NOVA
NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

Miguel Santos Araújo
Licenciado em Engenharia Informática

Autómatos de Pilha em OCamlFLAT/OFLAT

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Novembro, 2021

NOVA
NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTAMENTO DE
INFORMÁTICA

Autómatos de Pilha em OCamlFLAT/OFLAT

Miguel Santos Araújo

Licenciado em Engenharia Informática

Orientador: Artur Miguel Dias,
Professor Auxiliar, Universidade NOVA de Lisboa

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade Nova de Lisboa
Novembro, 2021

Autómatos de Pilha em OCamlFLAT/OFLAT

Copyright © Miguel Santos Araújo, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.

A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Agradecimentos

Em primeiro lugar gostaria de agradecer aos meus pais que sempre estiveram presentes durante todo o meu percurso. Agradeço por todo o seu apoio, não só nos bons momentos, mas também nos mais difíceis, devo-lhes tudo. Obrigada por me terem dado as ferramentas necessárias para superar os obstáculos que foram surgindo no meu percurso de vida.

Um agradecimento ao resto da minha família, que mesmo estando longe ocupam um lugar especial na vida. Gostaria, também, de dar um agradecimento especial à Inês. Desde que a conheci que sempre acreditou em mim. Esteve presente durante o que me atreveria a dizer ser o período que mais me irá moldar como pessoa, e que mais relevo terá a nível profissional. Nunca desistiu de mim mesmo durante alturas mais difíceis. Tentou sempre fazer com que não desanimasse para não cair da tentação de desistir do meu desenvolvimento pessoal.

Queria também agradecer imenso por todo o esforço e dedicação que me foram prestados pelo meu orientador, o professor Artur Miguel Dias. Agradeço do fundo do coração a atenção, confiança e disponibilidade. Não existem palavras que descrevam a gratidão que sinto por nunca ter desistido de mim. Obrigada por todas as reuniões fora de horas e toda a paciência que teve comigo. Votos de que tenha ainda mais sucessos na sua carreira profissional.

Um carinho especial também ao Eduardo e a Rita por me terem sempre ajudado com opiniões e dicas fundamentais para o desenvolvimento do projeto.

Gostaria também de agradecer à Universidade Nova por me ter acolhido e ajudado a definir o meu percurso profissional, bem como à fundação Tezos pela inserção no projeto "FACTOR - A Functional Programming Approach to Teaching Portuguese Foundational Computing Courses".

Por último, um agradecimento ao professor João Lourenço pelo *template* da dissertação e aos meus amigos, por me ajudarem a manter a minha saúde mental durante o período de pandemia.

“Life can only be understood backwards;
but it must be lived forwards.”
(Søren Kierkegaard)

RESUMO

Com a crescente procura de cursos de Engenharia Informática, são muitos os alunos que precisam de realizar uma disciplina de Teoria da Computação e depois de usar o que aprenderam na sua vida profissional. Mas os tópicos lecionados não são simples. Para ajudar no ensino, têm vindo a ser desenvolvidas as mais diversas ferramentas pedagógicas.

É esta a motivação por trás da elaboração da biblioteca OCamlFLAT e da aplicação web OFLAT. Como o nome indica, OCamlFLAT é uma implementação de alguns dos conceitos referidos usando a linguagem de programação OCaml. Quanto à ferramenta OFLAT, desenvolvida sobre a biblioteca anterior, permite a visualização e manipulação gráfica dos conceitos teóricos, incluindo a animação interativa de determinados processamentos.

Esta dissertação visou estender as duas ferramentas com a adição de suporte para autómatos de pilha. Foram desenvolvidas as funcionalidades normais esperadas neste domínio, tanto na biblioteca, ao nível lógico, como na aplicação gráfica, através duma interface gráfica ambiciosa com preocupações pedagógicas. Exemplos de funcionalidades desenvolvidas: aceitação e geração de palavras; transformações diversas tais como eliminar os estados inacessíveis; predicados para testar propriedades como por exemplo o determinismo. De notar que estas funcionalidades se aplicam a autómatos de pilha deterministas e não-deterministas.

Neste documento apresenta-se e discute-se criticamente o resultado deste trabalho.

Palavas chave: Teoria das linguagens formais e autómatos; Programação Funcional; OCaml; OCamlFLAT; OFLAT

ABSTRACT

With the growing demand for Computer Engineering courses, more and more students are required to complete a Computer Theory course and use this knowledge in their professional life. However, the topic is not simple and for many students the lecture becomes an obstacle. In order to make the teaching of the Theory of Formal Languages and Automata more effective, several pedagogical tools focused on the topic have been developed.

This is the motivation behind the development of the OCamlFLAT library and the web application OFLAT. OCamlFLAT, as the name implies, is an implementation of some of the mentioned concepts developed in the OCaml programming language. Regarding OFLAT, developed on the previous library, it allows the visualization and graphic manipulation of theoretical concepts, supporting the animation of some of the processes and providing an easy interaction on behalf of the user.

Several modules have been implemented in order to enrich the system by providing support to pushdown automata in the OCamlFLAT library, as well as in the OFLAT tool. The modules contain a group of fundamental operations towards the study of these automata such as the process of acceptance of a word, transformation operations such as the cleanse of states that are not useful and the test of some properties such as checking the determinism of the automaton.

Keywords: Formal Languages and Automata Theory; Functional Programming; OCaml; OCamlFLAT; OFLAT

ÍNDICE

1 INTRODUÇÃO ... 1

1.1 Enquadramento e motivação ... 1

1.2 Objetivos ... 1

2 FERRAMENTAS PEDAGÓGICAS PARA FLAT ... 3

2.3 JFLAP ... 3

2.4 Automaton Simulator ... 5

2.5 SimStudio/CMSimulator ... 6

2.6 jFAST ... 7

3 BIBLIOTECA OCAMLFLAT ... 9

3.1 Estrutura ... 9

3.2 Módulos ... 9

4 OFLAT ... 13

4.1 Js_of_ocaml ... 13

4.2 Cytoscape.js ... 14

4.3 Model-View-Controller(MVC) ... 14

5 PROGRAMAÇÃO FUNCIONAL EM OCAML ... 15

5.1 Vantagens ... 15

5.2 Desvantagens ... 16

5.3 OCaml ... 16

6 TEORIA FLAT ... 19

6.1 Hierarquia de Chomsky ... 19

6.2 Autómatos Finitos ... 20
6.3 Autómatos de Pilha ... 20
6.3.1 Exercício ... 23
6.3.2 Resolução ... 23
6.3.3 Algumas das operações a implementar ... 24
7 IMPLEMENTAÇÃO DE CONCEITOS DA BIBLIOTECA OCAMLFLAT ... 27
7.1 Representação interna ... 27
7.2 Validate ... 29
7.3 Accept ... 30
7.4 Generate ... 31
7.5 nextConfs ... 32
7.6 getReachableStates ... 32
7.7 getProductiveStates ... 33
7.8 getUsefulStates ... 33
7.9 getUsefulTransitions, getUsefulSymbols e getUsefulStackSymbols ... 33
7.10 clean ... 34
7.11 isDeterministic ... 34
7.12 Outros métodos ... 34
8 APRESENTAÇÃO DA INTERFACE GRÁFICA OFLAT ... 37
8.1 Botão "Browse" ... 37
8.2 Definição de um autómato ... 38
8.3 Botões internos ao autómato ... 39
8.3.1 Botão "Clean" ... 40
8.3.2 Botão "See Automaton Specification" ... 40
8.3.3 Botões "Useful States", "Accessible States" e "Productive States" ... 41
8.4 Botão "Generate word with size x" ... 41
8.5 Botões Step-by-step word acceptance ... 41

8.6 Botão "Test word" ... 42

9 IMPLEMENTAÇÃO DA COMPONENTE GRÁFICA ... 45

9.1 Controller ... 46

9.1.1 definePushdownAutomaton ... 46

9.1.2 Listeners ... 47

9.1.3 getWords ... 47

9.1.4 accept ... 48

9.1.5 Accept Iterativo ... 48

9.2 PushdownAutomatonGraphics ... 49

9.2.1 Adaptação do código ezjs_cytoscape ... 50

9.2.2 Métodos internos ... 53

9.3 StateVariables ... 58

9.4 HtmlPageClient ... 58

9.5 Considerações finais de capítulo ... 59

10 AVALIAÇÃO E ANÁLISE CRÍTICA ... 61

10.1 Comparação com JFLAP ... 61

10.1.1 Autómato determinista ... 61

10.1.2 Autómato não-determinista ... 62

10.2 Avaliação e análise crítica ... 63

11 CONCLUSÕES E TRABALHO FUTURO ... 65

11.1 Conclusões ... 65

11.2 Trabalho Futuro ... 65

BIBLIOGRAFIA ... 67

ÍNDICE DE FIGURAS

Figura 1: Exemplo de um autómato de pilha em JFLAP 4
Figura 2: Exemplo de um autómato de pilha no Automaton Simulator 5
Figura 3: exemplo de um autómato de pilha em CMSimulator 6
Figura 4: exemplo de um autómato de pilha em jFAST 7
Figura 5: representação gráfica do autómato de pilha definido 22
Figura 6: implementação do método accept da biblioteca OCamlFLAT 31
Figura 7: implementação do método nextConfs da biblioteca OCamlFLAT 32
Figura 8: botão browse e a janela resultante ao carregamento do autómato 38
Figura 9: autómato de pilha definido em JSON 39
Figura 10: Botões internos à janela referentes ao autómato de pilha 40
Figura 11: resultado da janela após o clique do botão "clean" 40
Figura 12: resultado da janela após o clique no botão "See automaton specification" 41
Figura 13: resultado da janela após o clique do botão "Generate word with size x" 41
Figura 14: exemplo da palavra de entrada "0011" aceite por um autómato através dos botões de accept sequencial 42
Figura 15: implementação da função definePushdownAutomaton 46
Figura 16: implementação da função getWords 48
Figura 17: implementação do método changeRE 49
Figura 18: bindings implementados para a criação de um grafo 52
Figura 19: implementação do método startPDA 54
Figura 20: implementação do método next 56
Figura 21: implementação do método autoAccept 58
Figura 22: comparação da interface das duas ferramentas no caso de o autómato ser determinista 62

Figura 23: comparação da interface das duas ferramentas no caso de o autómato ser não-deterministaFigura 22: comparação da interface das duas ferramentas no caso de o autómato ser determinista... 62
Figura 23: comparação da interface das duas ferramentas no caso de o autómato ser não-determinista... 62
Figura 23: comparação da interface das duas ferramentas no caso de o autómato ser não-determinista... 62

ÍNDICE DE TABELAS

Tabela 1: tabela de transições do autómato definido 22
Tabela 2: demonstração da aceitação da palavra de entrada "0011" 22
Tabela 3: tabela de transições que define o mesmo autómato mas segundo o critério de pilha vazia 24

Siglas

FLAT	Teoria de Linguagens Formais e Autómatos (*Formal Languages and automata theory*)
MVC	Model-View-Controller
JFLAP	Java Formal Languages and Automata Package
OCaml	Objective Caml
HTML	HyperText Markup Language
jFAST	Java Finite Automata Simulation Tool
JSON	JavaScript Object Notation

SÍMBOLOS

ε	Transição que não consome símbolo / palavra de entrada vazia / substituição do topo da pilha por uma nova sequência vazia
Σ	Alfabeto de entrada
Γ	Alfabeto da pilha
δ	Relações de transição

1

Introdução

1.1 Enquadramento e motivação

No âmbito da promoção da linguagem de programação OCaml, o projeto FACTOR (Functional ApproaCh Teaching pOrtuguese couRses) surgiu com o intuito de produzir ferramentas com fins pedagógicos na área da Lógica Computacional e Fundamentos de Computação, o que inclui teoria de linguagens formais e autómatos (FLAT - *Formal Languages and Automata Theory*).

A compreensão dos fundamentos da informática é fundamental, uma vez que propicia o pensamento crítico dos alunos, bem como a compreensão mais aprofundada de variados conteúdos da área de informática. Ao longo dos anos tem havido um aumento da procura de métodos que facilitem esse ensino em diferentes áreas.

A biblioteca OCamlFLAT e a aplicação web OFLAT foram desenvolvidas no contexto do projeto Factor, tratam-se de duas peças de software complexas e o trabalho que se perspetiva não é trivial. A realização desta tese consiste em estender o seu suporte para mais modelos de computação.

O código produzido para a elaboração deste projeto encontra-se no repositório:
https://github.com/SaboneteAssado/PDA-in-OCamlFLAT-OFLAT

1.2 Objetivos

Ao longo dos anos foram criadas várias ferramentas, muitas com o intuito de facilitar o ensino de variados temas, sendo usadas, sobretudo, em contexto pedagógico. Estas ferramentas são vistas também como um meio de cultivar mais interesse por parte dos alunos nesta área.

Com este trabalho pretende-se integrar mecanismos teóricos de autómatos de pilha na ferramenta que já existe para que sejam usados de forma pedagógica, sendo que se deu prioridade às operações que se consideraram essenciais sobre os mesmos.

2

FERRAMENTAS PEDAGÓGICAS PARA FLAT

Com o aparecimento dos computadores eletrónicos tentou-se satisfazer a necessidade de representar modelos FLAT, sendo que a primeira ferramenta disponível surgiu nos anos 60 com a capacidade de representar máquinas de Turing. A maioria das ferramentas existentes têm, como já foi referido, objetivos pedagógicos. Algumas são ferramentas baseadas em texto; outras dão prioridade à componente gráfica.

Apesar da extensa lista de ferramentas enumeradas no artigo Fifty Years of Automata Simulation: A Review [5] acredita-se que a ferramenta que está a ser desenvolvida tenha potencial para ser uma ferramenta de topo no que se refere ao estudo de FLAT, visto que possui suporte para a manipulação gráfica de autómatos de pilha deterministas e não-deterministas.

De modo a entender como abordar a implementação destes autómatos foi necessário realizar um estudo aprofundado ao estado da arte. Assim, serão analisadas criticamente algumas ferramentas que possibilitam a visualização gráfica e, de forma mais geral, recolher informações que possam vir a ser usadas para melhorar a solução proposta.

2.3 JFLAP

O Java Formal Languages and Automata Package (JFLAP) é a ferramenta mais popular que irá ser mencionada. Criada em meados dos anos 90 por Procopiuc et al. deve todo o seu sucesso por mérito próprio, fruto de um grande trabalho de desenvolvimento da ferramenta ao longo das décadas seguintes. Apesar de inicialmente ter começado a ser desenvolvida em C++, foi reescrita em Java. Esta ferramenta desktop começou por ser utilizada na universidade de Duke, mas muito depressa alastrou-se e é hoje utilizada em muitas outras, tanto de um ponto de vista nacional como internacional.

Muito do seu sucesso deve-se também ao facto de ser *open-source* e de ter muita documentação disponível, complementada com um guia de utilização e explicação da teoria FLAT, bem como bastantes resoluções de exercícios.

O JFLAP possui muitas funcionalidades, cobrindo tópicos como a linguagens regulares, linguagens livres de contexto e linguagens recursivamente enumeráveis.

Relativamente às linguagens regulares, podem-se fazer simulações de autómatos finitos e de gramáticas e expressões regulares. A ferramenta permite também a transformação de autómatos não-deterministas finitos em deterministas, bem como de autómatos deterministas finitos para gramáticas e expressões regulares.

Relativamente aos autómatos de pilha, o JFLAP permite a sua simulação e transformação em gramáticas livres de contexto e vice-versa. Apesar de não ser perfeita, cobre na totalidade as funcionalidades fundamentais que podem ser realizadas sobre os mesmos. Estão ainda disponibilizados geradores para diversos tipos de *parsers*, nomeadamente LL e LR.

Por último, relacionado às linguagens recursivamente enumeráveis, é possível a simulação de máquinas de Turing básicas e de n fitas. Em todos os autómatos pode-se verificar iterativamente o processo de aceitação de uma palavra introduzida e se a mesma pertence à gramática dos autómatos. Estão ainda disponíveis algumas funcionalidades relativas a máquinas de Moore e de Mealy, bem como sistemas de Lindenmayer que já não são de igual interesse para o projeto.

