegularExpression

The RegularExpression module is used to define regular expressions. All elements of a regular expression, as well as every function where we directly manipulate the mechanism itself, are defined and implemented within this module.

5.2.12 ContextFreeGrammar

The ContextFreeGrammar module is used to define CFGs. All elements of a CFG, as well as every function where we directly manipulate the mechanism itself, are defined and implemented within this module.

Chapter 6

Implementation strategy

During the preparation phase of our project, before initiating the development phase, there was an analysis on what would be the best implementation strategy for our project, given our set of goals.

There would be various properties we wished for our end product to possess. The most important aspect of our code would be for it to, whenever possible, be as legible, intuitive and faithful to the formalisms adopted by the Computation Theory class, even if it meant sacrificing some efficiency.

In this chapter, we aim to guide the reader through the though process that lead us to the implementation strategy on which we eventually settled.

To do this, we will start by discussing the first approach we thought of, which comprised of using explicit depth-first and breath-first exploration as the basis of our strategy. Then, after analysing the shortcomings of the previous approach, we will discuss the strategy based on set and closure operations that we ultimately decided on using for our project.

For both these strategies, we will use as an example the problem of testing the acceptance of a word in a FA. This will establish a commonality between the two approaches that will allow the reader a better understanding on our analysis.

6.1 Initial strategy using depth and breath-first exploration

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

The solution we found involve the usage of depth and breadth-first exploration. To

better illustrate these concepts applied to our program, we must first demonstrate how they can be used in a functional context. To do this, we will start by implementing a simpler example than our FLAT problems, we will demonstrate depth and breadth-first applied to a binary search tree algorithm.

6.1.1 Binary search tree example using depth-first

In OCaml, we can define a binary tree [32] using the following ADT:

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

Listing 6.1: binary tree ADT

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 [33].

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

Listing 6.2: depth-first binary tree search

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.

6.1.2 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.

```
rec belongs2 val Level =
   match level with
  [] -> false
```

```
4   | Nil::ls -> belongs2 val ls
5   | Node(x,l,r)::ls -> x = val || belongs2 val (ls@[l,r])
```

Listing 6.3: breadth-first binary tree search

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.

Now that we introduced the concepts of depth-first and breadth-first in a functional context, we can discuss how we initially envisioned to implement our program with this strategy.

6.1.3 Deterministic Finite Automata accept example using depth-first

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.

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

Listing 6.4: next states function

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

```
1 rec accept1 w st t sta =
2   match w with
3     [] -> isMember st sta
4     | x::xs -> let n = nextStates x st t in
5       exists (fun c -> accept1 xs c t sta) n
```

Listing 6.5: accept function

The function receives as arguments a word, a current state, a list of the transitions of an automaton 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 automaton 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 *exists* 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 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 automaton does not contain any empty transitions.

6.1.4 Non-deterministic Finite Automata accept example using depth-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.

Listing 6.6: accept2 function

```
1 rec accept2 cf t sta =
2   match cf with
3     [] -> false
4     | (st, []):::ls -> isMember st sta || acc ls t sta
5     | (st,x::xs)::ls -> let n = nextStates x st t in
6       let cfn = map(fun c -> (c,xs)) n in
7         accept2 (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, we can then 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.

6.2 Finite Automata accept example using set and closure operations

As seen in the previous example, if we were to use our initial strategy, we would also have to implement a mechanism for detection of duplicate configurations. This way, we could ascertain the termination of our algorithms with greater confidence, as the identification of said duplicates would allow us to not explore the respective configuration, thus avoiding an infinite loop without compromising our results.

However, while this approach would theoretically lead to correct results, it did not follow the adopted formalisms as closely as we desired.

Upon realising this, we decided to re-analyse our source material, with the intention of attempting to conceive a different strategy that would better cater to our goals.

The resulting solution to our accept problem was an algorithm that, instead of configurations, would receive a set of current states sts and a word w. The idea would that be that, by traversing the set of transitions of the automaton according to w, we could obtain the set of states reachable from any state in sts through the word w. For each symbol of w, the transition would be effectuated on all states of the set before proceeding to the next symbol, unlike our depth-first approach which would apply the transitions through w one state at a time.

The following chapter will discuss in detail the various FLAT algorithms and their implementation. Since the accept problem is included in that discussion, we decided to omit certain implementation details in our following analysis.

Listing 6.7: apply transitions function

```
1 applyTransitions sts sy t =
2   flatMap (fun st -> nextStates st sy t) sts
```

The applyTransitions function produces, for every state in sts, the set of states reachable from the state st through the symbol sy according to the set of transitions t.

Listing 6.8: inductive accept function

```
rec acceptX sts w t =
  match w with
       [] -> (inter sts acceptStates) <> empty
    |x::xs -> let nextSts = applyTransitions sts x t in
      nextSts <> empty && acceptX nextSts xs t
```

The acceptX function uses the logic expressed in applyTransitions to produce the set of states reachable from any state in a set sts through a word w according to a transition set t. If the resulting set contains at least one acceptance state, then we can assert that w is accepted by the automaton in question. This is thus a relatively direct translation of our previously defined solution.

For automaton without ε-transitions, this solution thus far is enough to give us confidence in the termination of our algorithm, due to the decrease of the argument w with every consecutive recursive call.

Notice however that, if the automaton were to have ε-transitions, the above solution can produce incorrect results.

To remedy this, and following the logic of our strategy so far, we can implement the ε-closure operation, which given a state st generates the set of states reachable from st by any number of ε-transitions. This can be integrated in our solution with a slight modification to the applyTransitions function.

Listing 6.9: closeEmpty function

```
let rec closeEmpty sts t =
  let ns = flatMap (fun st -> nextEpsilon1 st t) sts in
    if (subset ns sts) then ns
    else closeEmpty (union sts ns) t
in
applyTransitions sts sy t =
  let nsts = flatMap (fun st -> nextStates st sy t) sts in
    union nsts (closeEmpty nsts t)
```

The closeEmpty function is an implementation of ε-closure applied to multiple states at once. In this case, it is applied to nsts, the set of all states reachable by any state in sts through a symbol sy.

Notice that the approach we used to implement the closeEmpty function was to obtain all neighbours of any state in sts through an ε-transitions and recursively replicate the procedure until no new state is generated, thus increasing our confidence in its termination.

With all of this, we have successfully implemented a solution whose strategy we feel can be used to correctly implement our desired FLAT algorithms as closely to our goals as possible.

Chapter 7

Implementation discussion

As stated previously, the main properties we wish for our program, is for its code to be as elegant, readable and as close to the adopted formalisms as possible. It was with this goal in mind that we chose to implement all our operations and algorithms using the strategy previously discussed in this document.

In this chapter, we will present for all the chosen main algorithms, their definition, our implementation, and how we tested them for correctness evaluation. We will approach them by mechanism.

7.1 Finite Automaton

According to the source material of the courses, we define a finite automaton as a quintuplet $A = (S, \Sigma, s, \delta, F)$, where S is the set of states, Σ is the set of symbols recognized by the automaton (its alphabet), $s \in S$ is the initial state, $\delta \in S \times \Sigma \rightarrow S$ is the transition function, and $F \subseteq S$ is the set of acceptance states.

7.1.1 Accept

According to the source material of the course, we can formally define when a word is accepted by an automaton as $\{w \in \Sigma^* | \exists t(\delta^*(s, w) = t \wedge t \in F)\}$. Informally, a word is considered accepted by an automaton if there exists a chain of transitions for which, given the initial state and the word to be tested, the resulting set of reached states contains at least one state belonging to the set of acceptance states.

For example, the automaton in Figure 7.1 accepts all words starting with an a followed by a n number of b's and all words starting with a b followed by a n number of a's, where $n \in N$. As an example, for the word abb, starting on state 1, we have the following

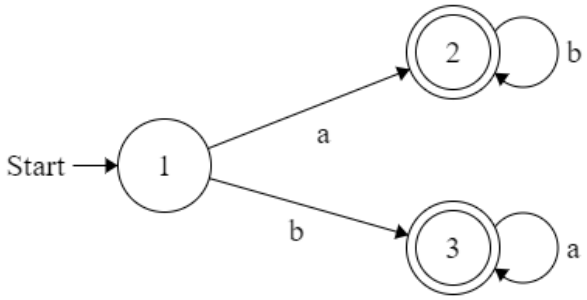


Figure 7.1: Automaton for accept example

transition chain: $(1,a) = 2$; $(2,b) = 2$; $(2,b) = 2$. Since the word abb leads to the state 2, and since 2 is an acceptance state, we prove that the automaton accepts the word abb.

By interpreting the formalism for the acceptance of a word as a mechanical procedure, we could say that it describes a semi-algorithm. To find if a word w is accepted by an automaton, we would have to explore all paths setting out from the initial state, disregarding all states that were not reachable through any sub-word of w, until we have used the entirety of w, and then test if the last reached state set contains at least an acceptance state. While it may seem that this process is guaranteed to terminate due to the reduction of the argument (the word w) with each traversed transition, when we introduce transitions with ε, we allow for the possibility of infinite loops while exploring these transitions, as ε allows us to traverse states without decreasing w, which can cause non-termination when the automaton does not actually accept w.

As an example, given an automaton with two states, each with $\varepsilon\text{-}transitions$ to each other, when testing for any word, the algorithm would loop endlessly between the two states in hopes of reaching a new unexplored state that could hypothetically lead to the acceptance of the word, not realising it had already explored both.

Our solution for this problem was to use the concept of $\varepsilon\text{-}closure$ and follow, in a single step, all the $\varepsilon\text{-}transitions$ we come across. When exploring the paths leading from any state st through a symbol sy of w, if we first apply $\varepsilon\text{-}closure$ to st, then simultaneously explore all resulting states for transitions with sy and to those reached states also apply $\varepsilon\text{-}closure$, we can then ignore $\varepsilon\text{-}transitions$ when exploring states without affecting the correctness of our result. Finally, by being able to circumvent the $\varepsilon\text{-}transitions$, we can guarantee that w will always decrease for each step of our algorithm, thus guaranteeing termination.

This functionality was implemented as a method that receives as arguments a word (which in our program is a list of characters) and tests its acceptance. Before explaining the main code, we first must clarify the following functions.

Listing 7.1: ApplyTransitions function

```
let applyTransitions sts sy t =
  let nsts = flatMap (fun st -> nextStates st sy t) sts in
    union nsts (closeEmpty nsts t)
```

The *applyTransitions* function receives a set of states *sts*, a symbol *sy* and a set of transitions *t*; its result is the set of all states reached by any state of *sts* through, firstly, exactly one *sy-transition*; after that, possibly, multiple *ε-transitions*.

The *nextStates* function receives a state *st*, a symbol *sy* and a set of transitions *t*; its result is the set of all states reachable from *st* through transitions of *t* with symbol *sy*.

The *closeEmpty* function receives a set of states *nsts* and a set of transitions *t*; its result is the set of all states reached by any state of *nsts* through *ε-transitions* (we represent these transitions with the help of a special "empty symbol"). This is our implementation of *ε-closure* applied to multiple states; we will use *ε-closure* repeatedly throughout the other automata operations.

Listing 7.2: Accept function

