NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

MIGUEL DE SOUSA LOURENÇO

Licenciado em Engenharia Informática

MÁQUINAS DE TURING EM OCAML-FLAT/OFLAT

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa

2 de Outubro, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTAMENTO DE
INFORMÁTICA

MÁQUINAS DE TURING EM OCAML-FLAT/OFLAT

MIGUEL DE SOUSA LOURENÇO

Licenciado em Engenharia Informática

Orientador: Artur Miguel Dias

Prof. Auxiliar, Universidade NOVA de Lisboa

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
2 de Outubro, 2023

Máquinas de Turing em OCaml-Flat/OFLAT

Este documento foi gerado com o processador (pdf/Xe/Lua)LaTeX e o modelo NOVAthesis (v6.9.15) [8].

Resumo

Para ajudar no ensino dos conceitos de Teoria de Linguagens Formais e Autómatos (*FLAT - Formal Languages and Automata Theory*) são desenvolvidas diversas ferramentas pedagógicas. É esta a motivação que justificou, em 2018, o início da elaboração da biblioteca OCaml-FLAT e da aplicação *web* OFLAT. Ambas as ferramentas estão escritas em OCaml, tentando usar deliberadamente um estilo funcional declarativo, que em alguns casos se consegue aproximar dos formalismos teóricos ensinados aos alunos.

Esta dissertação visou estender as referidas ferramentas com a adição de suporte para máquinas de Turing. Foram desenvolvidas as funcionalidades normais destes modelos, tanto na biblioteca, ao nível lógico, como na aplicação gráfica, através duma interface gráfica ambiciosa com preocupações pedagógicas. Alguns exemplos de funcionalidades desenvolvidas para as máquinas de Turing são: criação; edição; simulação da execução; aceitação e geração de palavras; determinação de propriedades particulares; conversões de/para outros tipos de modelos. Também foi introduzido suporte para máquinas de Turing linearmente limitadas.

Destaque para a animação gráfica da execução de máquinas de Turing possivelmente não deterministas, e para a operação de conversão direta de autómatos de pilha para máquinas de Turing com fita simples (não foram encontradas soluções concretas na literatura).

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

Palavras-chave: Máquinas de Turing, FLAT, OCaml, OCAML-Flat, OFLAT

Índice

Índice de Figuras

1

Introdução

1.1 Contexto

Em algumas das cadeiras do curso de Engenharia de Informática, é habitual estudarmos conceitos de elevada complexidade, devido à sua abstratividade e relação lógica com a matemática. A cadeira de Teoria da Computação é um exemplo onde podemos encontrar vários desses conceitos, nomeadamente no ensino de Teoria de Linguagens Formais e Autómatos(FLAT). Inevitavelmente, este tipo de matéria acaba por ser mais complicada de interiorizar por parte dos estudantes e difícil de lecionar por parte dos professores. Para mitigar este problema, podemos recorrer ao uso de ferramentas pedagógicas durante o ensino de matéria FLAT. Estas ferramentas permitem estimular a aprendizagem dos alunos ao criar um ambiente onde eles podem mexer com os conceitos dados de uma forma mais interativa e aprofundada.

Apesar de já existirem ferramentas pedagógicas vocacionadas para estes tópicos, foi decidido por alguns professores do Dep. Informática (DI) ligados ao ensino Teoria de Linguagens Formais e Autómatos(TLFA) que seria do seu interesse criar uma ferramenta nova. A criação de uma nova aplicação tem algumas vantagens, sendo uma delas o facto de podermos construir algo melhor do que já existe, ao olhar e analisar o que já foi feito para perceber o que deve ser reproduzido, o que deve ser melhorado e o que falta. Outra vantagem de criar a nossa versão da ferramenta, resulta de a tornar mais disponível, manutenível e escalável, algo que não seria possível fazer utilizando as aplicações já existentes. Isto é importante porque se a ferramenta for de facto importante para o lecionamento de TC, é igualmente importante assegurar que vai ser possível usá-la e de que ela pode evoluir se necessário.

O projeto foi construído em duas partes: a parte da biblioteca lógica OCaml-Flat e a parte da aplicação gráfica web OFLAT. Para o desenvolvimento do código, decidiu-se adotar a linguagem OCaml, por ser uma linguagem muito bem adaptada ao processamento de estruturas e conceitos matemáticos e por nos permitir seguir as definições e formalismos teóricos de forma próxima, tornando o código mais fácil de manter e de entender por

parte dos alunos. Também importante, foi o facto de ter surgido uma oportunidade de financiamento por parte de uma empresa interessada na divulgação do OCaml.

No contexto de algumas dissertações de mestrado, já cerca de meia dúzia de alunos do DI colaboraram no projeto OCaml-Flat/OFLAT. Esta dissertação é mais uma, nesta linha. O trabalho que faremos consiste em adicionar a este projeto suporte para máquinas de Turing e ilustração dos conceitos associados.

1.2 Estrutura do Documento

A estrutura do documento pode ser apresentada da seguinte forma:

- **Capítulo 1** - No primeiro capítulo, realizamos uma breve introdução ao tema da dissertação, o contexto em que surge e a sua estrutura.
- **Capítulo 2** - Para situar o leitor, é efetuada uma breve introdução aos conceitos FLAT mais importantes para o contexto desta dissertação.
- **Capítulo 3** - Neste capítulo trata-se do paradigma de programação escolhido para desenvolver a aplicação, a programação funcional.
- **Capítulo 4** - Aqui serão apresentados os resultados da investigação efetuada às ferramentas pedagógicas que têm sido desenvolvidas ao longo dos anos.
- **Capítulo 5** - Neste capítulo iremos falar sobre a biblioteca OCaml-Flat, e o trabalho feito para estender as suas funcionalidades.
- **Capítulo 6** - À semelhança do capítulo anterior, faremos uma introdução à aplicação OFLAT, e apresentaremos o trabalho feito.
- **Capítulo 7** - Este capítulo serve para avaliar o estado tanto da biblioteca como da aplicação.
- **Capítulo 8** - Finalmente, fazemos uma breve conclusão sobre o trabalho feito e aquele que deve ser feito futuramente.

2

Teoria FLAT

A teoria FLAT é um ramo da Matemática e da Informática que estuda linguagens formais, autómatos e a relação entre os dois. As linguagens são usadas como base material para estudar a teoria da computação. A definição de uma linguagem formal é normalmente feita com recurso a duas técnicas alternativas:

- Geração, sendo que nesse caso a linguagem é definida usando uma gramática formal, que representa um conjunto de regras aplicadas a símbolos do alfabeto da linguagem.
- Reconhecimento, em que a linguagem é definida usando algum tipo de autómato.

2.1 Hierarquia de Chomsky

Em 1959 o linguista Noam Chomsky criou um modelo de classificação para gramáticas formais que se estende também às linguagens geradas pelas gramáticas e aos autómatos que reconhecem essas linguagens. Nesta classificação, uma linguagem pode ser do **tipo 0, 1, 2 ou 3**[3], onde:

- O **tipo 0** corresponde às linguagens recursivamente enumeráveis, geradas por gramáticas irrestritas e reconhecidas por máquinas de Turing.
- O **tipo 1** corresponde às linguagens sensíveis a contexto, geradas por gramáticas sensíveis a contexto e reconhecidas por autómatos linearmente limitados.
- O **tipo 2** corresponde às linguagens livres de contexto, geradas por gramáticas livres de contexto e reconhecidas por autómatos de pilha.
- O **tipo 3** corresponde às linguagens regulares, geradas por expressões regulares e reconhecidas por autómatos finitos.

A diferença entre os vários tipos de linguagem reside na sofisticação da estrutura das palavras por ela suportadas. Ao nível da gramática, a diferença ocorre nas restrições

que se aplicam às regras que se podem usar. Por exemplo, numa gramática de tipo 2, a cabeça de uma regra só pode ter um elemento, que tem de ser uma variável. Quanto maior liberdade houver na escrita das regras, maior será o conjunto de linguagens que a gramática consegue gerar e maior será o poder de computação do autómato associado.

O tipo 0 é o tipo mais universal, que captura a noção de computação na sua forma mais poderosa. Os limites da computabilidade, ou seja, saber o que está ou não ao alcance do processamento automático de dados, são estudados sempre no contexto do tipo 0. No extremo oposto, o tipo 3 tem associado um poder computacional muito restrito e só sabe lidar com linguagens com estrutura muito regular.

Uma propriedade dos tipos de Chomsky é que o conjunto de linguagens geradas por uma gramática de tipo i será sempre um subconjunto daqueles gerados por gramáticas de tipo $i - 1$. Sabendo isto, podemos dizer que qualquer linguagem definida por uma gramática de tipo i, para $i > 0$, pertence à linguagem definida pela gramática de tipo 0[3]. Por outras palavras, podemos dizer que as linguagens regulares, livres de contexto e sensíveis a contexto são também recursivamente enumeráveis.

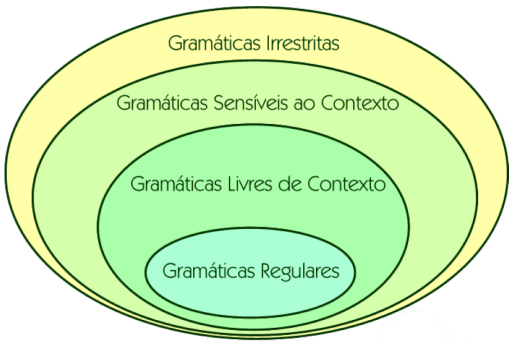


Figura 2.1: Representação visual da Hierarquia de Chomsky

2.2 Problemas de Decisão

Dado um problema de decisão P, composto por um conjunto de afirmações que podem ser verdadeiras ou falsas, queremos construir um algoritmo que determina a resposta correta para cada pergunta contida no problema. Segundo a conjetura de Church-Turing, qualquer procedimento mecânico capaz de executar um algoritmo que resolva um problema de decisão será equivalente a uma máquina de Turing. Quando interpretamos o problema P como uma linguagem L, uma máquina de Turing M é considerada capaz de resolver P se, para cada palavra da linguagem L, M para e fornece uma resposta. Nesse cenário, a linguagem L é dita recursiva e o problema P é considerado decidível.

No entanto, M também pode resolver parcialmente o problema P, o que significa que ela reconhece a linguagem L e fornece uma solução parcial para P. Isso implica que sempre que M encontra uma resposta verdadeira, ela pode responder "verdadeira", mas o mesmo pode não ser possível no caso de uma resposta falsa que, em alguns casos, M pode não

dar uma resposta porque fica eternamente a realizar transições sem nunca parar. Neste contexto, a linguagem L é dita recursivamente enumerável e o problema P é considerado semi-decidível. Eis alguns factos muito importantes sobre as características dos diversos tipos de Chomsky, que também ajudam a discernir melhor a essência das máquinas de Turing:

- Todas as linguagens dos tipos 1, 2, 3 são decidíveis (i.e. recursivas).
- Muitas linguagens de tipo 0 são semi-decidíveis (i.e. recursivamente enumeráveis).
- Mas também há linguagens de tipo 0 que são decidíveis (i.e. recursivas).

2.3 Máquinas de Turing

A máquina de Turing (MT) é um modelo matemático de computação que surgiu durante discussões teóricas sobre a automatização de resolução de problemas e é primeiro apresentada em 1936 por Alan Turing no seu artigo *On Computable Numbers, with an Application to the Entscheidungs problem*[13]. Neste artigo, Turing demonstra que, para um problema matemático X com resposta sim ou não, não é possível confirmar se existe um algoritmo Y que pare e determine o resultado.

Turing faz esta demonstração através do uso de uma máquina U que simula a operação de outra qualquer máquina M, da qual quer perceber se está livre de *loops* infinitos ou não. Para tal, é preciso provar primeiro que existe uma máquina universal U que possa simular qualquer máquina M, e que quando esta recebe uma palavra w, consegue prever se a máquina para ou não. Grande parte do artigo é dedicado à prova de que esta máquina, a qual chamou máquina automática ou *a-machine* não existe, sendo este problema chamado *Halting Problem*. No entanto, no processo de investigação, ele acabou por desenvolver a máquina que conhecemos hoje como máquina de Turing. Esta máquina é um dispositivo teórico capaz de simular a lógica de qualquer algoritmo e semi-algoritmo. Pode ser usada para definir o conceito de computabilidade, sendo este o conjunto de todos os problemas que podem ser resolvidos por um algoritmo.

Na literatura lida para o estudo deste tema, as máquinas de Turing são maioritariamente introduzidas de uma forma determinista. No entanto, nesta tese, foi decidido, por duas razões, que o objetivo seria o de criar uma máquina não determinista.

Em primeiro lugar, temos que o trabalho a desenvolver será integrado na biblioteca OCaml-FLAT, a qual introduz todos os outros mecanismos na forma não determinista. Seria confuso abrir uma exceção para as máquinas de Turing.

Em segundo lugar, resolver problemas usando uma máquina não determinista, é geralmente mais simples e lógico para os alunos, já que a insistência no determinismo força a introdução de artificialismos. Em todo o caso, as duas formalizações são equivalentes, na medida em que é possível converter automaticamente entre as duas formas.

2.3.1 Descrição informal

Uma máquina de Turing consiste numa definição formal, teórica, de uma máquina abstrata que opera sobre uma fita infinita, que está dividida em células, cada uma das quais contém um símbolo. A máquina está equipada com uma cabeça de leitura e escrita que se desloca sobre a fita, uma posição de cada vez, (conforme a figura 2.2).

O que determina o funcionamento da máquina é um conjunto de estados e de transições entre estes. Em cada passo a máquina lê o símbolo da célula corrente. Esse símbolo, conjugado com estado corrente, determina a escolha duma transição que é aplicada.

Aplicar a transição consiste em escrever um novo símbolo na célula corrente, transitar para um estado (que pode diferir ou o mesmo), e deslocar a cabeça uma posição para a esquerda ou direita.

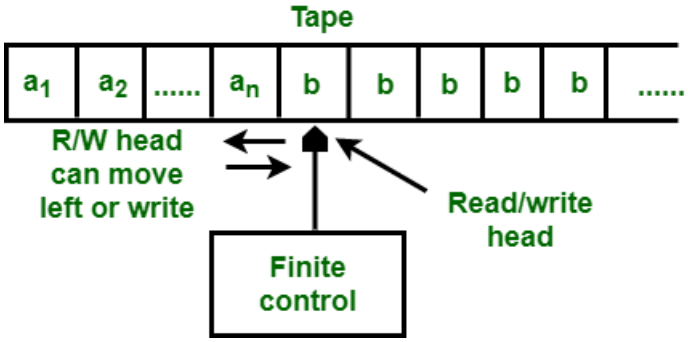


Figura 2.2: Representação dos elementos da máquina

2.3.2 Definição Formal

A máquina de Turing pode ser formalmente introduzida de maneiras diferentes, mas a notação habitualmente utilizada é a de 7-tuplos e será essa a que iremos adotar nesta tese. Uma máquina de Turing tem a forma $\boldsymbol{M = (Q; T; B; Z; qI; F; \delta)}$[7] onde:

- $\boldsymbol{Q}$: conjunto finito de estados;
- $\boldsymbol{T}$: Depende de Z, que está definido a seguir;
- $\boldsymbol{B}$: Depende de Z, que está definido a seguir;
- $\boldsymbol{Z}$: alfabeto infinito, não vazio, dos símbolos da fita;
- $\boldsymbol{qI}$: estado inicial, com $qI \subseteq Q$;
- $\boldsymbol{F}$: aceitação, com $F \subseteq Q$;
- $\boldsymbol{\delta}$: relação de transição, representada por um conjunto de tuplos sob a forma $\boldsymbol{(q, X, p, Y, D)}$, onde:
  - $\boldsymbol{q}$: estado atual da máquina, com $q \subseteq Q$;

- $\boldsymbol{X}$: símbolo lido na fita, com $X \subseteq T$;
- $\boldsymbol{p}$: próximo estado da máquina, com $p \subseteq Q$;
- $\boldsymbol{Y}$: símbolo a escrever na fita, no lugar de X, $Y \subseteq T$;
- $\boldsymbol{D}$: direção para a qual a cabeça se movimenta na fita, podendo esta ser para a esquerda ou direita, representadas por $\boldsymbol{L}$ e $\boldsymbol{R}$, respetivamente.