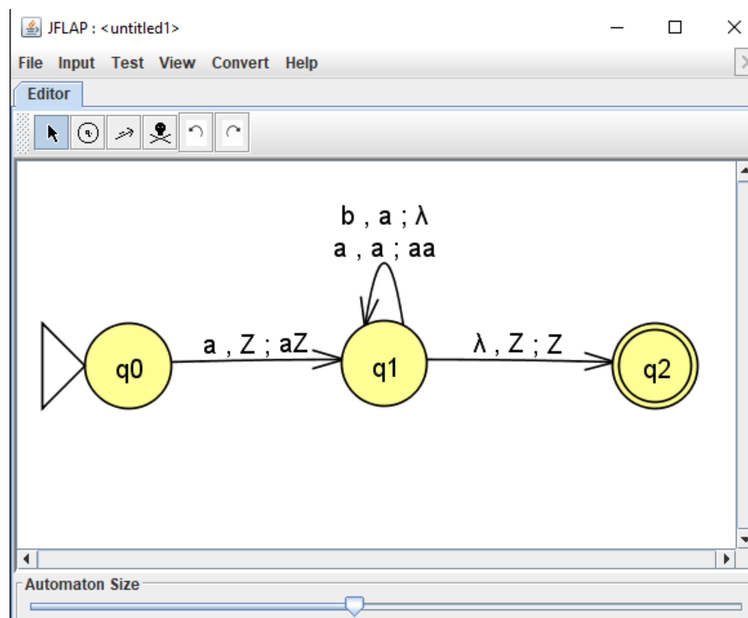

Figura 1: Exemplo de um autómato de pilha em JFLAP

O JFLAP é de longe a ferramenta mais completa das que irão ser mencionadas, seria a ferramenta ideal se tivesse uma facilidade de utilização mais próxima da que se quer atribuir à ferramenta OFLAT. Queremos que o OFLAT possa estar disponível via browser e sem qualquer download ou instalação. A interface consegue ser pouco intuitiva e visualmente é muito

pouco apelativa. Relativamente aos autómatos de pilha, houve uma quantidade desnecessária de aprendizagem anexada para se conseguir iniciar a experimentação das funcionalidades da aplicação. Existem demasiados botões com as suas funcionalidades mal agrupadas. A aprendizagem não é excessiva, mas seria ótimo se fosse evitada.

2.4 Automaton Simulator

Criada no virar do século, apesar de muito limitada a nível de funcionalidades, a ferramenta está disponível num site ao qual podemos aceder em qualquer dispositivo através dum browser.

Esta ferramenta suporta autómatos finitos, autómatos de pilha e máquinas de Turing. Permite concebê-los, desenhando na ferramenta o autómato de forma gráfica e posteriormente efetuar testes de aceitação de palavras. O processo de reconhecimento de uma palavra pode ser observado interactivamente, de forma sequencial e animada, ou podemos obter o resultado final instantaneamente, dispensando qualquer animação. Porém, não possui qualquer suporte para gramáticas, o que é uma lacuna importante se quiser ser considerada uma ferramenta de topo. Após pouca utilização também se observaram alguns bugs de impacto menor no desenho do autómato e alguns aspetos relativos à nomenclatura que são menos comuns.

Por último, gostava de fazer referência à simplicidade e à facilidade de acesso. A interface é simples e apresenta as suas funcionalidades por meio de imagens ou pequenos textos, tornando-se bastante intuitiva. Parece-me relevante, a nível pedagógico, a implementação de

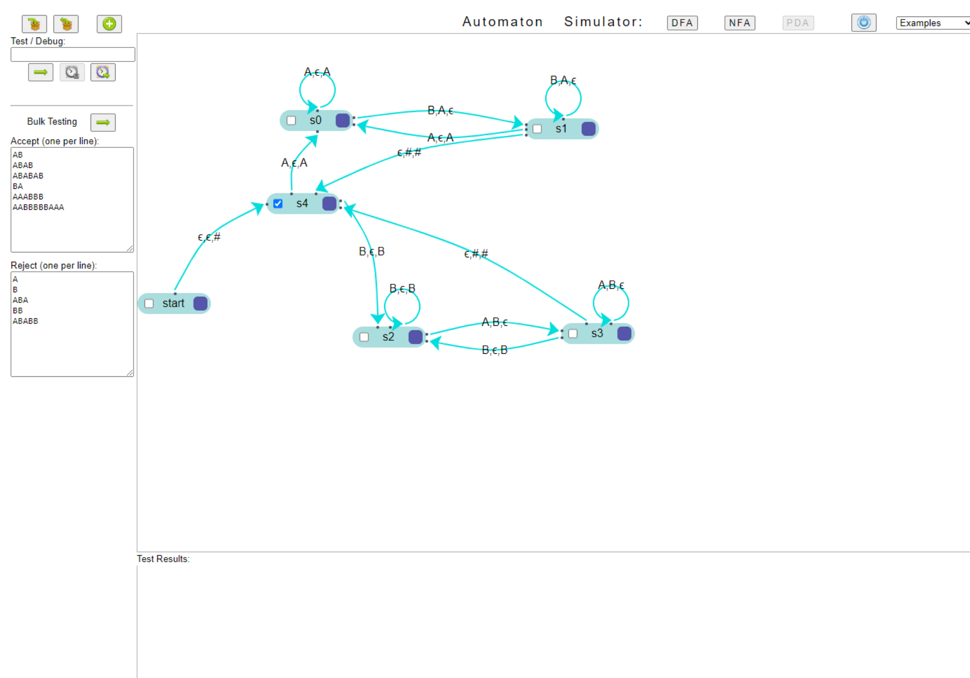

Figura 2: Exemplo de um autómato de pilha no Automaton Simulator

mecanismos semelhantes para os autómatos de pilha que estão em falta nas ferramentas que me proponho a estender.

2.5 SimStudio/CMSimulator

O SimStudio é uma ferramenta para Windows desenvolvida por Čerňanský et al., Chudá e Rodina no âmbito académico e com o intuito de auxiliar alunos da Universidade Eslovaca de Tecnologia em Bratislava.

Suporta a simulação de autómatos finitos, autómatos de pilha e máquinas de Turing. O autómato é introduzido através do desenho de um diagrama gráfico. O comportamento do autómato em reconhecimento, pode posteriormente ser observado usando animação automática ou dirigida pelo utilizador usando um botão.

Apesar de segundo o artigo Fifty Years of Automata Simulation: A Review [5] mencionar o sucesso desta ferramenta na Eslováquia, devido à falta de documentação e de tradução da língua eslovaca existe uma barreira que me impossibilita o uso fácil da ferramenta, não sendo possível realizar o estudo da ferramenta e confirmar o sucesso referido. No entanto, foi também desenvolvido o CMSimulator, uma versão do SimStudio adaptada a Android e que, aparentemente, não trouxe nenhuma funcionalidade nova quando comparado com o seu antecessor destinado a desktop. Essa aplicação já se encontra disponível em inglês e o seu uso é fácil

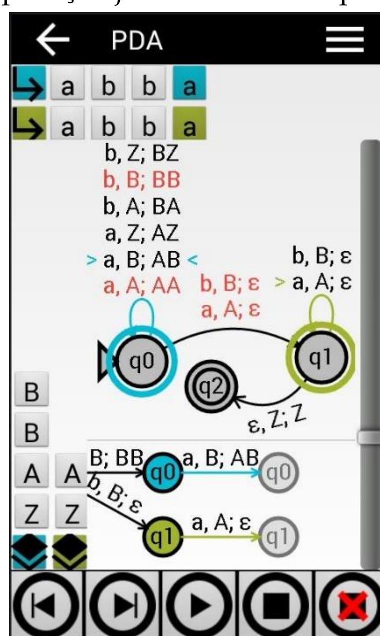

Figura 3: exemplo de um autómato de pilha em CMSimulator

e intuitivo, apesar de não muito apelativo esteticamente. Talvez algum do seu sucesso também se deva a esta disponibilização móvel que possibilita o seu uso a qualquer momento.

2.6 jFAST

O Java Finite Automata Simulation Tool (jFAST) é uma ferramenta programada em Java com uma interface simples e que esteticamente podia ser melhorada.

A nível de funcionalidades, encontra-se bastante limitada depois das ferramentas referidas. A nomenclatura também não é standard, pelo que o seu uso pode ser dificultado. Um aspeto interessante desta ferramenta é, no entanto, a sua extensibilidade. Apesar de algumas das ferramentas já mencionadas serem *open-source* e assim dar azo à adição de funcionalidades, esta aplicação foi direcionada quase que exclusivamente para esse ramo. Existe uma tentativa clara de incentivo à criatividade e desenvolvimento por parte dos utilizadores devido à divisão dos módulos de forma lógica e à fácil adição de funcionalidades à ferramenta. Existe ainda diversa documentação acerca da extensibilidade da aplicação e ainda tutoriais para as funcionalidades mais comuns.

A aplicação suporta autómatos finitos, autómatos de pilha e máquinas de Turing. Podem ser criados de forma simples, tudo através da sua seleção com o rato, arrastar para a posição desejada e alterar as suas propriedades com o lado direito do rato, embora este método não seja favorável para autómatos mais complexos. Um utilizador pode ver o processo de aceitação de palavras de forma interativa, podendo mesmo selecionar novos caminhos durante o processo. Permite também a geração de autómatos complexos através da geração de subautómatos menos complexos.

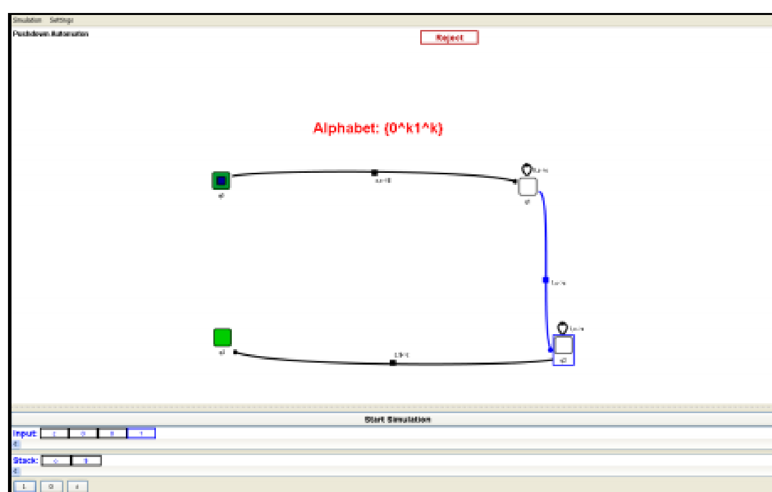

Figura 4: exemplo de um autómato de pilha em jFAST

A aceitação no autómato de pilha é feita unicamente por pilha vazia, ou seja, a aceitação de uma palavra por estados de aceitação não está contemplada.

3

BIBLIOTECA OCAMLFLAT

A biblioteca OCamlFLAT, desenvolvida na FCT/UNL, permite a criação e análise de exemplos de alguns tipos de modelos FLAT estudados na cadeira de Teoria de Computação da FCT/UNL. Cada exemplo de modelo FLAT é definido pelo utilizador usando sintaxe JSON, que a biblioteca sabe interpretar.

3.1 Estrutura

Com o tempo, a biblioteca passou dum estado mais primitivo onde, devido à existência de poucos módulos, a sua constituição passava por apenas 3 ficheiros, para o estado onde se encontra, onde todos os módulos se encontram separados em função do conceito teórico que representam. Sentiu-se a necessidade de haver esta separação de conceitos para facilitar a expansão da biblioteca no futuro, como é o caso deste trabalho.

3.2 Módulos

Cada módulo representa ou um conceito teórico diferente ou um conceito abstrato para ser usado na construção desses módulos. Vão ser enumerados os diferentes módulos e explicitado o seu intuito.

3.2.1.1 Entity

O módulo contém uma classe que está na raiz de uma hierarquia de classes, a maioria das quais representam modelos FLAT. Portanto, na classe da raiz estão fatorizados aspetos comuns a todas as entidades tal como um nome, uma descrição, etc.

3.2.1.2 Model

O módulo está encarregue de representar todos os mecanismos FLAT definidos. As funções comuns a todos os conceitos vão estar aqui definidas. Contém uma classe que herda de "Entity" e da qual herdam todas as classes que representam modelos FLAT.

3.2.1.3 Exercise

O módulo permite a criação de exercícios através da definição parcial de linguagens. O utilizador recebe uma descrição de uma linguagem pretendida, bem como um conjunto de palavras que pertencem à referida linguagem, e um segundo conjunto de palavras que não lhe pertencem. O objetivo é que o utilizador tente definir um exemplo de mecanismo (dependendo do que o exercício especifica) onde a linguagem que ele define é a mesma linguagem definida nos exercícios. Os dois conjuntos de palavras podem ser usados para testar se o exemplo está correto. Se o exercício está definido corretamente, o exemplo pode ser considerado correto se todas as palavras do primeiro conjunto são aceites pelo exemplo, e se nenhuma palavra do segundo conjunto for aceite.

3.2.1.4 FiniteAutomaton

O módulo é usado para definir os autómatos finitos. Todas as características e funções que processam os autómatos de forma direta são definidas aqui.

Eis uma seleção dos métodos mais importantes definidos neste módulo:

- method validate: unit
- method accept: word -> bool
- method generate: int -> words
- method reachable: state -> states
- method productive: states
- method cleanUselessStates: model
- method toDeterministic: model
- method minimize: model

3.2.1.5 RegularExpression

O módulo é usado para definir expressões regulares. O que foi referido para o módulo anterior é válido aqui, mas aplicado a expressões regulares.

Eis uma seleção dos métodos mais importantes definidos neste módulo:

- method validate: unit
- method accept: word -> bool

- method generate: int -> words

3.2.1.6 ContextFreeGrammar

O módulo é usado para definir gramáticas independentes de contexto. O que foi referido para o módulo anterior é válido aqui, mas aplicado a gramáticas independentes de contexto.

Eis uma seleção dos métodos mais importantes definidos neste módulo:

- method validate: unit
- method accept: word -> bool
- method generate: int -> words

3.2.1.7 Error

O módulo regista todos os erros que são detetados, por exemplo durante a validação dos modelos.

3.2.1.8 Util

O módulo contém todas as funções que não estejam relacionadas de forma direta com qualquer outro módulo, mas que vão sendo precisas por toda a biblioteca. Possui funções como conversões de tipo e de leitura de ficheiros. As funções estando aqui suprime a necessidade de duplicação de código.

3.2.1.9 Set

O módulo contém as funções relativas a conjuntos. Como se sabe, conjuntos têm um papel fulcral em FLAT.

3.2.1.10 JSON

O módulo contém as funções necessárias para manipular ficheiros JSON da biblioteca. Estes ficheiros vão possuir informação sobre as gramáticas e autómatos.

3.2.1.11 RegExpSyntax

O módulo faz o parsing de expressões regulares.

3.2.1.12 CFGSyntax

O módulo faz o parsing de gramáticas livres de contexto.

4

OFLAT

A aplicação web OFLAT, desenvolvida na FCT/UNL, permite criar e manipular graficamente exemplos de alguns tipos de modelos FLAT estudados na cadeira de Teoria de Computação da FCT/UNL. Algumas das operações são animadas, como por exemplo o reconhecimento duma palavra usando um autómato finito.

Para explicar a ferramenta OFLAT, há que salientar os seguintes pontos principais:

• É uma aplicação web que funciona exclusivamente do lado do cliente, portanto sem aceder a qualquer servidor.

• Está programada em OCaml, JavaScript e HTML.

• Recorre de forma essencial a duas peças de software: a aplicação Js_of_ocaml, que traduz código OCaml em código JavaScript, capaz de correr no browser; e a biblioteca Cytoscape.js, que suporta a representação e manipulação gráfica interativa de grafos.

• O código da aplicação está organizado de acordo com o padrão de desenho de software Model-Controller-View (MVC).

• A interface do utilizador é bastante intuitiva e a aplicação está preparada para crescer, aceitando novas funcionalidades.