```
let rec acceptX sts w t =
  match w with
       [] -> (inter sts acceptStates) <> empty
    |x::xs -> let nextSts = applyTransitions sts x t in
      nextSts <> empty && acceptX nextSts xs t
in

method accept (w: word): bool =
  let i = closeEmpty (Set[initialState]) transitions in
    acceptX i w transitions
```

The central function is the *acceptX* function. It receives as arguments the current set of states *sts*, the word to be tested *w* and the set of the transitions *t*. Its result is true if the word leads to a state of acceptance and false otherwise. The code is quite clear in this function. For its base case, the empty word argument means that the result is only true if *sts* contains at least one state of acceptance. When *w* is a non-empty word, the result is true if, for the first symbol, the set of reached states can transition through the rest of the word to a state of acceptance. Note that, given the way this function was implemented, for the method to give correct results, the first call of the *acceptX* function needs to receive as its first argument, the result of the *closeEmpty* function for the initial state.

To better illustrate our implementation, for the automaton in Figure 7.2, if we were to test the acceptance of the word *a*, it would return as true. The reason is that for state 1, we obtain its *ε-closure* which results in the set of states 1, 2 and 4; since there is a transition through a from state 2 to the acceptance state 3, we thus prove that the automaton accepts word *a*. Notice how the *ε-transition* from state 1 to itself, which could lead to an infinite loop if the accept algorithm were to be implemented with a naïve depth-first strategy, does not present itself as an obstacle to the correctness and termination of our implemented strategy, due to how the *ε-closure* operates.

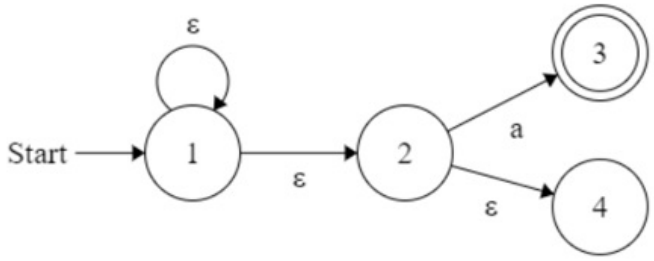


Figure 7.2: Automaton for second accept example

As previously explained, the termination of this algorithm is guaranteed by the decrease of our argument (the word) with each consecutive call of our recursive function, due to our use of the closure function applied to sets of states. For extra reassurance, we developed several unit tests. We verified the results of various automata for certain properties such as non-determinism, transitions through an empty symbol, transitions from and to the same state and cycles (when there exist two states that are both reachable from one another).

7.1.2 Generate

According to the source material of the course, the formal definition of the language of an automaton is the set of all accepted words, defined by $\{w \in \Sigma^* | \exists t(\delta^*(s,w) = t \wedge t \in F)\}$.

Informally, the language is the set of all words w where there is at least one transition chain using w to reach an acceptance state from the initial state.

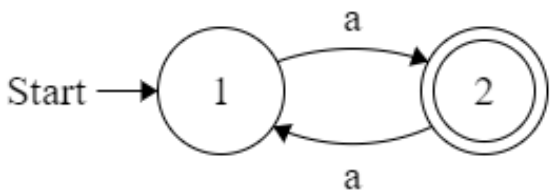


Figure 7.3: Automaton for generate example

As an example, the language of the automaton in Figure 7.3 is the set containing all words with an odd number of *a*'s. Notice how the language is infinite, and thus generating the set of all words of the language would naturally never terminate.

To combat this, we needed to establish an artificial termination point, which in our implementation was to impose a maximum length for the generated words.

The generate operation produces all words belonging to the language recognized by the automaton up to a specified size.

Listing 7.3: Generate function

```
let rec gen n st trns accSts =
  let clsEmpty = (closeEmpty (Set [st]) trns) in
    if n = 0 then
      if hasAcceptState clsEmpty accSts then Set [[]] else emptySet
    else
      let trnsSet = Set.flatMap (fun st -> nxtNonEmptyTrns st trns ) clsEmpty in
      let genX sy st l = addSyToRWords sy (gen (l-1) st trns accSts) in
      let lenOneOrMore = flatMap (fun (_,sy,st) -> genX sy st n) trnsSet in
      let lenZero = if hasAcceptState clsEmpty accSts then Set.make [[]] else Set.empty in
        Set.union lenOneOrMore lenZero
 in

method generate (length: int): words =
  gen length initialState transitions acceptStates
```

The generate method receives as argument the maximum length for all generated words and produces the set off all words recognized by the automaton whose length does not exceed that length.

The main function is the *gen* function that receives an integer n, a state st, the set of transitions $trns$ and the set of acceptance states $accSts$, it returns the set of generated words. Our base case is if n is zero, we consider the result to be the empty word, only if the set of states obtained from the ε-*closure* of st contains a state of acceptance, otherwise the result is the empty set. If n is greater than zero, for ε-*closure* of st the function produces all neighbour states, and concatenates each symbol (from the transitions that produced the neighbour states) to the result of the *gen* function applied to the obtained states, all while accumulating the remaining words accepted by the automaton. The result of the *gen* function is thus the set of all words recognized by the automaton with length not greater than the given maximum.

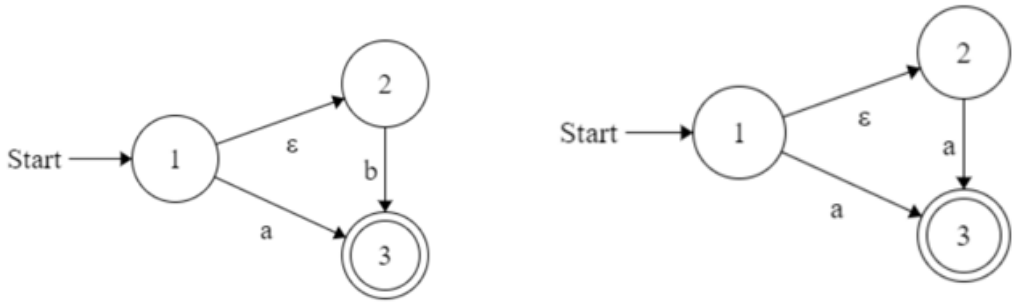
Again, if we observe the automaton in Figure 7.3, the result of our generate method for $n = 3$ for example, would be the set of words a and aaa.

The termination of this algorithm is guaranteed by the decrease of the argument n with each consecutive recursive call of the *gen* function.

7.1.3 IsDeterministic

According to the source material of the course, a non-deterministic finite automaton (NFA) can be defined in the same way as a deterministic one, with the exception of its transition set that becomes $\delta \in S \times (\Sigma \cup \varepsilon) \rightarrow P(S)$ where $P(S)$ is the power set of the set of states of the automaton. What is introduced are ε-*transitions* and multiple transitions for the symbol coming from the same state.

The above formalism defines the syntactic concept of what is a NFA, note however that by the definition, a DFA will always be a NFA (all DFAs are NFAs but not all NFAs are DFAs). What we want however, is to verify if the automaton, be it a DFA or an NFA, has a deterministic behaviour.



(a) NFA with deterministic behaviour (b) NFA with non-deterministic behaviour

Figure 7.4: Comparison between deterministic and non-deterministic NFA

To better illustrate the difference between syntactic and semantic determinism, lets observe the automata in Figure 7.4. While both can be classified as an NFA, due to both having a transition involving ε, we can see that the automaton in Subfigure 7.4a is deterministic, while the automaton in Subfigure 7.4b is non-deterministic. This is because the transition function of the automaton in 7.4a affirms that while state 1 can transition to state 2 through ε, it will only transition to one state for either symbols a or b, respectively states 3 and 2. If we then observe the automaton in 7.4b, it is non-deterministic not because of its ε-*transition*, but due to the fact that, unlike in 7.4a, its state 1 can transition to either states 2 or 3 through exactly the same symbol a.

To correctly infer if a given automaton displays deterministic or non-deterministic behaviour, we must resort to an analysis of the semantic concept behind the definition of non-determinism.

Since the definition of a deterministic automaton states that its set of transitions describes a partial function, that is, for every pair (state, symbol) of a deterministic automaton must correspond no more than one state, then we can infer that an automaton is non-deterministic when its transition set does not describe a function.

This means that, by verifying if there exists a state from which more than one state can be reached through the same symbol of its alphabet, we can deduce that the automaton is non-deterministic. If no such state exists, it is deterministic.

This can be achieved using the following algorithm: For every state n, obtain the set $s = \varepsilon\text{-}closure(n)$ and test if there are two or more transitions from any state of s with the same symbol.

If there is at least one state of the automaton for which the previous predicate wields true, then the automaton is non-deterministic, otherwise it is deterministic.

Listing 7.4: IsDeterministic function

```
1 let isDeter st ts =
2   let allSts = closeEmpty (Set[st]) ts in
3   let allTs = flatMap (fun st -> trnsFromSt st ts) allSts in
```

```
4   let sys = transitionGetSymbol allTs in
5     size allTs = size sys
6 in
7
8 method isDeterministic: bool =
9   exists (fun st -> not (isDeter st transitions)) allStates
```

If we were to apply our algorithm to the automata in Figure 7.4, the results would be that the automaton in 7.4b is deterministic while the one in 7.4a is non-deterministic. Since for both automata the algorithm applies the ε-*closure* to state 1, states 1 and 2 are treated as the same state when analysing if the transition set describes a function or a relation. Since in 7.4b each pair (state, symbol) has only one output state in its transition function, the automaton is deterministic, while in 7.4a the transition relation states that the pair $(1, a)$ can produce as output states 2 and 3, thus making the automaton in 7.4a non-deterministic.

Our implementation is thus a relatively direct translation of the defined algorithm, guaranteeing termination due to the non-recursive nature of our functions and our usage of library functions over sets that we know to always terminate.

7.1.4 Determinization

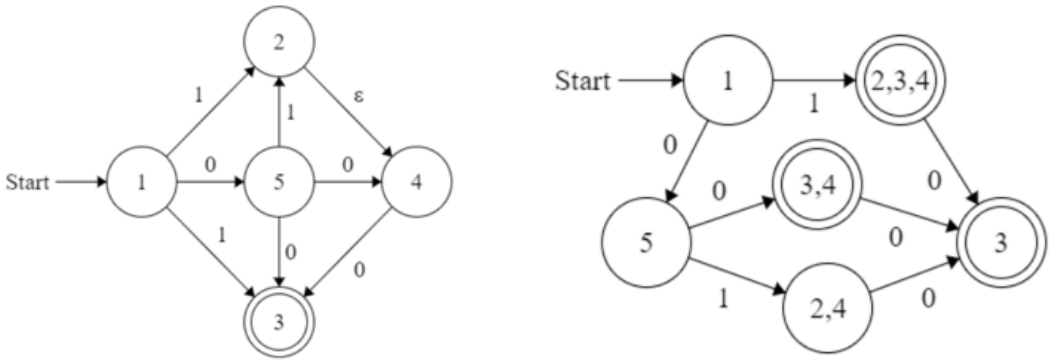
The chosen algorithm for conversion of non-deterministic to deterministic finite automaton is the one used in the course, the "Rabin-Scott powerset construction"[34]. According to the source material of the course, the formal definition of this algorithm is as follows: given the automaton $A = \{SA, \Sigma A, sA, \Delta A, FA\}$ we obtain the DFA $D = \{SA, \Sigma D, sD, \delta D, FD\}$ in the following way:

- $SD = \wp(SA)$.
- $\Sigma D = \Sigma A$
- $sD = closeempty(\{sA\})$
- $\delta D = \{(s, a) \rightarrow s' \in SD \times \Sigma D \times SD | s = closeempty(move(\{s\}, a))\}$
- $FD = \{s \in SD | s \cap FA = \varnothing\}$

Where $closeempty \subseteq \wp(S) \rightarrow \wp(S)$, $closeempty = \{U \rightarrow V | V = \{s | \exists t.t \in U \wedge (t, \varepsilon, s) \in \Delta^* A\}\}$ and $move \subseteq \wp(S) \times \Sigma \rightarrow \wp(S)$, $move = \{U \times a \rightarrow V | V = \{s | \exists t.t \in U \wedge (t, a, s) \in \Delta A\}\}$.

Informally, the algorithm explores all non-deterministic transitions (either ε-*transitions* or transition starting on the same state for different symbols) and replaces individual states for the set of states that, along with adjusting the transition set and the acceptance states accordingly, preserves the language of the automaton while simultaneously turning it deterministic.

As an example, in Figure 7.5 we have the NFA in Figure 7.5a and its equivalent DFA in 7.5b. In 7.5b, the state "2,3,4" originates from exploring all non-deterministic transitions starting from state 1 and acknowledging that they can be all replaced by a single transition through the symbol 1 to a state representative of states 2, 3 and 4 (the



(a) Before determinization (b) After determinization

Figure 7.5: Example of determinization

states reachable from all non-deterministic transitions starting in state 1). Notice how the language accepted by each of the automata in Figure 7.5 is identical.

Listing 7.5: Deter function

