NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

CARLOS MANUEL MENDES FREITAS
Licenciatura em Ciência e Engenharia Informática

AUTÓMATOS DE PILHA NAS FERRAMENTAS OCAML-FLAT/OFLAT

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
March, 2023

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

AUTÓMATOS DE PILHA NAS FERRAMENTAS OCAML-FLAT/OFLAT

CARLOS MANUEL MENDES FREITAS

Licenciatura em Ciência e Engenharia Informática

Orientador: Doutor Artur Miguel de Andrade Vieira Dias
Prof. Auxiliar, Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa

Júri

Presidente: Doutora Carla Maria Gonçalves Ferreira
Prof. Associada, Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa.

Arguente: Doutor Simão Melo Patrício de Sousa
Prof. Associado com Agregação, Faculdade de Engenharia da Universidade da Beira Interior

Vogal: Doutor Artur Miguel de Andrade Vieira Dias
Prof. Auxiliar, Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
March, 2023

Autómatos de Pilha nas ferramentas OCaml-FLAT/OFLAT

Copyright © Carlos Manuel Mendes Freitas, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.18) [12].

Acknowledgements

Agradeço ao meu orientador, Artur Miguel Dias, pela disponibilidade e paciência ao longo deste projeto. À minha família e amigos pelo apoio. E por fim, à Isabel Trancoso que esteve sempre do meu lado quando mais precisei.

Resumo

Os conceitos de Teoria de Linguagens Formais e Autómatos (FLAT - Formal Languages and Automata Theory) são abstratos, formais e matemáticos. Devido a isso, a aprendizagem desses tende a ser complicada para alguns alunos. Para apoiar o ensino dos conceitos FLAT, historicamente têm sido criadas múltiplas ferramentas informáticas.

Duas ferramentas que foram desenvolvidas na FCT/UNL nos últimos anos são a biblioteca OCaml-FLAT e a aplicação Web OFLAT. Ambas as ferramentas foram escritas em OCaml, tentando usar deliberadamente um estilo funcional declarativo, que em alguns casos se consegue aproximar dos formalismos teóricos que são ensinados aos alunos. As duas ferramentas estão escritas em OCaml e encontram-se em evolução e crescimento.

Esta dissertação visou adicionar autómatos de pilha ao conjunto de modelos suportados pelas ferramentas. Foram desenvolvidas as funcionalidades normais esperadas neste domínio como por exemplo: criar e editar autómatos, reconhecer palavras, gerar palavras, determinar propriedades, efetuar conversões de/para outros tipos de modelo.

Ao nível da biblioteca foi desafiante lidar com questões de não determinismo, estruturas cíclicas e casos de potencial não terminação. Na aplicação gráfica existiu a ambição de desenvolver uma interface gráfica criativa, intuitiva e pedagógica, oferecendo uma rica panóplia de operações. Destaque para o grande investimento colocado na animação da operação de reconhecimento de palavras para autómatos de pilha possivelmente não deterministas.

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

Palavras-chave: Teoria FLAT, Autómatos de Pilha, OCaml-FLAT, OFLAT, Ferramentas pedagógicas.

Abstract

The concepts of Theory of Formal Languages and Automata (FLAT - Formal Languages and Automata Theory) are abstract, formal and mathematical. Because of this, their learning tends to be complicated for some students. To support the teaching of FLAT concepts, historically, multiple computer tools have been created.

Two tools that were developed at FCT/UNL are the OCaml-FLAT library and the OFLAT Web application. Both tools were written in OCaml, deliberately trying to use a declarative functional style, which in some cases managed to approach the theoretical formalisms taught to the students. Both tools are written in OCaml and are evolving and growing.

This dissertation aimed to add pushdown automata to the set of models supported by the tools. The normal functionalities expected in this domain were developed, such as: creating and editing automata, recognizing words, generating words, determining properties, converting to/from other types of models.

At the library level it was challenging to deal with issues of non-determinism, cyclical structures and cases of potential non-termination. In the graphical application, there was the ambition to develop a creative, intuitive and pedagogical graphical interface, offering a rich range of operations. A highlight is the large investment made in animating the word recognition operation for possibly non-deterministic pushdown automata.

This document presents and discusses the result of this work.

Keywords: FLAT Theory, Pushdown Automata, OCaml-FLAT, OFLAT, Pedagogical Tools

Contents

List of Figures ix

1 Introdução 1

1.1 Enquadramento e motivações . . . 1

1.2 Objetivos . . . 1

1.3 Contribuições . . . 2

2 Ferramentas Pedagógicas para FLAT 3

2.1 JFLAP . . . 4

2.2 Automaton Simulator . . . 5

3 Programação Funcional em OCaml 7

4 Teoria FLAT 9

4.1 Hierarquia de Chomsky . . . 9

4.2 Autómatos pilha . . . 10

4.2.1 Definição Formal . . . 10

4.2.2 Determinismo e não determinismo . . . 13

4.2.3 Exemplos Práticos e Diagramas de Transição . . . 14

4.3 Gramáticas Independentes de Contexto . . . 20

5 OCaml-FLAT/OFLAT 21

5.1 Biblioteca OCaml-FLAT . . . 21

5.2 OFLAT . . . 21

5.3 js_of_ocaml . . . 22

5.4 Cytoscape.js . . . 22

6 Apresentação e discussão das novas funcionalidades 23

6.1 Layout para a criação, manipulação e visualização de Autómatos de Pilha 24

6.1.1 Layout criação e edição de um autómato pilha . . . 24

6.1.2 Layout operação de aceitação de uma palavra . . . 25

6.2 Operação *Accept* vantagens dos mecanismos implementados 27
6.3 Interface de geração de palavras reconhecidas pelo autómato 30
6.4 Operações de conversão de autómato de pilha para outros modelos e vice-versa 31
6.5 Operações complementares implementadas para Autómatos de Pilha . . 32
6.5.1 Operações de categorização dos estados do autómato 32
6.5.2 Informação autómato de pilha 34
6.5.3 Operação conversão aceitação por estados finais 34
6.5.4 Operação conversão aceitação por pilha vazia 34

7 OCAML-FLAT - Implementação 36
7.1 Tipos de dados criados para o autómato de pilha em OCaml 38
7.2 Operações implementadas 39
7.2.1 Função accept da biblioteca 39
7.2.2 Função auxiliares para a implementação do accept 40
7.2.3 Conversão de modo de aceitação por pilha vazia para estados finais 41
7.2.4 Conversão de modo de aceitação de estados finais para pilha vazia 42
7.2.5 Análise sintáctica, equivalência autómato finito 43
7.2.6 Operação de verificação de existência de não determinismo . . . 44
7.2.7 Conversões entre os tipos existentes na biblioteca 45
7.2.8 Operações de categorização de estados 48
7.2.9 Limpeza dos estados inúteis do autómato 49
7.3 Operações principais de utilização e criação da *Search Tree* 50
7.3.1 Tipos auxiliares definidos para o algoritmo de geração da *Search Tree* 51
7.3.2 Operação de criação da *Search Tree* 52
7.3.3 Tratamento da árvore de procura inicialmente obtida 54
7.3.4 Operação de procura do melhor caminho 55
7.4 Operação de geração de palavras reconhecidas pelo autómato de pilha . 57
7.5 Testes Unitários 57

8 OFLAT - Implementação 60
8.1 Interoperabilidade 60
8.1.1 Js_of_ocaml construção de objetos javascript 60
8.1.2 Js_of_ocaml bindings 61
8.2 Arquitectura MVC usada 62
8.2.1 Arquitetura geral para autómatos 62
8.3 PushdownAutomatonGraphics 65
8.4 Controlador 66

9 Avaliação e trabalho futuro 68
9.1 Avaliação 68
9.2 Conclusão 68

9.3 Trabalho Futuro 69

Bibliography **70**

List of Figures

2.1 Funcionalidades da ferramenta JFlap 4
2.2 Construção de um autómato de pilha em JFLAP 5
2.3 Interface Gráfica da ferramenta Automaton Simulator 6

4.1 Hierarquia de Chomsky 10
4.2 Autómato de Pilha 12
4.3 Tabela de transição do autómato A1 14
4.4 Exemplo de um autómato de pilha determinista que reconhece a gramática $\{0^n1^n : n > 0\}$ 15
4.5 Exemplo de um autómato de pilha não determinista que reconhece a gramática $\{0^n1^n : n > 0\}$ 16
4.6 Exemplo de um autómato de pilha que reconhece a Linguagem $\{wcw^-1 : w \in \{a,b\}^*\}$ 17
4.7 Tabela de transição do autómato $\{ww^-1 : w \in \{a,b\}^*\}$ 18
4.8 Exemplo de um autómato de pilha que reconhece a Linguagem $\{ww^-1 : w \in \{a,b\}^*\}$ 18
4.9 Tabela de transição do autómato 19
4.10 Exemplo de um autómato que pode entrar em ciclo infinito 19

5.1 OFLAT - Ferramenta Pedagógica para FLAT 22

6.1 OFLAT Interface gráfica accept 23
6.2 OFLAT Interface gráfica operações criação do autómato 24
6.3 OFLAT Interface gráfica criação de nova transição 24
6.4 OFLAT Interface gráfica operações de edição do autómato 25
6.5 Interface gráfica menu da operação accept 25
6.6 A subfigure 26
6.7 A subfigure 26
6.8 Configurações de um estado/configuração tabela 26
6.9 OFLAT Interface gráfica accept passo número 1 28

6.10 OFLAT Interface gráfica accept passo número 2 28
6.11 OFLAT Interface gráfica accept passo número 3 29
6.12 OFLAT Interface gráfica accept passo número 6 29
6.13 OFLAT Interface gráfica accept passo número 10 30
6.14 Interface da geração de palavras 30
6.15 Exemplo de geração de palavras de um autómato de pilha 31
6.16 Menu de conversão entre modelos da ferramenta 31
6.17 Conversão de autómato de pilha para expressão regular 32
6.18 Operação de identificação de estados produtivos 32
6.19 Operação de identificação de estados acessíveis 33
6.20 Operação de identificação de estados úteis 33
6.21 Operação de limpeza do autómato 33
6.22 Informação do autómato de pilha 34
6.23 Operação de conversão do autómato em aceitação por estados finais 35
6.24 Operação de conversão do autómato em aceitação por pilha vazia 35

7.1 Autómato de pilha com transições epsilon problemáticas 37
7.2 Conversão de CFG para PDA 47

8.1 Diagrama de classes OFLAT e OCAML-FLAT *controllers* e *graphics* 63
8.2 Diagrama de classes OFLAT controlador e graphics para autómatos 64

1
Introdução

1.1 Enquadramento e motivações

O estudo de Linguagens Formais e de Autómatos é caracterizado por ter um caráter matemático, muito detalhado e formal, o que dificulta a sua aprendizagem para muitos alunos. Neste sentido, existiu a motivação para desenvolver ferramentas pedagógicas que incentivassem os alunos a aprender mais sobre estes temas, através de uma experiência visual apelativa e intuitiva, com exemplos concretos e a realização de exercícios práticos, para complementar os métodos de ensino usados atualmente.

As ferramentas OCaml-FLAT e OFLAT estão em crescimento e evolução. Este trabalho corresponde a mais uma contribuição para ampliar e melhorar essas ferramentas.

1.2 Objetivos

No âmbito desta dissertação, foram estendidas as duas ferramentas já existentes: a biblioteca OCaml-FLAT e a aplicação *web* OFLAT. Foi adicionado suporte para autómatos de pilha. As ferramentas já suportavam autómatos finitos, expressões regulares, gramáticas livres de contexto, parsing LL(1) e parsing LR(1).

Na biblioteca, as maiores dificuldades relacionaram-se com a gestão do não determinismo e com o tratamento do infinito. O infinito não coloca problemas nas definições matemáticas, mas é problemático num contexto computacional: por exemplo, ciclos nos autómatos criam problemas de potencial não terminação.

A interface gráfica web do OFLAT assume procurar ser intuitiva e pedagógica, e estes também foram objetivos prosseguidos neste projeto.

Um objetivo secundário foi a racionalização de parte do código existente e da descoberta de algumas oportunidades de factorização, por exemplo no código dos autómatos finitos e dos autómatos de pilha.

1.3 Contribuições

Novas versões do OCaml-FLAT e OFLAT com suporte adicionado para autómatos de pilha. As lições aprendidas com este trabalho são descritas nesta dissertação.

A versão corrente da aplicação *web* OFLAT encontra-se aqui disponível: http://ctp.di.fct.unl.pt/FACTOR/OFLAT/.

2

Ferramentas Pedagógicas para FLAT

De maneira introdutória, serão descritas algumas ferramentas pedagógicas para FLAT, que ajudam a contextualizar o trabalho que foi realizado.

Atualmente, qualquer currículo na área da Engenharia Informática inclui, como disciplina essencial, Teoria de Linguagens Formais e Autómatos. Pelo caráter abstrato dos conceitos, ajuda ensinar esta Teoria com o apoio de ferramentas pedagógicas, idealmente gráficas e interativas [10].

As ferramentas para FLAT orientadas para aos autómatos, podem ser classificadas em duas grandes categorias: simuladores de autómato baseados em texto e simuladores de autómato baseados em interfaces gráficas [1]. As primeiras envolvem a especificação de um modelo através de texto estruturado, que tem de ser processado usando um parser. As segundas utilizam uma interface gráfica que permite criar, editar e usar um autómato para reconhecer palavras [1].

As diferentes ferramentas pedagógicas existentes também podem suportar apenas um tipo de autómato ou suportar vários tipos de autómatos, como autómatos finitos, autómatos de pilha e máquinas de Turing [2].

Também existem ferramentas pedagógicas que lidam com expressões regulares e gramáticas e, nesse caso, é normal estarem disponíveis operações de conversão para autómatos e vice-versa.

De seguida, apresentam-se algumas ferramentas pedagógicas e classificadas de acordo com os indicadores acima descritos, i.e., se são ferramentas baseadas em texto ou ferramentas baseadas em interfaces gráficas; se implementam apenas um tipo de autómato ou vários; e se suportam expressões regulares e gramáticas.

Das muitas ferramentas disponíveis como DEM, Automaton Simulator, JFLAP, JCT, Minerva, IPAA, entre outras, vai ser dado destaque a duas: o JFLAP e o Automaton Simulator. O JFLAP é muito completo e suporta todo o tipo de modelos, incluindo autómatos de pilha; o Automaton Simulator é especializado em autómatos finitos e autómatos de pilha.

2.1 JFLAP

O JFLAP é uma ferramenta *open source*, desenvolvida em Java, que implementa uma interface gráfica para as mais variadas operações e conceitos de Linguagens Formais e Teoria de Autómatos [14]. O JFLAP destaca-se por ser a ferramenta mais popular e isso acontece porque abrange grande parte dos tópicos referentes a estes temas e porque têm uma grande qualidade.

O JFLAP permite a definição de expressões regulares, gramáticas livres de contexto, gramáticas sensíveis ao contexto, assim como autómatos de pilha, máquinas de Turing, etc. Para além disso, o JFLAP também possibilita a conversão entre gramáticas e autómatos, bem como a observação do funcionamento dos autómatos aquando da aceitação ou rejeição de uma palavra.

A variedade de conceitos suportados pelo JFLAP está patente na figura (2.1) [6].

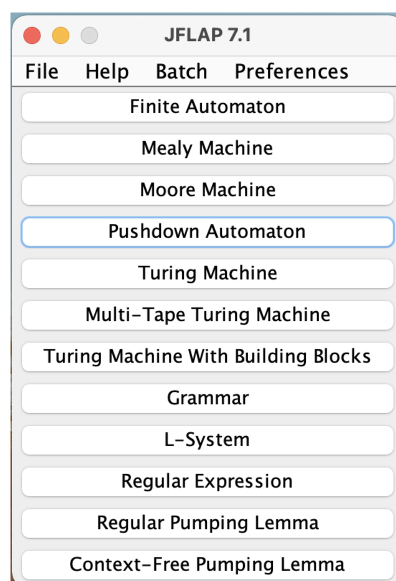

Figure 2.1: Funcionalidades da ferramenta JFlap

Como exemplo, ao selecionar *Pushdown Automaton* é gerada a interface que consta na figura 2.2. Seguidamente, serão descritas as funcionalidades referentes à construção deste tipo de autómato, nesta ferramenta.

As três funcionalidades mais relevantes do JFLAP na construção de um autómato de pilha foram numeradas por (1), (2) e (3) na figura 2.2. A funcionalidade (1) corresponde a um painel que demonstra a construção do autómato de pilha através de círculos e setas que expressam os estados (q_1, q_2 e q_3) e as transições entre estes. Neste painel também pode ser observado um círculo a verde escuro, o que significa que o autómato se encontra nesse estado (q_1). Por sua vez, a funcionalidade (2), que opera em conjunto com a (1), é representado por um painel no canto inferior esquerdo da página, o qual permite observar, interativamente, o funcionamento do autómato, ou seja, contém a palavra da fita de leitura e as operações para dar seguimento à iteração, assim como a representação

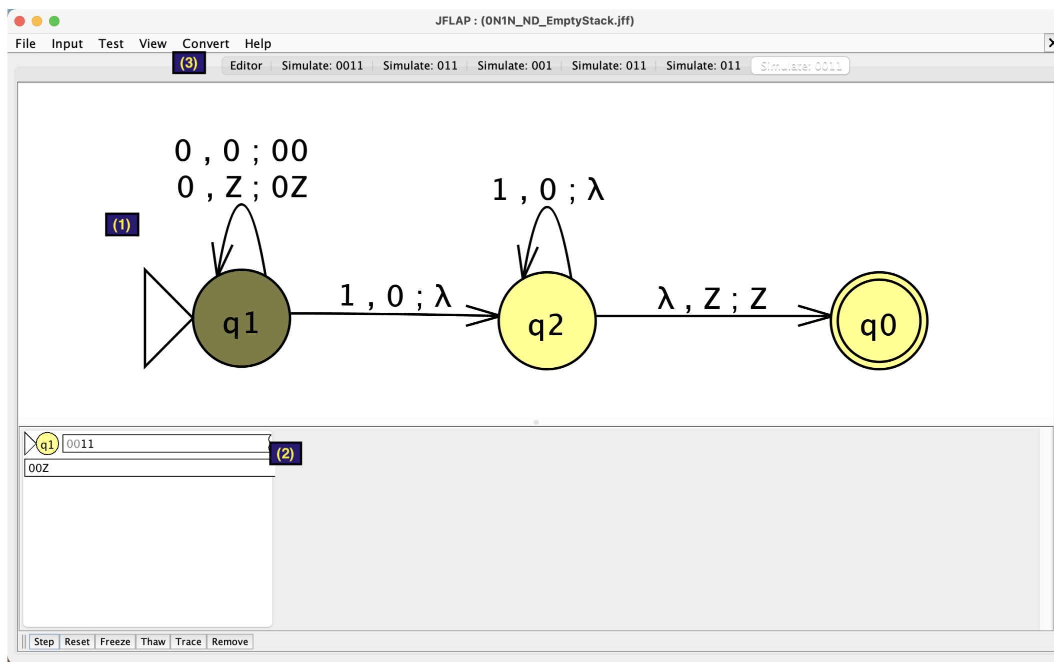

Figure 2.2: Construção de um autómato de pilha em JFLAP

da configuração do autómato em cada estado (estado corrente, palavra contida na fita de leitura e o estado da pilha). Já em (3), existe a opção de converter o autómato de pilha numa gramática livre de contexto.

Como fora referido anteriormente, embora a ferramenta seja, de facto, muito completa, no sentido de ser passível de ser utilizada com vários tipos de modelos, a sua interface gráfica é pouco intuitiva em alguns aspetos, por exemplo na funcionalidade de conversão de autómato de pilha para gramática (3).

2.2 Automaton Simulator

O Automaton Simulator, consiste numa ferramenta que dá apoio à construção, através de uma interface gráfica, de três tipos de autómatos: autómatos finitos deterministas, não deterministas e autómatos de pilha. Esta ferramenta tem a particularidade de ser executada no *browser*, facilitando o acesso a ela [7].

Em relação à ferramenta JFLAP, é também possível, com o Automaton Simulator, observar o estado do autómato ao longo do processo de aceitação ou rejeição de cada *input*. Mas a interface do Automaton Simulator é mais simples. Por outro lado, só contém um subconjunto das operações implementadas pelo JFLAP. Mesmo assim, por não existir documentação e por não existirem explicações para os ícones existentes, o uso do Automaton Simulator é um pouco difícil de compreender e manejar.

Esta ferramenta tem 3 partes gráficas principais. A parte maior permite a construção do autómato através de cliques no ecrã, para criar os estados, e arrastando o rato entre estados para criar transições. Do lado esquerdo da interface (contendo 2 caixas de texto),

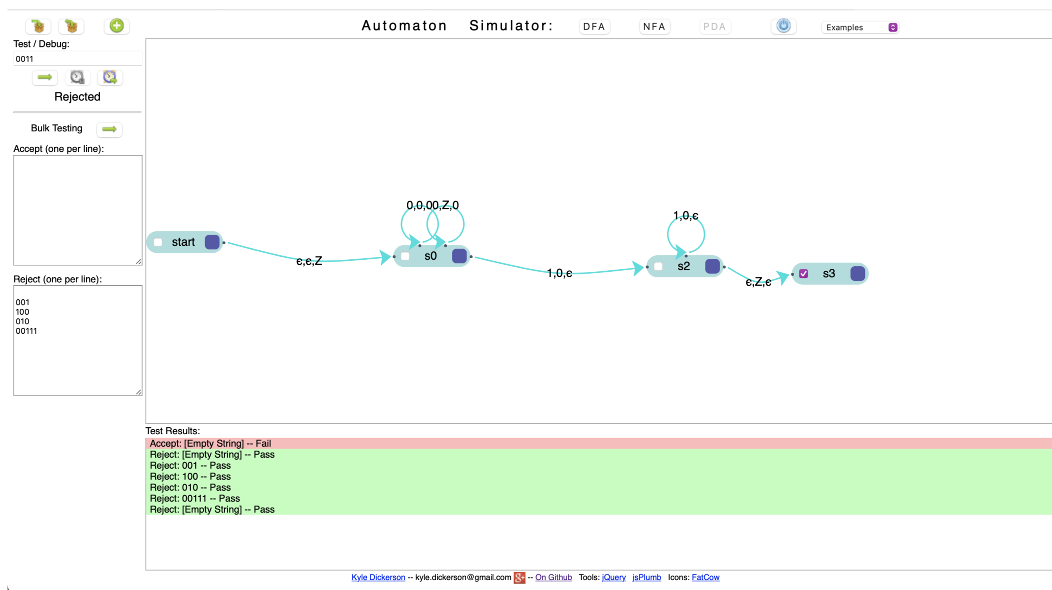

Figure 2.3: Interface Gráfica da ferramenta Automaton Simulator

existe a possibilidade de testar palavras, sendo possível correr várias de uma vez, com 2 categorias: palavras que devem ser ou não aceites pelo o autómato, aparecendo os resultados na parte inferior da ferramenta, um para cada teste realizado. Também existe a possibilidade de introduzir uma palavra e observar o seu reconhecimento passo a passo, utilizando os botões a caixa de texto situada no canto superior esquerdo.