4.1 Js_of_ocaml

A aplicação web usa um misto de código OCaml e JavaScript, sendo maioritariamente escrita em OCaml. Dito isto, o leitor deve estar a perguntar-se como se corre código escrito em OCaml num browser. A resposta é simples, com a biblioteca Js_of_ocaml. É esta a motivação para o seu uso.

O Js_of_ocaml é um compilador de OCaml *bytecode* para JavaScript. Esta biblioteca traduz o código OCaml e ao mesmo tempo tem o poder de ligar esse código a código JavaScript que seja invocado a partir de código OCaml.

4.2 Cytoscape.js

O Cytoscape.js é uma biblioteca implementada em JavaScript para representação e processamento de grafos. Foi desenvolvida no contexto da plataforma de visualização em áreas científicas, sobretudo em Bioinformática, embora já seja usada em muitas outras áreas. É uma biblioteca orientada para a representação e manipulação gráfica interativa de grafos, compatível com todos os browsers modernos. Tem licença de software permissiva e foi concebida para ser bastante extensível, tendo uma comunidade dedicada que ativamente desenvolve extensões para a biblioteca.

4.3 Model-View-Controller(MVC)

O MVC é um padrão de software design muito usado para o desenvolvimento de interfaces de utilizador que divide o programa em três componentes interligadas.

O "Model" representa os dados dinâmicos da aplicação. No caso do OFLAT, consiste num conjunto de variáveis imperativas de OCaml que guardam o estado corrente da aplicação, incluindo os mecanismos FLAT que estão correntemente a ser visualizados.

O "View" trata da visualização de dados. No caso do OFLAT, consiste numa página HTML estática, de partida, e tem diversas funcionalidades de alteração dinâmica do HTML que são chamadas a partir do controlador.

O "Controller", constituído também por um conjunto de listeners, define como é que a aplicação reage à interação do utilizador, por exemplo o que acontece quando o utilizador clica num botão ou escolhe uma opção do menu dum nó do grafo.

5

PROGRAMAÇÃO FUNCIONAL EM OCAML

Apesar de pouco sucesso no desenvolvimento comercial de aplicações e de ser principalmente usado no âmbito académico, o paradigma funcional pode ser muito conveniente devido à sua simplicidade. Baseia-se na construção e aplicação de funções em que a sua definição são árvores de expressões que apontam para outros valores na árvore, em vez de um conjunto de regras imperativas que atualizam o estado do programa. Em linguagens puramente funcionais essas funções são tratadas como funções matemáticas deterministas, pelo que vai retornar sempre o mesmo resultado para os mesmos argumentos, pois não é afetado por qualquer mudança de estado ou outros efeitos secundários.

Vamos aprofundar estas noções de seguida referindo algumas vantagens e desvantagens deste tipo de programação e falar um pouco no caso específico da linguagem OCaml com que a biblioteca foi maioritariamente desenvolvida e que terei de usar para adicionar conteúdo.

5.1 Vantagens

Dependendo do objetivo que se pretende atingir, o paradigma funcional traz algumas vantagens inerentes, quando contrastado com outros paradigmas. As linguagens funcionais puras não possuem noção de estado, pelo que por norma o seu comportamento pode ser comparado a funções matemáticas. Uma função é definida, fornece-se um input e é produzido um output, sem haver necessidade de acesso a variáveis declaradas anteriormente. A maior vantagem desta propriedade é a capacidade de apresentação de métodos e funções, tornando quase trivial para um olho habituado o seu comportamento. Não é necessário perceber conteúdo fora de uma função, basta pensar no fim que se quer atingir, não sendo necessária grande ginástica mental, que normalmente está implícita noutros paradigmas de programação. Não é necessário também perceber em que ordem o programa executa as funções, visto que uma função não interfere com o input de outra pois nunca correm de forma concorrente, facilitando

implementações em paralelo. Outra vantagem é a alocação automática de memória, sendo que em momento algum é necessário a alocação explícita.

Nas implementações deste projeto tentou-se usar ao máximo as características da programação funcional de modo a usufruir destas vantagens, tentando ao máximo afastar dos mecanismos de paradigma imperativo que são oferecidos em OCaml.

Destaca-se também a ponte entre modelos matemáticos e a sintaxe OCaml.

5.2 Desvantagens

Apesar de útil em muitos casos onde é mais fácil expressar determinados conceitos de forma puramente matemática, que se consegue ter vantagem usando linguagens de programação funcional, muitos problemas do mundo real necessitam da noção de estado. No entanto, a maioria das linguagens puramente funcionais evoluíram ou deram origem a linguagens multiparadigma com mecanismos de representação de estado adicionados. Nestes casos já é possível a definição de variáveis imperativas. Apesar de parecer algo positivo superficialmente, perdemos por completo as vantagens que estão inerentes a este mecanismo, arriscando a potencialmente perder a legibilidade e o funcionamento das funções, pelo que se torna pertinente perguntar se é benéfico o uso dessa linguagem ou seria mais vantagem usar outra. Outra questão que se pode levantar é o facto de apesar de muitas vezes ser positivo a alocação automática de memória, não deixa de ser sempre uma espada de dois gumes, pois pode ser do interesse do programador fazer a gestão manual da memória.

5.3 OCaml

Em 1996, devido à necessidade de suporte de técnicas de orientação de objetos em Caml, foi criada Objective Caml, conhecida mais comummente como OCaml, pequeno para Objective Categorical Abstract Machine Language. Em anos mais recentes tem ganho em parte alguma notoriedade devido à sua capacidade de geração de código rápido por parte do compilador, ao seu desempenho a nível de tempos de execução, biblioteca base extensa e *pattern matching*. O OCaml, bem como Caml, são da família "ML". As linguagens desta família são consideradas linguagens funcionais impuras, por permitirem programação imperativa.

Como OCaml é de tipo estático previne erros de *type mismatch* retirando verificações de tipo em *runtime* e *safety checks* que perturbam a performance de linguagens de tipo dinâmicas.

6

TEORIA FLAT

Neste capítulo vão ser introduzidos os conceitos teóricos mais importantes para o projeto proposto.

A Teoria da Computação é um ramo da ciência da computação que se dedica ao estudo de máquinas abstratas e aos problemas computacionais que estas podem resolver. Por exemplo, um algoritmo que possa ser expresso usando um autómato finito, também pode ser expresso usando um autómato de pilha; mas não vice-versa.

Para discutir e comparar diferentes níveis de poder expressivo, é preciso considerar os algoritmos no contexto de algum domínio de aplicação, sendo que o tradicionalmente preferido nestes estudos é o da sintaxe de linguagens formais. Foram identificadas diversas classes de linguagens formais, com diferentes níveis de complexidade de especificação, a requerer modelos de computação com diferentes níveis de poder expressivo para resolver o problema do reconhecimento dessas linguagens.

6.1 Hierarquia de Chomsky

Há duas técnicas de definição de linguagens que geralmente se usam. A primeira técnica baseia-se no conceito de geração e usa o formalismo das gramáticas. Outra técnica baseia-se no mecanismo de reconhecimento e usa o formalismo dos autómatos.

A hierarquia de Chomsky-Schützenberger, também comummente denominada por apenas hierarquia de Chomsky, consiste numa classificação de gramáticas formais por níveis de complexidade de especificação, tal como sugerido por Noam Chomsky em 1956.

A hierarquia está dividida em quatro níveis de gramáticas formais, cada nível com uma classe de linguagem que pode gerar, o tipo de autómato que a reconhece e ainda as regras que têm de possuir.

A hierarquia define-se da forma seguinte:

- Tipo 3: Gramáticas lineares direitas ou esquerdas (equivalentes a expressões regulares), que correspondem aos autómatos finitos;
- Tipo 2: Gramáticas independentes de contexto que correspondem aos autómatos de pilha;
- Tipo 1: Gramáticas dependentes do contexto que correspondem aos autómatos linearmente limitados;
- Tipo 0: Inclui todas as gramáticas, sem qualquer restrição que correspondem às máquinas de Turing.

Dado ser uma hierarquia, temos que Tipo 3 $\subseteq$ Tipo 2 $\subseteq$ Tipo 1 $\subseteq$ Tipo 0 [7].

No entanto, existem algumas noções que precisam de ser introduzidas antes de os referir de modo a dar algum encadeamento de forma lógica e atribuir algum enquadramento a algumas noções que irão ser referidas.

6.2 Autómatos Finitos

Os autómatos finitos têm um poder expressivo que cai na categoria 3 da hierarquia de Chomsky. As linguagens reconhecidas por estes autómatos são as mesmas que as reconhecidas pelas gramáticas regulares. Podem ser definidos informalmente através de diagramas de estado que nos facilitam a perceção do funcionamento dos mesmos de forma intuitiva. No entanto, existe uma abordagem mais formal.

Podem-se então definir formalmente como sendo um énuplo do tipo $(Q, \Sigma, \delta, q_i, F)$, onde:

- Q é um conjunto finito de estados;
- Σ é um conjunto finito chamado de alfabeto;
- $\delta : Q \times \Sigma \times Q$ é a relação de transição, que representa a passagem de um estado de Q para outro estado de Q através do consumo de um símbolo de Σ;
- $q_i \in Q$ é o estado inicial;
- $F \subseteq Q$ é o conjunto de estados de aceitação.

6.3 Autómatos de Pilha

Neste subcapítulo desenvolve-se o tema dos autómatos de pilha que já foi mencionado como sendo o aspeto fulcral deste projeto. Para além de definições formais e análises intuitivas, identificam-se as operações principais pretendidas para a biblioteca e a aplicação, pensando na sua qualidade pedagógica.

Os autómatos de pilha têm um poder expressivo que cai no tipo 2 da hierarquia de Chomsky. As linguagens reconhecidas por estes autómatos são as mesmas que as reconhecidas pelas gramáticas independentes de contexto. Pode-se definir formalmente um autómato de pilha de forma semelhante aos autómatos finitos, havendo a necessidade extra de definir o símbolo inicial da pilha e o alfabeto da pilha, bem como um conjunto de regras de transição mais complexas.

Assim, um autómato de pilha pode ser definido pelo énuplo $(S, \Sigma, \Gamma, s, Z, \delta, F)$ em que:

- S é o conjunto de estados;
- Σ é o alfabeto de entrada;
- Γ é o alfabeto da pilha;
- $s \in S$ é o estado inicial do autómato;
- $Z \in \Gamma$ é o símbolo inicial da pilha;
- $\delta \subseteq (S \times \Gamma) \times (\Sigma \cup \{\varepsilon\}) \times (S \times \Gamma*)$ é a relação de transição;
- $F \subseteq S$é o conjunto de estados de aceitação.

A relação de transição específica o funcionamento do autómato durante o reconhecimento de uma palavra. A palavra é progressivamente reconhecida através dum percurso da esquerda para a direita, e em cada momento é considerado apenas um dos símbolos da palavra, o símbolo corrente.

Cada énuplo, de tipo $(S \times \Gamma) \times (\Sigma \cup \{\varepsilon\}) \times (S \times \Gamma*)$, pertencente à relação de transição, tem um significado que ao primeiro olhar pode não ser óbvio. Num determinado momento, com o autómato num dado estado de S e com um dado elemento de Γ no topo da pilha, consome-se o símbolo corrente, e estes três elementos determinam uma mudança para outro estado de S, com possível modificação do topo da pilha - o símbolo no topo pode ser trocado por uma qualquer sequência de símbolos. Existem ainda as transições-ε, que são transições especiais, que não dependem do símbolo corrente e não causam o consumo do símbolo corrente.

Para exemplificar, vamos considerar um autómato A1 que reconhece a linguagem {0n1n | $n \geq 0$}. Definimos A1 = $(S, \Sigma, \Gamma, p, Z, \delta, F)$ com:

- $\Sigma = \{0,1\}$
- S = {p,q,r}, onde p é o estado inicial do autómato
- $\Gamma = \{A,Z\}$, onde Z é o símbolo inicial da pilha
- F = {r}
- δ :

δ	0	1	ε
p,Z	p,AZ		q,Z
p,A	p,AA		q,A
q,A		q,ε	
q,Z			r,Z

Tabela 1: tabela de transições do autómato definido

O diagrama seguinte corresponde a uma forma gráfica de representar o mesmo autómato de pilha:

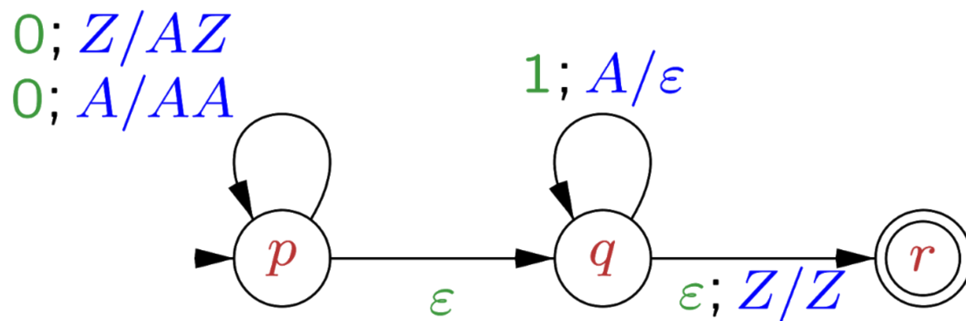

Figura 5: representação gráfica do autómato de pilha definido

Exemplifiquemos com o reconhecimento da palavra "0011". Esse reconhecimento envolve sete passos, ou seja, sete usos das transições disponíveis. Note que se começa no estado p com o símbolo Z na pilha.

Estado	Pilha	Palavra
p	Z	0011
p	AZ	011
p	AAZ	11
q	AAZ	11
q	AZ	1
q	Z	ε
r	Z	ε

Tabela 2: demonstração da aceitação da palavra de entrada "0011"

Como podemos observar, esta palavra é aceite, pois, a palavra foi processada até ao seu final e o processo atingiu o estado final r ∈ F. Fizemos o que se chama um "reconhecimento

baseado em estados de aceitação". A teoria também prevê a possibilidade de reconhecimento baseado na pilha vazia, sem usar estados de aceitação.

De notar que a existência duma transição- ε entre os estados p e q permitiria explorar outras possibilidades durante o reconhecimento desta palavra. Na verdade, essa transição- ε torna o autómato não-determinista. Contudo, no caso deste autómato, essas possibilidades adicionais conduzem sempre a um beco sem saída e ao não reconhecimento da palavra. A sequência de reconhecimento mostrada, com sete passos, é a única que permite obter sucesso.

Este autómato foi apresentado já completamente elaborado. Mas, uma questão permanece: qual o processo de pensamento para chegar a esse autómato? Observemos agora uma tentativa de exposição do procedimento de forma lógica, tentando fornecer alguma intuição.

6.3.1 Exercício

Defina um autómato que reconheça a linguagem definida por {0n1n | n ≥ 0}. O autómato deverá fazer reconhecimento baseado em estados de aceitação.

6.3.2 Resolução

Vamos primeiro tentar perceber a linguagem a reconhecer. A linguagem é constituída por palavras possivelmente vazias que começam por uma sequência de 0's e continuam por uma sequência de 1's. As sequências de 0's e 1's têm igual comprimento. Decorre da explicação anterior que o alfabeto de entrada é Σ = {0,1}.

Podemos agora identificar a necessidade dum estado inicial que representa a situação de se estar a processar uma sequência de 0's, um segundo estado que representa a situação de se estar a processar a sequência de 1's, e ainda um estado separado para a aceitação. Vamos começar por definir o conjunto de estados como S = {p,q,r}, com o estado inicial p e com o conjunto de estados de aceitação F = {r}.

Uma questão agora se levanta. Como arranjar um mecanismo que permita garantir que a sequência de 1's tem o mesmo comprimento do que a sequência de 0's. A resposta é simples e recorre ao uso da pilha. De acordo com as regras, a pilha começa contendo apenas o símbolo inicial Z. Por cada símbolo 0 da palavra, empilhamos o símbolo A e, desta forma, usamos a pilha para contar o número de 0's. Depois, quando começarem a aparecer 1's, desempilhamos um A por cada 1 que apareça, e assim conseguimos começar a garantir uma correspondência entre cada 0 e cada 1. Agora o que faz falta é uma transição para um estado de aceitação, sendo que essa transição deverá requerer que a pilha contenha apenas o símbolo Z, para garantir igualdade entre o número de 0's e o número de 1's.