```
let rec deter stsD rD trnsD alph =
  let nxtTs = flatMap (fun stSet -> newTrns stSet ) rD in
  let nxtRs = map (fun (_,_,z) -> z) nxtTs in
  let newRs = filter (fun r ->
    not (belongs r stsD)) nxtRs in
  if newRs = emptySet then (union trnsD nxtTs) else
    deter (union newRs stsD) newRs (union trnsD nxtTs) alph
```

The deter function closely follows the fourth rule of the formal definition, as it produces the set of transitions for the new deterministic automaton. This is the main function of our implementation of the determinization algorithm; the rest of the deterministic automaton (its new states, initial state and acceptance states) is extrapolated through a logical analysis of the resulting transitions. We believe in the termination of the algorithm due to the implementation being a relatively direct translation of the presented formal definition.

We believe in the termination of the algorithm due to the implementation being a relatively direct translation of the presented formal definition.

7.1.5 Minimization

The minimization algorithm makes use of various concepts whose formal definitions are as follows:

A state $t \in S$ is considered accessible when there is a word $w \in \Sigma$ such that $\delta^*(s, w) = t$; that is, if it there exists a chain of transitions to t from the initial state.

A state $t \in S$ is considered productive when there is a word $w \in \Sigma^*$ such that $\delta^*(t, w) \in F$; that is, if there exists a chain of transitions from t to an acceptance state.

A state is considered useful if it is simultaneously accessible and productive.

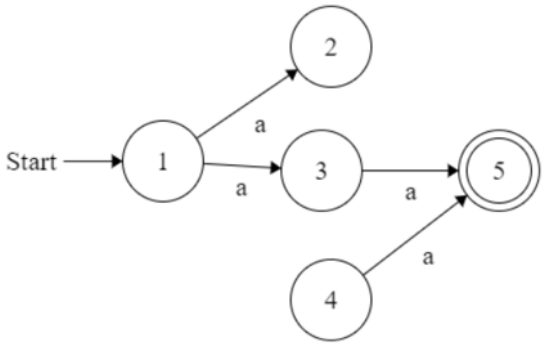


Figure 7.6: Automaton with two non-useful states

As an example, if we observe the automaton in 7.6, we see that state 2 is not considered productive, state 4 is not considered accessible and both states 2 and 4 are not considered useful. Only states 1, 3 and 4 are considered useful, as they are both accessible and productive.

Given a pair of states a and b, they are considered equivalent when for any $w \in \Sigma^*$,$\delta^*(a,w) \in F$ only if $\delta^*(v,w) \in F$. The intuition is that merging states a and b, as well as their transitions, do not modify the language identified by their automaton.

Two states are considered distinguishable if they are not equivalent.

The minimization algorithm is applied to deterministic automaton, it consists of:

• removing all useless states;

• identifying equivalent states through discovering the set of all pairs of distinguished states ;

• fusing equivalent states and adjusting the transitions accordingly.

According to the source material of the course, considering the automaton $D = \{S, \Sigma, \delta, s, F\}$ with two distinct equivalent states u and v where $u \neq s$, we could define the result of the minimization of D as the automaton $D' = \{S', \Sigma, \delta', s, F'\}$ where:

• $S' = S \setminus \{u\}$ • $\Delta' \subseteq S' \times \Sigma \rightarrow S'$, with o $\delta'(r,a) = v$, if $\delta(r,a) = u$ o $\delta'(r,a) = \delta(r,a)$, if $\delta(r,a) \in S'$ o $\delta'(r,a) = \perp$, if $\delta(r,a) = \perp$ • $F' = F \setminus \{u\}$

What the previous definition intuitively means is that, by removing either states u or v and adjusting transitions accordingly, we successfully minimize the automaton.

The complex part of this operation lies on finding the set of all pairs of distinguished states. We can define the algorithm for finding distinguishable states as $Dist = APED(I \cup A)$, where:

• $I = \{(u,v) \in S \times S | u \in F \wedge v \notin F\} \cup \{(u,v) \in S \times S | \exists a.\delta(u,a) \in S \wedge \delta(v,a) = \perp\}$

• $A = \{(u,v) \in S \times S | (u,v) \notin I \wedge \exists a.(\delta(u,a), \delta(v,a)) \in I\}$

• $APED(P,Q) = P \cup Q$, if $P \cup Q = S \times S \vee Q = \varnothing$; $APED(P,Q) = APED(P \cup Q, Q \cup R)$ otherwise, where $R = \{(u,v) \in S \times S | (u,v) \notin P \wedge \exists a.(\delta(u,a), \delta(v,a)) \in Q\}$

The general idea behind the algorithm is as follows. We first start by defining the initial arguments. I is the union of the set of all pairs of states where one is of acceptance and the other is not, with the set of all pairs where one state has a transition for a given symbol and the other state does not. A is the set of all state pairs (a,b) not belonging to I, where there exists a symbol that transitions (a,b) to a pair (u,v) belonging to I.

APED starts with $Dist = I \cup A$; it is the iterative process of adding to $Dist$ the rest of all pairs of distinguishable states by adding the pairs (a,b) not belonging to $Dist$, where there exists a symbol that transitions (a,b) to a pair (u,v) belonging to $Dist$. The process terminates when no new pair is added to $Dist$.

(a) Before minimization (b) After minimization

Figure 7.7: Example of automata minimization

As an example, by analysing the minimization of the automaton in Subfigure 7.7a resulting in the automaton in Subfigure 7.7b, we can see that by applying the algorithm for distinguishable state identification, the results would show that states 2 and 3 are equivalent, and thus the minimization operation eliminates one of them (in this case state 3), modifying the transition function in order to maintain the language recognized by the automaton.

Notice that the definition of the minimization algorithm terminates by design. As such, since our implementation of the minimization operation is a relatively direct translation of the aforementioned definitions, we believe in the termination of our algorithm, as well as its correctness, assuming the formal definition of the algorithm also guarantees correctness.

7.1.6 To Regular Expression

The implemented algorithm for conversion of deterministic finite automata into regular expressions was the transitive closure method[34].

According to the source material of the course, given the expression Rij denoting the set of words that lead from state i to state j, we can define the following recursive function:

$$R_{ij}^{0} = \begin{cases} a & if \quad i \neq j \quad \wedge \quad \delta(i,a) = j \\ a + \varepsilon & if \quad i = j \quad \wedge \quad \delta(i,a) = j \\ \emptyset & if \quad otherwise \end{cases} \tag{7.1}$$

$$R_{ij}^{n} = R_{ij}^{n-1} + R_{ik}^{n-1}(R_{kk}^{n-1})^{*}R_{kj}^{n-1} \tag{7.2}$$

In this function, the maximum value of n is equal to the number of the states of the automaton, thus guaranteeing termination of the algorithm.

The regular expression resulting from the conversion can be obtained by summing all $R_s f$ where s is the initial state and f is a state of acceptance.

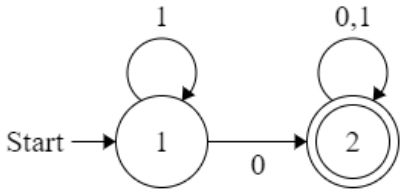


Figure 7.8: Automaton for example of conversion to regular expression

As an example, by strictly applying our algorithm to the automaton in Figure 7.8, we would get in return a very convoluted regular expression, that nonetheless would be equal to the regular expression $1^{*}0(0+1)^{*}$. Naturally, without minimizing the output, the resulting regular expression can wield to be fairly complex, redundant and long.

Our implementation of this algorithm is a relatively direct translation of the definition presented, so we believe in the termination and correctness of our method. Of course the code was thoroughly checked using unit tests.

7.2 Regular expression

According to the source material, given the alphabet Σ, we can define a regular expression as the set $RegExp(\Sigma)$ where:

- $\varnothing \in RegExp(\Sigma)$
- $\varepsilon \in RegExp(\Sigma)$
- $a \in \Sigma \Rightarrow a \in RegExp(\Sigma)$
- $E \in RegExp(\Sigma) \wedge F \in RegExp(\Sigma) \Rightarrow (EF) \in RegExp(\Sigma)$
- $E \in RegExp(\Sigma) \wedge F \in RegExp(\Sigma) \Rightarrow (E+F) \in RegExp(\Sigma)$
- $E \in RegExp(\Sigma) \Rightarrow (E^{*}) \in RegExp(\Sigma)$

7.2.1 Accept

The easiest and most efficient way to test if a given word is accepted by a regular expression would be to convert the regular expression to its equivalent automaton and then test the word. This is the traditional approach, used by real-world tools such as Lex of Linux, for example. Furthermore, this approach is available in our system, anyway.

However, given our main goal of following the course as closely as possible, we also provide a relatively direct implementation of the definition.

To define said algorithm, we must first start with its base cases. According to the source material of the course, for a word w and a regular expression Re over the alphabet Σ:

- $Re = \varnothing \Rightarrow w \notin L(Re)$
- $Re = \varepsilon \wedge w \in L(Re) \Rightarrow w = \varepsilon$
- $Re = a \in \Sigma \wedge w \in L(Re) \Rightarrow w = a$
- $Re = E1E2 \wedge w \in L(Re) \Rightarrow w = w1w2 \wedge w1 \in L(E1) \wedge w2 \in L(E2)$
- $Re = E1 + E2 \wedge w \in L(Re) \Rightarrow w \in L(E1) \vee w \in L(E2)$
- $Re = E^* \wedge w \in L(Re) \Rightarrow w \in Ln(F) \wedge n \in N$

Intuitively, the above formalism specifies the following cases:

If the regular expression Re is empty, then its language is also empty, thus no word can belong to its language; if Re is ε, then the empty word belongs to its language; if Re is a symbol a of its alphabet, then the word a belongs to its language; if Re is the sequence of two regular expressions $E1$ and $E2$, and w belongs to language of Re, then there is a decomposition $w1w2$ of w, where $w1$ belongs to the language of $E1$ and $w2$ belongs to the language of $E2$; if Re is the alternation of expressions $E1$ and $E2$, and if w belongs to the language of Re, then w belongs to either the language of $E1$, $E2$ or both; if Re is the Kleene star of expression E, and w belongs to the language of Re, then there is a word, formed by concatenating n times arbitrary words of F, that is equal to w.

By analysing the above definition, we can identify two obstacles to our algorithm. The first obstacle is the implicit non-determinism of finding the correct decomposition of a word w that justifies w being accepted by the sequence of two regular expressions. The second obstacle lies in the definition for the acceptance of a word when the regular expression involves the Kleene star, as the definition does not describe either an algorithm or a semi-algorithm, which complicates our implementation of said case.

The solution for our first problem was simple, we simply obtain all possible decompositions in two parts of our given word and test if any of them justify that w is accepted by the regular expression. This brute-force approach is a consequence of following the direct definition.

For example, if we were to test if the word $w = 01$ is accepted by the regular expression $r = 01$, we would decompose w in the set of pairs ("ε","01"), ("0","1") and ("01", "ε"). Since r is the sequence of regular expressions $r0 = 0$ and $r1 = 1$, we can deduce that $0 \in L(r0)$ and $1 \in L(r1)$, and so the second pair of the decomposition of w proves that the word is accepted by r.

The solution to our second problem starts by acknowledging that the regular expression e^* can be rewritten as $\varepsilon + ee^*$; so, we will try to reduce the e^* case to the concatenation case.

We can then develop the following algorithm: Given a word w and a regular expression e^*, w is accepted by e^* when $w = \varepsilon$ or when there exists a decomposition $w1w2$ of w

for which $w1$ is accepted by e and $w2$ is accepted by e^*, with this last case being recursive.

As an example, if we take the word w = "0101" and the regular expression $r = (01)^*$, our algorithm would find that r can be rewritten as $r2 = \varepsilon + 01r$. Since w can be decomposed as ("01","01"), we can see that for $r2$ we have that $01 \in L(01)$ and "01"$\in L(((01)^*)$. Then, for "01"$\in L((01)^*)$, through the decomposition ("01"," ε") of "01", we would see that "01" $\in L((01)^*)$ is true because of "01"$\in L(01)$ and "ε"$\in L((01)^*)$. Thus, the algorithm would indicate that the word w would be accepted by r.

Listing 7.6: accept function