Diz-se que uma máquina de Turing é determinista se a relação de transição for funcional, onde $(q, X) \rightarrow (p, Y, D)$ produz um triplo (p, Y, D) para cada combinação de (q, X). No caso de produzir mais do que 1, ela é não determinista.

Adicionalmente, dependendo do tipo de máquina com que lidamos, podem ser introduzidos elementos suplementares na definição. Um exemplo disto é o caso das MTs linearmente limitadas, que veremos mais à frente.

2.3.3 Configurações

Para operacionalizar uma máquina de Turing e definir computação, precisamos da noção de *configuração*. Uma configuração caracteriza um estado particular da máquina. A evolução do funcionamento da máquina é descrito através duma sequência de configurações onde a MT transita de configuração em configuração, formando um caminho.

Uma configuração é caracterizada por três elementos: o *estado atual*; a *posição da cabeça* e o *conteúdo da fita*. Relativamente ao conteúdo da fita, cada célula contém um símbolo pertencente ao alfabeto da fita (Z), que pode ser o símbolo vazio. A notação que usaremos para estas descrições seguirá aquela apresentada por John E. Hopcroft no seu livro[3], onde:

$$X_1 X_2 \ldots X_{i-1} \boldsymbol{q} X_i X_{i+1} \ldots X_n$$

- q é o estado atual da MT
- A cabeça lê o símbolo X_i, e podemos visualizar a sua posição no estado q
- A sequência $X_1, X_2, ..., X_n$ representa toda a parte preenchida da fita, sendo geralmente colocado o estado corrente algures no meio. Pressupõe-se que o resto da fita está em branco, e também se permite que a cabeça da máquina circule pela zona em branco.

Durante o funcionamento da máquina, se quisermos concretizar uma transição δ, (q, E, p, F, L), podemos descrevê-la usando configurações da seguinte forma:

$$X_1 X_2 \ldots X_{i-1} \boldsymbol{q} E X_{i+1} \ldots X_n \quad \vdash \quad X_1 X_2 \ldots X_{i-1} \boldsymbol{p} X_i F X_{i+2} \ldots X_n$$

Como podemos ver, a máquina muda do estado q para o estado p; no lugar do símbolo E (X_i), com i sendo a posição do símbolo a ser lido, é escrito o símbolo F e a cabeça move-se uma célula na direção esquerda(L). Para demonstrar o efeito de uma transição na configuração, usamos o símbolo $\vdash$ quando executamos uma transição e $\vdash^*$ para mais do que uma transição.

2.3.4 Autómato Linearmente Limitados

Como vimos anteriormente, a máquina de Turing tem o poder de reconhecer linguagens de nível 0, e por consequência também as de nível inferior na hierarquia de Chomsky.

Uma questão importante é a de saber se existe algum tipo de autómato que reconheça exatamente as linguagens de tipo 1. A resposta é afirmativa, e na literatura descrevem-se os chamados autómatos linearmente limitados, que são máquinas de Turing com memória limitada. Tecnicamente a restrição da memória introduz-se através das seguintes três condições[3]:

- É necessário adicionar dois símbolos à fita, colocados à direita e à esquerda do input. Eles funcionam como marcadores que limitam o início e fim da fita.
- Não pode ter transições que escrevam por cima de um dos marcadores.
- Não pode ter transições que se movam para a direita do marcador da direita nem para a esquerda do marcador da esquerda.

Para transformar uma MT numa MT linearmente limitada, basta adicionar os marcadores à sua definição, e na fita, para que as condições em cima se verifiquem. No entanto, isto não quer dizer que a máquina resultante funcione como a original.

Mais adiante veremos um exemplo que pode ser transformado desta forma, e reconhece a mesma linguagem da MT definida no exemplo.

2.3.5 Exemplos

2.3.5.1 Exemplo 1

Consideremos o problema de definir uma máquina de Turing para a seguinte linguagem: $L = \{wacu : w, u \in T*\}$. Esta é a linguagem das palavras que contêm a sílaba ac. A máquina pretendida pode ser definida de diferentes maneiras, usando diferentes ideias de implementação, mas uma possibilidade é a seguinte:

- $\boldsymbol{Q} = \{q_0, q_1, q_2\}$
- $\boldsymbol{T} = \{a, b, c, d, e\}$
- $\boldsymbol{B} = \{B\}$

- $\boldsymbol{Z} = T \cup \{B\}$
- $\boldsymbol{qI} = \{q_0\}$
- $\boldsymbol{F} = \{q_2\}$
- $\boldsymbol{\delta}$, está definida na tabela da figura 2.3 e nos diagramas das figuras 2.4 e 2.5.

Apresentaremos a tabela da relação de transição apenas para este exemplo, pois ela representa a mesma informação que o diagrama e tendo em conta que a representação utilizada na aplicação OFLAT é o diagrama, a melhor solução é mostrar apenas os diagramas.

Nesta implementação optámos por uma máquina não determinista. Podíamos ter definido uma máquina determinista, mas geralmente fica mais simples inventar uma máquina não determinista. Uma máquina determinista geralmente é mais complexa em detalhes operacionais, como podemos ver na figura 2.5.

δ	a	b	c	d	e	B
q0	-	-	-	-	-	q1, B, R
q1	q1, a, R q2, a, R	q1, b, R	q1, c, R	q1, d, R	q1, e, R	-
q2	-	-	q3, c, R	-	-	-
q3	-	-	-	-	-	-

Figura 2.3: Tabela de transição do exemplo 1

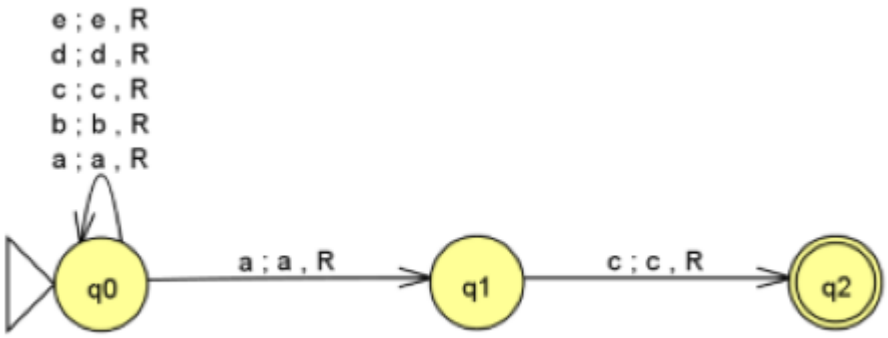


Figura 2.4: Diagrama de transição do exemplo 1

Tendo a máquina definida, vamos agora simular a sua execução, para o input $bdaecdace$. A máquina começa na configuração inicial: $Bq_0bdaecdaceB$.

Começamos pelos dois primeiros passos da simulação:

$$B\boldsymbol{q_0}bdaecdaceB \vdash Bb\boldsymbol{q_0}daecdaceB \vdash Bbd\boldsymbol{q_0}aecdaceB$$

É visível que a máquina executa duas transições que não modificam nem o estado da máquina, nem o conteúdo da fita. Isto porque o seu objetivo é apenas movimentar a

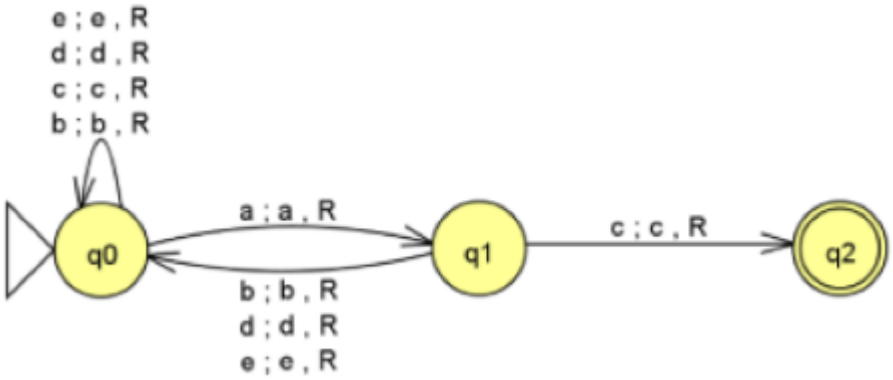


Figura 2.5: Diagrama de transição do exemplo 1 se o autómato fosse determinista

cabeça ao longo da fita, até encontrar o símbolo procurado. Classificamos este tipo de transições como transições laterais. Neste caso, estamos à procura do primeiro símbolo da sílaba ac, o a.

Quando encontramos o símbolo a, estamos perante uma bifurcação, onde temos dois caminhos diferentes que podemos seguir.

Comecemos pelo caminho onde a máquina transita para o estado $q1$. Agora que encontrámos o símbolo a, precisamos de ler a seguir o símbolo c.

$$Bbd\boldsymbol{q_0}aecdaceacB \vdash Bbda\boldsymbol{q_1}ecdaceacB$$

Como podemos ver, o próximo símbolo a ler é e e olhando para o diagrama, sabemos que não existe nenhuma transição que saia de $q1$ ao ler e. Como não existe nenhuma transição que possamos explorar, chegámos ao final do caminho, sem ter chegado a um estado de aceitação. Agora, a máquina pode continuar a executar, no caso de nenhum dos outros caminhos ter ainda chegado a um estado de aceitação, caso contrário ela para.

Como temos um caminho que ainda podemos explorar, regressamos à configuração onde nos deparámos com a bifurcação e seguimos o outro caminho que continua com o estado corrente em $q0$.

$$Bbd\boldsymbol{q_0}aecdaceBB \vdash BBbda\boldsymbol{q_0}ecdaceBB \vdash BBbdae\boldsymbol{q_0}cdaceBB \vdash$$
$$Bbdaec\boldsymbol{q_0}daceB \vdash Bbdaecd\boldsymbol{q_0}aceacB \vdash Bbdaecda\boldsymbol{q_1}ceacB \vdash Bbdaecdac\boldsymbol{q_2}eacB$$

Depois de algumas transições laterais, encontramos de novo outro a. De novo, se seguirmos o caminho que transita para o estado q_1, temos o seguinte conjunto de configurações.

$$Bbdaecd\boldsymbol{q_0}aceacB \vdash Bbdaecda\boldsymbol{q_1}ceacB \vdash Bbdaecdac\boldsymbol{q_2}eacB$$

Olhando para a última configuração, podemos confirmar que chegámos a um estado de aceitação, e por isso a máquina para. Apesar de haver outro caminho possível de percorrer, a solução à qual este poderá chegar nunca será melhor que aquela que encontrámos,

podendo apenas ser igualmente tão boa. Para avaliar se um caminho é melhor ou pior que outro em comparação, o critério usado é o do caminho mais curto, ou seja, quanto menor for o número de configurações necessárias para chegar a um estado de aceitação, melhor.

De forma geral, a execução de uma MT pode ter os desfechos seguintes:

- **aceitação** — Quando um dos caminhos chega a um estado de aceitação, a máquina termina a sua execução e, neste caso, concluímos que a palavra é aceite e pertence à linguagem reconhecida.
- **rejeição** — Se todos os caminhos param num estado que não seja de aceitação, então a máquina para de executar, e concluímos que a palavra é rejeitada e não pertence à linguagem.
- **resultado indefinido** — Se a máquina não tiver nenhum caminho que aceita a palavra e pelo menos um caminho em que não é possível determinar o seu resultado.

2.3.5.2 Exemplo 2

Neste exemplo, observaremos uma máquina que percorre a fita infinitamente e não chega a um resultado, passando toda a execução a trocar 0's por 1's e 1's por 0's na fita. Tal máquina pode ser definida da seguinte forma:

- $\boldsymbol{Q} = \{q_0, q_1, q_2\}$
- $\boldsymbol{T} = \{0, 1\}$
- $\boldsymbol{B} = \{B\}$
- $\boldsymbol{Z} = T \cup \{B\}$
- $\boldsymbol{qI} = \{q_0\}$
- $\boldsymbol{F} = \{q_2\}$
- $\boldsymbol{\delta}$, está definida no diagrama na figura 2.6.

Se executarmos a máquina, com o input *10010110*, a cabeça irá deslocar-se ao longo da fita, e trocar todos os símbolos 1 por 0 e vice-versa. Ao completar a transição que lê o último símbolo do input, o próximo símbolo a ler será o símbolo vazio, desencadeando uma mudança de estado.

$$B\boldsymbol{q_0}10010110B \vdash B0\boldsymbol{q_0}0010110B \vdash B01\boldsymbol{q_0}010110B \vdash *B01101001\boldsymbol{q_0}B \vdash B0110100\boldsymbol{q_1}1B$$

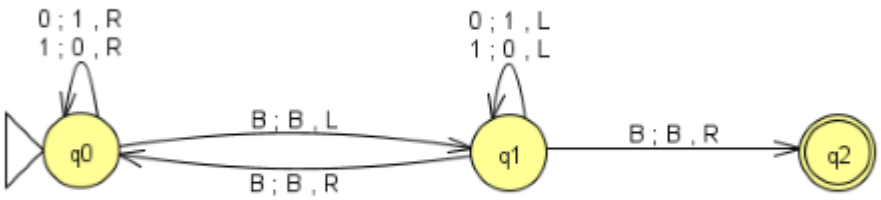


Figura 2.6: Diagrama de transição do exemplo 2

Após mudar para o estado q_1, a máquina irá desempenhar a mesma tarefa, ao trocar os símbolos, mas percorrendo a fita no sentido contrário. Quando encontrar um símbolo vazio, o estado da máquina retorna ao estado q_0, tendo assim retornado também a configuração inicial.

Teoricamente, a máquina nunca pára, independentemente do input que receber. No entanto, quando lidamos com um modelo prático da máquina, existem soluções que podemos usar para prevenir estas situações.

2.3.5.3 Exemplo 3

Este exemplo, mostra uma máquina de Turing mais complexa do que as anteriores e que utiliza técnicas mais elaboradas. Consideremos o problema de definir uma máquina de Turing que reconheça a linguagem $L = \{a^n b^n c^n | n > 0\}$[9]. Eis a nossa solução:

- $\boldsymbol{Q} = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$
- $\boldsymbol{T} = \{a, b, c\}$
- $\boldsymbol{B} = \{B\}$
- $\boldsymbol{Z} = T \cup \{X, Y, Z, B\}$
- $\boldsymbol{q_I} = \{q_0\}$
- $\boldsymbol{F} = \{q_6\}$
- $\boldsymbol{\delta}$, está definida no diagrama na figura 2.7. Trata-se duma máquina que ficou determinista, por mero acaso.

Agora que temos a máquina definida, vejamos a execução da máquina durante o reconhecimento da palavra *aaabbbccc*.

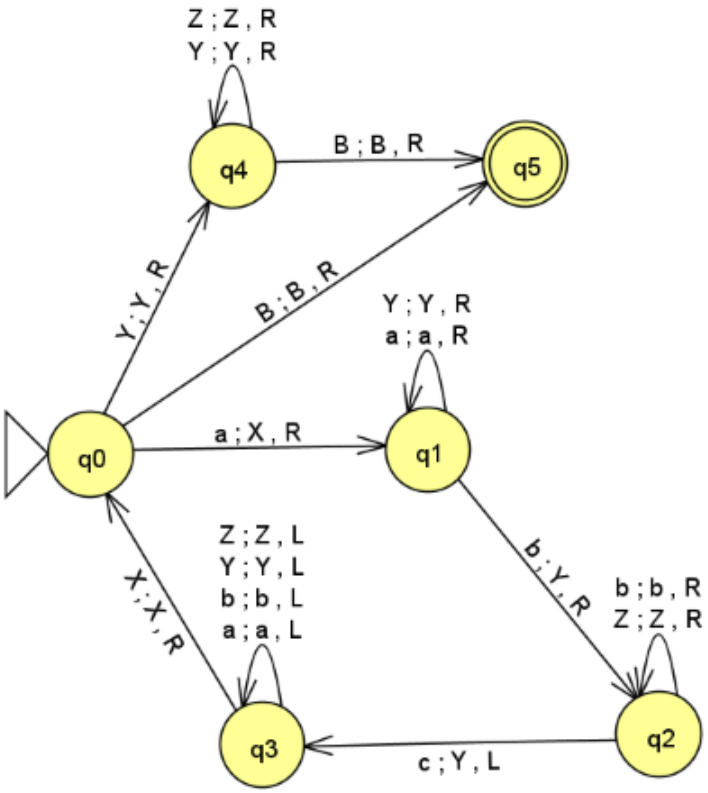


Figura 2.7: Diagrama de transição do exemplo 3

Para os estados q_1, q_2, q_3, q_4, a máquina inicia um *loop* onde transformará qualquer a em X, b em Y e c em Z. Se tudo correr bem, a máquina sai do *loop* quando não houver mais letras para transformar.

