Rita Pedroso Macedo

Bachelor in Computer Science and Engineering

OCaml-Flat on the Ocsigen framework

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

Master of Science in
Computer Science and Engineering

Adviser:	Artur Miguel Dias, Auxiliar Professor, NOVA School of Science and Technology
Co-adviser:	António Ravara, Associate Professor, NOVA School of Science and Technology

Examination Committee

Chair:	Doutor Jorge Cruz, FCT-NOVA
Rapporteur:	Doutora Nelma Moreira, FCUP
Member:	Doutor Artur Miguel Dias, FCT-NOVA

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

August, 2020

OCaml-Flat on the Ocsigen framework

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

First and foremost, I would like to express my gratitude to the Tezos Foundation which grant supported the development of my project.

My utmost appreciation goes to my Advisers, Professors Artur Miguel Dias and António Ravara, whose commitment and support throughout the development of this project have been relentless and inspirational.

I would also like to thank my Colleagues at Room P3/14 for helping me all along the way.

To all my Friends, who are always willing to share my burdens and joys, my deepest thanks.

Last but not least, my heartfelt thanks to my Family for always supporting my choices - even when I have decided to change from Arts to Computer Engineering - and for believing in my path, come rain or shine.

Abstract

Formal Languages and Automata Theory are important and fundamental topics in Computer Science. Due to their rigorous and formal characteristics, learning these becomes demanding. An important support for the assimilation of concepts is the possibility of interactively visualizing concrete examples of these computational models, thus facilitating their understanding. There are many tools available, but most are not complete or do not fully support the interactive aspect.

This project aims at the development of an interactive web tool in Portuguese to help understand, in an assisted and intuitive way, the concepts and algorithms in question, watching them work step-by-step, through typical examples pre-loaded or built by the user (an original aspect of our platform). The tool should therefore enable the creation and edition of an automaton or a regular expression, as well as execute the relevant classical algorithms such as word acceptance, model conversions, etc. Another important feature is that the tool has a clean design, in which everything is well organized and it is also extensible so that new features can be easily added later.

This tool uses the *Ocsigen Framework* because it provides the development of complete and interactive web tools written in *OCaml*, a functional language with a strong type checking system and therefore perfectly suitable for a web page without errors. *Ocsigen* was also chosen because it allows the creation of dynamic pages with a singular client-server system.

This document introduces the development of the tool, its design aspects that enable showing different conversions and processes as well as the development of the several functionalities related to the mechanisms already presented.

Keywords: Formal Languages, Automata, OCaml, Ocsigen, Teaching applications, Interactive Web Pages.

Resumo

Linguagens Formais e Teoria de Autómatos são bases importantes na formação em Engenharia Informática. O seu carácter rigoroso e formal torna exigente a sua aprendizagem. Um apoio importante à assimilação dos conceitos é a possibilidade de se visualizarem interactivamente exemplos concretos destes modelos computacionais, facilitando a compreensão dos mesmos. Há um grande número de ferramentas disponíveis, mas a maioria não está completa ou não oferece suporte total ao aspecto interativo.

Este projeto visa o desenvolvimento de uma ferramenta web interativa, em português, para ajudar de forma intuitiva e assistida a entender os conceitos e algoritmos em questão, observando-os passo a passo, através de exemplos típicos pré-carregados ou construídos pelo utilizador (um aspecto original desta plataforma). A ferramenta deve, portanto, permitir a criação e edição de autómatos e expressões regulares, bem como executar os algoritmos clássicos relevantes, como aceitação de palavras, conversões de modelos etc. Outro foco importante é o design limpo da ferramenta, bem organizada e extensível para que novos mecanismos possam ser facilmente adicionados.

Esta ferramenta usa o *Framework Ocsigen*, pois este proporciona o desenvolvimento de ferramentas web completas e interactivas, escritas em *OCaml*, uma linguagem funcional com um forte sistema de verificação de tipos e, por isso, perfeita para se obter uma página web sem erros. O *Ocsigen* foi, ainda, escolhido porque permite a criação de páginas dinâmicas com sistema de cliente-servidor único.

Este documento apresenta o desenvolvimento da ferramenta, os aspectos de design que permitem mostrar diferentes conversões e processos, assim como as diversas funcionalidades relacionadas com os mecanismos já apresentados.

Palavras-chave: Linguagens Formais, Autómatos, OCaml, Ocsigen, Aplicações para ensino, Páginas Web Interactivas.

Contents

List of Figures

List of Tables

Listings

Acronyms

AF Finite Automaton

FCT Faculty of Science and Technology

FDA Finite Deterministic Automaton

FLAT Formal Languages and Automata Theory

NFA Non-Deterministic finite automaton

OFLAT OCaml - Formal Languages and Automata Theory

RE Regular Expression

UNL Nova University of Lisbon

Chapter 1

Introduction

1.1 Context and Motivation

Given the mathematical and formal character of the topics covered in subjects such as languages and automata, its teaching and learning processes are demanding and challenging. Several studies have confirmed these difficulties and evaluated the use of applications to study this theme [14, 17, 36, 43]. It is important to support students' autonomous work with interactive tools that allow them to view examples and solve exercise. However, most applications are in English and, because they have different focuses, they do not always meet all needs. While some are quite complete, but are *Desktop*, and not always available in different kinds of devices, others are easy to access via *browser*, although they are underdeveloped.

In the Portuguese context, in terms of the programs and bibliographies of most Theory of Computation subjects (or equivalents) at the main universities, it is noticeable that most of the material is theoretical sometimes missing the use of practical tools. It is also understood although the material used in class may be Portuguese the bibliography is mostly in English. Therefore, there is room for an interactive tool, in Portuguese, which is capable of complementing the theoretical study that is already done in class today.

This thesis arises in the context of a project, called FACTOR, which aims to promote the use of OCaml in the Portuguese-speaking academic community, namely by supporting teaching approaches and tools. In particular, it aims to expand and consolidate the user bases of software and teaching materials in OCaml in Portuguese language for subjects in the area of Computational Logic and Computer Fundamentals in IT courses.

1.2 Objectives

The objective of this project is to develop an application, in Portuguese, that facilitates the study of Theory of Computation subjects for students of Computer Science, available through a browser. The idea is to create a tool, OFlat (OCaml - Formal Languages and Automata Theory), that represents and animates graphically Formal Languages and Automata Theory algorithms to allow students to visualize the mechanisms and its processes to more easily understand them. The FLAT algorithms were created in the context of another thesis, where it was developed a library called OCaml-FLAT. We intend for our tool, in the future, to support all topics within Theory of Computation, such as finite deterministic and non-deterministic automata, stack automata, regular languages, context-independent languages, LL languages [26] and all the features inherent to these topics, such as conversions, minimizations and tests. This means that the tool needs to be extensible, to allow the addition of the new functionalities, easily and effectively. It is also intended that this tool should be, firstly, adapted to the course of Theory of Computation [38] taught at FCT-UNL and ready to use in the next edition and also that it includes not only exercises evaluated automatically and with given feedback, but also allowing students to create their own exercises.

Due to the project in which this thesis is inserted and the fact that the OCaml-FLAT library is written in *OCaml* we decided, as a proof of concept, to try to create a web tool completely in *OCaml*. The goal is to develop the application using the *Ocsigen Framework*, that allows the creation of interactive web systems, entirely written in *OCaml*. By using the features of *OCaml*, it is possible to obtain fully functional and less error-prone web pages. *Ocsigen* also facilitates the creation of web tools, as it allows to write client and server in the same language, thus facilitating system programming. The use of *OCaml* also enables the algorithms to be the most similar to the FLAT mathematical definitions given in class, which follow the classical literature [30, 44].

1.3 Contributions

The key contributions of this dissertation are the following:

- The development of an extensible application, called OFLAT, achieved through an organization of the code that allows the easy addition of new mechanisms.

- A web tool developed almost exclusively in *OCaml* that is simple, interactive, and user friendly.

- Graphical representation and animation of several functionalities related to Finite Automata and Regular Expressions, even if not yet fully animated.

- Creation and representation of simple exercises where the user is asked to define a language by means of a finite automaton or a regular expression and that gives the user feedback through a set of unit tests.
- A web tool ready to be experimented and analyzed in class, which is also a way to understand what is most helpful to the students and what to improve.

The application can be accessed in *http://ctp.di.fct.unl.pt/FACTOR/OFLAT* and the code can be seen in *https://gitlab.com/releaselab/factor/oflat*.

1.4 Document Organization

This document is divided into 9 Chapters and it works at the same time as a User Manual, a presentation of the application and a code documentation. To do so, we start by doing two types of contextualization: the first in Chapter 2, addresses the emergence of FLAT tools and highlights a few that distinguish themselves for some of their characteristics; the second in Chapter 3 summarizes the elements needed for development of web tools and its usual languages, and exposes the framework and library used for the development of the application and specifies their usage. Once the context is finalized we make an in-depth explanation of the application, starting with a general description of the application and its code structure in Chapter 4, followed by an analysis of each one of its mechanisms in Chapters 5, 6 and 7. These follow the same structure: first they analyze the presentation of the application or functionality and its usage and then explain the related code. Furthermore in Chapter 8 we explain the testing that the application went through to make sure that it is not only usable but correctly implemented. Finally, in Chapter 9 we present our conclusions and the work that is planned (for the future).

Chapter 2

Related Work

Early on it was realized that learning difficulties, and even teaching, in the Study of Formal Languages and Theory of Automata was a recurring problem. And that, being a theme that revolves around processes and abstract machines, the best way to understand this problem would be through pedagogical tools. The development of these tools has been done since the beginning of the 1960's [11]. The article *Fifty Years of Automata Simulation: A Review* [11] also argues that although there are already many tools, the scientific community continues to receive new ones because each one is different, having its own principles and often new uses. In addition, each tool is influenced by the development tools currently available.

There are textual tools, such as the Prolog Toolkit described in [55], which allows the creation and testing of automata using text or code, without these being graphically visible, and tools based on graphical visualization [4, 6, 21, 23]. Some of these will be analyzed later, in greater depth, because they have similarities with the theme of this project.

In the research work carried out, many examples of applications were found that, in some way, aim to overcome the difficulties mentioned by making use of differentiated solutions. As an example, we can refer the following tools: *FSM Simulator* [48] a Java program that makes it possible simulate the acceptance of a word in a finite automata; *Language Emulator* [51] a tool that allows working with different concepts within Automata Theory and that has been used by students at the University of Minas Gerais in Brazil; *jFAST* [32] a graphic software that allows the study of finite state machines; *RegeXeX* [10] an interactive system for studying Regular Expressions; *Forlan* [47] tool embedded in the Standard ML language that allows the experimentation of formal languages and Automata.

It is also important to mention the tool described by *Coffin et al.* in [13] because it was probably the first to be developed in the area. Many others are referred in the article *Fifty Years of Automata Simulation* [11].

One can also speak of libraries developed for Formal Languages and Automata, such as Awali [7] (evolution of Vaucanson [12, 31] and Vaucanson 2 [19]), Grail [39], written in C++; and FAdo, written in Python. Awali and FAdo are also examples of libraries that are evolving into graphic applications.

It is also worth mentioning applications such as *TANGO* [46], *JAWAA* [35], *GUESS* [2] and *VisualAlgo* [52], for their purpose of animating algorithms or data structures, being related with this project because of their strong interactive component. Finally, the platform *Learn OCaml* [29] stands out for allowing teachers to create different types of exercises not only for classes but also for evaluation, giving students *feedback* about their resolutions.

In the remainder of this chapter some FLAT libraries and visualization tools are going to be analyzed in terms of ease of installation and access, forms of visualization (including if it allows the comparison of results), step-by-step execution of algorithms (and if it allows going forward and backwards), exercises and feedback and comprehensiveness.

2.1 FLAT Libraries

As this project aims at creating an interactive application for working with Regular Expressions and Automata Theory, it is important to highlight Libraries that in some way already try to visually represent them. Table 2.2 shows a comparison of the characteristics analyzed in each Library.

Table 2.1: Evaluation of the criteria for each of the highlighted Libraries

	Awali	**FAdo**
Easy access and installation	✗	✗
Visual representation	✓✗	✓✗
Comparison of conversions	✓	✓
step-by-step	✗	✗
Exercises and feedback	✗	✗
Comprehensiveness	✓✗	✓

2.1.1 Awali

Awali [7] is a Library written in C++ that is the evolution of two other platforms: Vaucanson and Vaucanson2, which allow representing and working with different kinds of Automata and Rational Expressions (Regular Expressions).

Awali is divided into three layers called static, dynamic and interface. The static layer is the core of the platform, it is where the data structures and the algorithms are implemented. The dynamic layer is the one that allows the call of functions of the static level. The interface layer allows a command-line interface and a *Python* interface.

The installation is not the easiest one and the user is expected to install a large set of components (if he does not already have them installed). After the installation to use the Python interface there is advised to use Jupyter notebook. The Python interface is very simple, after the user programs the automaton or regular expression it allows him to visualize it graphically or textually. The user can modify or transform the automaton or regular expression but, after the changes, the user must issue the command to display the result again.

Despite already allowing the graphical visualization of Automata and Rational Expressions (as trees), we must always display the mechanism at each transformation in order to be able to compare the results. It also does not allow the testing if a word is accepted by the automaton or the Regular Expression.

Since the documentation is still in development, there are probably some mechanisms that are yet to be documented and for that reason, a common user does not know they exist.

2.1.2 FAdo

FAdo [20] is a software library written in *Python* for the symbolic manipulation of Regular Expressions and Automata. Its main purpose is to be used as a research tool and not so much as a teaching mechanism even though it has most of the mechanisms and functionalities taught in subjects like Theory of Computation.

It allows two types of installation, one easier than the other and like Awali it expects that the user has installed a large set of components. After the installation it can be used through a Python 2 command line. This means that this library is mainly used as textual tool but allows to graphically representing automata as a static image.

This library is very complete and allows working with Regular Languages, Finite Languages, Transducers and Codes. It allows the creation of models of computation, like DFA, NFA, Regular Expressions, Context Free Grammars and many others (that are not mentioned in the context of this thesis) and grants most of the conversions and transformations, and also enables the testing of the acceptance of a word but only returns true or false (not showing the acceptance step-by-step).

Automata can be generated graphically. For this reason, if the user saves the graphical representation and then makes a conversion, when he opens the two images side by side he can compare the results.

FAdo has a very complete documentation which not only describes thoroughly its API but also teaches how to use it through some examples.

For a student to use these two platforms he must download them into a computer, install them and then he has to have a Python console to work on. For this reason they are not usable in all types of devices and are not as easy to access and experiment like a web application.

2.2 FLAT Visualization tools

As stated earlier in the introduction of this chapter, it is understood that there are many tools that could have been mentioned here. However, we decided to develop a little further on a few that stand out for different specificities: *JFlap*, because it is probably the most complete tool available; *Automaton Simulator* for being a tool that allows the study of FA, verifying the acceptance of sentences; *FSM Simulator, Regular Expression Gym and FSM2Regex* for being a web applications and allowing the study of FA and RE and their conversions; *Automata Tutor v2.0* because it has an evaluation and feedback system; and *Automate* because it is a tool which has objectives that are similar to the purpose of this project and it is being developed at the same time.

Table 2.2 shows a comparison of the characteristics analyzed in each platform.

Table 2.2: Evaluation of the criteria for each of the highlighted Visualization Tools

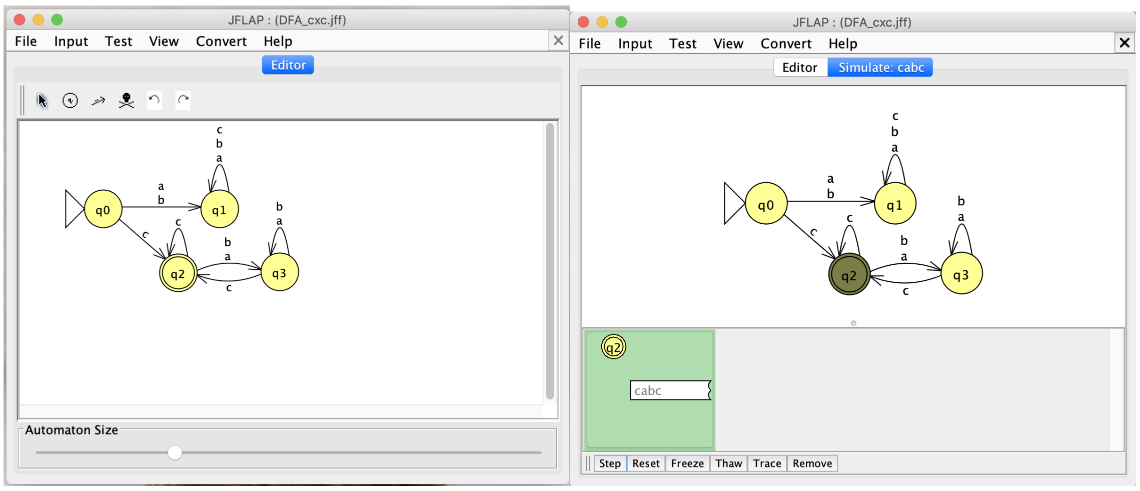
	JFlap	**Automaton Simulator**	**FSM**	**Automata Tutor**	**AutoMate**
Easy access and installation	✗	✓	✓	✓	✓
Visual representation	✓	✓✗	✓	✓	✗
Comparison of conversions	✓	✗	✓	✗	✗
step-by-step	✓✗	✓✗	✓	✗	✗
Exercises and feedback	✗	✗	✗	✓	✓
Comprehensiveness	✓	✗	✓✗	✗	✗

2.2.1 JFlap

It is a desktop tool that is being developed since 1990 [23]. It stands out for being one of the most complete tools for the study of Formal Languages and Automata Theory. It is the result of the work of Susan H. Rodger and a few of her students who, over time, have developed new functionalities [24, 25]. Although it was initially written in *C++* and *x windows*, it was later rewritten in *Java* and *swing* in order to improve the graphical interface. The source code is available on the web page [23] and *GitHub*, allowing any user to modify it.

This software is complemented by a web page that provides explanations and resolutions of exercises and by a guide book for the application usage, which is out of date.

JFLAP has been used for over 20 years in various universities all over the world, and its positive impact has already been tested [33, 41, 42].



(a) Automata Editor (b) Page to test words step-by-step

Figure 2.1: JFLAP examples [5]

In terms of features, it is possible to work interactively with Finite Automata (convert NFAs in DFA, NFAs in Regular Expressions or Grammars, minimize DFAs, testing if they accept words and visualize the acceptance process); Mealy Machines; Moore Machines; Push-down Automata (creation from context-free languages e vice-versa); three types of Turing Machines (One Tape, Multi-Tape and with Building Blocks); Grammars; L-system; Regular Expressions (creation of DFAs, AFAs, Regular Grammars and Regular Expressions); Regular Pumping Lemma; and Context-free Pumping Lemma.

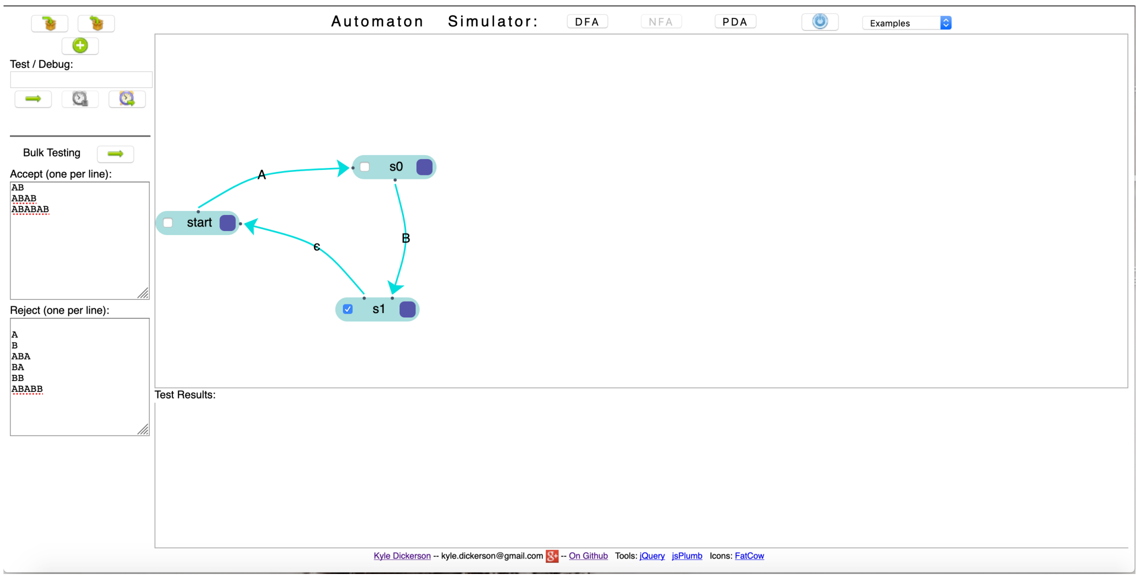
Despite all of its good points, *JFLAP* is a desktop application, which means that it is not accessible in all circumstances (it is necessary to have a computer and the software downloaded and installed on it to be able to use it). Note that these days students use mostly mobile equipments.

In JFLAP any conversion process, minimization or of acceptance can be visualized step-by-step but it does not allow to go a step back. The application is made for the student to be able to perform the processes on their own, but sometimes, even with instructions, it may be difficult to understand the rules that are being used for each step of the process.

Although the functionalities have evolved, the design has become a bit outdated and its usage is not always intuitive. Very often the support of the online page is needed to fully understand the correct usage of the program.

2.2.2 Automaton Simulator

The *Automaton Simulator* [6] is a very simple web tool, with one single web page (Figure 2.2). It was written in *JavaScript*, *jQuery* and *jsPlumb* by Kyle Dickerson, Software Developer and Technical Leader at the National Laboratory of Lawrence Livermore. The source code is available at GitHub under the MIT license.



Accept (one per line):
AB
ABAB
ABABAB

Reject (one per line):
A
B
ABA
BA
BB
ABABB

Kyle Dickerson -- kyle.dickerson@gmail.com -- On Github Tools: jQuery jsPlumb Icons: FatCow

Figure 2.2: Automaton Simulator Web Page [6] with examples of words to be accepted and words to be rejected

In this page one can draw graphically three different types of automata - Deterministic finite automata, Non Deterministic Finite Automata and Pushdown Automata. Nevertheless, it is neither possible to generate automata from a regular expression, nor to convert from NFA to DFA.

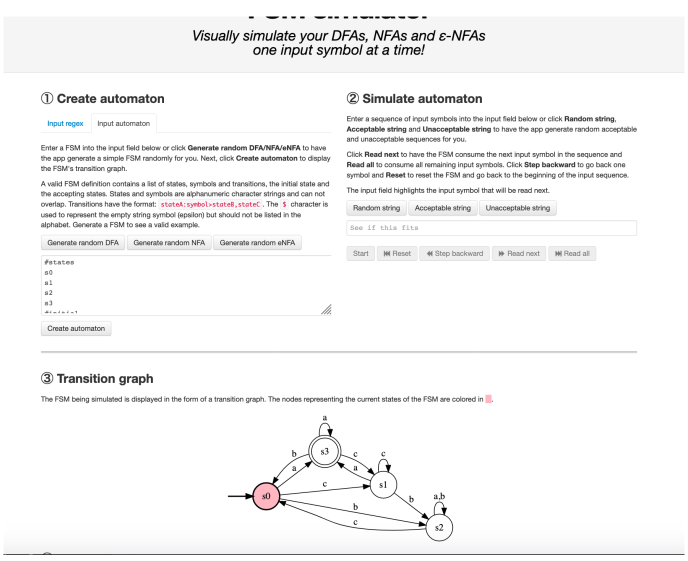
After the creation of the automaton it is possible to test the acceptance and rejection of words, as well as the step-by-step recognition of a word by the automaton. Yet, it only allows going forward and never backward.

Despite its simplicity, this page is not very intuitive, since it is drawn with the help of icons, which do not include any type of explanation. Besides that, it has only a few features.

2.2.3 FSM simulator, Regular Expressions Gym, FSM2Regex

The *FSM Simulator* [21], *Regular Expression Gym* [40] and *FSM2Regex* [22] are three complementary tools that enable the study of Regular Expressions and Automata Theory. Each of these tools is a web page developed since 2012, by Ivan Zuzak, Web Engineer and a former professor at the University of Zagreb, and Vedrana Jankovic, Software Engineer at Google. They are developed in *Noam* - a *JavaScript* library that let you work with finite state machines and regular grammars and expressions, *Bootstrap*, *Viz.js* e *jQuery*. The source code is available in GitHub, under the Apache v2.0 license.

The *FSM Simulator* (Figure 2.3) is used for the creation and testing of automata. The Automata can be generated through regular expressions or text. However, it is not possible to create them graphically. The user may visualize the processes of recognition of a word by the automaton, step-by-step, and also move forward or backward whenever necessary.



Visually simulate your DFAs, NFAs and ε-NFAs one input symbol at a time!

① Create automaton

Enter a FSM into the input field below or click **Generate random DFA/NFA/eNFA** to have the app generate a simple FSM randomly for you. Next, click **Create automaton** to display the FSM's transition graph.

A valid FSM definition contains a list of states, symbols and transitions, the initial state and the accepting states. States and symbols are alphanumeric character strings and can not overlap. Transitions have the format: `stateA:symbol>stateB,stateC`. The `$` character is used to represent the empty string symbol (epsilon) but should not be listed in the alphabet. Generate a FSM to see a valid example.

```
#states
s0
s1
s2
s3
```

② Simulate automaton

Enter a sequence of input symbols into the input field below or click **Random string**, **Acceptable string** and **Unacceptable string** to have the app generate random acceptable and unacceptable sequences for you.

Click **Read next** to have the FSM consume the next input symbol in the sequence and **Read all** to consume all remaining input symbols. Click **Step backward** to go back one symbol and **Reset** to reset the FSM and go back to the beginning of the input sequence.

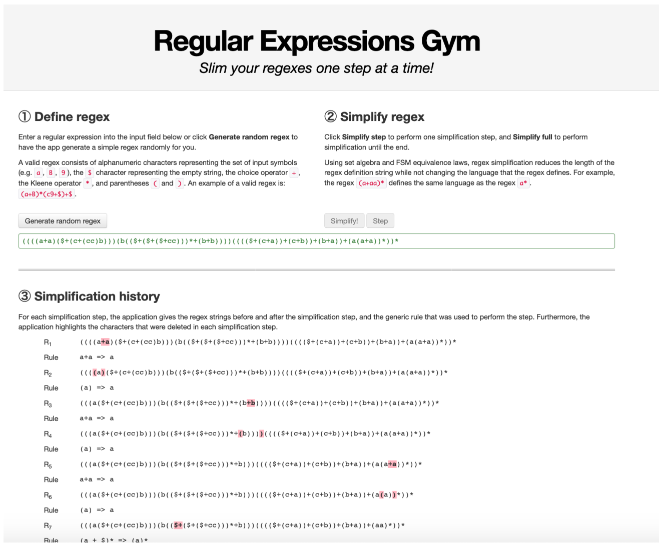
The input field highlights the input symbol that will be read next.

③ Transition graph

The FSM being simulated is displayed in the form of a transition graph. The nodes representing the current states of the FSM are colored in 🟥.

Figure 2.3: FSM Simulator Page [21] with an example of an Automaton

Despite this, the states are painted always in the same color, not indicating whether the word is accepted or not. The user looking at the last state has to make his own conclusions.



Regular Expressions Gym

Slim your regexes one step at a time!

① Define regex

Enter a regular expression into the input field below or click **Generate random regex** to have the app generate a simple regex randomly for you.

A valid regex consists of alphanumeric characters representing the set of input symbols (e.g. `a`, `B`, `9`), the `$` character representing the empty string, the choice operator `+`, the Kleene operator `*`, and parentheses `(` and `)`. An example of a valid regex is: `(a+B)*(c9+$)+$`.

② Simplify regex

Click **Simplify step** to perform one simplification step, and **Simplify full** to perform simplification until the end.

Using set algebra and FSM equivalence laws, regex simplification reduces the length of the regex definition string while not changing the language that the regex defines. For example, the regex `(a+aa)*` defines the same language as the regex `a*`.

```
((((a+a)($+(c+(cc)b)))(b(($+($+($+cc)))*+(b+b))))(((($+(c+a))+(c+b))+(b+a))+(a(a+a))*))*
```

③ Simplification history

For each simplification step, the application gives the regex strings before and after the simplification step, and the generic rule that was used to perform the step. Furthermore, the application highlights the characters that were deleted in each simplification step.