3

Programação Funcional em OCaml

Ao usar diferentes paradigmas, é possível olhar para o mesmo problema de formas diferentes. É mais simples e intuitivo em certos problemas uma determinada forma de pensar. Em alguns casos, através de uma nova perspetiva, pensar sobre um problema e resolvê-lo, torna-se mais eficaz usando a ferramenta certa. Para o tema desta dissertação, devido à sua natureza matemática, uma boa solução é usar o paradigma funcional e, idealmente, tentar resolver os problemas de forma declarativa, tentando seguir de forma próxima as definições matemáticas dos conceitos envolvidos.

A programação funcional pura tem como base o uso de funções, sendo que os resultados duma função são sempre derivados dos argumentos passados, e não existem quaisquer tipos de efeitos secundários da chamada da função. [9]. Adicionalmente, as variáveis representam sempre valores constantes. Estas duas propriedades eliminam uma vasta classe de erros no código. [9]. As funções podem ser chamadas múltiplas vezes e por diferentes ordens, obtendo-se sempre o mesmo resultado, uma vez que dependem apenas dos argumentos passados, sem mais contexto. [9].

A programação funcional tem esta grande vantagem: não é necessário haver um esforço acrescido para manter um mapa mental que representa o estado do programa em cada momento. As características do paradigma permitem usar um estilo de programação declarativo, em que resolvem os problemas usando as propriedades do problema, e não inventando algoritmos com estado e detalhes operacionais.

No entanto, estas propriedades não são exclusivas ao paradigma funcional. Há outros paradigmas com estas propriedades, como por exemplo o paradigma de programação em lógica.

Usar uma linguagem como o OCaml ajuda bastante. O tema desta dissertação introduz alguns problemas técnicos difíceis, nomeadamente, lidar com não determinismo e ciclos nos autómatos. Além disso, o processamento dos autómatos é orientado pela estrutura da sua representação, situação em que o emparelhamento de padrões do OCaml ajuda a exprimir soluções legíveis.

Obter uma grande velocidade de execução não é uma prioridade neste projeto. Os exemplos pedagógicos com que os alunos lidam têm naturalmente uma dimensão pequena

ou média. É verdade que algumas soluções declarativas não são as mais eficientes, mas têm provado ser amplamente suficientes para as necessidades do projeto.

4

Teoria FLAT

O ramo da matemática que estuda linguagens formais e computabilidade costuma ser designado por teoria FLAT. Esta teoria identifica diferentes categorias de autómatos e/ou gramáticas com capacidades distintas ao nível da complexidade estrutural das linguagens que se estão a definir. Uma linguagem corresponde a um conjunto de palavras, formadas através de um alfabeto de símbolos [4]. Na teoria FLAT, uma linguagem é descrita tipicamente de duas formas alternativas: usando geração, através de uma gramática formal, ou usando reconhecimento, através de um autómato.

De acordo com Chomsky [5], o estudo de uma linguagem é realizado através de um conjunto de regras, designado por gramática [3]. Uma gramática formal G é composta pelo seguinte quádruplo, $G = (V, T, S, P)$, sendo que V corresponde ao conjunto de variáveis, T ao conjunto de símbolos terminais, S é um símbolo especial que corresponde à variável inicial e P define o conjunto de regras de produção [11].

Outra forma de descrever uma linguagem é utilizando o reconhecimento, através de um autómato [8]. Um autómato é um modelo abstrato de um computador, pelo que todos os autómatos possuem várias funcionalidades comuns, como por exemplo, ler o *input* da esquerda para a direita, um símbolo de cada vez; produzir um *output*; pode ou não conter memória temporária, na qual consegue ler, escrever e alterar símbolos que nela estão contidos; e, possuir um conjunto de estados e uma tabela de transições, sendo que esta última define, para cada símbolo e estado atual, o próximo estado (a memória pode ser alterada aquando da mudança de estados)[11].

Posto isto, é possível classificar as gramáticas formais em quatro categorias com um poder computacional diferente, de acordo com a chamada na Hierarquia de Chomsky.

4.1 Hierarquia de Chomsky

A hierarquia de Chomsky, formulada por Noam Chomsky em 1959, fundador da Teoria das Linguagens Formais, identifica os diferentes poderes expressivos (capacidade de exprimir computações mais ou menos complexas) de diferentes classes de gramáticas formais e os seus autómatos correspondentes. Os poderes expressivos são divididos em

quatro classes definidas hierarquicamente, nas quais, em cada nível, se obtém uma maior expressividade que o nível anterior. [11].

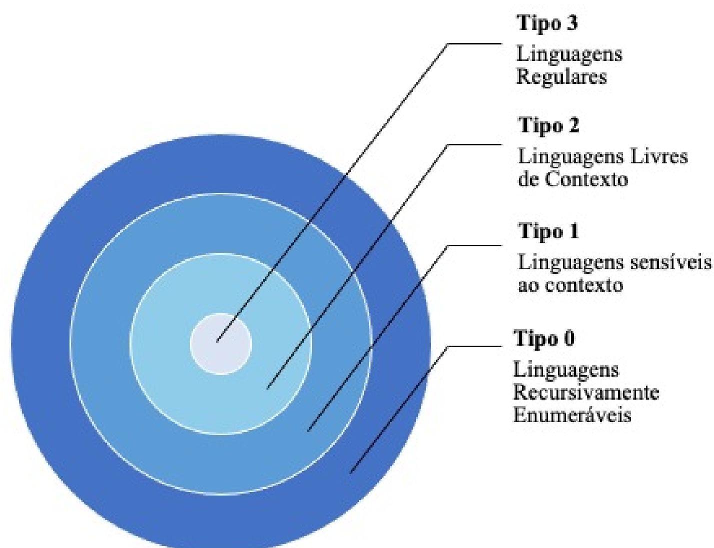

Figure 4.1: Hierarquia de Chomsky

Assim, de um forma ascendente de expressividade, representada pela figura 4.1, na classe de tipo 3 encontram-se linguagens regulares, reconhecidas por autómatos finitos; o tipo 2 inclui linguagens independentes de contexto, reconhecidas por autómatos de pilha; no tipo 1 situam-se linguagens sensíveis ao contexto, identificadas por autómatos linearmente limitados. E finalmente, a classe de tipo 0 é constituída por linguagens recursivamente enumeráveis, reconhecidas por máquinas de Turing, com poder computacional equivalente aos computadores.

Este trabalho centrar-se-á nos autómatos de pilha que se encontram no tipo 2 desta hierarquia.

4.2 Autómatos pilha

Uma vez descrita a Hierarquia de Chomsky e as várias classes que distinguem os diferentes tipos de linguagens, neste tópico serão abordados os autómatos de pilha (AP), pertencentes à classe de tipo 2, linguagens livres de contexto (LIC, e em inglês, context-free languages). Desta forma, definir-se-á o conceito de autómato de pilha, as suas funcionalidades e o seu propósito, para além de se descreverem outros conceitos, como função, relação, determinismo e pilha, que são relevantes para o melhor entendimento do tema.

4.2.1 Definição Formal

Posto isto, numa primeira instância, serão analisados os autómatos de pilha não deterministas dado que estes correspondem à forma geral deste tipo de autómatos, possuindo uma

correspondência direta com as LIC. A sua definição formal é apresentada de acordo com o sétuplo seguinte:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$$

,

sendo que,

Q representa o conjunto finito de estados de M,

Σ indica o conjunto finito das ações de M (alfabeto),

Γ corresponde ao conjunto finito de símbolos da pilha,

δ é a relação de transição de M, na qual, $Q \times (\Sigma \cup \lambda) \times \Gamma \times Q \times \Gamma^*$

$q_0 \in Q$ representa o estado inicial de M,

$z \in \Gamma$ é o símbolo inicial da pilha e

$F \subseteq Q$ corresponde ao conjunto dos estados finais.

Para operacionalizar a definição anterior no reconhecimento de uma palavra concreta, é conveniente ter uma noção de configuração. O AP percorre diferentes configurações até atingir uma que permita determinar se a palavra é aceite ou rejeitada, para além de que existe a possibilidade de ciclo infinito e, nesse caso, o autómato não consegue decidir sobre a aceitação da palavra.

A noção de configuração pode ser definida por:

$$(q, w, u)$$

,

em que,

q indica o estado atual da unidade de controlo,

w, o conjunto de símbolos da fita de leitura, por consumir e

u, o conteúdo da pilha.

Deste modo, a transição entre estados pode ser representada pela seguinte expressão:

$$(q_1, aw, bx) \vdash (q_2, w, yx)$$

,

se e só se,

$$(q_1, a, b, q_2, y) \in \delta.$$

No que diz respeito à leitura do *input*, a fita de leitura pode ser separada em "cabeça" e "corpo", sendo representada na primeira expressão por a e w, respetivamente. Em cada iteração, a "cabeça" é consumida e o "corpo" segue para o estado seguinte para ser processado nas restantes iterações. Ou seja, a transição do estado q_1 para o estado q_2 implica a leitura do primeiro símbolo da fita de leitura, que neste caso corresponde a a, e a alteração do símbolo presente no topo da pilha ($b \rightarrow y$), se e só se, existir uma transição

entre q_1 e q_2 que faça correspondência entre o símbolo a ser consumido e o símbolo no topo da pilha.

Assim, os AP são uma extensão aos autómatos finitos, onde são definidos à sua semelhança, sendo estendidos pela pilha, o que permite que estes autómatos alcancem o segundo nível da hierarquia de Chomsky, como referido inicialmente, obtendo uma maior capacidade no tipo de gramáticas que estes conseguem reconhecer. Sendo assim, através dos autómatos de pilha obtém-se maior força, representada pelo número de linguagens que é possível reconhecer. Uma vez que esta estrutura de dados é infinita, por consequência existe uma memória infinita, caracterizando um grande avanço face ao autómato finito. Para além de possuir a capacidade ilimitada de contar, os AP permitem que a informação permaneça armazenada e que se faça, durante o armazenamento e a sua posterior leitura, a correspondência entre símbolos pela ordem inversa.

Por norma, o AP rege-se por dois critérios de aceitação, i.e., duas maneiras diferentes de aceitar uma palavra: por estados de aceitação ou por pilha vazia. No primeiro caso, a palavra é aceite se no fim da leitura da mesma, o autómato se encontrar num estado de aceitação; e no segundo, se no fim da leitura, a pilha do autómato estiver vazia. Apesar de existirem estes dois modos de funcionamento, é sempre possível fazer a sua conversão, sendo trivial converter do critério de pilha vazia para o critério de estados de aceitação, sendo apenas necessário, adicionar uma última transição ao autómato, que verifica que a pilha está, de facto, vazia.

Para descrever o AP, é usual utilizar o λ, tanto para a leitura de símbolos, que neste caso significa que o símbolo não será consumido nem utilizado para a transição, tanto para indicar que nenhum símbolo será empilhado na pilha.

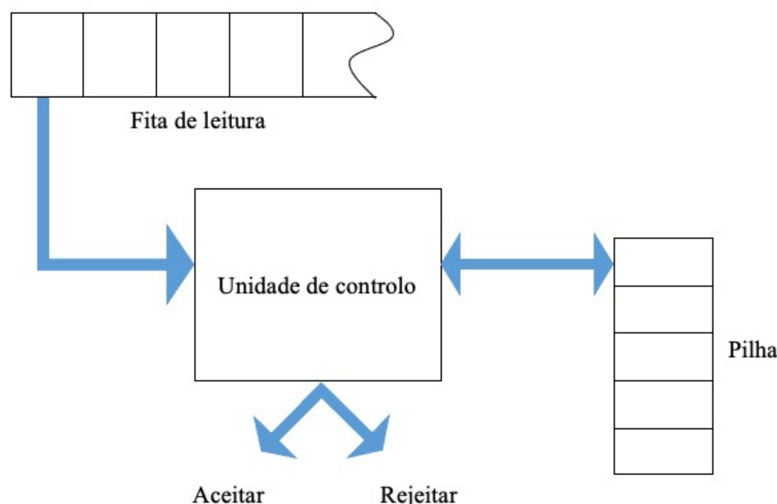

Figure 4.2: Autómato de Pilha

De maneira a compreender melhor o conceito de autómato de pilha é apresentado o

esquema acima (figura 4.2), composto por uma fita de leitura (*input file*), uma unidade de controlo e uma pilha.

Tal como o nome indica, unidade de controlo diz respeito à estrutura que controla o funcionamento do autómato, sendo que esta depende dos símbolos que constam na fita de leitura, bem como do símbolo presente no topo da pilha. Assim, a unidade de controlo lê a fita de leitura, símbolo a símbolo, e altera o conteúdo da pilha, de acordo com as operações pretendidas, resultando num novo estado na unidade de controlo e numa alteração na pilha.

A pilha, uma das componentes fundamentais deste tipo de autómato, no que diz respeito à sua estrutura e operacionalidade, define-se por uma sequência de símbolos, onde as operações são realizadas no elemento mais recente da pilha, ou seja, aquele que se encontra no topo da mesma. De notar que o símbolo inicial da pilha z é essencial para se detetar a pilha vazia. Relativamente ao que o AP pode fazer para observar da pilha, só existe a possibilidade de aceder ao elemento do topo. Se esse elemento for z, o AP fica a saber que a pilha está vazia. Presume-se que o elemento z é colocado no início da pilha e que não aparecerá repetido na pilha.

De forma a completar a descrição do processo de transição entre estados, são apresentadas as operações da pilha:

- Desempilhar z, que corresponde a desempilhar z e empilhar λ;
- Ler z sem desempilhar, que corresponde a a desempilhar z e empilhar z de novo;
- Empilhar γ, que corresponde a estando z no topo, desempilhar z e empilhar γz

Sendo que z é um símbolo, γ uma palavra e λ a sequência vazia.

4.2.2 Determinismo e não determinismo

Uma função define-se pela relação existente entre um conjunto de objetos e um conjunto de imagens, i.e., para cada objeto, existe apenas uma imagem correspondente. Em contrapartida, numa relação podem existir uma ou mais imagens para um único objeto. Deste modo, todas as funções são uma relação, contudo, nem todas as relações constituem funções.

Relativamente aos autómatos, estes podem ser classificados em deterministas ou não deterministas, dependendo do tipo da sua relação de transição. Caso esta corresponda a uma relação, o autómato é classificado como não determinista. Por outro lado, se esta for uma função nem sempre o autómato é determinista, sendo necessário garantir que:

Em termos práticos, para um autómato não determinista, ou seja, um autómato que contém mais que uma transição para o mesmo símbolo de input e símbolo no topo da pilha, para um mesmo estado, é necessário realizar esta computação em paralelo, i.e., sempre que houver mais que do que uma hipótese de transição, há que explorar os vários caminhos existentes. Ou de um ponto de vista matemático, podemos também pensar

que iremos simplesmente "escolher o caminho certo" em cada ramificação. Sendo apenas necessário um "caminho" que aceite a palavra, para a mesma ser aceite.

4.2.3 Exemplos Práticos e Diagramas de Transição

Um método de esquematizar AP (deterministas ou não deterministas) consiste em diagramas de transição. Estes diagramas descrevem o comportamento do autómato em cada estado, pelo que cada transição é legendada com o símbolo da fita de leitura a ser consumido, o símbolo do topo da pilha e o conjunto de símbolos a empilhar (figura 4.4). A utilização deste tipo de diagramas pode facilitar a compreensão das alterações que ocorrem no autómato, porém dificultam a realização de provas formais, visto que, para além de ser necessário acompanhar visualmente os símbolos presentes na fita de leitura, é igualmente importante ter em conta o conteúdo da pilha.

Os exemplos seguintes definem três autómatos de pilha - A1, A2 e A3 - através de tabelas de transição e os seus respetivos diagramas de transição. Para além disso, os autómatos A1 e A3 serão acompanhados por uma descrição detalhada do seu funcionamento.

O autómato A1 pode ser definido por:

$$A1 = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$$

,

em que,

$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{0, 1\}$
$\Gamma = \{0, 1, z\}$
$q_0 = q_0$
$z = Z$
$F = \{q_2\}$

A tabela de transição e o diagrama de transição referentes a este exemplo são apresentados nas Figuras 4.3 e 4.4, respetivamente.

δ	0	1	λ
q_0, Z	q_0, 0Z		
q_0, 0	q_0, 00	q_1, λ	
q_1, 0		q_1, λ	
q_1, Z			q_2, Z

Figure 4.3: Tabela de transição do autómato A1

Relativamente à descrição do funcionamento do autómato de pilha A1, dois possíveis *inputs* são 0011 e 001.

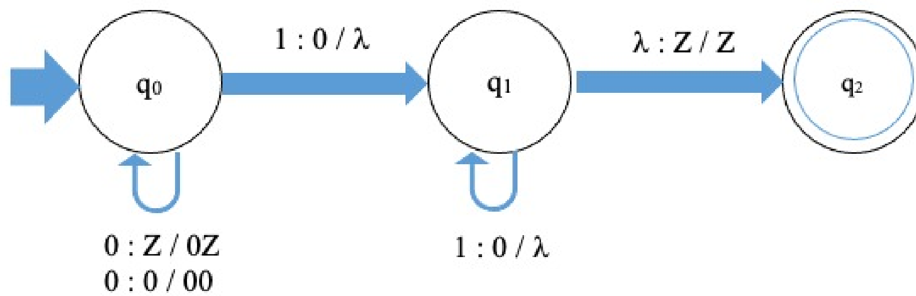

Figure 4.4: Exemplo de um autómato de pilha determinista que reconhece a gramática $\{0^n1^n : n > 0\}$

Para a palavra 0011, o autómato tem início no estado inicial q_0 e assume-se que a pilha contém o símbolo inicial Z. De seguida, o primeiro símbolo da palavra (0) é lido e consumido e é adicionado o símbolo 0 à pilha. Após esta transição, o autómato mantém-se no estado q_0 e a fita de leitura contém os símbolos 011. O próximo símbolo a ser lido é 0, o qual é processado do mesmo modo que o primeiro. Neste momento, o autómato ainda se encontra no estado q_0, a fita de leitura é composta por 11 e a pilha por 00. Seguidamente, o autómato lê o símbolo 1 e desempilha o 0 que se encontra no topo da pilha, transitando para o estado q_1. Para o último símbolo a ser lido (1), ocorre a desempilhamento do 0 e a transição para o mesmo estado (q_2), estando agora presente na pilha apenas o símbolo Z. Finalmente, é realizada a transição do estado q_1 para o estado q_2, dado que Z é o símbolo que se encontra no topo da pilha e esta transição não requer a leitura de nenhum símbolo. Uma vez que o autómato já consumiu todos os símbolos da fita de leitura e se encontra no estado final, a palavra é aceite.

Para o mesmo autómato, desta vez com a palavra 001 dá-se o seguinte funcionamento: nos três primeiros passos, realizam-se exatamente as mesmas operações que para a palavra 0011, pelo que o autómato encontra-se no estado q_1, a fita de leitura está vazia e a pilha contém os símbolos 0 e z. Deste modo, como não existem mais símbolos da fita leitura a serem consumidos, o autómato não pode realizar mais transições, e visto que não se encontra num estado final (que neste exemplo seria o estado q_2, a palavra é rejeitada.

Os autómatos de pilha estão diretamente relacionados com gramáticas independentes de contexto, o que significa que ambos possuem o mesmo poder de expressão e que para qualquer linguagem criada com base em gramáticas independentes de contexto, existe um AP que a implementa/que reconhece a linguagem. Para garantir a equivalência entre autómatos de pilha e gramáticas independentes de contexto, é preciso obter uma fórmula geral de traduzir uma gramática em AP e vice-versa. Desta maneira, a equivalência entre ambos pode ser provada quando, para qualquer gramáticas independentes de contexto, for possível construir um autómato de pilha que a reconheça, assim como, quando, a partir de qualquer autómato de pilha, exista uma gramática independente de contexto equivalente.

Os autómatos de pilha deterministas apenas são equivalentes a um sub conjunto das LIC.

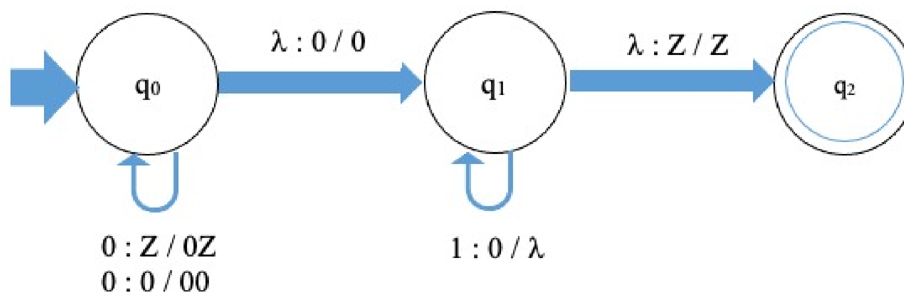

Figure 4.5: Exemplo de um autómato de pilha não determinista que reconhece a gramática $\{0^n 1^n : n > 0\}$

No seguinte exemplo apresentado na figura 4.5, é usado um autómato não determinista que reconhece a linguagem $\{0^n 1^n : n > 0\}$, sendo equivalente ao autómato da figura 4.4, ou seja, que reconhece a mesma linguagem. Neste caso, temos um autómato não determinista que tem um autómato determinista equivalente mas não sempre é este o caso, como irá ser mostrado de seguida.

Neste sentido, pode ser usado um AP determinista, pois a linguagem gerada tem a propriedade de ser divisível em dois conjuntos de passos definidos de maneira recursiva, i.e., no primeiro ocorre o empilhamento de 0 (zeros) e de seguida, ao encontrar o primeiro 1, a sucessivo desempilhamento destes. Numa primeira fase da computação, recorrendo à pilha, é realizada uma contagem de 0's, e numa segunda etapa, apenas é necessário verificar que, de facto, o número de 1's é equivalente ao número de 0's empilhados. Este fenómeno não é restrito a palavras que contenham a mesma cardinalidade à esquerda e à direita do seu meio, mas sim, para todas as linguagens que possa ser identificado uma partição da computação, ou seja, para gramáticas como por exemplo $0^n 1^{2n}, n > 0$ e $0^{3n} 1^n, n > 0$. Assim, é sempre possível dividir a computação nestes dois conjuntos de passos, construindo-se, então, um AP determinista que reconhece estas linguagens, apenas se existirem símbolos que se encontram em posições previsíveis, isto quer dizer que, por exemplo, na linguagem gerada pela gramática $0^n 1^n$, logo que é lido um 1 pelo autómato, conclui-se que este se encontra na segunda fase da computação. Assim, após se estabelecerem os dois grupos de símbolos, através de uma barreira, correspondentes às duas fases da computação, torna-se fácil encontrar uma solução determinista para os mesmos.