Vejamos em detalhe como a máquina executa esta tarefa.

Quando a máquina começa a executar, se a palavra for vazia para imediatamente, caso contrário ela pode percorrer um de dois caminhos diferentes. No primeiro, se o input dado for vazio, então, a máquina termina a sua execução. Caso contrário, ela lê um a na fita e, portanto, é escrito X no seu lugar; a máquina muda para o estado q_2 e a cabeça anda para a direita.

$$B\boldsymbol{q_1}aaabbbcccB \vdash BX\boldsymbol{q_2}aabbbcccB$$

Neste novo estado, a máquina segue transições literais, que movem a cabeça para a direita sempre que o símbolo encontrado for um a ou Y. Isto acontecerá até que seja encontrado um b e nesse caso ele é substituído por Y seguido de uma mudança de estado para q_3 e de um movimento para a direita.

$$BX\boldsymbol{q_2}aabbbcccB \vdash *BXaaY\boldsymbol{q_3}bbcccB$$

Similarmente ao que aconteceu no estado q_2, a máquina ignora todos os b's e Z's que encontre, até que consiga ler um c. Quando o fizer, o c é transformado num Z, o estado atual é atualizado para q_4 e a cabeça movimenta-se para esquerda.

$$BXaaY\boldsymbol{q_3}bbcccB \vdash *BXaaYb\boldsymbol{q_4}bZccB$$

A partir daqui, a máquina vai fazer o caminho de volta no input, até encontrar o último X escrito e para isso a máquina ignora todos os a's, b's, $Y's$ e Z's, movimentando-se para a esquerda sempre que encontra um destes símbolos. Quando de facto encontra um X, a máquina lê-o, volta a escrevê-lo, move-se para a direita e retorna ao estado q_1, retornando assim ao início do loop.

$$BXaaYb\boldsymbol{q_4}bZccB \vdash *B\boldsymbol{q_4}XaaYbbZccB \vdash BX\boldsymbol{q_1}aaYbbZccB$$

De volta no q_1, a máquina pode continuar o loop ou, se não existirem mais a's entre o último X e o primeiro Y escritos, a máquina vai verificar se existem apenas Y's e Z's. Quando esta situação acontece, chegámos ao fim da substituição dos símbolos do input e iniciamos o percurso até ao final do input. Para os casos de sucesso, se o número de a's for igual ao número de b's e c's, só irão existir Y's e Z's no percurso e a máquina irá chegar eventualmente ao estado q_6, aceitando assim a palavra. Caso contrário, a máquina fica presa no estado q_5 e a palavra não é aceite.

$$BXXX\boldsymbol{q_1}YYYZZZB \vdash *BXXXYYYZZZ\boldsymbol{q_5}B$$

Para dar uma ideia melhor, vejamos uma sequência de configurações que pode representar o estado da máquina ao longo da sua operação:

$$\begin{gathered}B\boldsymbol{q_1}aaabbbcccB \vdash BX\boldsymbol{q_2}aabbbcccB \vdash *BXaaq_2bbbcccB \vdash \\ BXaaY\boldsymbol{q_3}bbcccB \vdash *BXaaYbb\boldsymbol{q_3}cccB \vdash BXaaYb\boldsymbol{q_4}bZccB \vdash * \\ B\boldsymbol{q_4}XaaYbbZccB \vdash BX\boldsymbol{q_1}aaYbbZccB \vdash BXX\boldsymbol{q_2}aYbbZccB \vdash * \\ BXXXYYYZZ\boldsymbol{q_4}ZB \vdash *BXXX\boldsymbol{q_1}YYYZZZB \vdash BXXXYYYZZZB\boldsymbol{q_6}B\end{gathered}$$

Tendo em conta os resultados obtidos, podemos confirmar que a palavra $aaabbbccc$ é aceite pela máquina.

Este exemplo é importante, pois demonstra um uso que se pode dar à capacidade de escrever na fita, que não tínhamos ainda visto. O facto de a máquina poder escrever na fita, é um fator importante para o poder de computação da máquina

A partir deste exemplo é também é possível criar uma MT linearmente limitada que identifique a mesma linguagem. Como observamos neste exemplo, a escrita e leitura de símbolos na fita é feita sempre no espaço da palavra. Isto significa que mesmo que fosse imposto o limite com os marcadores, a cabeça nunca precisa de trespassá-los. No entanto,

para que a MT linearmente limitada reconheça a mesma linguagem da MT deste exemplo, seria necessário alterar as transições que leem o símbolo vazio, para invés de lerem esse símbolo, lerem o marcador direito.

3

Programação Funcional

Para a construção do OCaml-FLAT e do OFLAT, foi decidido que a linguagem de programação a usar seria o OCaml, e será com esta linguagem com que irei trabalhar nesta tese. O OCaml é uma linguagem que permite programar em diversos paradigmas, sendo por isso multiparadigma, sendo um deles o paradigma funcional. O paradigma funcional tem várias características que justificam o seu uso no contexto deste projeto.

3.1 Funções de primeira classe

Em programação, o estatuto de primeira classe é dado a entidades de uma linguagem às quais não são aplicadas restrições ao seu uso. Temos como exemplos de entidades com este estatuto as variáveis primitivas *integers*, *strings* e *booleans*. Quando dizemos que uma função é de primeira classe, praticamente, isso quer dizer que ela[2]:

- Tem um literal próprio
- Pode ser elemento de uma estrutura de dados
- Pode ser passado como argumento de funções
- Pode ser retornada como resultado de funções

Quando estamos perante uma situação onde uma função pode ser passada como argumento ou retornada como resultado, dizemos que ela é de ordem superior. O uso de funções de primeira classe permite-nos explorar mais formas de programar e torna as funções mais contidas e focadas.

3.2 Funções puras

Neste paradigma, as funções programadas são puras[2]. Uma função pura é um tipo de função com duas importantes características:

- Dado um input, uma função vai sempre retornar o mesmo *output*, fazendo com que esta seja determinista.

- Funções puras não conseguem aceder ou modificar a estados partilhados como variáveis globais, por isso não produzem efeitos secundários no programa.

O uso de funções puras melhora a qualidade do código e torna-o mais fácil de perceber, testar e manter.

3.3 Recursividade

Na programação funcional, usar funções recursivas é a única forma de criar repetição. Uma função recursiva define-se à custa dela mesma, sendo preciso cuidado para garantir que se está mesmo a definir alguma coisa (evitando as chamadas definições circulares). Normalmente espera-se que a recursão seja usada para suportar um estilo de programação declarativa, em que resolvem os problemas enfatizando a lógica da solução e esquecendo considerações de natureza operacional, como o "estado" e a ordem de execução. Contudo, de forma geral, não há nada que impeça o uso dum paradigma para simular outro paradigma e é muito frequente ver-se a recursividade ser usada para escrever código com estado simulado, código que é conceptualmente imperativo e não declarativo. Mas, se a ideia é programar usando lógica imperativa, talvez fosse melhor usar uma linguagem imperativa que suporte os conceitos imperativos de forma mais clara e direta.

3.4 Declaratividade

A programação declarativa tem como foco a criação de código que resolve os problemas focando-se puramente nas propriedades lógicas de cada problema em vez da procura ativa dum algoritmo cheio de considerações operacionais. Pode considerar-se a programação declarativa como sendo de muito alto nível por estar próxima dos métodos matemáticos e do pensamento humano, contrastando com a alternativa que consiste em programar imitando uma máquina. [12]

As funções recursivas programadas com uma atitude declarativa são sempre funções indutivas, onde se efetua a redução de problemas a versões mais pequenas dos mesmos problemas. Muitos problemas complexos ficam mais fáceis de resolver. O código fica mais legível, pois lê-se como uma equação e não precisamos de simular uma execução mental para o entender. Para a escrita das funções indutivas, ajuda muito os próprios dados serem indutivos — por exemplo, listas e árvores. Uma estrutura de dados indutiva também é definida recursivamente e cada elemento de dados não trivial, contém outro elemento do mesmo tipo, mas menor (por exemplo, uma lista não vazia tem uma cauda, que também é uma lista).

A descrição anterior aplica-se a funções individuais. Num programa grande é sempre preciso pensar em questões de conceptualização, de organização, e conseguir identificar as funções necessárias, e como essas funções devem ser compostas.

Eis algumas limitações da programação declarativa:

- Há problemas que fogem a uma lógica declarativa, ao serem na sua essência imperativos, por exemplo, o cálculo dum ponto fixo. Para calcular um ponto fixo precisamos de escrever uma função com um argumento suplementar para simular estado. Mas também é possível e mais elegante introduzir uma função genérica para cálculo de pontos fixos.

- Frequentemente o estilo declarativo conduz a programas que gastam mais recursos e são mais lentos, comparando com o estilo imperativo. Mas esse é um preço que geralmente se aceita pela possibilidade de praticar programação de muito alto nível. Se num caso particular o prejuízo for inaceitável, então será preciso rever a solução.

4

Ferramentas Pedagógicas para FLAT

Como parte do trabalho de estender as ferramentas OCaml-FLAT e OFLAT para que estas possam passar a suportar MTs, é importante saber se existem ferramentas que já o façam e se existirem, quais são as funcionalidades que oferecem. A análise do trabalho já feito dá-nos a oportunidade de, porventura, descobrir novas funcionalidades para além daquelas inicialmente planeadas e aprender com as diferentes execuções das funcionalidades que temos planeadas. Uma boa ferramenta pedagógica deverá conseguir facilitar a compreensão de conceitos difíceis, como os conceitos FLAT que lidam com matemática e abstração.

Segundo o artigo *Fifty Years of Automata Simulation: A Review*[1], o desenvolvimento de ferramentas pedagógicas para conceitos FLAT começou há mais de 60 anos e ao longo dos anos têm sido desenvolvidas várias e diversificadas ferramentas.

Em termos gerais, estas ferramentas podem ser classificadas em *online* ou *desktop* e se o autómato é gerado por texto ou diagrama. Cada um destes tem as suas vantagens e desvantagens:

- ***online***: Este tipo de ferramentas correm a aplicação num browser e estão disponíveis através da *internet*. Elas facilitam o seu uso e disponibilidade em vários dispositivos e de diferentes tipos, mas necessitam de acesso à *internet* e deixam o utilizador dependente de terceiros para que o *website* seja disponibilizado e mantido.

- ***offline***: Estas ferramentas correm diretamente no computador acedidos via ficheiro. Elas podem ser acedidas a qualquer momento, mas obrigam o utilizador a ter um computador onde este possa correr a ferramenta e o ficheiro que contém a ferramenta em si, quer seja guardado no computador ou num dispositivo de armazenamento externo.

- **texto**: As ferramentas deste tipo permitem ao utilizador editar os modelos por um editor de texto cujas regras de escrita são definidas pelo criador. Elas têm como vantagem o facto de que ocupam pouco espaço e são mais facilmente usadas em diferentes dispositivos, desde que estes tenham um teclado. Por outro lado, quando comparado com a forma seguinte, é menos intuitivo e interativo de usar.

- **diagrama**: O modelo é apresentado através de uma representação visual do modelo na qual podemos fazer alterações ao interagir com a representação em si. Esta abordagem à edição do modelo permite uma experiência mais interativa, tornando a aprendizagem mais intuitiva e cativante, o que incentiva e melhora a aprendizagem do aluno. No entanto, este tipo de ferramentas são mais difíceis de desenvolver, especialmente se for possível interagir com o diagrama, dada a complexidade envolvida na funcionalidade.

Neste capítulo, falaremos sobre 3 ferramentas que consideramos importantes, mas antes, apresentemos duas ferramentas que merecem estar aqui apenas para uma descrição rápida.

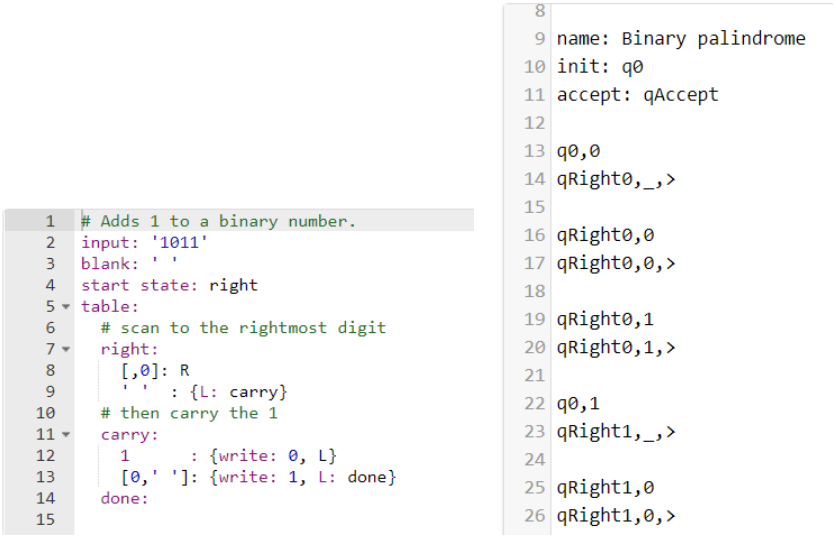
A primeira ferramenta existente que resolvemos destacar chama-se *Mr. Turing* [10], um *bot* do *Discord* que está programado para criar uma MT ao receber uma representação de texto da sua estrutura e testar se uma palavra é aceite pela máquina, tudo isto através de mensagens de *chat*. Apesar de não ser uma ferramenta pedagógica, achei interessante o facto de estarem a usar o Discord como ferramenta base, visto que este é um programa utilizado por milhões de pessoas, fazendo parte deste grupo de pessoas grande parte dos alunos de hoje em dia.

A segunda aplicação importante a mencionar é a *Turing Machine Visualization* [14] a qual é uma das ferramentas online mais visitadas. Esta aplicação gera uma MT usando texto, mas ao contrário de muitas outras ferramentas do mesmo tipo, ela apresenta graficamente um diagrama de transições, usado durante a simulação da execução da máquina. Este tipo de representação é útil por contribuir para a compreensão do conceito de como funciona uma MT.

4.1 Online Turing Machine

A *web app* que apresentaremos é um dos primeiros resultados no Google com pesquisas relacionadas a simuladores online de MTs. Criada pelo professor Martin Ugartes da Universidade PCU Chile com o aluno José Antonio Matte, esta aplicação [11] é do tipo que gera uma MT através de uma configuração em texto. A maioria dos exemplos de web apps encontradas são todas semelhantes, havendo pequenas diferenças entre cada, sendo que uma dessas diferenças costuma estar na sintaxe usada. Tomemos como exemplo os seguintes bocados de código.