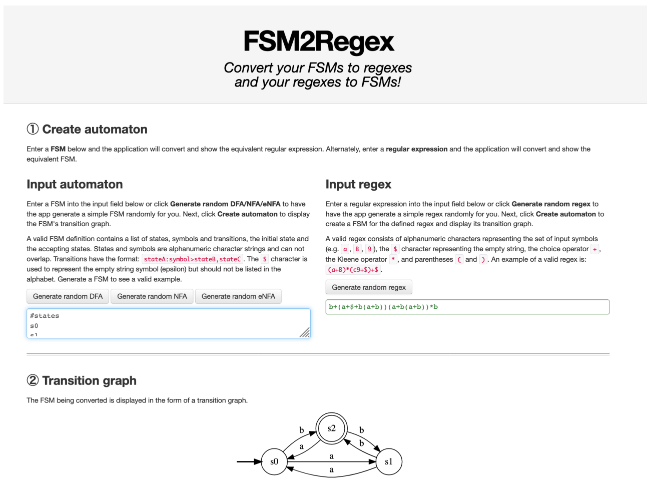
```
R1     ((((a+a)($+(c+(cc)b)))(b(($+($+($+cc)))*+(b+b))))(((($+(c+a))+(c+b))+(b+a))+(a(a+a))*))*
Rule   a+a => a
R2     ((((a)($+(c+(cc)b)))(b(($+($+($+cc)))*+(b+b))))(((($+(c+a))+(c+b))+(b+a))+(a(a+a))*))*
Rule   (a) => a
R3     (((a($+(c+(cc)b)))(b(($+($+($+cc)))*+(b+b))))(((($+(c+a))+(c+b))+(b+a))+(a(a+a))*))*
Rule   a+a => a
R4     (((a($+(c+(cc)b)))(b(($+($+($+cc)))*+(b))))(((($+(c+a))+(c+b))+(b+a))+(a(a+a))*))*
Rule   (a) => a
R5     (((a($+(c+(cc)b)))(b(($+($+($+cc)))*+b)))(((($+(c+a))+(c+b))+(b+a))+(a(a+a))*))*
Rule   a+a => a
R6     (((a($+(c+(cc)b)))(b(($+($+($+cc)))*+b)))(((($+(c+a))+(c+b))+(b+a))+(a(a))*))*
Rule   (a) => a
R7     (((a($+(c+(cc)b)))(b(($+($+($+cc)))*+b)))(((($+(c+a))+(c+b))+(b+a))+(aa)*))*
Rule   (a + $)* => (a)*
```

Figure 2.4: Regular Expression Gym Page [40] with an example of a Regular Expression simplification

The *Regular Expressions Gym* (Figure 2.4) is a very simple web page that lets the user visualize the simplification of a regular expression, all at once or step-by-step. For each step of the simplification, in both options, the rule used to simplify is indicated.

The *FSM2Regex* (Figure 2.5) is used to convert a regular expression in an automaton and vice versa, but it only gives the final solution, without allowing the visualization of the steps required to reach it.

These three pages are very simple and intuitive tools, but it would be more useful if they could be integrated in the same page. This way, it would be easier to add new



FSM2Regex

Convert your FSMs to regexes and your regexes to FSMs!

① Create automaton

Enter a **FSM** below and the application will convert and show the equivalent regular expression. Alternately, enter a **regular expression** and the application will convert and show the equivalent FSM.

Input automaton

Enter a FSM into the input field below or click **Generate random DFA/NFA/eNFA** to have the app generate a simple FSM randomly for you. Next, click **Create automaton** to display the FSM's transition graph.

A valid FSM definition contains a list of states, symbols and transitions, the initial state and the accepting states. States and symbols are alphanumeric character strings and can not overlap. Transitions have the format: `stateA:symbol>stateB,stateC`. The `$` character is used to represent the empty string symbol (epsilon) but should not be listed in the alphabet. Generate a FSM to see a valid example.

```
#states
s0
s1
```

Input regex

Enter a regular expression into the input field below or click **Generate random regex** to have the app generate a simple regex randomly for you. Next, click **Create automaton** to create a FSM for the defined regex and display its transition graph.

A valid regex consists of alphanumeric characters representing the set of input symbols (e.g. `a`, `B`, `9`), the `$` character representing the empty string, the choice operator `+`, the Kleene operator `*`, and parentheses `(` and `)`. An example of a valid regex is: `(a+B)*(c9+$)+$`.

```
b+(a+$+b(a+b))(a+b(a+b))*b
```

② Transition graph

The FSM being converted is displayed in the form of a transition graph.

Figure 2.5: FSM2Regex Page [22] with an Automaton and its corresponding Regular Expression

functionalities.

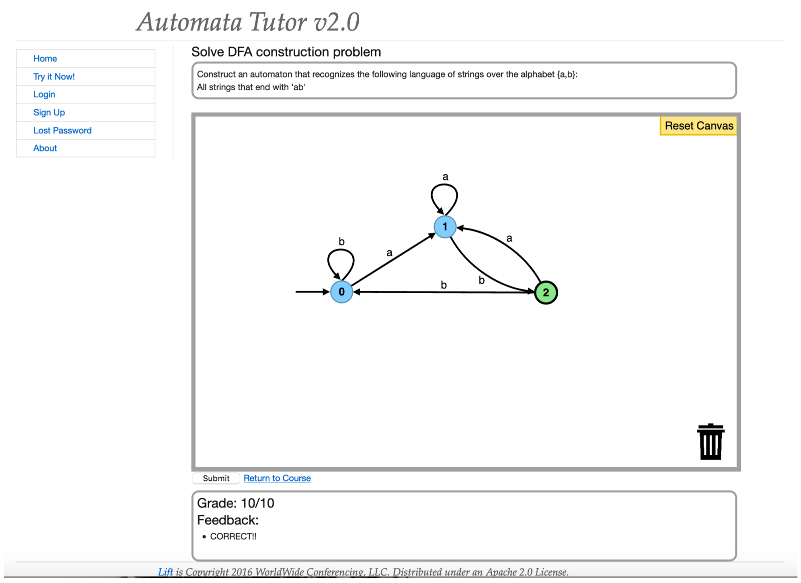
Unlike *Automaton Simulator* referred to in 2.2.2 these three pages use too much text to explain the functionalities, which is unnecessary since they are quite intuitive.

2.2.4 Automata Tutor v2.0

The *Automata Tutor* [4, 18] is a web tool that stands out because it provides a system for evaluating exercise of Finite Automata and Regular Expressions, thus facilitating the task for teachers. This tool has gone through three phases of development and several user tests [17, 18], aimed at improving the application.

The page allows the registration and login into the system as two different kinds of users: the teacher, to whom is given the possibility of creating a course with his own exercises and visualize the grades at the end of the course; and the student, who can sign in to a specific course or do the exercises available on the page outside the courses. The focus of this page is the resolution of exercises of creation of Regular Expressions and Finite Automata corresponding to a sentence given in English. When a student submits a solution, he receives as an answer the grade and some feedback, enabling him to understand what he did wrong and make corrections if necessary (an example can be seen in Figure 2.6).

Despite being a very complete application regarding the evaluation of exercises, its purpose is to give feedback, not allowing the user to experiment solutions freely in order to understand why it is right or wrong (for example the verification step-by-step of the acceptance of a word).



Solve DFA construction problem

Construct an automaton that recognizes the following language of strings over the alphabet {a,b}:
All strings that end with 'ab'

Grade: 10/10
Feedback:
- CORRECT!!

Lift is Copyright 2016 WorldWide Conferencing, LLC. Distributed under an Apache 2.0 License.

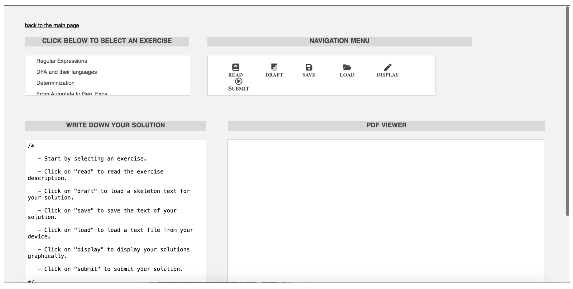
Figure 2.6: Example of an exercise resolution in Automata Tutor [4]

2.2.5 AutoMate

AutoMate [5] is a very recent web page, launched in 2019, which is still at a very early state. It is, however relevant to talk about it, since it is being developed at the same time as the one described in this paper. The main goal of this tool is to support the study of computer science students by helping to solve exercises on Finite Automata and Regular Expressions.

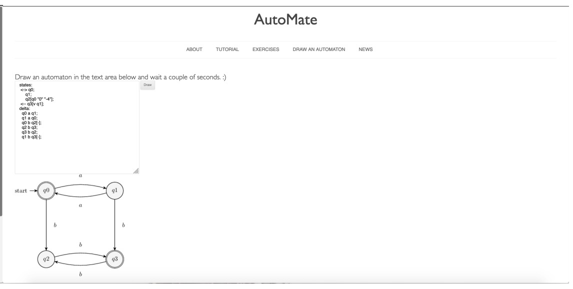
In terms of functionalities, this tool is, for now, very simple, mostly focusing on the verification and correction of exercises (Figure 2.7a). The resolution of the exercises is done through text, not being possible to see the automata graphically.

The tool allows the user to carry out exercises on different topics within the Theory of Computation - Regular Expressions, creating DFAs, transforming NFAs into DFAs and transforming DFAs into regular expressions. After submitting the resolution of the exercises the user receives an automatic feedback, in pdf, which is relevant for the student to know how to improve. However the tool has a limited number of exercises and does not allow the user to create his own.



```
/*

   - Start by selecting an exercise.

   - Click on "read" to read the exercise
description.

   - Click on "draft" to load a skeleton text for
your solution.

   - Click on "save" to save the text of your
solution.

   - Click on "load" to load a text file from your
device.

   - Click on "display" to display your solutions
graphically.

   - Click on "submit" to submit your solution.

*/
```

(a) Page to solve and see exercises



Draw an automaton in the text area below and wait a couple of seconds. :)

```
states:
 <-> q0;
     q1;
     q2[q0 "0" "-4"];
 <-- q3[v q1];
delta:
  q0 a q1;
  q1 a q0;
  q0 b q2[-];
  q2 b q3;
  q3 b q2;
  q1 b q3[-];
```

(b) Page to draw the Automaton

Figure 2.7: Example of two pages in Automate [5]

This tool also contains a page where an automaton can be drawn (see Figure 2.7b), through text, so that it may be viewed it through an image. However, it is not possible to perform any type of action on the created automaton.

All of the mentioned tools have their own characteristics. A number of these are in some way common to the project developed but, excluding the *AutomataTutor* none of them allows the lecturers to create exercises in order to use the tool both in class and as a student assessment system. Our tool stands out because it is prepared to in the future integrate the *Learn OCaml* system[1] and for this reason, it will allow teachers to create classroom and assessment exercises that are integrated into the discipline they teach in Automata Theory.

[1] https://try.ocamlpro.com/learn-ocaml-demo/

Chapter 3

Development Tools

The tool developed in the context of this thesis is a web application developed mostly in *OCaml*. In it the user can work with FLAT mechanisms in order to improve understanding of how they work. The next chapter will explain the design and code structure of the application. To provide some background, the current chapter offers a brief summary of general web technologies and presents the tools selected for the development of the application.

Typically, a web tool assumes two or three layers, also called tiers: client, server and database. These are normally developed in different languages, and therefore, to allow the sharing of data it is necessary to write them in a predefined format.

The client side is the part of the web application in which the user interacts. In each interaction a request is sent to the server, that responds with an action execution or some information from the database. This side is mainly developed using three languages: HTML, CSS and JavaScript.

The server side is the part of the application that determines how it works. The server receives requests from the client and returns the requested information or the action that can be executed. The server may be written in several different languages like C, C++, C#, PHP, Python, Java or even JavaScript.

For the client to communicate with the server, a communication protocol, like HTTP, is used. The message must also be in an agreed format, the most common being HTML, XML or JSON. And for the server to be able to communicate with the database it is also necessary to know how to write queries, normally in SQL or XQuery. Actually, it is not easy to link all three and make sure that the communication is always correct.

With the need to create more dynamic web pages, and since it is necessary to learn many languages and components, different frameworks and libraries have started to appear in order to facilitate the work of the programmer. Some aim to ease the development of the page design, like *Bootstrap*, with the purpose of creating responsive web pages. Others intend to facilitate the creation of single page web applications, i.e., reactive. We have, as examples, *AngularJS*, a framework that extends the *HTML* language; and *React*, a JavaScript library. Also worth mentioning is *jQuery* a library *JavaScript* that aims to simplify the writing of queries of the language, and frameworks that intend to facilitate the development of servers with many accesses to the database and that assume code reuse, like *Django* and *Ruby on Rails*.

Despite the emerging of so many frameworks and libraries, the programmer, to create web applications, always needs to know at least *HTML*, *CSS*, *JavaScript* (or a correspondent) and a language for the server.

With that in mind, other languages and Frameworks started to appear, called tierless, which means that they aim to allow the creation of websites without having to program three separate layers. Examples of those are Links and Ocsigen Framework.

Links [50] is a functional programming language for web applications that, when compiled, generates code for all three tiers from a single source file. It translates into JavaScript the part to run in the browser, into bytecode the server parts and into SQL the parts to run in the database. [15].

Ocsigen Framework [34] allows writing web applications completely in OCaml, which like Links, is a functional programming language. The aim in Ocsigen is for the programmer to be able to write Client and Server code as a single layer. When compiled the client code is translated to JavaScript. It also allows the creation of databases using MaCaQue (or macaque) that is a DSL for SQL Queries in Caml.

Both Links and Ocsigen seemed like viable options for this project but Ocsigen was chosen for two main reasons: (1) The library that this project is built on top of is already written in *OCaml*; (2) Links is a programming language used for academic purposes, with a small number of users and active projects, thus more likely to suffer transformations between versions. Ocsigen is already in use, and there are relevant projects online created with it.

3.1 Ocsigen Framework

The *Ocsigen Framework* is a very complete tool that stands out mainly for allowing the creation of interactive web pages written entirely in *OCaml*. This framework arises from the idea that functional programming is an elegant solution to some interaction problems on web pages [9]. It tries to respond to the new challenges of web pages, that is, the need for them to behave increasingly like applications [8]. One of its great advantages is to compile the client's *OCaml* code for *JavaScript*, which makes it possible to work together with this language and thus use a wide number of libraries, which would otherwise not be available.

3.1.1 Component Description

The *Ocsigen Framework* is actually a set of different components, which reflect the complexity of this tool.

Eliom is the main component of *Ocsigen*; is an extension of *OCaml* for programming without layers (*Tierless*) [37]. *Eliom* aims to be a new style of programming that fits the needs of modern web applications better than the usual programming languages (designed many years ago, for much more static pages [34]). Its main objective is to allow the development of a distributed application entirely in *OCaml* and as a single program [34] where there is no separation between client and server. To make this possible, there is a special syntax to distinguish the two. The client can easily access server variables, as the support system implements this mechanism transparently. Another important advantage results from the use of static typing of *OCaml*, making it possible to check errors and bugs at the time of compilation, making sure that correct web pages are obtained, with no problems with links or client-server communication. In addition, *Eliom* automatically solves frequent security problems on web pages. The *Eliom* is also based on the assumption that complex behavior is written in a few lines of code [34]. The applications developed in this tool run on any *browser* or mobile device, with no need for customization.

Js_of_ocaml is the compiler of *OCaml bytecode* to *JavaScript*. It is the component of *Ocsigen* that allows running the application written in *OCaml* in *JavaScript* environments like browsers [34] [53]. It also enables the integration of *JavaScript* code in the *OCaml* program.

Lwt is the cooperative *threads* library for *OCaml* that allows to deal with data concurrency and *deadlocks* problems. It is the standard way of building competing applications. It allows to place orders asynchronously, through promises. These are simply references that will be filled out asynchronously when the answer arrives.

Tyxml is the library that allows the construction of statically correct HTML documents. Tyxml provides a set of combiners, which use the *OCaml* type system to confirm the validity of the generated document.

Ocsigen-start is the library and *template* of an *Eliom* application with many typical components of web pages, which aims to facilitate the construction of interactive web applications.

The **Ocsigen-toolkit** is the library of *widgets* that, like *Ocsigen-start*, aims to facilitate the rapid development of interactive web applications.

3.1.2 Usage

In order to make the proof of concept - that it is possible to create a web application almost only in OCaml -, *OCaml* and *Ocsigen* are the languages used to program the core of the application. This is where the page is created, every modification is decided and every algorithm calculated.

Since the main objective of this project was to animate the OCaml-FLAT library, most of the algorithms are inherited and are written in OCaml. To be able to animate them sometimes they suffered a few modifications or were remade to fit a purpose.

Every component of the Ocsigen Framework was used in some way. Eliom, since it's the core of the framework was used to develop the top layer of the application. This is basically the use of OCaml language with extra components, like the creation of services to generate the page or create internal or external links. An example of the creation of a service can be seen in Listing 3.1. This service allows the creation of a link to access the Cytoscape.js Library. A service is created through three components: the prefix (line 4), the main link; the path (line 5), that specifies the path to the page; and the *meth* (line 6), that specifies the HTTP method and the HTTP parameters of the service. In the example we obtain *https://unpkg.com/cytoscape/dist/cytoscape.min.js*.

```
1  let script_uri1 =
2    Eliom_content.Html.D.make_uri
3    (Eliom_service.extern
4       ~prefix:"https://unpkg.com"
5       ~path:["cytoscape";"dist"]
6       ~meth:
7         (Eliom_service.Get
8            Eliom_parameter.(suffix (all_suffix "suff")))
9       ())
10     ["cytoscape.min.js"]
```

Listing 3.1: Creation of a service to access the *Cytoscape.js* Library

Lwt is used for the animation of the automata, this is going to be developed further in Chapter 5 but in short it is used to allow the page time to make animated changes to the automaton.

Tyxml was used to write the page correctly, not only in the initial generation but also every time a change occurs in it. An example is the creation of an input box that can be seen in Listing 3.2, this defines the *HTML* element input with *id* 'box' and the input type as text.

```
let inputBox = input ~a:[a_id "box"; a_input_type `Text]()
```

Listing 3.2: Input box creation

Js_of_ocaml is not only used to compile the program but also to make the dynamic changes in the page (Listing 3.3) and to communicate with the *Cytoscape.js* Library (as explained in the Section 3.2). In the example of the Listing 3.3 we use the *Dom_html* Module of the *Js_of_ocaml* Library to access *HTML* elements and change its characteristics.

As normally happens when using a new and different framework, the usage of *Ocsigen* was not always easy. Throughout the development of the application some difficulties

```
let oneBox () =
      let box1 = Dom_html.getElementById "Box1" in
      box1##.style##.width:= Js_of_ocaml.Js.string "100%";
      let box2 = Dom_html.getElementById "Box2" in
      box2##.style##.width:= Js_of_ocaml.Js.string "0%";
      let buttonBox1 = Dom_html.getElementById "buttonBox1" in
      buttonBox1##.innerHTML := Js_of_ocaml.Js.string "";
```

Listing 3.3: Changing *HTML* elements

have arisen that made the development challenging (a few of the problems are described in Appendix A).

3.2 Cytoscape.js Library

Cytoscape.js [16] is a graph network library written in JavaScript. It was designed to make it easier for programmers and scientists to use Graph Theory in their applications, whether to do analysis on the server or to create complete interfaces.

Cytoscape.js is the evolution of another project called Cytoscape web and is an open-source project created at the Donnelly Center at the University of Toronto in 2011, which is still under development to this day, through the contribution of more than 49 different collaborators. It is part of the Cytoscape Consortium, a non-profit organization that promotes the use of this software (and others) in the areas of Bioinformatics. This project is funded by the United States National Institute of Health.

It is a very complete library that lets you easily display and manipulate interactive graphs and that may be used both in desktop browsers and in browsers of mobile systems, which in addition contains many functions for graph analysis.

Cytoscape.js is an intuitive, easy to use and very complete library, which also contains a very developed API, well explained and with examples. Its website also includes several explanations about the library and a vast set of demos that the user can use as a basis for his projects.

3.2.1 Usage

Since the Ocsigen Framework allows the joint work of OCaml with JavaScript, for the graphical part of drawing automata and the syntax trees, we decided to use Cytoscape.js graph library. The main reason was that while having a complete and easy to use library do draw the graphs we could focus on other main issues of the project, like how to animate, how to show and how to organize all the information and not just about the representation of the graphs.

This library is used in a very simple way. The JavaScript file that creates and changes the graphs has no power of decision. All the decisions are made in the Ocsigen code and the JavaScript functions are only called in with all the information, to change or create the representation on the screen.

Listing 3.4 represents the creation (without states and transitions) of an automaton in the *Cytoscape.js* library. It defines the division in the page in which the automaton is going to be drawn (Line 4) and then defines the layout (Lines 5 to 9) and the style (Lines 10 to 47) of the automata elements, the states (nodes), the transitions (edges) and its information (names and symbols). If a state is final it will be defined as being part of the class *SUCCESS*, which means that it will have double border (defined in Lines 41 to 45). In Line 51 we can see an example of the creation of an element (in this case a node). Through this code we can also see that it is possible to define new characteristics of a node, like its position or if it can be moved by the user, using its *id* (Lines 55 to 57).

```
 1  function start () {
 2    number = 0;
 3    cy = window.cy = cytoscape({
 4      container: document.getElementById('cy'),
 5      layout: {
 6        name: 'grid',
 7        rows: 2,
 8        cols: 2
 9      },
10      style: [
11        {
12          selector: 'node[name]',
13          style: {
14            'content': 'data(name)',
15            'width': '40px',
16            'height': '40px',
17            'text-valign': 'bottom',
18            'text-halign': 'center'
19          }
20        },
21        {
22          selector: 'edge[symbol]',
23          style: {
24            'content': 'data(symbol)'
25          }
26        },
27        {
28          selector: 'edge',
29          style: {
30            'curve-style': 'bezier',
31            'target-arrow-shape': 'triangle'
32          }
33        },
34      {
35        selector: '#transparent',
36        style: {
        'visibility': 'hidden'
      }
    },
    {
      selector: '.SUCCESS',
      style: {
        'border-width': '7px',
        'border-color': 'black',
        'border-style': 'double'
      }
  },
    ],
    elements: {
      nodes: [
        {data: { id: 'transparent', name: 'transparent' }}
      ]
    }
  });
  cy.$('#transparent').position('y', 200);
  cy.$('#transparent').position('x', -200);
  cy.$('#transparent').lock();
}
```

Listing 3.4: Creation of a graph in *Cytoscape.js*

In this chapter we made a resume of the different tools that could have been used to create the OFlat application. We clarified the framework used to make the proof of concept and how it was going to be used and explained the library used to draw the graphs in the application and why its usage was not going to disrupt the proof of concept.

Now that the reader is aware of the tools used to develop the application we can move forward and demonstrate how it works.

Chapter 4
Page Architecture

This chapter intends to give a general explanation of the structure of the OFlat web application and the respective code developed in the context of this thesis, as well as to explain details of the code that the different functionalities make use of.

Section 4.1 starts by explaining the structure of the page and its design. It gives a general idea on how it works and defines why we chose the presented organization of the elements, we also explain which different types were tested for the menu and why we kept the one in use. In the end we explain the color scheme and why some color were chosen for specific functionalities.

After the explanation of the general page, in Section 4.2, we define the Architectural Pattern used as a basis to organize the code, we describe how it was applied and how it had to be adapted in order to work with the usage of *OCaml*.

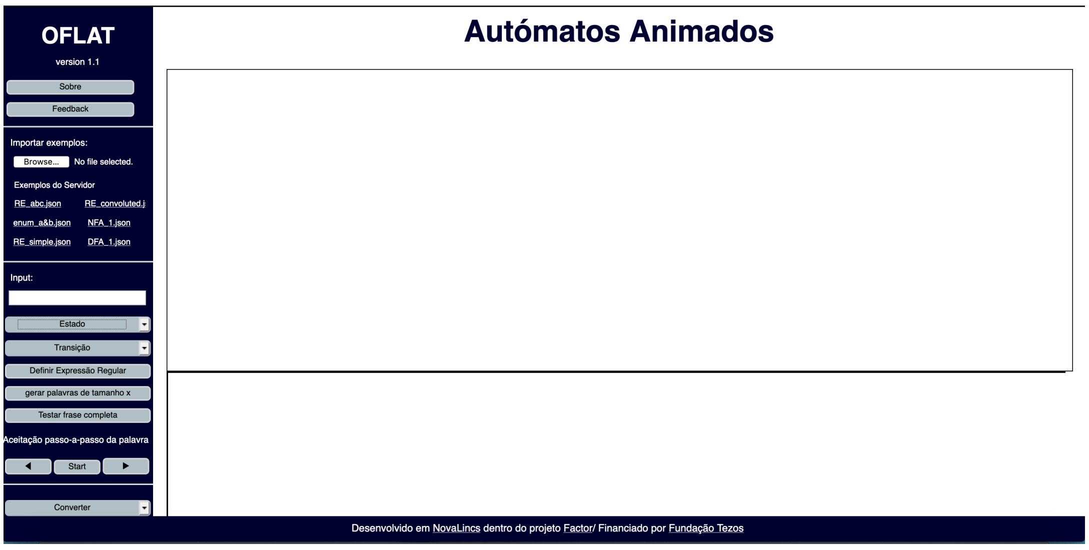
Furthermore, and now that the reader has a broad sense of how the application works and is organized, in Section 4.3 we go deeper into the code and explain how the page is generated in *Ocsigen* so that the reader has better sense on how this type of programming works and how the file importing was developed, a functionality that all the mechanisms make use of.

In the end to better demonstrate how the system works an example is given.

4.1 Page Design

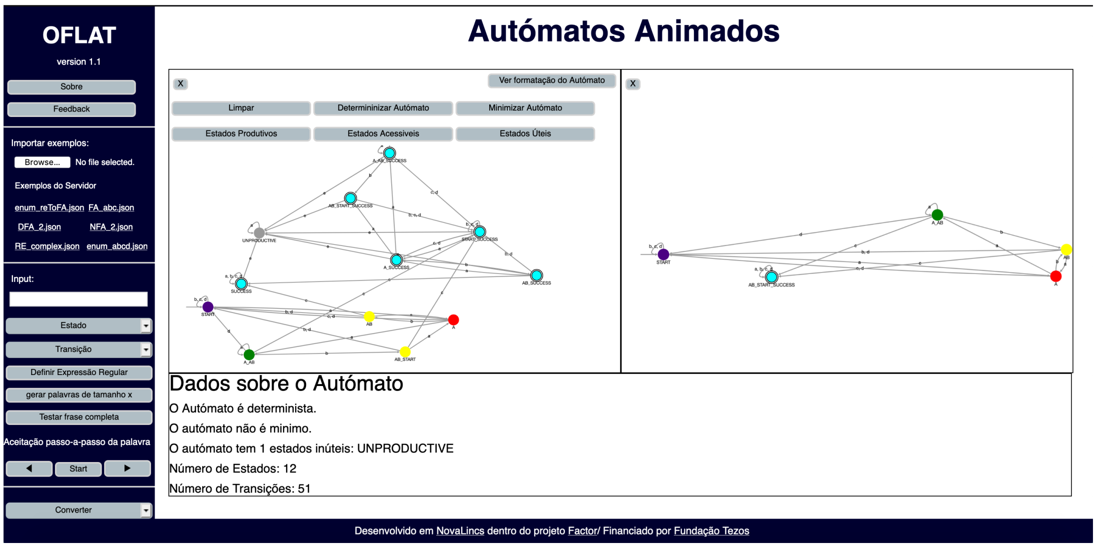
As far as the user interface is concerned, the OFlat application is a very simple web page with a menu on the left side and a white box on the rest of the page (as it can be seen in Figure 4.1).

Since users tend to focus their attention on the center of the page, we decided that this part of the screen should be reserved for the representation of the mechanisms like



Desenvolvido em NovaLincs dentro do projeto Factor/ Financiado por Fundação Tezos

Figure 4.1: Initial Page



Dados sobre o Autómato

O Autómato é determinista.

O autómato não é minimo.