```
let rec acc rep w =
  match rep with
    | Plus(l, r) -> (acc l w) || (acc r w)
    | Seq(l, r) -> let wpl = partition w in
      exists (fun (wp1,wp2) ->
        (acc l wp1) && (acc r wp2)) wpl
    | Star(re) -> w = [] ||
      (let wpl = remove ([],w) (partition w) in
        exists (fun (wp1,wp2) ->
          (acc re wp1) && (acc (Star re) wp2)) wpl)
    | Symb(c) -> w = [c]
    | Empty -> w = []
    | Zero -> false
in

method accept (w: word): bool =
  acc regExpression w
```

The acc function is a relatively direct translation of the accept algorithm as we described in this section. Notice however that for the case of the Kleene star, when obtaining the set of all partitions of w to test its acceptance, we discard the particular decomposition (ε, w). This is because when testing (ε, w), we are passing the unchanged w as the argument of the recursive call; since the argument w does not decrease, we cannot ascertain termination for our function.

Since the set of decompositions of w already contains the particular decomposition (w, ε), note with ε in the second half, we can discard the problematic case (ε, w) because the regular expression ee^* is equivalent to e^*e and because the two decompositions (ε, w), (w, ε) represent the same word.

We thus believe in the termination of our algorithm due to the decrease of the argument w with each successive recursive call, as well as the realisation of various unit tests.

7.2.2 Generate

According to the source material of the course, given a regular expression $E \in RegExp(\Sigma)$, we can formally define the language $L(E)$ as follows:

- $L(\varnothing) = \varnothing$
- $L(\varepsilon) = \{\varepsilon\}$
- $a \in \Sigma \Rightarrow L(a) = \{a\}$
- $L(E) = LE \wedge L(F) = LF \Rightarrow L(EF) = LELF$
- $L(E) = LE \wedge L(F) = LF \Rightarrow L(E + F) = LE \cup LF$
- $L(E) = LE \Rightarrow L(E^*) = (LE)^*$

Intuitively, this states that:

The language of an empty expression is empty; the language of the expression ε is the set containing only the empty word ε; the language of the expression a is the set containing only the word a; the language of the sequence of expressions E and F are the set of words resulting from concatenating each word of $L(E)$ with each word of $L(F)$; the language of the alternation of the expressions E and F is the set containing all words of $L(E)$ and $L(F)$; the language of the Kleene star of expression E is the set of word ε and all words formed through concatenating words of $L(E)$ a finite number of times.

Notice that this formalism explicitly describes a semi-algorithm. The reason is that it guarantees termination for all cases except for the Kleene star.

Our solution is to limit the length of the generated words by passing a maximum size as the argument of our function, this way we guarantee termination for the case of the Kleene star operation.

Listing 7.7: Generate function

```
let rec gen rep ln =
  match rep with
    |Plus(l, r) -> Set.union (lang l ln) (lang r ln)
    |Seq(l, r) -> let left = lang l ln in
      let rigth w = lang r (ln - (length w)) in
      let conc w = concatAll w (rigth w) in
        flatMap (fun lw -> (conc lw)) left
    | Star r -> let exp = lang r ln in
      star exp ln
    | Symb(c) -> if ln > 0 then Set[c] else emptySet
    | Empty -> Set[[]]
    | Zero -> emptySet
in

method generate (length: int): words =
  gen regExpression length
```

The generate method is a relatively direct implementation of the presented formalisms, with the incorporation of our solution in the form of controlling the lengths of our generated words.

As an example, if we were to apply our algorithm to the regular expression $r = 0 + 1^*$ for the maximum size $n = 3$, the resulting set of words would be "ε ", "0", "1", "11" and "111".

We thus believe in the termination of our algorithm since we limit the Kleene star operation to stop when it could only produce words which would exceed the imposed limit.

7.2.3 To Finite Automaton

According to the source material of the course, given a regular expression E, we can formally define the $compile(E)$ function for conversion of E into a finite automaton $A = (S, \Sigma, s, \delta, F)$ as follows:

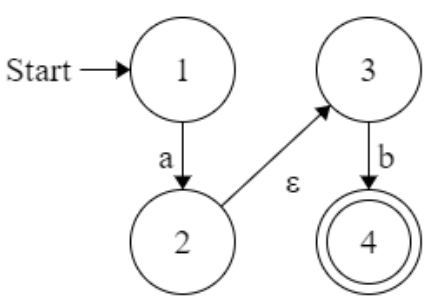
- $Compile(\varnothing) = \{\{1\}, \varnothing, 1, \varnothing, \varnothing\}$
- $Compile(\varepsilon) = \{\{1\}, \varnothing, 1, \varnothing, \{1\}\}$
- $Compile(a) = \{\{1,2\}, \{a\}, 1, \{(1,a) \rightarrow 2\}, \{2\}\}$
- $Compile(EF) = \{SE \cup SF, \{\varepsilon\} \cup \Sigma E \cup \Sigma F, sE, \delta\varepsilon \cup \delta E \cup \delta F, FF\}$ where $\delta\varepsilon = \{(fE, \varepsilon) \rightarrow sF | fE \in FE\}$ and $SE \cap SF = \varnothing$
- $Compile(E + F) = \{\{i\} \cup SE \cup SF, \Sigma E \cup \Sigma F, i, \Delta\varepsilon \cup \Delta E \cup \Delta F, FE \cup FF\}$ where $(i \notin SE \cup SF) \wedge (SE \cap SF = \varnothing)$ and $\Delta\varepsilon = \{(i, \varepsilon, sE), (i, \varepsilon, sF)\}$
- $Compile(E^*) = \{\{i\} \cup SE, \Sigma E, i, \Delta\varepsilon \cup \Delta E, \{i\}\}$ when $i \notin SE$ and $\Delta\varepsilon = \{(i, \varepsilon, sE)\} \cup \{(f, \varepsilon, i) | f \in SF\}$

Intuitively, this states that:

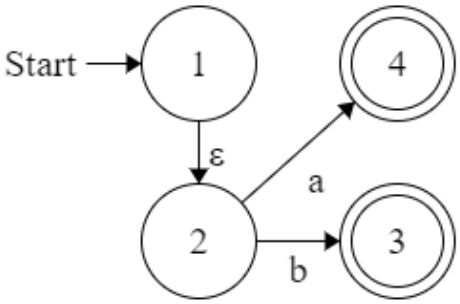
The empty regular expression can be converted into the finite automaton with only one initial state and no transitions nor acceptance states; the ε regular expression can be converted into the finite automaton with no transitions and only one state 1 that is both the initial and an acceptance state; the regular expression consisting only of one symbol a can be converted into the finite automaton with two states 1 and 2, where 1 is the initial state and 2 is the acceptance state, and the transition $(1, a) \rightarrow 2$; the regular expression consisting of the sequence of two regular expressions E and F can be converted into the automaton with all states and transitions of E and F, where its initial state is the initial state of E, its acceptance states are the acceptance states of F, and that foreach acceptance state of E has a ε-*transition* to the initial state of F; the regular expression consisting of the alternation of E and F can be converted into the automaton with all states, acceptance states and transitions of E and F, whose new initial state is a state i and with two ε-*transitions* from i to the initial states of E and F; the regular expression consisting of the Kleene star of E can be converted into the automaton with all states and transitions of E but whose initial state and single acceptance state are now the state i, with a ε-*transition* from i to the initial state of E and with a ε-*transition* from each acceptance state of E to the state i.

For example, the finite automata in figures 7.9a, 7.9b and 7.9c are the result of applying the compile function to regular expressions $r1 = ab$, $r2 = a + b$ and $r3 = a^*$,respectively.

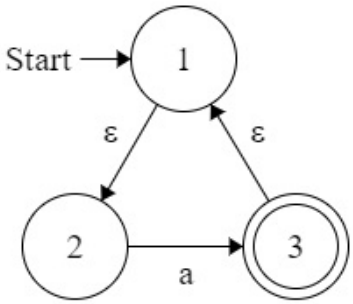
The defined formalism essentially describes an algorithm, which naturally guarantees termination. As such, our implementation of the regular expression to automaton conversion is a relatively direct translation of the defined algorithm.



(a) converted from "ab"



(b) converted from "a+b"



(c) converted from "a*"

Figure 7.9: Automata for conversion from regular expression example

7.2.4 To regular grammar

The conversion of regular expressions to regular grammars is an algorithm that is not discussed in the materials of the course, nevertheless we decided that it would be of interest to implement it in our program. The formalisms we based our solution on can be found in the book "introdução à teoria da computação" by Cristina Sernadas[35].

We can formally define the function $compile(E)$ for conversion of a regular expression E to the regular grammar $G = \{V, T, P, S\}$, as follows:

First, we define our base cases.

- $Compile(\varnothing) = \{\{A, B\}, \varnothing, \{A \rightarrow B, B \rightarrow A\}, A\}$
- $Compile(\varepsilon) = \{\{A\}, \varnothing, \{A \rightarrow \varepsilon\}, A\}$
- $Compile(a) = \{\{A\}, \{a\}, \{A \rightarrow a\}, A\}$

Intuitively, this states that:

The empty regular expression can be converted to a regular grammar with no alphabet; the ε regular expression can be converted into the regular grammar with an initial variable A and with a single rule $(A \rightarrow \varepsilon)$; the regular expression consisting of a single symbol a can be converted into the regular grammar with an initial variable A, the alphabet consisting only of a and the rule $(A \rightarrow a)$.

Then, given two regular expressions E and F, for $compile(E) = \{V1, T1, P1, S1\}$ and

$compile(F) = \{V2, T2, P2, S2\}$, we can define the remaining cases.

• $Compile(EF) = \{V1 \cup V2, T1 \cup T2, Ps, S1\}$ where $Ps = \{A \to a : A \to a \notin T1, a \notin \varepsilon\} \cup \{A \to aS2 : A \to a \in P1, a \in T1\} \cup \{A \to a : A \to a \in P2\} \cup \{A \to a : S2 \to a \in P2, A \to \varepsilon \in P1\}$

• $Compile(E+F) = \{(V1 \cup V2 \cup S)\tilde{}\{S1, S2\}, T1 \cup T2, P1 \cup P2 \cup Ps, S\}$ where $Ps = \{S- > S1\} \cup \{S \to S2\}$

• $Compile(E^*) = \{V1, T1, Ps, S1\}$ where $Ps = \{A \to a : A \to a \in P1\} \cup \{A \to aS : A \to a \in P1, a \in T1\} \cup \{S \to \varepsilon\} \cup \{A \to a : A \to \varepsilon \in P1, S1 \to a \in P1\}$

Intuitively, for $G1 = compile(E)$ and $G2 = compile(F)$, we can interpret this definition as follows:

The sequence of the regular expressions E and F can be converted into the regular grammar whose variables and alphabet are the union of the variables and the union of the alphabets of both $G1$ and $G2$, with the initial variable of $G1$, and whose rules are the union of the rules of $G1$ and $G2$ where all rules $(A \to a)$ of $G1$ are replaced by $(A \to aS2)$, where $S2$ is the initial state of $G2$; the alternation of the regular expressions E and F can be converted into the regular grammar whose variables and alphabet are the union of the variables and alphabets of both $G1$ and $G2$, whose initial variable is a new variable S, and whose rules are the union of the rules of $G1$ and $G2$ with the addition of the rules $\{S- > S1\}$ and $\{S- > S2\}$; the Kleene star of regular expression E can be converted into the regular grammar that is equal to $G1$ except that for every rule of $G1$ with the format $(A \to a)$ we add the rule $(A \to aS)$ where S is the initial variable of G, as well as adding the rule $(A \to \varepsilon)$.

For example, the regular expressions $r1 = ab$, $r2 = a+b$ and $r3 = a^*$ can be respectively converted by the above algorithm to the context-free grammars $cfg1 = \{\{S, A\}, \{a, b\}, \{S- > aA, A- > b\}, S\}$, $cfg2 = \{\{S\}, \{a, b\}, \{S- > a, S- > b\}, S\}$ and $cfg3 = \{\{S\}, \{a\}, \{S- > aS, S- > \varepsilon\}, S\}$.

The described formalism essentially describes an algorithm, which naturally guarantees termination. As such, our implementation of the regular expression to regular grammar conversion is a relatively direct translation of the defined algorithm.

7.3 Context-free grammar

According to the source material of the course, we can define a context-free grammar (CFG) as a quadruplet $G = \{V, T, P, S\}$, where V is the set of variables, T is the alphabet, P is the set of rules where $P \subseteq V \times (V \cup T)^*$, and S is the initial variable.

For the rules of a context-free grammar, we refer to the variable as the head and its possible substitution as the body.

7.3.1 Accept

Informally, we can define an algorithm for testing the acceptance of a word w as follows:

We first define the function *nextGeneration*, which receives set of pseudo-words (a list of symbols from either the alphabet or variable set of the grammar) as an argument. Then, it generates the set of all pseudo-words obtained from applying all valid combinations of rules to each pseudo-word of the argument. After generating the new set of pseudo-words, we then filter each pseudo-word whose number of symbols from the alphabet exceeds the length of w. The function is concluded by filtering the resulting pseudo-words based on their prefix and suffix (respectively, their sub-words to the left and to the right of their left-most variable and their right-most variable) compared to the word w.

The accept method is thus the implementation of the *nextGeneration* function applied recursively to each new output. Termination occurs when no new pseudo-word is generated, and w is considered accept if it occurs in the output set of our function.

Listing 7.8: Accept function