Até para linguagens geradas por gramáticas como $0^n 1^{n+m} 2^m$, é possível separar em fases de empilhamento e desempilhamento de símbolos, realizando-se contagens entre estes. Uma vez que as barreiras são bem definidas, todos estes exemplos pertencem a uma categoria de gramáticas que podem ser reconhecidas por AP deterministas. Para este exemplo em concreto, a separação é feita em 4 grupos, agrupados em 2 fases de

contagem. Na primeira fase, para cada 0 lido, são empilhados 0's, sendo que se faz uma correspondência com n 1's e desempilhando-se um 0 por cada um destes. De seguida, através da mesma sequência de passos, na segunda fase, empilham-se 1's e à *posteriori*, os mesmos são desempilhados m vezes.

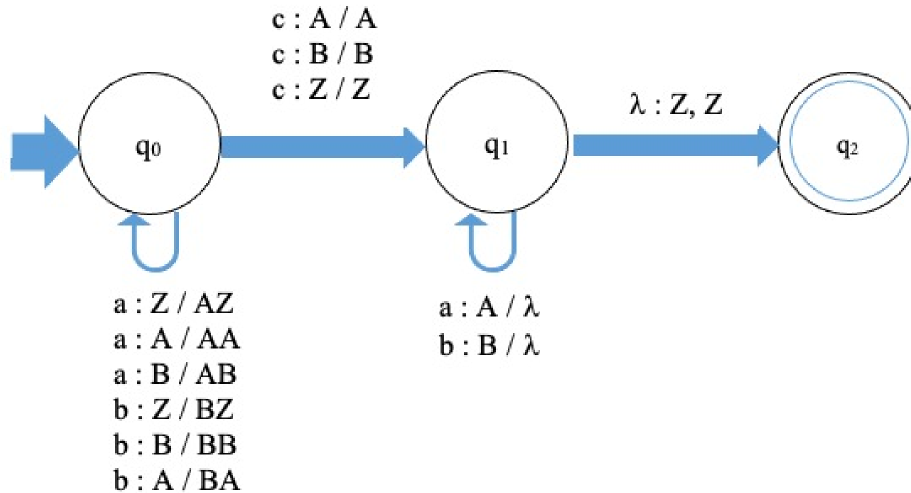

Figure 4.6: Exemplo de um autómato de pilha que reconhece a Linguagem $\{wcw^{-}1 : w \in \{a,b\}^{*}\}$

No que diz respeito à gramática $\{wcw^{-1} : w \in \{a,b\}^{*}\}$, reconhecida pelo autómato de pilha na figura 4.6, que define um palíndroma, em que w corresponde a uma palavra e w^{-1}, à mesma palavra mas invertida, divididas pelo símbolo c, como por exemplo $abbcbba$, também pode ser construído um autómato determinista que reconhece esta linguagem, visto que esta gramática gera palavras que contêm um símbolo que "separa" concretamente as 2 fases da computação: neste caso, o símbolo c.

Por outro lado, para uma linguagem definida por $\{ww^{-}1 : w \in \{a,b\}*\}$, reconhecida pelo AP apresentado na 4.8 (com a respetiva tabela de transição presente na figura 4.7), que gera palavras como por exemplo $aabb$, $abba$ e $aaaa$ não é possível construir autómatos de pilha deterministas. Para os dois primeiros exemplos de palavras geradas por este, a divisão é bem definida, contudo, para a palavra $aaaa$, esta propriedade já não se verifica, uma vez que não se verifica uma separação concreta que identifique as fases de empilhamento e de desempilhamento. Visto que se trata de uma categoria onde não é possível gerar autómatos deterministas para reconhecer este tipo de gramáticas, é necessário recorrer ao não determinismo. Isto deve-se ao facto de não ser possível prever o momento exato em que existe uma separação das fases de contagem.

Sendo assim possível concluir que para os autómatos de pilha, o não determinismo e o determinismo não possuem a mesma capacidade de reconhecimento, e apenas se obtém o mesmo poder de expressividade que as GICs através do uso de não determinismo.

O autómato A2 que reconhece $\{wcw^{-}1 : w \in \{a,b\}^{*}\}$ pode ser definido por:

$$A2 = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$$

,

em que,

$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{a, b\}$
$\Gamma = \{A, 1B, z\}$
$q_0 = q_0$
$z = Z$
$F = \{q_2\}$

δ	a	b	λ
q_0, Z	q_0, AZ	q_0, BZ	
q_0, A	q_0, AA q_1, A		
q_0, B	q_0, AB	q_0, BB q_0, B	
q_1, A	q_1, λ	q_0, BA	
q_1, B		q_1, λ	
q_1, Z			q_2, Z

Figure 4.7: Tabela de transição do autómato $\{ww^{-}1 : w \in \{a, b\}^*\}$

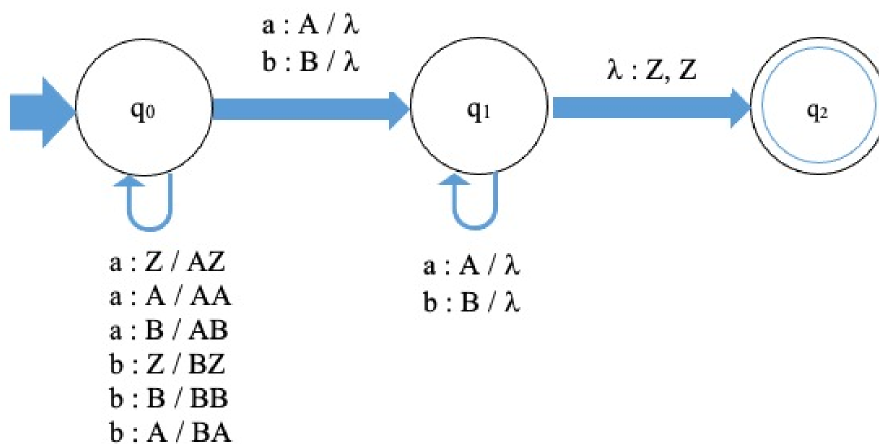

Figure 4.8: Exemplo de um autómato de pilha que reconhece a Linguagem $\{ww^{-}1 : w \in \{a, b\}^*\}$

O autómato A3 definido por:

$$A3 = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$$

,

em que,

$Q = \{p, q, r\}$
$\Sigma = \{a, b\}$
$\Gamma = \{Z\}$
$q_0 = q$
$z = Z$
$F = \{r\}$

δ	a	b	λ
p, Z	q, Z		
q, Z	r, Z		q, Z
r, Z	r, Z	r, Z	

Figure 4.9: Tabela de transição do autómato

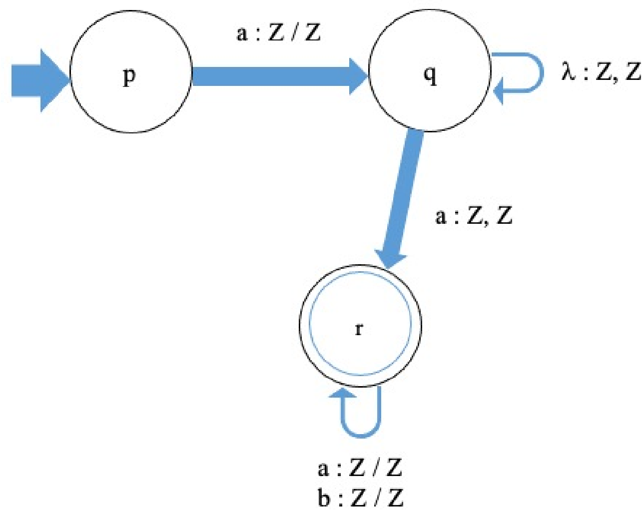

Figure 4.10: Exemplo de um autómato que pode entrar em ciclo infinito

Os autómatos de pilha, para além da aceitação ou rejeição da palavra, podem também permanecer num ciclo infinito. Esta situação deve-se ao facto de também ser possível ignorar o símbolo de input, em virtude de uma transição lambda, criando assim um ciclo, onde o autómato poderá permanecer, sem fazer progresso no seu reconhecimento. Por exemplo, no autómato A3 (figura 4.10) este autómato reconhece palavras começadas por dois símbolos "a" seguidas por um ou mais símbolos "a" ou "b". Rejeita palavras

começadas por b e a palavra vazia. Os casos restantes entram em ciclo infinito. Para este caso, o autómato irá entrar num ciclo infinito. Este caso, terá que ser abordado de forma especial na implementação dos autómatos.

4.3 Gramáticas Independentes de Contexto

Nesta secção irá ser abordadas as gramáticas independentes de contexto, que encontram-se no mesmo nível da hierarquia de Chomsky e porque irão ser usadas em parte do trabalho proposto.

As gramáticas independentes de contexto, são dispositivos geradores, estas geram palavras usando certas regras de produção. Sendo que o mecanismo de geração é realizado através da reescrita. As regras contêm símbolos terminais e símbolos não terminais, onde cada símbolo não terminal pode ser substituído por uma sequência de símbolos terminais e não terminais. Quando obtemos uma palavra que apenas contém símbolos terminais, o processo está concluído e dessa forma, esta é gerada.

A gramática é formalizada através de um conjunto de regras de produção, como por exemplo,

$$A- > bAb$$

$$A- > a$$

A primeira regra indica que podemos gerar palavras que contêm um b em cada extremidade. Usando esta regra de forma recursiva, podemos obter palavras como $bAb, bbAbb, bbbAbbb$. A segunda regra indica que podemos substituir o A pelo símbolo terminal a. Logo através da mesma conseguimos fechar o ciclo, ou seja, após as substituições obter uma palavra que apenas contém símbolos finais, como por exemplo $bbabb$.

A definição formal geral de uma gramática independente de contexto é a seguinte:

- T, o conjunto dos símbolos terminais.
- N, o conjunto dos símbolos não terminais
- $S \in N$, o símbolo inicial
- P, o conjunto das produções, $\alpha \longrightarrow \beta, \alpha \in V^*NV^*, \beta \in V^*$

Sendo, T e N disjuntos e $V = T \cup N$

5 OCaml-FLAT/OFLAT

O OCaml-FLAT e o OFLAT são as duas componentes principais que fazem parte da estrutura já existente deste projeto. O primeiro consiste na lógica principal da ferramenta, e o segundo na aplicação web, que implementa a interface gráfica. Ambas usam o OCaml para toda a construção.

O OCaml-FLAT é uma biblioteca composta por vários módulos, cada modelo é construído numa classe. Esta biblioteca, até ao momento da escrita deste documento, implementa autómatos de pilha, expressões regulares, gramáticas livres de contexto, assim como alguns módulos de apoio, como por exemplo, operações sobre ficheiros de JSON, erros, parsers, entre outros.

5.1 Biblioteca OCaml-FLAT

Na Elaboração da Dissertação esta biblioteca irá ser estendida com um módulo que irá implementar os autómatos de pilha com todas as suas operações essenciais, nomeadamente: limpeza do autómato, teste de determinismo, aceitação de palavras, geração de palavras, conversões de e para outros tipos de modelo. As principais dificuldades que serão precisas enfrentar na programação são: como lidar com o não determinismo, como lidar com os ciclos nos autómatos, como lidar com a possibilidade de não terminação de alguns algoritmos.

5.2 OFLAT

O OFLAT (figura 5.1) é uma aplicação web que tem por base o desenvolvimento em OCaml, onde usa a biblioteca js_of_ocaml como base para a interoperabilidade entre o OCaml e JS. A ferramenta OFLAT oferece uma interface gráfica interativa e permite ao utilizador criar os mais diversos modelos da Teoria FLAT, observar a evolução do autómato dada uma certa palavra, converter entre autómatos e gramáticas, entre outros.

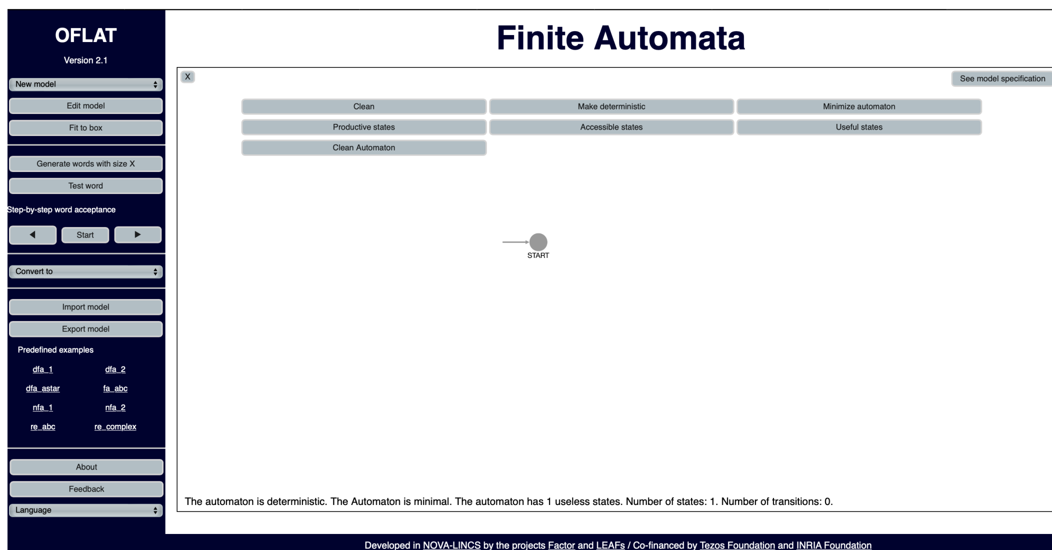

Figure 5.1: OFLAT - Ferramenta Pedagógica para FLAT

5.3 js_of_ocaml

O *js_of_ocaml* [13] é uma framework "open source" bastante completa que permite usar OCaml para desenvolvimento "web" e "mobile", onde o código é traduzido e compilado automaticamente para javascript para ser usado no browser. Isto permite construir toda a ferramenta em OCaml, usando-o tanto para a biblioteca OCaml-FLAT ("backend") implementando a lógica principal dos autómatos, como para construir a interface gráfica. Para além de uma integração mais consistente, em torno de toda a ferramenta, permite usar todas as características funcionais do OCaml na construção da mesma, tirando ainda partido das vantagens de uma linguagem com um sistema de tipos forte, identificando possíveis erros em tempo de compilação.

5.4 Cytoscape.js

A Cytoscape.js é uma biblioteca JavaScript "open source" que permite a construção de grafos dos mais variados tipos dentro de páginas Web. Neste contexto é usada para o desenho dos autómatos na interface gráfica, sendo suportada as mais diversas operações no desenho de nós e arcos que são os principais blocos de construção na representação visual de autómatos e também de outras estruturas como árvores de derivação.

6

Apresentação e discussão das novas funcionalidades

A nova versão do OFLAT permite ao utilizador criar um autómato de pilha sobre a forma gráfica (a forma de tabela ainda não foi considerada). O utilizador tem a possibilidade de representar estados através de nós e transições através de arcos. Pode também indicar qual o estado inicial, quais os estados finais e as condições associadas a cada transição.

A interface gráfica dos autómatos de pilha foi concebida de maneira a colmatar algumas das desvantagens das aplicações semelhantes. A operação principal da aplicação é o 'accept', esta permite observar o comportamento do autómato ao longo da aceitação, mostrando as configurações do mesmo em cada passo. Dada uma palavra, um passo representa o consumo de um carácter. Assim, em cada passo, é possível observar a evolução das configurações do autómato.

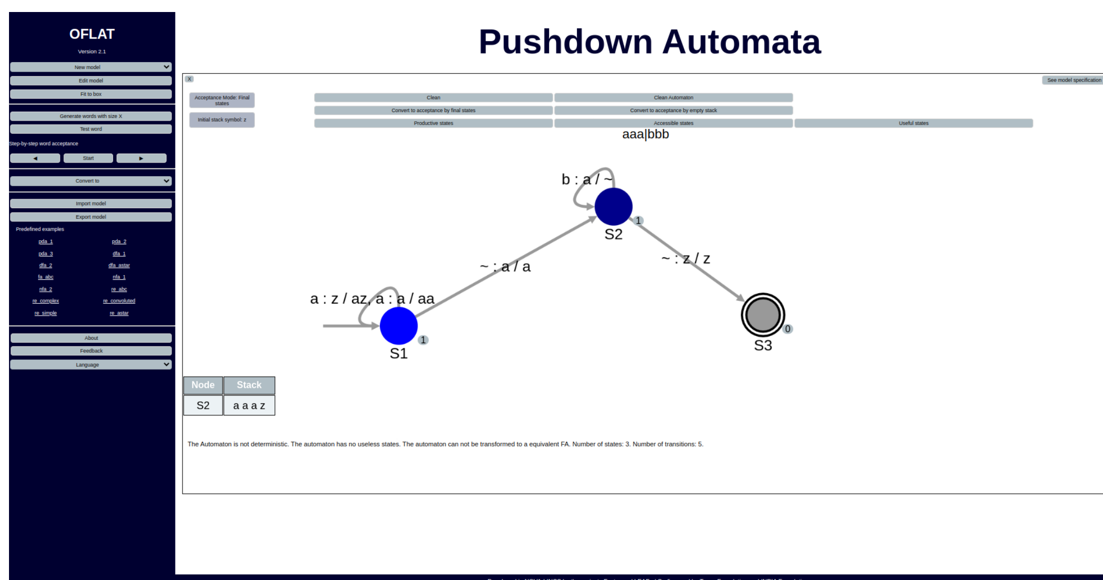

Figure 6.1: OFLAT Interface gráfica accept

6.1 Layout para a criação, manipulação e visualização de Autómatos de Pilha

A ferramenta permite criar um autómato de pilha através de menus e pop-ups, assim como realizar várias operações sobre os mesmo que também serão discutidas no subcapítulo seguinte. Estes menus já existiam implementados para os autómatos pilha, foi apenas necessário realizar uma adaptação dos mesmos e corrigir alguns detalhes de implementação. Por exemplo, como o objetivo destes aquando da sua implementação era a utilização com autómatos finitos, as operações de manipulação do grafo na interface gráfica apenas funcionavam com um símbolo em cada arco. Com a introdução dos autómatos de pilha, e tendo em conta que as transições são relativamente mais complexas, houve a necessidade de generalizar parte das funcionalidades já existentes para ser usáveis pela nova categoria de autómatos.

6.1.1 Layout criação e edição de um autómato pilha

O OFLAT através dos menus de forma circular permite criar, editar e eliminar componentes do autómato. No primeiro exemplo temos os botões de criação de novos estados (figura 6.2). De seguida (figura 6.3) é apresentada o mecanismo de criação de novas transições através da ligação de estados existentes. E por fim, o menu de edição do autómato (figura 6.4) que engloba as operações de eliminar estados e transições, assim como tornar um estado final ou inicial e por fim renomear os estados existentes.

Figure 6.2: OFLAT Interface gráfica operações criação do autómato

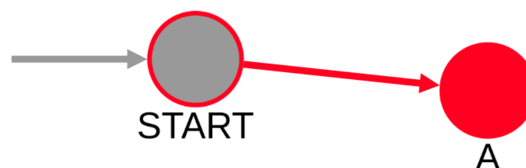

Figure 6.3: OFLAT Interface gráfica criação de nova transição

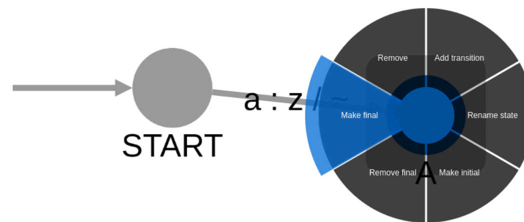

Figure 6.4: OFLAT Interface gráfica operações de edição do autómato

6.1.2 Layout operação de aceitação de uma palavra

No menu do lado esquerdo com operações gerais sobre todos os modelos implementados pela ferramenta existe a interface da operação de accept. Esta funcionalidade tem dois modos principais de operação, o *Test word* e o *Start*. Em ambos estes modos é pedida uma palavra para ser realizada a operação de aceitação do autómato, após a introdução da palavra pretendida, o *Test word* irá realizar uma simulação passo a passo automática, onde o utilizador poderá observar a evolução do autómato. Na segunda operação, *Start*, o utilizador irá através dos botões indicados com uma seta (figura 6.5), controlar passo a passo a evolução do algoritmo de aceitação do autómato, podendo assim observar o comportamento do mesmo de forma interativa.

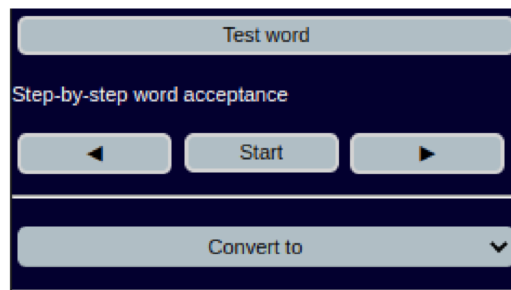

Figure 6.5: Interface gráfica menu da operação accept

Sendo a configuração um triplo (estado corrente, símbolos por consumir e o conteúdo da pilha) estas são apresentadas da seguinte forma ao utilizador. Os símbolos por consumir são apresentados no topo, com o carácter '|' a separar os que foram consumidos de aqueles que faltam ser consumidos. Sendo que em cada passo o autómato avança um carácter, os símbolos por consumir são comuns a todas as configurações correntes do autómato. De seguida na configuração, o estado corrente do autómato é indicado com um nó com a cor azul, assim como numa tabela no canto inferior esquerdo. E por fim, o conteúdo da pilha é indicado de duas formas, quando é colocado o rato por cima de um dos nós a azul figura, ou tal como o estado corrente, no canto inferior esquerdo, pela tabela que indica o estado atual do autómato e o conteúdo da pilha, figura 6.8.

De facto, a representação é duplicada, sendo que para um autómato determinista o

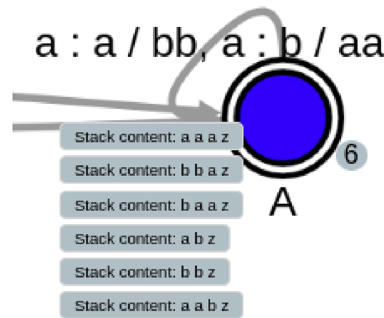

Figure 6.6: A subfigure

Node	Stack
START	a a a a z

Figure 6.7: A subfigure

Figure 6.8: Configurações de um estado/configuração tabela

acesso a esta informação pode ser realizada das duas formas. Esta decisão no design foi pensada para manter a coerência entre a representação das configurações para um autómato determinista e para não deterministas.