O autómato tem 1 estados inúteis: UNPRODUCTIVE

Número de Estados: 12

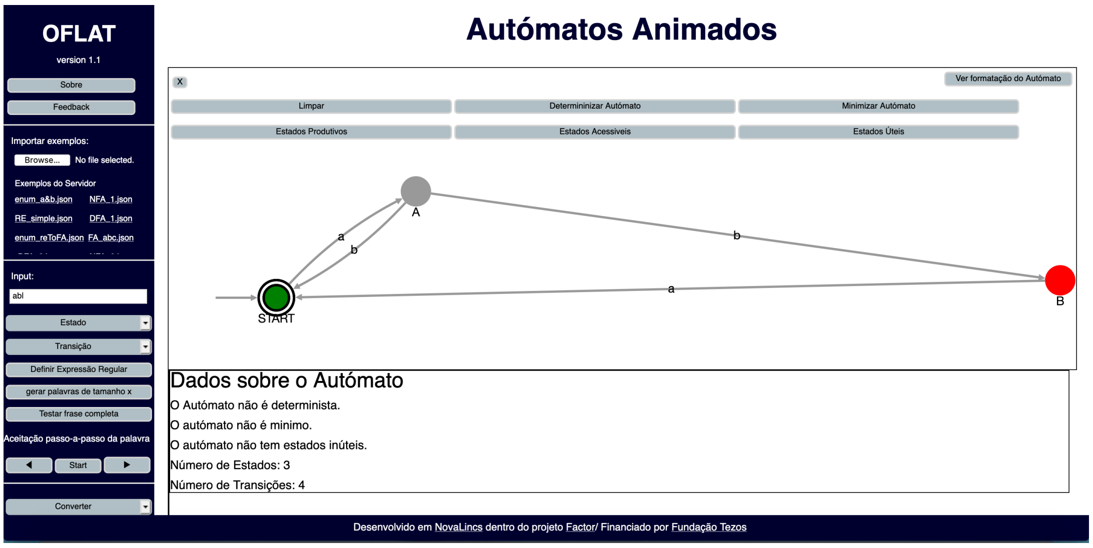
Número de Transições: 51

Desenvolvido em NovaLincs dentro do projeto Factor/ Financiado por Fundação Tezos

Figure 4.2: Minimization - Functionality that assumes two boxes

Automata or Regular Expressions. In this central page we attempt a reuse of the space in some functionalities. In general when a mechanism is created it is formatted to use the whole central box, but this can be divided into two smaller ones when a functionality demands it. The mechanism is set to readjust to the size of the box. For example, if the user is working with an Automaton and minimizes it, the minimization appears on the right side as shown in Figure 4.2. The main reason for this is that it is important for the user to be able to compare the two automata in order to understand the mechanism. This is a process that will happen with all the mechanisms that imply a transformation or to show extra information. After making the transformation and analyze it the user can choose to work with either mechanism, left or right, by clicking the "x" on the top left of the box with the mechanism to close.

Under the central box we choose to add a new one that will show extra or information about the automaton (Figure 4.3) or, if the mechanisms where imported with errors. The placement was chosen because users normally scan screens [28, 54], they do not read everything that is shown; this way users can scan the page and read the extra information if they find it necessary but will not lose focus on the important part of the screen, i.e. the representation of the mechanisms.



Dados sobre o Autómato

O Autómato não é determinista.

O autómato não é minimo.

O autómato não tem estados inúteis.

Número de Estados: 3

Número de Transições: 4

Figure 4.3: Acceptance of a work in an Automaton where there are two results, a right and a wrong one

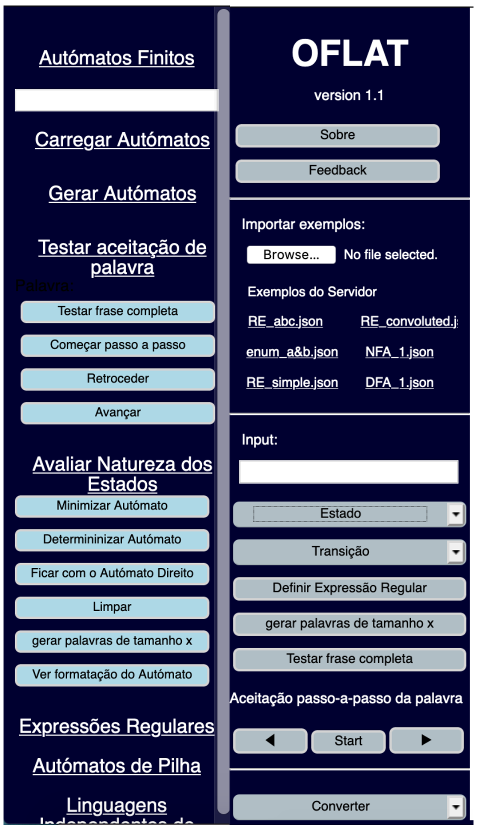
For the menu different options were experimented: A dropdown menu on the top, a dropdown menu on the left (Figure 4.4a) and a button menu as it is right now (Figure 4.4b).

The dropdown menu on the top did not work out because, as the user was clicking, there were so many options, that those would open on the top of the figure represented on the central box.

The dropdown menu on the left looked good but, actually, when the user was working it would become tiresome to always have to think and look for the button we would want to use. For example, if the user had an automaton and wanted to test the acceptance of a word he would on click *"Autómatos Finitos -> Testar aceitação da palavra"* and choose the button (Figure 4.4a).

In the end, the menu with just the buttons (Figure 4.4b) worked better. With this menu the user can easily scan for the buttons, and in terms of code the buttons can be used for different mechanisms that have the same functionalities.

The menu is divided in four parts. The first is the title and the buttons that are related to the general application. The second part is the one were the user can import pre-defined examples, from the server or from the filesystem. The third and fourth are the ones related to the mechanisms, they are divided in two because in the first one all the buttons depend on the input box and in the second one they are independent. The



(a) Left menu as a dropdown (b) Left menu with only buttons

Figure 4.4: Two different menus experimented

concept here was to put things that are related close to each other in order to help the user [54].

In Figures 4.2 and 4.3 it can be seen that, when an automaton is represented, there are also buttons inside the central or left box. Except the one that is meant to close the box, all the other buttons correspond to transformations that are only related to finite automata. For this reason it was decided that it would make more sense if they only appeared when this mechanism was being used, keeping together what belongs together.

In terms of colors, the ones used for the general page are blue and white. Blue because it is a color associated with harmony and calmness and therefore it cannot be connoted with or does not produce negative feelings. When used with grey (color of the buttons) it is also associated with the practical and functional, which are two fundamental characteristics of this web page. [49]

For the acceptance of words and the evaluation of exercises, green and red were chosen (Figure 4.3). The main reason is their general significance in our day to day life, as they are generally associated with right and wrong, go and stop, etc. [49] and thus are self-explanatory when on screen. For functionalities in which we needed to use more than one color but these did not have to be associated with any particular significance, we tried to use colors that stand apart on the color spectrum and stand apart from each other, to keep it easy to read and understand. (Figure 4.2).

4.2 Architectural Pattern

Following the organization of the page in this section we explain the pattern chosen to organize the code and how it was adapted to fit the usage of the *OCaml* language.

It is important to refer that at a basis of this project there is a library developed in the context of another thesis, called OCaml-FLAT. In this project the algorithms offered by the library are graphically represented and animated for the user to be able to interact with them.

OCaml-FLAT is an OCaml library that supports various FLAT concepts, namely finite automata, regular expressions and grammars context-free. The most important goal is for the library code to be understandable to students, meaning that students are able to recognize the relationship between class definitions and implementation code. The original class definitions are free of requirements related to computability. It was required a high level of sophistication to obtain computable and very readable versions those definitions. Dealing with non-determinism and ensuring termination were the biggest challenges.

The code organization pattern chosen for this project was the Model-View-Controller. To maintain this pattern the project is organized in two files from the OCaml-FLAT library, and two *Eliom* files, *OFlatGraphics* and *OFlat* and one *JavaScript* file, *graphLibrary*, that is required to program the graph library and define the *JavaScript* functions that create or modify the graphs. The *Eliom* files are separated because of their functionalities, the *OFlat* file contains all the functions and methods related to the web page and its generation and the *OFlatGraphics* has all the methods that can be reused in other applications, for example if we want to integrate the *Learn OCaml* system.

MVC or Model-View-Controller is a classic architectural pattern typically used to organize user interfaces. The purpose is to divide the system in three logical components that interact with each other [45], the presentation, the interaction and the system data. Since it separates the components, it allows an efficient reuse of the code [1] and each part can change without affecting the others [45]. In this project each file mentioned above corresponds to one of the logical parts, except the *OFlat* that has modules corresponding to controllers and to views, which will be covered later in this chapter.

We use the MVC pattern for conceptual orientation even though, at some rare points, we decided not to apply it strictly to the modular organization of the program. This happens in some specific point, first when we need to separate the view in different files and second in some algorithm animations like the acceptance of the word when we need to turn the methods into a hybrid of the model and the view. What happens in this case is that the code had to be copied to the view file and modified. The new "animated" version

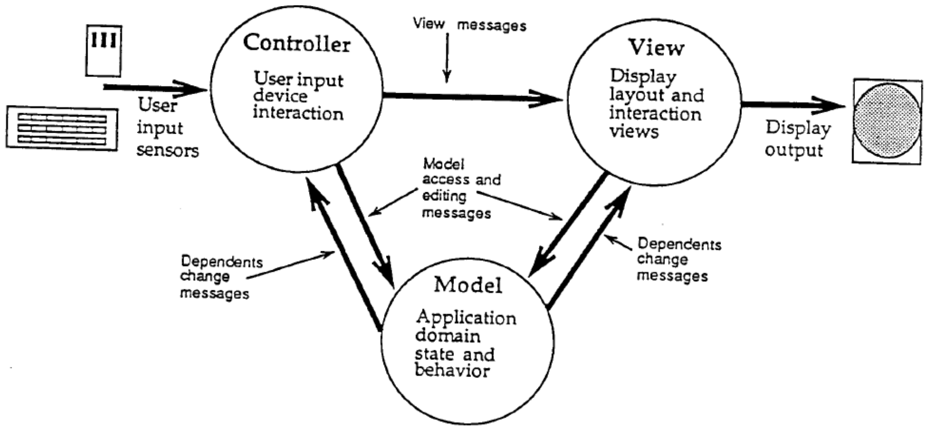

View messages
Controller
User input device interaction
User input sensors
View
Display layout and interaction views
Display output
Model access and editing messages
Dependents change messages
Model
Application domain state and behavior
Dependents change messages

Figure 4.5: Model-View-Controller State and Message Sending as seen in [27]

of the code, instead of just giving a result, now changes the image of the automaton while the operation is being processed.

4.2.1 Model

The model is the application's central structure, as it contains the main functionalities [27]. Since this project is made on top of the OCaml-FLAT library, the model was inherited.

The files used on this project that represent parts of the model are:

- *OCamlFlatSupport* is where the data structures are defined.
- *OCamlFlat* is where all the manipulation algorithms are developed.

4.2.2 View

The view is the component of the system that deals with the graphical part. It's where the aspects of the model are displayed for the user to see [27]. It is in this component that the mechanisms are drawn, and the algorithms are graphically shown to the user.

In this project the view is organized in different modules that are distributed between three files:

- *OFlatGraphics.eliom* is where all the code is related to the graphical representation or animation and also the hybrid methods explained before.
- *oflat.eliom* which has two different modules that are part of the view, one that makes the dynamic modifications of the elements of the page and the other that generates the initial page.
- the *JavaScript* file *graphLibrary.js* that draws and modifies the graphs on the specific division of the page.

The second file, *oflat.eliom*, containing a view module, has a simple reason, and it has to do with the organization of the code. There are two modules, one controller, called *Controller* and one view, called *HtmlPageClient*, that must be mutually recursive because the view has to associate buttons with handlers (part of the controller) and the controller has to call view methods. The buttons put on the page when a mechanism is drawn are an example of this necessity. When the user chooses to draw a mechanism the correspondent *Controller* function is called to do it. In its turn this function must call another from the *HtmlPageClient* module in order to put the necessary buttons on the page. Each of these buttons are created in the *HtmlPageClient* module but if clicked they have to call the *Controller* method to define the right action to take. Since these two modules call each other in *OCaml* they must to be mutually recursive.

In *OFlatGraphics* there are four modules:

- *JS* is a module with *js_of_ocaml* functions to make logs or alerts.
- *Graphics* is the module with functions that call the functions on the JavaScript file.
- *FiniteAutomatonAnimation* is the module with the animation of all the Automata algorithms, thus being an extension of the module *FiniteAutomaton* from the library, it inherits all its class methods.
- *RegularExpressionAnimation* which, like the previous module, it is the one with the all the methods that animate the Regular Expressions and extends the module *RegularExpression* from the library.

In *OFlat* there are two view modules:

- *HtmlPageClient* which is the module that draws the changes on the page, puts buttons and text inside the boxes or cleans them. This module is part of the client side of the application.
- *HtmlPage* that is the module which has all the *HTML* elements, such has buttons and input boxes, that are put on the page when it is generated. Since the page is generated on the server, this module is on the server side of the page.

The generation of the page is also considered part of the view.

4.2.3 Controller

Controllers are the part of the code that handles the input from the user [1]. Controllers contain the interface between their associated models and views and the input devices. [27].

The Controller is activated by callbacks that are registered as *on_click* and *on_change* attributes in graphical elements of the View and then depending on what was clicked decides which system information is changed and what View functions are going to be called.

There are two controller modules and they can be found on the *OFlat* file:

- **Module Controller** is the main controller which indicates to the Views the changes that are going to happen according to the user input.
- **Module FileReaderController** that is in charge of the actions taken when reading a file from the *filesystem*.
- **Module StateVariables** that, as the name convenes, is where all the state of the page is stored and changed. The variables that represent the mechanisms, (automata, regular expressions and exercises) are handlers for the model (the object represents the model or the view on top of it but the variables are in the controller).

4.3 Generating the page and importing files

Now that the reader has a general sense of the organization of the application, how the page is designed and how the code is organized, we can go further deep into explaining more specific parts of the code.

We start by defining how the page is generated, demonstrating how we connect the *Javascript* library *Cytoscape.js*, and how we use *Tyxml* to create and HTML page.

Afterwards we explained the two different methods of importing files that are used to import the different types of mechanisms.

4.3.1 Generating the page

In *Ocsigen* the generation of the page must be done on the server side and implies the creation of services. These are entry points to a website and are generally associated with an URL [34]. For the generation of the page we need to create a registration module and the main service as seen in Listing 4.1. The main service is where we define the path for the main page and its parameters (Line 10 and 11).

At the point of the creation of the main page we register the service with the registration module (Listing 4.2). It is at this point that through the *Tyxml* Library we define the aspect of the main page. Starting on Line 17 and until the end we define different *HTML* divisions where the main elements of the page are inserted, like the buttons of the menu and the titles of the page.

As it can be seen in Listing 4.2 there are scripts imported at lines 9 to 16. These scripts are used to access not only the *Cytoscape.js* library but also the code *Javascript* and *CSS* codes. Like is customary in *Eliom* these scripts and the links used on the footer of the page were created as services. An example of a service used to generate a link for an external service can be seen in Listing 4.3. As explained in Chapter 3, to create a link we

```
module OFlat_app =
  Eliom_registration.App (
    struct
      let application_name = "oflat"
      let global_data_path = None
    end)

let main_service =
  Eliom_service.create
    ~path:(Eliom_service.Path [])
    ~meth:(Eliom_service.Get Eliom_parameter.unit)
    ()
```

Listing 4.1: Registration module and creation of the main service

define its prefix, its path and the meth. In this case we also define how it is presented in the page using *Tyxml* (Line 10).

4.3.2 Importing

4.3.2.1 Import from filesystem

The first option of the menu is to import examples from the local filesystem. The imported files must be JSON files.

The big question was how to access the file system with *Ocsigen*? For security and privacy issues web apps do not have direct access to the files on the user's device. Direct access would allow any JavaScript app to steal or erase documents. To read one local file, we need to use the FileReader API that JavaScript provides. This API is secure because it requires the user to interactively select the particular file to be read. The FileReader API functionality is available in *Eliom*, although with details adapted to the way *Eliom* works. The FileReader usage is demonstrated in Listing 4.5 and will be explained in the remainder of the section.

First, for the user to access the filesystem, it is necessary to create the input box with its input type as a file (Listing 4.4 line 2) and link to it an action to it, in this case *fileWidgetEvents*.

fileWidgetEvents is, actually, a function that associates an action to a change on the input box through the *Js_of_ocaml* module *Lwt_js_events* which programs mouse events with the use of *Lwt* (Listing 4.5 line 38). When there is a new input on the box, the new input is read (Listing 4.5 line 37) and the function *fileWidgetHandle* is called.

fileWidgetHandler makes used of the *Js_of_ocaml* module *Js.Optdef* to manipulate possible undefined values. This is used for the cases in which the user ends up canceling the input of the file and there is no file to be read. With the use of the function *case* (Listing 4.5 line 23), that makes pattern matching on optional values, first the files are read from the input box (Listing 4.5 line 24), if there is no file the function *fileWidgetCanceled* is called (4.5 line 25). This alerts the user that the action has been cancelled. If there is a

```
let () =
  OFlat_app.register
    ~service:main_service
    (fun () () →
      let open Eliom_content.Html.D in
      Lwt.return
         (html
            (head (title (txt "Autómatos Animados"))
                [script ~a:[a_src script_uri1] (txt "");
                 script ~a:[a_src script_uri5] (txt "");
                 script ~a:[a_src script_uri7] (txt "");
                 script ~a:[a_src script_uri8] (txt "");
                 script ~a:[a_src script_uri9] (txt "");
                 css_link ~uri: (
                    make_uri (Eliom_service.static_dir ()) ["codecss.css"]) ();
                 script ~a:[a_src script_uri] (txt "");])
            (body [div ~a:[a_class ["sidenav"]] [
                    div [h2 ~a: [a_id "title"] [txt "OFLAT"];
                         p ~a: [a_id "version"][txt "version 1.1"]];
                    div [HtmlPage.about];
                    div [HtmlPage.feedback]; hr();
                    div ~a:[a_id "fromFilesystem"][HtmlPage.fileWidgetMake ()];
                    hr();
                    div [HtmlPage.serverExamples];
                    div ~a:[a_id "examplesServer"] []; hr();
                    div ~a: [a_id "inputTitle"] [txt "Input:"];
                    div ~a: [a_id "input"] [HtmlPage.inputBox];
                    div [HtmlPage.selectNode];
                    div [HtmlPage.selectTransitions];
                    div [HtmlPage.defineRegExp];
                    div [HtmlPage.words];
                    div [HtmlPage.completeSentence];
                    div [p ~a:[a_id "passo"] [
                        txt "Aceitação passo-a-passo da palavra"]];
                    div [HtmlPage.backwards;
                         HtmlPage.step_by_step; HtmlPage.forward]; hr();
                    div [HtmlPage.selectConvert];];
                div ~a:[a_class ["main"]][
                    div ~a: [a_id "mainTitle"] [h1 [txt "Autómatos Animados"]];
                    div ~a: [a_class ["test"]][
                        div ~a:[a_id "Box1"] [
                            div ~a:[a_id "buttonBox"] [];
                            div ~a: [a_id "regExp"] [];
                            div ~a:[a_id "cy"][]];
                        div ~a:[a_id "Box2"] [
                            div ~a:[a_id "buttonBox1"] [];
                            div ~a: [a_id "textBox"] [];
                            div ~a:[a_id "cy2"] []];
                        div ~a:[a_id "infoBox"][]]];
          footer ~a: [a_class ["footer"]] [txt "Desenvolvido em "; lincs_service;
              txt " dentro do projeto "; factor_service;
              txt "/ Financiado por "; tezos_service]
              ])))
```

Listing 4.2: Creation of the main page

```
let tezos_service =
  Eliom_content.Html.D.a
  (Eliom_service.extern
     ~prefix:"https://tezos.com/"
     ~path:[""]
     ~meth:
       (Eliom_service.Get
          Eliom_parameter.(suffix (all_suffix "suff")))
     ())
    [div ~a:[a_id "footerButton"][txt "Fundação Tezos"]]
    [""]
```

Listing 4.3: Registration module and creation of the main service

```
let fileWidgetMake () =
  let filewidget = input ~a: [a_id "file_input"; a_input_type `File] () in
  let _ = [%client (
      Lwt.async (FileReaderController.fileWidgetEvents ~%filewidget): unit
  )] in filewidget
```

Listing 4.4: Creation of the File button

file, the *Js.Opt* module is used to try and read the file. This module allows to work with optional values. If the file could not be read it activates the function *fileWidgetCanceled* mentioned above.

To read the file the function *onFileLoad* is used (Listing 4.5 lines 11 to 20). If there is no file or the file could not be read it returns a boolean value false, if the file was read the function *fileWidgetAction* is used to start the display of the model represented in the file. This function calls two others, *createText* and *printErrors*.

The *printErrors* is a simple Controller function that upon the verification of the model gives an alert to the user if the representation has some kind of flaw.

The *createText* is used to verify which type of mechanism is going to be drawn and then call the functions that are going to do it. First it makes a series of transformations, from JavaScript string to OCaml string (Listing 4.6 line 2) and from string to the JSon type (Listing 4.6 line 3). Then it reads the element “type” of the representation and afterwards it creates an Automaton, a Regular Expression or an Exercise (defined in the model as enumeration).

4.3.2.2 Import from server

The second option to create an automaton that is given to the user is to choose an example from the server. The general idea is that at the moment of the generation of the page all the files on the example folder are read and is created a button to access each of them individually. The question here was how to access a server folder and then a file from that specific folder. To make this possible it was necessary to use the OCaml module *Sys*

```ocaml
module FileReaderController
 =
  struct
  let fileWidgetCanceled () =
    JS.alert "Canceled"

  let fileWidgetAction txt =
    Controller.createText txt;
    Controller.printErrors ()

  let onFileLoad e =
    Js.Opt.case
      (e##.target)
      (fun () → Js.bool false)
      (fun target →
        Js.Opt.case
          (File.CoerceTo.string target##.result)
          (fun () → Js.bool false)
          (fun data → fileWidgetAction data; Js.bool false)
      )

  let fileWidgetHandle filewidget =
    Js.Optdef.case
      (filewidget##.files)
      (fileWidgetCanceled)
      (fun files →
        Js.Opt.case
          (files##item 0)
          (fileWidgetCanceled)
          (fun file →
            let reader = new%js File.fileReader in
              reader##.onload := Dom.handler onFileLoad;
              reader##readAsText (Js.Unsafe.coerce file)
      )   )

  let fileWidgetEvents filewidget () =
    let fw = Eliom_content.Html.To_dom.of_input filewidget in
      Js_of_ocaml_lwt.Lwt_js_events.changes
        fw
        (fun _ _ → fileWidgetHandle fw; Lwt.return ())
```

Listing 4.5: Module with all the functions used to read a file

```
let createText texto =
      let txt = Js_of_ocaml.Js.to_string texto in
      let j = JSon.from_string txt in
      let kind = JSon.field_string j "kind" in
        if FiniteAutomaton.modelDesignation() = kind then
          (let fa = new FiniteAutomatonAnimation.model (JSon j) in
          defineExample fa)
        else
          if RegularExpressionAnimation.modelDesignation() = kind then
            (let re = new RegularExpressionAnimation.model (JSon j) in
            defineRegularExpression re)
          else
            ( let enu = new Enumeration.enum (JSon j) in
            defineEnum enu)
```

Listing 4.6: Method that defines the type of mechanism to be draw

that gives a system interface, as seen in Listing 4.7 and the *load_file* function from the *OCaml* module *Yojson.basic.util* (Listing 4.7 line 19). But there is a catch: the *Sys* module can only be used in the server side of the code which means that the methods that use it must be in a server module only (Listing 4.7). This also means that the buttons, that have client side functions, could not access this functions. To overcome this problem an intermediate client module was created (Listing 4.8), this one has its methods called by the buttons and in its turn accesses the necessary server functions.

```
 1  module%server Server =
 2    struct
 3      let examplesDirPath =
 4          Sys.getcwd () ^ "/static/examples/"
 5
 6      let log (str: string) =
 7          print_string (str^"\n");
 8          flush stdout;
 9          Lwt.return ()
10
11      let getExamplesList () =
12          let fileList =
13              try
14                  Array.to_list (Sys.readdir examplesDirPath)
15              with _ → [] in
16          Lwt.return fileList
17
18      let getExample fname =
19          Lwt.return (Util.load_file (examplesDirPath ^ fname))
20    end
```

Listing 4.7: Server-side functions to access the list of files on the server

The function *examplesDirPath* (Listing 4.7 lines 3 and 4) provides the directory where all the files are stored thanks to the function from the *Sys* module *getcwd* (Listing 4.7 line

```
module%client Server =
  struct
    let log = (* logs a message in the server console *)
      ~%(Eliom_client.server_function [%derive.json: string] Server.log)

    let getExamplesList =
      ~%(Eliom_client.server_function [%derive.json: unit] Server.getExamplesList)

    let getExample =
      ~%(Eliom_client.server_function [%derive.json: string] Server.getExample)
  end
```

Listing 4.8: Client-side functions to access the list of files on the server

4) that allows the access to the directory with a given path.

With the directory obtained with *examplesDirPath*, the function *getExampleList* (Listing 4.7 lines 11 to 16) tries to make a list of all the files in the folder through the *Sys* function *readdir*. This returns the names of all files in the given directory as a string. If for some reason the directory could not be read an empty list is returned (Listing 4.7 line 15).

```
1 let serverexamples_handler =
2     [%client (fun _ →
3         Lwt.ignore_result (
4           let%lwt lis = Server.getExamplesList () in
5           List.iter (fun el → HtmlPageClient.putButton el) lis;
6           Lwt.return ()
7         )
8     )]
```

Listing 4.9: Function that makes the list of the server examples

At the moment of the generation of the page the function *serverexamples_handler* is called, this gets the list of files with *getExampleList* (Listing 4.9 line 4) and using the list iterator for each element of the list creates a button through the function *putButton* (Listing 4.9 line 13). The *ignore_result* (Listing 4.9 line 3) is seen a few times along the code and is a *Lwt* function that raises an exception if the result is rejected, which makes it easier to debug the program.

To create each button *putButton* was called. This function is used to put the button on its *div* in the page (Listing 4.10).

Each button is created with the *TyXML* format (Listing 4.10 line 2). Each button gets an onclick function that corresponds to getting the specific example with the function *getExample* (Listing 4.10 line 4) and then with its result creating the automaton. To create an automaton the function *createText* is called. This function and the consequent ones were explained in section 4.3.2.1.

```
1  let createServerExampleButton name =
2    button ~a:[a_id "exampleButton"; a_onclick (fun _ →
3      Lwt.ignore_result (
4       let%lwt str = Server.getExample name in
5       Lwt.return (Controller.createText (Js.string str))
6     )
7    )] [txt name]
8
9   let putButton name =
10   (let example =
11     Eliom_content.Html.To_dom.of_button (createServerExampleButton name) in
12   let examples = Dom_html.getElementById "examplesServer" in
13   Js_of_ocaml.Dom.appendChild examples example)
```

Listing 4.10: Function to create the buttons

getExample (Listing 4.7 line 18) is a server function that using the directory given in Listing 4.7 line 4 and the name of the file loads a file.

After the button is created it is transformed into JavaScript DOM element with the *Eliom* module *Eliom_content.Html.To_dom* (Listing 4.10 line 11). To get the identification of the box where the button is going to be put we use *JS_of_ocaml* module *Dom_html* funtion *getElementById* that allows to get html elements from the page through the id (listing 4.10 line 12). Finally we use *JS_of_ocaml* module *Dom* function *appendChild* to add the button to the ones that are already inside the box (Listing 4.10 line 13).

4.4 MVC by Example

This section aims to present a specific example of how the MVC pattern works in this project. By giving an example we intend to provide a better understanding on how each component communicates with the others.

The general idea is illustrated by the Figure 4.5. After the generation of the page, every time there is a user input, like a button click, a controller method indicates what action is supposed to be taken, it verifies the state of the program, defines the new state and then indicates to the view what to represent. The view asks the model the information that it needs to know for the representation and finally it displays on the browser the output. For this process to be better understood we provide an example as follows.

Let's imagine that the page has just been generated and the user chooses to click on an automaton example from the server. For the automaton to be drawn a few modules are going to be used: *HtmlPageClient*, *Controller*, *StateVariables* and *FiniteAutomatonAnimation*.

As the button is clicked, the Controller function *createText* that can be seen in Listing 4.6 is called. The function *createText* transforms the string into the desired format, in which it verifies which mechanism is going to be drawn (in this case it is the finite automaton) and calls the method that organizes the drawing: defineExample (also part of

```
 1 let defineExample example =
 2     if StateVariables.getCy2Type() = StateVariables.getEnumerationType() then
 3         HtmlPageClient.twoBoxes ()
 4     else
 5         HtmlPageClient.oneBox();
 6     HtmlPageClient.putCyAutomataButtons ();
 7     Graphics.destroyGraph();
 8     Graphics.startGraph();
 9     StateVariables.changeCy1ToAutomaton();
10     StateVariables.changeAutomata example;
11     (StateVariables.returnAutomata())#drawExample;
12     defineInformationBoxAutomaton ()
```

Listing 4.11: Code for the creation of the Automaton in module Controller

the Controller module), that can be seen starting in line 1.