```
1 method accept (testWord:word) : bool =
2   let nextGeneration pws =
3     let npws = flatMap (fun w -> subVar w vs rs) pws in
4     let npws = filter (fun w -> not (exceedsMaxLen w l alph)) npws in
5       filter (fun w -> toKeep w testWord ) npws
6   in
7   let res = historicalFixedPoint nextGeneration set[initialVariable] in
8     exists (fun x -> x = testWord ) res
```

The accept function is a relatively direct translation of the presented definitions.

7.3.2 Generate

We can formally define the language generated by the context-free grammar G as $L(G) = \{w \in T^* | S \Rightarrow G^* w\}$ where $\Rightarrow G^*$ is the reflexive and transitive closure of $\Rightarrow G$.

Informally, this states that the language of G is the set of all words obtained by consecutively applying the possible rules of G, starting on its initial variable.

By interpreting this definition as a mechanical procedure, we could say that it describes a semi-algorithm, as the language defined by G could be infinite, thus our algorithm could never terminate.

To guarantee termination, we impose that our algorithm will only generate words up to a given length n.

Listing 7.9: Generate function

```
1 method generate (length:int) : words =
2   let nextGeneration pws =
3     let npws = Set.flatMap (fun w -> subVar w vs rs) pws in
4       filter (fun w -> not (exceedsMaxLen w length alph)) npws
5   in
6   historicalFixedPoint nextGeneration set[initialVariable]
```

As an example, if we were to generate all words up to length 3 for the context-free grammar $\{\{S,A\},\{a,b,c\},\{S->a,S->bA,A->c,A->cS\},S\}$, the result would be the set of words a, bc and bca.

The generate function is a relatively direct translation of the presented definition.

7.3.3 To Finite Automaton

By analysing the formal definition of context-free grammars and comparing it to the formal definition of finite automaton, we observe that they are very similar.

To better explain the entire conversion algorithm, we must first define the function $ruleToTrans$, which transforms the rules of a right-linear grammar into transitions of an equivalent automaton.

Listing 7.10: RuleToTransition function

```
1 ruleToTrans rh rb =
2   match rb with
3     |[c;v] when belongs c alp && belongs v vrs -> Set[(rh,c,v)]
4     |[v] when belongs v vrs -> Set[(rh,epsilon,v)]
5     |[c] when belongs c alp -> Set[(rh,c,accSt)]
6     |[e] when e = epsilon -> Set[(rh,epsilon,accSt)]
```

Given a rule r consisting of the head rh and the body rb, where v is a variable and c is a symbol of the alphabet, we define $ruleToTrans$ as follows:

When r is in the form $(v1->cv2)$, we can convert r into the transition $\delta(v1,c)=v2$; when r is in the form $(v1->v2)$, we can convert r into the transition $\delta(v1,\varepsilon)=v2$; when r is in the form $(v->c)$, we can convert r into the transition $\delta(v,c)=S$, where S is the acceptance state of our pretended automaton; when r is in the form $(v1->\varepsilon)$, we can convert r into the transition $\delta(v,\varepsilon)=S$.

We can thus informally define the function convert for conversion of a right-linear grammar G to a finite automaton A as follows:

The states of A will consist of the variables of G plus the new acceptance state S; the alphabet of A will simply be the alphabet of G; the acceptance state of A will be the new state S; the initial state of A will be the initial variable of G; the transitions of A will be based on the transformation of the rules of G according to our function $ruleToTrans$.

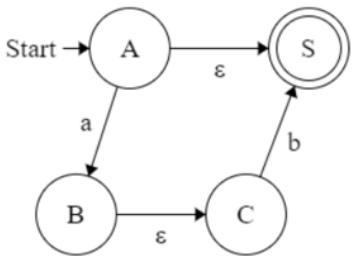


Figure 7.10: Automaton for example of conversion from CFG

As an example, given the regular grammar $G = \{\{A, B, C\}, \{a, b\}, \{A -> \varepsilon, A -> aB, B -> C, C -> b\}, A\}$, the automaton in Figure 7.10 would be the resulting automaton of our algorithm applied to G.

Our implementation of the conversion of a right-linear grammar to its equivalent finite automaton is a relatively direct translation of the defined algorithm.

7.4 Exercises

In our program, we also allow for a partial definition of languages through the module exercises. Essentially, an exercise consists of two sets of words denominated *inside* and *outside*. All words from *inside* belong to the language partially defined by the exercise, while for *outside*, none of its words belongs to said language.

The purpose of this module is to proportionate classes with an interactive process of effectuating exercises over the various studied FLAT mechanism.

7.4.1 CheckExercise

All other mechanisms inherit the method checkExercise, which basically, given an exercise e tests if all words of *inside* are accepted by the mechanism and if no word from *outside* is accepted by the mechanism. If the predicate fails, we can say that, if the exercise was well made, that the language of the mechanism in question is not equal to the language defined by e.

We achieve this by simply calling the accept method of the pretended mechanism for all words of the exercise.

As an example, given an exercise Ex where $inside = \{$"a","ab","abb"$\}$ and $outside = \{$"b","aba","ε"$\}$, when tested for the regular expression $Re = ab^*$, the results would be that all words of *inside* are accepted by Re, while no word from *outside* is accepted by Re. This would thus indicate that the language partially defined by ex is part of the language of Re.

Chapter 8

Interactive command-line interface

The OCaml-Flat tool is planned to be used in conjunction with a graphical interface that is outside the scope of our project. However, one of our main goals for our program was for it to be usable in the context of an OCaml interpreter. This way, the tool could be used as a stand-alone program with which users can manipulate our various FLAT concepts according to their interests.

As such, we implemented an interface for the usage of our program at the top-level. The following is a list of execution examples for each of the implemented functions, accompanied by their respective description.

The reader can find a manual of all the implemented functions in the appendix.

8.1 Finite Automaton

8.1.1 Function fa_load

The fa_load function receives as argument a Json file and returns the finite automaton defined in said file.

Listing 8.1: fa_load example

```
# let fa = fa_load "fa_example.json";;
  val fa : finiteAutomaton =
     {alphabet = ['a'; 'b'; 'c']; allStates = ["A"; "B"; "START"; "SUCCESS"];
      initialState = "START";
      transitions =
      [("A", 'a', "A"); ("A", 'c', "SUCCESS"); ("B", 'b', "B");
       ("B", 'c', "SUCCESS"); ("START", 'a', "A"); ("START", 'b', "B");
        ("SUCCESS", 'a', "START")];
     acceptStates = ["SUCCESS"]}
```

8.1.2 Function fa_accept

The fa_accept function receives as arguments a finite automaton and a String representing a word. It is used to test if the word belongs to the language defined by the given automaton.

Listing 8.2: fa_accept example

```
# let fa = fa_load "fa_example.json";;
  val fa : finiteAutomaton =
    {alphabet = ['a'; 'b']; allStates = ["START"; "SUCCESS"];
     initialState = "START";
     transitions = [("START", 'a', "START"); ("START", 'b', "SUCCESS")];
    acceptStates = ["SUCCESS"]}

# fa_accept fa "aab";;
- : bool = true
```

8.1.3 Function fa_traceAccept

The fa_traceAccept function receives as arguments a finite automaton and a String representing a word. It is used to trace the rationale behind the conclusion of whether the word is accepted by the automaton or not.

Listing 8.3: fa_traceAccept example

```
# let fa = fa_load "fa_example.json";;
val fa : finiteAutomaton =
  {alphabet = ['a'; 'b']; allStates = ["A"; "B"; "START"; "SUCCESS"];
   initialState = "START";
   transitions =
    [("A", 'b', "SUCCESS"); ("START", 'a', "A"); ("START", 'b', "B")];
   acceptStates = ["SUCCESS"]}