Na imagem que vemos em 4.1, podemos observar duas sintaxes diferentes: a da esquerda é eficiente na quantidade de código necessária para configurar a máquina, enquanto a da direita é mais simples de escrever e perceber a lógica de como funciona. A imagem da esquerda corresponde ao código usado na Turing Machine Visualization que mencionamos anteriormente, enquanto a da direita representa o código usado na aplicação que apresentamos.



```
# Adds 1 to a binary number.
input: '1011'
blank: ' '
start state: right
table:
  # scan to the rightmost digit
  right:
    [,0]: R
    ' '  : {L: carry}
  # then carry the 1
  carry:
    1      : {write: 0, L}
    [0,' ']: {write: 1, L: done}
  done:
```

```

name: Binary palindrome
init: q0
accept: qAccept

q0,0
qRight0,_,>

qRight0,0
qRight0,0,>

qRight0,1
qRight0,1,>

q0,1
qRight1,_,>

qRight1,0
qRight1,0,>
```

Figura 4.1: Sintaxes de ferramentas diferentes

A simulação é controlada através de botões como o *play*, *stop*, *step* e *reset* sendo que ao clicar no *play*, a app vai a demonstrar passo a passo o estado da configuração. Durante a execução é possível parar, recuar para o início ou saltar um passo para a frente, e controlar a velocidade da demonstração usando um *slider*.

O controlo da velocidade é muito importante, pois permite à aplicação correr a simulação a uma velocidade que possa ser acompanhada pelo utilizador, ou corrê-la de forma tão rápida que demora apenas uns segundos.

Para auxiliar os utilizadores que não conheçam máquinas de Turing, é disponibilizado material teórico e exemplos de máquinas já construídas, para que estes possam aprender enquanto interagem com a aplicação. Uma das razões que levou à escolha desta aplicação foi que, segundo o website, esta app é muito utilizada por outras instituições.

4.2 The Interactive Turing Machine

Esta aplicação [4] é uma ferramenta online que permite criar modelos de MTs graficamente e simulá-los. Apesar de haver muitos simuladores online de MTs, o número de aplicações que permitem criar MTs via diagrama, tal como no OFLAT, aparenta ser muito reduzido, sendo esta ferramenta uma das poucas que foi possível encontrar.

A página web é composta por: um menu; um espaço para representar grafos; uma fita e um painel de controlo.

Para interagir com uma MT, podemos criá-la no espaço do gráfico; carregar um dos exemplos ou importar um modelo guardado previamente. A criação de um gráfico de raiz pode ser feita com um rato ou toque/caneta, para poder maximizar a eficiência da app para cada tipo de dispositivo.

A simulação da máquina com a fita é muito similar à da web app anterior, sendo que

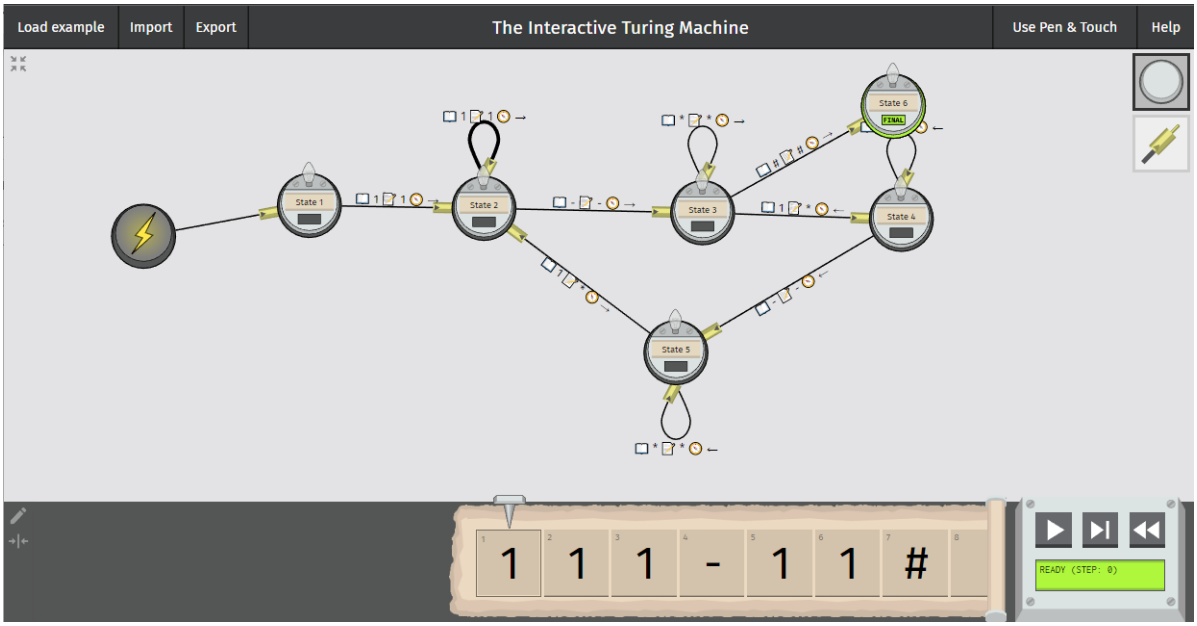


Figura 4.2: Interface da aplicação

existem duas principais diferenças entre os dois. Primeiro, a fita desta app é semi-infinita para a direita enquanto a fita do exemplo anterior é infinita para ambos os lados. Quando clicamos no play, a aplicação simula todos os passos simultaneamente e para que se consiga ver passo a passo, é necessário carregar no step sempre que quisermos dar um passo.

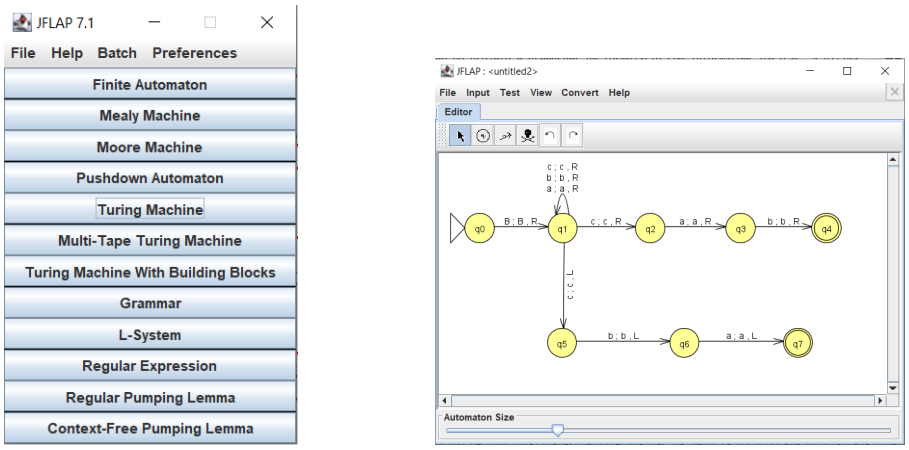
Durante uma simulação, a máquina para quando estamos perante uma de três situações: se a palavra for aceite, quando não existem mais transições a percorrer ou se a máquina tiver dado 1000 passos.

Para facilitar o uso da ferramenta, é disponibilizado um glossário com os diferentes elementos gráficos do *site*.

4.3 JFLAP

Este estudo não estaria completo se não introduzíssemos a ferramenta FLAT mais popular, sendo esta a mais reconhecida e utilizada por instituições em todo o mundo [5]. O desenvolvimento do JFLAP foi iniciado em 1990 pela professora Susan H. Rodger, quando ela sentiu que os alunos, a quem ensinava um curso relacionado com FLAT, tinham dificuldades a interiorizar a matéria.

A ferramenta foi construída para ser usada no desktop recorrendo ao uso do Java, portanto para a usar é necessário fazer o *download* da aplicação e ter instalada uma versão do Java que corra a aplicação. Para fazer o *download* da app, basta ir à página web do projeto e preencher um formulário com informações básicas. A última versão do JFLAP foi disponibilizada no verão de 2018.



(a) Ecrã principal (b) Ecrã do diagrama

Figura 4.3: Ecrãs relevantes do JFLAP

Quando iniciamos a aplicação, no menu principal, podemos observar os vários conceitos FLAT com que o programa nos permite trabalhar. Para cada conceito, o número de funcionalidades disponíveis é bastante completo.

Relativamente a MTs, temos 3 opções diferentes: *Turing Machine*; *Multi-Tape Turing Machines*; *Turing Machine With Building Blocks*. A primeira opção permite-nos criar máquinas de Turing simples com uma fita; a segunda opção permite criar MTs com mais do que uma fita e a terceira opção permite criar MT com macros, onde as macros representam MTs com objetivos simples às quais é possível ligar a outras macros para se criar MTs mais complexas.

Começando pela primeira opção, ao selecionar a mesma, abre-se uma nova janela composta por um menu, um *tooltip* com botões e uma tela, bem como um controlador deslizante. O menu dispõe de várias funcionalidades que nos permite interagir com uma MT. Neste menu encontramos numerosas operações, sendo as mais importantes:

- **Operações de simulação** — Estas operações permitem simular a operação da máquina, sendo que é possível fazê-lo passo a passo ou todos de uma só vez. É também possível testar mais do que uma palavra de cada vez. No caso de uma máquina de Turing não determinista, o simulador exibe as várias configurações alternativas que estão a ser exploradas em paralelo.

- **Operações de destaque** — Estas funcionalidades pegam no diagrama representado, e aplicam uma operação que vai analisar o diagrama e retorna como resultado um diagrama transformado, onde os estados relevantes à operação feita, têm agora uma cor diferente. Com as operações disponíveis, podemos verificar se um estado é ou não determinista e se existem transições lambda.

- **Operações gráficas** — Este grupo de funcionalidades permite guardar na memória do programa um diagrama e rearranjar o layout do gráfico no ecrã, usando algoritmos de layout aleatórios ou personalizados.
- **Operações de conversão** — Esta parte das funcionalidades possibilita ao utilizador transformar a MT representada numa gramática irrestrita ou combinar máquinas de Turing.

Na janela localizada por baixo do menu, podemos criar autómatos graficamente recorrendo ao uso de botões que permitem trocar entre selecionar elementos, criar e eliminar nós e transações e refazer e desfazer ações.

5

OCaml-Flat

Parte do trabalho desta tese consistiu em aumentar as funcionalidades do OCaml-FLAT, sendo esta uma biblioteca para teoria FLAT desenvolvida em OCaml.

Ela implementa e trabalha com conceitos FLAT, nomeadamente: autómatos finitos, expressões regulares, gramáticas livres de contexto e autómatos de pilha. Na sequência deste trabalho de dissertação, passou a suportar máquinas de Turing.

5.1 Organização

Esta biblioteca é composta por diversos módulos, sendo estes de um de dois tipos:

- **Suporte** — implementam funcionalidades diversas de apoio aos módulos principais, por exemplo, um sistema de gestão de erros, algumas estruturas de dados, reconhecedores sintáticos (para as sintaxes do JSon, expressões regulares e gramáticas), etc.
- **Principais** — contêm toda a lógica diretamente relacionada com os conceitos abstratos que queremos poder usar no programa, como os seus modelos e as operações que lhes podemos aplicar.

A organização do programa é feita em partes, onde cada uma é composta por funcionalidades com aspetos em comum. Este tipo de organização ajuda a melhorar a estrutura e legibilidade do código, facilitando mais tarde o trabalho dos contribuidores que forem fazer uma eventual manutenção ou extensão da biblioteca.

Os módulos principais, que implementam os conceitos centrais da biblioteca, estão estruturados numa hierarquia onde o módulo no topo é a **Entity**. Este módulo é herdado por todos os outros módulos e contém as propriedades mais básicas e comuns entre estes módulos como o nome e a descrição.

No nível seguinte da hierarquia, ou seja, os módulos que herdam diretamente da *Entity*, temos o **Exercise**, que permite a criação de exercícios de construção de linguagens. Para criar um exercício é necessário providenciar uma descrição textual da linguagem que

queremos que o aluno defina, bem como dois conjuntos de palavras: um com palavras aceites pela linguagem e outro em que nenhuma é aceite. Para avaliar a resposta dada, verifica-se o resultado das operações de aceitação para as palavras definidas. Um modelo é aceite como resposta correta se aceitar todas as palavras do primeiro conjuntos e nenhuma do segundo.

Para além do *Exercise*, temos também no mesmo nível hierárquico o **Model**, que agrupa as propriedades e funções comuns a todos os modelos dos conceitos FLAT disponíveis no programa.

No último nível desta hierarquia temos todos os módulos que herdam do Model, que representam os conceitos FLAT, nomeadamente o **FiniteAutomaton**, **RegularExpression**, **ContextFreeGrammar**. O trabalho desta dissertação servirá, entre outras coisas, para criar o módulo **TuringMachine**, que permite a um utilizador interagir com máquinas de Turing.

5.2 Solução e Implementação

5.2.1 Representação dos conceitos das MT em OCaml

As definições formais ligadas às MT foram cuidadosamente traduzidas para tipos da linguagem OCaml, de maneira a respeitar os conceitos originais. Existe um tipo, designado por t, que corresponde à definição sintática de MT. Os outros tipos mais importantes são relativos à execução duma MT e representam configurações da máquina ($configuration$), caminhos de execução ($path$), e sequências de patamares da árvore de pesquisa *breadth-first* ($trail$).

Relativamente às definições da teoria, tornou-se conveniente registar na definição da MT, o critério de paragem da máquina e os símbolos delimitadores para os casos em que a MT é LB.

```
1  type transition =
2      state (* Current State *)
3      * symbol (* Symbol Consumed*)
4      * state (* Next State *)
5      * symbol (* Replace Symbol *)
6      * direction (* Head Movement Direction*)
7
8  type transitions = transition set
9
10 type t = {
11     entryAlphabet : symbols; (* Entry Alphabet *)
12     tapeAlphabet : symbols; (* Tape Alphabet *)
13     empty: symbol; (* Empty Symbol *)
14     states : states; (* State Set *)
15     initialState : state; (* Initial State *)
16     transitions : transitions; (* Transition Relation *)
```

```
17      acceptStates : states; (* Accept States*)
18      criteria: bool (* true = Accept States | false = Stop *)
19      markers: symbols (* MT Linear Bounded tape markers *)
20  }
21
22  type configuration =
23      state (* State Stored in Head / Current State *)
24      * symbols (* Left Side of Tape *)
25      * symbols (* Right Side of Tape *)
26
27  type configurations = configuration set
28
29  type path = configuration list
30  type paths = path list
31
32  type direction = L | R
33  type 'c trail = 'c set list
34  type 'c path = 'c list
```

Listagem 5.1: Turing Machine types

5.2.2 Operações

Em termos das operações implementadas, o objetivo principal foi o de implementar, pelo menos, aquelas que já eram suportadas para outros autómatos, como as operações de aceitação, geração, conversões e transformações, entre outras.

Para obter código mais percetível e legível, em todas as operações implementadas é feita uma chamada para outra operação definida fora do modelo, que contém toda a lógica da operação.

5.2.2.1 Aceitação

A operação de aceitação é a mais importante desta biblioteca e uma das mais importantes da aplicação. Inicialmente foi desenvolvida uma versão da operação destinada apenas a MT deterministas. Isto foi feito com o intuito de perceber melhor como funciona a operação e aplicar esse conhecimento a uma versão da operação para MT não deterministas. Antes de começar a desenvolver a versão não determinista da operação, foi decidido criar uma versão universal, que funciona para qualquer modelo, da operação. Foi a partir desta função universal e da versão anterior, que a segunda versão de aceitação foi criada.

5.2.2.2 Aceitação Versão 1

A operação de aceitação recebe como argumentos uma palavra e a representação da MT. Antes de começar a operação, é preciso inserir nas extremidades da palavra o símbolo vazio para representar o resto da fita, preenchida por símbolos vazios.

A seguir, a operação chama uma função recursiva. Um aspeto importante que vimos anteriormente, é o de que uma MT pode implementar procedimentos semi-decidíveis. Isto implica o risco de que a máquina fique presa numa computação longa e por isso criámos dois mecanismos para prevenir esta situação.

O primeiro permite impor um limite no número de chamadas recursivas que podem ser feitas e o segundo verifica se a máquina já esteve numa configuração igual ao da configuração atual. Se algum deles se verificar, a função acaba e retorna o estado reservado (~), que representa a terminação da execução via uma destas medidas.