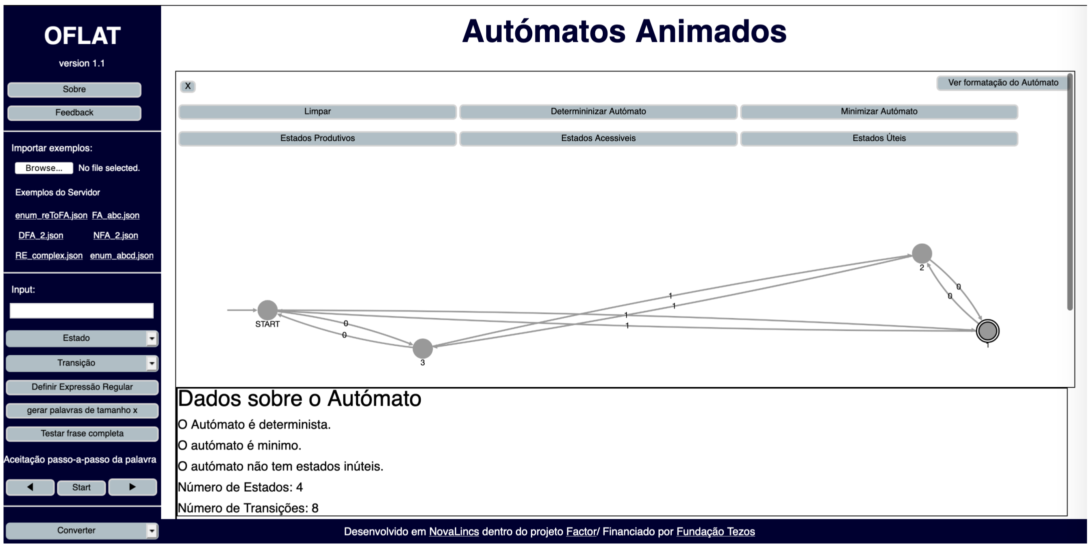
The function *defineExample* (Listing 4.11 line 1 to 12) first accesses a model module, the *StateVariables*, to check the state of the program as seen in line 2 and according to the result it calls the view model *HtmlPageClient* to draw on the screen one or two boxes. After that, everything is set to draw the automaton, and for that different modules are called:

- the view module *HtmlPageClient* is called to put the necessary buttons on the page (Line 6).
- the view module *Graphics* is called to prepare the box or division where the automaton is going to be drawn (Lines 7 and 8).
- the controller module *StateVariables* is called to change the state of the box and the variable that stores the automaton formatting is changed (Line 9 to 10).
- through the state variable automata the view module *FiniteAutomatonAnimation* is called to draw it on the screen (Lines 11 and 12).

In the end we obtain an automaton as represented in Figure 4.6.

All the actions in the page have to go through this system. When an user input happens, a controller function is called to verify the state of the page, sends an edit message to change it and then sends messages to the different views to display the intended graphics and the interaction views.

In this chapter we have attempted to give a better understanding of the application and its code. The reader now has a general idea of how the application works and we can go further by explaining each mechanism and its functionalities.



Dados sobre o Autómato

O Autómato é determinista.

O autómato é minimo.

O autómato não tem estados inúteis.

Número de Estados: 4

Número de Transições: 8

Desenvolvido em NovaLincs dentro do projeto Factor/ Financiado por Fundação Tezos

Figure 4.6: Example of the representation of an Automaton

Chapter 5

Finite Automata

In this Chapter, we are going to explain and analyze the functionalities related to the Automata. The representation of the automata is general and for that reason the automaton can be deterministic or non-deterministic without being specifically defined which. There are multiple options when defining an automaton to work on. We can load it either from the server or from the filesystem or create it step by step on the application. After the creation of the automaton there are several actions that can be carried out: verify if a word is accepted as an animation or step by step; generate all the accepted words unto a given size; convert to regular expression; visualize the type of each state; erase the useless states; make the automaton deterministic; and minimize it.

In Section 5.1 we start by explaining how the Automata are visualized and why and then explain what each functionality does and how it is shown.

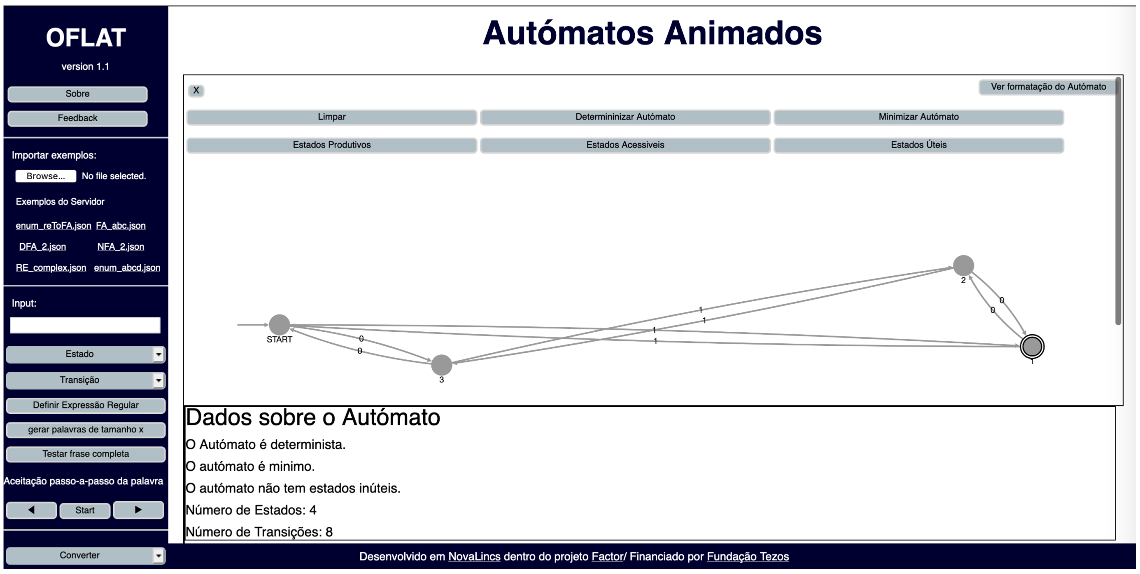
Section 5.2 explains how the automata are represented in the code, its attributes and how the functionalities were developed.

5.1 Functionalities and its presentation

5.1.1 Automata representation

The graphical representation of an automaton (Figure 5.1) is the usual in the literature: the initial state is represented by an arrow going to it, the states are represented as circles, the final states are two concentric circles and the transitions between two states are represented by an arrow going from one to another.

As explained in Chapter 4 and seen in Figure 5.1, some options are available on the lateral menu and others only become available upon the creation of the automaton. This choice has nothing to do with the code but with the fact that the second set of options are



Dados sobre o Autómato

O Autómato é determinista.

O autómato é minimo.

O autómato não tem estados inúteis.

Número de Estados: 4

Número de Transições: 8

Desenvolvido em NovaLincs dentro do projeto Factor/ Financiado por Fundação Tezos

Figure 5.1: Example of a Finite Automaton

only related with the automaton and the ones that appear on the menu can be applied to more than one of the mechanisms that the application supports.

When there is an automaton in the screen it makes use of the whole central box (except for the space occupied by the buttons) since there is no other mechanisms to be shown simultaneously.

While the automaton is uploaded, some information about it is generated and shown in the bottom box. This is extra information that can be helpful for the user but it is not essential. For that reason it does not have to be in a prominent position.

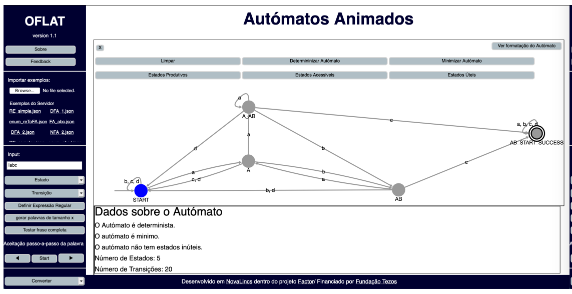
As stated earlier the automaton can be created in three different ways: imported from the filesystem, imported from the server or created step by step. The first two generate a pre-created automaton that fits to its designated space in the screen. The third allows the user to create, modify and erase states and transitions at will. For that the user has to input the name of the state in the input box and then choose if he wants to add an initial, a final or a regular state or erase the state. In the case of the transitions he has to input the name of the starting state, the transition symbol and the name of the arrival state and then choose add or erase transition. But there are a few rules:

- The first state, no matter which option the user chooses is going to be an initial state - the graph needs to have an initial state to be considered an automaton;
- when adding a transition, the states must exist in the representation;
- an initial state cannot be eliminated - the user must first change the initial state and then erase the one he wants to eliminate.

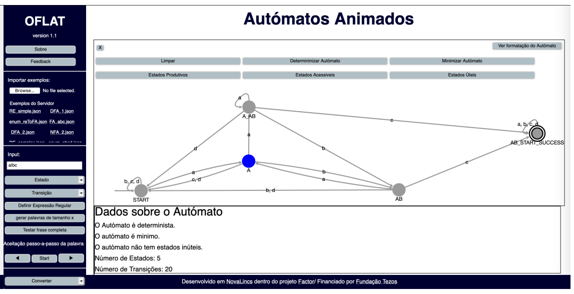
The options to add and erase state and transitions can also be used after importing an example, also, the options to add initial or final state can be used to modify an already existent state, to do so, the user only has to indicate the name of the state and choose the option.

Every time the figure is changed, its size is adapted to fit the size of the box.

5.1.2 Accept

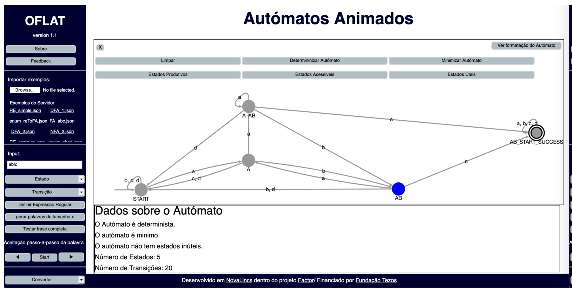


Dados sobre o Autómato
O Autómato é determinista.
O autómato é mínimo.
O autómato não tem estados inúteis.
Número de Estados: 5
Número de Transições: 20

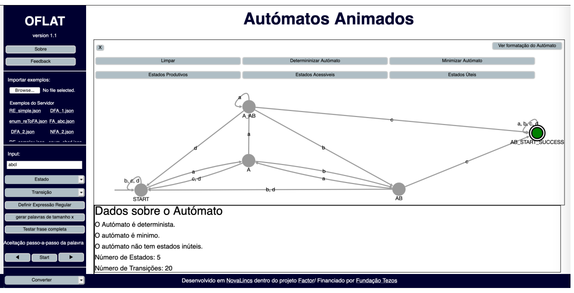


Dados sobre o Autómato
O Autómato é determinista.
O autómato é mínimo.
O autómato não tem estados inúteis.
Número de Estados: 5
Número de Transições: 20

(a) Initial State (b) Step State



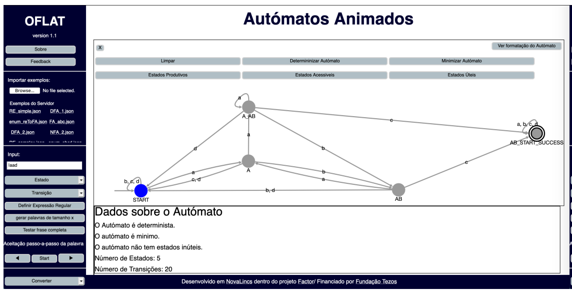
Dados sobre o Autómato
O Autómato é determinista.
O autómato é mínimo.
O autómato não tem estados inúteis.
Número de Estados: 5
Número de Transições: 20



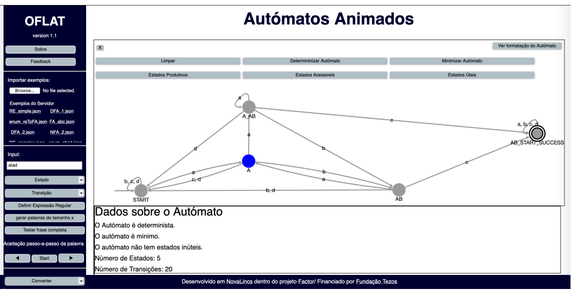
Dados sobre o Autómato
O Autómato é determinista.
O autómato é mínimo.
O autómato não tem estados inúteis.
Número de Estados: 5
Número de Transições: 20

(c) Step State (d) Final State

Figure 5.2: Verification of the acceptance of a word - accepted word

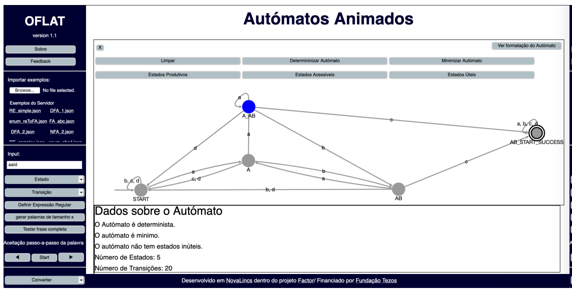


Dados sobre o Autómato
O Autómato é determinista.
O autómato é mínimo.
O autómato não tem estados inúteis.
Número de Estados: 5
Número de Transições: 20

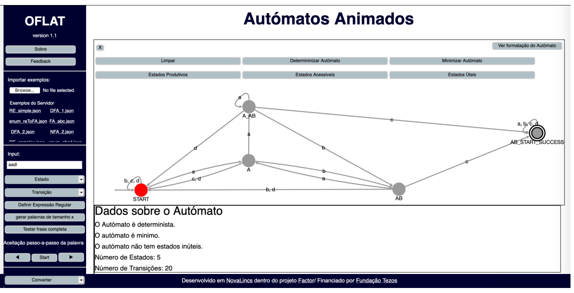


Dados sobre o Autómato
O Autómato é determinista.
O autómato é mínimo.
O autómato não tem estados inúteis.
Número de Estados: 5
Número de Transições: 20

(a) Initial State (b) Step State



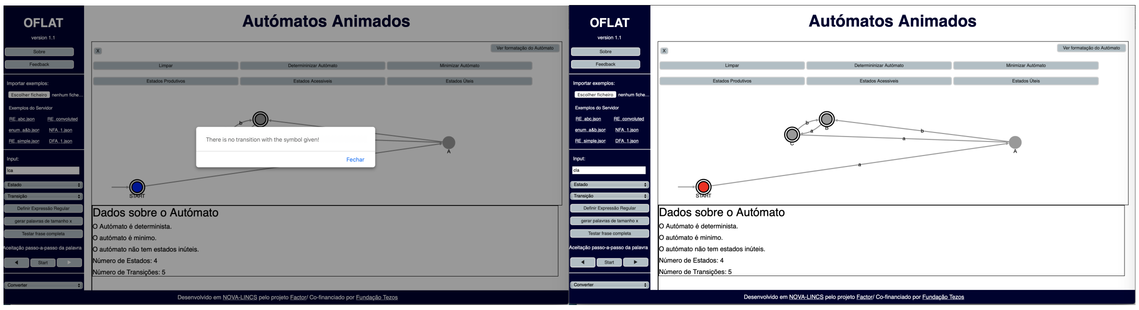
Dados sobre o Autómato
O Autómato é determinista.
O autómato é mínimo.
O autómato não tem estados inúteis.
Número de Estados: 5
Número de Transições: 20



Dados sobre o Autómato
O Autómato é determinista.
O autómato é mínimo.
O autómato não tem estados inúteis.
Número de Estados: 5
Número de Transições: 20

(c) Step State (d) Final State

Figure 5.3: Verification of the acceptance of a word - non-accepted word



lca

There is no transition with the symbol given!

cla

Dados sobre o Autómato
O Autómato é determinista.
O autómato é mínimo.
O autómato não tem estados inúteis.
Número de Estados: 4
Número de Transições: 5

(a) No transitions alert

(b) Painted last reached state

Figure 5.4: Verification of the acceptance of a word where there is no transitions with given symbol

Figure 5.2 shows the process of accepting a word by an automaton (in this case the word is accepted). The idea is simple. The user inputs a word in the text box and by clicking the button "Testar frase completa" he can see the states changing color until the whole word is tested: blue, if it is an intermediate state; red, if the word has come to an end but the state is not final (Figure 5.3); and green, if the word has come to an end and the state is final (Figure 5.2). If the word has a symbol that does not correspond to a transition when reaching that symbol the user is alerted and the verification is finalized by painting the last reached state in red (Figure 5.4).

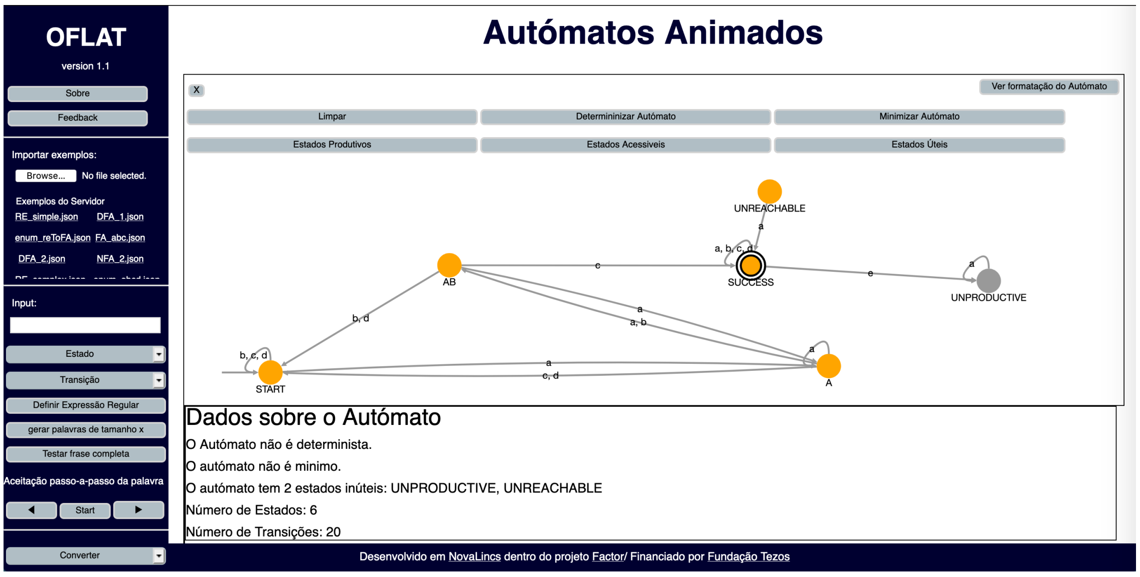
There are two types of functionalities related to the word acceptance: one, already described, allows the user to visualize an animation of the process, the states change colors automatically until the word is accepted or rejected; the second allows the user to control the animation through button clicks. This second option relies on three buttons - start, forward and backwards - and the general idea is to let users walk through the automaton at their own pace to better understand where its reasoning is going, wrong or right.

In the step-by-step option the word in the input box is changed at each step for the user to know what part of the word has been consumed. For example, let us say we are testing the word "abc". If we are in the beginning and only click in the "start" button the word is going to be shown as "|abc" but if we already tested "ab" it is going to show "ab|c".

5.1.3 Type of states

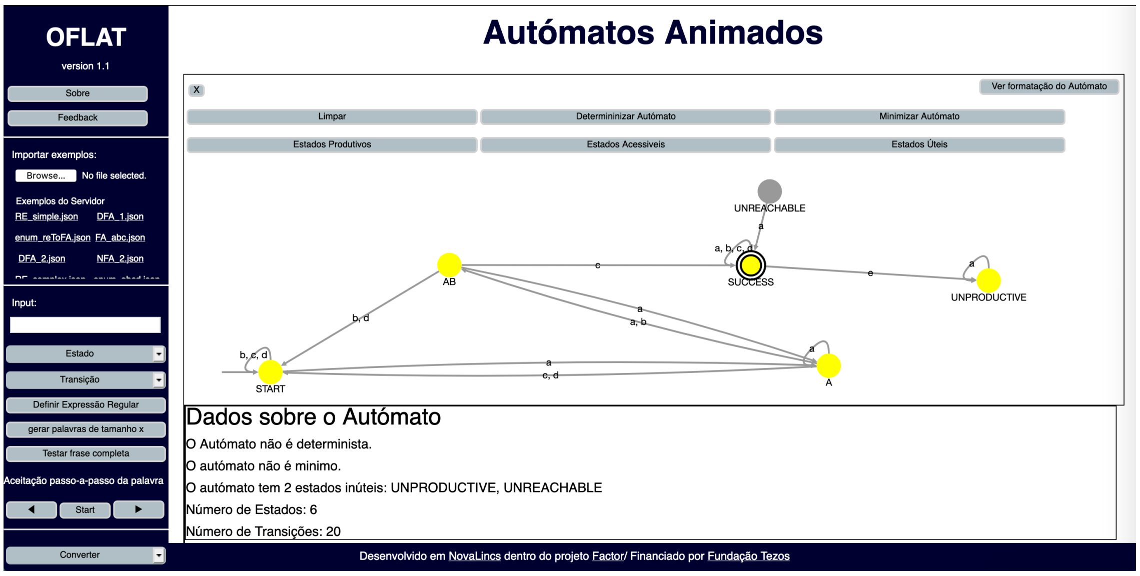
There are three reachability related properties that a state can have. Out application provides three commands to allow the user to understand if the states are:

- **productive** (Figure 5.5), a state from which it is possible to reach a final state.
- **accessible** (Figure 5.6), a state reachable from the initial state.
- **useful** (Figure 5.7), a productive and accessible state.



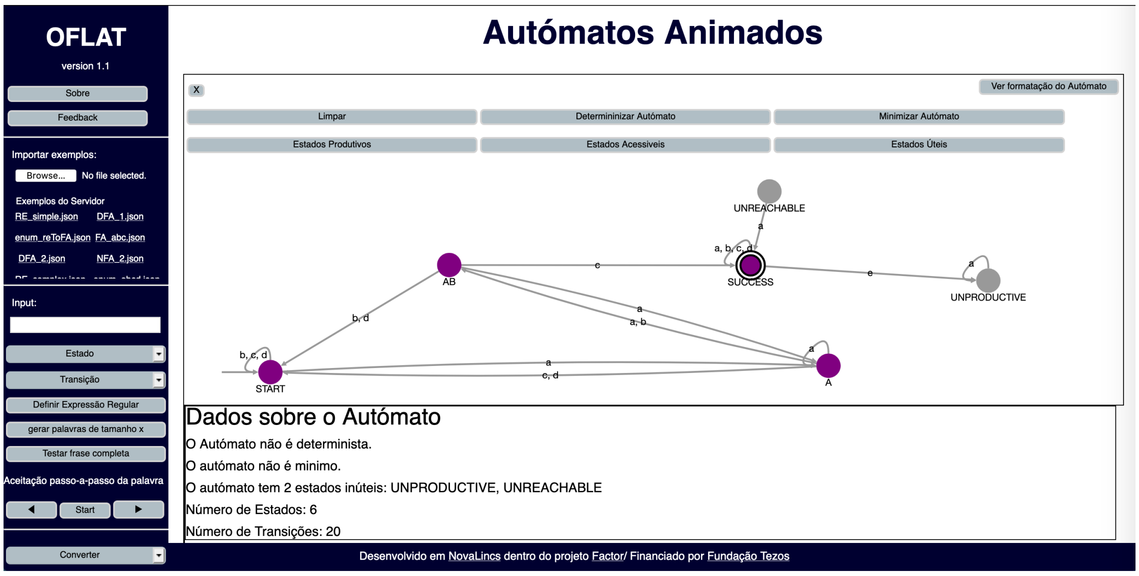
Dados sobre o Autómato
O Autómato não é determinista.
O autómato não é minimo.
O autómato tem 2 estados inúteis: UNPRODUCTIVE, UNREACHABLE
Número de Estados: 6
Número de Transições: 20

Figure 5.5: Indication of the productive states



Dados sobre o Autómato
O Autómato não é determinista.
O autómato não é minimo.
O autómato tem 2 estados inúteis: UNPRODUCTIVE, UNREACHABLE
Número de Estados: 6
Número de Transições: 20

Figure 5.6: Indication of the reachable states

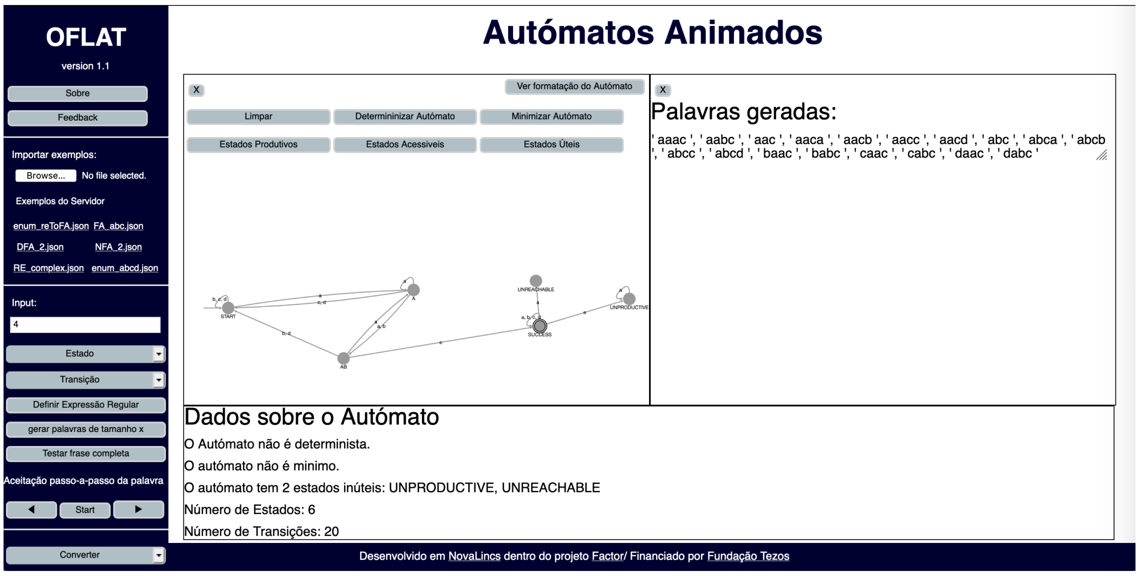


Dados sobre o Autómato
O Autómato não é determinista.
O autómato não é minimo.
O autómato tem 2 estados inúteis: UNPRODUCTIVE, UNREACHABLE
Número de Estados: 6
Número de Transições: 20

Figure 5.7: Indication of the useful states

For each of these options there is a button that marks, simultaneously, all of the states that satisfy that characteristic.

5.1.4 Generation



Palavras geradas:
' aaac ', ' aabc ', ' aac ', ' aaca ', ' aacb ', ' aacc ', ' aacd ', ' abc ', ' abca ', ' abcb ', ' abcc ', ' abcd ', ' baac ', ' babc ', ' caac ', ' cabc ', ' daac ', ' dabc '

Dados sobre o Autómato
O Autómato não é determinista.
O autómato não é minimo.
O autómato tem 2 estados inúteis: UNPRODUCTIVE, UNREACHABLE
Número de Estados: 6
Número de Transições: 20

Figure 5.8: Generating accepted words with maximum size 4

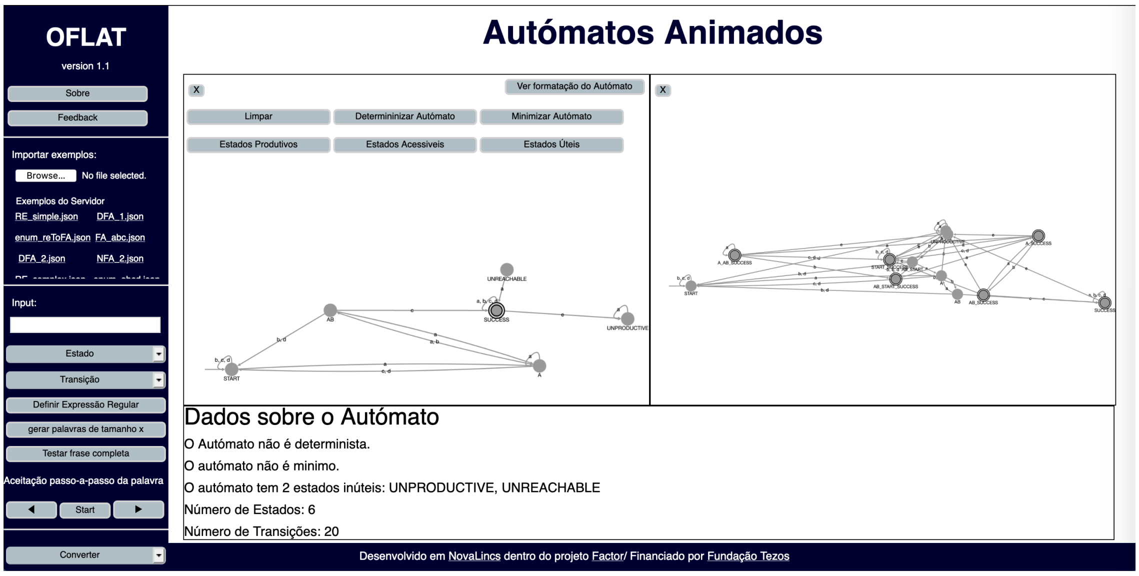
To help the user understand the language defined by an automaton, there is a generate command that shows a list of words that are accepted by the automaton. The user has to input a number in the input box and then the page shows all the words that the automaton accepts with a length up to the given number, so if the user input a 4 it generates all the accepted words with size 0, 1, 2, 3, and 4.