A grande diferença está quando é representado um autómato não determinista, sendo necessário representar várias configurações ao mesmo tempo. Para este, a decisão foi ir de encontro à experiência do utilizador e tentar encontrar uma forma mais simples de mostrar as múltiplas configurações do autómato. Através dos próprios estados, colocando o rato em cima de um nó, são mostrados todos os estados da pilha 6.8 para todas as configurações que estão nesse estado. Esta escolha de design tem a vantagem de manter bastante coesa a informação, sendo mais intuitivo que apenas mostrar todas as configurações em tabelas, separadas do grafo. Para além desta forma de visualização foi adicionada uma única tabela no canto inferior esquerdo. Em que, para autómatos deterministas, esta contém um estado corrente e o estado da pilha e, para autómatos não deterministas, contém a mesma informação, mas apenas de uma das configurações possíveis. A configuração apresentada é a referente a um dos caminhos mais curtos para a aceitação da palavra. Ou seja, é calculado anteriormente o caminho mais curto para a aceitação, se existente, e em cada passo é apresentada a configuração do mesmo. Sendo a informação do caminho mais curto complementada com um nó com um tom azul-escuro, de maneira a ser visualizada esta informação também através do grafo.

Esta decisão traz dois grandes benefícios, em primeiro lugar, para o caso em que o autómato é determinista, tendo este apenas uma configuração durante toda a aceitação, a configuração do autómato está sempre visível durante todos os passos, sem ser necessário usar o mecanismo de colocar o rato sobre os estados. E em segundo lugar quando o

autómato é não determinista, nos passos em que existe mais que uma configuração esta tabela indica um dos possíveis caminhos mais curtos para a aceitação da palavra.

Uma das desvantagens observadas noutras ferramentas para autómatos de pilha não deterministas, é a visualização das configurações em cada passo, sendo que estas geralmente são visualizadas em tabelas, uma tabela por cada configuração, quando o número de configurações aumenta torna-se difícil a leitura da interface de maneira a ter uma visualização intuitiva do comportamento do autómato. A separação do grafo e das configurações torna a experiência pouco coerente e de difícil leitura do ponto de vista do utilizador. Sendo assim, a construção gráfica do OFLAT foi desenhada para colmatar este problema. Tornando a experiência de utilização mais fácil e intuitiva de maneira a compreender e de obter uma visão coesa e simples do funcionamento do autómato.

Para além disso, foi adicionado um número em cada nó que representa o número de configurações presentes em cada estado num determinado momento. Esta adição torna a utilização mais intuitiva, de maneira que é possível observar parte do estado do autómato sem ser necessário aceder à informação de cada configuração de um nó. Desta forma, mantemos a representação gráfica de o número de configurações em cada momento do autómato, tal como as restantes ferramentas, mas não é introduzido ruído provocado pela explosão de informação gerada pelas configurações atuais estarem todas visíveis durante o processo interativo da aceitação de uma palavra.

6.2 Operação *Accept* vantagens dos mecanismos implementados

A introdução da tabela que representa um dos possíveis caminhos mais curtos para a aceitação da palavra juntamente com o mecanismo de visualizar as configurações através de um nó faz com que sejam introduzidos alguns aspetos relevantes na forma como o utilizador analisa o comportamento do autómato.

Podemos observar a utilidade da introdução do mecanismo do caminho mais curto durante a aceitação da palavra no seguinte autómato que reconhece a linguagem $ww^{-}1$ (figura 6.9).

Neste momento está a ser realizada a aceitação interativa da palavra "aabbaabbaa" pelo autómato de pilha, sendo que o utilizador poderá avançar e retroceder, iterando pelos passos do autómato até à aceitação da mesma. No momento demonstrando pela figura 6.9 o autómato situa-se com uma configuração no estado S2, este estado representa o início do empilhamento de a's ou de b's pelo autómato. Como este não é estado final, a partir de agora para aceitar a palavra o mesmo terá que desempilhar os símbolos empilhados pelas transições de S1 para S2 e de S2 para S2. Sendo assim, podemos concluir que o Estado S2 apenas terá configurações que se encontram na primeira parte da palavra a ser reconhecida, ou seja, que ainda será empilhando mais símbolos de w para à posteriori serem desempilhados, de maneira a construir$w^{-}1$.

Na figura 6.10 for realizado mais um passo do processo interativo de aceitação e os símbolos já consumidos são "aa". Sendo que falta consumir os símbolos "bbaabbaa".

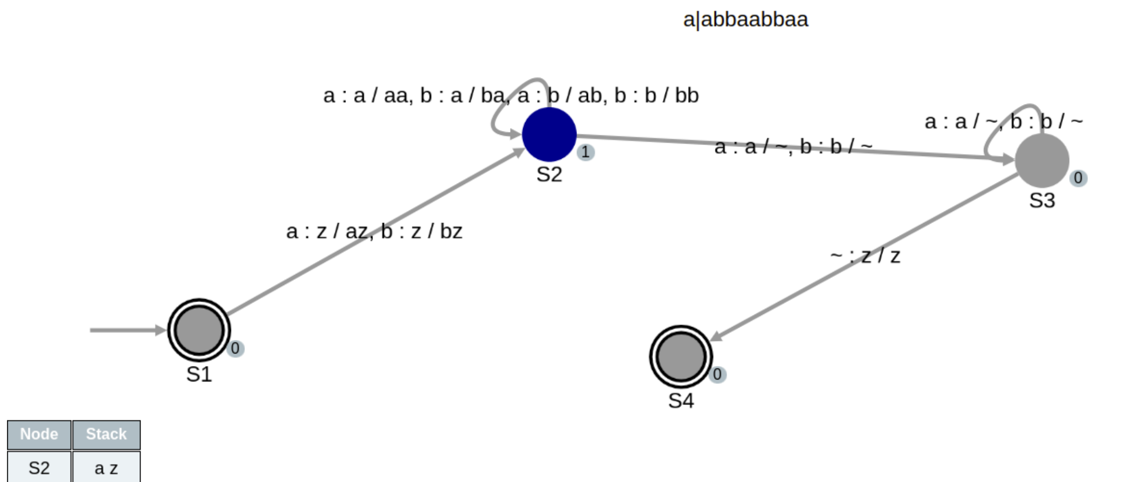

Figure 6.9: OFLAT Interface gráfica accept passo número 1

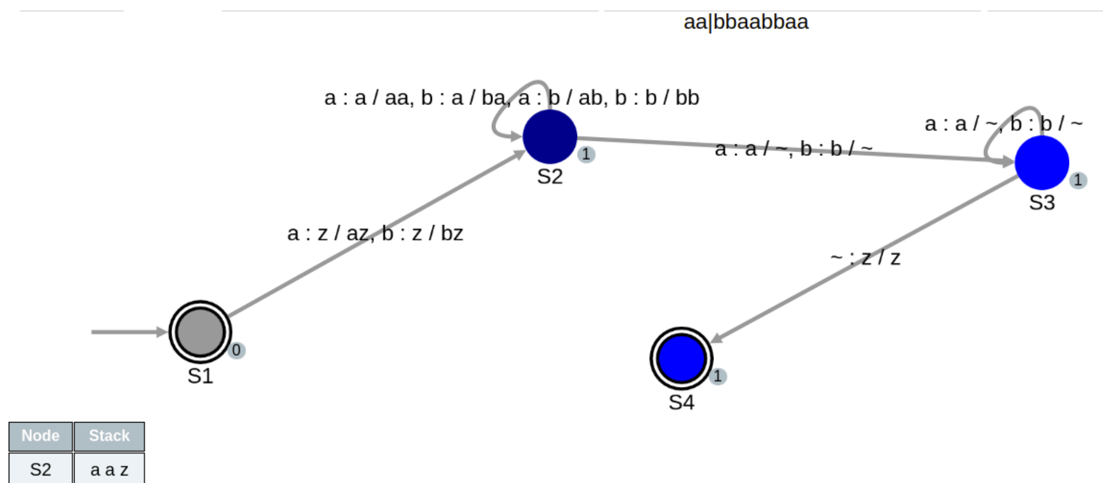

Figure 6.10: OFLAT Interface gráfica accept passo número 2

Tendo em conta que o autómato de pilha está a reconhecer $ww^{-}1$, como existe uma configuração ativa em S4 podemos assim concluir que "aa" faz parte da linguagem reconhecida pelo autómato de pilha. Para além disso é possível observar que o autómato tem uma configuração em S2 e em S3. Em S2, como explicado anteriormente, significa que estamos numa fase de empilhamento, ou seja construção de w. E em S3 será o autómato numa fase de desempilhamento. Isto significa que esta configuração tem a "esperança" que os 2 símbolos seguintes da palavra sejam "aa". E desta forma esta configuração irá de facto "chegar" ao fim e ser aceite. Como o próximo símbolo é um "b", esta configuração irá "morrer". Mas deste ponto de vista, é possível observar que cada configuração do autómato representa a esperança do mesmo evoluir até um estado de aceitação.

Em relação ao estado marcado com azul-escuro, este representa uma configuração que faz parte do caminho mais curto aceite pelo autómato para a palavra a ser reconhecida. Deste modo é possível observar de maneira muito simples a evolução da mesma ao longo

da aceitação da palavra pelo autómato. Dando assim uma perspetiva visual nova do comportamento do autómato de pilha.

Com este estado azul-escuro, concluímos que o autómato neste ponto da palavra ainda se encontra a empilhar (construir w), dando assim uma visualização simples da aceitação de um autómato não determinista.

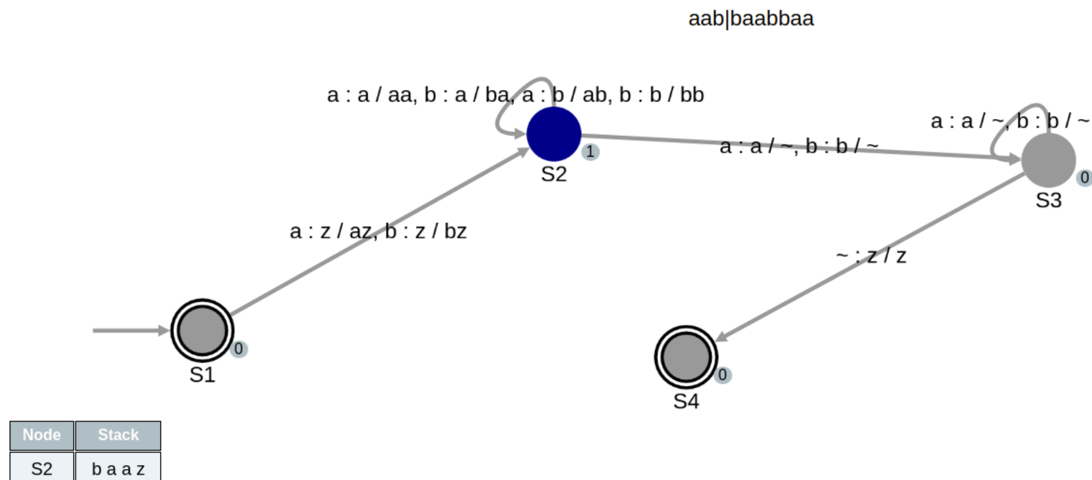

Figure 6.11: OFLAT Interface gráfica accept passo número 3

Na figura 6.11 observamos o comportamento anteriormente descrito, da "morte" dos estados. Neste momento, último símbolo consumido foi um 'b', tendo em conta que as configurações em S3 estão em modo de desempilhar, como anteriormente apenas existem a's, não existe qualquer 'b' na pilha para desempilhar.

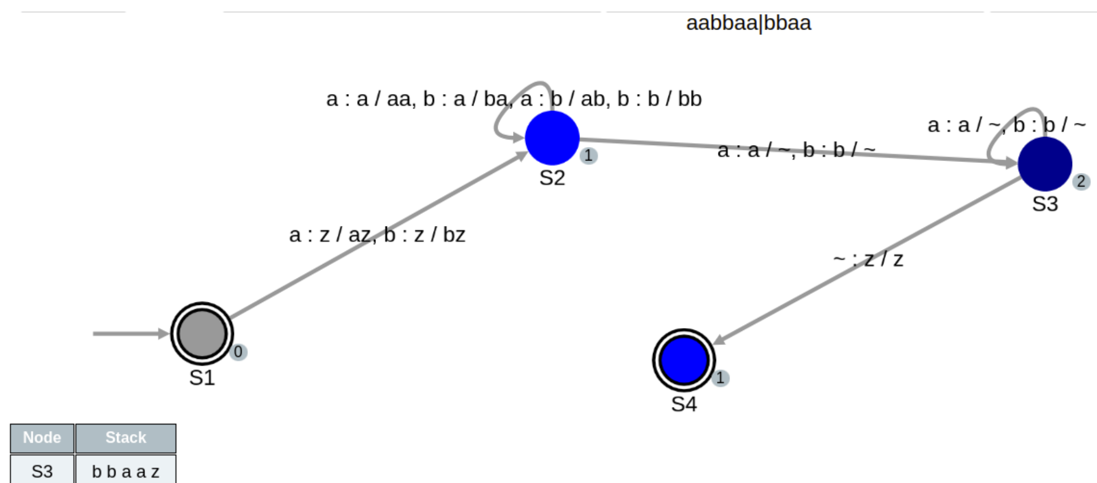

Figure 6.12: OFLAT Interface gráfica accept passo número 6

A partir do ponto da figura 6.12 observamos a mudança da "melhor configuração aceite". Esta passa a estar no estado S3, neste ponto observamos que de facto metade da palavra encontra-se consumida. Para a mesma ser aceite o autómato tem necessariamente

de estar em modo de desempilhar, reconhecendo agora através do mecanismo da pilha pares de símbolos pela ordem inversa. Sendo assim, de seguida, todas as configurações em S2 em todos os passos seguintes são "esperanças" de o reconhecimento da palavra não ter alcançado o meio da palavra. Mas como na realidade sabemos que já a alcançamos, todas estas configurações irão eventualmente "morrer".

Novamente podemos observar na mesma figura que existe uma configuração em S4 (estado final) sendo que representa que a palavra reconhecida até este ponto faz parte da linguagem reconhecida pelo autómato de pilha.

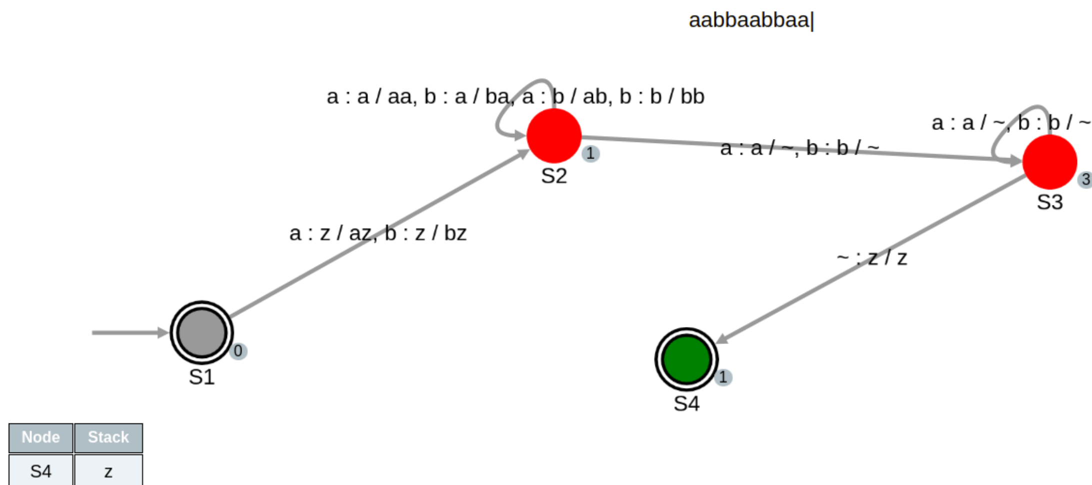

Figure 6.13: OFLAT Interface gráfica accept passo número 10

Finalmente na figura 6.13 observamos o estado após o consumo de todos os símbolos da palavra, e, de facto, concluímos que a palavra é aceite.

Em todos os passos, é também possível observar na tabela do canto inferior esquerdo, qual é o estado corrente do "melhor caminho aceite" e o estado da pilha atual, referentes à configuração que pertence ao caminho mais curto. Para as restantes configurações seria possível a qualquer momento observar todos os estados da pilha em cada estado do autómato complementando assim de maneira simples e visual o processo de aceitação da palavra pelo autómato de pilha.

6.3 Interface de geração de palavras reconhecidas pelo autómato

A operação que permite gerar palavras reconhecidas pelo autómato até um determinado número de símbolos está acessível no menu fixo à esquerda da janela. (figura 6.14).

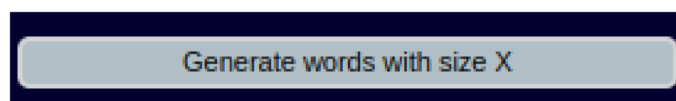

Figure 6.14: Interface da geração de palavras

O primeiro passo da utilização desta funcionalidade, requer ao utilizador o input de um número que será usado para gerar palavras de tamanha até n (sendo n o número introduzido pelo utilizador). As palavras geradas com tamanho até n que são reconhecidas pelo autómato iram ser apresentadas num novo painel à direita do painel principal, ocupado pelo autómato a ser usado (figura 6.15).

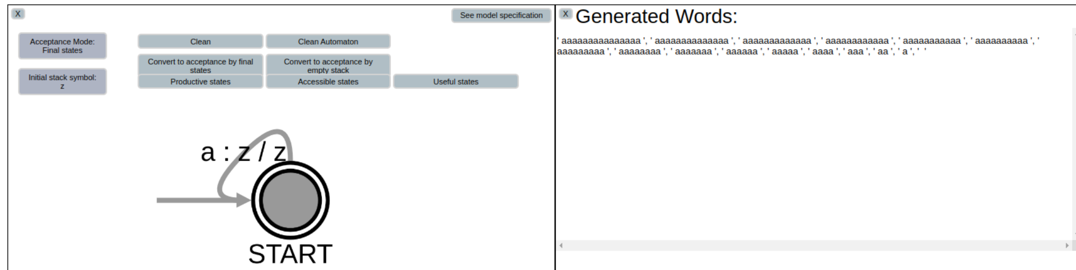

Figure 6.15: Exemplo de geração de palavras de um autómato de pilha

6.4 Operações de conversão de autómato de pilha para outros modelos e vice-versa

A ferramenta permite realizar a conversão de todos os modelos implementados na biblioteca para qualquer outro modelo. Neste sentido, com a introdução dos autómatos de pilha ao OFLAT foram implementadas as conversões de autómato de pilha para expressões regulares, gramáticas livres de contexto e autómatos finitos e vice-versa. A ferramenta permite realizar esta conversão a partir do menu fixo no lado esquerdo da página, mais concretamente, através de um dropdown menu que permite selecionar o modelo para qual é pretendido realizar a conversão. (figura 6.16).

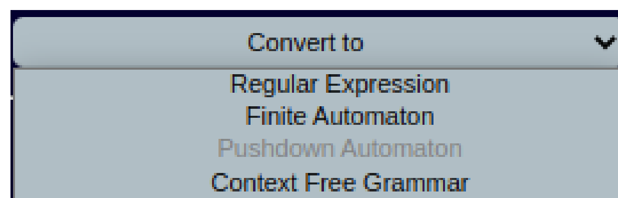

Figure 6.16: Menu de conversão entre modelos da ferramenta

Após selecionar o modelo pretendido é aberta uma caixa com o modelo gerado pretendido à direita do modelo original (figura 6.17). Desta forma, é fácil de comparar e analisar a conversão efetuada e estudar de forma simples a mesma. Caso o utilizador queira continuar a utilizar as operações sobre o modelo gerado, basta fechar o visualizador do modelo original e de seguida, o modelo gerado passará a estar como modelo principal de edição com todas as funcionalidades implementadas para este. Desta maneira a ferramenta permite uma grande interoperabilidade entre os vários modelos da teoria FLAT existentes.

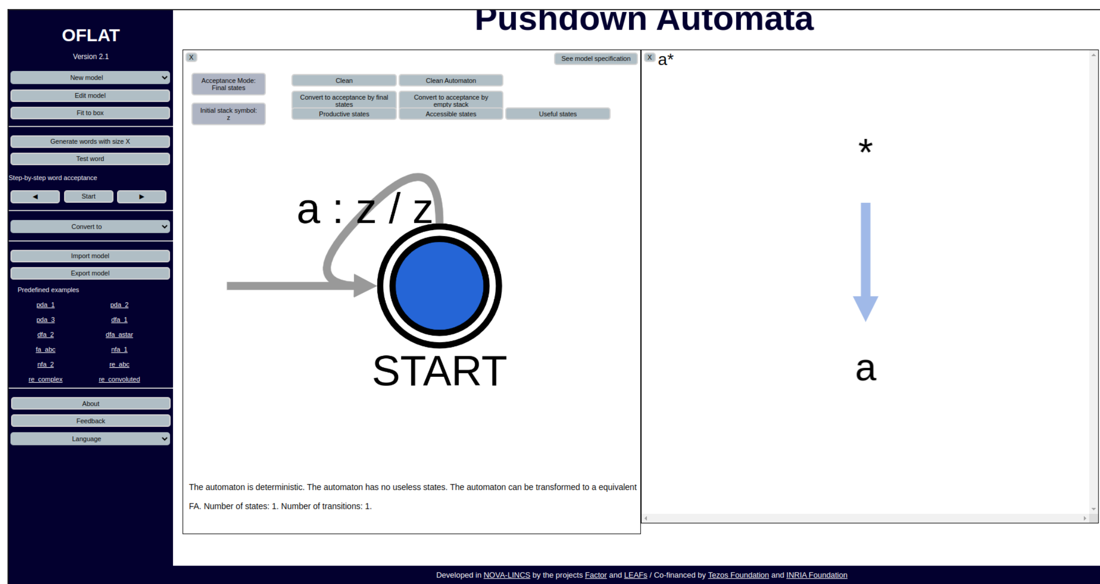

Figure 6.17: Conversão de autómato de pilha para expressão regular

6.5 Operações complementares implementadas para Autómatos de Pilha

6.5.1 Operações de categorização dos estados do autómato

De seguida iremos percorrer as várias funcionalidades implementadas para além do *accept*. A ferramenta gráfica tem um mecanismo de identificação de estados especiais de um autómato de pilha, sendo possível observar os estados produtivos, estados que têm um caminho desde o próprio até um estado final (figura 6.18), estados acessíveis, estados onde existe um caminho desde o estado inicial (figura 6.19) e os estados úteis, estados que são simultaneamente produtivos e acessíveis (figura 6.20).