# fa_traceAccept fa "ab";;
('ab',[START;]);('b',[A;B;]);('',[SUCCESS;]);
- : unit = ()
```

8.1.4 Function fa_generate

The fa_generate function receives as arguments a finite automaton and an integer n. It produces the set of all words with the maximum size of n that belong to the language of the automaton.

Listing 8.4: fa_generate example

```
# let fa = fa_load "fa_example.json";;
  val fa : finiteAutomaton =
    {alphabet = ['a'; 'b']; allStates = ["START"; "SUCCESS"];
     initialState = "START";
```

```
5       transitions = [("START", 'a', "START"); ("START", 'b', "SUCCESS")];
6      acceptStates = ["SUCCESS"]}
7
8  # fa_generate fa 4;;
9  - : string list = ["b"; "ab"; "aab"; "aaab"]
```

8.1.5 Function fa_reachable

The fa_reachable function receives as argument a finite automaton fa. It produces the set of all reachable states of fa.

Listing 8.5: fa_reachable example

```
1  # let fa = fa_load "fa_example.json";;
2    val fa : finiteAutomaton =
3     {alphabet = ['a'; 'b']; allStates = ["A"; "B"; "START"; "SUCCESS"];
4       initialState = "START";
5       transitions = [("START", 'b', "A"); ("START", 'b', "SUCCESS")];
6       acceptStates = ["SUCCESS"]}
7
8  # fa_reachable fa;;
9  - : state list = ["A"; "START"; "SUCCESS"]
```

8.1.6 Function fa_productive

The fa_productive function receives as argument a finite automaton fa. It produces the set of all productive states of the automaton fa.

Listing 8.6: fa_productive example

```
1  # let fa = fa_load "fa_example.json";;
2    val fa : finiteAutomaton =
3     {alphabet = ['a'; 'b']; allStates = ["A"; "B"; "START"; "SUCCESS"];
4       initialState = "START";
5       transitions = [("START", 'b', "A"); ("START", 'b', "SUCCESS")];
6       acceptStates = ["SUCCESS"]}
7
8  # fa_productive fa;;
9  - : state list = ["START"; "SUCCESS"]
```

8.1.7 Function fa_clean

The fa_clean function receives as argument a finite automaton fa. It produces the automaton resulting from eliminating all non-useful states of fa.

Listing 8.7: fa_clean example

```
1  # let fa = fa_load "fa_example.json";;
2    val fa : finiteAutomaton =
```

```
3    {alphabet = ['a'; 'b']; allStates = ["A"; "B"; "START"; "SUCCESS"];
4      initialState = "START";
5      transitions = [("START", 'b', "A"); ("START", 'b', "SUCCESS")];
6      acceptStates = ["SUCCESS"]}
7
8  # fa_clean fa;;
9  - : finiteAutomaton =
10 {alphabet = ['b']; allStates = ["START"; "SUCCESS"]; initialState = "START";
11  transitions = [("START", 'b', "SUCCESS")]; acceptStates = ["SUCCESS"]}
```

8.1.8 Function fa_toDeter

The fa_toDeter function receives as argument a finite automaton fa. It produces the automaton resulting from the determinization of fa.

Listing 8.8: fa_ToDeter example

```
1  # let fa = fa_load "fa_example.json";;
2  val fa : finiteAutomaton =
3    {alphabet = ['a'; 'b'; 'c']; allStates = ["A"; "B"; "START"; "SUCCESS"];
4     initialState = "START";
5     transitions =
6      [("A", 'a', "SUCCESS"); ("B", 'b', "SUCCESS"); ("START", 'c', "A");
7       ("START", 'c', "B")];
8     acceptStates = ["SUCCESS"]}
9
10 # fa_toDeter fa;;
11 - : finiteAutomaton =
12 {alphabet = ['a'; 'b'; 'c']; allStates = ["A_B"; "START"; "SUCCESS"];
13  initialState = "START";
14  transitions =
15   [("A_B", 'a', "SUCCESS"); ("A_B", 'b', "SUCCESS"); ("START", 'c', "A_B")];
16  acceptStates = ["SUCCESS"]}
```

8.1.9 Function fa_isDeter

The fa_isDeter function receives as argument a finite automaton fa. It verifies if fa is deterministic or not.

Listing 8.9: fa_isDeter example

```
1  # let fa = fa_load "fa_example.json";;
2    val fa : finiteAutomaton =
3     {alphabet = ['a'; 'b']; allStates = ["A"; "B"; "START"; "SUCCESS"];
4      initialState = "START";
5        transitions =
6        [("A", 'a', "B"); ("B", 'b', "SUCCESS"); ("START", 'b', "A");
7        ("START", 'b', "B")];
8      acceptStates = ["SUCCESS"]}
9
```

```
# fa_isDeter fa;;
- : bool = false
```

8.1.10 Function fa_minimize

The fa_minimize function receives as argument a finite automaton fa. It produces the automaton resulting from the minimization of fa.

Listing 8.10: fa_minimize example

```
# let fa = fa_load "fa_example.json";;
val fa : finiteAutomaton =
  {alphabet = ['a'; 'b']; allStates = ["S1"; "S2"; "S3"; "S4"; "S5"];
   initialState = "S1";
   transitions =
    [("S1", 'a', "S2"); ("S1", 'b', "S3"); ("S2", 'a', "S3");
     ("S2", 'b', "S4"); ("S3", 'a', "S2"); ("S3", 'b', "S4");
     ("S4", 'a', "S2"); ("S4", 'a', "S5"); ("S4", 'b', "S3")];
   acceptStates = ["S4"]}

# fa_minimize fa;;
- : finiteAutomaton =
{alphabet = ['a'; 'b']; allStates = ["S1"; "S2"; "S4"]; initialState = "S1";
 transitions =
  [("S1", 'a', "S2"); ("S1", 'b', "S2"); ("S2", 'a', "S2");
   ("S2", 'b', "S4"); ("S4", 'a', "S2"); ("S4", 'b', "S2")];
 acceptStates = ["S4"]}
```

8.1.11 Function fa_toRegex

The fa_toRegex function receives as argument a finite automaton fa. It produces the regular expression equivalent to fa.

Listing 8.11: fa_toRegex example

```
# let fa = fa_load "fa_example.json";;
val fa : finiteAutomaton =
  {alphabet = ['a'; 'b'; 'c']; allStates = ["A"; "B"; "START"; "SUCCESS"];
   initialState = "START"; transitions = [("START", 'b', "SUCCESS")];
   acceptStates = ["SUCCESS"]}

# fa_toRegex fa;;
- : regularExpression =
"b+!!*b+b!*!+(!+!!*!+b!*!)(!+!!*!+b!*!)*(b+!!*b+b!*!)+(b+!!*b+b!*!)(!+!!*!+!!*b)
    ↪ *(!+!!*!+!!*b)"
```

8.2 Regular Expression

8.2.1 Function re_load

The re_load function receives as argument a Json file and returns the regular expression defined in said file.

Listing 8.12: re_load example

```
# let re = re_load "re_example.json";;
val re : regularExpression = "a*+bc"
```

8.2.2 Function re_alphabet

The re_load function receives as argument a regular expression and returns its alphabet.

Listing 8.13: re_alphabet example

```
# let re = re_load "re_example.json";;
val re : regularExpression = "a*+bc"

# re_alphabet re;;
- : symbol list = ['a'; 'b'; 'c']
```

8.2.3 Function re_accept

The re_accept function receives as arguments a regular expression *re* and a string representing a word. It verifies if the word belongs to the language of *re*.

Listing 8.14: re_accept example

```
# let re = re_load "re_example.json";;
val re : regularExpression = "a*+bc"

# re_accept re "aa";;
- : bool = true
```

8.2.4 Function re_trace

The re_trace function receives as arguments a regular expression *re* and a string representing a word. It is used to trace the rationale behind the conclusion of whether the word is accepted by *re* or not.

Listing 8.15: re_trace example

```
# let re = re_load "re_example.json";;
val re : regularExpression = "a*+bc"

# re_trace re "bc";;
bc -> a*+bc
```

```
   bc -> bc
      b -> b
      c -> c
- : unit = ()
```

8.2.5 Function re_generate

The re_generate function receives as arguments a regular expression *re* and an integer *n*. It produces the set of all words with the maximum size of *n* that belong to the language of *re*.

Listing 8.16: re_generate example

```
# let re = re_load "re_example.json";;
val re : regularExpression = "a*+bc"

# re_generate re 3;;
- : string list = [""; "a"; "aa"; "aaa"; "bc"]
```

8.2.6 Function re_simplify

The re_simplify function receives as argument a regular expression *re*. It produces a simplified regular expression equivalent to *re*.

Listing 8.17: re_simplify example

```
# let re = re_load "re_example.json";;
val re : regularExpression = "a+a*"

# re_simplify re;;
- : regularExpression = "a*"
```

8.2.7 Function re_toFa

The re_simplify function receives as argument a regular expression *re*. It produces the automaton equivalent to *re*.

Listing 8.18: re_toFA example

```
# let re = re_load "re_example.json";;
val re : regularExpression = "a*+bc"

# re_toFA re;;
- : finiteAutomaton =
{alphabet = ['a'; 'b'; 'c'];
 allStates =
  ["new_St00"; "new_St01"; "new_St02"; "new_St03"; "new_St04"; "new_St05";
   "new_St06"; "new_St07"];
 initialState = "new_St07";
```

```
 transitions =
  [("new_St00", 'a', "new_St01"); ("new_St01", '~', "new_St02");
   ("new_St02", '~', "new_St00"); ("new_St03", 'b', "new_St04");
   ("new_St04", '~', "new_St05"); ("new_St05", 'c', "new_St06");
   ("new_St07", '~', "new_St02"); ("new_St07", '~', "new_St03")];
 acceptStates = ["new_St02"; "new_St06"]}
```

8.3 Context-Free Grammar

8.3.1 Function cfg_load

The cfg_load function receives as argument a Json file and returns the context free grammar it defines.

Listing 8.19: cfg_load example

```
# let cfg = cfg_load "cfg_example.json";;
val cfg : contextFreeGrammar =
  {alphabet = ['0'; '1']; variables = ['F'; 'P'; 'S']; initial = 'S';
   rules = ["F␣->␣1"; "F␣->␣1F"; "P␣->␣0"; "S␣->␣0F"; "S␣->␣0P"]}
```

8.3.2 Function cfg_accept

The cfg_accept function receives as arguments a CFG and a string representing a word. It verifies if the word belongs to the language of the CFG.

Listing 8.20: cfg_accept example

```
# let cfg = cfg_load "cfg_example.json";;
val cfg : contextFreeGrammar =
  {alphabet = ['0'; '1']; variables = ['F'; 'P'; 'S']; initial = 'S';
   rules = ["F␣->␣1"; "F␣->␣1F"; "P␣->␣0"; "S␣->␣0F"; "S␣->␣0P"]}

# cfg_accept cfg "011";;
- : bool = true
```

8.3.3 Function cfg_trace

The cfg_trace function receives as arguments a CFG and a string representing a word. It is used to trace the rationale behind the conclusion of whether the word is accepted by the CFG or not.

Listing 8.21: cfg_trace example

```
# let cfg = cfg_load "cfg_example.json";;
val cfg : contextFreeGrammar =
  {alphabet = ['0'; '1']; variables = ['F'; 'P'; 'S']; initial = 'S';
   rules = ["F␣->␣1"; "F␣->␣1F"; "P␣->␣0"; "S␣->␣0F"; "S␣->␣0P"]}
```