Contudo, no caso do primeiro mecanismo, o resultado obtido não implica que a palavra não poderia ser eventualmente aceite pela máquina, sendo esta uma desvantagem da solução. No entanto, ela consegue sempre prevenir um caso em que a máquina fica presa.

Através do segundo mecanismo, conseguimos detetar vários casos de indecidibilidade, o que nos permite aumentar o número de recursos disponíveis do primeiro mecanismo e possibilita atingir um resultado mais consistente para máquinas mais complexas.

Antes de a função retornar, é guardada a configuração corrente, na história das configurações já visitadas e exploradas. Se para a configuração corrente existir uma transição que possamos explorar, chamamos a função novamente, com uma nova configuração. Caso contrário, é retornado o estado corrente.

Tendo o último estado q da execução da máquina, é feita uma avaliação para determinar o resultado da operação:

- Se a máquina parar por estados de aceitação e q for de aceitação, a máquina aceita a palavra e retorna *true*;
- Se o estado for ~, então a máquina retorna *false*;
- Se nenhum dos casos anteriores se verificar, a máquina aceita por paragem e, portanto, tendo a máquina parado, aceita a palavra e retorna *true*;

```
1  let rec acceptX left right st trs seen limit =
2      if limit = 0 then state "~"
3      else
4          let nwLmt = limit - 1 in
5          let stateSymb = symb (state2str st) in
6          let config = (reverse left)@[stateSymb]@right in
7          if Set.belongs config seen then state "~"
8          else
9              let nwSe = Set.add config seen in
10             match left, right with
11             | [], [] -> st
12             | [], x::xs ->
```

```
13                acceptX [empty] right st trs nwSe nwLmt
14            | x::xs, [] ->
15                acceptX left [empty] st trs nwSe nwLmt
16            | x::xs, y::ys ->
17                let fFunc = fun (a,b,_,_,_) -> st = a && y = b in
18                let getTransi = Set.filter fFunc trs in
19                if Set.isEmpty getTransi then st
20                else
21                    let (_,_,c,d,e) = Set.nth getTransi 0 in
22                    if e = R then
23                        let nwL = d::left in
24                            acceptX nwL ys c trs nwSe nwLmt
25                    else
26                        let nwR = x::d::right in
27                            acceptX xs nwR c trs nwSe nwLmt
28
29 let acceptOld w rp =
30     let bWord = w@[empty] in
31     let lastST: state = acceptX [empty] bWord rp.initialState rp.
    transitions Set.empty 100 in
32         if rp.criteria then Set.exists (fun x -> x = lastST) rp.
    acceptStates
33         else if lastST = state "~" then false
34         else true
```

Listagem 5.2: Accept V1

5.2.2.3 Accept Versão 2

Esta versão do accept foi desenvolvida em grande parte usando um novo módulo testado pela primeira vez neste projeto, o Model. Este novo módulo introduz duas funções para o accept e duas para o generate bem como uma função de estatísticas. Neste subcapítulo, o que nos interessam são as funções de ***accept*** e ***acceptFull***.

Começando pela função *accept*, esta recebe também como argumentos o modelo e a palavra que queremos testar. Adicionalmente, recebe mais 3 argumentos:

- ***initial***, função que retorna a configuração inicial do modelo;
- ***next***, função que, a partir de um dado modelo e conjunto de configurações, devolve o próximo conjunto de configurações a explorar;
- ***isAccepting***, função que recebendo um modelo e configuração, determina se a configuração recebida é de aceitação, sendo que uma configuração de aceitação é aquela onde o estado corrente é de aceitação.

Para prevenir que a função fique presa num ciclo, foram implementadas as mesmas medidas que criámos para o accept anterior. Sendo que para além de se impor um limite no número de passos que podem ser dados recursivamente é também imposto um limite no tempo de execução da função. Este limite é seguido através de um cronómetro, inicializado antes da primeira chamada recursiva da função, com outras estatísticas da função. As estatísticas criadas permitem guardar o número de configurações visitadas e um booleano que avalia se a função conseguiu chegar a um resultado.

A partir da configuração inicial da máquina, retornada pelo ***initial***, a função usa a recursividade, para explorar cada uma das configurações obtidas. Esta exploração é feita usando o método do *breadth-search*, que a cada passo, utiliza o *next*, para gerar novas ondas de configurações para avaliar e explorar.

Quando a função terminar, ela retorna $true$, se a função *isAccepting* também o retornar. Caso contrário, não existem mais configurações para explorar ou foram esgotados os recursos disponíveis, e é retornado $false$.

O objetivo desta função é o de agregar e exprimir a parte da lógica do accept, presente em todos os modelos, que é feita recursivamente. Isto facilita a criação de melhores funções de accept para diferentes autómatos por várias razões. Uma delas é a de que permite simplificar a complexidade lógica da função ao separar a lógica de implementação da função, da lógica do modelo ao qual aplicamos a função. Esta modularidade conferida à função, permite-nos criar código mais simples, legível e manutenível.

Para além do *accept*, temos também o *acceptFull*. Esta função, funciona semelhantemente, apesar de ter objetivos e retornar resultados adicionais. Para além da avaliação da aceitação da palavra, a função retorna também uma árvore com todos os caminhos percorridos a cada passo e a sequência de configurações que formam o caminho mais curto. Apesar da biblioteca OCaml-FLAT ser construída em separado, idealmente, sem contexto da parte da aplicação OFLAT, a criação do *acceptFull*, surgiu da necessidade de poder representar graficamente a execução da máquina durante a operação de accept.

5.2.2.4 Operação de geração de palavras reconhecidas pela MT

Além das operações de accept, o módulo do Model também providencia duas operações de geração de palavras aceites pelo autómato: o *generate* e o *generateDumb*.

O *generate* partilha alguns argumentos com o *accept*, como o modelo (m) e as funções $initial$, $next$ e $isAccepting$, tendo para além destes, mais dois: o $\boldsymbol{l}$, que é o tamanho máximo para as palavras geradas; e o ***getWord***, sendo uma função que retira a palavra de uma configuração.

Esta função executa de forma semelhante ao *accept*, pois explora e acumula configurações, com o objetivo inicial, de gerar o conjunto daquelas que podem ser exploradas durante a execução. A este conjunto, filtramos todas as configurações que não são de aceitação e cuja palavra na fita tem tamanho superior l, e geramos um novo conjunto com as palavras presentes nas configurações obtidas, através da função map.

Para gerar palavras, podemos também utilizar o *generateDumb*, que recebe quase os mesmos argumentos, mas invés de ter o *getWord*, ele recebe o alfabeto da fita do modelo (*alphabet*).

Esta função começa por gerar todas as combinações possíveis de palavras até tamanho *l*, cujos símbolos pertencem ao alfabeto da fita, que podem ser testadas. Para verificar quais destas pertencem à linguagem reconhecida pela máquina, aplicamos a função de $accept$.

Por ser um método de força bruta, esta função pode ter de verificar um elevado número de palavras candidatas à geração, sendo que o seu número aumenta de forma exponencial com o número de símbolos pertencentes ao alfabeto da fita. Isto acaba por resultar numa função menos eficiente que a função anterior, que apesar disto, foi a escolhida para a operação de geração.

A razão pela qual utilizamos o *generateDumb* é a de que, na função *generate* e no caso de ser aplicada a uma MT, o método *getWord* não consegue obter a palavra. Para reconhecer certas linguagens, é necessário implementar estratégias que implicam escrever símbolos auxiliares, que não pertencem ao input, o que acaba por poluir a fita, e consequentemente a palavra.

5.2.2.5 Conversões entre os tipos existentes na biblioteca

Em FLAT, um tipo pode sempre ser convertido em outro, desde que o tipo para o qual queremos converter seja de um nível de menor índice na Hierarquia de Chomsky comparado com o tipo a ser convertido. Mas o contrário nem sempre é possível, ou seja, converter um tipo de nível de menor índice para um de maior índice. No caso da MT, por estar no nível 0 e de menor índice desta hierarquia, ela nem sempre pode ser convertida em qualquer outro autómato, mas qualquer outro autómato pode ser convertido numa MT.

Para a maioria dos modelos, é difícil de perceber se uma MT é equivalente a um tipo de complexidade mais baixa, e, portanto, não foram desenvolvidas operações de conversão de MT para outros modelos, exceto para AF's.

Observemos os tipos de conversões implementadas na aplicação, relacionadas com MT.

5.2.2.6 Conversão de Autómatos Finitos para Máquinas de Turing e vice-versa

Em termos práticos, a diferença entre uma MT e uma FA é a de que a MT pode escrever na fita, mover a cabeça para a esquerda e para além de aceitar por estados de aceitação, pode também aceitar por paragem.

Sabendo isto, concluímos que um AF reconhece a mesma linguagem de uma MT e vice-versa, se essa MT aceita por estados de aceitação e, para cada transição, reescrever o símbolo lido e a sua cabeça anda para a direita.

No caso da conversão de MT para AF, antes da operação ser chamada, é feita uma verificação à MT para saber se cumpre as condições necessárias.

```
let tm2fa (tm : TuringMachine.model) =
    let rep: TurMachTypes.t = tm#representation in
    let transitionsTM2FA trns = Set.map (fun (a,b,c,_,_) -> (a,b,c))
    trns in
    let tm: FinAutTypes.t = {
                    alphabet = rep.alphabet;
                    states = rep.states;
                    initialState = rep.initialState;
                    transitions = transitionsFa2Tm rep.transitions;
                    acceptStates = rep.acceptStates;
                } in
    new FiniteAutomaton.model (Arg.Representation tm)

let fa2tm (fa : FiniteAutomaton.model) =
    let rep: FinAutTypes.t = fa#representation in
    let transitionsFa2Tm trns = Set.map (fun (a,b,c) -> (a,b,c,b,R))
    trns in
    let tm: TurMachTypes.t = {
                    entryAlphabet: rep.alphabet;
                    tapeAlphabet: rep.alphabet;
                    empty: "B";
                    states: rep.states;
                    initialState: rep.initialState;
                    transitions: transitionsFa2Tm rep.transitions;
                    acceptStates: rep.acceptStates;
                    criteria: true;
                    markers: []
                } in
    new TuringMachine.model (Arg.Representation tm)

let isEquivalentFA (repTM: TurMachTypes.t) trs =
    (Set.for_all(fun (a,b,c,d,e) -> b = d && e = R) trs) && criteria =
    true
```

Listagem 5.3: Conversão de MT para AF e vice-versa

5.2.2.7 Conversão de Expressões Regulares e Gramáticas Livres de Contexto para Máquinas de Turing

Para a conversão das expressões regulares e gramáticas livres de contexto em MT, o processo é igual entre ambas e passa por primeiro convertê-las num FA e converter o resultado em MT usando a operação de conversão anterior.

```
let re2tm (re: RegularExpression.model) =
```

```
2     let reFA = re2fa re in
3         fa2tm reFA
4
5 let cfg2tm (cfg: ContextFreeGrammar.model) =
6     let cfgFA = cfg2fa re in
7         fa2tm cfgFA
```

Listagem 5.4: Conversão de RE e CFG em MT

5.2.2.8 Conversão de Autómatos de Pilha para Máquinas de Turing

A operação de conversão direta de um AP que aceita pelo critério de estados de aceitação em MT com uma fita foi algo que não foi possível encontrar na documentação estudada e, portanto, teve de ser pensada e criada de raiz. Ela apresenta diversas dificuldades, sendo a primeira conseguir representar a fita e a pilha de um AP na fita de uma MT. A solução encontrada foi a de dividir a fita com o marcador $, em duas secções, uma para cada componente.

O segundo problema é o de que para cada transição da AP, é necessário criar um mecanismo que permita à cabeça movimentar-se e escrever na fita e na pilha. Para atingir isto, foi criada uma função que para cada transição do AP, cria um conjunto de estados e transições para aceder ao topo da pilha movimentando a cabeça para a direita até ao topo da pilha, e mais um conjunto de estados e transições para aceder ao próximo símbolo a ler movimentando a cabeça para a esquerda.

Como exemplo, apliquemos, a função a uma transição de AP, que não leva a um estado de aceitação e com a forma **(p, a, α, q, β)**, onde: **p** é o estado corrente da máquina; **a** o símbolo lido na fita; $\boldsymbol{\alpha}$ o símbolo lido no topo da pilha; **q** o próximo estado da máquina; e $\boldsymbol{\beta}$ a lista de símbolos a inserir no topo da fita.

Neste exemplo, a primeira transição criada começa por substituir o símbolo a por um vazio, que serve como marcador que representa o último símbolo lido na fita e será mais tarde utilizado pela máquina. A seguir, construímos um conjunto de transições laterais, que movem a cabeça na fita para a direita até que ela chegue ao topo da pilha. Estando no topo, as transições seguintes vão escrever os símbolos em β e quando terminar, percorrerá a fita para a esquerda até ao marcador através de transições laterais que movimentam a cabeça para a esquerda. No final, é criada uma transição que ao ler o símbolo vazio, anda para a direita e o estado corrente transita para q.

Um aspeto importante a considerar, é o de que a pilha está disponível quando a AP começa a operação, e, portanto, também a representação da pilha precisa de estar operacional quando a MT executa. Para isto, é necessário que a máquina insira o marcador $ no final da palavra e o símbolo inicial da pilha na célula a seguir, sempre que começa a execução. Neste contexto, são criadas transições laterais que movimentam a cabeça até ao final da palavra, escrevem os símbolos, e voltam à posição inicial na fita.

Agora, precisamos de converter as transições de uma AP. Para isso, recorremos ao conjunto de transições que criámos anteriormente para a cabeça poder transitar entre a fita e a pilha.

No final da conversão, todos os novos estados e transições criados são agregados e inseridos na estrutura da MT criada. Dada a complexidade da emulação de uma transição de um AP numa MT, o resultado da operação leva a que o modelo gerado seja significativamente maior, quanto maior for o número de transições pertencentes ao AP.