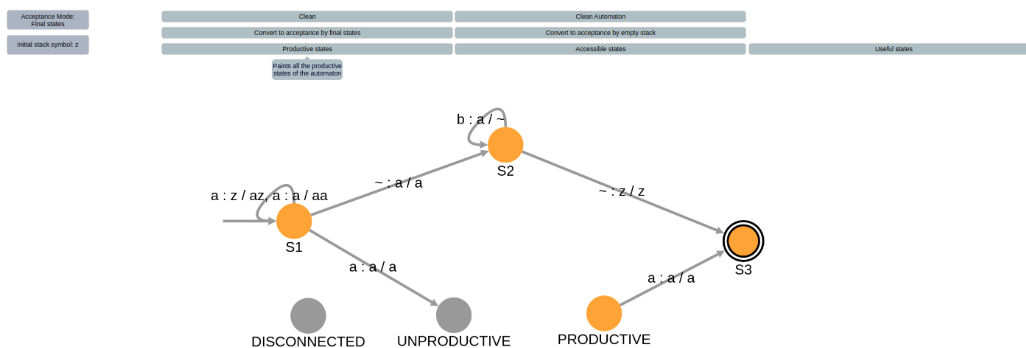

Figure 6.18: Operação de identificação de estados produtivos

A ferramenta também permite a limpeza do autómato. Esta operação elimina estados inúteis copiando o autómato original para uma janela suplementar na secção mais à direita

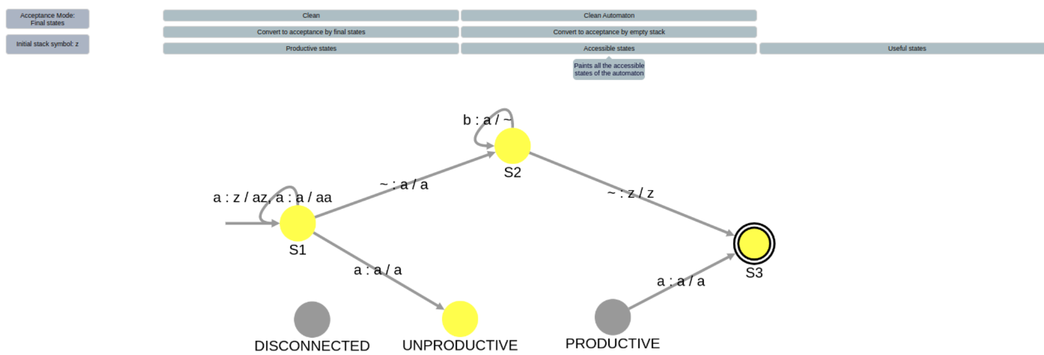

Figure 6.19: Operação de identificação de estados acessíveis

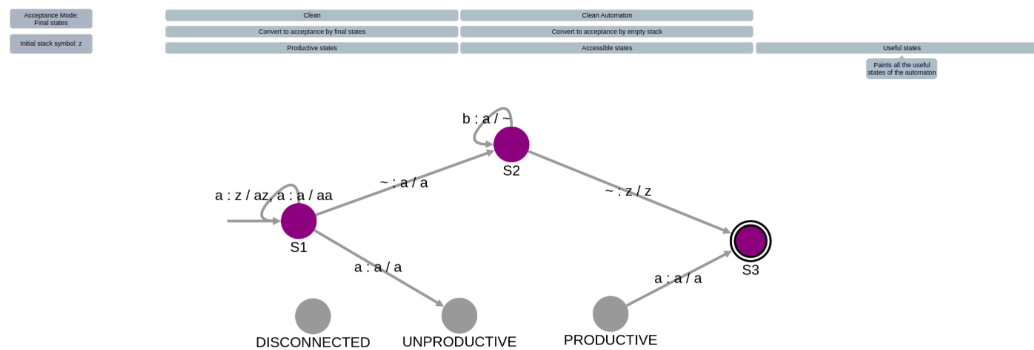

Figure 6.20: Operação de identificação de estados úteis

da página (figura 6.21).

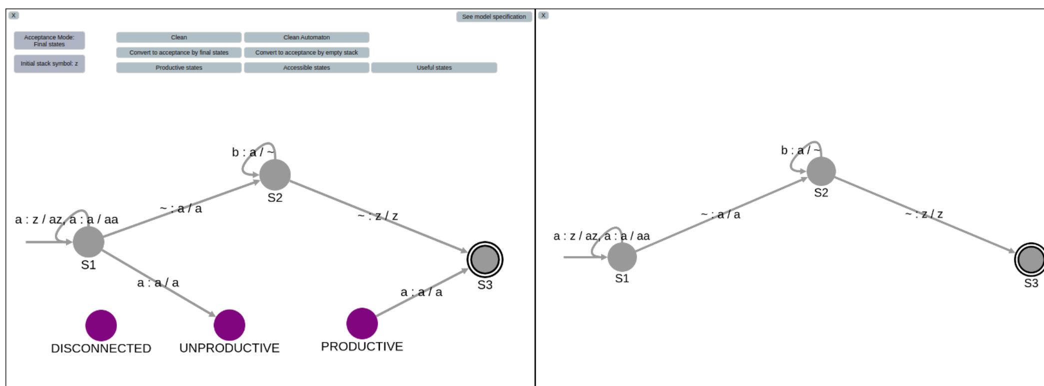

Figure 6.21: Operação de limpeza do autómato

Todas as operações de conversão que abram um novo modelo na secção à direita, permitem passar a usá-la como elemento principal da aplicação, fechando assim a secção principal da esquerda. Assim, é possível realizar operações automáticas tal como a limpeza do autómato acima descrita, assim como transformações/conversões do autómato de pilha noutros modelos como linguagens livres de contexto e autómatos finitos se a operação for

possível, ou seja, se o autómato for de facto equivalente ao autómato finito gerado.

6.5.2 Informação autómato de pilha

Na secção inferior da janela do autómato de pilha é mostrada informação relevante sobre o autómato apresentado na janela principal (figura 6.22). Esta informação é atualizada durante a criação do autómato e permite que o utilizador visualize de forma simples se o autómato construído é determinista, se existe algum estado inútil, se o mesmo pode ser transformado num autómato finito equivalente, o número de estados do autómato e por fim o número de transições do mesmo.

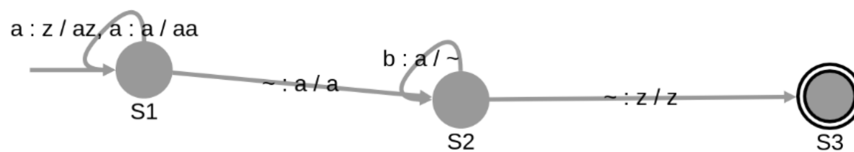

Figure 6.22: Informação do autómato de pilha

6.5.3 Operação conversão aceitação por estados finais

A operação de conversão para estados finais transforma um autómato de pilha no modo de aceitação por pilha vazia e gera um autómato equivalente, ou seja, que aceita a mesma linguagem usando o modo de aceitação por estados finais (figura 6.23). Esta conversão é realizada através da introdução de 2 estados novos, o estado Si e Sf e de várias transições novas. A primeira transição de Si para o estado inicial do autómato original, neste caso o estado START tem o objetivo de empilhar um símbolo especial que indica o fundo da pilha. E as restantes, são conectadas desde todos os estados do autómato original para o novo estado final Sf. Usando este símbolo novo empilhado à priori é possível determinar se o autómato original estaria com a pilha vazia, verificando que o símbolo especial introduzido está no topo da pilha e usar este como um mecanismo para transitar para o estado final.

6.5.4 Operação conversão aceitação por pilha vazia

Esta operação realiza a conversão contrária à anteriormente descrita. Permite ao utilizador converter um autómato de pilha que reconhece palavras usando um mecanismo de aceitação por estados finais noutro autómato equivalente que aceita a mesma linguagem com modo de aceitação por pilha vazia (figura 6.24). Esta transformação é realizada com o auxílio de 2 novos estados, um novo estado inicial *Si* e um estado extra *Sf*. A transição adicional do novo estado inicial *Si* para o estado inicial anterior *START* tem o objetivo

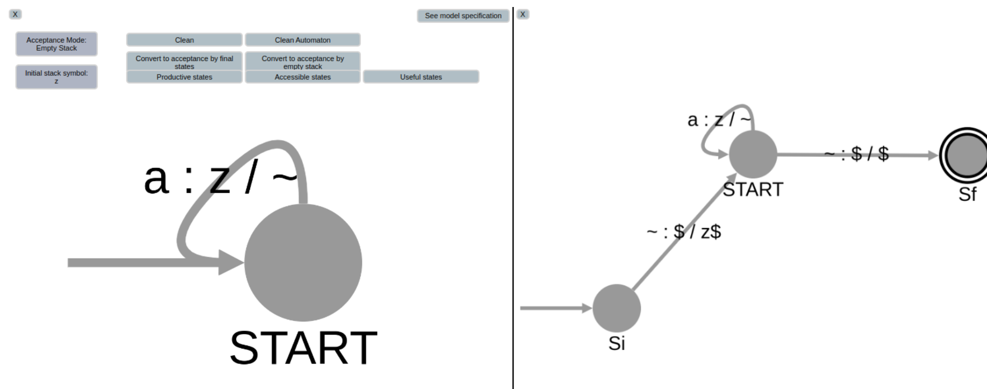

Figure 6.23: Operação de conversão do autómato em aceitação por estados finais

único de empilhar um símbolo especial na pilha, garantindo assim que se o estado inicial do autómato não for estado final que o autómato é equivalente. Sem esta adição, em alguns casos a transformação do autómato não gera um autómato equivalente ao original passando a aceitar a palavra vazia. As transições do autómato dos estados finais do autómato original para o estado *Sf*, assim como as transições que formam um arco de *Sf* para *Sf* têm como objetivo o desempilhamento de todos os símbolos contidos na pilha. Desta forma conseguimos produzir um autómato de pilha equivalente que aceita a mesma linguagem com modo de aceitação por pilha vazia.

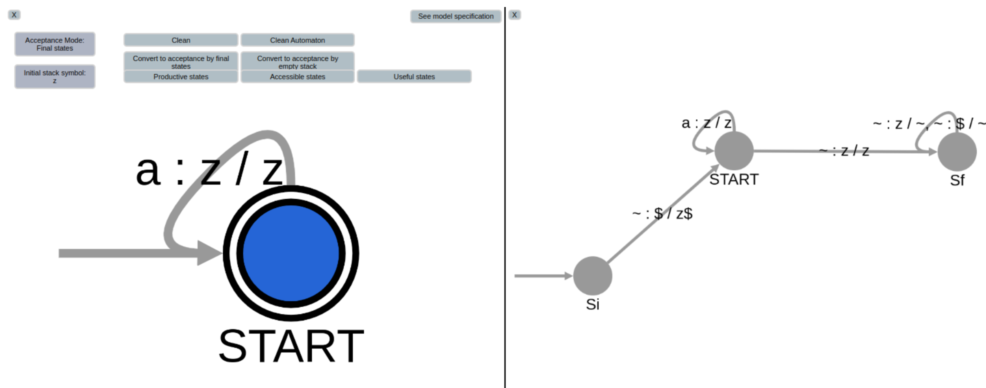

Figure 6.24: Operação de conversão do autómato em aceitação por pilha vazia

7
OCAML-FLAT - Implementação

O OCaml-FLAT e o OFLAT são as duas componentes principais que fazem parte da estrutura já existente deste projeto. O primeiro consiste na lógica principal da ferramenta, e o segundo a aplicação web, que implementa a interface gráfica. Ambas usam o OCaml para toda a construção. Neste capítulo, irá ser abordado a primeira componente que contém as principais operações implementadas para autómatos de pilha.

O OCaml-FLAT é uma biblioteca composta por vários módulos. Esta biblioteca, até ao momento da escrita deste documento, implementa autómatos de pilha, expressões regulares, gramáticas livres de contexto, assim como alguns módulos de apoio, como por exemplo, operações sobre ficheiros de JSON, erros, parsers, entre outros.

Para a implementação do algoritmo de navegação de configurações do autómato de pilha, ou seja o mecanismo de procura através das transições, que é utilizado para todas as principais funções sobre autómatos de pilha (accept, generate, bestPath..) surgiram duas ideias principais para garantir a implementação de um semi-algoritmo, ou seja, um algoritmo que garanta a terminação se o autómato de pilha aceitar a palavra. A primeira solução seria em cada passo recursivo executar duas operações principais. Em primeiro lugar, a partir das configurações correntes obter as novas configurações que consumam um símbolo e de seguida gerar num modo de procura em profundidade todas as configurações *epsilon* alcançáveis, sendo que em cada passo, são adicionadas novas configurações ao set encontrado. Esta segunda operação é uma *closure*, garantindo que na existência de loops são geradas todas as configurações possíveis e para quando não existe nenhuma configuração nova encontrada.

Este algoritmo garante a paragem para os autómatos finitos, sendo que estes têm um conjunto finito de estados e são identificadores únicos das configurações. Por outro lado, para os autómatos de pilha o mesmo não se sucede. Isto deve-se à pilha nos autómatos de pilha, por exemplo, um autómato que tenha uma transição para o mesmo estado, que não consuma qualquer símbolo e que altere o estado da pilha, tal como empilhar um novo símbolo faz com que o algoritmo anteriormente descrito nunca pare, e com que a *closure* nunca "estabilize" (alcance todas as configurações possíveis), ficando assim preso num loop infinito. Posto isto, a solução descrita não é suficiente para garantir a paragem do

algoritmo para todos os autómatos de pilha no caso em que a palavra a ser reconhecida é aceite pelo mesmo.

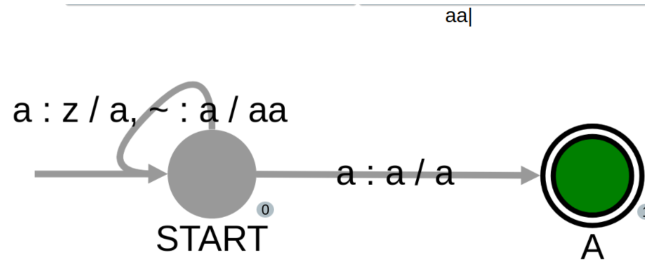

Figure 7.1: Autómato de pilha com transições epsilon problemáticas

De forma a perceber melhor as limitações desta solução, podemos analisar o comportamento no seguinte autómato de pilha da figura 7.1. Usando o algoritmo descrito o primeiro passo será realizar uma closure, percorrendo todas as transições epsilon. A partir da configuração inicial não existe nenhuma, logo o algoritmo continua para o passo seguinte que será percorrer uma transição consumindo um símbolo. O autómato irá consumir 'a' e encontra-se no estado START com o "az"empilhado na pilha. A partir deste momento surge o grande problema deste algoritmo. O próximo passo será realizar uma closure através de transições epsilon, tendo em conta que no topo da pilha existe o símbolo 'a', o autómato segue a transição epsilon que desempilha 'a' e empilha 'aa'. Após esta transição o autómato está novamente numa configuração que pode voltar a seguir esta transição. Encontramos aqui um loop onde a closure não irá apanhar pela constante alteração do estado da pilha, alterando sucessivamente a configuração criando configurações novas. Se esta transição apenas empilhasse 'a', o conteúdo da pilha manteria-se inalterado, e, através da closure o algoritmo iria parar. Esta closure pode ser imaginada como a adição de novas configurações ao set de configurações existentes, quando não são encontradas configurações novas a closure termina. Mas para o caso anterior descrito existem sempre novas configurações pois a pilha está constantemente a crescer com um símbolo adicional 'a'. Este comportamento exemplifica o porquê de não ser possível seguir esta abordagem como solução do problema.

A segunda solução passa por uma procura puramente em largura, em que cada passo, dadas as configurações atuais são geradas as configurações seguintes percorrendo apenas um arco (na representação gráfica do autómato). Deste modo há garantia de paragem quando a palavra a ser reconhecida é aceite pelo autómato, gerando um algoritmo mais robusto que o anteriormente descrito. Este algoritmo também é mais simples que o anterior, mas dado o contexto em que irá ser utilizado tem grandes desvantagens. Para o OFLAT (parte gráfica) o objetivo é visualizar o autómato de pilha a realizar a aceitação de uma palavra de forma interativa em que cada passo o autómato consome um símbolo da palavra. Posto isto, facilmente observamos que a definição de passo no algoritmo

de procura e a mesma para a aplicação gráfica é diferente. Na primeira, um passo representa avançar um arco, para a segunda representa avançar um símbolo consumido (podendo assim avançar vários arcos através das transições epsilon). Esta diferença traz complexidade acrescida, por um lado seria ideal usar a primeira solução dado que esta vai diretamente ao encontro do modo de funcionamento da aplicação gráfica. Por outro lado, não garante um algoritmo suficientemente robusto.

A solução implementada passa por numa primeira fase criar a *search tree*, que representa a procura feita pelo algoritmo até a aceitação ou rejeição da palavra pelo autómato de pilha. E numa segunda fase usar a *search tree* gerada para simular o comportamento esperado na interface gráfica, em que cada passo são visualizadas todas as configurações que o autómato se encontra quando consome um símbolo.

Esta solução será discutida com mais pormenores de seguida, mas para já, iremos percorrer algumas das funções da biblioteca mais simples, por exemplo o accept, que não necessita de criar a *search tree*, assim como os tipos usados para construir o autómato de pilha usado em todas as operações.

7.1 Tipos de dados criados para o autómato de pilha em OCaml

Os tipos principais de dados definidos de maneira a simplificar a manipulação dos autómatos de pilha foram o tipo *t* que representa um autómato de pilha, *transition* que define uma transição do autómato e *configuration* que representa uma configuração.

```
1   type transition =
2         state       (* state *)
3       * symbol      (* current symbol on top of the stack *)
4       * symbol      (* consumed input symbol *)
5       * state       (* next state *)
6       * symbol list (* new top of stack *)
7
8   type transitions = transition set
9
10  type t = {
11      inputAlphabet: symbols;  (* Input Alphabet *)
12      stackAlphabet: symbols;  (* Stack Alphabet *)
13      states: states;         (* States *)
14      initialState: state;    (* Initial state *)
15      initialStackSymbol: symbol; (* Initial Symbol on the Stack *)
16      transitions: transitions; (* Transition relation *)
17      acceptStates: states;    (* Accept states *)
18      criteria: bool          (* true = acceptStates | false = emptyStack *)
19  }
20
21  type stack = symbol list
```

```
22 type word = symbol list
23
24 type configuration =
25       state
26     * stack
27     * word
28 type configurations = configuration set
```

7.2 Operações implementadas

Na implementação das operações sobre autómatos finitos a operação de accept, generate assim como as funções principais de geração de árvores de procura usadas pela ferramenta gráfica foram as mais desafiantes, que requereram mais atenção e precisaram de um esforço acrescido. Estas funções sejam as principais da biblioteca, posto isto serão apresentadas com mais detalhe c maior pormenor. Também irão ser apresentadas as restantes funções implementadas. Como por exemplo, conversões entre os vários tipos da biblioteca, conversões do mecanismo de aceitação do autómato, assim como várias outras funções que têm um output booleano relevantes para caracterizar os autómatos de pilha.

7.2.1 Função accept da biblioteca

A função accept é uma das funções principais da biblioteca, recebe como argumentos uma palavra e um autómato de pilha e retorna um booleano que representa se a mesma foi aceite pelo autómato ou não. Esta função começa por definir uma função auxiliar recursiva *acceptRec* que tem como argumentos um set de configurações e retorna um booleano. A função *acceptRec* em primeiro lugar verifica se o set de configurações está vazio. Se estiver retorna falso, representando que o autómato não alcançou nenhuma configuração aceite. Se o set de configurações não estiver vazio, é verificado se alguma das configurações se encontram em num estado de aceitação, no caso positivo é retornado o booleano *true* representado a aceitação da palavra pelo autómato de pilha. Se nenhuma das configurações se encontrar num estado de aceitação são geradas as configurações seguintes, dando um único passo, ou seja, percorrendo apenas um arco (na representação gráfica do autómato), através da chamada da função auxiliar *advanceOneTransition*, que irá gerar o próximo set de configurações aplicando as transições do autómato a cada uma das configurações atuais. Caso o set das novas configurações seja igual ao set atual é retornado falso, esta verificação faz com que o algoritmo pare quando a aceitação da palavra gera um loop causadas pelas transições epsilon, tornando-o mais robusto. Se o set de configurações for diferente é feita uma chamada recursiva para as novas configurações obtidas, até ser encontradas configurações aceites ou esgotar todas as transições possíveis.

Por fim a função *accept* inicializa o set de configurações com uma configuração apenas com o estado inicial do autómato, o estado inicial da pilha e a palavra e chama a função

acceptRec que irá realizar a aceitação da palavra pelo autómato usando uma abordagem de pesquisa em largura.

```
let accept word pda: bool =
    let rec acceptRec configurations: bool =
        Set.match_ configurations
        (fun () -> false)
        (fun _ _ ->
            if configsAreInAcceptState_ pda.acceptStates pda.criteria
                configurations then true
            else
                let nextConfigs = Set.flatMap (
                    advanceOneTransition pda.transitions
                ) configurations in
                if Set.equals configurations nextConfigs then false
                else acceptRec nextConfigs
        )
    in
        let getInitialConfig =
            Set.make [(pda.initialState, [pda.initialStackSymbol], word)] in
                acceptRec getInitialConfig
```

7.2.2 Função auxiliares para a implementação do accept

A função *advanceOneTransition* é uma função auxiliar usada pelo *accept* que tem como input uma configuração e produz o set de configurações seguintes através da aplicação das transições na mesma. Se a pilha da configuração estiver vazia retorna um conjunto vazio, indicando que não existe nenhuma transição possível a partir da mesma, não sendo gerada nenhuma configuração seguinte, caso contrário devolve as configurações obtidas através da aplicação das transições avançando um arco.

No caso em que a configuração de input da função contenha a palavra vazia são apenas calculadas as configurações seguintes alcançadas por transições que não consomem nenhum símbolo. Para o caso em que a palavra não está vazia, indicando que ainda existem símbolos para consumir, é feita uma união entre as configurações alcançadas por transições que consomem o símbolo seguinte e as configurações que são geradas por transições que não consomem qualquer símbolo. Deste modo, esta função retorna todas as configurações alcançáveis quando aplicada o conjunto de transições válidas para a mesma.