The list of words was firstly presented in the bottom box but from the moment we started having two boxes, we found it better to display it on the right one. This was because it did not fit the purpose of the bottom box which shows characteristics of the automaton.

An example can be seen in Figure 5.8. In this case, the automaton does not accept words with length smaller than 3 which means that by inputting the number 4 it only generate words with size 3 and 4.

Since the accept method does not make use of the second box, it is also possible to test the acceptance of the words while they are shown in the right box.

5.1.5 Convert to Deterministic



Dados sobre o Autómato

O Autómato não é determinista.

O autómato não é minimo.

O autómato tem 2 estados inúteis: UNPRODUCTIVE, UNREACHABLE

Número de Estados: 6

Número de Transições: 20

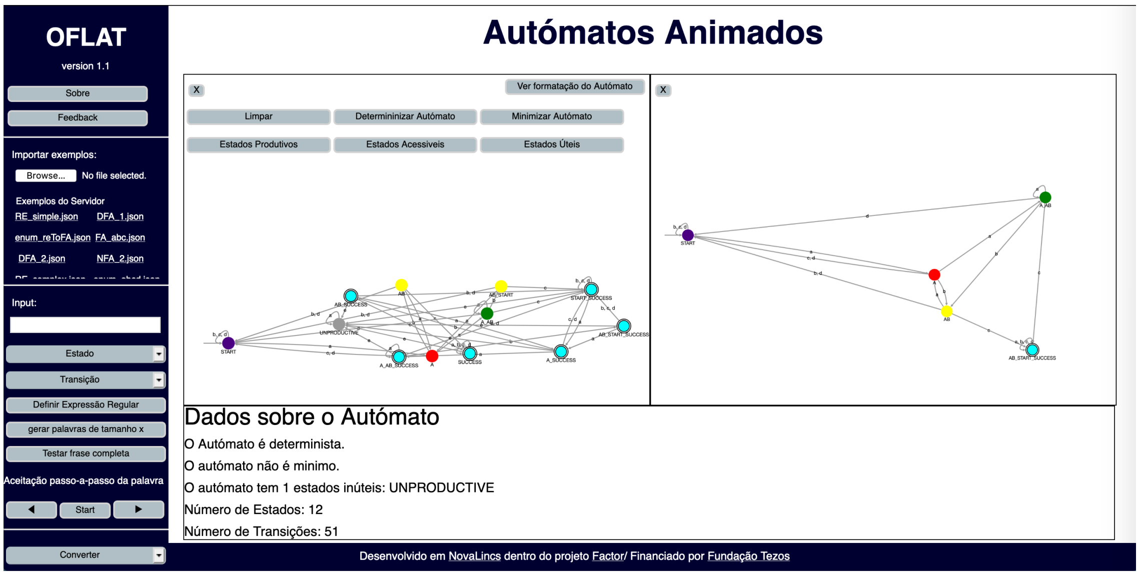
Figure 5.9: Transforming the Automaton in a Deterministic one

With this functionality, a new question, that extended to other functionalities, emerged: how to show two automata at the same time. It was necessary not to substitute the automaton the user was working with but to show the two versions side by side. This way the can compare the two options and understand the modifications.

To achieve the desired result, we decided to use the two-box system. When there is only one automaton in the page the main box is full and occupies the width of the page. When one of these functionalities is executed, the main box shrinks by half and a new box appears on the right with the new automaton. An example of the result can be seen in Figure 5.9.

At this point there is no actual animation for this element, mostly because we are defining mechanisms that are taught in class one at a time. For each mechanism we try to have the maximum functionalities possible even if it means that they are not already animated. To show the transformation from one automaton to the other, in this functionality it was necessary to use a table of transitions (that we see as new mechanism), represented in the library algorithm as different types of sets, and find a way to draw it in the page. Even though the functionality is represented, its animation is listed in the future work.

5.1.6 Minimize



Dados sobre o Autómato
O Autómato é determinista.
O autómato não é minimo.
O autómato tem 1 estados inúteis: UNPRODUCTIVE
Número de Estados: 12
Número de Transições: 51

Figure 5.10: Minimization of the Automaton

This functionality, much like the one before, makes use of the two boxes of the page. In one box it shows the original automaton and in the other it shows the minimized automaton. In the original automaton each set of states that are minimized to a single one is painted in same color and the matching state in the minimized automaton is painted in the same color as can be seen in Figure 5.10.

The minimization algorithm, analyzes which states could be just one and merges them together into a new automaton. Separating the states by clusters to show which became which, facilitates the comparison and helps the student understand the algorithm.

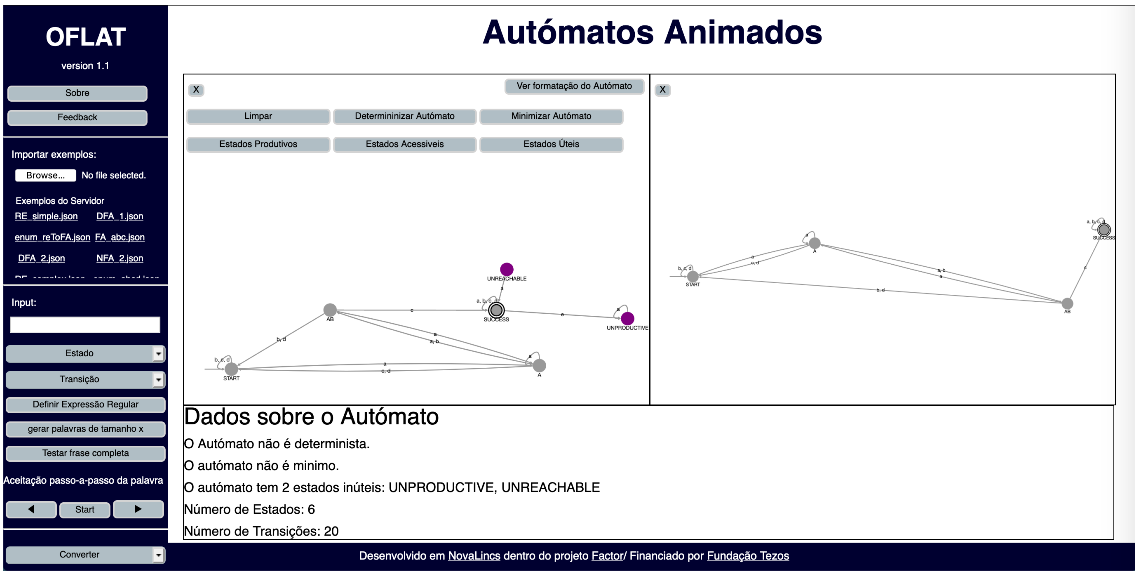
5.1.7 Clean the Useless States

This option erases all the useless states of the automaton, that is, all the states that are not productive or reachable.

This is one of the functionalities that divide the central box in two smaller ones. In the right box a new automaton appears without the useless states and on the left box the useless states of the automaton are colored. In this way, the user can see what are the states that disappear.

5.1.8 Convert

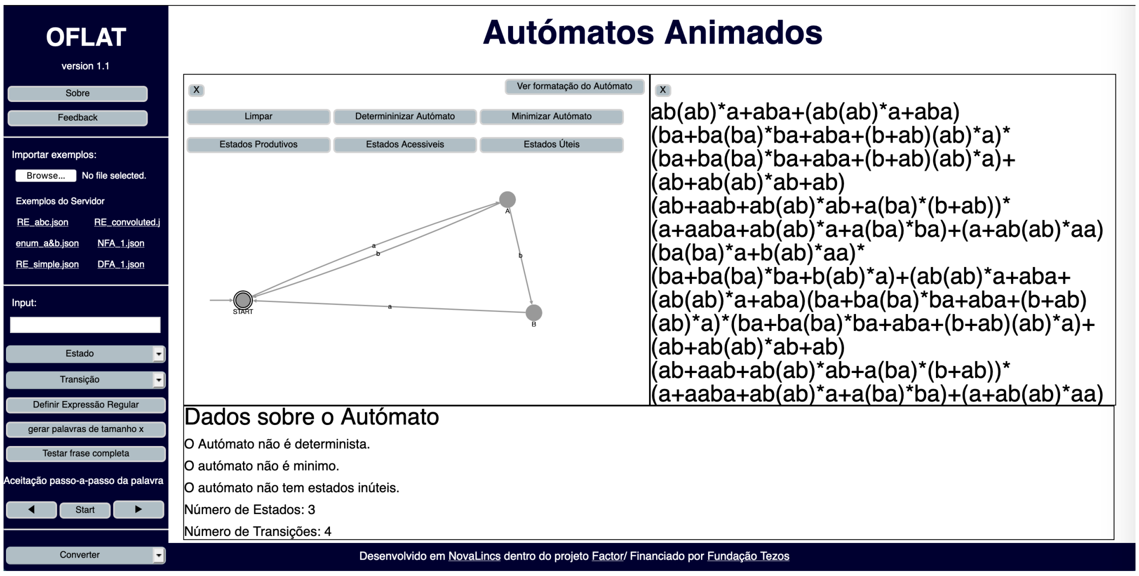
The last functionality available for the Automata mechanism is the convert functionally. Since now the application has two mechanisms, Finite Automata and Regular Expressions



Dados sobre o Autómato
O Autómato não é determinista.
O autómato não é mínimo.
O autómato tem 2 estados inúteis: UNPRODUCTIVE, UNREACHABLE
Número de Estados: 6
Número de Transições: 20

Figure 5.11: Clean the Useless States

the conversion, possible at this point is to Regular Expression.



ab(ab)*a+aba+(ab(ab)*a+aba)
(ba+ba(ba)*ba+aba+(b+ab)(ab)*a)*
(ba+ba(ba)*ba+aba+(b+ab)(ab)*a)+
(ab+ab(ab)*ab+ab)
(ab+aab+ab(ab)*ab+a(ba)*(b+ab))*
(a+aaba+ab(ab)*a+a(ba)*ba)+(a+ab(ab)*aa)
(ba(ba)*a+b(ab)*aa)*
(ba+ba(ba)*ba+b(ab)*a)+(ab(ab)*a+aba+
(ab(ab)*a+aba)(ba+ba(ba)*ba+aba+(b+ab)
(ab)*a)*(ba+ba(ba)*ba+aba+(b+ab)(ab)*a)+
(ab+ab(ab)*ab+ab)
(ab+aab+ab(ab)*ab+a(ba)*(b+ab))*
(a+aaba+ab(ab)*a+a(ba)*ba)+(a+ab(ab)*aa)

Dados sobre o Autómato
O Autómato não é determinista.
O autómato não é mínimo.
O autómato não tem estados inúteis.
Número de Estados: 3
Número de Transições: 4

Figure 5.12: Conversion of finite Automaton in Regular Expression

As it can be seen in Figure 5.12, the conversion appears in a second box, much like some of the functionalities already demonstrated, consisting of text only. Why is it so? As explained before, one of the reasons to use *OCaml* was to create a library were the algorithms are the most similar to those taught in the class of Theory of Computation. This means that sometimes the algorithms are not the best to animate and could give somewhat extended results. There is already a simplifying method under use, but to simplify it even further would mean going beyond the algorithms given in class which was not the purpose of the project at this stage, it is however an objective for the future.

5.2 Implementation

This section will explain how the automaton and its functionalities were implemented to obtain the visual representations explained in the previous section. We start by explaining how the automata is represented in the system and the information to insert in the file to be imported to construct an Automaton, followed by an explanation of the code for each of the functionalities.

5.2.1 Definition of the Automaton

In terms of *OCaml* the Automaton is an instance of a class on the *FiniteAutomatonAnimation* module and its definition consists of:

- alphabet - a set of *chars* corresponding to the alphabet accepted by the automaton.
- allStates - a set with all the states that compose the automaton
- initialState - the initial state of the automaton
- transition - a set of all the automaton transitions with the format (start state, char belonging to the alphabet, finish state)
- acceptStates - a set with all the accept states of the automaton

This is a general representation of the automaton that can represent deterministic or non-deterministic automata. These are semantic properties that are taken care of or treated in the application. Theory separates the two but in order to simplify the application and avoid having repeated operations, they are translated to the same data structure.

The automaton drawn in the page is always stored in a system variable called *Automata* or *Atomata1* depending on which box displays it. Also, there are two state variables that represent the boxes where the mechanisms are drawn and indicate which mechanism is drawn in each box, *cyType* and *cyType2*. These variables are part of the module *StateVariables*, which gives the controller functions to read and modify the variables.

5.2.2 Load file

As explained before there are two types of import operations: from filesystem and from server. The files have to be in JSon or txt format and the encoding of an automaton is depicted in Listing 5.1.

The field *kind* identifies the mechanism that is going to be represented, it is important for the import method to correctly define the mechanism. The *description* and the *name* fields are not mandatory but allow a better identification of the examples. The remaining fields compose the representation of the automaton mentioned in Section 5.1.1 and need to be correctly filled for the system to create the correct automaton. The *alphabet* is a list of

```
{
  kind : "finite automaton",
  description : "",
  name : "010",
  alphabet : ["0", "1"],
  states : ["00", "01", "10", "11"],
  initialState : "00",
    transitions : [
            ["00","1","01"], ["00","0","10"], ["01","1","00"], ["01","0","11"],
            ["11","0","01"], ["11","1","10"], ["10","1","11"], ["10","0","00"]
        ],
    acceptStates : ["01"]
}
```

Listing 5.1: Example of a file with the representation of an automaton

characters, the *states* and the *acceptStates* are a list of strings, the *initialState* is a string and the *transitions* is a list of triples composed by start state, character of the alphabet and end state. It is also important that all the states that appear in the *initialState*, *acceptStates* and *transition* correspond to the ones in the *states* list and that the characters which appear in the transitions are part of the alphabet.

The Section 4.3.2 explains how the two import systems work right to the point where the method *createText* is called. Then it is decided which mechanism is going to be represented.

5.2.3 Creation

There are two types of automaton creation: by importing a complete file and through addition of states and transitions (the second one also allows the addition of this component to imported automata). Even though they pass through different stages when accessing the JavaScript file, they do it in the same way. There is a function to initiate the graph, one to add states and one to add transitions.

When the automaton is imported to draw it the function *defineExample* (Listing 4.6 line 7) is called.

How is the box prepared to have the automaton and how is the automaton drawn? It is at this point that we start using the *Cytoscape.js* library and the functions created in the *JavaScript* file to work with it. As written in Chapter 4, all the imports to access the *Cytoscape.js* and the *JavaScript* functions are put at the beginning of the *HTML*. Then we need to use *Js_of_ocaml* to access a specific function on the *JavaScript* file. The functions that implement access to *JavaScript* from *OCaml* are in a module called *JS*. Two of these functions can be seen in Listing 5.3, the exec one is called with the name of the *JavaScript* function. This means that *JavaScript* functions are called as a string, which, due to some issues related with the *Ocsigen* API, was the best way found to do it.

To prepare the automaton, we make use of two functions (Listing 5.2 line 9 and 10):

```
1  let defineExample example =
2      if StateVariables.getCy2Type() = StateVariables.getEnumerationType() then
3          HtmlPageClient.twoBoxes ()
4      else
5          HtmlPageClient.oneBox();
6      HtmlPageClient.putCyAutomataButtons ();
7      StateVariables.changeCy1ToAutomaton();
8      StateVariables.changeAutomata example;
9      Graphics.destroyGraph();
10     Graphics.startGraph();
11     (StateVariables.getAutomata())#drawExample;
12     defineInformationBoxAutomaton ()
```

Listing 5.2: Controller method that handles the creation of the automaton

```
 let eval s = Js_of_ocaml.Js.Unsafe.eval_string s

let exec s = ignore (eval s)
```

Listing 5.3: JS module functions to call JavaScript

destroyGraph() cleans the HTML page division, box, if it already has a graph drawn in it. *startGraph* defines in the box the characteristics of the new automaton even though it does not draw states and transitions immediately.

It is now possible to draw the new automaton. For that the method *drawExample* (Listing 5.2 Line 11) calls two other methods, first *inputNodes* (Listing 5.4) and second *inputEdges* (Listing 5.5). It must be in this order, because to draw the transitions, the states must exist already, hence all the states are drawn before the transition. This prevents the user from worrying about the order in which the transitions are put in the file.

```
method inputNodes  =
    Set.iter (fun el →
        (Graphics.createNode el (el = self#representation.initialState) (Set.belongs el self#representation.acceptStates))
    ) self#representation.allStates
```

Listing 5.4: Method to call the drawing of the states of the automaton

The *inputNodes* (Listing 5.4) for each state calls the Graphics function *createNode*, giving the name of the state, if it is an initial state, and if it is a final one. *createNode* calls the *JavaScript* function with the same information, which, in turn, creates the nodes with the corresponding characteristics.

The method *inputEdge* (listing 5.5) for each transition of the automaton calls the method *createEdge* that, in turn, calls the *JavaScript* method responsible for creating transitions.

The last option to create an Automaton is step-by-step. This method not only allows

```
method inputEdges  =
   Set.iter (fun el →
       (Graphics.createEdge el)
   ) self#representation.transitions
```

Listing 5.5: Method to call the drawing of the transitions of the automaton

the users to create an automaton by inputting states and transition at will, but also enables the user to modify the automata that is imported from server or filesystem.

Since there were several options related to states and transitions, we decided to use select boxes instead of buttons to represent each element, state or transition, in an attempt to simplify the menu. In the select box for the states, the user can choose to add a state, an initial state, a final state or erase them. For the transitions the user can choose to add or erase a transition. Before creating or eliminating a state or transition, the user must input the name of the state in the input box or a triple with start state transition and arrival state separated by a space (ex. state transition state).

5.2.3.1 Creating a state

There are three different methods to create the three different types of states (each corresponding to an option on the select box): initial state, final state and regular state. Both the option, initial state and final state, allow to modify an already existent state.

The three functions are very similar (an example of one of these functions can be seen in Listing 5.6 that represents the creation or change of an initial node) and start by doing the same verification, if an automaton already exists, if not then it means that it is the first state to be created and is going to be an initial one. If we are adding the first state of the automaton the function prepares the box and the variables set to start a new automaton in the main box (Listing 5.6 Lines 9 to 10), creates a new automaton with the specified initial state (Line 15) and finally calls the function to draw a state on the screen (Line 20).

If an automaton already exists, each function makes its own changes. If we are adding an initial state, we know that one already exists since there is no automaton without it, so we first need to turn the previous initial node into a normal one, hence we eliminate it and add it again in the representation (Lines 3 to 6). Then *addInitialNode* method is called to add a new initial state (Line 9) and finally the automaton is graphically redrawn (Line 11). Why do we not just add the two states to the graphics? If we had erased the graphical representation of the previous initial state (and the state we are adding if it existed) we would have also erased all the transitions, which meant that to add then we would have to find which transition was erased and which one was not. It seemed simpler to reuse the methods we already had and to redraw the automaton.

Adding a final state differs in one thing: there can be more than one final state which means no state has to be erased. If the indicated state does not exist it simply adds it as a new state and to the list of final states, if it does it changes the definition of the automaton

```
let addInitialNode node =
    if StateVariables.getCy1Type() = StateVariables.getAutomatonType() then
        let getInitial = (StateVariables.returnAutomata())#representation.initialState in
        let isFinal = Set.belongs getInitial (StateVariables.returnAutomata())#representation.acceptStates in
        StateVariables.changeAutomata ((StateVariables.returnAutomata())#eliminateNode getInitial true isFinal);
        StateVariables.changeAutomata ( (StateVariables.returnAutomata())#addNode blah false);
        let stateExists = Set.belongs node (StateVariables.returnAutomata())#representation.allStates in
          StateVariables.changeAutomata ((StateVariables.returnAutomata())#addInitialNode node false stateExists);
          Graphics.createNode node true false);
        Graphics.destroyGraph();
        defineExample (StateVariables.returnAutomata()));
        defineInformationBoxAutomaton ())
      else
        (if StateVariables.getCy2Type() ≠ StateVariables.getEnumerationType() then
          (HtmlPageClient.oneBox());
        HtmlPageClient.putCyAutomataButtons();
        Graphics.startGraph ();
        StateVariables.changeCy1ToAutomaton ();
        StateVariables.changeAutomata ((StateVariables.returnAutomata())#addInitialNode node true false);
        Graphics.createNode node true false;
        defineInformationBoxAutomaton ())
```

Listing 5.6: Controller method to create initial state

to add it in the final states. In the end it changes the graphical representation.

Adding a normal state differs for a reason: it does not allow to modify and existent state it only allows to add new ones. This means that if we want to transform a final state into a normal one, for example, we need to eliminate it and then add it again.

5.2.4 Eliminating a state

To eliminate a state, it is necessary to specify its name in the input box. When the option is clicked the handler reads the input and the controller function *eliminateNode* (Listing 5.7) is called. This method before making the elimination executes the following verifications:

- **if the state is initial** (Lines 2 to 3) - a initial state can never be eliminated because the automaton needs to have an initial state to be considered one. In an alert message it is suggested that the user changes the initial state before erasing it.

- **if the state does not belongs to the automaton** (Lines 5 and 12) - an nonexistent state can not be eliminated and the user is alerted that the state does not exist.

```
1  let eliminateNode node =
2      if (node = (StateVariables.returnAutomata())#representation.initialState) then
3          JS.alert ("Não é possível eliminar estado inicial, troque o estado inicial para outro e depois elimine o indicado!")
4      else
5          if (Set.belongs node (StateVariables.returnAutomata())#representation.allStates) then
6              (let isFinal = Set.belongs node (StateVariables.returnAutomata())#representation.acceptStates in
7              StateVariables.changeAutomata ((StateVariables.returnAutomata())#eliminateNode node false isFinal);
8              Set.iter (fun el → (eliminateNodeTransitions el node)) (StateVariables.returnAutomata())#representation.transitions;
9              Graphics.eliminateNode node;
10             defineInformationBoxAutomaton ())
11         else
12             JS.alert ("O estado indicado não existe!")
```

Listing 5.7: Controller function to eliminate a state

If none of the previous are true, then it eliminates the node from the representation (Line 7) followed by the elimination of all transitions that have that state (as a start or an arriving state) with a specific function (Line 8). Then it calls the function that eliminates the node from the graphical representation (Line 9). The *cytoscape.js* library function to eliminate the nodes automatically erases all the transitions related to it from the graphical representation.

5.2.5 Creating a transition

To create a transition, it is necessary to indicate the state where it starts, the symbol of the transition and the state where it arrives, these three elements are inserted in the input box separated by spaces. When the create transition option is clicked the handler reads the input and calls the controller function *createTransition* (Listing 5.8). This method before creating the transition, executes several verifications:

- **if an automaton is represented on the screen** (line 2) - to create a transition it is necessary to have states.
- **if the transition already exists** (Line 2) - a transition cannot be created twice.
- **if the starting state exists** (Line 8) - to create a transition it needs to start in an existing state.
- **if the arriving state exists** (Line 11) - to create a transition it needs to end in an existing state.

If any of the verifications fail, users have made a mistake and it is necessary to alert them. If not, then the methods add the transition to the representation with the automata

```
1  let createTransition (v1, c3, v2) =
2        if (StateVariables.getCy1Type() ≠ StateVariables.getAutomatonType()) then
3           JS.alert ("Antes de criar uma transição é necessário ter dois estados")
4        else
5        if (Set.belongs (v1, c3, v2) (StateVariables.returnAutomata())#representation.transitions) then
6          JS.alert ("A transição (" ^ v1 ^ ", " ^ String.make 1 c3 ^ ", " ^ v2 ^ ") já existe!")
7        else
8        (if (Set.belongs v1 (StateVariables.returnAutomata())#representation.allStates) != true then
9              JS.alert ("O estado de partida não existe!")
10           else
11             if (Set.belongs v2 (StateVariables.returnAutomata())#representation.allStates) != true then
12               JS.alert ("O estado de chegada não existe!")
13             else
14               ((ignore (StateVariables.changeAutomata ((StateVariables.returnAutomata())#newTransition (v1, c3, v2))));
15               Graphics.createEdge (v1, c3, v2);
16               defineInformationBoxAutomaton ()))
```

Listing 5.8: Controller function to create a transition

method *createTransition* (Line 14) and calls the method to change the graph on the screen *createEdge* (Line 15).

5.2.6 Eliminating a transition

```
let eliminateTransition (v1, c3, v2) =
    if (Set.belongs (v1, c3, v2) (StateVariables.returnAutomata())#representation.transitions) then
        (StateVariables.changeAutomata ((StateVariables.returnAutomata())#eliminateTransition(v1, c3, v2));
        Graphics.eliminateEdge (v1, c3, v2);
        defineInformationBoxAutomaton ())
    else
        JS.alert ("A transição (" ^ v1 ^ ", " ^ String.make 1 c3 ^ ", " ^ v2 ^ ") não existe!")
```

Listing 5.9: Controller function to eliminate a transition

To eliminate a transition is as simple as adding one, the user inputs the sequence start state - transition - arrival state and the controller function (Listing 5.9) calls the library method that verifies it exists; if so, it calls the automaton method to eliminate the transition and calls the method from the Graphics module that eliminates the transition from the graphical representation.

5.2.7 Accept

As previously mentioned, there are two types of visualization of the acceptance of a word by an automaton: animated and step-by-step.

For the acceptance animation to happen, when the button is clicked the input of the text box is read and the function *accept* on the module controller is called.

```
1 let completeSentence_handler = [\%client (fun _ →
2     let i = (Eliom_content.Html.To_dom.of_input ~\%inputBox) in
3     let v = Js_of_ocaml.Js.to_string i##.value in
4       Controller.accept v
5     )]
```

Listing 5.10: Handler of the Accept Button

Listing 5.10 shows the handler for the accept button and as can be seen, to test the word it is necessary to go through two steps: first we need to read the text box (Line 2) and then we have to transform the variable to an *OCaml* string. To achieve the conversion, we make use of two functions. *Eliom_content.Html.To_dom* is an *Eliom* module that makes the conversion from *HTML5 elts* to *Javascript DOM* elements and *Js_of_ocaml.Js.to_string* is a *Js_of_ocaml* function to transform a JavaScript string into an OCaml string. With the string in OCaml the method accept is called.

As explained in Chapter 4, in *Eliom* the page is generated in the server. Due to the input box being put on the page at the time it is generated, the box is on the server side. Since the handler is client code to access the text on the input box, we must use "~%". This is used in all the methods that require reading information from the server.

```
1 let accept word =
2     if StateVariables.getCy1Type() = StateVariables.getAutomatonType() then
3         acceptAutomaton word
4     else
5         acceptRegularExpression word
```

Listing 5.11: Accept function of the Controller module

The method *accept* verifies the system variable (Listing 5.11 line 2) *cyType* through the *StateVariable* functions *getCy1Type()* and *getAutomatonType()* to see if we are working with an automaton or a regular expression and then calls the corresponding method.

```
let acceptAutomaton word =
    let w = (StateVariables.returnAutomata())#stringAsList1 word in
        ignore ((StateVariables.returnAutomata())#accept3 w)
```

Listing 5.12: Automaton's accept in the Controller module

The function *acceptAutomaton* called by the *accept* function transforms the received string into a list of chars and calls the method *acceptAnimated* from the *FiniteAutomatonAnimation*. This function has "!" prepending the call since *automata* is a reference, and we need "!" to call one of its methods. "#" tells us that we are calling the method *acceptAnimated* of the automaton.

```
let rec delay n = if n = 0 then
  Lwt.return ()
else
  Lwt.bind (Lwt.pause()) (fun () → delay (n-1))
```

Listing 5.13: Delay function

```
1  method acceptAnimated (w: word) =
2    let transition sts sy t =
3      let nsts = Set.flatMap (fun st → nextStates st sy t) sts in
4      Set.union nsts (nextEpsilons nsts t) in
5
6    let rec accept2X sts w t exists =
7        match w with
8          [] → Lwt.bind (delay 100)
9            (fun () → Lwt.bind (
10             Lwt.return (self#paintStates (List.length w) sts exists) )
11             (fun () → Lwt.return ((Set.inter sts self#representation.acceptStates) ≠ Set.empty)))
12         |x::xs → Lwt.bind (delay 50)
13             (fun () →
14               Lwt.bind (Lwt.return (self#paintStates (List.length w) sts exists))
15               (fun () → let nextTrans = transition sts x t in
16                 if (Set.size nextTrans) = 0 then
17                   accept2X sts [] t false
18                 else
19                   accept2X (transition sts x t) xs t true
20             )) in
21
22     let i = closeEmpty (Set.make [self#representation.initialState]) self#representation.transitions in
23       accept2X i w self#representation.transitions true
```

Listing 5.14: Main method to animate the automaton

The basis of the method *acceptAnimated* (Listing 5.14) was inherited from the Library OCaml-FLAT but it had to be modified to, instead of returning only true or false, in each step paint some state of the automaton in a different color. The great difference between this method and the original is the recursive function *accept2X* presented from line 6 to 20, that is originally is used to decide if the word is accepted or not and now is used to paint the states.