Os nossos estados foram todos ligados por transições- ε , que é uma forma simples de tratar a questão das transições, e que funciona em muitos casos. Contudo, esta decisão tornou o autómato não-determinista. Para o autómato ficar determinista, bastaria que a transição entre o estado p e o estado q fosse feita aquando da deteção do primeiro 1 da palavra em reconhecimento.

Há uma outra variante dos autómatos de pilha que usa um critério diferente para a aceitação das palavras. Nesta variante não existem estados de aceitação. O critério de paragem baseia-se no facto da pilha ter ficado vazia quando a máquina para (e claro, a palavra de entrada foi processada até ao fim). Por exemplo, se quisermos modificar o autómato exemplificativo anterior para usar o critério da pilha vazia, teríamos de alterar para S = {p,q} pois r já não é mais necessário e F =∅. Relativamente à tabela de δ faríamos esta alteração:

δ	0	1	ε
p,Z	p,AZ		q,Z
p,A	p,AA		q,A
q,A		q,ε	
q,Z			q,ε

Tabela 3: tabela de transições, mas segundo o critério de pilha vazia

6.3.3 Algumas das operações a implementar

Visto que o trabalho que venho propor tem a finalidade de ser usado num ambiente pedagógico, gostava de propor alguns recursos de estudo para os alunos, bem como algumas ideias relativas à componente gráfica dos autómatos.

Nas ferramentas onde pretendo implementar os autómatos de pilha e as máquinas de Turing, há dois métodos - "Accept" e "Generate" - que é suposto estarem disponíveis para todos os mecanismos suportados. Portanto estes dois métodos serão implementados e neles residirá uma parte importante do trabalho.

O método "Accept" verifica se uma dada palavra é reconhecida pelo autómato. Ou seja, verifica se o autómato a partir do estado inicial e através das relações de transição, se após a palavra ser consumida o autómato se encontra num dos estados de aceitação definidos. Este processo apesar de fácil de definir e sem grandes complicações a nível matemático, acarreta alguns obstáculos a nível computacional. O caso mais favorável é aquele em que o autómato

é determinista, pois nunca existe qualquer dúvida sobre qual é a transição a efetuar em cada momento, e também se garante que chegará a uma conclusão sobre a aceitação ou a rejeição em tempo finito, por maior que seja a palavra. O autómato é determinista se para cada par (estado, pilha), existir no máximo uma transição por cada símbolo do alfabeto de entrada; além disso, se existir uma transição- ε para esse par, então não pode existir qualquer transição para nenhum símbolo do alfabeto de entrada.

Mas o método tem de lidar com a situação mais geral de não-determinismo. Uma abordagem que podemos realizar a este problema é efetuar as transições de forma paralela até algum dos percursos que está a ser tentado acabar por reconhecer a palavra. À superfície aparenta ser um problema simples. No entanto, podem ocorrer ciclos e ainda problemas relativamente às mudanças de estado, sem consumo de símbolos de entrada. Por exemplo, num autómato que contenha a transição q,Z, ε -> q,Z, a computação em princípio não termina, o que será um problema no caso duma palavra que deva ser rejeitada pelo autómato. Estamos perante um problema semi-decidível. A matemática lida bem com o infinito e não há problema se a definição do autómato tiver aspetos infinitos. Mas é evidente que para uma máquina, no reino da computação, já poderão existir problemas de não-terminação da computação. Pragmaticamente, precisamos de definir um limite para a procura, potencialmente através de alguma forma de noção temporal. Outra questão negativa que se levanta agora é que, o facto de existir um limite na procura, acarreta a possibilidade de não conseguirmos concluir o processo de reconhecimento de uma palavra que devia ser aceite pelo autómato.
O método “Accept” precisa de endereçar estes problemas e temos de aceitar que a necessidade de dar três respostas possíveis: “palavra aceite”, “palavra rejeitada” e “resultado desconhecido”. Uma técnica que permite mitigar o problema é, durante a exploração em paralelo de todas as alternativas, detetar que já não estão a ser geradas configurações novas e concluir “palavra rejeitada”.

A solução final para o problema será conseguir reescrever o autómato numa forma que garanta a terminação do método “Accept” (note bem, não significa que o autómato fique determinista). Isso parece ser possível: converter o autómato numa gramática independente de contexto, posterior redução à forma normal de Chomsky e voltar a transformar num autómato. Será interessante investigar se existe alguma solução na literatura que não obrigue a converter o autómato numa gramática.

Do ponto de vista gráfico, no contexto da aplicação OFLAT, interessa animar o semi-algoritmo de reconhecimento de uma palavra. No caso de o autómato ser determinista, não há dúvidas sobre como animar os acontecimentos. No caso não-determinista, irá requerer alguma meditação, para descobrir uma boa forma de lidar com o problema.

De notar que eu encaro estes problemas de decidibilidade, mais do que como um problema, como uma boa oportunidade de tornar este trabalho mais interessante.

O método "Generate" tem como intuito a produção de todas as palavras reconhecidas pelo autómato de comprimento máximo n. Uma abordagem ingénua seria criar todas as permutações de símbolos de entrada e testar cada uma delas usando o método "Accept". Naturalmente, esta abordagem é muito pouco eficiente, com complexidade exponencial. Uma abordagem mais lógica será explorar incrementalmente as possibilidades que existem na relação de transição, até se atingir o comprimento máximo pretendido da palavra gerada, descartando as possibilidades que já não fazem sentido explorar e conseguindo assim reduzir o número de possibilidades a cada iteração. Esta segunda via generaliza o método "Accept", pois funciona como o reconhecimento de uma palavra genérica desconhecida, que se irá concretizando a pouco e pouco.

De forma geral, nas animações gráficas, seria interessante ir revelando o conteúdo da pilha nos vários passos dos algoritmos.

7

Implementação de Conceitos da Biblioteca OCamlFLAT

O modelo definido para o autómato de pilha é constituído por 8 campos. O modelo é igual ao previsto na teoria com a exceção de um campo extra que se refere ao critério de aceitação. Caso este campo tenha o valor true, então o critério de aceitação do autómato é por estados de aceitação. Caso o valor seja false, então o critério de aceitação é por pilha vazia.

Como foi falado anteriormente, os mecanismos FLAT fundamentais estão representados no módulo Model. Para a generalidade dos métodos a sua implementação é linear ou não apresenta grande desafio, pelo que não é necessário debruçar-me no assunto previamente. No entanto, os métodos Validate, Accept e Generate são de alguma complexidade. Houve ainda a necessidade de implementar outros métodos adicionais de complexidade variada que permitem realizar um estudo mais aprofundado do autómato em causa.

Relembra-se o leitor de que durante toda a implementação houve uma atitude de programação funcional, passando pela prática constante da redução de cada problema a uma versão simplificada do problema (definição indutiva).

Vão ser mencionados todos os métodos da biblioteca referentes aos autómatos de pilha, mas os que são considerados mais importantes vão ser explicados detalhadamente.

7.1 Representação interna

Como seria de esperar, houve uma tentativa de corresponder a representação interna do autómato exatamente igual a como exposto na teoria. No entanto, foram tomadas algumas decisões de modo a garantir a correção do autómato inserido, para garantir que as ideias do utilizador foram transmitidas corretamente.

O Autómato é constituído pelos os seguintes campos:

- inputAlphabet, o alfabeto de entrada do autómato.

- type inputAlphabet: symbols

- stackAlphabet, o alfabeto da pilha;

    type stackAlphabet: symbols
- states, o conjunto dos estados da pilha;

    type states: states
- initialState, o estado inicial do autómato;

    type initialState: state
- initialStackSymbol,o símbolo inicial presente na pilha;

    type initialStackSymbol: symbol
- transition, o conjunto de transições. Uma transição é definda por um énuplo de 5 elementos, estado de origem, símbolo no topo da pilha, símbolo de entrada, estado de destino e novos símbolos no topo da pilha, respetivamente;

    type transitions : transitions
- acceptStates, o conjunto de estados de aceitação;

    type acceptStates : states
- criteria, de modo a indicar o critério de aceitação.

    type criteria: bool

Expostos os campos que definem o autómato, há um detalhe que pode ser observado e até esclarecido relativo ao campo *criteria*. Primeiramente, o leitor, caso tivesse abordado o problema de um modo mais superficial, podia argumentar que o campo não seria necessário, pois caso o campo *acceptStates* não fosse o conjunto vazio o critério seria por estados de aceitação, caso contrário, seria o critério de pilha vazia. No entanto, essa abordagem não está correta dum ponto de vista teórico. Apesar de um autómato cujo critério de aceitação de uma palavra seja por estados de aceitação e tenha esse conjunto vazio, pouca ou nenhuma análise há para ser feita, visto que não reconhece nenhuma palavra. Contudo, pode ser esse o caso de estudo desse autómato para o exercício em causa e vai de acordo com a teoria. De modo a impedir essa potencial ambiguidade, tomou-se a decisão de adicionar o campo *criteria*. Como se este argumento não fosse convincente, ainda se considerou útil incluir a inclusão do campo de modo a reduzir erros de definição do autómato que possam ocorrer por parte do utilizador e não induzir o mesmo em ideias e resultados errados. É de fácil observação por parte do leitor que tal se sucede, pois, é facilitada a transmissão da ideia à aplicação, sendo que erros no campo *acceptStates* ficam automaticamente notórios.

Este campo também facilita a compreensão e modelação de métodos mais complicados que irão ser falados futuramente.

7.2 Validate

Assinatura do método:
method validate: unit

O método não passa de uma convenção interna cujo intuito era sobretudo fornecer erros relativos à má formação do modelo.

Numa primeira abordagem à implementação do método, pensou-se em reunir o conjunto de condições que permitiriam a um autómato de pilha realizar os métodos exigidos pelo módulo Model. Depressa se apercebeu que essa primeira abordagem ao problema é desnecessariamente complexa, sendo até um pouco ingénua. A abordagem que se considerou de seguida melhor, foi a de definir um conjunto de condições que tornassem óbvio o resultado do método à partida. Isto poderá ser um sinal que o autómato foi definido de forma errada pelo utilizador. O método retorna quais as condições que não foram cumpridas através de uma mensagem de erro. Se forem satisfeitas todas as condições, então não produz nenhuma mensagem de erro.

Foram contempladas as 4 condições seguintes:

- Estado inicial válido

O estado inicial definido tem de pertencer ao conjunto de todos os estados do autómato.

- Símbolo inicial válido

O símbolo inicial da pilha tem de pertencer ao conjunto de todos os símbolos definidos para a pilha do autómato.

- Critério válido

Dependendo do critério de aceitação, uma condição tem de ser satisfeita. Caso o critério seja por estados de aceitação, então o conjunto dos estados de aceitação têm de pertencer a todos os estados do autómato. Se o critério for o de pilha vazia então o conjunto de estados de aceitação tem de ser vazio.

- Transições válidas

Para todas as transições serem válidas, todos os campos da transição têm de pertencer ao conjunto correspondente e não podem ser iguais ao conjunto vazio, exceto no caso da última componente da transição, que corresponde à substituição do topo da pilha, que pode ser vazio.

7.3 Accept

Assinatura do método:
method accept: word -> bool

O método recebe uma palavra e retorna true se for aceite pelo autómato e false caso contrário. É necessário explorar todas as transições que são feitas através do consumo de símbolos da palavra, bem como de todas as transições que podem ser feitas sem o consumo de nenhum símbolo (ε). A maneira mais fácil de modular este comportamento será uma adaptação do algoritmo de busca em largura. Dependendo do critério de aceitação a palavra pode ser aceite ou não.

Uma abordagem exequível seria explorar enquanto possível todas as transições que não consomem um símbolo. Desse modo, antes de se consumir um símbolo da palavra, já todos os estados do autómato possíveis de transição estariam explorados antes do consumo do símbolo seguinte. A este processo chama-se fecho transitivo. No fim desse fecho tentar-se-ia consumir o próximo símbolo da palavra e far-se-ia o fecho transitivo novamente. Após a palavra estar totalmente consumida e se realizar o último fecho, confirmar-se-ia se a palavra é aceite. Esta abordagem é, no entanto, um pouco ingénua. Caso se verifiquem ciclos no fecho transitivo, o algoritmo nunca termina.

Por esse motivo pensou-se numa abordagem nova. Verificar se ao fazer as transições se existem alterações na pilha. Caso não existam alterações, consome-se o próximo símbolo da palavra e o algoritmo prossegue. No entanto o problema mantém-se e é fácil para o leitor de o perceber. A pilha apesar de ser alterada não implica que não existam ciclos infinitos. Tal é observável em casos em que o autómato possui um ciclo de transições que não consomem símbolo, mas, no entanto, empilham símbolos da pilha.

A solução que se acabou por optar implica mais alguma complexidade. A ideia é efetuar as transições que não consomem símbolos, se existirem, seguidas pelas transições que consomem o próximo símbolo da palavra. Esta abordagem implica que o algoritmo a cada transição que faça tenha de memorizar o resto da palavra que falte consumir para cada caminho explorado. Caso a palavra seja totalmente consumida e se cumpra o critério o algoritmo termina e retorna que a palavra foi aceite, senão explora transições que não consumam símbolo até que não consiga.

Com esta solução tento contornar o máximo possível o problema da indecidibilidade que sabia que iria encontrar à partida. Uma maneira de evitar este método correr

indefinidamente seria definindo um limite através de alguma noção temporal, seja por tempo real (por exemplo 2s) ou por tempo lógico (por exemplo 500 iterações, como escolhido no JFLAP).

```
1.  method accept(w: word): bool =
2.
3.      let ist = representation.initialState in
4.      let trans = representation.transitions in
5.      let istack = [representation.initialStackSymbol] in
6.      let iconf = Set.make [(ist,istack,w)] in
7.      let acceptSts = representation.acceptStates in
8.      let criteria = representation.criteria in
9.
10.     let applyTransitions conf isy =
11.        self#nextConfs conf isy trans in
12.
13.     let eApplyTransitions conf =
14.        self#nextConfs conf epsilon trans in
15.
16.     let nextConfsZ (a,b,c) =
17.        match c with
18.           | []->
19.              let confs1 = eApplyTransitions (a,b) in
20.              Set.map ( fun (a,b) -> (a,b,[]) ) confs1
21.           | x::xs ->
22.              let confs1 = eApplyTransitions (a,b) in
23.              let confs2 = Set.map ( fun (a,b) -> (a,b,c) ) confs1 in
24.              let confs3 = applyTransitions (a,b) x in
25.              let confs4 = Set.map ( fun (a,b) -> (a,b,xs) ) confs3 in
26.              Set.union confs2 confs4
27.     in
28.
29.     let finished confs =
30.        if criteria then
31.           Set.exists ( fun (a,_,w) -> w = [] && Set.belongs a acceptSts ) confs
32.        else
33.           Set.exists ( fun (_,b,w) -> w = [] && b = [] ) confs
34.     in
35.
36.     let rec acceptX confs =
37.        if finished confs then
38.           true
39.        else
40.           let confs1 = Set.flatMap ( fun c -> nextConfsZ c ) confs in
41.           let interConfs = Set.inter confs confs1 in
42.           (* repeated conf detection *)
43.           if Set.size interConfs = Set.size confs1 then
44.              false
45.           else
46.              acceptX confs1
47.     in
48.
49. acceptX iconf
```

Figura 6: implementação do método accept da biblioteca OCamlFLAT

7.4 Generate

Assinatura do método:

method generate : int -> words

Apesar de um autómato ser um mecanismo reconhecedor de palavras e não gerador, considerou-se de grande utilidade para a perceção da linguagem reconhecida na sua totalidade.

O método recebe um valor inteiro e gera todas as palavras com esse comprimento. Depara-se com o mesmo problema relativamente à decidibilidade que o método anterior pelo que se tentou ter uma abordagem semelhante quanto ao fecho transitivo para contornar esta limitação. É semelhante ao anterior, mas mais fiel ao algoritmo de busca em largura.