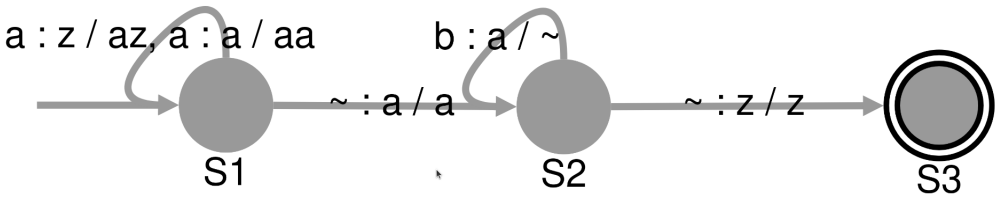


Figura 5.1: Autómato de Pilha a converter

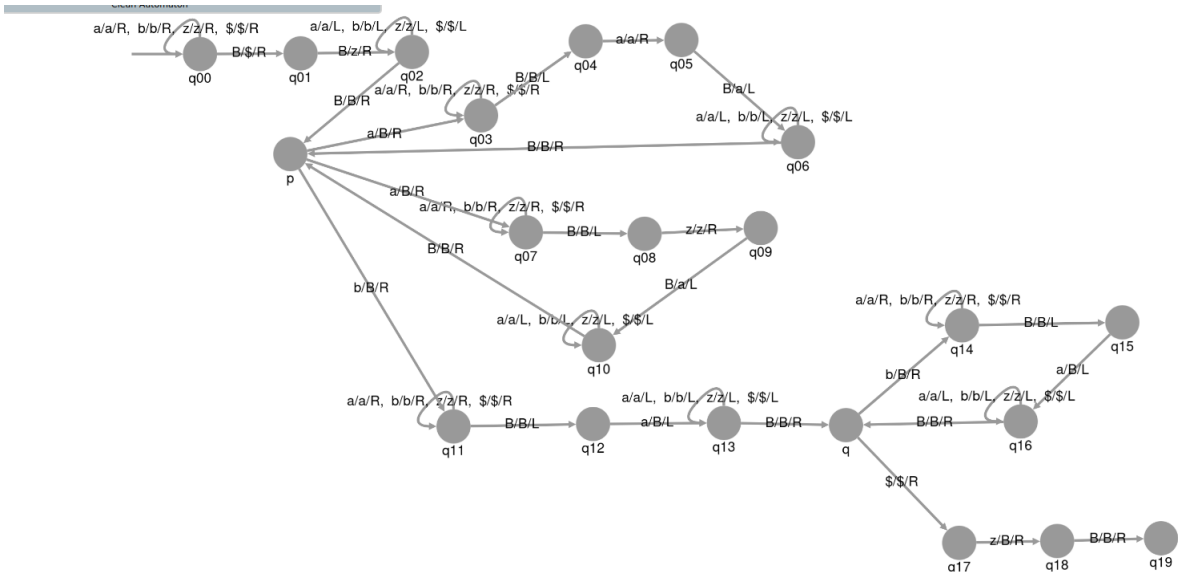


Figura 5.2: Autómato de Pilha convertido em Máquina de Turing

```
let convertNormalTransition trs alphEntr alphPD =
    let (startState,unstackedSymbol,readSymbol,nextState,writeSymbolL)
    = trs in

    let st1 = IdGenerator.gen("q") in
    let st2 = IdGenerator.gen("q") in
    let st3 = IdGenerator.gen("q") in
    let st4 = IdGenerator.gen("q") in
```

```
8
9     let ftrs = (startState,readSymbol,st1,"B",R) in
10    let trsTPD = Set.add ftrs (generateTransitionsToPD st1 alphEntr
      alphPD) in
11    let trsRTOP = Set.add (st1,"B",st2,"B",L) trsTPD in
12
13    let firstDirection = if (writeSymbolL.length == 1) then L else R
      in
14    let lastSt = if (writeSymbolL.length == 1) then st3 else st4 in
15
16    let replaceTop = Set.add (st2,unstackedSymbol,st3,writeSymbolL.hd,
      firstDirection) trsRTOP in
17    let additionalSymbolTrs = Set.union replaceTop (
      fillStackTransition lastSt st3 Set.empty writeSymbolL.tl) in
18    let trsFPD = Set.union additionalSymbolTrs (
      generateTransitionsFromPD lastSt alphEntr alphPD) in
19    let trsLast = Set.add (stLast,"B",nextState,"B",R) trsFPD in
20        Set.add lastSt (Set.add st3 (Set.add st2 (Set.add st1 Set.
      empty))), trsLast
21
22 let convertAcceptTransition trs alphEntr alphPD initialStackSymb =
23    let (startState,unstackedSymbol,readSymbol,nextState,writeSymbolL)
      = trs in
24
25    let st1 = IdGenerator.gen("q") in
26    let st2 = IdGenerator.gen("q") in
27    let st3 = IdGenerator.gen("q") in
28
29    let ftrs = Set.union (startState,dollar,st1,dollar,R) Set.empty in
30    let checkInitSS = Set.union (st1,initialStackSymb,st2,"B",R) ftrs
      in
31    let lastCheck = Set.union (st2,"B",st3,"B",R) checkInitSS in
32        Set.add st3 (Set.add st2 (Set.add st1 Set.empty)), lastCheck
```

Listagem 5.5: Conversão de AP em MT

5.2.2.9 Operações de categorização de estados

As operações de categorização desenvolvidas permitem identificar, para o modelo de uma dada MT, quais os estados que são alcançáveis, produtivos, úteis e inúteis. Sendo que um estado pode ser:

- **Alcançável**, se, a partir do estado recebido como argumento, for possível chegar ao estado em questão, via transições.

- **Produtivo**, se for um estado, que a partir do qual, é possível alcançar um estado de aceitação.
- **Útil**, se for um estado simultaneamente alcançável e produtivo.
- **Inútil**, se não for um estado alcançável ou produtivo.

Na implementação das operações, foi notado que a operação de *reachable* para MT e AF's utilizam a mesma lógica, e por isso foi decidido.

Para utilizar da função *reachable*, precisamos de primeiro, criar um AF a partir da MT, mesmo que as linguagens reconhecidas não sejam equivalentes, pois esta conversão é apenas usada para executar a função.

```
1  method downgradeModelToFiniteAutomaton: FiniteAutomaton.model =
2      let alphaB = Set.union (Set.make [empty]) representation.alphabet
       in
3      let fa: FinAutTypes.t = {
4              alphabet = alphaB;
5              states = representation.states;
6              initialState = representation.initialState;
7              transitions = transitionsTm2Fa representation.transitions
       ;
8              acceptStates = representation.acceptStates
9          } in
10     new FiniteAutomaton.model (Arg.Representation fa)
11
12 method reachable (s:state): states =
13     let fa = self#downgradeModelToFiniteAutomaton in
14         fa#reachable s
```

Listagem 5.6: Função de criacao de um AF a partir de uma MT

Para as restantes operações, inicialmente, a intenção era também de usar as operações utilizadas para AF's. Mas após testá-las, apercebemo-nos de que no caso de uma MT aceitar por paragem, a operação que deteta estados produtivos não funcionava da forma esperada. No caso de aceitar por paragem, a função deve retornar todos os estados da máquina, sendo neste caso cada um produtivo. Como a função de estados úteis está dependente da função de estados produtivos e a função de estados inúteis depende desta, também elas foram alteradas.

```
1
2  method productive : states =
3      if representation.criteria then
4          let fa = self#downgradeModelToFiniteAutomaton in
```

```
5             fa#productive
6     else representation.states
7
8 method getUsefulStates: states =
9     Set.inter self#productive (self#reachable representation.
    initialState)
10
11 method getUselessStates: states =
12     Set.diff representation.states self#getUsefulStates
```

Listagem 5.7: Restantes funções de categorização

5.2.2.10 Operação de verificação de existência de determinismo

Como vimos anteriormente, para detetar se uma máquina é determinista, basta ver se a relação de transição for uma função que produz apenas um triplo (p, Y, D) para cada combinação de (q, X). Dadas as semelhanças entre as transições de AF e MT, também é possível avaliar o determinismo da MT ao criar um AF e executar a sua função de verificação.

```
1 method isDeterministic: bool =
2     let fa = self#downgradeModelToFiniteAutomaton in
3         fa#isDeterministic
```

Listagem 5.8: Operação de verificação de existência de determinismo

5.2.2.11 Limpeza dos estados inúteis de uma MT

Para criar uma MT sem estados inúteis, precisamos de filtrar do conjunto de estados e de estados de aceitação, aqueles que forem inúteis e do conjunto de transições, aquelas que usam os estados que acabámos de filtrar. Finalmente, e apesar de não ser crítico ao funcionamento da aplicação, procedemos também à criação de um novo alfabeto que contém todos os símbolos usados nas transições que restaram.

```
1 let transitionGetSymbs trns = Set.union (Set.map (fun (_,b,_,_,_) -> b
    ) trns) (Set.map (fun (_,_,_,d,_) -> d) trns)
2
3 method cleanUselessStates: model =
4     let usfSts = self#getUsefulStates in
5     let usfTrs = Set.filter (fun (a,_,c,_,_) -> Set.belongs a usfSts
    && Set.belongs c usfSts) representation.transitions in
6     let alf = transitionGetSymbs usfTrs in
7     let newAccSts = Set.inter representation.acceptStates usfSts in
8
```

```
9     let tm = {
10        alphabet = alf;
11        states = usfSts;
12        initialState = representation.initialState;
13        transitions = usfTrs;
14        acceptStates = newAccSts;
15        criteria = true
16    } in
17        new model (Arg.Representation tm)
```

Listagem 5.9: Função de limpeza dos estados inúteis

5.2.2.12 Operações para MT Linearmente Limitadas

A implementação de uma MT linearmente limitada é feita, com base numa MT normal, sendo necessário adicionar ao registo *markers* da representação *t* os marcadores esquerdo e direito. Evidentemente o utilizador tem a responsabilidade de criar um autómato linearmente limitado correto em que não exista nenhuma transição que conduza a cabeça a sair dos limites estabelecidos e não altere os marcadores estabelecidos.

5.2.2.13 Operação *isLB*

Para verificar se uma MT é linearmente limitada, basta verificar as condições anteriores.

```
1 let isLB rep =
2     let hasMarkers = (Set.size rep.markers) = 2 in
3     let boundedDirection dir rev =
4         let trns = Set.filter (fun (_,b,_,_,_) -> b = dir) rep.
    transitions in
5         let boundedTrns = Set.filter (fun (_,b,_,d,e) -> b = dir && d
    = dir && e = rev) rep.transitions in
6         Set.equals trns boundedTrns in
7
8     let bounded = (boundedDirection leftMarker R) && (boundedDirection
     rightMarker L) in
9       hasMarkers && bounded
```

Listagem 5.10: Função de accept para uma MT linear bounded

5.2.2.14 Operação de *acceptLB*

Para a função de accept basta colocar nas extremidades da palavra os marcadores, e chamar a função de *accept*.

Na nossa implementação deste modelo, os marcadores são colocados nas células adjacentes à palavra. Isto implica que o utilizador tenha de criar um estado inicial e uma

transição que vá deste novo estado para o anterior, com o objetivo de mover a cabeça entre o marcador esquerdo e o início da palavra.

```
1 let newWord = [symb "LEFT"]@w@[symb "RIGHT"] in
2     self#accept newWord
```

Listagem 5.11: Função de accept para uma MT linear bounded

6

OFLAT

O OFLAT é uma ferramenta pedagógica que permite criar e interagir com gráficos interativos que representam modelos de conceitos *FLAT*, sendo estes suportados pela biblioteca OCaml-FLAT.

Esta aplicação corre na *web*, com recurso ao js_of_ocaml[6], que traduz o bytecode de OCaml para código *JavaScript*. Além disso, para representar os modelos, usa a biblioteca Cytoscape.js, sendo uma ferramenta de teoria de grafos que facilita a análise e a visualização de grafos e árvores, tornando a representação destes de maneira simples e clara.

Ainda hoje a aplicação continua a evoluir e esta dissertação é mais um passo nesse sentido, ao permitir a um utilizador da aplicação trabalhar com máquinas de Turing.

Ao longo do capítulo iremos: observar o estado atual da aplicação, visual e funcionalmente; falar um pouco sobre a estrutura sobre a qual foi construído o código; como funciona a conversão de código OCaml para JS e as funcionalidades desenvolvidas, bem como a sua implementação.

6.1 Descrição da Interface

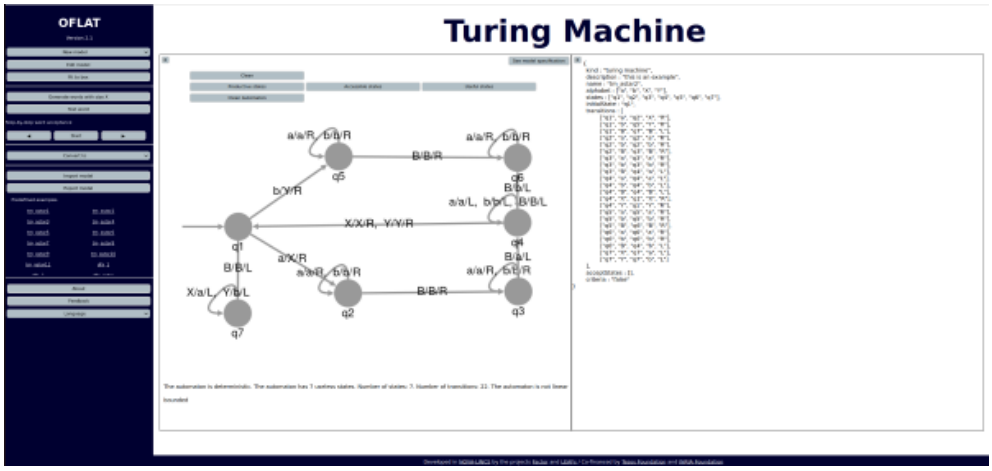


Figura 6.1: OFLAT application in use

Quando a aplicação é aberta, a sua interface é composta por 1 menu principal e uma zona em branco, que dependendo da situação, pode conter dois painéis, o principal e o de resultados. No menu podemos executar diversas operações que permitem interagir com o modelo, representado no painel principal onde pode ser editado e transformado. Já o painel de resultados, tem como propósito apresentar o resultado de algumas das operações aplicadas à representação.

O menu principal foi arquitetado para que todas as operações disponíveis estejam agrupadas em cinco secções:

- Na secção 1 temos as de criação e edição do modelo.

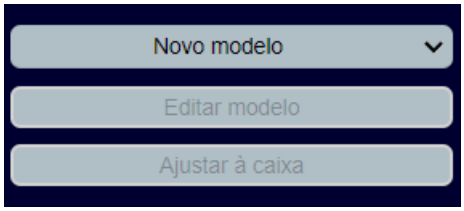


Figura 6.2: Secção de operações de criação e edição do modelo

- Na secção 2, temos aquelas relacionadas com palavras aceites pelo modelo, como a geração de palavras e o teste de palavras.

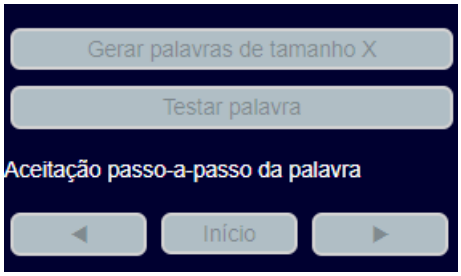


Figura 6.3: Secção de operações de palavras aceites pelo modelo

- Na secção 3, temos as de conversão onde é possível converter os diferentes modelos disponíveis na aplicação uns nos outros.

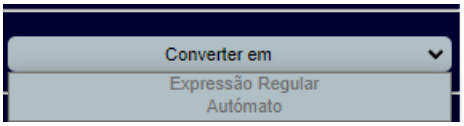


Figura 6.4: Secção de operações de conversão

- Na secção 4, temos as operações relativas a exemplos de modelos. Usando ficheiros JSON, podemos exportar o modelo representado na tela e importar um que tenhamos

guardado nesse formato. É também possível carregar exemplos disponibilizados pela aplicação, pensados para dar a conhecer ao utilizador, exemplos relevantes para compreender melhor (ter mais experiência/conhecimento sobre) os conceitos FLAT trabalhados na aplicação.

Figura 6.5: Secção de operações relativas a exemplos de modelos

- Na secção 5, temos as operações de suporte e informação da ferramenta. Aqui, é possível alterar o idioma da página, saber mais sobre as pessoas por trás do projeto, o formato dos dados do ficheiro JSON que podemos usar para importar e exportar modelos e os contactos caso o utilizador tenha comentários ou perguntas para fazer.

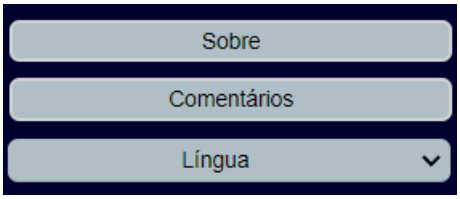


Figura 6.6: Secção de operações de suporte e informação da ferramenta

Quando um modelo é escolhido, independentemente do tipo, é aberto um novo painel principal que contém na tela a representação do modelo, e dois botões. Do lado esquerdo temos um botão com uma cruz que fecha o painel, e do lado direito um botão que abre, a representação interna do modelo em formato JSON.

O espaço em branco é dividido entre os dois painéis e quando o painel de resultados é aberto, ele ocupa exatamente metade do espaço disponível. Neste painel são exibidos os resultados de operações como a transformação de um modelo e a representação interna do modelo na tela. Uma funcionalidade interessante, é a de que quando o utilizador clica no botão com uma cruz do painel principal, se o painel de resultados estiver aberto e estiver nele representado um modelo, ele passa a ser representado no painel principal, e o painel de resultados fecha.