This is what it does: starting on the current state (the beginning is always the initial state) it calls the function to paint the new state we arrive to by the given symbol (the next letter of the word). But simply calling the method to paint is not enough and the page has no time to change. To allow the graphical update events to modify the visualized automaton, there are two options: using the *OCaml Unix* library or using a *Lwt thread* library. Since this part of the code is on the client side, the first option turned out to be impracticable (the *Unix* library can only be used on the server side). For that reason, we use *Lwt thread* library combined with a delay function (Listing 5.13). What happens is that every time a set of transitions is found, a delay is executed. During this delay the pause function of the *Lwt* is used to handles pending events, including the ones that update the screen.

```
method paintStates length states alphExists =
    Graphics.resetStyle();
    Set.iter (fun el → self#paint el length (Set.belongs el self#representation.
    ↪ acceptStates) alphExists) states
```

Listing 5.15: Method that calls the painting function for each state

The method *paintStates* (Listing 5.15) is the one that is going to function while the pause is uphold. It resets the style of the graph, so the previous painted states go back to non colored and then for each of the new states it calls the method *paint* (Listing 5.16. This method makes the verification to see which color the state is going to be painted in. First it makes sure that the label is part of the alphabet, if not a message is sent to the user and the state is painted in red. If the transition exists then it verifies if the word has ended or if it is just a step state.

```
1  method paint state length final alphExists =
2            if alphExists then
3              (if (length != 0) then
4                Graphics.paintNode state stepState
5              else
6                  (if (final) then
7                    Graphics.paintNode state acceptState
8                  else
9                   Graphics.paintNode state wrongFinalState))
10           else
11             (Graphics.paintNode state wrongFinalState;
12             JS.alert ("There is no transition with the symbol given!"))
```

Listing 5.16: Method that decides which color the state is painted with

The *paintNode* (Listing 5.16 Line 4) is the method which calls the *JavaScript* function that changes the color of a state and is used in several other functionalities.

The step-by-step option is based in three button clicks: start, forward and backwards and since it reacts to *on_click* actions it does not have to use the *Lwt* library.

The methods to paint the steps *paintStates* (Listing 5.15) and *paint* (Listing 5.16) explained for the accept animation are reused in this functionality.

5.2.7.1 Starting the step-by-step

Users still need to input the word to be tested in the input box, but instead of testing the complete word they clicks "Start", this button handler reads the input box and transforms the string in an *OCaml* string. With the string, it calls the method *startStep* that initiates the step-by-step. In the end the word in the input box changes to show which part of the word has been tested (Listing 5.17 Line 5).

```
1 let stepbystep_handler = [%client (fun _ →
2         let i = (Eliom_content.Html.To_dom.of_input ~%inputBox) in
3         let v = Js_of_ocaml.Js.to_string i##.value in
4         Controller.startStep v;
5         i##.value:= Controller.getNewSentence()
6       )]
```

Listing 5.17: Button handler to start the step-by-step option

```
1 let startStep word =
2       (StateVariables.returnAutomata())#changeTheTestingSentence word;
3       ignore ((StateVariables.returnAutomata())#startAccept)
```

Listing 5.18: FiniteAutomatonAnimation method and function to paint the productive states

The function *startStep* (Listing 5.18) calls the method *changeTheTestingSentence* (Line 2) that transforms the word in a list of chars and saves it in a variable inside the class. Then it calls the method to initiate the accept (Line 3).

```
method startAccept =
    steps ← Array.make 1000 Set.empty;
    position ← 0;
    isOver ← false;
    let i = closeEmpty (Set.make [self#representation.initialState]) self#representation.transitions in
        Array.set steps position i;
    self#paintStates ((List.length !sentence) - position) (Array.get steps position) true;
    if (position = (List.length !sentence)) then
        (isOver ← true);
    self#changeSentence ()
```

Listing 5.19: FiniteAutomatonAnimation method to start the accept step-by-step functionality

The method *startAccept* (Listing 5.19) starts by initiating three mutable variables, *steps* is where the transitions are stored ordered in an array (Line 2), *position* is the variable that indicates in which step we are (Line 3) and *isOver* (Line 4) is the variable that indicates if the whole was processed or not. These *steps* and *position* are important for the step back and is *isOver* to the step forward, as we will see next. Afterwards, it calculates the next transitions (since we are working with any type of finite automaton it can be more than one) (Line 5), saves them in the step variable (Line 6) at the index defined by the variable position, which is 0 taking into account that we are still starting the automaton. Then to show the user the initiation of the accept (Line 7), it paints the initial state in the defined color (it follows the same rules as the normal accept and uses the same paint method) using the method *paintStates* (Listing 5.15). Finally it verifies if the word was empty and if so indicates that the automaton is finished (Line 9), this allows the user to receive an alert that the word is over if he clicks the button forward. Finally, the word is changed to show the bar before the word as explained in the beginning of the chapter (Line 10).

From this point on the user can go forward or backwards to visualize the evolution of the word in the automaton.

5.2.7.2 Going forward

When the user clicks the button forward represented by an arrow pointing to the right, the controller function calls automatically the method *next* of the automaton.

```
method next =
    if isOver then
        (JS.alert ("A palavra terminou. Não existem mais estados seguintes."))
    else
        (position ← position + 1;
        let letter = List.nth !sentence (position-1) in
        let nextSteps = (transition (Array.get steps (position-1)) letter self#
    ↪ representation.transitions) in
          steps.(position) ← nextSteps;
          if (Set.size nextSteps) = 0 then
            (self#paintStates ((List.length !sentence) - position) (Array.get
    ↪ steps (position-1)) false;
            isOver ← true)
          else
            (self#paintStates ((List.length !sentence) - position) (Array.get
    ↪ steps position) true;
            if (position = (List.length !sentence)) then
              (isOver ← true;)
            );
        self#changeSentence ())
```

Listing 5.20: FiniteAutomatonAnimation method to go forward on the accept functionality

The method *next* (Listing 5.20) starts by verifying if the word is finished (using the variable *isOver*), if so there is no path forward and the user receives a warning that the word is finished. If not, the algorithm starts. It sets the position variable one step forward (plus one), calculates the next states (Line 7) saves them in the *steps* variable (Line 8) and verifies if the next steps are empty (Line 9). If the next steps are empty, it means that there is not a transition with the given symbol, meaning that there are no paths possible and the word is not accepted. This is the information given to the paint states method - the state the user is on and the false that represents the non-existence of the symbol (Line 10) - and then the variable *isOver* (Line 11) is changed to indicate that the word is over because if there was no path from that symbol the remaining of the word could not accepted. If there are transitions it calls the method to paint the new states and verifies if the word has been fully read, and if so, it sets the variable *isOver* to true (Lines 13 to 16).

This method not only finds and paints the next states but also prepares the variables so that, if the user wants to go back, he only has to access the *steps* variable and not calculate the path backwards.

5.2.7.3 Going backwards

If the user chooses to go back, the corresponding controller calls the automaton method *back*.

```
    method back =
      position ← position - 1;
      if position < 0 then
        (position ← 0; JS.alert ("Não é possível andar para trás do estado inicial"))
      else
        (self#paintStates ((List.length !sentence) - position) (Array.get steps position) (Set.belongs (List.nth !sentence (position-1)) self#representation.alphabet);
        self#changeSentence();
        isOver ← false)
```

Listing 5.21: FiniteAutomatonAnimation method to go back on the accept functionality

This method sets the position one step back (minus 1) and then verifies if it became a negative number. If so, it means that we are already in the initial state and there is no more going back, so the method alerts the user for that fact. If not, then it paints the states in the referred position, changes the sentence and sets the variable *isOver* to false in case that in the previous step we had reached the end of the sentence.

5.2.8 Type of the states

Since the three options to see the type of the states are similar it is only going to be demonstrated the option that allows to see all productive states.

When the button is clicked the corresponding method in the Controller module is called in this case *paintAllProductive* (Listing 5.23). First the Graphics function *resetStyle*

```
1 let paintAllProductives () =
2     Graphics.resetStyle();
3     (StateVariables.returnAutomata())#productivePainting
```

Listing 5.22: Controller function to paint the productive states

(Line 2) is called to clean any style modifications that could have occurred to the automaton before. Then, we get the variable that represents the automaton to call the method *productivePainting* (Line 3).

```
1 let productiveColor = "orange"
2
3 let paintProductive state =
4         Graphics.paintNode state productiveColor
5
6 method productivePainting =
7     let list1 = Set.toList self#productive in
8         iterateList paintProductive list1
```

Listing 5.23: FiniteAutomatonAnimation method and function to paint the productive states

This an example of a non-hybrid function, we call the method *productive* from the OCaml-FLAT library to get the list of all productive states as seen in Listing 5.23 Line 7 and then for each state the Graphics function *paintNode* (Line 4) calls the JavaScript function to paint it. Actually, this is the same function called in the acceptance of the word. The greatest difference between these two options is that in here we want to paint more than one state at the same time. That is why in the acceptance the *resetStyle* is called each time and in this case it is only called in the Controller function. At each state, painting the style is updated and never erased.

An example of the three options can be seen in Figures 5.5, 5.6, 5.7.

5.2.9 Generate

To generate all the words until a given size the controller method simply creates the box on the right with its buttons (Listing 5.24 Lines 3 to 4) and then verifies which mechanism it is using. If the mechanism in use is an automaton, it calls the method *generateUtil* from the OCaml-FLAT library (Line 7) and with the result the function *putWords* is called (Line 10). *putWords* is the view function that displays the result, using *TyXML* it simply defines the elements that are going to be put in the box, creating a textarea and then inserting the result inside it.

```
let getWords v =
      StateVariables.changeCy2ToInfo();
      HtmlPageClient.twoBoxes();
      HtmlPageClient.putCy2Buttons();
      let var =
      if (StateVariables.getCy1Type() = StateVariables.getAutomatonType()) then
        (StateVariables.returnAutomata())#generateUntil v
      else
        (StateVariables.returnRe())#generate v in
      HtmlPageClient.putWords var
```

Listing 5.24: FiniteAutomatonAnimation method and function to paint the productive states

```
let twoBoxes () =
      clearBox2 ();
      let box1 = Dom_html.getElementById "Box1" in
        box1##.style##.width:= Js_of_ocaml.Js.string "49.5%";
      let box2 = Dom_html.getElementById "Box2" in
        box2##.style##.width:= Js_of_ocaml.Js.string "49.5%";
        StateVariables.halfSize();
        Graphics.fit()
```

Listing 5.25: HtmlPageClient function to create two boxes

5.2.10 Convert to deterministic

As explained in section 5.1.5, this was the functionality that brought the question: How to show two mechanisms at the same time? To do so, we did some research in the *Js_of_ocaml* API in order to find a way to change the style of the page. We found that after getting an element with the *Dom_html* module we could access the style and its elements as show in listing 5.25. This way we could change the page elements to allow the representation of two elements.

Having two boxes means having a new variable that represents the automaton drawn in the second box, this is the *automata1* variable and is represented in the module *StateVariable*.

So, what does it do? It simply calls the library algorithm that generates the deterministic automaton, it creates the representation and then represents it in the box on the right.

To draw the automaton in the box the *drawExamplel* function from the *FiniteAutomatonAnimation* module is called. This function works exactly like the *drawExample* explained in Chapter 4 but instead of calling *JavaScript* functions to draw into box one, it calls the one that draws on box two.

```
  let setColor number =
   if (number ≤ 20) then
    listColorsBig := listColors
   else
    (for i=0 to 19 do
     Array.set !listColorsBig i (Array.get listColors i)
    done;
    for i=20 to number-1 do
     let newColor = Graphics.getRandom() in
     Array.set !listColorsBig i newColor
    done)
```

Listing 5.26: Controller function to define which colors to be used in the minimization

```
1  let defineMinimized () =
2    if ((StateVariables.returnAutomata())#isDeterministic) then
3      if ((StateVariables.returnAutomata())#isMinimized) then
4        JS.alert ("O Autómato já é minimo")
5      else
6       (HtmlPageClient.twoBoxes();
7       StateVariables.changeCy2ToAutomaton();
8       HtmlPageClient.clearBox2();
9       HtmlPageClient.putCy2Buttons ();
10      Graphics.startGraph1();
11      Graphics.fit();
12      StateVariables.changeAutomata1 ((StateVariables.returnAutomata())#
   ↪ minimize1);
13      let number = (StateVariables.returnAutomata())#getNumberColors in
14      setColor number;
15      (StateVariables.returnAutomata())#paintMinimization !listColorsBig
16      (StateVariables.returnAutomata1())#drawMinimize !listColorsBig number)
   ↪
17   else
18     JS.alert ("O Autómato tem de ser determinista para poder ser minimizado
   ↪ ")
```

Listing 5.27: Controller function to minimize the automaton

5.2.11 Minimize

For the two automata to be able to paint the cluster and corresponding state in the same color, we defined a list of 20 contrasting colors that we give to each method. If there is more than 20 clusters there is a function that generates the number of extra colors necessary (Listing 5.26). This function is called after the minimized automaton is created (Listing 5.27 Line 13), because we need the minimization to know the number of clusters.

The main reason for the list of colors and the new ones generated in the controller is that the list given to each automaton method must be exactly the same.

Since the automaton has to be deterministic to be minimized this is the first verification that the minimization function does (Listing 5.27 Line 2). If the automaton is ready to be minimized the function goes through the process of representing the minimization.

After the page is prepared (Lines 6 to 11) the representation of the minimized automaton is created on the respective system variable (line 12), the number of clusters is defined to set the necessary colors (lines 13 and 14), the list of colors is then used to paint the states on the original automaton (line 15) and to create an already painted automaton in the right box (line 16).

The methods *paintMinimization* and *drawMinimized* are from the *FiniteAutomatonAnimation* module and they are used to change the appearance of the automaton. The function *paintMinimization* is used to change the original automaton, for each cluster it gets a color and calls the Graphics function to paint the states (same function used in the accept and in the type of states). The method *drawMinimized* works in a way that is similar to the *drawExample1* but instead of only calling the function to create the nodes, it calls one that creates and paints each one in the respective color. Then, it inputs the transitions in the same way.

5.2.12 Clean useless states

```
 1  let cleanUseless () =
 2        if ((StateVariables.returnAutomata())#areAllStatesUseful) then
 3          JS.alert ("O Autómato não tem estados para limpar, não existem estados
      ↪ inuteis!")
 4        else
 5          (HtmlPageClient.twoBoxes();
 6          HtmlPageClient.putCy2Buttons ();
 7          StateVariables.changeCy2ToAutomaton ();
 8          Graphics.startGraph1();
 9          StateVariables.changeAutomata1 (
10              (StateVariables.returnAutomata())#cleanUselessStates1);
11          (StateVariables.returnAutomata1())#drawExample1;
12          defineInformationBoxAutomaton ())
```

Listing 5.28: Controller function to clean the useless states

```
method cleanUselessStates1: FiniteAutomatonAnimation.model =
    Graphics.resetStyle();
    let uss = self#getUselessStates in
    Set.iter (fun el → paintUseful el) uss;
        let useless = super#cleanUselessStates in
        let rep = useless#representation in
        new FiniteAutomatonAnimation.model (Representation rep)
```

Listing 5.29: FiniteAutomatonAnimation method to clean the useless states

The controller method (Listing 5.28) follows the same scheme as seen before. First it asks the model if there is anything to erase, if not it alerts the user (Lines 2 and 3). If there is states to erase, it divides the page into two boxes and puts the buttons on the page (Line 5 and 6), changes the state of the page (line 7), initiates the graph by calling the *JavaScript* methods (Line 8) and then while it defines the state variable with the new automaton it paints the states in the already represented one (Line 10) with method *cleanUselessStates1* (that can be seen in Listing 5.29). To draw the automaton in the box, we use the function *drawExamplel* that is similar to *drawExample* explained in 5.2.3.

5.2.13 Convert

```
1 let automatonToRegExp() =
2     if StateVariables.getCy1Type() = StateVariables.getRegexType() then
3         JS.alert ("Já está a trabalhar com uma expressão regular")
4     else
5     (let reg = (StateVariables.returnAutomata())#toRegularExpression1 in
6     let rep = reg#representation in
7     let re = new RegularExpressionAnimation.model (Representation (rep)) in
8           defineRegularExpression1 re)
```

Listing 5.30: Controller functions to make the conversion from Automaton to Regular Expression

This functionality follows a lot of the logic already explained. When a user chooses the option, a method from the Controller is called (Listing 5.30), it verifies which mechanism the user is working with and gives an alert if the user is trying to convert it to the same type of mechanism (Lines 2 and 3), if not, it calls the respective FiniteAutomatonAnimation method *toRegularExpression1* that generates the Regular Expression (Line 5), gets its representation (Line 6) and transforms it into a *RegularExpressionAnimation* (Line 7) and then calls the Controller function *defineRegularExpression1* (Line 8) to draw it on the screen.

Why making the transformation into *RegularExpressionAnimation*? Since *FiniteAutomatonAnimation RegularExpressionAnimation* modules are not mutually recursive the module *FiniteAutomatonAnimation* only has access to the model functionalities and modules in the OFlat file, which means that it can only create a *RegularExpression*. If at some point there is the necessity for this regular expression to access the methods for the animation it has to be in the *RegularExpressionAnimation* format, this way this is already taken care of.

```
1  method toRegularExpression1 =
2         let reg = self#toRegularExpression in
3           reg#simplify
```

Listing 5.31: *FiniteAutomatonAnimation* method to convert into Regular Expression

The method *toRegularExpression1* (Listing 5.31) simply calls the model method that returns the necessary regular expression (line 2) and returns it simplified (achieve with a call to the model) (line 3).

In this chapter we explained the Automata mechanism and its functionalities. Some of them are common to the Regular Expression whilst implemented in a different way, as we are about to see in the next chapter.

Chapter 6

Regular Expressions

In this Chapter we explain the functionalities related to the Regular Expressions. Regular Expressions can be imported from the filesystem or the server, or else manually entered in the input box. After the creation of the regular expression there are several actions that can be carried out: verify if it matches a particular word; generate all the matched words up to a given maximum length; and convert it to a Finite Automata.

"Match" is the strictly correct verb to convey the meaning that a word belongs to the language represented by a regular expression. However, in this document, we tend to use the verb "accept" with broad meaning across all FLAT models. So, in the case of regular expressions, on many occasions we will use "accept" instead of "match".

Following the formatting of the previous chapters, in Section 6.1 we start by explaining how the Regular Expression is displayed on the screen and then how each functionality related to the Regular Expression is shown and why.

After the graphical representations are demonstrated we present in Section 6.2 an explanation of how the code works for each functionality.

6.1 Functionalities and its presentation

6.1.1 Regular Expression Representation

When we were developing the functionalities related to this mechanism, one of the biggest questions that emerged was how to represent the regular expression in a way that helped the user to understand it. In general, it is only presented as a string. We decided that it should be displayed both by the string and by the syntax tree that represents it. The tree emphasizes the regular expression structure and helps the user to understand it, an example of which can be seen in Figure 6.1. As shown in the figure, by default, the tree is

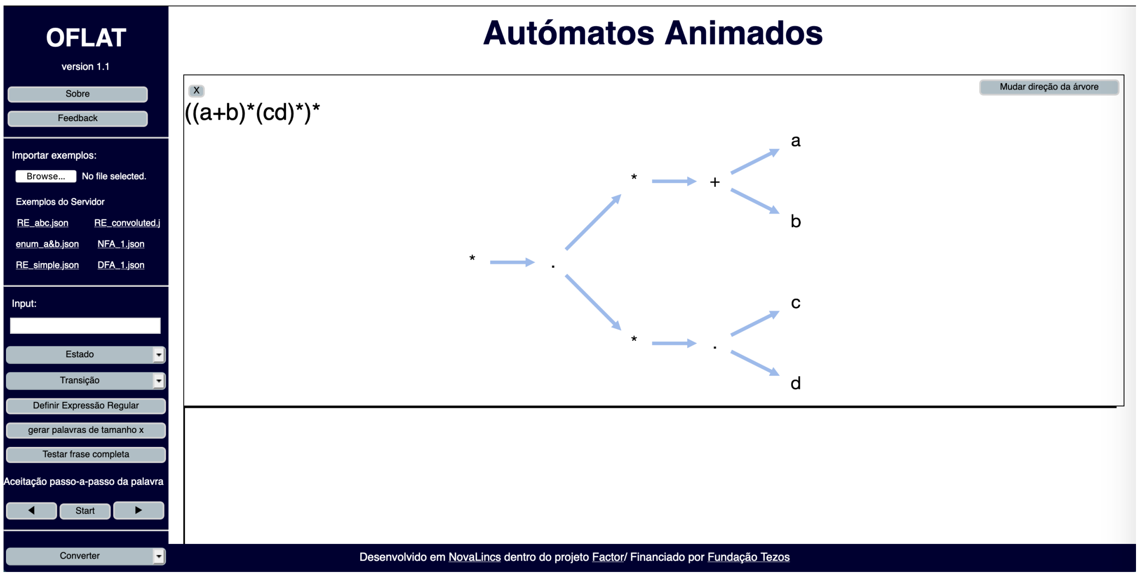
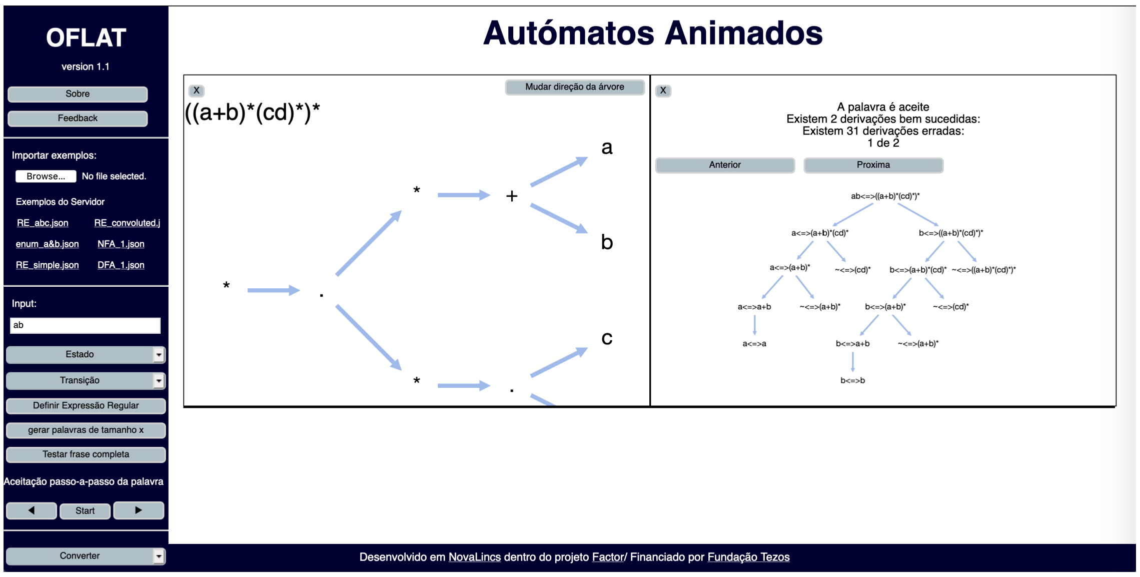


Figure 6.1: Representation of a Regular Expression

presented horizontally to make the best use of the space, however, there is a button in the upper right corner that allows the user to turn the tree clockwise and place it vertically.

6.1.2 Accept

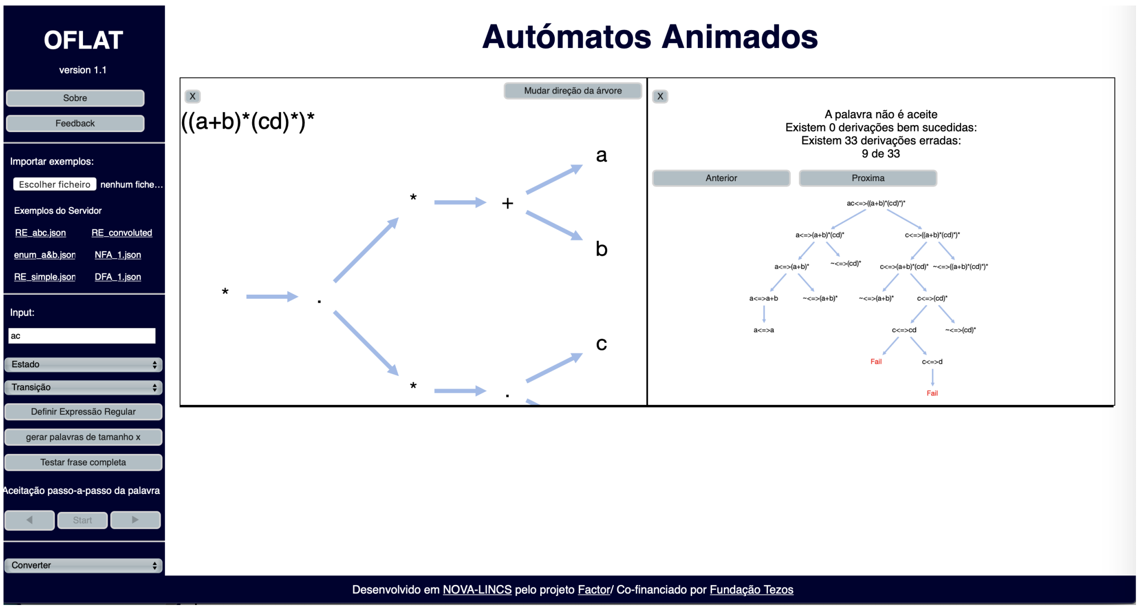


A palavra é aceite
Existem 2 derivações bem sucedidas:
Existem 31 derivações erradas:
1 de 2

Figure 6.2: Example of an accepted word

The acceptance of a word by a Regular Expression is a functionality not often seen in applications but a very useful one for students since they must apply it in class.

Given that a Regular Expression, even represented as a syntax tree, is not a mechanism that can be as easily animated as the automaton, there is not a visible sequence of steps. To bypass this problem, it was decided that we would use a derivation tree to show the



A palavra não é aceite
Existem 0 derivações bem sucedidas:
Existem 33 derivações erradas:
9 de 33

Figure 6.3: Example of a non accepted word

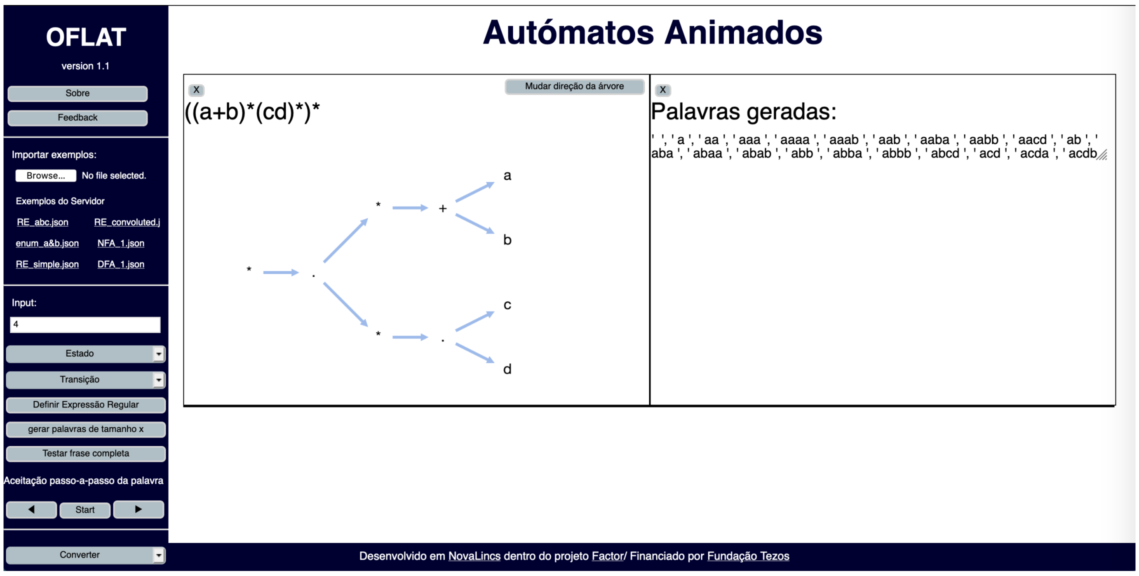
acceptance of the word. Each tree shows the several steps taken to break the word into different parts that try to match each part of the regular expression.