```
5
6  # cfg_trace cfg "011";;
7  [S;]
8  [0F;0P;]
9  [01F;]
10 [011;011F;]
11 - : unit = ()
```

8.3.4 Function cfg_generate

The cfg_trace function receives as arguments a CFG and an integer *n*. It produces the set of all words with the maximum size of *n* that belong to the language of the CFG.

Listing 8.22: cfg_generate example

```
1  # let cfg = cfg_load "cfg_example.json";;
2  val cfg : contextFreeGrammar =
3    {alphabet = ['0'; '1']; variables = ['F'; 'P'; 'S']; initial = 'S';
4     rules = ["F␣->␣1"; "F␣->␣1F"; "P␣->␣0"; "S␣->␣0F"; "S␣->␣0P"]}
5
6  # cfg_generate cfg 3;;
7  - : string list = ["00"; "01"; "011"]
```

8.3.5 Function cfg_toFA

The cfg_toFA function receives as argument a CFG. It produces the automaton equivalent to the CFG.

Listing 8.23: cfg_toFA example

```
1  # let cfg = cfg_load "cfg_example.json";;
2  val cfg : contextFreeGrammar =
3    {alphabet = ['0'; '1']; variables = ['F'; 'P'; 'S']; initial = 'S';
4     rules = ["F␣->␣1"; "F␣->␣1F"; "P␣->␣0"; "S␣->␣0F"; "S␣->␣0P"]}
5
6  # cfg_toFA cfg;;
7  - : finiteAutomaton =
8  {alphabet = ['0'; '1']; allStates = ["AccSt"; "F"; "P"; "S"];
9   initialState = "S";
10  transitions =
11   [("F", '1', "AccSt"); ("F", '1', "F"); ("P", '0', "AccSt");
12    ("S", '0', "F"); ("S", '0', "P")];
13  acceptStates = ["AccSt"]}
```

8.3.6 Function cfg_toRe

The cfg_toRe function receives as argument a CFG. It produces the regular expression equivalent to the CFG.

Listing 8.24: cfg_toRe example

```
# let cfg = cfg_load "cfg_example.json";;
val cfg : contextFreeGrammar =
  {alphabet = ['0'; '1']; variables = ['F'; 'S']; initial = 'S';
   rules = ["F␣->␣1"; "S␣->␣0F"]}

# cfg_toRe cfg;;
- : regularExpression =
"!+!!*!+0!*1+(!+!!*!+0!*!)(!+!!*!+0!*!)*(!+!!*!+0!*1)+(!+!!*!+0!*1)(!+!!*!+!!*1)
    ↪ *(!+!!*!+!!*1)+(0+!!*!+!!*0+0!*!)(!+!!*!+!!*0+1!*!)*(1+!!*!+!!*1+1!*!)
    ↪ +(!+!!*!+0!*!+(!+!!*!+0!*!)(!+!!*!+0!*!)*(!+!!*!+0!*!)+(!+!!*!+0!*1)(!+!!*!+!!*1)
    ↪ *(!+!!*!)+(0+!!*!+!!*0+0!*!)(!+!!*!+!!*0+1!*!)*(!+!!*!+1!*!))(!+!!*!"... (* string
    ↪  length 1509; truncated *)
```

8.4 Exercise

8.4.1 Function exer_load

The exer_load function receives as argument a Json file and returns its defined exercise.

Listing 8.25: exer_load example

```
# let ex = exer_load "enum_example.json";;
val ex : exercise =
  {inside = [""; "a"; "b"; "bb"; "bbbb"]; outside = ["aa"; "ab"; "bba"; "c"]}
```

8.4.2 Function exer_testFA

The exer_testFA function receives as arguments an exercise e and a finite automaton fa. It verifies if the language of e is compatible with the language of fa.

Listing 8.26: exer_testFA example

```
# let ex = exer_load "enum_example.json";;
val ex : exercise =
  {inside = [""; "a"; "b"; "bb"; "bbbb"]; outside = ["aa"; "ab"; "bba"; "c"]}

# let fa = fa_load "fa_example.json";;
val fa : finiteAutomaton =
  {alphabet = ['a'; 'b'; 'c']; allStates = ["A"; "B"; "START"; "SUCCESS"];
   initialState = "START";
   transitions =
    [("A", 'a', "SUCCESS"); ("B", 'b', "SUCCESS"); ("START", 'c', "A");
     ("START", 'c', "B")];
   acceptStates = ["SUCCESS"]}

# exer_testFA ex fa;;
- : bool = false
```

8.4.3 Function exer_testFAFailures

The exer_testFAFailures function receives as arguments an exercise e and a finite automaton fa. It produces all words of e that make its language incompatible with the language of fa.

Listing 8.27: exer_testFAFailures example

```
# let ex = exer_load "enum_example.json";;
val ex : exercise =
  {inside = [""; "a"; "b"; "bb"; "bbbb"]; outside = ["aa"; "ab"; "bba"; "c"]}

# let fa = fa_load "fa_example.json";;
val fa : finiteAutomaton =
  {alphabet = ['a'; 'b'; 'c']; allStates = ["A"; "B"; "START"; "SUCCESS"];
   initialState = "START";
   transitions =
    [("A", 'a', "SUCCESS"); ("B", 'b', "SUCCESS"); ("START", 'c', "A");
     ("START", 'c', "B")];
   acceptStates = ["SUCCESS"]}

# exer_testFAFailures ex fa;;
- : string list * string list = ([""; "a"; "b"; "bb"; "bbbb"], [])
```

8.4.4 Function exer_testRe

The exer_testRe function receives as arguments an exercise e and a regular expression re. It verifies if the language of e is compatible with the language of re.

Listing 8.28: exer_testRe example

```
# let ex = exer_load "sss.json";;
val ex : exercise =
  {inside = [""; "a"; "b"; "bb"; "bbbb"]; outside = ["aa"; "ab"; "bba"; "c"]}

# let re = re_load "re_abc.json";;
val re : regularExpression = "a*+bc"

# exer_testRe ex re;;
- : bool = false
```

8.4.5 Function exer_testReFailures

The exer_testReFailures function receives as arguments an exercise e and a regular expression re. It produces all words of e that make its language incompatible with the language of re.

Listing 8.29: exer_testReFailures

```
# let ex = exer_load "sss.json";;
```

```
val ex : exercise =
  {inside = [""; "a"; "b"; "bb"; "bbbb"]; outside = ["aa"; "ab"; "bba"; "c"]}

# let re = re_load "re_abc.json";;
val re : regularExpression = "a*+bc"

# exer_testReFailures ex re;;
- : string list * string list = (["b"; "bb"; "bbbb"], ["aa"])
```

8.4.6 Function exer_testCFG

The exer_testCFG function receives as arguments an exercise *e* and a CFG. It verifies if the language of *e* is compatible with the language of the CFG.

Listing 8.30: exer_testCfg

```
# let ex = exer_load "sss.json";;
val ex : exercise =
  {inside = [""; "a"; "b"; "bb"; "bbbb"]; outside = ["aa"; "ab"; "bba"; "c"]}

# let cfg = cfg_load "cfg_abc.json";;
val cfg : contextFreeGrammar =
  {alphabet = ['0'; '1']; variables = ['F'; 'S']; initial = 'S';
   rules = ["S␣->␣01"]}

# exer_testCfg ex cfg;;
- : bool = false
```

8.4.7 Function exer_testCfgFailures

The exer_testCfgFailures function receives as arguments an exercise *e* and a CFG. It produces all words of *e* that make its language incompatible with the language of the CFG.

Listing 8.31: exer_testCfgFailures

```
# let ex = exer_load "sss.json";;
val ex : exercise =
  {inside = [""; "a"; "b"; "bb"; "bbbb"]; outside = ["aa"; "ab"; "bba"; "c"]}

# let cfg = cfg_load "cfg_abc.json";;
val cfg : contextFreeGrammar =
  {alphabet = ['0'; '1']; variables = ['F'; 'S']; initial = 'S';
   rules = ["S␣->␣01"]}