As funções auxiliares *configIsInAcceptState* e *configsAreInAcceptState* verificam se uma ou várias configurações estão num estado aceite pelo autómato de pilha. Ou seja, verificam que a palavra de uma configuração está vazia e simultaneamente se dependendo do critério de aceitação, se a configuração se encontra num estado final ou se a pilha presente

na configuração está vazia. Esta lógica é implementada apenas para uma configuração e a função auxiliar que verifica um conjunto de configurações, faz uso desta verificando se existe alguma configuração que se encontre num estado de aceitação.

```
let configIsInAcceptState_ acceptStates criteria (state, stack, word): bool =
    word = [] && (if criteria then Set.belongs state acceptStates else stack =
        [])

let configsAreInAcceptState_ acceptStates criteria configs: bool =
    Set.exists (configIsInAcceptState_ acceptStates criteria) configs

let advanceOneTransition transitions (state, stack, word): configurations =
    let getNextConfig restStack wordLeft (_,_,_,nextState,toPutInStack) =
        (nextState, toPutInStack@restStack, wordLeft) in
    let buildNewConfigurations restStack restWord validTrns = Set.map
        (getNextConfig restStack restWord) validTrns in
    let getNextConfigs inputSymbol restWord topStack restStack =
        let validTransitions = getValidTransitions state topStack inputSymbol
            transitions in
            buildNewConfigurations restStack restWord validTransitions
    in
    match stack, word with
        | [], _ -> Set.empty
        | s::ss, [] -> getNextConfigs epsilon [] s ss
        | s::ss, w::ww ->
            let nextConfigConsumed = getNextConfigs w ww s ss in
            let nextConfigNotConsumed = getNextConfigs epsilon (w::ww) s ss in
                Set.union nextConfigConsumed nextConfigNotConsumed
```

7.2.3 Conversão de modo de aceitação por pilha vazia para estados finais

A conversão do modo de aceitação por pilha vazia para estados finais é uma operação que gera um novo autómato de pilha que aceita a mesma linguagem que o original mas usa o modo de aceitação por estados finais, contrariamente ao original. a função definida para esta operação é *transformPdaToAcceptStates* e tem como input um autómato de pilha. Esta conversão cria dois estados novos Si e Sf e várias transições para gerar o novo autómato. Em primeiro lugar é colocado o estado Si como novo estado inicial do autómato e criada a transição se Si para o estado inicial do autómato original. Esta transição não consome qualquer símbolo e tem o objetivo de colocar um símbolo especial na pilha. Visto que a operação passa de modo de aceitação por pilha vazia para estados finais, apenas é necessário criar um mecanismo de deteção de pilha vazia. Este novo símbolo especial da pilha que é colocado à priori tem exatamente esse propósito. Visto que já existe uma maneira de detetar a pilha vazia o próximo passo será criar uma transição para cada estado do autómato de pilha original e conectar ao novo estado final Sf. Estas transições

não consomem qualquer símbolo e apenas requerem que o novo símbolo especial da pilha esteja no topo da mesma. Desta forma é gerado um mecanismo que deteta a pilha vazia e "salta" para o estado final. Gerando assim o autómato pretendido.

```
let transformPdaToAcceptStates pda: t =
  let buildNewTransitions si sf: transitions = (*si estado inicial, sf estado
      final*)
    let initialTrsn: transition = (si, stackConverterSymb, epsilon,
        pda.initialState, [pda.initialStackSymbol; stackConverterSymb]) in
    let buildTrasitionToFinalState s = (state s, stackConverterSymb, epsilon,
        sf, [stackConverterSymb]) in
    let buildTransitions states = Set.map buildTrasitionToFinalState states in
       Set.add initialTrsn (buildTransitions pda.states)
  in
  let convertedPda: t = {
    inputAlphabet = pda.inputAlphabet;
    stackAlphabet = Set.add stackConverterSymb pda.stackAlphabet;
    states = Set.union (pda.states) (Set.make [state "Si"; state "Sf"]);
    initialState = state "Si";
    initialStackSymbol = stackConverterSymb;
    transitions = Set.union (pda.transitions) (buildNewTransitions (state
        "Si") (state "Sf"));
    acceptStates = Set.make [(state "Sf")];
    criteria = true
    }
  in
    if pda.criteria then pda else convertedPda
```

7.2.4 Conversão de modo de aceitação de estados finais para pilha vazia

A conversão do modo de aceitação de estados finais para pilha vazia é a operação inversa à anteriormente analisada, a função tem como input um autómato de pilha e gera um autómato de pilha que aceita a mesma linguagem, mas pelo mecanismo de aceitação por estados finais. Esta conversão é realizada com o auxílio de 2 novos estados sendo que em primeiro lugar cria o novo estado Si que irá ser o novo estado inicial do autómato. É definida uma função auxiliar *buildNewTransitions* que constrói as novas transições geradas para a conversão. Em primeiro lugar é criada uma nova transição do novo estado inicial Si para o estado inicial do autómato original que tem como objetivo empilhar um novo símbolo na pilha. Esta operação é essencial e foi explicada em detalhe no capítulo anterior. De seguida são construídas todas as restantes transições entre os estados finais do autómato original e são conectados ao novo estado Sf. Estas transições não consomem qualquer símbolo e têm como objetivo desempilhar o conteúdo da pilha. E por fim, são criadas também as transições do estado Sf para Sf com o mesmo objetivo de desempilhar os símbolos da pilha. Visto que são criadas estas transições a partir dos estados finais do

autómato original, é fácil perceber que o objetivo seria caso a configuração no autómato original estivesse num estado de final, o equivalente para este autómato convertido será o desempilhamento do conteúdo da pilha de forma a ser aceite pelo mesmo em modo de aceitação por pilha vazia.

```
1  let transformPdaToAcceptEmptyStack pda: t =
2     let newStackAlphabet = Set.add stackConverterSymb pda.stackAlphabet in
3     let buildNewTransitions si sf: transitions = (*si estado inicial, sf estado
          final*)
4        let initialTrsn: transition = (si, stackConverterSymb, epsilon,
             pda.initialState, [pda.initialStackSymbol; stackConverterSymb]) in
5        let buildFinalTransitionsToConsumeStack stackAlphabet: transitions =
6           Set.map ( fun symbStack -> (sf, symbStack, epsilon, sf, []) )
               stackAlphabet
7        in
8        let buildTrasitionsFromAcceptState acceptState: transitions =
9           let symbsStack = getStackSymbols pda.transitions in
10             Set.map ( fun symbStack -> (acceptState, symbStack, epsilon, sf,
                  [symbStack])) symbsStack
11       in
12       let buildTransitions states: transitions = Set.flatten (Set.map
             buildTrasitionsFromAcceptState pda.acceptStates) in
13          Set.union (Set.add initialTrsn (buildTransitions pda.states))
               (buildFinalTransitionsToConsumeStack newStackAlphabet)
14    in
15    let convertedPda: t = {
16       inputAlphabet = pda.inputAlphabet;
17       stackAlphabet = newStackAlphabet;
18       states = Set.union (pda.states) (Set.make [state "Si"; state "Sf"]);
19       initialState = state "Si";
20       initialStackSymbol = stackConverterSymb;
21       transitions = Set.union (pda.transitions) (buildNewTransitions (state
             "Si") (state "Sf"));
22       acceptStates = Set.empty;
23       criteria = false
24       }
25    in
26       if pda.criteria then convertedPda else pda
```

7.2.5 Análise sintáctica, equivalência autómato finito

A operação definida pela função *isFiniteAutomaton* permite analisar se o autómato de pilha pode ser convertido num autómato finito equivalente. No entanto, esta verificação é apenas sintática. Assim sendo, pode de facto existir um autómato finito que aceita a mesma linguagem mas esta função pode, no entanto, devolver falso indicando que a

conversão não é possível. A operação limita-se a verificar se o autómato de pilha usa a pilha em alguma transição, alterando o estado da mesma. A função retorna true indicando que existe um autómato finito equivalente apenas se para todas as transições o estado da pilha mantém-se inalterado, ou seja, sem empilhar nem desempilhar símbolos da mesma.

```
let isFiniteAutomaton pda: bool =
   let validateTransition (_,a,_,_,b) = [a] = b in
      Set.for_all validateTransition pda.transitions
```

7.2.6 Operação de verificação de existência de não determinismo

A função *isDeterministic* tem como input um autómato de pilha retornando *true* se o mesmo for determinista, *false* caso contrário. Esta análise é realizada através das transições do autómato, por definição este é não determinista se para uma dada configuração existir mais que uma transição válida. Esta função define a função auxiliar *trnsFromStateAndSymbs* que tem com input um estado e um símbolo do alfabeto da pilha e verifica se para o mesmo estado existe alguma transição que desempilhe e consuma o mesmo símbolo. Se existir, isto indica que o autómato é não determinista, pois existe mais que um "caminho" para uma configuração. Para além disto, esta função auxiliar também engloba o símbolo especial epsilon na procura de transições, ou seja, devolve todas as transições que possam gerar 2 opções válidas para uma mesma configuração, tornando o autómato não determinista. Com a definição desta função auxiliar o passo seguinte é percorrer todos os símbolos do alfabeto do autómato e confirmar se existe mais do que 1 transição para este usando a função anteriormente descrita. De seguida, a função auxiliar *isStateNonDeterministic* alcança mais uma camada na abstração do problema e com o auxílio da função imediatamente anterior descrita, percorre todos os estados do autómato e confirma que apenas existe um caminho para uma dada configuração.

Esta função está à procura de um caso que verifique o não determinismo. Está implementada desta forma pois para ser determinista é preciso confirmar todos os casos, verificando que de facto nenhuma das transições torna o autómato não determinista. Por outro lado, quando a procura é feita pelo não determinismo basta encontrar uma para a função parar, isto faz com que seja mais eficiente na busca, apenas percorre todas as opções no caso que, de facto, o autómato é determinista e foi necessário realizar a procura até à exaustão.

```
let isDeterministic pda: bool =

   (*returns the trasitions for a given state, stackSymbol and inputSymbol,
      including epsilon transitions*)
   let trnsFromStateAndSymbs st stackSymb inputSymb =
      Set.filter (fun (s1,stackS,symb,_,_) -> (st = s1) && (stackSymb = stackS)
         && (symb = inputSymb || symb = epsilon)) pda.transitions
   in
```

```
7
8     (*Validates if there is more than one transition possible for a given state,
          stackSymbol*)
9     let isnotDeterministicForStateAndStackSymbol st stackSymb =
10       Set.exists (fun symb -> Set.size (trnsFromStateAndSymbs st stackSymb
             symb) > 1) pda.inputAlphabet
11    in
12
13    (*Validates if a state is non deterministic*)
14    let isStateNonDeterministic st =
15       Set.exists (fun stackSymb -> isnotDeterministicForStateAndStackSymbol st
             stackSymb) pda.stackAlphabet
16    in
17       not (Set.exists (fun st -> isStateNonDeterministic st) pda.states)
```

7.2.7 Conversões entre os tipos existentes na biblioteca

A biblioteca suporta conversões entre os tipos implementados, foram implementadas várias operações de conversão entre autómato de pilhas e os vários tipos já existentes, por exemplo, autómatos finitos, gramáticas livres de contexto e expressões regulares.

7.2.7.1 Autómato de pilha para autómato finito

A conversão de autómato de pilha para autómato finito é apenas uma truncagem do próprio autómato de pilha, isto é, dado que o autómato de pilha pode ser visto como uma extensão do autómato finito, esta conversão apenas mapeia as transições do PDA em transições simplificadas sem o uso da pilha. Para além disso é retirado o alfabeto da pilha e o critério de aceitação.

```
1  let pda2fa pda: FiniteAutomaton.model =
2     let transitionsFa trns = Set.map ( fun (s1,_,a,s2,_) -> (s1,a,s2) ) trns in
3     let fa: FinAutTypes.t = {
4           alphabet = pda.inputAlphabet;
5           states = pda.states;
6           initialState = pda.initialState;
7           transitions = transitionsFa pda.transitions;
8           acceptStates = pda.acceptStates
9        } in
10    new FiniteAutomaton.model (Arg.Representation fa)
```

7.2.7.2 Autómato finito para autómato de pilha

A operação de conversão de autómato finito para autómato de pilha é a conversão inversa da anteriormente descrita. O autómato finito é promovido a autómato de pilha,

adicionando um símbolo único ao alfabeto da pilha onde é usado em todas as transições geradas para o mesmo. Ou seja, as transições são equivalentes ao autómato finito com a adição de o desempilhamento e empilhamento de um símbolo constante. Desta forma o autómato de pilha não está a realizar nenhuma operação à estrutura de dados, sendo de facto equivalente ao autómato finito dado. Para esta operação foi definido que o critério de aceitação é o modo de aceitação por estados finais, pois é também o modo como o autómato finito opera.

```
let fa2pda (fa : FinAutTypes.t ): t =
    let upgradeTransitions trns = Set.map ( fun (s1,symb,s2) ->
        (s1,stackSpecialSymb,symb,s2,[stackSpecialSymb]) ) trns in
        {
            inputAlphabet = fa.alphabet;
            stackAlphabet = Set.make [stackSpecialSymb];
            states = fa.states;
            initialState = fa.initialState;
            initialStackSymbol = stackSpecialSymb;
            transitions = upgradeTransitions fa.transitions;
            acceptStates = fa.acceptStates;
            criteria = true
        }
```

7.2.7.3 Autómato de pilha para expressão regular

A operação de conversão de autómato de pilha para expressões regulares utiliza a função previamente existente na biblioteca de conversão entre autómato finito e expressões regulares. Deste modo a função *pda2re* converte o autómato de pilha em autómato finito pelas funções de conversão anteriormente descritas e de seguida converte o autómato finito gerado em expressão regular.

```
let pda2re pda: RegularExpression.model =
    let fa = pda2fa pda in
        fa2re fa
```

7.2.7.4 Expressão regular para autómato de pilha

A operação de expressão regular para autómato de pilha é idêntica à operação de conversão anterior descrita, mas realiza o caminho inverso. Transformando a expressão regular em autómato finito através da chamada de uma função previamente existente na biblioteca e de seguida este é transformado em autómato de pilha.

```
let re2pda (re: RegularExpression.model): PushdownAutomaton.model =
    let fa = re2fa re in
        fa2pda fa
```

7.2.7.5 Gramática independente de contexto para autómato de pilha

A operação de conversão entre gramática independente de contexto para autómato de pilha é uma operação relativamente simples onde é gerado um autómato com apenas um estado usando o critério de aceitação por pilha vazia. Para cada regra da gramática independente de contexto é gerado uma transição equivalente que não consome nenhum símbolo, a parte esquerda da regra tem de estar no topo da pilha e o lado direito da regra é empilhado. Estas transições geradas facilmente percebe-se o propósito, visto que está a expandir a própria regra colocando na pilha a sua expansão (figura 7.2). E adicionalmente uma transição para cada símbolo final, onde o mesmo é consumido e desempilhando-o (figura 7.2).

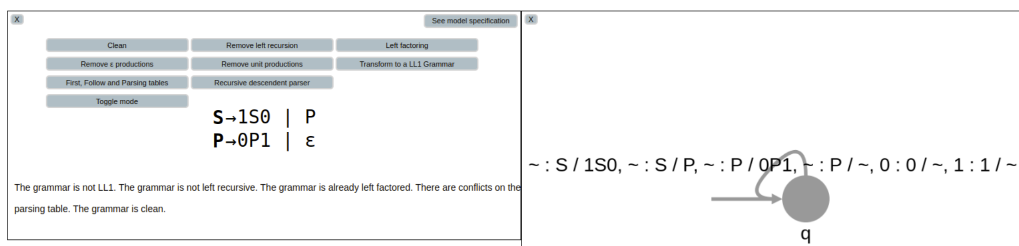

Figure 7.2: Conversão de CFG para PDA

```
1  let cfg2pda cfg: PushdownAutomaton.model =
2     let open CFGTypes in
3     let cfgRep: CFGTypes.t = cfg#representation in
4
5     let computeState = state "q" in
6
7     let makeNewTransition symbToConsume topStackSymbol toPutInStack:
          PushdownAutomaton.transition =
8        (computeState, topStackSymbol, symbToConsume, computeState, toPutInStack)
9     in
10
11    let buildTransitions: PushdownAutomaton.transitions =
12       let transitionsRules = Set.map (fun r -> makeNewTransition epsilon r.head
             r.body) cfgRep.rules in
13       let transitionsFinalSymb = Set.map (fun alph -> makeNewTransition alph
             alph []) cfgRep.alphabet in
14
15          Set.union transitionsRules transitionsFinalSymb
16    in
17
18    let pda: PushdownAutomaton.t = {
19       inputAlphabet = cfgRep.alphabet;
20       stackAlphabet = Set.union cfgRep.alphabet cfgRep.variables;
21       states = Set.make [computeState];
```

```
        initialState = computeState;
        initialStackSymbol = cfgRep.initial;
        transitions = buildTransitions;
        acceptStates = Set.empty;
        criteria = false
    } in
    new PushdownAutomaton.model (Arg.Representation pda)
```

7.2.8 Operações de categorização de estados

Foram implementadas várias operações de categorização dos estados do autómato que têm como por base o filtro e agrupamento de estados com características semelhantes. visto que estas operações são semelhantes às previamente implementadas pelo módulo do autómato finito, as mesmas foram reutilizadas de maneira a evitar duplicação de código na biblioteca. Desta maneira, para as operações analisadas de seguida, foram realizadas à priori uma conversão de autómato de pilha para autómato finito de maneira a reutilizar as mesmas. A categorização dos estados é feita meramente pelas ligações dos estados e pelos estados ligado através das transições, ou seja, na perspetiva do autómato de pilha ser um grafo direcionado, não sendo usadas operações sobre a pilha para realizar nenhuma pesquisa de estados alcançáveis, produtivos ou úteis.

7.2.8.1 Categorização de estados alcançáveis

Os estados alcançáveis, por definição, são estados que são alcançáveis a partir do estado ínicial.

```
let reachable s pda: states =
    let fa : FiniteAutomaton.model = pda2fa pda in
    fa#reachable (s)
```

7.2.8.2 Categorização de estados produtivos

Os estados produtivos, por definição, são estados que a partir dos mesmo é possível alcançar um estado final.

```
let productive pda: states =
    let pdaTransformed = transformPdaToAcceptStates pda in
    let fa : FiniteAutomaton.model = pda2fa pdaTransformed in
    Set.inter fa#productive pda.states
```

7.2.8.3 Categorização de estados úteis

Os estados úteis, por definição, são estados que são simultaneamente alcançáveis e produtivos.

```
1 let getUsefulStates pda: states =
2    let pdaTransformed = transformPdaToAcceptStates pda in
3    let fa : FiniteAutomaton.model = pda2fa pdaTransformed in
4    Set.inter fa#getUsefulStates pda.states
```

7.2.8.4 Categorização de estados inúteis

Os estados inúteis, inversamente ao conjunto definido anteriormente, são os estados que não são utéis, isto é, não são alcançáveis ou produtivos.

```
1 let getUselessStates pda: states =
2    let pdaTransformed = transformPdaToAcceptStates pda in
3    let fa : FiniteAutomaton.model = pda2fa pdaTransformed in
4    Set.inter fa#getUselessStates pda.states
```

7.2.9 Limpeza dos estados inúteis do autómato

A função *cleanUselessStates* recebe um autómato de pilha e juntamente com as operações definidas anteriormente de categorização de estados realiza uma limpeza do autómato eliminando todos os estados inúteis. Esta operação usa as funções definidas no módulo de autómatos finitos, convertendo assim à priori o autómato de pilha para autómato finito e realizando a operação de clean previamente existente na biblioteca. O único detalhe que tem de ser tido em conta ao realizar esta operação será quando o autómato de pilha se encontra com o critério de aceitação em modo de pilha vazia. Neste caso, dado que para calcular os estados produtivos é necessário estar no modo de aceitação por estados finais, é realizada a operação de conversão do modo de aceitação para estados finais e de seguida é aplicada a operação de limpeza do autómato. Após a limpeza do autómato é realizada a construção das transições originais do autómato de pilha, para as transições que se mantiveram após a limpeza dos estados inúteis.

```
1 let cleanUselessStates pda: t =
2    let getEquivalentTransitions (s1,sym,s2) =
3       Set.filter (fun (a,b,c,d,e) -> a = s1 && c = sym && d = s2 )
            pda.transitions in
4    let getCleanedTransitions faCleanedTransitions =
5       Set.flatMap getEquivalentTransitions faCleanedTransitions in
6    let pdaTransformed = transformPdaToAcceptStates pda in
7    let fa : FiniteAutomaton.model = pda2fa pdaTransformed in
8    let faClean: FiniteAutomaton.model = fa#cleanUselessStates in
```

```
9       {
10          inputAlphabet = pda.inputAlphabet;
11          stackAlphabet = pda.stackAlphabet;
12          states = Set.inter faClean#representation.states pda.states;
13          initialState = pda.initialState;
14          initialStackSymbol = pda.initialStackSymbol;
15          transitions = getCleanedTransitions
                faClean#representation.transitions;
16          acceptStates = Set.inter fa#representation.acceptStates
                pda.acceptStates;
17          criteria = pda.criteria
18      }
```

7.3 Operações principais de utilização e criação da *Search Tree*

O OFLAT, como explicado anteriormente, assenta sobre uma lógica em que cada passo, da operação de aceitação de uma palavra, consome um símbolo. Isto implica que o algoritmo, em cada passo, para além de realizar as transições que consomem um símbolo terá também de avançar para todas as configurações alcançáveis por transições que não consomem qualquer símbolo. Esta segunda parte traduz-se numa procura em profundidade, pois é necessário chegar a todas as configurações possíveis antes de continuar a procura em largura realizada pelo primeiro passo através de transições que consomem um símbolo. Se esta lógica for traduzida para um algoritmo de forma direta, existem casos em que se torna problemática, entrando num loop infinito em busca das configurações alcançáveis por transições que não consomem símbolos. Logo, este algoritmo que é derivado de forma direta do comportamento do autómato não garante a paragem no caso da palavra a ser reconhecida ser aceite pelo autómato de pilha.

A solução para este problema passa por eliminar a busca em profundidade e apenas realizar uma pesquisa em largura em todos os passos do autómato. Assim, em cada passo todas as configurações avançam um arco. Desta forma, mesmo que existam transições que formem um ciclo que seja infinito, há sempre configurações que irão estar a evoluir em busca de um estado de aceitação, garantido a paragem do algoritmo quando o autómato aceita a palavra que está a ser reconhecida. Por outro lado, quando a palavra não é reconhecida pelo autómato, poderá haver casos em que o algoritmo nunca irá parar. Neste caso, não é possível confirmar que as configurações do autómato, de facto, não estão a progredir para um estado em que são aceites, logo, o algoritmo não para.

O comportamento pretendido para a aplicação gráfica e a solução anteriormente explicada que garante paragem não são compatíveis. Ou seja, ao realizar esta procura em largura, se a mesma for mostrada passo a passo pela aplicação gráfica, iremos visualizar um autómato possivelmente com configurações distintas que contenham palavras em estados diferentes. Um dos requisitos do OFLAT é a visualização de todas as configurações

que se encontram na mesma etapa de consumo da palavra. Por outras palavras, o que todas as configurações têm em comum em todos os passos do autómato, para a ferramenta gráfica é a palavra estar exatamente igual em todas as configurações.

Para manter o algoritmo o mais robusto possível e de forma simultânea apresentar o comportamento pretendido na ferramenta gráfica foi necessário realizar uma pré-computação do comportamento do autómato, guardando o histórico de execução. Através deste histórico, composto por configurações ligadas entre si, é possível simular o comportamento pretendido para a ferramenta gráfica. Este histórico é constituído por uma árvore de configurações, onde a raiz é a configuração inicial do autómato e que em cada passo são geradas as novas configurações do próximo nível da árvore de procura. Como resultado temos uma árvore em que cada arco liga duas configurações, onde pode ter sido consumido um símbolo ou não dependendo se as configurações têm a mesma palavra. Com esta árvore criada e com o acesso à informação se foi consumido um símbolo ou não entre duas configurações, temos toda a informação necessária para realizar a simulação pretendida, bastando percorrer a estrutura de dados gerada de forma conveniente.