How does it work? The user must insert the word to test in the input box and then by clicking the button "testar frase completa" the word is evaluated. This means that the system is going to test if the word matches the regular expression. It is important to refer that some of the rules that define the match operations are non-deterministic. This means that different combinations of the rules allows the system to find matches and non-matches in different ways.

To show the answer, the box on the right is opened and in it is inserted first the result (if the word is accepted or not), how many ways the word could be matched by the regular expression and how many ways it could not. And finally it represents all the syntax trees as a list, each syntax tree shows the steps made to match the word to the regular expression showing which part of the word is matched to which part of the regular expression. Those trees can be navigated using the buttons inserted in the box. If the word is accepted, the mechanism only shows the trees with the successful word derivations (Figure 6.2) and if not, it shows all the unsuccessful trees (Figure 6.3). In the latter case the trees show a Fail in red for each time the matching fails. It is important to refer that in this system "~" represents the empty word.

6.1.3 Generate

The generate functionality works the same way has the one presented in Chapter 5. The user inputs a number in the input box and, after clicking the generate button, the system generates all the words that matches the regular expression until a given length.



Palavras geradas:

```
'  ', ' a ', ' aa ', ' aaa ', ' aaaa ', ' aaab ', ' aab ', ' aaba ', ' aabb ', ' aacd ', ' ab ', '
aba ', ' abaa ', ' abab ', ' abb ', ' abba ', ' abbb ', ' abcd ', ' acd ', ' acda ', ' acdb
```

Figure 6.4: Generating accepted words with maximum size 4

To keep the logic of the application the result is presented as a list in the box on the right as is happened with the automaton. An example can be seen in Figure 6.4.

6.1.4 Convert

The last functionality available for the Regular Expression mechanism is the convert functionally. Since the application currently supports only two mechanisms, Finite Automata and Regular Expressions, the conversion that is possible at this point is to Finite Automaton. The method used in the OCaml-FLAT library to convert the regular expression into an equivalent NFA is the a variant of the Thompson's construction algorithm [3].

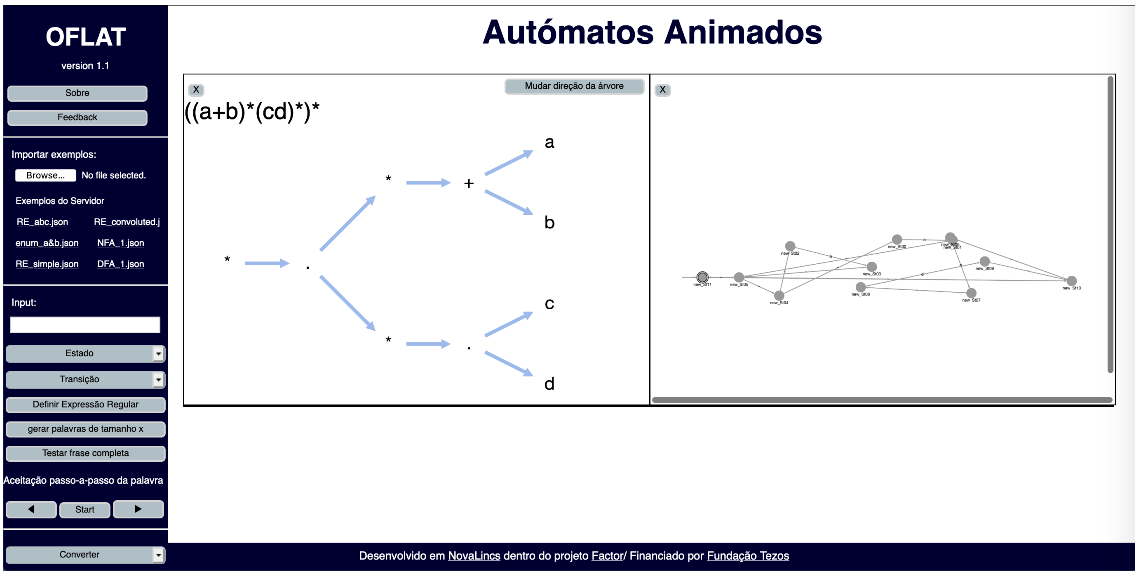


Figure 6.5: Conversion of a Regular Expression to a Finite Automaton

Following the logic of the functionalities that generate a new mechanism, the result of the conversion (an automaton) is generated in the box on the right (Figure 6.5). This allows the user to analyze and compare the two mechanisms and understand the conversion from RE to NFA.

For now, this functionality is not yet animated or step-by-step. The reason for this is that there was not sufficient time to find a good way to animate the algorithm and make it work correctly so we made the decision to postpone it and include in the future work.

6.2 Implementation

Now that each of the functionalities is explained we can demonstrate how they were implemented and how the code works. To understand the functionalities we start by explaining how the Regular Expression is represented in the code, and then go forward by explaining what is necessary to put in the file representing a Regular Expression if we want to import a RE, how a RE is created and then the action that can be achieved with it.

6.2.1 Definition of the Regular Expression

In terms of *OCaml* the Regular Expression represented on the screen is saved in the *RE* variable stored on the *StateVariables* module. This variable is a handler for the *RegularExpressionAnimation* module and is defined by a type specified in the library. The user is oblivious to this definition and only has to define the regular expression as a string unlike the automaton definition. The type defines the regular expression through its elements: star, plus, sequence, etc.

6.2.2 Load file

As stated before, much like the Automaton, the Regular Expression can be imported from the server or from a local filesystem. The information on the file is simpler than the one for the Automata as seen in Listing 6.2. The first three fields have the same objectives as the ones for the Automaton and the field *re* is simply the string that represents the regular expression.

```
{
  kind : "regular expression",
  description : "this is an example",
  name : "abc",
  re : "((a+b)*(cd)*)*"
}
```

Listing 6.1: Example of a file with the representation of a regular expression

6.2.3 Creation

The three types of Regular expression creation, from server, filesystem or by inputting it on the input box, make use of the same methods to create and display the Regular Expression. When there is a user action it means that the controller is called. The function from the controller that starts the drawing of the Regular Expression is *defineRegularExpression*.

```
 1 let rec func (re: RegExpSyntax.t) =
 2     match re with
 3         | Plus (l, r) → "+"^ func l ^ func r
 4         | Seq (l, r) → "."^ func l ^ func r
 5         | Star (re) → "*"^ func re
 6         | Symb (b) → String.make 1 b
 7         | Empty → "E"
 8         | Zero → "Z"
 9 
10 let defineRegularExpression example =
11     if StateVariables.getCy2Type() ≠ StateVariables.getEnumerationType() then
12         (HtmlPageClient.oneBox());
13         StateVariables.changeCy1ToRegex ();
14         StateVariables.changeRe example;
15         Graphics.destroyGraph();
16         HtmlPageClient.putCyREButtons();
17         let test = RegExpSyntax.toString (StateVariables.returnRe())#
    ↪ representation in
18             HtmlPageClient.defineRE test;
19         let test1 = func ((StateVariables.returnRe())#representation) in
20         Graphics.startGraph2(test1)
```

Listing 6.2: Functions to create the Regular Expression

What does the function do? First, it prepares the page to accommodate the Regular Expression (Listing 6.2 lines 12 to 16). Then using *Js_of_ocaml* the string representing the regular expression is written on the screen through the calling of the function *defineRE* in the HtmlPageClient module (Line 18). Finally, the tree is created (Line 20). This graphical creation differs greatly from the automaton for a reason. In the automaton, since we have the step-by-step creation those methods where reused when creating the complete automaton. In this case it made no sense to have a step-by-step creation. What happens is this: from the *OCaml* tree representation we create a pre-order traversal of the tree, codified in a simple string. Then that string is parsed in a JavaScript method to create a list of nodes and a list of edges that represent the RE. The function used to create the string can be seen in Listing 6.2 lines 1 to 8, this is a recursive function that reads the outermost element of the regular expression setting it as an element of the string and recursively doing the same with its components until the RE is decomposed.

As an example, we have the RE represented in Figure 6.1, "(a+b)*(cd)*)*" that is passed to the *JavaScript* function as "*.*+ab*.cd". This means that the first node is "*" (star) and its son, (since star only has one argument) is "." (the concatenation). The concatenation in its turn has two sons "+" (plus), followed by its own sons "a"and "b", and "*" (star) and so on until the whole RE is redefined.

The *JavaScript* function parses the string recursively to create node and edges, as can be seen in Listing 6.3. This function with the used of a switch (Lines 6 to 16) reads the first element of the string, creates its node and through a call to the same function the same is done to its sons, using their creation to define the edges between them. With the list of nodes and the list of edges, the graph can be created automatically with its elements.

```
 1  function makeTree1 (s) {
 2    var idgen = "n"+ number;
 3    number ++;
 4    var str = s;
 5    var st = str[0];
 6    switch (st) {
 7      case 'E': return [idgen, [{data: {id: idgen, name: "()"}}], [], str.substr(1)];
 8      case '+': var [lid, lnode, ledge, lret] = makeTree1 (str.substr(1));
 9                var [rid, rnode, redge, rret] = makeTree1 (lret);
10                return [idgen, [{data: {id: idgen, name: "+"}}].concat(lnode).concat(rnode),
                      ↪ [{data: {source: idgen, target: lid}}].concat([{data: {source: idgen,
                      ↪ target: rid}}]).concat(ledge).concat(redge), rret];
11      case '*': var [cid, cnode, cedge, cret] = makeTree1 (str.substr(1));
12                return [idgen, [{data: {id: idgen, name: "*"}}].concat(cnode), [{data: {
                      ↪ source: idgen, target: cid}}].concat(cedge), cret];
13      case '.': var [lid, lnode, ledge, lret] = makeTree1 (str.substr(1));
14                var [rid, rnode, redge, rret] = makeTree1 (lret);
15                return [idgen, [{data: {id: idgen, name: "."}}].concat(lnode).concat(rnode),
                      ↪ [{data: {source: idgen, target: lid}}].concat([{data: {source: idgen,
                      ↪ target: rid}}]).concat(ledge).concat(redge), rret];
16      default: return [idgen, [{data: {id: idgen, name: st}}], [], str.substr(1)];
17    }
18  }
```

Listing 6.3: Method to parse the string to create the syntax tree

6.2.4 Accept

```
type reTree =
   Fail
   | Tree of word * RegExpSyntax.t * reTree list
```

Listing 6.4: Type created to do the list of trees

The accept functionality for the regular expressions is one of the most complex ones since it needs to generate several syntax trees. To create the trees, we have defined a type tree in *OCaml* (Listing 6.4). The tree can be a Fail which means that the element of the

```
method getTrees w =
    let partition w =
      let rec partX w pword =
      match w with
        [] → Set.empty
            | x::xs → let fwp = pword@[x] in
            Set.add (fwp, xs) (partX xs fwp) in
            Set.add ([],w) (partX w []) in

  let rec acc w rep =
    match rep with
      | Plus(l, r) →
                let l1 = acc w l in
                let r1 = acc w r in
                 List.map (fun t → Tree (w, rep, [t])) (l1 @ r1)
      | Seq(l, r) → let wps =  partition w in
                let wpl = Set.toList wps in
                List.flatten (List.map (fun (wp1, wp2) →
                    let tl = acc wp1 l in
                    let tr = acc wp2 r in
                    List.flatten (List.map (fun x → List.map (fun y → Tree (w,
    ↪ rep, [x; y])) tr) tl)
                      ) wpl)
      | Star(re) → if w = [] then
                    [Tree (['~'], rep, [])]
                else
            (let wps = Set.remove ([],w) (partition w) in
                    let wpl = Set.toList wps in
                    List.flatten (List.map (fun (wp1, wp2) →
                        let tl = acc wp1 re in
                        let tr = acc wp2 (Star re) in
                        List.flatten (List.map (fun x → List.map (fun y → Tree
    ↪ (w, rep, [x; y])) tr) tl)
                      ) wpl))
      | Symb(c) →
                if w = [c] then
                    [Tree (w, rep, [])]
                else
                    [Tree (w, rep, [Fail])]
      | Empty →
                if w = [] then
                    [Tree (['~'], rep, [])]
                else
                    [Tree (w, rep, [Fail])]
      | Zero → [Tree (w, rep, [Fail])]
    in
      acc w self#representation

method startAllTrees w =
    allTrees ← self#getTrees w;
    position ← 0
```

Listing 6.5: Methods to generate all the derivation trees

word was not matched, or a triple composed by a word, a RE and a list with the remaining elements of the tree.

The controller method starts by calling the *RegularExpressionAnimation* method *startAllTrees* (Listing 6.5 Line 47), which through the method *getTrees* restarts the list that contains all the generated trees and restarts the variable that indicates the position in the list of the tree represented in the screen.

The method *getTrees* (Listing 6.5 Line 1) is a replica of the Ocaml-FLAT library method called *accept* that recursively tries to match the word to the RE and in the end indicates if the word was accepted (through a boolean). In this case the method was modified to save all the steps taken and make a list of all the possible trees. Through a recursive function (Line 10) the word is recursively partitioned and matched to an element of the RE. To test all the possibilities of matching the word to the RE the word is partitioned in three ways "~ (Empty word) + word", "first element of the word + rest of the word" and "word + ~" this is done with the *partition* function (Line 2). The same happens for each sub-word until all the word is tested.

```
method getRightTrees =
    let rightTrees = List.filter (fun x → isNotFail x) allTrees in
        allTrees ← rightTrees;
        List.nth allTrees position
```

Listing 6.6: RegularExpressionAnimation method to define a list with all the accepted trees

```
method printTree t =
    match t with
        Fail → "Fail" ^ "|/"
        | Tree ([], re, []) → ""
        | Tree ([], re, x::xs) →
            (self#printTree x) ^ "" ^ (self#printTree (Tree ([], re, xs))) ^ ""
        | Tree (w, re, []) →
            let blah = String.concat "" (List.map (String.make 1) w) in
            let regex = RegExpSyntax.toString re in
            blah ^ "≤>" ^ regex ^ "|/"
        | Tree (w, re, x::xs) →
            let regex = RegExpSyntax.toString re in
            let blah = String.concat "" (List.map (String.make 1) w) in
            blah ^ "≤>" ^ regex ^ "|"  ^ (self#printTree x) ^ "" ^ (self#printTree (Tree ([], re, xs))) ^ "/"
```

Listing 6.7: Method to create the string that will result in the accept graphical tree

Once all the trees are generated the original method of the library is used to know if the word is accepted or not. If so the result accepted is written in the right box and the method in Listing 6.6 is called to regenerate the list with only the trees that give an accepted result and return the first one (position was set to 0). With the methods *position* and *length* we get the number of the tree that is represented in the page and the number of trees that are going to be represented, and then put this information in the box. At this point the buttons previous and next are also defined, and finally the graphical tree is finally generated. If the word is not accepted everything is done in the same way except for the list which is not separated with the method 6.6.

How is the tree generated? To generate the tree we apply the same logic used in the creation of the RE syntax tree. A string representing a pre-order traversal is created and passed to the *JavaScript* that breaks it into nodes and edges. But in the creation, we had only symbols and now we have sentences. A new method represented in Listing 6.7 was developed to create a string that allowed the *JavaScript* function to know were each node ended, we decided to use a "|" to indicate that the node is finished and the its son should be read and a "/" to indicate that there are no more sons at that level and a level should be raised in the tree. This means that the *JavaScript* function reads first all the left sub-trees and then when it reaches a "/" it starts from the bottom reading the right ones.

An example can be seen in Figure 6.2. Imagine that we are matching the word "ab" with the Regular Expression "((a+b)*(cd)*)*". To draw one of the accepted trees the *acceptRegularExpression* function would give to *JavaScript* the string:

ab<=>((a+b)(cd)*)*| a<=>(a+b)*(cd)*| a<=>(a+b)*| a<=>a+b| a<=>a|// ˜<=>(a+b)*|// ˜<=>(cd)*|// b<=>((a+b)*(cd)*)*| b<=>(a+b)*(cd)*| b<=>(a+b)*| b<=>a+b| b<=>b|// ˜<=>(a+b)*|//˜<=>(cd)*|//˜<=>((a+b)*(cd)*)*|///.*

"ab<=>((a+b)*(cd)*)*", the first node, indicates that we are matching "ab" to the complete expression. The next step is to break the word into "a" and "b" and match "a" with "(a+b)*(cd)*" and "b" with the complete RE. The sub-trees are separated with "/". The left side of the tree is "a<=>(a+b)*(cd)*| a<=>(a+b)*| a<=>a+b| a<=>a|// ˜<=>(a+b)*|// ˜<=>(cd)*|/" (Figure 6.6) and the right side the rest of the string (Figure 6.7).

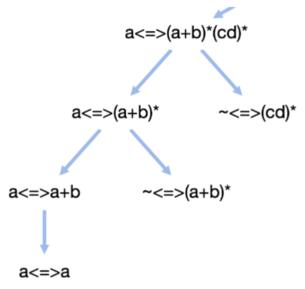


Figure 6.6: Left sub-tree that matches "a" with "(a+b)*(cd)*"

To develop the left sub-tree (Figure 6.6) the RE is broken into two parts "(a+b)*" and "(cd)*" where we match "a" with "(a+b)*" and "˜" to "(cd)*". To match "a<=>(a+b)*" (Figure 6.8) the law that indicates that "e* = ˜ + ee*" is used. Which means "(a+b)* = ˜ + (a+b)(a+b)*", since the union is ignored we obtain "a˜<=>(a+b)(a+b)*". We get "a<=>a+b" and "˜<=>(a+b)*". This two are also divide by a "/". The only thing missing is "a<=>a"

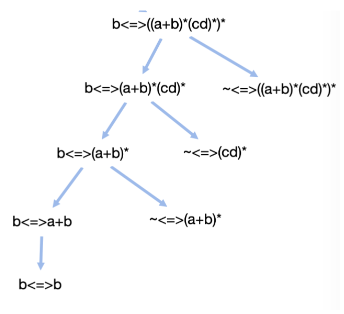


Figure 6.7: Right sub-tree that matches “b” with “((a+b)*(cd)*)*”

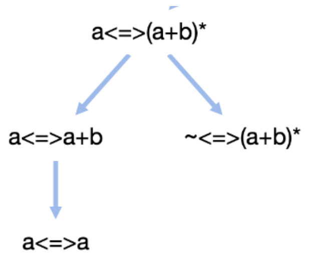


Figure 6.8: Left sub-tree that matches “a” with “(a+b)*”

that is in it self a sub-tree of “a<=>a+b” (and for that reason follows it in the string separated by “|”).

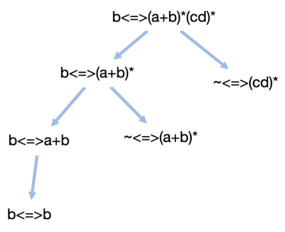


Figure 6.9: Sub-tree that matches “b” with “(a+b)*(cd)*”

The right sub-tree (Figure 6.7) to match “b<=>((a+b)*(cd)*)*” starts by using the law previously mentioned where the word is partitioned into “b” + “˜” and matched “b<=>(a+b)*(cd)*” (this generates another sub-tree that can be seen in Figure 6.9) and “˜” to “((a+b)*(cd)*)*”. For the new sub-tree the word is yet again partitioned and so on.

This string parsed by the *JavaScript* that returns the tree represented in the referred Figure 6.2 (in page 70).

An example of a tree of a word that fails to be match the RE can be seen in Figure 6.3. In it we are matching “ac” to the regular expression “((a+b)*(cd)*)*”. To show the represented tree the *acceptRegularExpression* function would give to *JavaScript* the string: *ac<=>((a+b)*(cd)*)*| a<=>(a+b)*(cd)*| a<=>(a+b)*| a<=>a+b| a<=>a|// ˜<=>(a+b)*|// <=>(cd)*|// c<=>((a+b)*(cd)*)*| c<=>(a+b)*(cd)*| ˜<=>(a+b)*|/ c<=>(cd)*| c<=>cd| Fail|/ c<=>d| Fail|/// ˜<=>(cd)*|/// ˜<=>((a+b)*(cd)*)*|///*

The first node indicates that we are matching “ac” to the complete regular expression, “((a+b)*(cd)*)*”. The next step is to break the word into “a” and “c” and match “a” with “(a+b)*(cd)*” (Figure 6.6) and “c” with the complete RE (Figure 6.10). The sub-trees are separated with “/”.

The left tree is the same as in the previous example (Figure 6.6).

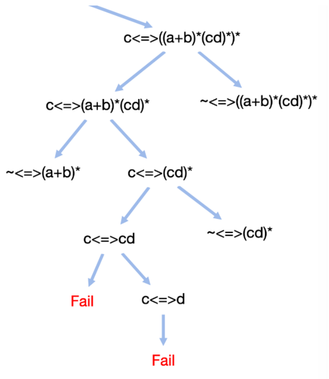


Figure 6.10: Right sub-tree that matches “c” with “((a+b)*(cd)*)*”

To try and match “c” the word is partitioned into “c” + “~” (Figure 6.10). Using the law previously mentioned, “c~” is matched to “(a+b)*(cd)* ((a+b)*(cd)*)*” (Figure 6.11).

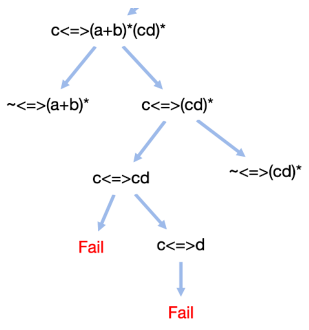


Figure 6.11: Sub-tree that matches “c” with “((a+b)*)(cd)*”

We keep partitioning “c”, first into “~” + “c” to match “(a+b)*” and “(cd)*” respectively and then as “c” + “~” to match “cd” and “(cd)*”. In the last sub-tree, “c<=>cd” (Figure 6.12) we partition the word into “~” + “c” and we match “~” to “c” which fails and “c” to d” that also fails. Since “cd” is a concatenation it was necessary to have the two symbols. For that reason even if we matched “c” to “c”, when we did “~<=>d” it would also fail.

This string parsed by the *JavaScript* that returns the tree represented in the referred Figure 6.3 (in page 71).

When clicking the button “next” the correspondent method is called, the variable *position* is increased and the correspondent tree in the list *allTrees* is return and then printed in the screen using the process explained before. The same happens when the button “previous”is clicked but the variable position is decreased. All the values of the box are updated in the same way they were in the first generation.

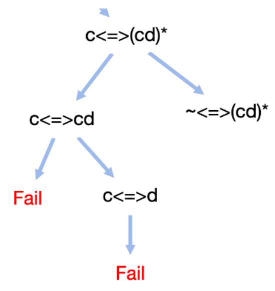


Figure 6.12: Sub-tree that matches “c” with “cd”

6.2.5 Generate

Since this functionality works exactly like the automaton one, the only difference is that the library method from that is accessed to retrieve the words is part of the Regular Expression module, it won’t be reexplained in this chapter.

6.2.6 Convert

```
1  let fromExpressionToAutomaton () =
2      if StateVariables.getCy1Type() = StateVariables.getAutomatonType() then
3          JS.alert ("Já está a trabalhar com um autómato finito")
4      else
5          (HtmlPageClient.twoBoxes ();
6          HtmlPageClient.clearBox2();
7          StateVariables.changeCy2ToAutomaton ();
8          HtmlPageClient.putCy2Buttons ();
9          Graphics.startGraph1();
10         Graphics.fit();
11         let auto = (StateVariables.returnRe())#toFiniteAutomaton   in
12         let maton = auto#representation in
13         StateVariables.changeAutomata1 (new FiniteAutomatonAnimation.model (
    ↪ Representation (maton)));
14             (StateVariables.returnAutomata1())#drawExample1)
```

Listing 6.8: Functions to generate an Automaton from the Regular Expression

The method used to convert the Regular expression to an Automaton can be seen in Listing 6.8. Following the logic of the other convert method, when *fromExpressionToAutomaton* is called it verifies if the mechanism represented is a regular expression (Lines 2 to 2). If not, it alerts the user. If so, it starts by preparing the two boxes (Lines 5 a 10), then it generates a new automaton by accessing the library method *toFiniteAutomaton* (Line 11) and afterwards transforms it in a *FiniteAutomatonAnimation* representation so that it can use the animation methods and stores it in the respective representation variable (Lines 12 to 13). Finally, it draws the automaton in the right box using the method *drawExample1* (Line 14) explained previously.

In this chapter we have explained the Regular Expression and its functionalities. At this point the two main mechanisms are explained, these two give the student some autonomy to work and experiment at his own pace but it is also important to be able to solve exercises. The next chapter presents the last mechanism that allows the user to solve simple exercises in which he is asked to define a language through an automaton or a regular expression.

CHAPTER 7

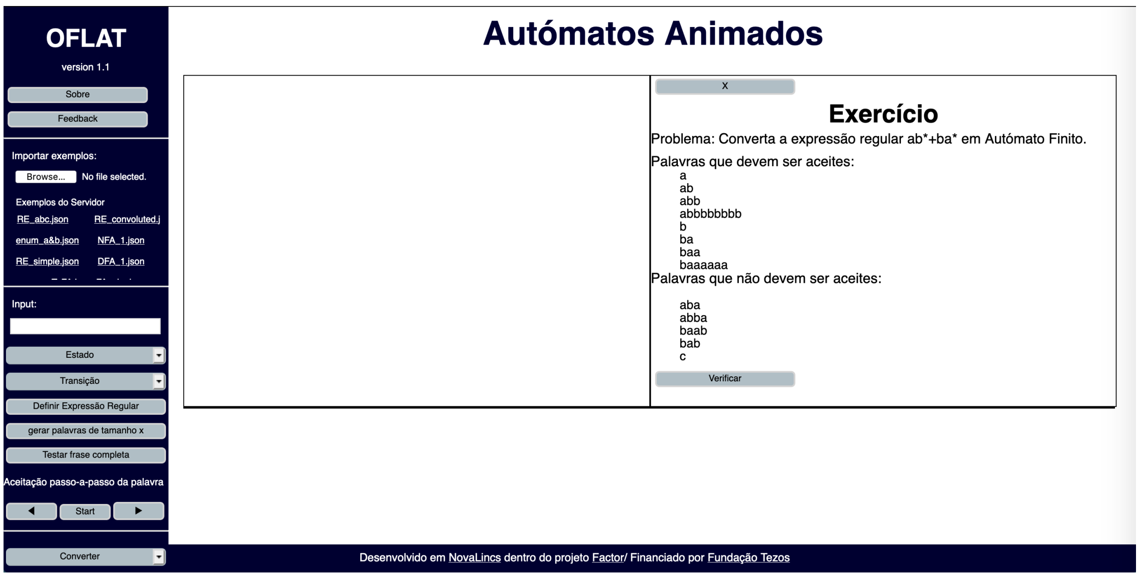
Exercises

In addition to working with Automata and Regular Expressions there is a possibility of solving exercises. In this chapter we explain what they are and how they work.

In Section 7.1 we start by explaining how Exercises are displayed, and why. We shed a light on how the user can answer the exercises and explain how the feedback is represented on the screen.

After the analysis of the graphical representation, in Section 7.2 we explain how this mechanism was implemented.

7.1 Functionality and its representation



Exercício

Problema: Converta a expressão regular ab*+ba* em Autómato Finito.

Palavras que devem ser aceites:
a
ab
abb
abbbbbbbb
b
ba
baa
baaaaaa

Palavras que não devem ser aceites:
aba
abba
baab
bab
c

Desenvolvido em NovaLincs dentro do projeto Factor/ Financiado por Fundação Tezos

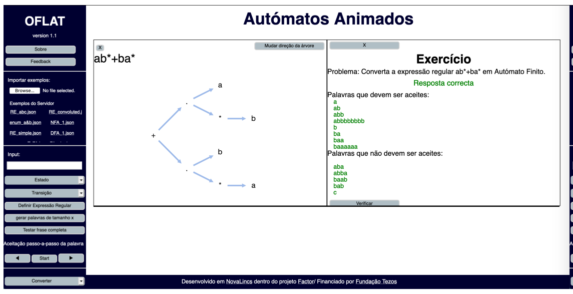
Figure 7.1: Representation of an Exercise

An “exercise” corresponds to the statement of a problem where the user is asked to define a language by means of a finite automaton or a regular expression. The statement is accompanied by a set of unit tests that are used to verify the solution. An example of an exercise will be given in Figure 7.1.

As can be seen in Figure 7.1 the exercise is presented to the user in the box on the right. This is the result of a technical decision, made for two reasons: the code was already prepared to always import automata and regular expressions for the box on the left and for that same reason, the user is already accustomed to work with the mechanisms on that box.

When the user imports an exercise, what is shown to him is the description of the problem and a list of words that should be accepted by the created mechanism and a list of words that should not. This can help the user not only understand which words are going to be used to test the solution, but also where the solution is failing (or not).