# exer_testCfgFailures ex cfg;;
- : string list * string list = ([""; "a"; "b"; "bb"; "bbbb"], [])
```

Chapter 9

Conclusion

For this dissertation, we proposed to develop an OCaml library of types and functions for the manipulation of various FLAT concepts. This library could then be used as a pedagogical tool for the teaching of FLAT.

9.1 Summary

In the first half of this document, we explained our reasoning for using the functional programming style in the implementation of our tool, we discussed the importance of the materials of FLAT in the learning of the computer science students, and we compared our solution to the already existing body of pedagogical tools in the area of FLAT. In the second half, we explained the architecture of our solution, followed by a discussion and analysis on the various algorithms that were part of our program, focusing greatly on the implementation of said algorithms compared to their formal definitions. The algorithms implemented consisted on those referring to FA, regular expressions and CFGs.

9.2 End Results

The end result of this dissertation was a pedagogical tool whose implementation of the various CFG concepts closely followed the formal definitions presented in the materials of the computation theory course in our faculty. This proximity was further accentuated by our usage of the OCaml language following the functional programming style. To reassure correctness and termination of our algorithms, various unit tests were conducted during the development of this project, which in conjunction with the proximity to the source materials, allowed us for a greater confidence in our results.

9.3 Future Work

Since in this dissertation we ultimately focused on implementing algorithms of only the three previously referred FLAT mechanisms, an obvious possibility for the extension of our program would be for the integration of the remaining FLAT concepts taught in the course, namely push-down automata and Turing machines. Another possible worthwhile addition to our tool could be the implementation other FLAT operations that are not covered in the materials of the course. Finally, seeing as this tool is intended for use in future editions of the Theory of Computation course in our faculty, it would be interesting to conduct an enquiry on the students, asking them on their opinions and experience when given the option of aiding their learning process with our tool. This could allow us to better gauge the impact our tool could have in helping students to better learn OCaml.

Bibliography

[1] M. Sipser. *Introduction to the Theory of Computation*. Third. Boston, MA: Course Technology, 2013. ISBN: 113318779X.

[2] P. Chakraborty, P. C. Saxena, and C. P. Katti. "Fifty Years of Automata Simulation: A Review." In: *ACM Inroads* 2.4 (Dec. 2011), 59–70. ISSN: 2153-2184. DOI: 10.1145/2038876.2038893. URL: https://doi.org/10.1145/2038876.2038893.

[3] C. I. Chesñevar, M. L. Cobo, and W. Yurcik. "Using Theoretical Computer Simulators for Formal Languages and Automata Theory." In: *SIGCSE Bull.* 35.2 (June 2003), 33–37. ISSN: 0097-8418. DOI: 10.1145/782941.782975. URL: https://doi.org/10.1145/782941.782975.

[4] A. Ravara. *Lecture notes in Computational Theory*. These notes are currently private and are only available for authentified students or teachers. 2019.

[5] *LAP*. URL: http://ctp.di.fct.unl.pt/miei/lap/teoricas/02.html (visited on 03/23/2020).

[6] R. Rojas. *A Tutorial Introduction to the Lambda Calculus*. 2015. arXiv: 1503.09060 [cs.LO].

[7] P. Hudak. "Conception, Evolution, and Application of Functional Programming Languages." In: *ACM Comput. Surv.* 21.3 (1989), 359–411. ISSN: 0360-0300. DOI: 10.1145/72551.72554. URL: https://doi.org/10.1145/72551.72554.

[8] *OCaml – OCaml*. URL: https://ocaml.org/ (visited on 03/23/2020).

[9] M. Vujosevic-Janicic and D. D. Tosic. "THE ROLE OF PROGRAMMING PARADIGMS IN THE FIRST PROGRAMMING COURSES." In: 2008.

[10] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press, 2018. ISBN: 0262534800.

[11] J. Hughes. "Why Functional Programming Matters." In: *Computer Journal* 32.2 (1989), pp. 98–107.

[12] *Online Turing Machine Simulator*. URL: https://turingmachinesimulator.com/ (visited on 03/23/2020).

[13] R. W. Coffin, H. E. Goheen, and W. R. Stahl. "Simulation of a Turing machine on a digital computer." In: *AFIPS '63 (Fall)*. 1963.

[14] I. Gilbert and J. Cohen. "A Simple Hardware Model of a Turing Machine: Its Educational Use." In: *Proceedings of the ACM Annual Conference - Volume 1*. ACM '72. Boston, Massachusetts, USA: Association for Computing Machinery, 1972, 324–329. ISBN: 9781450374910. DOI: 10.1145/800193.569940. URL: https://doi.org/10.1145/800193.569940.

[15] M. Cernanský, M. Nehéz, D. Chuda, and I. Polický. "On Using of Turing Machine Simulators in Teaching of Theoretical Computer Science." In: 1 (Oct. 2008), pp. 301–312.

[16] J. Barwise and J. Etchemendy. *Turing's World 3.0: An Introduction to Computability Theory*. USA: University of Chicago Press, 1993. ISBN: 1881526100.

[17] D. G. Hannay. "Hypercard Automata Simulation: Finite-State, Pushdown and Turing Machines." In: *SIGCSE Bull.* 24.2 (June 1992), 55–58. ISSN: 0097-8418. DOI: 10.1145/130962.130971. URL: https://doi.org/10.1145/130962.130971.

[18] M. Losacco and S. H. Ttodger. "Flap:: a tool for drawing and simulating automata." In: 1993, pp. 310–317.

[19] L. F. M. Vieira, M. A. M. Vieira, and N. J. Vieira. "Language emulator, a helpful toolkit in the learning process of computer theory." In: *SIGCSE*. 2004.

[20] E. Luce and S. H. Rodger. "A visual programming environment for Turing machines." In: *Proceedings 1993 IEEE Symposium on Visual Languages*. 1993, pp. 231–236. DOI: 10.1109/VL.1993.269602.

[21] M. Robinson, J. Hamshar, J. Novillo, and A. Duchowski. "A Java-based tool for reasoning about models of computation through simulating finite automata and Turing machines." In: vol. 31. Mar. 1999, pp. 105–109. DOI: 10.1145/384266.299704.

[22] H Bergström. "Applications, Minimisation, and Visualisation of Finite State Machines." Doctoral dissertation. Stockholm University, 1998.

[23] *App: Home*. URL: http://www.automatatutor.com/ (visited on 03/23/2020).

[24] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. "Automated grading of DFA constructions." In: Aug. 2013, pp. 1976–1982.

[25] *RACSO*. URL: https://racso.cs.upc.edu/juezwsgi/index (visited on 03/23/2020).

[26] *JFLAP*. URL: http://www.jflap.org/ (visited on 03/23/2020).

[27] M. Wermelinger and A. M. Dias. "A Prolog Toolkit for Formal Languages and Automata." In: *SIGCSE Bull.* 37.3 (June 2005), 330–334. ISSN: 0097-8418. DOI: 10.1145/1151954.1067536. URL: https://doi.org/10.1145/1151954.1067536.

[28] *FAdo - FAdo | FAdo*. URL: http://fado.dcc.fc.up.pt/ (visited on 03/23/2020).

[29] S. Konstantinidis, C. Meijer, N. Moreira, and R. Reis. "Implementation of Code Properties via Transducers." In: vol. 9705. July 2016, pp. 189–201. ISBN: 978-3-319-40945-0. DOI: 10.1007/978-3-319-40946-7{_}16.

[30] M. Almeida, N. Moreira, and R. Reis. "Enumeration and Generation of Initially Connected Deterministic Finite Automata." In: (Jan. 2014).

[31] *FAdo - Other Resourses*. URL: http://fado.dcc.fc.up.pt/OResourses/ (visited on 03/23/2020).

[32] G. Heineman, G. Pollice, and S. Selkow. *Algorithms in a Nutshell*. O'Reilly Media, Inc., 2008. ISBN: 059651624X.

[33] D. C. Kozen. *The Design and Analysis of Algorithms*. Berlin, Heidelberg: Springer-Verlag, 1992. ISBN: 0387976876.

[34] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. 2nd. USA: Prentice Hall PTR, 1997. ISBN: 0132624788.

[35] C. Sernadas. *introdução à teoria da computação*. Lisboa, 1992.

Appendix

A.1 Top-level manual

---

fa_load *file*

Creates a new automaton based on the given file

Parameters:

file (String) – name of the file containing the information on the automaton

Returns:

(finiteAutomaton) – the new automaton

---

fa_accept *fa w*

Tests if the automaton fa accepts word w

Parameters:

fa (finiteAutomaton) – the loaded automaton;

w (String) – string of word to be tested

Returns:

(finiteAutomaton) – the new automaton

---

fa_traceAccept *fa w*

Traces the acceptance of the word w by the automaton fa

Parameters:

fa (finiteAutomaton) – the loaded automaton;

w (String) – string of word to be tested

Returns:

(unit) – the process of automaton *fa* accepting the word *w*

---

fa_generate *fa l*

Generates the set of word with a maximum length of l recognized by the automaton fa

Parameters:

fa (finiteAutomaton) – the loaded automaton;

l (int) – maximum length of the generated words

Returns:

(String list) - the set of generated words

---

fa_reachable *fa*

Produces the set of all reachable states of the automaton fa

Parameters:

fa (finiteAutomaton) – the loaded automaton

Returns:

(String list) - the set of reachable states

---

fa_productive *fa*

Produces the set of all productive states of the automaton fa

Parameters:

fa (finiteAutomaton) – the loaded automaton

Returns:

(String list) - the set of productive states

---

fa_clean *fa*

Eliminates all non-useful states and their transitions from fa

Parameters:

fa (finiteAutomaton) – the loaded automaton

Returns:

(finiteAutomaton) - the new automaton

---

fa_toDeter *fa*

Converts the automaton fa into its equivalent deterministic automaton

Parameters:

fa (finiteAutomaton) – the loaded automaton

Returns:

(finiteAutomaton) - the new automaton

---

fa_isDeter *fa*

Verifies if the automaton fa is deterministic

Parameters:

fa (finiteAutomaton) – the loaded automaton

Returns:

(bool) – if *fa* is deterministic

---

fa_minimize *fa*

Minimizes the automaton fa

Parameters:

fa (finiteAutomaton) – the loaded automaton

Returns:

(finiteAutomaton) – the minimized automaton

---

fa_toRegex *fa*

Converts the automaton fa to its equivalent regular expression

Parameters:

fa (finiteAutomaton) – the loaded automaton

Returns:

(String) – the minimized automaton

---

re_load *file*

Creates a new regular expression based on the given file

Parameters:

file (String) – name of the file containing the information on the regular expression

Returns:

(string) – the new regular expression

---

re_alphabet *re*

Produces the alphabet of regular expression re

Parameters:

re (String) – loaded regular expression

Returns:

(char list) – the alphabet of *re*

---

re_accept *re w*

Verifies if the regular expression re accepts the word w

Parameters:

re (String) – loaded regular expression;

w (String) – word to be tested

Returns:

(bool) – if *re* accepts *w*

---

re_trace *re w*

Shows the derivation tree for the acceptance of w by re

Parameters:

re (String) – loaded regular expression;

w (String) – word to be tested

Returns:

(unit) – the derivation tree

---

re_generate *re l*

Generates the set of word with a maximum length of l recognized by the regular expression re

Parameters:

re (string) – the loaded regular expression;

l (int) – maximum length of the generated words

Returns:

(String list) - the set of generated words

---

re_simplify *re*

Simplifies the regular expression re

Parameters:

re (string) – the loaded regular expression

Returns:

(String) – the simplified regular expression

---

re_toFA *re*

Converts the regular expression re to its equivalent automaton

Parameters:

re (string) – the loaded regular expression

Returns:

(finiteAutomaton) – the resulting finite automaton

---

cfg_load *file*

Creates a new context-free grammar based on the given file

Parameters:

file (String) – name of the file containing the information on the context-free grammar

Returns:

(contextFreeGrammar) – the new context-free grammar

---

cfg_accept *cfg w*

Verifies if the grammar cfg accepts the word w

Parameters:

cfg (contextFreeGrammar) – loaded grammar;

w (String) – word to be tested

Returns:

(bool) – if cfg accepts w

---

cfg_trace *cfg w*

Traces the acceptance of the word w by the grammar cfg

Parameters:

cfg (contextFreeGrammar) – loaded grammar;

w (String) – string of word to be tested

Returns:

(unit) – the process of grammar *cfg* accepting the word *w*

---

cfg_generate *cfg l*

Generates the set of word with a maximum length of l recognized by the grammar cfg

Parameters:

cfg (contextFreeGrammar) – the loaded grammar;

l (int) – maximum length of the generated words

Returns:

(String list) - the set of generated words

---

cfg_toFA *cfg*

Converts the grammar cfg to its equivalent finite automaton

Parameters:

cfg (contextFreeGrammar) – the loaded grammar

Returns:

(finiteAutomaton) – the resulting finite automaton

---

cfg_toRe *cfg*

Converts the grammar cfg to its equivalent regular expression

Parameters:

cfg (contextFreeGrammar) – the loaded grammar

Returns:

(string) – the resulting regular expression

---

exer_load *file*

Creates a new exercise based on the given file

Parameters:

file (String) – name of the file containing the information on the context-free grammar

Returns:

(exercise) – the new exercise

---

exer_testFA *exer fa*

Tests if the language partially defined in exer is part of the language of fa

Parameters:

exer (exercise) – the loaded exercise;

fa (finiteAutomaton) – the loaded automaton

Returns:

(boolean) – if exer was correct

---

exer_testFAFailures *exer fa*

Produces the words which caused exer to be incorrect

Parameters:

exer (exercise) – the loaded exercise;

fa (finiteAutomaton) – the loaded automaton

Returns:

((String list)*(String list)) – the pair containing the list of words which failed to be accepted and list of words which failed to be rejected

---

exer_testRe *exer re*

Tests if the language partially defined in exer is part of the language of re

Parameters:

exer (exercise) – the loaded exercise;

re (string) – the loaded regular expression

Returns:

(boolean) – if *exer* was correct

---

exer_testReFailures *exer re*

Produces the words which caused exer to be incorrect

Parameters:

exer (exercise) – the loaded exercise;

re (string) – the loaded regular expression

Returns:

((String list)*(String list)) – the pair containing the list of words which failed to be accepted and list of words which failed to be rejected

---

exer_testCfg *exer cfg*

Tests if the language partially defined in exer is part of the language of cfg

Parameters:

exer (exercise) – the loaded exercise;

cfg (contextFreeGrammar) – the loaded grammar

Returns:

(boolean) – if *exer* was correct

---

exer_testCfgFailures *exer cfg*

Produces the words which caused exer to be incorrect

Parameters:

exer (exercise) – the loaded exercise;

cfg (contextFreeGrammar) – the loaded grammar

Returns:

((String list)*(String list)) – the pair containing the list of words which failed to be accepted and list of words which failed to be rejected

---