6.2 Arquitetura

A estrutura de organização da aplicação segue o padrão do Model-View-Controller (MVC), sendo este um padrão de desenvolvimento de *software* recomendado para a criação de *web apps*. Neste tipo de padrão, o código é estruturado em 3 componentes, onde cada um desempenha uma responsabilidade diferente, o que contribuiu para ter uma aplicação mais robusta e manutenível:

- O componente *Model* agrega o código que representa os dados das estruturas principais com que trabalharemos na aplicação, como os modelos de conceitos FLAT e as operações que lhes podemos aplicar. Ao contrário dos dois outros componentes, os módulos que compõe o Model são definidos exclusivamente na biblioteca OCaml-Flat, o que confere ao sistema ainda maior modularidade.

- O componente *View* é o componente que contém a lógica de tudo o que é gráfico. Neste componente estão incluídos os módulos que criam e definem o comportamento, através do js_of_ocaml, dos elementos HTML que vemos na aplicação.

- O componente do *Controller* funciona como uma interface entre o *Model* e o *View*, que recebe e trata *input*, recebido sob a forma de eventos, e decide como irão afetar o estado da aplicação.

6.2.1 Interoperabilidade

Para fazer a conversão do código OCaml para JS, usamos a ferramenta js_of_ocaml[6]. O js_of_ocaml é um sistema de software, criado na Universidade de Paris Diderot, que traduz bytecode OCaml para JavaScript, permitindo assim a interoperabilidade entre as duas linguagens. Este código JavaScript pode correr no contexto de um *browser web*, algo que não seria possível com o código OCaml original.

O compilador js_of_ocaml integra uma extensão ppx à sintaxe OCaml habitual, para tornar mais confortável a manipulação de entidades JavaScript do lado do OCaml. Por exemplo, para criar do lado do OCaml um objeto JavaScript, o ppx permite que se use a seguinte forma sintática: *new%js constructor args*.

6.2.1.1 Biblioteca associada

O js_of_ocaml tem uma grande biblioteca associada que define um conjunto de tipos e de operações de suporte à interoperabilidade.

Para representar um objeto de JavaScript usa-se o tipo genérico α *Js.t*. Por exemplo: o tipo dos *arrays* JavaScript escreve-se *Js.Js_array Js.t*.

Como sabemos, os objetos JavaScript contêm propriedades e métodos. As propriedades são representadas através do tipo paramétrico *Js.prop* e os métodos são representados através do tipo paramétrico *Js.meth*.

Os objetos são representados por tipos cuja forma se exemplifica: *<number: int Js.prop; increment: unit Js.meth> Js.t*

Usando estes tipos mais um conjunto de *bindings*, a biblioteca oferece ainda acesso a toda a API DOM do JavaScript.

6.2.1.2 Bindings

A biblioteca do Js_of_ocaml oferece um módulo chamado *Unsafe* que permite aceder aos dados e funções do JavaScript com toda a liberdade, mas sem qualquer validação de tipos. Felizmente que o Js_of_ocaml também oferece a possibilidade de criar *bindings* de acesso ao JavaScript com tipificação segura. Geralmente, os *bindings* são um mecanismo crucial à interoperabilidade entre duas linguagens, ao permitirem, por exemplo, o uso de uma biblioteca específica a uma linguagem na outra.

A criação de um binding para um tipo-objeto de JavaScript é realizada através da simples declaração de um *class type*. Os *class types* em OCaml normalmente representam tipos de objetos OCaml, mas o Js_of_ocaml permite interpretar esses tipos como descrevendo a estrutura de objetos JavaScript. Evidentemente, os tipos dos elementos constituintes do tipo usam os tipos do módulo Js que foram referidos na secção anterior. O seguinte exemplo define um binding:

```
1    class type edge =
2     object
3         method firstNode : int Js.prop
4         method lastNode : int Js.prop
5         method weight : int Js.prop
6         method updateWeight : int -> unit Js.meth
7    end
```

Listagem 6.1: Turing Machine types

Se tivermos um objeto JavaScript *e* do tipo *edge*, podemos-lhe aplicar facilmente os métodos previstos usando a sintaxe que se exemplifica: *e##updateWeight 5*.

A biblioteca do Js_of_ocaml já traz uma grande coleção de bindings predefinidos, incluindo para todo o HTML DOM. No nosso projeto temos necessidade de usar a biblioteca JavaScript Cytoscape.js e naturalmente criámos um razoavelmente grande conjunto de bindings para aceder com a máxima conveniência às funcionalidades que ela oferece.

6.3 Funcionalidades desenvolvidas

Tendo sido já apresentadas a interface e a estrutura da aplicação, vamos agora analisar as funcionalidades implementadas para MT na aplicação (na mais recente edição do OFLAT).

6.3.1 Criação e Edição de MT

Para criar uma MT na aplicação existem 3 formas diferentes: ao clicar em *New Model*, podemos criar um modelo de raiz a partir do modelo mais básico que contém apenas o estado inicial; podemos clicar em *Import*, para importar um modelo que já esteja definido ou podemos carregar um dos modelos disponíveis no menu da esquerda.

Na aplicação, a representação de um modelo é feita com recurso a um diagrama de estados, composto por nós interligados por arcos, sendo que estes representam estados e transições, respetivamente. No caso do diagrama que representa uma MT, os nós representam todos os estados pertencentes a Q e as condições dos arcos são legendadas por X/YD para transições com a forma (q, X, p, Y, D).

Dado existirem diferentes tipos de estados numa MT, precisamos de conseguir representá-los distintamente. Os estados de aceitação são representados por um duplo círculo, enquanto um estado inicial é identificado com uma seta direcionada para ele.

Para representar o modelo, tínhamos a opção de recorrer a uma tabela ou a um diagrama de estados. A razão pela qual escolhemos o diagrama, foi a de que esta representação é a mais intuitiva de usar e interagir. Ela permite separar os estados da máquina, das condições e consequências de cada transição, o que permite mais facilmente visualizar os diferentes caminhos que a máquina pode percorrer durante a sua execução. Esta característica é útil tanto na análise da MT e da sua relação de transição como na visualização da execução da máquina.

Tendo o modelo representado no painel, a sua edição pode ser feita diretamente na representação gráfica com recurso a menus e pop-ups. Estes menus aparecem quando o utilizador clica no botão direito do rato enquanto o cursor estiver sobre o painel principal e dependendo do elemento do diagrama em que este estiver pousado, é aberto um menu diferente.

Se o utilizador clica num estado, é aberto o menu que podemos observar na imagem 6.7, que dispõem de opções para alterar diferentes atributos do estado (tais como o nome, se é ou não é final, e se é inicial), removê-lo do autómato e adicionar novas transições que comecem nele.

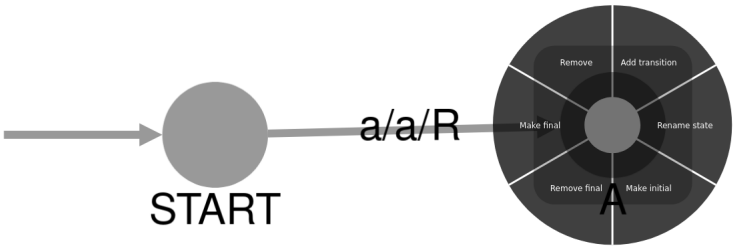


Figura 6.7: Menu quando clica num estado

Caso o utilizador clique numa transição, como podemos visualizar em 6.8, surge um menu com uma única operação que permite remover a transição.

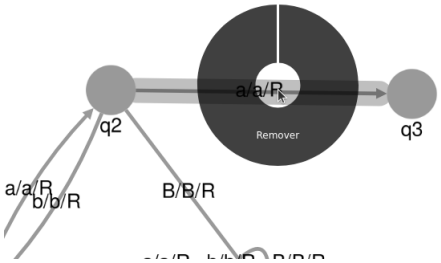


Figura 6.8: Menu quando clica numa transição

O terceiro caso é talvez menos intuitivo, mas consiste em clicar no espaço em branco que envolve o autómato. Neste caso, surge o menu que vemos na imagem 6.9, o qual permite criar um estado, existindo as opções de ser normal, final ou inicial.

Figura 6.9: Menu quando clica no espaço branco

6.3.2 Accept

No menu principal, encontram-se as operações gerais aplicáveis a todos os modelos implementados na ferramenta em questão. Uma delas é a de *accept* e existem no menu, dois botões que desencadeiam modos de operação diferentes, apesar de em ambos, ser mostrada a execução da máquina passo-a-passo. No botão de *Test word*, a operação corre automaticamente, enquanto no botão de *Start*, ela corre manualmente. Para esta última, o controlo dos passos dados é feito por botões com setas, existindo um para dar um passo em frente, e outro para trás.

Mostrar a execução da máquina passo-a-passo é um mecanismo útil, ao permitir ao utilizador observar o funcionamento da máquina de uma forma interativa, progressiva, e

no modo de *Start*, controlada. Isto proporciona uma forma bastante intuitiva de compreender o funcionamento da máquina.

Quando um dos modos é selecionado, é pedida, por um pop-up, a palavra que queremos avaliar e quando é inserida, a tela passa de modo de edição para modo de accept. Se o utilizador tentar alterar o diagrama durante a execução da operação, o modo é imediatamente revertido para o de edição. Isto impede o utilizador de fazer alterações ao autómato enquanto o accept decorre.

Para além de mudar de modo, são adicionados novos elementos ao painel. Por cima do diagrama passa a estar representada a configuração corrente num qualquer passo da execução, obtida através do caminho mais curto, devolvido pela função *acceptFull*. No entanto, dado que vemos no diagrama o estado corrente, não é necessário que ele seja representado na configuração, sendo por isso substituído por '|'.

Outros elementos são um conjunto de listas de configurações e *poppers*, adicionados a todos os estados do diagrama. Cada estado passa ter no canto inferior esquerdo um número, que corresponde ao número de configurações, onde a máquina está naquele estado e quando um utilizador põe o rato por cima do estado, aparece a lista de configurações.

Estes dois últimos foram desenvolvidos no contexto de uma tese anterior onde o aluno se deparou com um problema na representação da operação para autómatos não deterministas. Dada a noção anterior de autómato não determinista, concluímos que durante a aceitação, este tipo de autómato pode percorrer vários caminhos, levando a que seja necessário representar todas as configurações possíveis. A solução encontrada é bastante intuitiva, pois permite ao aluno visualizar quantas configurações estão num dado estado diretamente na representação e para saber quais são, basta mover o rato.

A única ferramenta estudada que também implementa a visualização das configurações a serem percorridas é aplicação JFLAP. Nela, as configurações são mostradas numa janela em baixo do diagrama com todos os caminhos que estão a ser percorridos. Apesar de ser possível redimensionar esta janela, tornando-a mais pequena quando queremos focar no autómato e maior quando queremos analisar os caminhos, existe o problema de que se houver vários caminhos a serem percorridos, torna-se difícil a leitura da *interface* de maneira a ter uma visualização intuitiva do comportamento do autómato.

Em contraste, a solução encontrada, tem a vantagem de manter bastante coesa a informação, sendo mais intuitivo mostrar as configurações em listas, separadas do grafo.

Ao longo da operação, o estado corrente é destacado a azul-escuro, se pertencer à configuração do melhor caminho, caso contrário está a azul.

Quando ela termina, se a palavra for aceite, ou seja, existe pelo menos um caminho que aceita, o estado corrente do melhor caminho fica verde, e os estados correntes dos restantes caminhos ficam a amarelo. Esta cor significa que qualquer um deles ainda tem a possibilidade de chegar a um estado de aceitação, não sendo explorados porque já foi encontrada a resposta. Por outro lado, se terminar por não aceitação, os estados correntes

ficam a vermelho e se for por falta de recursos ou deteção de configurações repetidas, ficam a castanho.

6.3.3 Generate

Esta operação consiste em gerar todas as palavras que têm comprimento maior ou igual a n que são aceites pelo modelo. No menu principal da esquerda, temos o botão *Generate words with size* 6.3 e quando clicado, este abre um pop-up onde o utilizador insere o comprimento que quer para n. O resultado aparece no painel de resultados, como vemos em 6.10.

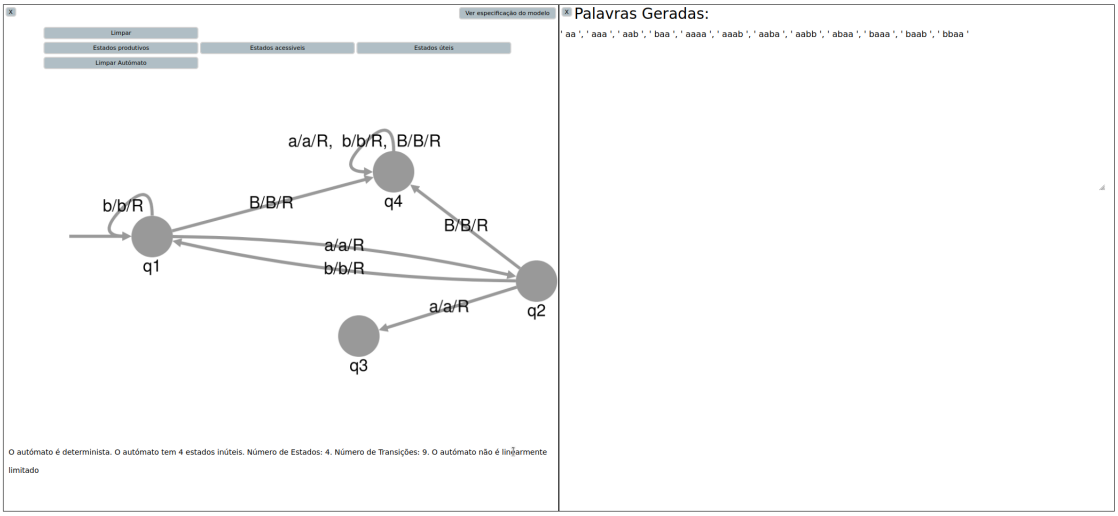


Figura 6.10: Resultado da operação de geração

6.3.4 Operações de conversão

A operação de conversão implementa todas as conversões desenvolvidas no OCaml-FLAT, sendo possível fazer a conversão de qualquer modelo para MT e de MT para AF. Quando o utilizador quiser converter o modelo em que edita para outro, basta clicar em *Convert To* e selecionar o modelo para o qual queremos converter no *dropdown* menu. O autómato resultante aparece no painel de resultados ao lado, permitindo ao utilizador comparar os modelos com mais facilidade.

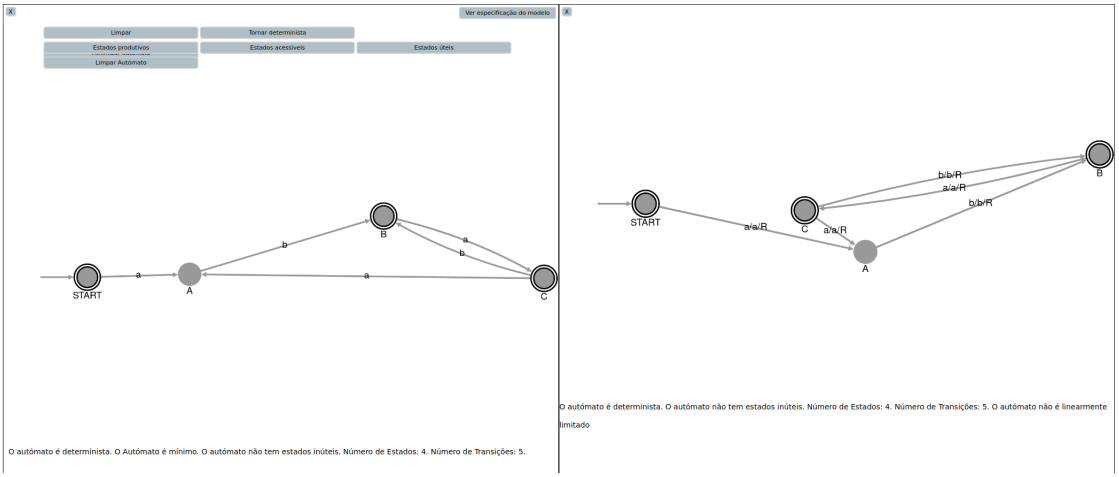


Figura 6.11: Resultado da operação de conversão

6.3.5 Operações de categorização dos estados do autómato

Para além das operações discutidas, temos ainda aquelas que enriquecem o conhecimento do utilizador sobre o autómato sobre o qual trabalha. Existem 3 que destacam na representação do autómato, os estados que são: produtivos 6.13, acessíveis 6.12 e úteis 6.14.