7.3.1 Tipos auxiliares definidos para o algoritmo de geração da *Search Tree*

Foram definidos dois tipos de árvores que facilitam a implementação dos algoritmos de construção e manipulação do histórico do comportamento do autómato de pilha na operação de accept. O primeiro *searchTreeRef* é uma árvore N-ária que contém os tipos *AcceptLeafRef* e *NotAcceptLeafRef*, compostos por uma única configuração, que representam as folhas da árvore, indicando se a configuração foi aceite ou não pelo autómato e o tipo *NodeRef* que representa uma configuração intermédia na procura, que é composta por uma configuração e uma lista de configurações seguintes. O segundo tipo *searchTree* é utilizado nas funções de pós processamento da árvore gerada, que têm como input a árvore do tipo searchTreeRef e geram uma nova árvore do tipo *searchTree*. Esta segunda estrutura de dados definida contém o tipo adicional *BestNode* que tem a mesma função que um nó intermédio, mas indica que a configuração contida no mesmo faz parte de um caminho que é aceite pelo autómato de pilha.

A função implementada para a criação da árvore de procura está feita num estilo de programação indutivo. Para este problema em específico, que tem como objetivo realizar uma procura num grafo e retornar a árvore de procura gerada, a solução intuitiva e simples gera sempre uma procura em profundidade, não sendo trivial a implementação de uma solução que realize uma busca em largura e retorne os caminhos percorridos. De forma a manter a solução o mais simples possível foram utilizadas referências na definição do tipo *searchTreeRef* para a função que gera a árvore de procura. Por outro lado, as restantes funções implementadas, por exemplo, a operação que procura o melhor caminho (caminho mais curto até aceitação da palavra), têm como input a árvore gerada, desta maneira, já não necessitam de realizar uma pesquisa em largura, pois a árvore não contém ciclos e é finita. Por estas mesmas razões o segundo tipo definido *searchTree* não

necessita de ter referências e é usado por todas as funções que realizam manipulações da árvore de procura inicialmente obtida.

```
1  type searchTreeRef =
2        NotAcceptLeafRef of configuration
3      | AcceptLeafRef of configuration
4      | NodeRef of configuration * searchTreeRef set ref
5
6  type searchTree =
7        NotAcceptLeaf of configuration
8      | AcceptLeaf of configuration
9      | Node of configuration * searchTree set
10     | BestNode of configuration * searchTree set
```

7.3.2 Operação de criação da *Search Tree*

A função *buildSearchTree* tem como input um autómato de pilha e uma palavra e retorna uma árvore que representa todos os caminhos que são percorridos ao realizar a operação de accept do autómato. Esta função começa por criar a configuração inicial, contendo o estado inicial do autómato, o símbolo inicial da pilha e a palavra recebida como input da função. Se esta configuração já se encontrar num estado de aceitação a função retorna de imediato o resultado pretendido, caso contrário é inicializado um nó que representa a raiz da árvore, com a configuração inicial, e de seguida é chamada a função auxiliar recursiva *buildSearchTree*.

A função recursiva realiza uma procura em largura construindo um nível adicional da árvore em cada iteração. O algoritmo retorna quando produz um novo nível da árvore que contem pelo menos uma configuração num estado aceite pelo autómato, ou se o conjunto de configurações seguintes for vazio, representando a inexistência de caminhos a seguir. Em cada passo são geradas o conjunto de configurações seguintes através da função auxiliar *getNextConfigsPair* devolvendo as mesmas em conjunto com o nó que foi expandido. Desta maneira é realizada uma associação entre cada configuração atual e o conjunto de configurações geradas por esta. Este par é passado para a função *buildtree* que tem a função de adicionar ao nó atual as respectias configurações futuras, sendo assim criada a árvore de execução do autómato. Por fim, é apenas necessário realizar a chamada recursiva com o novo conjunto de nós gerado pela expansão de todas as configurações atuais, aplicando uma operação que filtra eventuais estados que foram rejeitados e contenham o tipo *NotAcceptLeaf*.

```
1  let buildSearchTree (word: word) pda: searchTreeRef =
2     let rec buildSearchTree (forest: searchTreeRef set) =
3        Set.match_ forest
4        (fun () -> ())
5        (fun _ _ ->
```

```
6           let pairTreesAndNextConfigs = getNextConfigsPair forest
                pda.transitions in
7           let nextNodes = buildTree pda.acceptStates pda.criteria
                pda.transitions pairTreesAndNextConfigs in
8           if foundAcceptConfig nextNodes || Set.equals forest nextNodes then ()
9           else buildSearchTree (filterSearchNode nextNodes)
10      )
11   in
12      let initialConfig = (pda.initialState, [pda.initialStackSymbol], word) in
13      if configIsInAcceptState_ pda.acceptStates pda.criteria initialConfig
            then AcceptLeafRef(initialConfig)
14      else
15          let searchTree: searchTreeRef = NodeRef(initialConfig, ref (Set.make
                []))  in
16              buildSearchTree (Set.make [searchTree]);
17              searchTree
```

7.3.2.1 Funções auxiliares definidas para o função *buildSearchTree*

As funções *getNodeElements*, *getNextConfigsPair*, *buildTree* e *buildEndNode* são usadas pela função principal *buildsearchTree* e realizam operações auxiliares, distribuindo e fatorizando o código de forma a que a implementação seja simples e clara de perceber e ser modificada.

A função *getNodeElements* apenas serve o propósito de retornar os elementos de cada nó. Tem como input um nó da árvore de procura, do tipo *SearchTreeRef*, e retorna os elementos do mesmo, a configuração atual e os nós seguintes.

```
1 exception ShouldNotHappen
2 let getNodeElements = function
3     | NodeRef(config, setNodes) -> (config, setNodes)
4     | _ -> raise ShouldNotHappen
```

A função *getNextConfigsPair* produz pares de nós e configurações seguintes. Tem como input um conjunto de nós da árvore, e com o auxílio da função *advanceOneTransition*, apresentada anteriormente, gera o próximo nível de configurações, sendo representada pelo próximo nível da árvore de procura.

```
1 let getNextConfigsPair searchTreeSet transitions =
2     Set.map (fun tree ->
3         let (config, _) = getNodeElements tree in
4         let nextConfigs = advanceOneTransition transitions config in
5             (tree, nextConfigs)
6     ) searchTreeSet
```

Por fim a função *buildTree* e *buildEndNode* são usadas para construir a árvore de procura. Para cada par de nó e configurações seguintes geradas para o mesmo são construidos os

nós seguintes da árvore e adicionadas ao conjunto de nós atuais. Através da função auxiliar *buildEndNode*, para cada configuração seguinte é verificada se a configuração se encontra num estado aceite, sendo assim, construída uma folha da árvore com o tipo *AcceptLeafRef*. No caso negativo, é realizada a verificação se a configuração contém a palavra vazia e não existe nenhuma transição possível para a mesma, construindo a folha *NotAcceptLeaf*. A verificação de transições válidas para a configuração é necessária pois, apesar da mesma se encontrar com a palavra completamente consumida, ainda poderá haver transições que não consomem qualquer símbolo, que façam com que a mesma progride até uma configuração aceite. Por fim, se os dois casos anteriores não se verificarem é construído o tipo *NodeRef* que representa um passo intermédia na computação dos caminhos percorridos pelo autómato.

```
1  let configHasNoSymbolToConsume (_,_,word) = word=[]
2
3  let buildEndNode acceptStates criteria transitions config =
4    if configIsInAcceptState_ acceptStates criteria config then
         AcceptLeafRef(config)
5        else if configHasNoSymbolToConsume config && (advanceOneTransition
             transitions config) = Set.empty then NotAcceptLeafRef(config)
6        else NodeRef(config, ref (Set.make[]))
7
8  let buildTree acceptStates criteria transitions pairTreesAndNextConfigs =
9     Set.flatMap (fun (tree, nextConfigs) ->
10       let (_, setNodes) = getNodeElements tree in
11       let newNodes = Set.map (buildEndNode acceptStates criteria transitions)
             nextConfigs in
12       setNodes := newNodes;
13          newNodes
14    ) pairTreesAndNextConfigs
```

7.3.3 Tratamento da árvore de procura inicialmente obtida

Com a operação de criação da árvore de procura é gerada uma árvore que contém todos os caminhos percorridos durante a operação de aceitação do autómato. De maneira a simplificar as operações futuras o objetivo é criar uma árvore que tem quatro tipos diferentes, dois tipos para as folhas, que indica que a configuração contida na mesma foi aceite ou rejeitada e para os passos intermédios também dois tipos de nós que indicam que a configuração presente no mesmo, faz, ou não, parte de um caminho que irá levar a um estado de aceitação. A motivação desta representação advém da simplicidade obtida pela existência de um tipo de dados compacto que exprima todo o histórico gerado pelo autómato durante a aceitação. Uma vez gerada esta árvore é possível usá-la para mostrar passo a passo, num modo de procura em largura, o comportamento que o autómato realizou ao aceitar uma palavra.

A função *mapBestPath* tem como input uma árvore do tipo *searchTreeRef*, criada pela operação de aceitação que gera o histórico do comportamento do autómato, e retorna uma árvore do tipo *searchTree*. A grande diferença entre estes dois tipos de árvores é o tipo usado para os nós intermédios, onde a segunda irá conter informação adicional indicando que a configuração encontra-se, ou não, num caminho que irá ser aceite.

A implementação é realizada com uma procura em profundidade, não tendo quaisquer problemas, pois está a ser executada num grafo que não contém ciclos e é finito. Esta operação mapeia os nós da árvore de input do tipo *searchTreeRef* em nós do tipo *searchTree*, e para cada configuração, caso o nó seguinte faça parte de um caminho que é aceite, o nó atual também é criado como tal, sendo desta forma criados os nós intermédios que irão distinguir quais caminhos, quando percorridos, levam a uma configuração que é aceite pelo autómato.

A função *getSearchTree* é uma função que engloba a operação de criação da árvore de procura e de seguida manipula a mesma retornando a árvore de o novo tipo pretendido, com as características anteriormente enumeradas. Esta função irá ter como input uma palavra e um autómato de pilha e irá retornar uma árvore do tipo *searchTree*.

```
1  let rec mapBestPath: searchTreeRef -> searchTree = function
2      | AcceptLeafRef c -> AcceptLeaf(c)
3      | NotAcceptLeafRef c -> NotAcceptLeaf(c)
4      | NodeRef(c, treeSet) ->
5          let nextLayer = Set.map mapBestPath !treeSet in
6          if hasBestPath nextLayer then BestNode(c, nextLayer) else Node(c,
               nextLayer)
7
8  let getSearchTree word pda =
9      buildSearchTree word pda |> mapBestPath
```

7.3.4 Operação de procura do melhor caminho

A operação de procura do melhor caminho é usada para construir o caminho mais curto, aceite pelo autómato de pilha. Esta função é usada na ferramenta gráfica para apresentar em simultâneo as configurações em cada passo da aceitação e uma configuração especial que irá progredir até ser aceite pelo autómato. O resultado deste algoritmo é uma lista de configurações que irá ser usada para apresentar no grafo a configuração especial, que tem como objetivo criar uma visualização mais completa do comportamento do autómato.

A função tem como input uma árvore de procura e retorna um *path* que representa o caminho mais curto até à aceitação. Como pré condição, esta função espera receber uma árvore que já foi pré-computada que irá conter um conjunto de nós do tipo *BestNode* ou *Node* em candeia, caso exista um caminho aceite ou não respetivamente, terminando com uma folha do tipo *AcceptLeaf* ou *NotAcceptLeaf*. Esta condição faz com que o algoritmo pare imediatamente caso a árvore gerada não tenha nenhum caminho aceite, pois a raiz

irá ser do tipo *Node* se não existir qualquer caminho aceite e do tipo *BestNode* caso exista.

Para os casos base definidos na função, usando um mecanismo de match do OCaml, caso o nó seja do tipo *AcceptLeaf* é garantido que a configuração correspondente a este nó se encontre num estado aceite, logo apenas terá de ser retornado uma lista com a mesma. Nos casos em que é do tipo *NotAcceptLeaf* e *Node*, pela razão como a árvore é construída, temos a garantia que não existe um caminho que seja aceite, retornando uma lista vazia. Por fim para o caso de o nó ser do tipo *BestNode*, significa que a configuração atual é um caminho aceite pelo autómato e que ainda não foi alcançado o estado de aceitação. Portanto, neste caso é realizado o passo indutivo que consiste em obter o nó seguinte e a palavra do mesmo, sendo definida como *nextWord* e a partir destes dois componentes irá ser realizado um de dois casos: adicionar à cabeça da lista a configuração atual ou ignorar a configuração atual e continuar o empilhamento de configurações. De maneira a entender a motivação que gera esta lógica é necessário compreender qual será o comportamento esperado ao percorrer um caminho de forma interativa na ferramenta. Para aplicação gráfica, o comportamento pretendido é que a configuração encontre-se num estado em que não é possível avançar sem o consumo de um símbolo, isto significa, que após obtidas as configurações que, de facto, consomem um símbolo, é esperado que todas as configurações "epsilon" sejam realizadas, mantendo o estado em cada passo, da operação accept, bloqueado à espera do próximo símbolo a ser consumido. Desta forma, o passo indutivo da função, tem de diferenciar os casos em que é consumido um símbolo e os casos em que não é. Logo, se a palavra da configuração atual for igual à seguinte, a configuração atual não é adicionada ao melhor caminho. Esta lógica é realizada através de uma chamada recursiva que não concatena a mesma ao resultado. Este comportamento faz com que haja uma compressão no caminho efetuado pelo autómato, eliminando configurações intermédias que não consomem qualquer símbolo. Sendo este o comportamento esperado na visualização do autómato de pilha em execução na ferramenta gráfica. No caso do passo indutivo em que a configuração atual e a seguinte diferem na palavra, esta tem de ser adicionada ao caminho gerado, pois cada um destes passos representa, de facto, um passo na ferramenta gráfica.

```
1  type path = configuration list
2
3  let isBestNode = function | AcceptLeaf _ -> true | BestNode _ -> true | _ ->
       false
4
5  let getBestNodeAndWord setNodes =
6       let node = Set.find isBestNode setNodes in
7         (node, getWordFromNode node)
8
9  let rec buildBestPath: searchTree -> path = function
10     | AcceptLeaf(c) -> [transformConfig c]
11     | NotAcceptLeaf(c) -> []
12     | Node(c, treeSet) -> []
```

```
13      | BestNode((_,_,word) as c, treeSet) ->
14         let (nextBestNode, nextWord) = getBestNodeAndWord treeSet in
15         if (word = nextWord) then buildBestPath(nextBestNode)
16         else (transformConfig c)::buildBestPath(nextBestNode)
17
18 let getBestPath word pda =
19         getSearchTree word pda |> buildBestPath
```

7.4 Operação de geração de palavras reconhecidas pelo autómato de pilha

A função *generate* está programada de acordo com a mesma estrutura de funcionamento da função *accept*. A diferença essencial é que o generate explora todas as possibilidades do espaço de pesquisa, ao contrário do *accept*, em que a palavra que está a ser reconhecida limita fortemente as possibilidades a explorar.

A operação de *accept* tem como argumentos um autómato de pilha (*pda*) e um inteiro (*length*) e retorna o conjunto de palavras que fazem parte da linguagem do autómato que têm tamanho *length*.

7.5 Testes Unitários

A biblioteca foi testada com testes unitários para cada operação. Para este efeito foi criado o módulo *PushdownAutomatonTests* que contém modelos de Autómatos de pilha criados para efeitos de teste assim como os testes em sí. Para cada teste é criado um modelo e é chamada a função a ser testada no mesmo, sendo realizado um *assert* do resultado com a bilbioteca de *assertions* do OCaml. Deste modo é assegurado que a biblioteca funciona de maneira correta durante o desenvolvimento da mesma. Também torna muito mais simples as tarefas de refatorização da mesma. Foram desenvolvidos testes para os casos mais simples de cada operação, assim como alguns casos especiais (*corner cases*).

```
1  module PushdownAutomatonTests : sig end =
2  struct
3
4  (...)
5
6  let pdaNonDeterministic = {| {
7     kind: "pushdown automaton",
8     description: "this is an example",
9     name: "pda_astar",
10    inputAlphabet: ["a"],
11    stackAlphabet: ["a", "$"],
```

```
12      states: ["S1"],
13      initialState: "S1",
14      initialStackSymbol: "$",
15      transitions: [
16          ["S1", "$", "a", "S1", "$"],
17          ["S1", "$", "~", "S1", "$"]
18      ],
19      acceptStates: ["S1"],
20      criteria: "true"
21      } |}
22
23  let pdaDeterministic = {| {
24      kind: "pushdown␣automaton",
25      description: "this␣is␣an␣example",
26      name: "pda_astar",
27      inputAlphabet: ["a", "b"],
28      stackAlphabet: ["a", "$"],
29      states: ["S1"],
30      initialState: "S1",
31      initialStackSymbol: "$",
32      transitions: [
33          ["S1", "$", "a", "S1", "$"],
34          ["S1", "$", "b", "S1", "$"],
35          ["S1", "a", "~", "S1", "$"]
36      ],
37      acceptStates: ["S1"],
38      criteria: "true"
39      } |}
40
41  let testNotDeterministic () =
42      let pda = new PushdownAutomaton.model (Arg.Text pdaNonDeterministic) in
43      let deterministic = pda#isDeterministic in
44      if deterministic then
45          Util.println ["automata␣is␣deterministic"] else Util.println ["automata␣
                is␣non-deterministic"];
46      assert (not deterministic)
47
48  let testDeterministic () =
49      let pda = new PushdownAutomaton.model (Arg.Text pdaDeterministic) in
50      let deterministic = pda#isDeterministic in
51      if deterministic then
52          Util.println ["automata␣is␣deterministic"] else Util.println ["automata␣
                is␣non-deterministic"];
53      assert (deterministic)
54
55  (...)
```

```
56
57 let pdaAccept = {| {
58     kind: "pushdown␣automaton",
59     description: "this␣is␣an␣example",
60     name: "pdaAccept",
61     inputAlphabet: ["a", "b"],
62     stackAlphabet: ["a", "$"],
63     states: ["S1", "S2"],
64     initialState: "S1",
65     initialStackSymbol: "$",
66     transitions: [
67         ["S1", "$", "a", "S1", "a$"],
68         ["S1", "a", "a", "S1", "aa"],
69         ["S1", "a", "b", "S2", "a"]
70     ],
71     acceptStates: ["S2"],
72     criteria: "true"
73     } |}
74
75 let testAccept() =
76    let pda = new PushdownAutomaton.model (Arg.Text pdaAccept) in
77    let accepted = pda#accept [symb "a"; symb "a"; symb "b"] in
78       if accepted then
79          Util.println ["Accepted␣word"] else Util.println ["Did␣not␣accept␣word"];
80          assert (accepted)
81
82 (...)
83
84 let runAll =
85     if Util.testing active "PushdownAutomaton" then begin
86         Util.header "PushdownAutomatonTests␣starting...";
87         (...)
88         testNotDeterministic ();
89         testDeterministic ();
90         (...)
91         testAccept ();
92         (...)
93     end
94 end
```

8 OFLAT - Implementação

O OFLAT é a ferramenta gráfica que faz uso da biblioteca OCAML-FLAT anteriormente descrita. A principal função desta ferramenta é criar uma interface gráfica que permita ao utilizador interagir com os modelos FLAT implementados.

8.1 Interoperabilidade

A aplicação web OFLAT está integralmente programada em OCAML e portanto as extensões realizadas neste trabalho também foram desenvolvidas em OCaml. A questão que naturalmente se levanta é como conseguir que o código OCaml funcione num contexto de um browser web. O que tornou possível esta solução foi a ferramenta js_of_ocaml, que consiste numa solução de interoperabilidade entre OCaml e JavaScript.

A ferramenta js_of_ocaml é constituída por duas componentes: um tradutor de OCAMl para Js que permite escrever código OCaml e que o mesmo seja executado num web browser e o segundo, uma biblioteca que permite manipular elementos DOM e realizar todas as principais operações comuns disponíveis quando programando num ambiente web.

Para além disso, através do Js_of_ocaml é possível usar qualquer biblioteca disponível em javascript. Este é o exemplo do cytoscape, uma biblioteca de manipulação de grafos que foi utilizada como componente principal para a implementação do OFALT com autómatos. Também foram usadas inúmeras outras bibliotecas, que permitem estender e adicionar funcionalidades na tela do cytoscape, como menus e elementos DOM.

8.1.1 Js_of_ocaml construção de objetos javascript

Já sabemos que o código OCaml que é escrito para correr no contexto de um browser acaba por ser traduzido para JavaScript. E esse código JavaScript naturalmente vai manipular objetos JavaScript, por exemplo objetos da DOM, ou objetos da biblioteca Cytoscape. Portanto, torna-se necessário disponibilizar na linguagem OCaml formas de exprimir o acesso e manipulação de objetos JavaScript.

Esta questão é resolvida através da biblioteca Js_of_ocaml que oferece um conjunto de tipos e de métodos adaptados à situação. Adicionalmente, um aspeto que ainda não foi referido, é que o Js_of_ocaml reconhece na verdade um OCaml com pequenas extensões sintáticas que também ajudam o programador a lidar com os objetos JavaScript.

Com o auxílio do js_of_ocaml é possível construir objetos javascript através do uso da notação *object%js end*. Tornando possível em OCaml definir objetos de js essenciais para a interoperabilidade com as bibliotecas usadas na implementação do OFALT. Por exemplo, o código que está imediatamente abaixo é um exemplo da criação de um objeto em JavaScript usando a notação do js_of_ocaml.

Para além disso o js_of_ocaml tem operações de conversão de tipos em OCaml para os tipos correspondentes em JavaScript. Como por exemplo, Js.string para definir uma string em js, Js.bool para definir um tipo booleano, Js.array para converter uma lista, etc.

```
1 Js.def (object%js
2      val id = Js.string menuID
3      val content = Js.string stackDisplay
4      val selector = Js.string "node"
5      val show = Js.bool false
6      val disabled = Js.bool true
7      val onClickFunction = fun element -> ()
8    end)
```

8.1.2 Js_of_ocaml bindings

O Js_of_ocaml permite chamar funções de Js usando a função *Js.Unsafe.fun_call*, aceder e modificar parâmetros através da função Js.Unsafe.get e Js.Unsafe.set respetivamente e também converter tipos js em ocaml usando a função Js.Unsafe.coerce.

O uso destas funções tem a desvantagem de não usar um sistema de tipos tipificado, aumentando desta forma, a probabilidade de introduzir bugs na ferramenta e de os detetar apenas durante a execução do programa.