7.5 nextConfs

Assinatura do método:

method nextConfs: state*word -> symbol -> transitions -> (state*word) Set.t

O método recebe 3 argumentos: um par ordenado que contém um estado do autómato e o conteúdo da pilha, o símbolo que é consumido da palavra e um conjunto de transições. O método retorna o conjunto de pares ordenados que contêm um estado do autómato e o conteúdo da pilha corresponde gerados pelas transições que foram efetuadas.

```
1.  method nextConfs (state,stack) sy trans: (state*word) Set.t =
2.    if stack = [] then
3.       Set.make []
4.    else
5.    let ts = select123 state (List.hd stack) sy trans in
6.       Set.map ( fun (_,_,_,d,e) -> (d,e@(List.tl stack)) ) ts
```

Figura 7: implementação do método nextConfs da biblioteca OCamlFLAT

7.6 getReachableStates

Assinatura do método:

method getReachableStates: states

O método retorna todos os estados que conseguem ser chegados a partir do estado inicial do autómato.

É executada uma busca em largura das transições do autómato a partir do estado inicial tendo apenas em consideração os estados que se visita e nenhum outro componente da transição como o estado da pilha ou o símbolo consumido. Os estados visitados são memorizados

de forma a que não se volte a explorar o mesmo estado caso este seja descoberto de novo. Isto é feito de modo a evitar ciclos que impossibilitem o término do método.

7.7 getProductiveStates

Assinatura do método:
  method getProductiveStates: states

O método retorna todos os estados que conseguem chegar a um estado de aceitação, caso o critério de aceitação de uma palavra do autómato seja esse. No caso do critério da pilha vazia, como não existem estados de aceitação, o algoritmo trata o conjunto de estados que conseguem desempilhar o símbolo inicial da pilha como se fossem estados de aceitação.

O método tem um comportamento idêntico ao anterior, mas efetua o algoritmo de forma inversa, ou seja, em vez de explorar a partir do estado inicial, explora a partir do conjunto de estados de aceitação.

7.8 getUsefulStates

Assinatura do método:
  method getUsefulStates: states

O método devolve o conjunto de todos os estados úteis. De modo a obter esse conjunto, realiza-se a interseção dos conjuntos dos estados resultantes dos métodos getProductiveStates e getReachableStates.

7.9 getUsefulTransitions, getUsefulSymbols e getUsefulStackSymbols

Assinatura dos métodos:
  method getUsefulTransitions: transitions
  method getUsefulSymbols: symbols
  method getUsefulStackSymbols: symbols

O método getUsefulTransitions filtra o conjunto de transições que usam os estados resultantes do método getUsefulStates. Os métodos getUsefulSymbols e getUsefulStackSymbols extraem de todas as transições o alfabeto do autómato e da pilha respetivamente.

7.10 clean

Assinatura do método:
method clean: model

O método cria uma representação do autómato equivalente ao definido, mas apenas com componentes úteis. Tal é possível através da definição de um novo autómato com os conjuntos resultantes dos métodos getUsefulStates, getUsefulTransitions, getUsefulSymbols e getUsefulStackSymbols.

7.11 isDeterministic

Assinatura do método:
method isDeterministic: bool

O método verifica se o autómato definido é determinista segundo a definição teórica. Devolve true caso seja determinista e false caso o contrário.

7.12 Outros métodos

Assinatura dos métodos:
method kind: string
method description: string
method name: string
method errors: string list
method handleErrors: unit
method toJSon: JSon.t
method tracing: unit
method checkExercise: Exercise.exercise -> bool
method checkExerciseFailures: Exercise.exercise -> words * words
method representation: t

O conjunto de métodos referidos fazem parte da assinatura do módulo Model e são herdados para o módulo PushdownAutomaton. Tirando o método toJSon que requereu converter

do formato interno para JSON, o trabalho envolvido foi inferior e são de fácil compreensão quando comparados com os anteriores, pelo que não irão ser falados como os restantes.

8

Apresentação da Interface Gráfica OFLAT

O módulo PushdownAutomaton foi criado com o intuito de ser inserido na ferramenta OFLAT que já foi previamente referida.

Neste capítulo vou apresentar a interface final da ferramenta e algumas das suas características, bem como uma comparação fundamentada entre trabalho realizado e o JFLAP, ferramenta que considero ser a melhor e mais completa disponível relativamente a autómatos de pilha.

A ferramenta OFLAT tem inúmeras funcionalidades às quais se teve de realizar um estudo prévio à realização deste trabalho. No entanto, apenas irão ser mencionadas as funcionalidades que tiveram de ser estendidas para a inserção do módulo PushdownAutomaton na mesma.

Houve um conjunto de objetivos que se tentou ter em consideração quando se criou a componente gráfica e que se tentaram manter na implementação nova, como ter uma interface que não crie ambiguidade, que seja simples e usar e que possa introduzir detalhes que sejam propícios ao desenvolvimento de intuições por parte do aluno. A tarefa nem sempre é fácil tendo em consideração que se tem de tentar manter a consistência de outras componentes do sistema.

8.1 Botão "Browse"

Esta funcionalidade permite carregar um autómato em formato JSON a partir do sistema operativo, fazendo com que uma representação gráfica do mesmo seja mostrada no ecrã, bem como um conjunto de botões que permitem realizar operações sobre esse mesmo autómato. Deste modo fica facilitado o carregamento do autómato para a aplicação, sendo que a definição de um autómato é das componentes que consome mais tempo.

A disposição dos nós do autómato, com a exceção do nó inicial, é aleatória dentro dum de valores predeterminados, sendo que cada vez que o mesmo for carregado irá ter uma disposição diferente.

Após o autómato ser carregado, irá estar presente no topo da página o tipo de autómato que está disposto e ainda um texto no fundo da janela com algumas propriedades de relevo do mesmo.

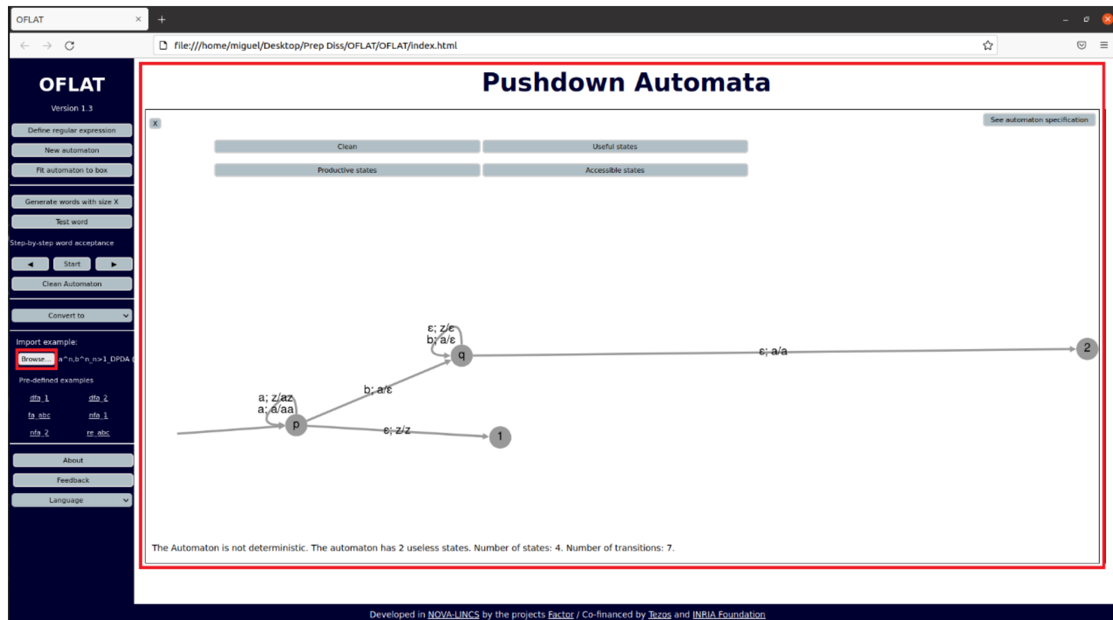

Figura 8: botão browse e a janela resultante ao carregamento do autómato

8.2 Definição de um autómato

Saber definir um autómato em formato JSON é fundamental para conseguir usar a ferramenta, visto que não existe outro tipo de suporte. Expondo de outra forma, o formato JSON é depois convertido para o formato interno dos autómatos.

Veja-se a definição de um autómato de pilha que reconhece a linguagem definida por $\{0n1n \mid n \geq 0\}$ (autómato apresentado no capítulo 6.3.1) em formato JSON:

```
1.  {
2.   kind : "pushdown automaton",
3.   description : "0n1n deterministic, n>0",
4.   name : "dpda-0n1n",
5.   inputAlphabet : ["0", "1"],
6.   stackAlphabet : ["A", "Z"],
7.   states : ["START", "A", "SUCCESS"],
8.   initialState : "START",
9.   initialStackSymbol : "Z",
10.  transitions : [
11.   ["A","A","1", "A", ""],
12.   ["A","Z","~", "SUCCESS", "Z"],
13.   ["START","Z","0", "START", "AZ"],
14.   ["START","A","0", "START", "AA"],
15.   ["START","A","1", "A", ""]
16.  ],
17.  acceptStates : ["SUCCESS"],
18.  criteria : "true"
19. }
```

Figura 9: autómato de pilha definido em JSON

Algumas observações podem ser concluídas após a análise do código anterior:

- Os campos *description* e *name* são opcionais e meramente de carácter informativo, mas de grande utilidade. A motivação por detrás destes campos consiste:
    - *description*: pode ser usado pelo utilizar para ajudar na perceção da linguagem reconhecida pelo autómato ou simplesmente deixar uma nota útil a outrem;
    - *name*: definir um nome ao autómato. Usualmente uma definição da linguagem aceite e o determinismo do autómato. Mas pode ser o que o utilizador bem entender.

O campo *kind* é herdado do módulo Model, mas, não só indica explicitamente o tipo de autómato a outros utilizadores de modo a que percebam erros a definir o autómato ou só meramente indicar que campos esperar de seguida, como também traz utilidade para outros métodos que façam a leitura do ficheiro JSON. Tal pode ser observado no subcapítulo seguinte, que se usa o campo para indicar à aplicação como definir e apresentar o autómato e outras funcionalidades na janela.

8.3 Botões internos ao autómato

Foi referido anteriormente que estariam presentes um conjunto de botões. Houve a preocupação de permitir invocar através deles as funções essenciais específicas dos autómatos de pilha.

Dito isto, esses botões permitem realizar as seguintes operações:

- fechar o autómato carregado;

- mostrar em formato JSON o autómato definido;
- mostrar uma representação do autómato limpa, ou seja, apenas com conteúdo produtivo;
- mostrar os estados úteis, produtivos e acessíveis.

Pushdown Automata

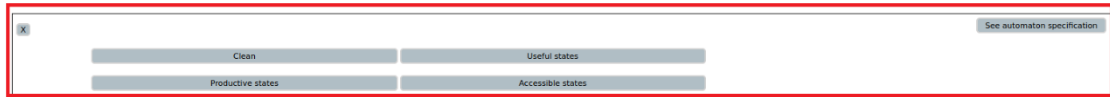

Figura 10: Botões internos à janela referentes ao autómato de pilha

8.3.1 Botão "Clean"

O botão divide a janela principal em duas do mesmo tamanho. O painel da esquerda apresenta o modelo com que se está a trabalhar, enquanto que o painel da direita apresenta o resultado da função, neste caso um novo autómato equivalente ao primeiro, mas apenas com o conteúdo resultante do método clean que foi falado anteriormente. Considerou-se de grande utilidade a possibilidade de ter ambos os autómatos presentes lado a lado.

Pushdown Automata

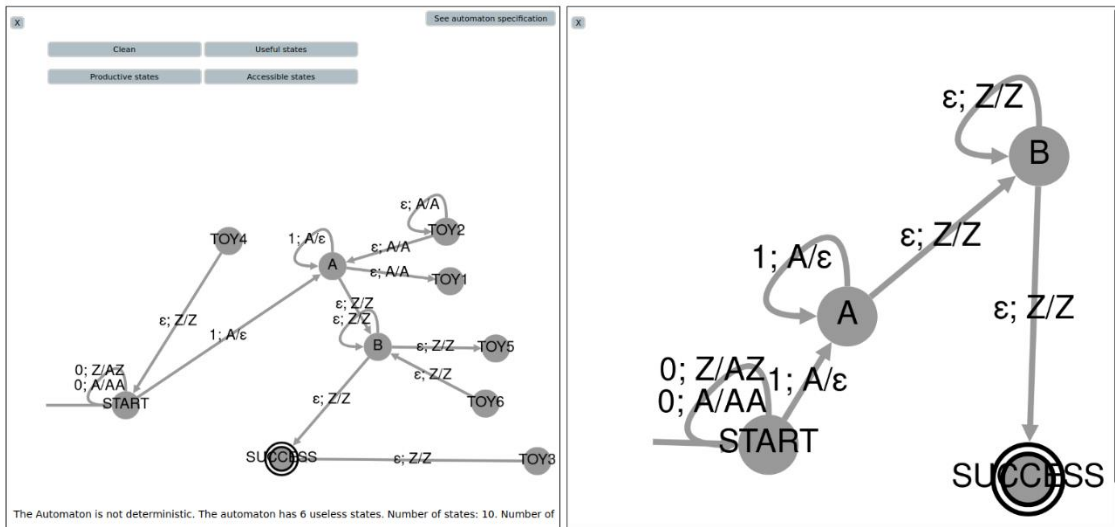

Figura 11: resultado da janela após o clique do botão "clean"

8.3.2 Botão "See Automaton Specification"

O botão é semelhante ao anterior, mas mostra a definição do primeiro autómato em formato JSON.

Pushdown Automata

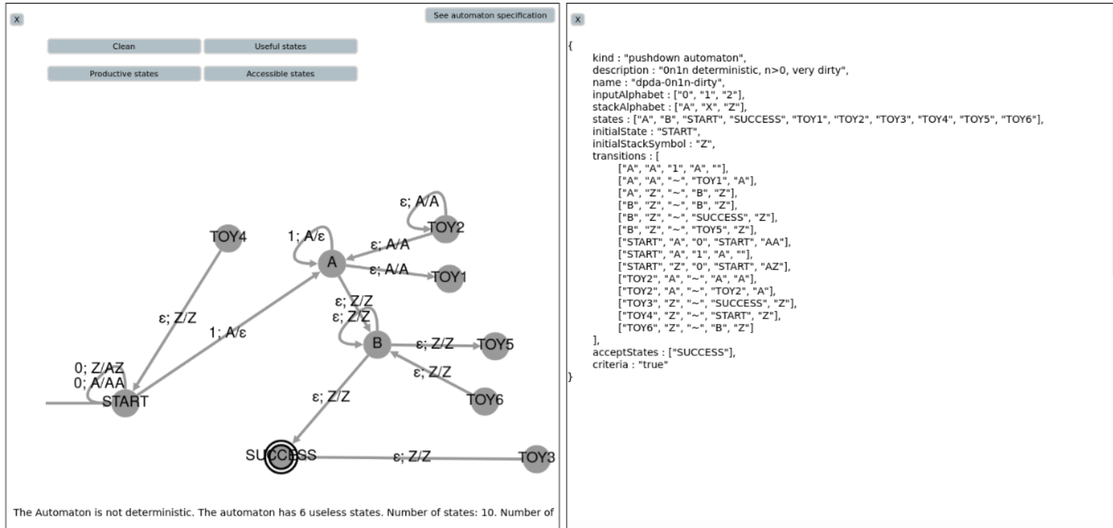

Figura 12: resultado da janela após o clique no botão "See automaton specification"

8.3.3 Botões "Useful States", "Accessible States" e "Productive States"

Estes botões pintam os nós do autómato presentes na janela mais à esquerda que correspondem à descrição do próprio botão. Cada característica tem uma cor associada.