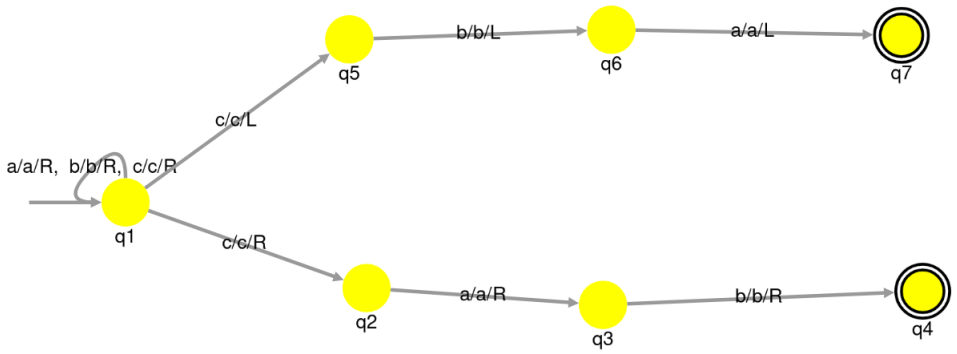


Figura 6.12: Estados acessíveis

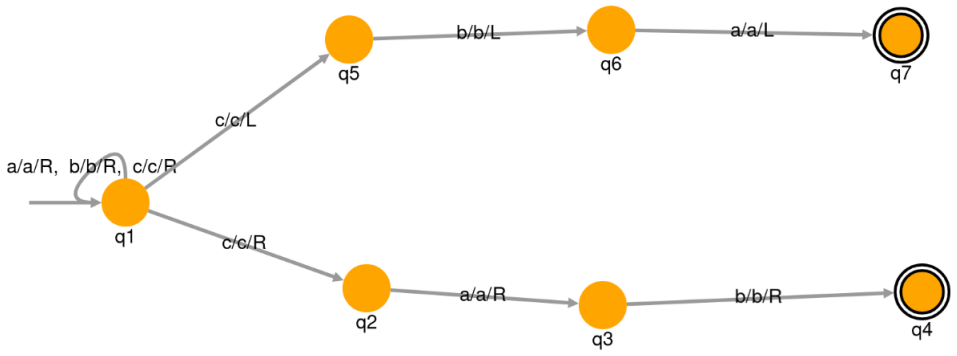


Figura 6.13: Estados produtivos

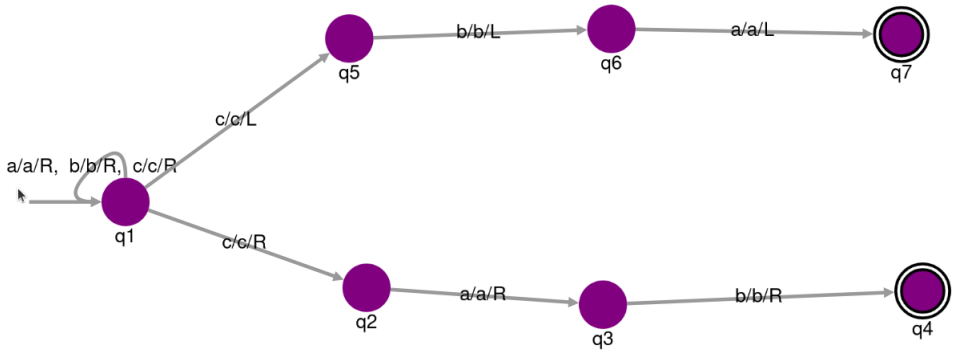


Figura 6.14: Estados úteis

6.3.6 Operação de limpeza de estados inúteis

Dado que estas operações podem poluir a representação, é disponibilizada uma operação para limpar a representação do autómato do filtro aplicado.

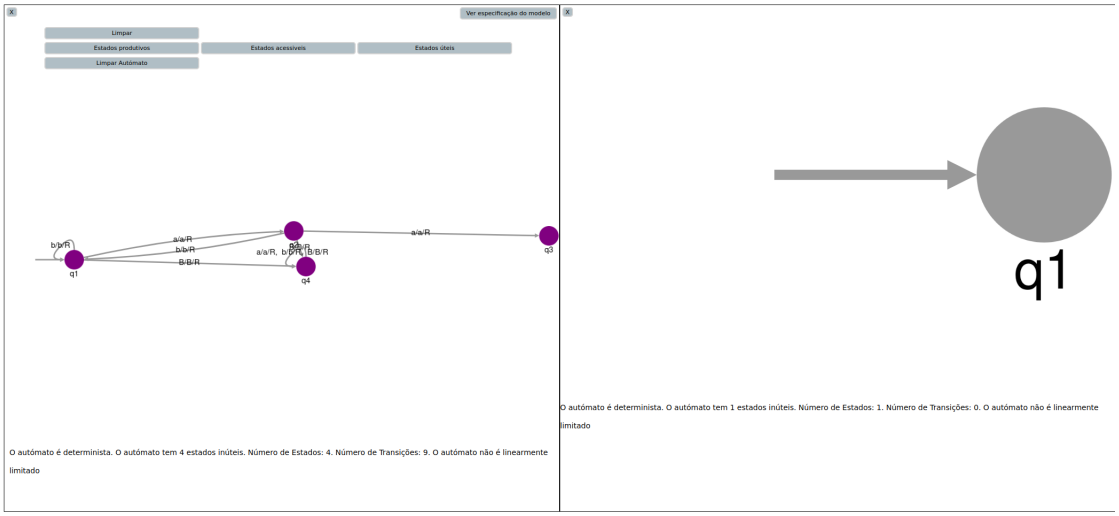


Figura 6.15: Resultado da limpeza

6.3.7 Informação sobre o autómato

Por baixo da tela, existe um parágrafo com várias informações relevantes sobre o modelo representado, que nos dizem se este é determinista; se tem estados inúteis; se é linearmente limitado; e o número de estados e transições. Esta informação é sempre atualizada quando é feita uma alteração no modelo.

> O autómato é determinista. O autómato tem 4 estados inúteis. Número de Estados: 4. Número de Transições: 9. O autómato não é linearmente limitado

Figura 6.16: Informação sobre o autómato

6.4 Implementação

6.4.1 Controller

Entre as operações que podemos executar no menu esquerdo e na representação, existem várias formas de interagir com a aplicação, sendo que os eventos por elas desencadeados são da responsabilidade dos controladores que lidam com eventos e dos ouvintes que os recebem. Cada um destes contém diversas funções que são chamadas, dependendo do evento que as chama.

Respetivamente aos controladores, existem sempre definidos dois, o *ctrlL* e o *ctrlR*, sendo que estes controlam a lógica dos painéis principal e de resultados, respetivamente. Eles podem ser de vários tipos, consoante o que estiver a ser representado no painel que controlam, como de conceitos FLAT como, por exemplo, o *tmController* se estiver a ser representada uma MT ou *textController*, no caso de exercícios ou especificação JSON de um modelo.

Quanto aos ouvintes, estes estão divididos: em tipo geral, de modelo, ou específicos a um conceito FLAT. O de tipo geral lida com eventos que afetam o estado dos controladores, enquanto os outros dois acionam quando é desencadeado um evento relativo a modelos de conceitos FLAT. Especificamente, o de modelo lida com eventos que chamam funções comuns a todos os modelos, como o *addNode* que adiciona um novo estado ao conjunto de estados ou o *addTransition* que adiciona uma nova transição ao conjunto de estados. Já nos ouvintes específicos a um conceito FLAT, são desencadeadas funções, que até podem ser comuns entre modelos, mas a sua implementação necessita de diferir para cada um.

Para poder criar, editar e aplicar operações relacionadas com MT, precisamos de criar um controlador e um ouvinte específicos para cada uma delas.

6.4.1.1 Accept

A animação interativa do accept tem uma natureza imperativa porque é preciso manter uma descrição da situação corrente, que vai sendo modificada em resposta aos *inputs* do utilizador. Para registar essa descrição, a aplicação usa um grupo de variáveis mutáveis. Por exemplo, uma variável chamada *steps* que regista todos os passos até ao momento (para permitir a navegação) e uma variável chamada *position* que indica qual desses passos é o corrente.

A operação de verificação se uma palavra é aceite pelo modelo começa quando o Controller chama a função *startAccept*. Para inicializar a simulação, é necessário preparar as variáveis mutáveis.

Para preparar estas variáveis, sendo que para umas temos já valores definidos para estas quando a operação inicia, e para outras como o *configsList* e *bestPath* recorremos à função $acceptFull$ para as inicializar.

Agora que temos as variáveis já devidamente inicializadas e tendo já o resultado do accept para a palavra, verificamos se a operação não tem nenhum passo para dar e nesse caso, acaba.

Quando damos um passo para a frente na simulação, temos de verificar se a operação já acabou, caso contrário, incrementamos o valor do *position*. Por outro lado, quando é para dar um passo para trás na simulação, verificamos se já estamos no primeiro passo, caso contrário, o *position* é decrementado.

No final de cada passo, atualizamos os *poppers*, as listas de configurações, a fita e as cores dos nós.

6.4.1.2 Operações restantes

A implementação das restantes funcionalidades, como a edição e criação de um modelo, operações de conversão e de categorização de estados, são feitas recorrendo a funções pertencentes ao OCaml-FLAT e aos controladores

7

Avaliação Crítica

Uma tarefa importante de desempenhar durante o desenvolvimento da ferramenta é o de verificar se as operações estão devidamente implementadas e funcionam como esperado. Neste capítulo vamos apresentar os métodos usados para fazer a verificação, para o OCaml-FLAT e para o OFLAT e os resultados obtidos.

7.1 OCaml-FLAT

A biblioteca OCaml-FLAT suporta a criação de módulos cujo objetivo é a criação e execução automática de testes unitários. Estes testes contêm vários exemplos de MT de diversos tipos e várias funções para cada operação do modelo. Cada função testa várias MT, e no caso da operação de accept, também para várias palavras, sendo que para cada uma, comparamos o resultado obtido com o esperado. Das operações testadas, houve um maior cuidado ao testar a operação de accept, por ser uma operação chave da ferramenta. Isto foi feito através de um aumento do número de palavras e exemplos a testar, bem como a criação de exemplos que desafiem os limites da operação. Um destes exemplos, que é importante testar, é o da possibilidade de a máquina entrar num ciclo infinito.

Durante os testes, foram encontrados alguns problemas com as operações de *productive* e *reachable*, cuja intenção inicial era a de reutilizar as funções utilizadas pelo AF. Mas devido à existência do critério de paragem para MT, a operação *reachable* tem de ser feita de forma diferente, e visto que o *productive* usa esta operação, temos também de criar um novo.

7.2 OFLAT

Por outro lado, a aplicação OFLAT não contém suporte para testes unitários. Para testar e avaliar a execução da aplicação, temos de recorrer ao uso da aplicação. Se possível, este tipo de teste seria feito com várias pessoas, de preferência alunos da cadeira de TC, no entanto, tal não foi possível. Apesar disso, em conjunto com o coordenador, as funcionalidades da aplicação foram sendo discutidas e avaliadas.

Na avaliação, foram detetados alguns problemas no funcionamento da aplicação, que têm impacto na experiência do utilizador.

Nomeadamente, quando é aberto um painel de resultados, ele ocupa exatamente metade do espaço disponível na página, podendo isto ser inconveniente se o espaço for necessário para o painel principal. Outro problema relacionado com o espaço é a de que quando queremos ajustar o *zoom* do painel principal, cada scroll do *mousewheel* aumenta e diminui o *zoom* exageradamente, pois o valor de *zoom* para cada scroll é demasiado alto.

O carregamento de um exemplo na aplicação é uma funcionalidade importante, e para certos exemplos é essencial os estados estarem organizados de forma clara e organizada. Apesar de ser possível mover os estados para as posições desejadas, elas não são guardadas. Isto obriga a ter de organizar o exemplo, sempre que este é carregado.

8

Conclusão e Trabalho Futuro

8.1 Conclusão

O trabalho feito nesta tese consistiu em estender as funcionalidades da biblioteca OCaml-FLAT e da aplicação OFLAT, para permitir o uso de MT.

Para o OCaml-FLAT, foram criadas estruturas de dados que representam uma MT e implementadas operações que nos permitem explorar as capacidades e características de uma MT. As operações mais importantes de mencionar são as de aceitação de palavras, geração de palavras aceites pelo modelo, conversão de modelos e de categorização de estados. Para além de suporte para MT, a biblioteca também suporta MT linearmente limitadas, e está também disponível uma operação que verifica se uma MT é linearmente limitada e de aceitação de palavras.

O OFLAT possui uma nova interface para a criação e edição de MT que implementa as funções da biblioteca. A implementação da função de accept é feita com recurso a uma representação gráfica e uma animação onde o utilizador pode seguir a simulação da máquina durante a operação. Para além do accept, são também implementadas as funções de geração de palavras, conversão de modelos e categorização de estados.

8.2 Trabalho Futuro

Apesar de terem sido implementadas as funcionalidades inicialmente planeadas, existem ainda melhorias que podem ser efetuadas.

Na parte da aplicação, estas melhorias passam por permitir ao utilizador um maior controlo sobre a edição do modelo representado ao permitir a definição dos marcadores de uma MT linearmente limitada diretamente no painel principal, editar as transições na representação gráfica.

Não só a edição, mas também a simulação da aceitação de uma palavra pode ser aperfeiçoada ao criar botões que, durante a simulação, permitam parar, ir para o fim e voltar ao início da mesma. Além disso, também se pode implementar na lista de configurações de um estado, um *scroll bar* que permite visualizar mais configurações,

quando existirem demasiadas.

Bibliografia

[1] P. Chakraborty, P. Saxena e C. Katti. *Fifty Years of Automata Simulation: A Review*. Fifth. McGraw-Hill, 2006. ISBN: 007-124476-X (ver p. 19).

[2] W. contributors. *Functional Programming— Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Functional_programming (acedido em 2022-11-17) (ver p. 16).

[3] J. E. Hopcroft, R. Motwani e J. D.Ullman. *Introduction to Automata Theory, Languages, and Computation*. Third. Pearson Education, 2006. ISBN: 0-321-45536-3 (ver pp. 3, 4, 7, 8).

[4] *Interactive Turing Machine*. URL: https://daru13.github.io/interactive-turing-machine/ (acedido em 2022-04-13) (ver p. 21).

[5] *Jflap*. URL: https://www.jflap.org/ (acedido em 2022-04-13) (ver p. 22).

[6] *Js_of_ocaml*. URL: https://ocsigen.org/js_of_ocaml/latest/manual/overview (acedido em 2022-05-09) (ver pp. 40, 43).

[7] P. Linz. *An Introduction to Formal Languages*. Fifth. Jones Bartlett Learning, 2012. ISBN: 978-1-4496-1552-9 (ver p. 6).

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

[9] L. Monteiro. «Linguagens Formais e Autómatos; Acetatos das aulas teóricas». Em: *Licenciatura de Engenharia Informática, DI/FCT/UNL* (2001) (ver p. 12).

[10] *Mr-Turing*. URL: https://github.com/timlg07/Mr-Turing (acedido em 2022-04-13) (ver p. 20).

[11] *Online Turing Machine Simulator*. URL: https://turingmachinesimulator.com/ (acedido em 2022-04-13) (ver p. 20).

[12] O. Torgersson. *A Note on Declarative Programming Paradigms and the Future of Definitional Programming*. S-412 96 Göteborg, Sweden (ver p. 17).

[13] A. M. Turing et al. «On computable numbers, with an application to the Entscheidungsproblem». Em: *J. of Math* 58.345-363 (1936), p. 5 (ver p. 5).

[14] *Turing Machine Visualization*. URL: https://turingmachine.io/ (acedido em 2022-04-13) (ver p. 20).