Uma das grandes vantagens do OCaml é o seu sistema de tipos, posto isto, é de facto conveniente que a interoperabilidade seja realizada através de uma tipificação estática. O js_of_ocaml permite realizar isto usando *bindings*. Para isto, apenas é necessário definir através da notação do js_of_ocaml uma *class type*, desta forma é possível definir um objeto que pode ser usado em OCaml. Desta maneira é possível manipular objetos js usando a tipificação estática do OCaml. O tipo prop da biblioteca js_of_ocaml é usado para definir uma propriedade do objeto, ou seja, tipos que podem ser acedidos e manipulados, e por exemplo, para definir uma função que tem como input uma string de js e retorna unit (nada), é apenas necessário criar um método com a assinatura: *method showMenuItem: js_string t -> unit meth*.

```
1  class type data =
2    object
3      method id : js_string t prop
4      method source : js_string t prop
5      method target : js_string t prop
6      method label : js_string t prop
7      method nodeType : js_string t prop
8    end
9
10 class type contextMenus =
11   object
12    method destroy : unit -> unit meth
13    method showMenuItem: js_string t -> unit meth
14    method hideMenuItem: js_string t -> unit meth
15   end
```

8.2 Arquitectura MVC usada

A arquitetura do OFLAT é baseada no modelo MVC (Model, View, Controller) separando assim a responsabilidade por vários componentes. A componente de *Model* é implementada pela biblioteca OCAML-FLAT contendo todas as principais operações dos vários modelos da teoria FLAT.

Os componentes *View* e *Controller* são implementados pela ferramenta FLAT. Para cada modelo implementado existe um módulo OCAML que contém a lógica principal de visualização das operações, sendo responsável por todos os componentes que são mostrados no ecrã. O controlador é implementado baseado em classes organizadas de modo hierárquico tirando partida da herança e polimorfismo clássico de um paradigma orientado a objeto. Assim, quando é escolhido um modelo na interface gráfica, como por exemplo, os autómatos de pilha, é criado uma nova instância de um controlador do tipo *pdaController* e atribuindo a uma variável global. Deste modo, quando é realizada uma operação na aplicação, o controlador dos autómatos de pilha é o responsável por chamar os métodos necessários para concluir a ação 8.1.

8.2.1 Arquitetura geral para autómatos

No ponto inicial desta dissertação, os autómatos finitos já se encontravam implementados, posto isto, um dos grandes objetivos passou pela criação de uma arquitetura para os controladores e classes de gráficos dos autómatos, que evite ao máximo repetição de código e que permitisse a extensão do código existente para novos tipos de autómatos de uma maneira fácil e que necessite do menor número de alterações possíveis ao código existente.

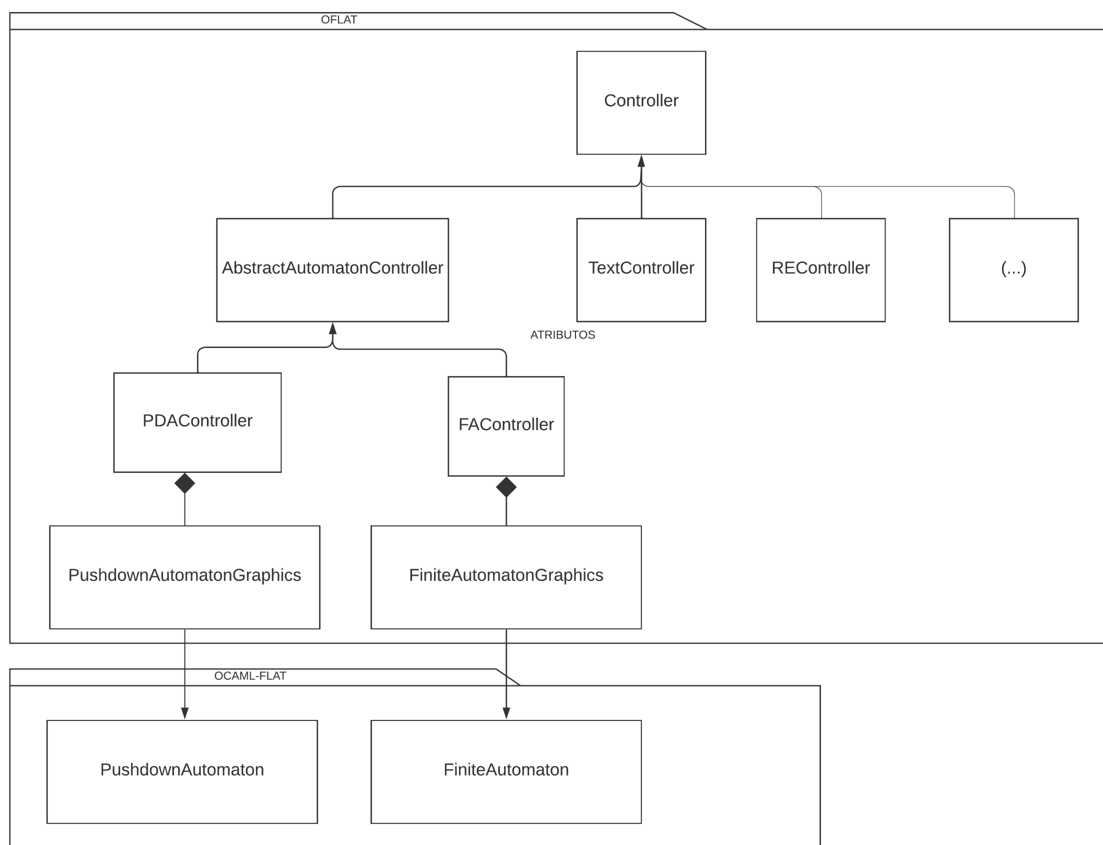

Figure 8.1: Diagrama de classes OFLAT e OCAML-FLAT *controllers* e *graphics*

Foi realizado à priori uma reestruturação na arquitetura utilizada para autómatos, garantindo assim que a extensão das funcionalidades do OFLAT para autómatos de pilha segue boas práticas mantendo uma noção coesa de autómato por toda a aplicação.

Todas as operações comuns aos vários tipos de autómatos foram colocadas nas classes AbstractAutomatonController e AbstractAutomatonGraphics. Operações para autómatos dum tipo específico, foram colocadas na respetiva classe específica, (figura 8.2). Para além disso, nas operações que têm conceitos semelhantes, mas alguns passos são realizados de maneira diferente dependendo do modelo utilizado, foram implementadas um conjunto de métodos abstratos que têm de ser implementados por cada uma das classes concretas, mantendo assim a lógica comum nas classes abstratas.

Estas alterações, por um lado reduzem o código repetido e melhoram a extensibilidade da biblioteca com novos tipos que usem autómatos como base. Por outro lado, torna o código mais complexo, havendo um esforço acrescido de modo a entender os métodos abstratos criados, sendo necessário seguir as abstrações e conceitos criados.

De seguida iremos analisar a implementação do método de inicialização da operação de accept, que permite ao utilizador observar passo a passo o comportamento do autómato a aceitar/rejeitar uma palavra.

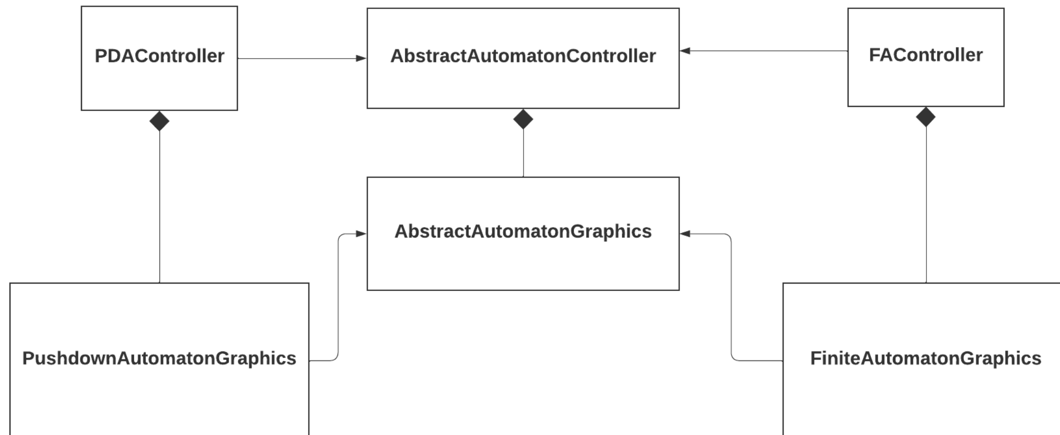

Figure 8.2: Diagrama de classes OFLAT controlador e graphics para autómatos

O método da classe AbstractAutomatonGraphics realiza um conjunto de operações que é comum a todas as classes de autómatos. Por exemplo, inicializa um vector do tipo Set que contém um set de configurações em cada passo da operação accept, define um inteiro *position* que representa a posição atual da operação accept, etc.. Estas operações são transversais a qualquer autómato, o pormenor essencial está na chamada do método setInitialStep abstrato. Este método irá ser implementado pela classe concreta de cada autómato, sendo possível realizar qualquer operação de *setup* necessária para o mesmo. Isto inclui a inicialização das funcionalidades gráficas específicas de cada autómato assim como dá liberdade para definir o modo de geração de novas configurações.

O método *startAccept* e *setInitialStep* é apenas um exemplo dos vários métodos e abstrações criados. Mais concretamente, para operação de accept foram criadas 3 operações, a primeira já referida *startAccept* que utiliza o método abstrato *setInitialStep*, o método *next* que chama o método abstrato *setNextStep* e por fim o método *back* que chama o método abstrato *setBackStep*. Desta maneira como referido anteriormente todo o código comum encontra-se nas classes abstratas e com a implementação destes 3 métodos abstratos as classes concretas de autómatos têm liberdade total para definir o modo como geram novas configurações, passo a passo, como por exemplo a implementação dos autómatos finitos que gera apenas o próximo conjunto de configurações quando é realizada a operação setNextStep ou realizar a pré computação no *setup*, que é o exemplo da implementação dos autómatos de pilha, que gera todo o histórico da aceitação de uma palavra, inicializando-a na chamada do método startAccept. Assim, com esta arquitetura existe completa liberdade para implementar extensões gráficas e operações especiais para cada classe concreta de autómatos, encapsulando os conceitos comuns dos mesmos numa única classe.

```
1 (* Metodos da classe AbstractAutomatonGraphics *)
2 method virtual setInitialStep: Cytoscape.cytoscape Js_of_ocaml.Js.t -> unit
```

```
3
4  method startAccept (cy:Cytoscape.cytoscape Js_of_ocaml.Js.t) =
5      steps <- Array.make 1000 Set.empty;
6      position <- 0;
7      isOver <- false;
8      self#setInitialStep cy;
9      self#paintCurrentStates cy true;
10     if (position = (List.length !sentence)) then
11       (isOver <- true);
12     self#changeSentence ()
13
14
15 (* Metodo da classe PushdownAutomatonGraphics *)
16 method setInitialStep cy =
17     let (configsList, bestPath) = self#getConfigsPathBySymbolConsumedAndBestPath
           (List.map char2symb !sentence) in
18     self#setConfigsAndBestPath configsList bestPath;
19     self#initAllMenusAndFeatures cy steps.(0)
```

8.3 PushdownAutomatonGraphics

A classe Model implementada no módulo PushdownAutomatonGraphics, módulo principal das operações gráficas de autómatos de pilha, estende a classe principal da biblioteca OCAML-FLAT. Desta maneira, é possível chamar os métodos da biblioteca utilizando o próprio objeto PushdownAutomatonGraphics. Posto isto, a classe Model do módulo PushdownAutomatonGraphics engloba todas as operações principais implementadas para autómatos de pilha e estende com operações gráficas do mesmo.

As operações de edição do autómato de pilha, como por exemplo, adicionar estados, adicionar transições, etc.., ou seja, todas as operações que alteram o próprio modelo estão implementadas de forma a construir uma nova instância resultante do modelo da instância atual e adicionando ou filtrando os estados ou transições pretendidas.

```
1  (metodos do classe Model do modulo PushdownAutomatonGraphics)
2
3  method addState state =
4    let rep: t = self#representation in
5      new PushdownAutomatonGraphics.model (Representation {
6        inputAlphabet = rep.inputAlphabet;
7        stackAlphabet = rep.stackAlphabet;
8        states = Set.add state rep.states;
9        initialState = rep.initialState;
10       initialStackSymbol = rep.initialStackSymbol;
11       transitions = rep.transitions;
12       acceptStates = rep.acceptStates;
```

```
13      criteria = rep.criteria
14    })
15
16  method addTransition (fromState, topStack, symbol, toState, putInStack) =
17    let rep: t = self#representation in
18    new PushdownAutomatonGraphics.model (Representation{
19      inputAlphabet = if symbol <> epsilon then Set.add symbol rep.inputAlphabet
          else rep.inputAlphabet;
20      stackAlphabet = Set.union (Set.make (topStack::putInStack))
          rep.stackAlphabet;
21      states = rep.states;
22      initialState = rep.initialState;
23      initialStackSymbol = rep.initialStackSymbol;
24      transitions = Set.add (fromState, topStack, symbol, toState, putInStack)
          rep.transitions;
25      acceptStates = rep.acceptStates;
26      criteria = rep.criteria
27    })
```

No caso do alfabeto do autómato de pilha assim como o alfabeto da pilha, estes parâmetros são extraídos das próprias transições existentes. Quando é criada uma transição, o método *addTransition* cria a nova transição e adiciona ao set de símbolos dos alfabetos os novos símbolos correspondentes.

8.4 Controlador

Todas estas operações anteriormente descritas são executadas através do controlador. Cada operação executada na interface gráfica faz com que uma cadeia de acontecimentos seja despoletada, desde por exemplo, o listener do botão carregado, percorrendo até ao controlador, que normalmente irá realizar tarefas de parsing e normalização do input do utilizador e por sua vez irá chamar o método da classe *Graphics* correspondente. Esta por sua vez irá realizar todas as alterações definidas para a operação a executar e irá novamente mostrar ao utilizador o resultado.

```
1  (*metodo do Controlador*)
2  method createTransition source target =
3    self#operationAutomaton "add␣transition";
4    let promptResult = (JS.prompt (Lang.i18nTextEnterTransitionPda ()) "a␣:␣z␣/␣
         ~") in
5    match Js.Opt.to_option promptResult with
6    | None -> ()
7    | Some v ->
8    if (PushdownAutomatonGraphics.isInputValid (Js.to_string v)) then
9      begin
```

```
10        let (topStack, inputSymb, toPutInStack) =
              PushdownAutomatonGraphics.parseUserInput (Js.to_string v) in
11        let transition = (source, topStack, inputSymb, target, toPutInStack) in
12        myPDA <- myPDA#addTransition transition;
13        Cytoscape.addEdge self#getCy
              (PushdownAutomatonGraphics.transitionPda2CytoscapeEdge transition)
14      end;
15    self#defineInformationBox
```

O OFLAt tem 2 variáveis globais principais para a gestão dos controladores. Estas variáveis correspondem ao painel principal apresentado quando é inicializado num novo modelo, seja criado a partir do zero ou carregado dos exemplos disponíveis da aplicação. E a segunda, ao painel apresentado quando é necessário ter mais que um modelo apresentado na aplicação, sendo que é possível ter 2 modelos, do mesmo tipo ou de tipos diferentes, abertos ao mesmo tempo.

```
1  (*modulo com os 2 controladores ctrL e ctrlR*)
2  module Ctrl = struct
3    let textCtrl s = new textController s
4
5    let ctrlL = ref (textCtrl false)
6    let ctrlR = ref (textCtrl true)
7
8    let changeCtrlL (nc: controller) =
9    ctrlL := nc;;
10
11   let changeCtrlR (nc: controller) =
12   ctrlR := nc;;
13
14   let _ = changeCtrlL (textCtrl false);
15        changeCtrlR (textCtrl true);;
16
17   CtrlUtil.changeToControllerCtrlRight := fun () -> (changeCtrlR (textCtrl
         true));;
18   CtrlUtil.changeToControllerCtrlLeft := fun () -> (changeCtrlL (textCtrl
         false));;
19 end
```

9

Avaliação e trabalho futuro

9.1 Avaliação

Durante a implementação dos algoritmos principais da biblioteca OCaml-FLAT, foram criados testes unitários para todas as operações, de maneira a dar mais garantias do correto funcionamento das mesmas. Houve um especial foco nos testes da operação de aceitação de uma palavra, visto que a operação é complexa e tem muita visibilidade. Também houve cuidado na criação de testes unitários de forma a testar casos potencialmente mais problemáticos, como por exemplo, autómatos de pilha que criem situações de loop alterando constantemente o conteúdo da pilha. No entanto, adicionalmente, teria sido melhor tornar a aplicação acessível a um número limitado de utilizadores, para que a mesma fosse testada de uma forma mais intensiva.

Para ajudar a avaliar a qualidade e correção do trabalho realizado, o próprio orientador desta dissertação foi muitas vezes consultado no papel de antigo docente da disciplina de Teoria da Computação.

9.2 Conclusão

Para este projeto foram estendidas duas bibliotecas, o OCaml-FLAT e o OFLAT, correspondendo à biblioteca que tem as principais operações sobre modelos FLAT e à ferramenta gráfica que permite criar e manipular modelos FLAT.

A biblioteca OCaml-FLAT foi entendida com autómatos de pilha, sendo criando todos os tipos necessários para a sua representação, assim como várias operações, desde a aceitação de uma palavra pelo o autómato, a geração de palavras de tamanho n, conversões entre os vários tipos implementados na biblioteca e conversões de autómatos. Para além destas operações foram também realizadas operações que geram árvores de procura representantes do histórico do comportamento do autómato durante a aceitação de uma palavra, que são usadas pela ferramenta gráfica para realizar a operação principal de *accept*.

A ferramenta gráfica OFLAT, suporta a visualização e manipulação de autómatos de

pilha, e foi enriquecida com a possibilidade de aceder a várias operações da biblioteca com visualização gráfica dos resultados, por exemplo operações de conversão entre modelos e operação alteração do critério de aceitação do autómato de pilha. Mas o maior destaque vai para a animação passo a passo da tentativa de reconhecimento duma palavra.

Uma melhoria secundária, foi a reestruturação e refatorização de bastante código relacionado com autómatos finitos e autómatos de pilha, tendo sido criadas duas classes abstratas novas: uma para o controlador e outra para o visualizador.

9.3 Trabalho Futuro

No que diz respeito à parte gráfica dos autómatos de pilha, existem algumas ideias de possíveis melhorias. Concretamente, tornar mais estruturada a indicação das propriedades de cada transição, tornar mais legíveis a apresentação visual dessas propriedades, e criar um algoritmo mais ambicioso para posicionar os nós do autómato no ecrã.

Também seria interessante criar a opção de visualizar autómatos no formato de tabela.

Antes da conceção da interface gráfica foram analisadas as principais ferramentas concorrentes, especialmente o JFLAP, nos seus pontos forte e fracos. Um aspeto do JFLAP que foi pouco apreciado foi a animação do reconhecimento duma palavra num autómato de pilha não determinista, pois coloca no ecrã demasiada informação, uma explosão de configurações visíveis, muito difíceis de acompanhar. Na alternativa que foi desenvolvida neste trabalho, as configurações estão escondidas nos nós do grafo, e só são visualizadas qual o utilizador clica nos nós. Isto ajuda realmente o utilizador a fazer uma análise mais profunda do comportamento do autómato.

Contudo, ainda existe margem para melhoria. Por exemplo, na operação de reconhecimento, quando existe um número muito alto de configurações para um único estado, deveria ser usado um menu hierárquico para ajudar a lidar potenciais explosões de configurações.

Bibliography

[1] P. Chakraborty, P. C. Saxena, and C. P. Katti. “Fifty years of automata simulation: a review”. In: *Inroads* 2.4 (2011), pp. 59–70. DOI: 10.1145/2038876.2038893. URL: https://doi.org/10.1145/2038876.2038893 (cit. on p. 3).

[2] C. I. Chesñevar, M. L. Cobo, and W. Yurcik. “Using theoretical computer simulators for formal languages and automata theory”. In: *ACM SIGCSE Bull.* 35.2 (2003), pp. 33–37. DOI: 10.1145/782941.782975. URL: https://doi.org/10.1145/782941.782975 (cit. on p. 3).

[3] N. Chomsky. *Context-free Grammars and Pushdown Storage*. 1962 (cit. on p. 9).

[4] N. Chomsky. “On Certain Formal Properties of Grammars”. In: *Inf. Control.* 2.2 (1959), pp. 137–167. DOI: 10.1016/S0019-9958(59)90362-6. URL: https://doi.org/10.1016/S0019-9958(59)90362-6 (cit. on p. 9).

[5] N. Chomsky and G. A. Miller. “Finite State Languages”. In: *Inf. Control.* 1.2 (1958), pp. 91–112. DOI: 10.1016/S0019-9958(58)90082-2. URL: https://doi.org/10.1016/S0019-9958(58)90082-2 (cit. on p. 9).

[6] S. Devakumar, D. Naik, and V. R. Sajja. “A TOOL FOR VISUALISATION OF PARSERS: JFLAP”. In: *Journal of Innovative Research and Solutions(JIRAS)* 1 (2014-03), pp. 1–4 (cit. on p. 4).

[7] K. Dickerson. *MS Windows NT Kernel Description*. URL: https://automatonsimulator.com/ (visited on 2022-07-08) (cit. on p. 5).

[8] R. Evey. *The Theory and Applications of Pushdown Store Machines*. Mathematical linguistic and automatic translation: Report to National Science Foundation. Harvard University, 1963 (cit. on p. 9).

[9] J. Hughes. “Why Functional Programming Matters”. In: *Comput. J.* 32.2 (1989), pp. 98–107. DOI: 10.1093/comjnl/32.2.98. URL: https://doi.org/10.1093/comjnl/32.2.98 (cit. on p. 7).

[10] Z. Khan. “REVIEW OF AUTOMATA SIMULATORS AS A TEACHING AID FOR THEORY OF COMPUTATION”. In: (2020-11) (cit. on p. 3).

[11] P. Linz. *An introduction to formal languages and automata, 4th Edition*. Jones and Bartlett Publishers, 2006. ISBN: 978-0-7637-3798-6 (cit. on pp. 9, 10).

[12] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[13] *Ocsigen Framework*. URL: http://ocsigen.org/js_of_ocaml/latest/manual/overview (cit. on p. 22).

[14] V. S. Shekhar et al. "JFLAP Extensions for Instructors and Students". In: *Sixth IEEE International Conference on Technology for Education, T4E 2014, Amritapuri, India, December 18-21, 2014*. Ed. by Kinshuk and S. Murthy. IEEE Computer Society, 2014, pp. 140–143. DOI: 10.1109/T4E.2014.22. URL: https://doi.org/10.1109/T4E.2014.22 (cit. on p. 4).

2023
Autómatos de Pilha nas ferramentas OCaml-FLAT/OFLAT
Carlos Freitas
NOVA
NOVA SCHOOL OF SCIENCE & TECHNOLOGY