8.4 Botão "Generate word with size x"

Ao contrário do que possa parecer devido ao nome, o método gera todas as palavras aceites pelo autómato até ao tamanho pretendido. Quando o botão é clicado aparece uma janela que pede para inserir o valor pretendido, e, de seguida, à semelhança dos botões Clean e de See Automaton Specification, este botão divide a janela principal e apresenta na segunda janela o conjunto de palavras aceites pelo autómato.

Pushdown Automata

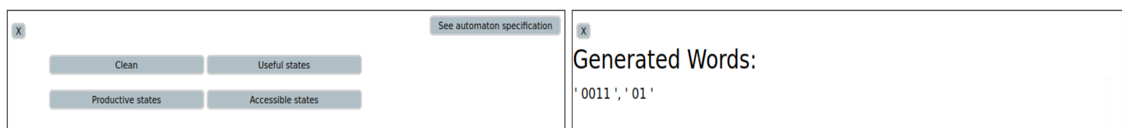

Figura 13: resultado da janela após o clique do botão "Generate word with size x"

8.5 Botões Step-by-step word acceptance

Este conjunto de botões permite ver o processo de aceitação duma palavra do autómato de forma animada e interativa. O utilizador clica no botão Start e insere a palavra que quer

avaliar. Seguidamente o utilizador pode usar ambas as setas para navegar na exploração do algoritmo de accept.

Os estados que se encontram nas configurações que estão a ser exploradas aparecem com uma cor distinta. No caso de o autómato ser determinista, o utilizador pode acompanhar ainda o estado da pilha no ecrã. Caso este seja não-determinista, como há múltiplos caminhos a serem explorados, o utilizador pode clicar nos estados coloridos e observar os conteúdos dos diferentes estados que a pilha pode tomar nesse momento.

Caso o algoritmo termine, se houver um caminho que a palavra é aceite, os estados do autómato em que tal acontece tomam a cor verde, caso contrário são assinalados com cor vermelha.

O facto de haver a possibilidade de retroceder no processo de aceitação da palavra é uma mais valia no ponto de vista da usabilidade. Facilita na compreensão do processo, contribuindo positivamente para as intuições do aluno.

O utilizador pode também acompanhar os símbolos já consumidos e os que faltam consumir da palavra introduzida.

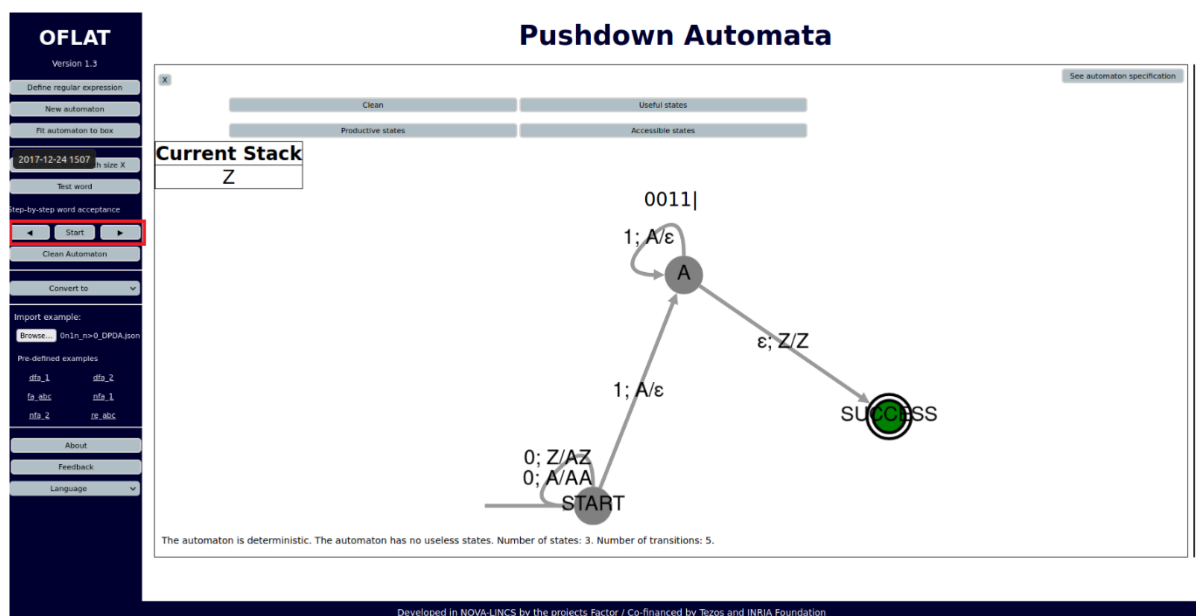

Figura 14: exemplo da palavra de entrada "0011" aceite por um autómato através dos botões de accept sequencial

8.6 Botão "Test word"

O botão faz o accept animado (equivalente ao botão mencionado no ponto anterior) automaticamente. Após o término do algoritmo, o utilizador pode usar as setas para navegar na

exploração do algoritmo e o autómato tem um comportamento semelhante ao accept incremental.

9

IMPLEMENTAÇÃO DA COMPONENTE GRÁFICA

No presente capítulo são explicados detalhes relativos à implementação das funcionalidades expostas no capítulo anterior. Vão ser seguidamente clarificados os métodos considerados mais importantes, vincando a ideia de que este documento não é um manual do utilizador. A exposição dos mesmos vai ser realizada em função do módulo onde se encontram.

Algumas funções auxiliares vão ser mencionadas e explicado o seu funcionamento. No entanto, como são de implementação simples, não se irá entrar em detalhe e o código não será exposto neste documento.

Sentiu-se a necessidade de criar um módulo novo chamado PushdownAutomatonGraphics que estende o módulo PushdownAutomaton, que acarreta todas as responsabilidades de representação gráfica do autómato usando Cytoscape.js.

Relembra-se que estes módulos vão ser inseridos num sistema já existente e que não é perfeito. Algumas partes do mesmo necessitariam de refatoração, mas o módulo novo foi implementado para ser compatível com o código já implementado. Todas as alterações e inserções de código são segundo o modelo MVC já referido. Para uma implementação mais correta, tendo em conta a expansão de funcionalidades no futuro, seria a de ter um controlador para cada modelo em vez de ter um único controlador a gerir todos. O controlador recebe chamadas de 2 tipos de listeners distintos:

- Os que estão presentes na coluna da esquerda da ferramenta, listeners estáticos cuja chamada é feita em "index.html"
- O que estão presentes no interior da janela que é permutada, sendo que os mesmos são botões dinâmicos, cujos listeners também são dinâmicos.

9.1 Controller

9.1.1 definePushdownAutomaton

Após selecionar o ficheiro JSON que contém a definição do autómato com o botão Browse, o campo *kind* é usado para perceber que módulos vão ser usados para apresentar o objeto no ecrã.

```
1.  let definePushdownAutomaton (pda: PushdownAutomatonGraphics.model) =
2.    StateVariables.changeCy1ToPDA();
3.
4.    HtmlPageClient.oneBox();
5.    HtmlPageClient.defineMainTitle (StateVariables.getCy1Type());
6.    HtmlPageClient.clearBox1();
7.    HtmlPageClient.disableButtons (StateVariables.getCy1Type());
8.    HtmlPageClient.enableAllButtons (StateVariables.getCy1Type());
9.    HtmlPageClient.putCyPDAButtons();
10.   Graphics.destroyGraph();
11.
12.   let actionShowStack =
13.   fun x ->
14.     let f = to_option x##.target in
15.       match f with
16.       | None -> ()
17.       | Some x ->
18.         let t = Js.Unsafe.coerce x in
19.         let d = t##data in
20.         let id = Js.to_string d##.id in
21.
22.         let confs = pda#getConfs in
23.         let position = ref 0 in
24.         HtmlPageClient.putCyPDAButtons();
25.         Set.iter( fun (a,b,c) ->
26.           if a = id then (
27.             let table = HtmlPageClient.createTable (Int.to_string !position)
28.               (id ^ (Int.to_string !position) ^ " | " ^ Util.word2str c) in
29.                 position := !position + 1;
30.                 eventAux table b;
31.                 HtmlPageClient.putTable table "buttonBox";
32.                 ()
33.           )
34.           else ()
35.         ) confs
36.   in
37.   let cyto = pda#startPDA("cy") in
38.   if pda#isDeterministic then (
39.   let table = HtmlPageClient.createTable "stack" "" in
40.     HtmlPageClient.putTable table "buttonBox"
41.   )
42.   else cyto##on (Js.string "click") (Js.string "node") actionShowStack;
43.
44.   StateVariables.changePDA pda;
45.
46.   !Listeners.defineInformationBoxListener();
47.   Graphics.fit()
```

Figura 15: implementação da função definePushdownAutomaton

Caso o valor desse campo seja "pushdown automaton", chama-se o método define-PushdownAutomaton presente no módulo Controller. Este método começa por realizar um

conjunto de tarefas relacionadas a destruição de qualquer eventual representação que esteja disposta no ecrã, a preservação do autómato a nível interno usando o módulo StateVariables e a apresentação dos botões correspondentes ao autómato na janela. Seguidamente, dá-se início à construção do autómato em Cytoscape com o método startPDA.

O método contém ainda a criação de um *listener* com o nome actionShowStack. Este *listener* está associado ao clique de um nó (estado) do container Cytoscape. Encarrega-se de ver o nome do mesmo e procurar todas as configurações que o estado possa ter e apresentar as pilhas correspondentes na janela. Este *listener* só é associado, no entanto, no caso de o autómato ser não-determinista. Caso seja determinista é mostrada a pilha logo de início.

9.1.2 Listeners

9.1.2.1 Conjunto paintAll...

Este conjunto de métodos já estava implementado no sistema para o caso de autómatos finitos. Foi apenas necessário verificar qual o tipo de autómato que estava carregado e modificar o comportamento em função do mesmo. Esta verificação estará presente em quase todos os métodos presentes no módulo Controller que irei apresentar neste capítulo pelo que deixará de ser mencionado.

9.1.2.2 cleanUselessListener

O método separa a janela principal em duas janelas e executa a função clean implementada no módulo PushdownAutomaton da biblioteca OCamlFLAT. O autómato resultante dessa função é então apresentado na janela da direita.

9.1.2.3 getCompleteAutomatonListener

A função trata de apresentar a representação interna do autómato convertida para o formato JSON. De forma idêntica à já vista em funções anteriores, a janela é dividida em duas e na segunda janela fica exposto o resultado da conversão do autómato para JSON.

9.1.3 getWords

A função é chamada quando o utilizador clica no botão que gera o conjunto de palavras aceites até ao comprimento pretendido. Como o autómato só gera as palavras de um determinado comprimento, optou-se por se definir uma função generateUntil que irá chamar a função generate da biblioteca com o argumento com comprimento pretendido de forma decrescente até o argumento ser 0.

```
1.  let getWords v =
2.        StateVariables.changeCy2ToInfo ();
3.        HtmlPageClient.twoBoxes();
4.        HtmlPageClient.putCy2Buttons ();
5.        let var =
6.          if (StateVariables.getCy1Type() = StateVariables.getAutomatonType())
    then
7.            (StateVariables.returnAutomata())#generateUntil v
8.          else if StateVariables.getCy1Type() = StateVariables.getPDAType() then
9.              let pda = StateVariables.returnPDA() in
10.             let rec generateUntil n =
11.                 if n = 0 then (
12.                     pda#generate n
13.                 )
14.                 else (
15.                     Set.union (generateUntil (n-1)) (pda#generate n)
16.                 )
17.             in
18.             generateUntil v
19.         else
20.           (StateVariables.returnRe())#generate v in
21.             HtmlPageClient.putWords var
```

Figura 16: implementação da função getWords

9.1.4 accept

A função é chamada quando o utilizador clica no botão de aceitação de uma palavra automática. No caso de ser determinista, a palavra que está a ser verificada é substituída no topo do ecrã. Tal é necessário por motivos informativos e para que após a animação esteja completa, caso o utilizador tente navegar no processo de aceitação através dos botões com setas pertencentes ao accept incremental consiga acompanhar que símbolos da palavra já foram consumidos.

9.1.5 Accept Iterativo

9.1.5.1 startStep

A função trata do processo do início do accept sequencial. Primeiramente é chamado o método startAccept do módulo PushdownAutomatonGraphics e, caso seja determinista, desenha-se na janela a tabela referente ao conteúdo inicial da pilha e coloca-se no centro da janela a palavra que se está a verificar se é aceite pelo autómato. É colocada uma barra no início da palavra de modo a informar o utilizador que nenhum símbolo foi consumido ainda. Por último, é necessário mudar no módulo StateVariables a palavra que estamos a verificar porque um processo de accept pode ser substituído pelo accept de uma nova palavra.

9.1.5.2 changeRE

A função apesar de possuir "RE" no nome de, significando "regular expression", apenas se destina a mudar a palavra que está a ser apresentada no centro da janela no processo de aceitação.

Optou-se por esse nome pois no código JavaScript e HTML existente essa divisão ficou com o nome "regExp". Todas as funções que fazem alterações a esse campo têm a referência "RE" e decidiu-se manter o estilo presente.

No entanto, a função simplesmente verifica se quando é efetuado algum possível consumo de símbolo, se de facto foi efetivamente realizado, e movimenta "|" pelos símbolos da palavra que se está a verificar durante o processo de accept iterativo, caso seja necessário.

```
1.  let rec subList list l =
2.      match l with
3.      | 0 -> []
4.      | _ -> (List.hd list)::(subList (List.tl list) (l-1))
5.
6.  let changeRE stack word =
7.      updateTable stack;
8.      let currRE = StateVariables.getSentence() in
9.      let strSplit = Util.word2str (subList (Util.str2word currRE)
10.                                 (String.length currRE - List.length word)) in
11.     let newString = strSplit ^ "|" ^ (Util.word2str word) in
12.     HtmlPageClient.replaceRE (newString)
```

Figura 17: implementação do método changeRE

9.1.5.3 nextStep

A função avança um passo no algoritmo de accept sequencial. Começa por chamar o método next do módulo PushdownAutomatonGraphics que veremos de seguida no capítulo seguinte. Caso o autómato seja determinista, chama-se a função auxiliar changeRE que atualiza a palavra o estado da palavra que é apresentada no ecrã, bem como a visualização do conteúdo da pilha.

9.1.5.4 backStep

A função é semelhante à anterior, pelo que a única diferença é a chamada do método back do módulo PushdownAutomatonGraphics que retrocede um passo no algoritmo de accept sequencial.

9.2 PushdownAutomatonGraphics

O módulo está encarregue de toda a componente gráfica e estende o módulo PushdownAutomaton da biblioteca OCamlFLAT.

A abordagem usada na conceção da ferramenta consistia em ter código OCaml a fazer chamadas de execução de porções de código escritos em JavaScript que manipulavam o comportamento dos grafos criados pela biblioteca Cytoscape.js.

No entanto, foi encontrada uma maneira alternativa que torna possível manipular a biblioteca escrevendo código apenas e unicamente em OCaml. Tal foi possível através da utilização de bindings que estão disponíveis na biblioteca Js_of_ocaml. Foi ainda encontrado um repositório com código [14] que permitia executar algumas tarefas básicas para a criação do grafo em Cytoscape.js. Este código não cobria, no entanto, todas as funcionalidades. Foi copiado na integra para o módulo PushdownAutomatonGraphics, onde posteriormente se adicionou e adaptou o código para que pudesse manipular o grafo da forma pretendida.

9.2.1 Adaptação do código ezjs_cytoscape

9.2.1.1 Criação do grafo manipulável no Cytoscape.js

Vão ser explicados alguns pontos fundamentais da biblioteca e, não só as alterações que foram feitas, bem como a motivação das mesmas.

De modo a que se consiga inicializar o grafo tem de ser chamada a função mk_graph. A função espera receber um conjunto de definições de apresentação do grafo, um layout, um conjunto de propriedades e uma *string* para ser o nome do container.

A apresentação utilizada será a definida em default_style. Através de bindings, consegue-se associar ao lado JavaScript um conjunto de características que podem afetar determinados grupos de objetos através do campo *selector*.