Once the exercise is loaded, in order to solve it, the user can import or generate an example as he would do if he was working with an Automaton or Regular Expression. When the exercise is ready to be checked, the user can click the button *verify*. Under the description of the problem appears the result (no mater if the solution is correct or not) painted in green if it is right and in red if it is not. The same happens with the set of unit tests, each word is painted in green if it fulfills their function, or in red if not. In this way the user may easily identify why the exercise is correct or not. Two examples of a verification of the exercises can be seen in Figure 7.2.



Exercício

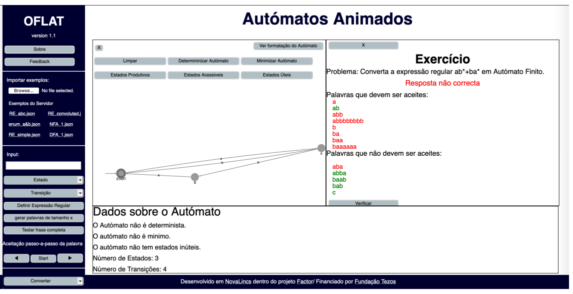
Problema: Converta a expressão regular ab*+ba* em Autómato Finito.

Resposta correcta

Palavras que devem ser aceites:
a
ab
abb
abbbbbbbb
b
ba
baa
baaaaaa

Palavras que não devem ser aceites:
aba
abba
baab
bab
c

(a) Example of an accepted resolution



Exercício

Problema: Converta a expressão regular ab*+ba* em Autómato Finito.

Resposta não correcta

Palavras que devem ser aceites:
a
ab
abb
abbbbbbbb
b
ba
baa
baaaaaa

Palavras que não devem ser aceites:
aba
abba
baab
bab
c

Dados sobre o Autómato

O Autómato não é determinista.
O autómato não é minimo.
O autómato não tem estados inúteis.
Número de Estados: 3
Número de Transições: 4

(b) Example of wrong resolution

Figure 7.2: Example of the correction of an Exercise

7.2 Implementation

Now that the reader understands how the exercises are represented and how they work we will explain how they are implemented.

First we explain how they are represented in *OCaml* and the information necessary on the file to be imported from the filesystem and then we define how this mechanism is implemented.

In terms of *OCaml* the exercise represented on the screen is saved in the *enum* variable stored on the *StateVariables* module. This variable works as a handler to the *Enumeration* module and its definition consists of:

- problem - a string that corresponds to the statement
- inside - a set with words to be accepted by the solution
- outside - a set with words that cannot be accepted by the solution

Much like the Automata and the Regular Expressions, exercises can be imported through the local filesystem as well as from the server, in JSon or txt and its format follows the ones from the previous mechanism as seen in Listing 7.1. In this case there is no other way to generate the exercises.

```
1 {
2   kind : "enumeration",
3   description : "this is an example",
4   name : "a&b",
5   problem : "Converta a expressão regular (a+b)*(c+d) em Autómato Finito.",
6   inside : ["abc","c","ab","b","abac"],
7   outside : ["","aba","bab","abba","baab","abcd"]
8 }
```

Listing 7.1: Example of a file with the representation of an exercise

The *kind*, *description* and *name* fields have the same functions as the ones in the Automata and Regular Expressions. The *problem* (Line 5) is the description of the exercise, the *inside* (Line6) is a list of words that must be accepted and the *outside* (Line 7) is a list of words that should not be accepted.

In the future we would like to add a new field to the definition that identifies which mechanism is supposed to be added as an answer. Hence, the user is driven to answer with the right mechanism.

How do the exercises work? The code seen in Listing 7.2 is the controller function defined to create the enumeration, it first defines the two system variables (Lines 2 and 3), one is used for the other mechanisms to know that they should maintain the two boxes on the screen (when we start to create an answer to the exercise, we want to keep seeing the problem), the other is used to store the exercise defined in the page and use it to evaluate the solution. After the variables are set, the box is created (Lines 4, and 5) and the information is placed on it through *HtmlPageClient* functions (Lines 6 to 17). These functions using *Tyxml* library create *HTML* elements with the necessary information on it and places those in the right place of the page.

One example of the *HtmlPageClient* functions is in Listing 7.3. This function puts the definition of the problem on the page. Using the *Js_of_ocaml* module *Dom_html* starts by getting the element where the problem is going to be put in on the page (Line 5), then

```
1  let defineEnum e =
2      StateVariables.changeEnum e;
3      StateVariables.changeCy2ToEnumeration ();
4      HtmlPageClient.twoBoxes ();
5      HtmlPageClient.putEnumButton ();
6      HtmlPageClient.addEnumTitle();
7      let prob = (StateVariables.returnEnum())#representation.problem in
8          HtmlPageClient.defineEnumProblem prob;
9      HtmlPageClient.addAcceptedTitle ();
10     Set.iter (fun el →
11         HtmlPageClient.createSpanList el "nothing" "inside")
12     (StateVariables.returnEnum())#representation.inside;
13     HtmlPageClient.addNonAcceptTitle ();
14     Set.iter (fun el →
15         HtmlPageClient.createSpanList el "nothing" "outside")
16     (StateVariables.returnEnum())#representation.outside;
17     HtmlPageClient.addEnumCheckButton ()
```

Listing 7.2: Controller method that organizes the Exercise rendering

```
1  let defineEnumProblem prob =
2      let textBox = Dom_html.getElementById "textBox" in
3        let test =  "Problema: " ^ prob in
4          let en =
5              Eliom_content.Html.To_dom.of_div (div ~a: [a_id "prob"][txt test])
6          in
7          Js_of_ocaml.Dom.appendChild textBox en;
8          let resultBox =
9               Eliom_content.Html.To_dom.of_div (div ~a: [a_id "resultBox"][txt
   ↪ ""])
10     Eliom_content.Html.To_dom.of_div (div ~a: [a_id "resultBox"][txt ""])
11         in
12         Js_of_ocaml.Dom.appendChild textBox resultBox
```

Listing 7.3: HtmlPageClient function to display the exercise on the screen

it creates the *HTML* element *div* (that represents a division on a page, a box) with the definition of the problem on it (Line 7.3). In the end with the function *appendChild* from *Js_of_ocaml* the new *HTML* element is added to the parent one (Line 7). The rest of the method is exactly the same and is used to create the box where the evaluation is going to be given, once the exercise is checked.

When the verification button is clicked the function *checkHelper* (Listing 7.4) is called with the solution proposed by the user (displayed on the left box). *checkHelper* first accesses the library to evaluate the answer (Line 2) and this is going to be a boolean result which is going to be used to define the information on the *HTML* box that contains the result (Line 3). Then it accesses the library to get the wrong results from the unit tests (Line 5). This is going to be a *tuple*, with two sets: the set of the ones that were not accepted and should have been and the set of the ones that where accepted and should not have been. Then each set is compared with the inside and outside list, respectively.

Each word is put on an *HTML* element (with the function *createSpanList*) identified with the *id* wrong or right (Lines 6 to 17). The same identifier is used for the box with the result. Why? Each identifier, right or wrong, corresponds to a *CSS* style that paints the text, in green or red, respectively.

```
1  let checkHelper isWhat =
2      let result = isWhat#checkEnumeration (StateVariables.returnEnum())  in
3      HtmlPageClient.defineResult result;
4      let (insideErrors, outsideErrors) =
5          isWhat#checkEnumerationFailures (StateVariables.returnEnum()) in
6      Set.iter (fun el →
7          if Set.belongs el insideErrors then
8              HtmlPageClient.createSpanList el "error" "inside"
9          else
10             HtmlPageClient.createSpanList el "right" "inside"
11     (StateVariables.returnEnum())#representation.inside;
12     Set.iter (fun el →
13         if Set.belongs el outsideErrors then
14             HtmlPageClient.createSpanList el "error" "outside"
15         else
16             HtmlPageClient.createSpanList el "right" "outside")
17     (StateVariables.returnEnum())#representation.outside
```

Listing 7.4: Method that organizes the presentation of the Exercise result

In this chapter the last available mechanism, Exercises, was explained. At this point we have already explained the context in which the application emerged, its design and organization and the mechanisms and functionalities already available to use. It is now time to explain how everything was tested and how we made sure it was ready to use.

Chapter 8 Testing

All along the development of this project, after each mechanism was finished each functionality was tested for correctness. In Section 8.1 we explain the testing process.

Our application is being used in the current edition of the Theory of Computation course at FCT/UNL and we decided to take advantage of this usage to make a large usability testing experiment. Prior to this, we also did an experimentation session with a group of more than ten people, including all the teachers of the said course. In Section 8.2 we explain the results found in that session, which was changed in the application afterwards and which is going to be change further in the future. Finally, we explain how we are going to benefit from the usage of the application in class.

8.1 Program Tests

The application is divided in two major components, the underlying library and the graphical implementation. Before being used as a base for the graphical implementation each mechanism presented in the library has undergone a big set of tests that are still available in the library code.

To test the web page we resorted to black-box testing in which we extensively tested each one of the mechanisms and functionalities with different examples. First, we have tested the creation of the mechanisms by verifying if the result represented on the screen matched to the downloaded ones or if the input had errors to which the user was alerted. The library's error system was helpful in the verification of the mechanism's creation by identifying the errors that occurred when downloading a file and in understanding if the file had errors or if the system was not reading it correctly.

After the creation of the mechanism was corrected while developing each functionality, different kinds of inputs were tested to make sure that the displayed results were the expected ones and that the user was alerted when necessary. Each functionality was tested with different types of examples, most of which may now be imported from the server, and for the ones that depend on input, the correct, incorrect and invalid ones were tested for each example.

The tests that were provided by the library were also used throughout the creation of the web component, not only to understand how to use the mechanisms and functionalities before animating them but also to compare results and understand where the graphical representation was failing.

The exercises are also useful to verify the *accept* functionality of the different mechanisms, since these returns the words that should be accepted and those that should not be, therefore we can test the *accept* functionality with the given words and compare the results.

```
1  (** Can be "finite automaton", "regular expression" or "clean" **)
2  let cyType = ref "clean"
3
4  let changeCy1ToAutomaton () =
5        cyType := "finite automaton"
6
7      let changeCy1ToRegex () =
8        cyType := "regular expression"
9
10     let cleanCy1Type () =
11       cyType := "clean"
12
13     let getCy1Type() =
14       !cyType
```

Listing 8.1: CyType variable and its respective functions

It is also important to refer that the system architecture was used to maintain a correct usage of the code and safeguard against any change of the system. The variables that represent the pages are stored in a module and they can only be changed with specific functions, to specific values. For example, the variable that represents that main box is called *cyType* (listing 8.1 line 2), and can only receive three values: finite automaton, regular expression and clean. To make sure no other value is given to this variable there are three functions to change to those three specific values (Lines 7 to 11), hence the controller never accesses the variable directly, but only through the setter functions. The same happens when reading the variable, as there is a function that returns the value of the variable (Line 13). This happens with all the other state variables.

8.2 User Tests

As previously explained to make sure that we had the most complete application possible for the use in class we have decided to keep developing the application and, for the time being, not spend too much time preparing a very complete user testing evaluation.

For the moment we did an experimentation session with a group of 16 people, some experts in the field and others not so much, but all from Computer Science. This session proved to be very helpful in terms of feedback such as usability suggestions. Besides, the session helped us to find two different types of problems: bugs in the system and usability problems.

The bugs in the system came with last minute changes in the code, since at the time the application was still on a finishing stage and all of those bugs where corrected:

- **Alert box in the Automata Accept** - there was an error in the accept method of the Automata that made an alert box appear in each step, even though the automaton was being correctly painted. The error was related with the verification of the alphabet.
- **Buttons missing** - in some functionalities, like the step-by-step creation of an automaton, the buttons were not appearing in the box.
- **Selection of the node** - when trying to select a node, there was a problem with click offset. Whenever the user clicked a position it was as if the user had clicked a different one. This arose from an incompatibility with the *cytoscape.js* library to draw a graph in the same division where other elements were represented. The problem was solved with the usage of different divisions.
- **Step by step accept buttons enabled for the Regular Expression** - these buttons were meant to be used with the automata. If used with the Regular Expression they would not create the desired effect and would change the format of the derivation tree, which was not the purpose. This problem was solved by the disabling the buttons when a regular expression is represented.

The usability problems were related to the fact that some of the usage of the page was not as easy as initially thought. Some users had difficulties finding the input box, and some of the buttons were not readable because of the text color. These two problems were solved by identifying the input box and changing the text color. As for the bigger problems are not yet completely solved:

- **When to input text before clicking the button** - some users did not understand when there was a need to input text before clicking the button. For now this problem has been addressed with an alert box that indicates the necessary information and format. For the future, we have planned to test a more guided use of the tool (based on one of the suggestions).
- **How to start working with the new generated automaton** - a few functionalities, as explained in the previous chapters, generate a new automaton in a new box. If the user clicks on the *close* button on the left box, the automaton generated on the

right becomes the main automaton. This mechanism not understood by all. A few solutions are still being evaluated to solve this problem.

At the end of the session we were able to understand which were the hardest parts in the usage of the application to create an instructions page in order to help guide the user if needed. There were also several ideas for new features that could facilitate the use and understanding of the page and mechanisms.

From the beginning of March until the end of the semester we plan to have the application go through a very broad testing system since the application is meant to be used in the Theory of Computation course. While writing of this dissertation, the application was presented and experimented in a theoretical class and more than 300 students were encouraged to experiment and use it as a learning mechanism throughout the semester and to share afterwards any doubts and suggestions through the feedback page of the application. This will help us to better understand where the greatest number of difficulties lie so that we can further improve the application.

We can say that through different types of tests and an experimentation session, we have made sure that the application was as better as it could be in order to be used in class, but we acknowledge that, upon its usage in class, we shall have a better understanding of what may be more suitable for a student who is dealing with this type of subjects for the first time.

Chapter 9

Conclusions and Future Work

This chapter presents the main conclusions and the future work that is intended for the application.

9.1 Conclusions

In this project we proposed to develop a Web application in Portuguese written in *OCaml* that allowing the students to study different topics of Formal Languages and Automata Theory. To do so, we started by understanding the *Ocsigen* Framework. Due to the difficulties arisen while studying this tool (as explained in Appendix A) the learning curve became steeper than expected but the obstacles have been overcome and we were able to take advantage of the tool's potential and create a web application already prepared to be used in class.

Although a *JavaScript* library was used to display the graphs, we have achieved the proof of concept by showing that it is possible to create a web page almost completely in *OCaml*. We were able to have a complete interaction with the support library - which by being written in *OCaml* facilitates the correction proofs and keeps the algorithms as similar as the ones given in class -, and to maintain all the core programming, like decisions and verifications, in the *OCaml* side of the code, using the *cytoscape.js* library only to show what in *OCaml* is decided to be shown.

Despite the fact that the application does not cover all the mechanisms taught in the course of Theory of Computation, we have built a version of the application which is finalized and ready to be used in class by doing a wide coverage of a few mechanisms, i.e. having the all the functionalities for each mechanism even if it meant not having all of them fully animated.

In the, end we obtained a robust, structured and extensible application, with a centralized client-server code, that is prepared to receive different types of mechanisms and that already allows for the study of Finite Automata and Regular Expression as well as enables the performing of simple exercises.

9.2 Future Work

Considering the broad objective of this application, even though we have paved the way to create a complete tool, there is still much work to be done. We intend to further develop the application and make it increasingly usable and complete.

Firstly, we would like to make some changes related to the interface and the suggestions presented at the testing session. We expect to test different ways of show the user that there is an input which is necessary and to better explain how to use the mechanism generated in the right box. We also intend to add a few functionalities that are not directly related to the algorithms given in class but were suggested and can facilitate the use of the page.

Secondly, we also aim to make an in-depth coverage of the functionalities already presented, trying to find the best way to show some of those, without departing too much from the algorithms given in class.

Lastly, there are still other mechanisms in Formal Languages and Automata Theory that we intend to add to the application, starting with Context Free Grammars and going further towards the Turing Machines.

9.3 Final Remark

Ultimately, we believe that the objectives we set ourselves for this thesis were fulfilled since 1/3 of the topics covered in the course of Computational Theory were developed and the system is prepared to be extended with many more.

We trust that a relevant contribution was given to the FACTOR project in the sense the tool is already useful and is prepared to be enhanced and completed.

Bibliography

[1] *10 Common Software Architectural Patterns in a nutshell*. https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013. Accessed: 2020-03-09.

[2] E. Adar. "GUESS: A language and interface for graph exploration." In: *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*. Vol. 2. Jan. 2006, pp. 791–800. doi: 10.1145/1124772.1124889.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. isbn: 0-201-10088-6.

[4] *Automata Tutor v2.0*. http://automatatutor.com/index. Accessed: 2019-02-11.

[5] *AutoMate*. https://idea.nguyen.vg/~leon/automata_experiments/index.html. Accessed: 2019-02-11.

[6] *Automaton Simulator*. http://automatonsimulator.com/. Accessed: 2019-02-11.

[7] *Awali*. http://vaucanson-project.org/Awali/index.html. Accessed: 2019-07-09.

[8] V. Balat. "Ocsigen: Typing Web Interaction with Objective Caml." In: *Proceedings of the ACM SIGPLAN 2006 Workshop on ML* (Sept. 2006), pp. 84–94. doi: 10.1145/1159876.1159889.

[9] V. Balat, J. Vouillon, and B. Yakobowski. "Experience Report: Ocsigen, a Web Programming Framework." In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP* 44 (Sept. 2009), pp. 311–316. doi: 10.1145/1596550.1596595.

[10] C. W. Brown and E. A. Hardisty. "RegeXeX: An Interactive System Providing Regular Expression Exercises." In: *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '07. Covington, Kentucky, USA: Association for Computing Machinery, 2007, 445–449. isbn: 1595933611. doi: 10.1145/1227310.1227462. url: https://doi.org/10.1145/1227310.1227462.

[11] P. Chakraborty, P C. Saxena, and C. Katti. "Fifty years of automata simulation: A review." In: *ACM Inroads* 2 (Dec. 2011). doi: 10.1145/2038876.2038893.

[12] T. Claveirole, S. Lombardy, S. O'Connor, L.-N. Pouchet, and J. Sakarovitch. "Inside Vaucanson." In: *Proceedings of Implementation and Application of Automata, 10th International Conference (CIAA)*. Ed. by Springer-Verlag. Vol. 3845. Lecture Notes in Computer Science Series. Sophia Antipolis, France, June 2005, pp. 117–128.

[13] R. W. Coffin, H. E. Goheen, and W. R. Stahl. "Simulation of a Turing Machine on a Digital Computer." In: *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*. AFIPS '63 (Fall). Las Vegas, Nevada: ACM, 1963, pp. 35–43. DOI: 10.1145/1463822.1463827. URL: http://doi.acm.org/10.1145/1463822.1463827.

[14] J. Cogliati, F. W. Goosey, M. T. Grinder, B. A. Pascoe, R. Ross, and C. J. Williams. "Realizing the Promise of Visualization in the Theory of Computing." In: *ACM Journal of Educational Resources in Computing* 5 (June 2005), pp. 1–17. DOI: 10.1145/1141904.1141909.

[15] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. "Links: Web Programming without Tiers." In: *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*. FMCO'06. Amsterdam, The Netherlands: Springer-Verlag, 2006, 266–296. ISBN: 3540747915.

[16] *Cytoscape.js*. http://js.cytoscape.org/. Accessed: 2019-06-04.

[17] L. D'antoni, D. Kini, R. Alur, S. Gulwani, M. Viswanathan, and B. Hartmann. "How Can Automatic Feedback Help Students Construct Automata?" In: *ACM Trans. Comput.-Hum. Interact.* 22.2 (2015), 9:1–9:24. ISSN: 1073-0516. DOI: 10.1145/2723163. URL: http://doi.acm.org/10.1145/2723163.

[18] L. D'Antoni, M. Weaver, A. Weinert, and R. Alur. "Automata Tutor and what we learned from building an online teaching tool." In: *Bulletin of the European Association for Computer Science* 117 (2015), pp. 143–160.

[19] A. Demaille, A. Duret-Lutz, S. Lombardy, and J. Sakarovitch. "Implementation Concepts in Vaucanson 2." In: *Proceedings of Implementation and Application of Automata, 18th International Conference (CIAA'13)*. Ed. by S. Konstantinidis. Vol. 7982. Lecture Notes in Computer Science. Halifax, NS, Canada: Springer, July 2013, pp. 122–133. ISBN: 978-3-642-39274-0. DOI: 10.1007/978-3-642-39274-0_12.

[20] *FAdo*. http://fado.dcc.fc.up.pt/. Accessed: 2020-03-09.

[21] *FSM simulator*. http://ivanzuzak.info/noam/webapps/fsm_simulator/. Accessed: 2019-02-11.

[22] *FSM2Regex*. http://ivanzuzak.info/noam/webapps/fsm2regex/. Accessed: 2019-02-14.

[23] *JFLAP*. http://www.jflap.org/. Accessed: 2019-02-11.

[24] *JFLAP History*. http://www.jflap.org/history.html. Accessed: 2019-02-17.

[25] *JFLAP History PowerPoint*. https://www2.cs.duke.edu/csed/jflapworkshop/sigcse06/WorkshopHistory.pdf. Accessed: 2019-02-17.

[26] J. D. U. John E. Hopcroft Rajeev Motwani. *Introduction to Automata Theory, Languages, and Computation, 2nd Edition*. Addison-Wesley, 2001.

[27] G. Krasner and S. Pope. "A cookbook for using the model - view controller user interface paradigm in Smalltalk - 80." In: *Journal of Object-oriented Programming - JOOP* 1 (Jan. 1998).

[28] S. Krug. *Don't Make Me Think: A Common Sense Approach to the Web (3nd Edition)*. USA: New Riders Publishing, 2014. ISBN: 0-321-96551-5.

[29] *Learn OCaml*. http://learn-ocaml.hackojo.org/. Accessed: 2019-07-15.

[30] H. R. Lewis and C. H. Papadimitriou. *Elements of the theory of computation, 2nd Edition*. Prentice Hall, 1998. ISBN: 978-0-13-262478-7.

[31] S. Lombardy, R. Poss, Y. Régis-Gianas, and J. Sakarovitch. "Introducing Vaucanson." In: *Proceedings of the 8th international conference on Implementation and application of automata*. Vol. 2759. June 2003, pp. 107–134. DOI: 10.1007/3-540-45089-0_10.

[32] T. M. White and T. Way. "jFAST: a java finite automata simulator." In: *ACM SIGCSE Bulletin* 38 (Mar. 2006), pp. 384–388. DOI: 10.1145/1124706.1121460.

[33] A. Merceron and K. Yacef. "Web-based learning tools: storing usage data makes a difference." In: *WBED'07 - Proceedings of the sixth conference on IASTED International Conference Web-Based Education*. Vol. 2. Mar. 2007, pp. 104–109.

[34] *Ocsigen - Multi-tier programming for Web and mobile apps*. https://ocsigen.org/home/intro.html. Accessed: 2019-02-11.

[35] W. C. Pierson and S. H. Rodger. "Web-based Animation of Data Structures Using JAWAA." In: *SIGCSE Bull.* 30.1 (Mar. 1998), pp. 267–271. ISSN: 0097-8418. DOI: 10.1145/274790.274310. URL: http://doi.acm.org/10.1145/274790.274310.

[36] N Pillay. "Learning difficulties experienced by students in a course on formal languages and automata theory." In: *SIGCSE Bulletin* 41 (Jan. 2009), pp. 48–52. DOI: 10.1145/1709424.1709444.

[37] G. Radanne, J. Vouillon, and V. Balat. "Eliom: A Core ML Language for Tierless Web Programming." In: *Asian Symposium on Programming Languages and Systems* (Nov. 2016), pp. 377–397. DOI: 10.1007/978-3-319-47958-3_20.

[38] A. Ravara. *Lecture notes in Computational Theory*. The notes are currently unavailable to the general public, only for students with authentication. 2019.

[39] D. Raymond and D. Wood. "Grail: A C++ Library for Automata and Expressions." In: *J. Symb. Comput.* 17.4 (Apr. 1994), pp. 341–350. ISSN: 0747-7171. DOI: 10.1006/jsco.1994.1023. URL: http://dx.doi.org/10.1006/jsco.1994.1023.

[40] *Regular Expressions Gym*. http://ivanzuzak.info/noam/webapps/regex_simplifier/. Accessed: 2019-02-14.

[41] S. Rodger. "An Interactive Lecture Approach to Teaching Computer Science." In: *ACM SIGCSE Bulletin* 27 (Mar. 1995). DOI: 10.1145/199691.199820.

[42] S. Rodger, E. Wiebe, K. Min Lee, C. Morgan, K. Omar, and J. Su. "Increasing engagement in automata theory with JFLAP." In: *SIGCSE'09 - Proceedings of the 40th ACM Technical Symposium on Computer Science Education*. Vol. 41. Mar. 2009, pp. 403–407. DOI: 10.1145/1508865.1509011.

[43] I. Sanders, C. Pilkington, and W. Van Staden. "Errors made by students when designing Finite Automata." In: *SACLA'15, Johannesburg, South Africa*. July 2015.

[44] M. Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997. ISBN: 978-0-534-94728-6.

[45] I. Sommerville. *Software Engineering*. 10th. London: Pearson Education, 2016. ISBN: 1-292-09613-6.

[46] J. T. Stasko. "Tango: A Framework and System for Algorithm Animation." In: *SIGCHI Bull.* 21.3 (Jan. 1990), pp. 59–60. ISSN: 0736-6906. DOI: 10.1145/379088.1046618. URL: http://doi.acm.org/10.1145/379088.1046618.

[47] A. Stoughton. "Experimenting with formal languages using forlan." In: *FDPE '08: Proceedings of the 2008 international workshop on Functional and declarative programming in education*. Jan. 2008. DOI: 10.1145/1411260.1411267.

[48] M. T. Grinder. "A preliminary empirical evaluation of the effectiveness of a finite state automaton animator." In: *ACM Sigcse Bulletin*. Vol. 35. Jan. 2003, pp. 157–161. DOI: 10.1145/611892.611958.

[49] A. R. V. M. T. Teixeira. "A cor enquanto elemento do projecto no design de produto." Master's thesis. Portugal: Faculdade de Belas Artes da Universidade de Lisboa, 2015. URL: http://hdl.handle.net/10451/22246.

[50] *The Links Programming Language*. https://links-lang.org/. Accessed: 2020-03-09.

[51] L. Vieira, M. Vieira, and N. Vieira. "Language emulator, a helpful toolkit in the learning process of computer theory." In: *ACM Sigcse Bulletin*. Vol. 36. Mar. 2004, pp. 135–139. DOI: 10.1145/971300.971348.

[52] *VisuAlgo*. https://visualgo.net/en. Accessed: 2019-02-11.

[53] J. Vouillon and V. Balat. "From Bytecode to JavaScript: the Js_of_ocaml Compiler." Anglais. In: *Software: Practice and Experience* (2013). DOI: 10.1002/spe.2187.

[54] S. Weinschenk. *100 Things Every Designer Needs to Know About People*. 1st. USA: New Riders Publishing, 2011. ISBN: 0321767535.

[55] M. Wermelinger and A. Dias. "A Prolog toolkit for formal languages and automata." In: *ACM SIGCSE Bulletin* 37 (Sept. 2005). DOI: 10.1145/1067445.1067536.

Appendix A

Problems in this version of Ocsigen

Throughout the Framework's learning process and the development of the first application example, some problems arose that were somehow overcome. Here is a summary of those:

- **Change in the fundamentals of *OCaml*** - the base language syntax has been changed, the syntax *camlp4* has been discontinued to make way for *ppx*. This change implied the alteration of most applications written in *OCaml*, as is the case of *Ocsigen*. Probably, due to this change, some of the few basic examples of learning the *framework* are now discontinued and with errors.

- **Moving from Ocsigen-Widgets to Ocsigen-Toolkit** - several of the examples proposed on the web page assume the use of the *Ocsigen-Widgets* library which has been discontinued and replaced by *Ocsigen-Toolkit*. The problem with this change is that *Ocsigen-Widgets* was already a very complete and developed library, which *Ocsigen-Toolkit* does not yet correspond to. Thus, it is not possible to finish the learning examples simply by replacing the library.

- **API is difficult to understand** - for a tool as complete as *Ocsigen*, the documentation is too synthetic and non-educational, which causes to the user who takes the first steps in using *Ocsigen*, the need to spend a lot of time understanding the tool. It has, indeed, a very complete API, but this one, for lack of explanations and examples, is not easy to understand.

- **Few examples** - for such a complete tool, it contains very few learning examples on which an inexperienced user can rely in order to learn how to work with the framework.

- **Difficulties in the use of Ocsigen-Start** - on the web page, the user is advised to use the Ocsigen-Start template as a starting point but, in fact, the use of *Ocsigen-Start* as a basis for learning is not easy. Despite being a very complete *template*, for a beginner programmer it is not obvious how to adapt and use it.