Os nós do grafo vão ter as características de *nodeStyle*, pois no *selector* definiu-se que ia afetar os objetos "node". Estes objetos vão apresentar o seu nome no centro.

Acontece algo semelhante aos arcos do grafo, definidos em *edgeSyle*. Este estilo vai afetar os objetos "edge" e vai conter a *label* no centro do arco. Estes arcos vão ter um triângulo na ponta de modo a indicar direção, pois a representação gráfica de um autómato é um grafo orientado. Têm ainda a propriedade de ser curvados com o estilo *Bezier*, que divide o ângulo de entrada e saída dum arco consoante o número de arcos que os liguem, evitando assim a sobreposição entre múltiplos arcos entre dois nós.

Existem ainda dois casos especiais relativos ao estilo do primeiro nó e dos nós que são estados de aceitação.

Como o Cytoscape.js não permite a criação de arcos sem ser entre dois nós, a ideia adotada antes da implementação do módulo passava na criação de um nó que estaria invisível e

que estaria ligado ao nó inicial do grafo. Com esse fim, foi utilizado o conceito de classe CSS, que se pode associar a um objeto. Para afetar um estilo a uma classe usa-se a notação de um ponto final antes do nome da classe. Tendo isto presente, os objetos de classe "HIDDEN", que correspondem na realidade apenas ao nó invisível ligado ao inicial, vão ter o campo visibility definido com o valor "hidden", que o tornam invisível.

Por último, como referido anteriormente, os nós finais do grafo têm também um estilo próprio. Usualmente usa-se como guia visual dois círculos, um dentro do outro, como indicação de que estamos presentes perante um estado de aceitação do grafo. De modo a replicar este standard, os nós finais têm um estilo semelhante, com borda dupla e pintada de preto.

```
1.  let default_style : Js.Unsafe.any style Js.t Js.js_array Js.t =
2.      let nodeStyle = Js.Unsafe.coerce @@ object%js
3.          val selector = Js.string "node"
4.          val style = Js.def (object%js
5.              val label = Js.string "data(id)"
6.              val textValign = Js.string "center";
7.              val textHalign = Js.string "center";
8.              end)
9.          end in
10.     let finalStyle = Js.Unsafe.coerce @@ object%js
11.         val selector = Js.string ".SUCCESS"
12.         val style = Js.def (object%js
13.             val borderWidth = Js.string "7px";
14.             val borderColor = Js.string "black";
15.             val borderStyle = Js.string "double";
16.             end)
17.         end in
18.     let hiddenStyle = Js.Unsafe.coerce @@ object%js
19.         val selector = Js.string ".HIDDEN"
20.         val style = Js.def (object%js
21.             val visibility = Js.string "hidden";
22.             end)
23.         end in
24.     let edgeStyle = Js.Unsafe.coerce @@ object%js
25.         val selector = Js.string "edge"
26.         val style = Js.def (object%js
27.             val textWrap = Js.string "wrap";
28.             val label = Js.string "data(label)";
29.             val curveStyle = Js.string "bezier";
30.             val targetArrowShape = Js.string "triangle";
31.             end)
32.         end in
33.     Js.array [| nodeStyle;edgeStyle;finalStyle;hiddenStyle |]
34.
35.   let default_layout : layout_options Js.t =
36.     object%js val name = Js.string "preset" end
37.
38.   let mk_graph ?(style=default_style) ?(layout=default_layout) ?(props=[])
39.    container_id =
40.     let container = Dom_html.getElementById container_id in
41.     let props = Js.array @@ Array.of_list props in
42.     let g : props Js.t = Js.Unsafe.obj [||] in
43.     g##.container := container;
44.     g##.elements := props;
45.     g##.style := style;
46.     g##.layout := layout;
47.     g
```

Figura 18: bindings implementados para a criação de um grafo

Para que o grafo seja inicializado com estas definições é necessário criar o elemento Cytoscape na componente JavaScript com o grafo definido com a função mk_graph vista anteriormente.

9.2.1.2 Adição de nós e arestas ao grafo

Os nós e arestas do grafo são objetos definidos por um conjunto de características e métodos. Para que sejam criados objetos com um comportamento semelhante do lado OCaml, têm de ser criadas classes com campos iguais. Grande parte desse trabalho está resolvido com a ajuda da biblioteca, que possui já definidos os tipos t e data.

Com os bindings, a criação dos objetos fica semelhante à criação em JavaScript. Existe uma função chamada node que recebe um id (importante para saber que nó estamos a aceder no futuro e se saber o nome do mesmo, uma posição, a sua classe, o tipo e ainda a legenda do mesmo, respetivamente. A criação começa com definir a data do nó e só depois outras informações relativas à apresentação do mesmo. A função devolve no fim este objeto.

No entanto, é preciso resolver dois problemas fulcrais para a obtenção de uma ferramenta sólida e apelativa, a posição e as classes que não estão implementadas. Numa fase inicial, todos os nós iniciavam-se sobrepostos na mesma posição, o que exigia o utilizador a mover todos os nós para a posição pretendida para conseguir analisar o autómato. A solução implementada para os outros objetos JavaScript passava por definir uma posição aleatória entre determinados valores, deste modo vários nós já se encontram numa posição útil e deslocam-se apenas os nós cujas posições sejam menos apelativas. Outra vantagem é a de haver uma leitura facilitada assim que o nó é carregado. Repare-se que existem duas exceções, as faladas anteriormente em que um nó corresponde ao nó escondido ou inicial, que têm a sua posição já definida.

A criação de arcos também é muito semelhante. A função recebe um id, a *string* do nó de origem, a *string* do nó de chegada e a sua legenda. Retorna posteriormente esse arco.

Por fim, para que os nós e arcos sejam adicionados ao grafo, existe o método add em JavaScript que espera receber um objeto. Duas funções já vinham implementadas pela biblioteca para fazer a adição de um nó ou arco, pelo que só foi necessário adicionar o argumento posição na adição do nó ao grafo.

9.2.2 Métodos internos

9.2.2.1 startPDA

O método começa por criar e mostrar na janela um grafo cujo o container é o argumento do método. De seguida, devido à maneira como o programa está implementado, é necessário criar a variável Cytoscape do lado do código JavaScript.

Por fim, inicia-se um método que irá realizar todo o desenho do autómato. A função começa por desenhar os nós e de seguida os arcos. Todos os arcos que originam e se destinam

aos mesmo nós têm a sua legenda concatenada de modo a se desenhar um só arco com as regras de todos. Caso um nó seja inicial ou final, têm as suas características diferentes pois são atribuídos às respetivas classes. É também criado um nó extra que será o nó invisível que será ligado ao nó correspondente ao estado inicial.

```
1.  method startPDA (name: string): cytoscape Js_of_ocaml.Js.t =
2.                      let graph1 = mk_graph name in
3.                      let cy = display graph1 in
4.
5.                      (if name = "cy" then
6.                          Js.Unsafe.global##.cy := cy
7.                      else
8.                          Js.Unsafe.global##.cy2 := cy);
9.
10.                     let rep = self#representation in
11.                     let states = rep.states in
12.                     let trans = rep.transitions in
13.                     let aStates = rep.acceptStates in
14.                     let criteria = rep.criteria in
15.                     let initSt = rep.initialState in
16.
17.                     defineStatesTrans cy states trans aStates criteria initSt;
18.                     add_node cy "hidden" ~pos:"h" "HIDDEN" "" "";
19.                     add_edge cy "hidden" initSt "";
20.                     cy
```

Figura 19: implementação do método startPDA

9.2.2.2 getConfs

Existem um conjunto de valores mutáveis que foram criados para realizar o accept sequencial. As mesmas irão ser faladas de seguida neste capítulo.

Tendo isto em consideração, o método devolve o conjunto de configurações que foram exploradas nesse passo do accept sequencial. Este método é útil para implementar os *listeners* dos autómatos não-deterministas, pois para tal é preciso saber todas as configurações que se encontram a ser exploradas.

9.2.2.3 areAllStatesUseful

O método devolve um true caso todos os seus estados sejam úteis, caso contrário devolve false.

9.2.2.4 cleanUselessStates

Este método mostra na segunda janela uma representação do autómato atual, mas só constituída por estados produtivos.

9.2.2.5 paintProductive, paintUseful e paintReachable

Os métodos pintam com a cor atribuída a essa característica os estados do autómato que sejam produtivos, úteis ou alcançáveis, respetivamente

9.2.2.6 numberStates e numberTransitions

Devolve o número de estados e o número de transições respetivamente.

9.2.2.7 Accept sequencial

Considerou-se bastante útil, não só a possibilidade de explorar as configurações seguintes do autómato durante o processo de aceitação de uma palavra, mas também de retroceder de modo a facilitar a compreensão do processo.

De modo a tornar isto possível, a solução a que se chegou foi a de criar um *array* onde se guarda todas as configurações exploradas cada vez que se tenta explorar mais uma vez. Para ser possível definir alguns comportamentos foi necessária a criação de alguns valores mutáveis auxiliares:

- lengthSteps acompanha o número total de passos que já foram feitos;
- position aponta a posição que está a ser apontada no *array* steps;
- ending indica se a procura terminou, quer seja por sucesso ou insucesso.

9.2.2.7.1 startStep

O método inicia o accept sequencial. São (re)inicializados todos os valores mutáveis e termina pintando o estado inicial da cor atribuída aos estados que estão a ser explorados.

9.2.2.7.2 getNextConfs

O método consiste numa mera cópia e adaptação do método accept já visto. A sua função é a de devolver todas as configurações que podem ser obtidas a partir duma configuração e dum conjunto de transições. É sobretudo utilizado pelo método seguinte.

9.2.2.7.3 next

O método devolve as configurações seguintes e pinta os estados existentes em função do que se sucede no decorrer do método.

Existem três linhas de execução distintas definidas por condições:

- A palavra foi aceite ou rejeitada, pelo que a exploração terminou e não se pode avançar. Neste caso devolve-se o conteúdo de todas as pilhas e de todas as palavras concatenadas num par ordenado. O motivo pelo qual se sucede é porque esta informação só é utilizada no caso de o autómato ser determinista, em que só existe um caminho a ser explorado.

```
13. method next: word*word =
14.   let rep = self#representation in
15.   let acceptSts = rep.acceptStates in
16.   let confs = steps.(position) in
17.   (* check if we explored this before *)
18.   if ending = true then (
19.     let stack = List.flatten (List.map (fun (_,b,_) -> b )
20.                                                             (Set.toList confs)) in
21.     let word = List.flatten (List.map (fun (_,_,c) -> c )
22.                                                             (Set.toList confs)) in
23.     (stack,word)
24.   )
25.   else if position < lengthSteps then (
26.     resetColor rep.states;
27.
28.     position <- position + 1;
29.     let confs = steps.(position) in
30.
31.     let (stack,wasAccepted) = paintExplored confs acceptSts rep.criteria in
32.     ending <- wasAccepted;
33.     let word = List.flatten (List.map (fun (_,_,c) -> c )
34.                                                             (Set.toList confs)) in
35.     (stack,word)
36.   )
37.   else ( (* begin exploring *)
38.     resetColor rep.states;
39.
40.     let trans = rep.transitions in
41.     let nextSteps = Set.flatMap (fun x -> self#getNextConfs x trans) confs in
42.
43.     let interConfs = Set.inter confs nextSteps in
44.     if Set.size interConfs = Set.size nextSteps then (
45.       let exploredStates = List.map (fun (a,_,_) -> a ) (Set.toList confs) in
46.       List.iter ( fun x -> paint x wrongFinalState) exploredStates;
47.
48.       ending <- true;
49.
50.       let stack1 = List.flatten (List.map (fun (_,b,_) -> b )
51.                                                             (Set.toList confs)) in
52.       let wordLeft = List.flatten (List.map (fun (_,_,c) -> c )
53.                                                             (Set.toList confs)) in
54.     (stack1,wordLeft)
55.     )
56.     else (
57.       position <- position + 1;
58.       steps.(position) <- nextSteps;
59.       lengthSteps <- lengthSteps+1;
60.
61.       let (newStack,wasAccepted) = paintExplored nextSteps acceptSts
62.                                                                    rep.criteria in
63.       ending <- wasAccepted;
64.       let newWordLeft = List.flatten (List.map (fun (_,_,c) -> c )
65.                                                         (Set.toList nextSteps)) in
66.       (newStack,newWordLeft)
```

Figura 20: implementação do método next

- A exploração das configurações já foi feita anteriormente, pelo que só é necessário retorná-las. Este caso acontece quando o utilizador usa o método back para

explorar passos anteriores do processo de aceitação. Como em todos os passos as configurações são guardadas, não existe a necessidade de serem exploradas de novo, pelo que apenas se devolve o valor existente no *array* Steps do passo em que se encontra. É também devolvida a concatenação de todas as palavras e pilhas pelos motivos já mencionados anteriormente.

- É, de facto, necessário explorar as configurações seguintes. Quando a posição em que nos encontramos é igual ao número de elementos então é necessário usar o método getNextConfs de modo a encontrar as configurações seguintes de cada configuração existente. Quando as configurações exploradas não são novas, o método chega à conclusão que a palavra não é aceite e pinta os estados que se encontram a ser explorados com a cor correspondente à palavra não aceite. Caso contrário, as configurações são adicionadas ao *array* que permite seguir o histórico de todas as configurações e são pintados os estados com a cor correspondente, quer seja de aceitação ou apenas indicação de que estão a ser explorados. É também devolvida a concatenação de todas as palavras e pilhas pelos motivos já mencionados anteriormente.

9.2.2.7.4 back

O método realiza o retrocesso no processo de aceitação de uma palavra. O método apenas retorna as configurações existentes em Steps na posição anterior à atual.

É também devolvida a concatenação de todas as palavras e pilhas pelos motivos já mencionados anteriormente.

9.2.2.8 resetColors

O método apaga todas as cores que foram pintadas do autómato, ficando com um aspeto de inicialização.

9.2.2.9 autoAccept

O método usa os métodos startAccept e next já referidos. Após iniciar o método que dá início ao processo de aceitação sequencial, o método next é repetidamente chamado enquanto for produtivo. De modo a que a animação do processo de accept seja percetível, foi necessário definir-se um intervalo de tempo de espera entre cada chamada. De forma a conseguir o devido efeito, optou-se por fazer a referida chamada com *threads*.

```
1.  method autoAccept (word:string): unit =
2.    let rec autoAcceptX () =
3.      if ending then
4.        Lwt.return ()
5.      else
6.        Lwt.bind (delay 100)
7.                 (fun () -> ignore self#next; autoAcceptX ())
8.    in
9.    self#startAccept word;
10.   ignore (autoAcceptX ())
```

Figura 21: implementação do método autoAccept

9.3 StateVariables

Este módulo permite que haja permanência dos autómatos no sistema e da palavra que está a tentar ser aceite por um autómato.

Tiveram de ser criadas duas representações distintas de autómatos para que as mesmas pudessem ser alteradas para os autómatos que fossem carregados ou alterados. O autómato da janela da esquerda tem o nome de pushdownAutomaton e o da direita tem o nome de pushdownAutomaton1.

Foram criados um conjunto de métodos que permitem devolver e alterar os autómatos e a palavra a ser aceite presentes no módulo.

9.4 HtmlPageClient

O módulo responsabiliza-se por toda a parte gráfica da janela que não seja relativa a alterações no autómato, como alterações da janela, botões, da palavra apresentada no centro e as tabelas usadas para representar a pilha e o seu conteúdo.

Houve a necessidade da criação de múltiplas funções de criação, eliminação e alteração da tabela. Foram ainda criadas funções que colocam a tabela na posição desejada. Quanto a esta última função, num plano inicial pensou-se que poderia ser útil, mas no final do trabalho as tabelas foram sempre adicionadas à mesma divisão HTML, mas considera-se útil porque permite que código seja mais extensível, o que trás utilidade.

Por último, algumas funções relativas ao texto informativo no fundo da página e de disposições de botões no ecrã foram criadas.

9.5 Considerações finais de capítulo

Faltaram ser mencionadas algumas funções auxiliares e alguns detalhes específicos de implementação de menor relevo, sendo que os que considerei mais importantes foram explicitados. De modo a inserir a totalidade do código desenvolvido no trabalho já existente, foram ainda necessários ajustes menores pelo código, sobretudo nos módulos Calling e Lang.

Os ajustes do módulo Calling foram relativos à manipulação do campo divisão "regExp" falada anteriormente e com a gestão do valor que lá se encontra.

As adaptações no módulo Lang foram relativas aos textos informativos, que permitem mudar a língua dos mesmos através da caixa de seleção presente na janela da ferramenta.

10

Avaliação e análise crítica

No presente capítulo vai-se comparar o trabalho desenvolvido com a ferramenta JFLAP e debater algumas escolhas de implementação com a esperança de, com algum otimismo, convencer o leitor a usar a ferramenta OFLAT. Irão ser expostos pontos considerados como mais valias e outros onde existem imperfeições ou, pelo menos, espaço para melhoria.

10.1 Comparação com JFLAP

O JFLAP é para muitos considerado como o pináculo das ferramentas de estudo de FLAT, tendo acumulado inúmeros prémios e menções por especialistas na área. Neste subcapítulo vão ser feitas algumas comparações, mas sobretudo diferenças, entre os novos módulos desenvolvidos e a ferramenta mencionada.

No JFLAP, visualmente o processo de aceitação de uma palavra por um autómato determinista ou não-determinista é idêntico para o utilizador, enquanto que no código que se implementou para estender a ferramenta OFLAT existem claras diferenças que vão ser expostas seguidamente.

10.1.1 Autómato determinista

No JFLAP, nos autómatos deterministas, o estado presente na configuração atual tem o seu valor *alpha* alterado para que apareça mais escuro. O estado da pilha e o consumo da palavra de entrada são apresentados no inferior da janela. Não existe opção de alterar o símbolo inicial da pilha Z, pelo que é preciso de ter isso em consideração quando se define as regras na aplicação porque não vem predefinido.

Na implementação realizada, o estado da pilha é apresentado quase no centro do ecrã, no canto superior esquerdo do *canvas*, logo debaixo dos botões internos. Podemos também observar a palavra de entrada no centro do ecrã e quanto falta consumir da mesma.

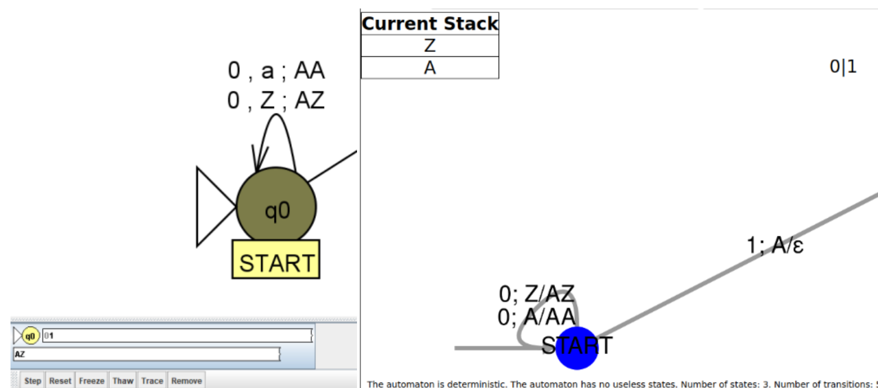

Figura 22: comparação da interface das duas ferramentas no caso de o autómato ser determinista

10.1.2 Autómato não-determinista

No JFLAP, a apresentação é muito semelhante aos autómatos deterministas. Toda as configurações possíveis estão apresentadas no mesmo sítio e os estados presentes nas mesmas estão destacados como referido no subcapítulo anterior.

No código desenvolvido, ao contrário do que acontece nos autómatos deterministas, nenhuma informação sobre a pilha ou a palavra de entrada é apresentada, apenas são apresentados a azul os estados presentes em configurações que estão a ser exploradas. O utilizador tem a possibilidade de clicar nesses estados, onde será apresentado tanto o conteúdo da pilha como o que resta consumir da palavra de entrada para o conteúdo da pilha correspondente.

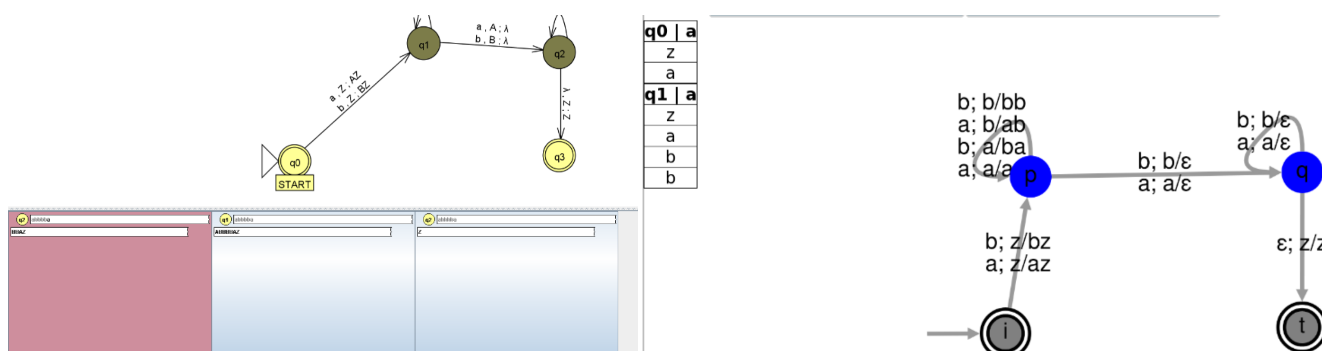

Figura 25: comparação da interface das duas ferramentas no caso de o autómato ser não-determinista

Figura 26: comparação da interface das duas ferramentas no caso de o autómato ser não-determinista

10.2 Avaliação e análise crítica

A elaboração dum projeto desta dimensão, sobretudo tendo em conta que muito do que foi desenvolvido foi para estender código já desenvolvido, abre sempre alas a melhoria ou, pelo menos, ao debate de algumas opções tomadas, quer pelo que se considere positivo ou negativo.

No código já existente relativo à parte gráfica havia funções com nomes que podem ser ambíguos e algumas repetições de código. Algumas vezes devido a isto, outras por culpa própria, as opções tomadas acabaram por levar também a algumas repetições de código. Vou falar dum exemplo em particular, mas é possível que existam outros locais onde algumas linhas pudessem ser extraídas para uma função que seria posteriormente chamada. Relembro que, ao contrário de línguas como Java, existem ainda poucas ferramentas para a análise e refatoração do código. Por exemplo, na função de accept e generate do módulo PushdownAutomaton e do método next do módulo PushdownAutomatonGraphics existe um padrão de comportamento que, apesar de ter havido algum esforço para minimizar as repetições, considera-se que ainda podia ser melhorado. Este processo seria facilitado através de ferramentas de refatoração que pudessem não só extrair como também indicar essas porções de código.

Existem também algumas limitações no código desenvolvido. O método next do módulo PushdownAutomatonGraphics só permite que se avance mil passos, pelo que se fosse chamado de novo daria um erro relativo ao limite do *array* usado para guardar as configurações, que foi inicializado com esse comprimento. Todos os métodos desta mesma classe também são executados sobre o autómato e não sobre uma representação equivalente resultante do método clean do mesmo. Se tal fosse feito, poupar-se-ia um mínimo de cálculo na maioria dos casos, mas em casos pontuais poderia ser muito vantajoso, sobretudo no caso de haver ciclos em estados que não são produtivos. Esta solução poderia garantir nestes casos o término do método accept e generate em casos onde a palavra não é aceite.

Durante a elaboração da dissertação, observou-se também um comportamento anómalo em casos pontuais de accept em autómatos não-deterministas. Se a palavra for totalmente consumida e não for aceite, todas as configurações que estão a ser exploradas sobre o mesmo estado terminam prematuramente. Foi desenvolvido um exemplo que expõe o problema.

Por último, é de salientar dois aspetos que justificam positivamente a distinção da ferramenta das restantes e que possa incentivar à sua adesão. O facto de se poder navegar não só em diante como também retroceder durante a análise dum processo de aceitação duma palavra. Para meu espanto durante o estudo do estado da arte, esta pequena diferença não é popular entre outras ferramentas e faz diferença na compreensão do comportamento do autómato. Esta funcionalidade não está disponível no JFLAP. Não só a funcionalidade referida,

como também a possibilidade de análise do conteúdo da pilha em autómatos não-deterministas através do clique no estado (nó). Esta ideia não estava presente em nenhuma das ferramentas estudadas e resolve um problema comum em todas elas, a incapacidade, ou pelos menos dificuldade, em analisar todos os caminhos que estão a ser explorados em simultâneo no processo de aceitação de uma palavra em autómatos mais complexos.

11

Conclusões e trabalho futuro

No capítulo irão ser apresentadas algumas conclusões e trabalho futuro do trabalho desenvolvido.

11.1 Conclusões

O projeto foi proposto com via desenvolver código em OCaml com o objetivo de promover a mesma em Portugal. Foi proposto desenvolver um conjunto de conceitos teóricos e a representação gráfica dos mesmos com o intuito de ajudar no estudo destes conceitos teóricos.

Apesar de se usar a biblioteca Cytoscape.js, implementada em JavaScript, com o uso de bindings conseguiu-se escrever todo o código em OCaml.

O código implementado não cobre todos os conceitos teóricos que podem ser feitos sobre os autómatos de pilha. No entanto, acabou por se implementar todas as funcionalidades que se consideraram mais importantes, apesar de não se ter produzido tanto como se tinha planeado e se desejava.

Durante a implementação da biblioteca surgiram problemas no limite da computabilidade derivados da indecidibilidade de alguns modelos matemáticos que se tentaram reproduzir na linguagem funcional OCaml. Tentou-se contornar o máximo possível estas barreiras sem comprometer demasiado a semelhança com esses modelos.

Concluindo, conseguiu-se implementar uma biblioteca com o conjunto de operações mais importantes sobre autómatos de pilha, bem como o suporte gráfico dos mesmos que ficará disponibilizado na ferramenta OFLAT. Espera-se que seja de utilidade para os estudantes durante o seu estudo.

11.2 Trabalho Futuro

Qualquer código suficientemente extenso abre sempre espaço para melhoria. Para além dos problemas mencionados no capítulo anterior, seria ótimo completar mais a biblioteca OCamlFLAT para ter mais funcionalidades relativas a autómatos de pilha, como a conversão para outros tipos de autómatos e gramáticas, bem como o processo inverso e a conversão do autómato para um equivalente, mas com um critério de aceitação diferente. Tendo em

consideração que um autómato de pilha não pode ser minimizado nem tornado determinista, seria interessante também converter o autómato a um equivalente, mas com os métodos accept e generate decidíveis.

A nível de funcionalidades, também se poderia adaptar o botão que fecha a janela para, caso haja um segundo autómato no painel mais à direita, em vez de ter o comportamento atual, tornar este segundo autómato no que está presente na primeira janela.

Apesar de difícil de usar, também seria útil permitir construir o autómato de forma livre, sem que fosse necessário carregar o autómato a partir de um ficheiro JSON.

Outra melhoria a ser estudada seria poder-se selecionar uma configuração de interesse através de um menu *dropdown* no autómato em vez da solução existente, que dependendo da complexidade do autómato pode dificultar o seu estudo.

Por último, e mais importante, introduzir um mecanismo temporal para limitar a execução de métodos não decidíveis, ou por uma noção lógica como um limite de número de passos ou usando tempo real, como já foi falado anteriormente.

BIBLIOGRAFIA

[1] A. Ravara, Lecture notes in Computational Theory. 2017.

[2] L. Monteiro, Lecture notes in Computational Theory. 2001.

[3] H. Lewis and C. Papadimitriou, Elements of the theory of computation, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1998.

[4] M. Sipser, Introduction to the theory of computation, 3rd ed. Boston, MA: Cengage Learning, 2013.

[5] P. Chakraborty, P. Saxena and C. Katti, "Fifty Years of Automata Simulation: A Re-view", ACM Inroads, vol. 2, no. 4, 2021. Available: https://doi.org/10.1145/2038876.2038893. [Accessed 23 December 2020].

[6] "Visual Paradigm Online - Suite of Powerful Tools", Online.visual-paradigm.com, 2021. [Online]. Available: https://online.visual-paradigm.com/pt/. [Accessed: 10- Feb- 2021].

[7] S. Sousa, Introdução às Linguagens Formais e as suas Gramáticas. 2021.

[8] Jochgem - Obra do próprio, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=5036988

[9] Shrodger - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/in-dex.php?curid=26443656

[10] R. Macedo, "OCamlFLAT no framework Ocsigen", FCT/UNL, 2020.

[11] D. Chuda, 2021, https://androidappsapk.co/detail-cmsimulator/

[12] T. White, 2021, http://jfast-fsm-sim.sourceforge.net/

[13] J. Gonçalves, "OCaml-Flat - An OCaml Toolkit for experimenting with formal languages theory", FCT/UNL, 2020.

[14] maxtori, 2020. ezjs_cytoscape/ezjs_cytoscape.ml at master · OCamlPro/ezjs_cytoscape. [online] GitHub. Available at: <https://github.com/OCamlPro/ezjs_cytoscape/blob/master/src/ezjs_cytoscape.ml> [Accessed 11 September 2021].

[15] Online Turing Machine Simulator. url: https://turingmachinesimulator.com/ [Accessed 20 August 2021].

[16] Ocaml.org. 2021. OCaml – OCaml. [online] Available at: <https://ocaml.org/> [Accessed 11 November 2021].

[17] Ocsigen.org. 2021. Js_of_ocaml. [online] Available at: <http://ocsigen.org/js_of_ocaml/latest/manual/overview> [Accessed 12 October 2021].

[18] Lourenço, J., 2021. GitHub - joaomlourenco/novathesis_word: Word templates for the Master and PhD thesis at FCT-NOVA (www.fct.unl.pt). [online] GitHub. Available at: <https://github.com/joaomlourenco/novathesis_word> [Accessed 13 November 2021].

[20] Sousa, S., 2021. Teoria da Computação (cod.14343). [online] Di.ubi.pt. Available at: <http://www.di.ubi.pt/~desousa/TC/tcomp.html> [Accessed 4 August 2021].

[21] Harry R. Lewis and Christos H. Papadimitriou. Elements of the Theory of Computation. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[22] J.E. Hopcroft, R. Motwani, and J.D. Ullman. Introduction to automata theory, languages, and computation. Pearson education, third edition, 560 pages. edition, 2006.

[23] Dexter Kozen. Automata and Computability. Springer-Verlag, New York, 1997.

[24] J.B. Almeida, M.J. Frade, J.S. Pinto, and S. Melo de Sousa. Rigorous Software Development, An Introduction to Program Verification, volume 103 of Undergraduate Topics in Computer Science. Springer-Verlag, first edition, 307 p. 52 illus. edition, 2011.

[25] Jflap.org. 2021. JFLAP. [online] Available at: <https://www.jflap.org/> [Accessed 23 October 2021].

[26] Rodger, S. and Finley, T., 2006. JFLAP-an interactive formal languages and automata package. Sudbury, MA: Jones and Bartlett.

[27] FAdo. 2021. FAdo | FAdo. [online] Available at: <https://fado.dcc.fc.up.pt/> [Accessed 28 April 2021].

[28] Teixeira, A., 2021. A cor enquanto elemento do projecto no design de produto. [online] Repositorio.ul.pt. Available at: <https://repositorio.ul.pt/handle/10451/22246> [Accessed 15 October 2021].

[29] Visualgo.net. 2021. VisuAlgo - visualising data structures and algorithms through animation. [online] Available at: <https://visualgo.net/en> [Accessed 4 September 2021].

[30] Wermelinger, M. and Dias, A., 2005. A Prolog toolkit for formal languages and automata.

[31] Pillay, N., 2009. Learning difficulties experienced by students in a course on formal languages and automata theory.

2021

Miguel Santos Araújo

Autómatos de Pilha em OCamlFLAT/OFLAT

NOVA

NOVA SCHOOL OF SCIENCE & TECHNOLOGY