Universidade de Évora
Escola de Ciências e Tecnologia
Departamento de Informática

Mestrado em Engenharia Informática

Autómatos Recursivos

Filipe dos Santos Vieira

Orientador: **Francisco Manuel Gonçalves Coelho**

Évora, 26 de Outubro de 2009

Universidade de Évora
Escola de Ciências e Tecnologia
Departamento de Informática

Mestrado em Engenharia Informática

Autómatos Recursivos

Filipe dos Santos Vieira

Orientador: **Francisco Manuel Gonçalves Coelho**

Évora, 26 de Outubro de 2009

Prefácio

Este documento contém uma dissertação intitulada «Autómatos Recursivos», um trabalho do aluno Filipe dos Santos Vieira[1], estudante de Mestrado em Engenharia Informática, da Universidade de Évora.

O orientador deste trabalho é o Professor Doutor Francisco Coelho[2], do Departamento de Informática da Universidade de Évora.

O autor do trabalho é licenciado em Engenharia Informática, pela Universidade de Évora. A presente dissertação foi entregue em 26 de Outubro de 2009.

[1] sv.filipe@gmail.com
[2] fc@uevora.pt

Agradecimentos

O trabalho de Mestrado é um desafio que exige do aluno total entrega e dedicação, mas é também um processo em constante evolução que envolve vários intervenientes. Assim dedico algumas palavras às pessoas que de alguma forma contribuíram para alcançar a finalização deste trabalho.

Agradeço ao meu orientador por toda a ajuda na parte escrita e teórica deste trabalho. Agradeço também à professora Irene Rodrigues, por todo o incentivo dado na finalização desta etapa académica, e por toda a ajuda no campo de Processamento de Linguagem Natural. Ao professor Vasco Pedro, pela ajuda em algumas questões sobre autómatos. Aos restantes professores da Universidade de Évora que contribuíram para a minha formação.

À minha família que sempre me apoiou em todos os aspectos da minha vida, e que continua a apoiar. Obrigado Mãe, Pai, Isabel, Joana, Avó, Tios e Primas.

E a todos os meus amigos, em especial às pessoas que sempre acreditaram no meu trabalho e que me deram um grande incentivo, Júlio e Carlos.

Ao Filipe, Luís, Feitor, Mesquita, Marques, João, Cris, Ana e Marina, por me terem aturado, especialmente este ano. Ao Pedro, não apenas pela amizade, mas também pelas varias sessões de estudo em Análise Matemática I.

Ao Paulo, por ter sido um mentor em Linux e código livre. Ao Tiago e João com quem tive o privilégio de partilhar um projecto em C++, e algumas horas de biblioteca.

Ao António pelas revisões gramaticais e linguísticas deste trabalho.

E por fim, a toda a comunidade de código livre, que tornou este trabalho possível.

Sumário

As sociedades humanas são complexas. Essa complexidade tende a aumentar, sendo a tecnologia um dos motores desse percurso. Não só promotora de certo progresso, a tecnologia requer a construção e gestão de sistemas cada vez mais sofisticados. Numa perspectiva histórica, só recentemente é que se deram avanços significativos no enquadramento teórico da tecnologia, com a Cibernética (em inglês *cybernetics*, «Ciência que estuda os mecanismos de comunicação e de controlo nas máquinas e nos seres vivos.» [PRI]) e a Teoria dos Autómatos [RM01].

Na vertente aplicada destes progressos teóricos, encontramos uma variedade de sistemas implementados com vista a:

- geração de código;
- análise sintáctica e lexical;
- construção e validação de sistemas digitais;
- demonstração automática de teoremas;
- pesquisa de soluções-caminhos nos espaços de estados de problemas.

Desta variedade resulta que cada implementação particular tende a especializar-se numa determinada tarefa/objectivo. Resulta então que uma aplicação que use uma biblioteca especializada (digamos, de análise sintáctica) fica limitada à especialidade dessa biblioteca. Também acontece que certas aplicações acabam por implementar as suas próprias ferramentas.

Neste trabalho foi definido um certo tipo de autómato (AR, autómato recursivo) que acreditamos ter a mesma capacidade computacional dos autómatos de pilha (AP, em inglês *push-down automata*, PDA) mas mais simples e intuitivo. Além dessa definição, associam-se metadados/acções a símbolos terminais e palavras aceites.

Com essa base teórica implementou-se uma biblioteca que procura facilitar o desenvolvimento de aplicações onde seja necessário ou útil a construção e execução de autómatos. A título demonstrativo foi construída uma aplicação, um gerador de analisadores sintácticos e lexicais, através de especificações FBNA (Forma Backus–Naur Aumentada, em inglês *Augmented Backus–Naur Form*) [Net08].

Recursive Automata (Abstract)

Human societies are complex. This complexity tends to increase, technology being one of the drivers of this progress. Technology not only promotes a certain progress but requires the construction and management of highly sophisticated systems. In a historical perspective, it was only recently that significant advances were made in the theoretical framework of technology, with Cybernetics (the science of the mechanisms of communication and control in machines and living things.) and Automata Theory [RM01].

In the applied aspects of these theoretical progress, we find a variety of systems implemented with the objective of:

- Generation of a code;
- Making syntactic and lexical analysis;
- Construction and validation of digital systems;
- Automatically proving theorems;
- Searching solutions-paths in problem's state-space.

From this variety follows that each particular implementation tends to specialize in a particular task/objective. It follows then that an application that uses a specialized library (say, parsing) is limited to that library expertise. It also happens that some applications ultimately implement their own tools.

In this work is defined a certain type of automaton (RA, recursive automaton) with the computing power of pushdown automata but simpler and more intuitive. Besides this definition, we associate meta-data/actions to the terminal symbols and the accepted words.

With this theoretical basis a library was implemented that seeks to facilitate the development of applications where it is necessary or useful the construction and implementation of automata. Based on the definition a library is implemented that allows applications to develop based on the construction and implementation of automata. For demonstration purposes, an example application was built, through ABNF (Augmented Backus–Naur Form) grammars specifications [Net08], a syntax/lexical parser generator.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

São descritos brevemente os conceitos e o tema que enquadram este trabalho. Também são delineados os principais objectivos.

1.1 Enquadramento e Motivação

As sociedades ocidentais/ocidentalizadas contemporâneas dependem vitalmente da informática. Essa dependência manifesta-se principalmente na realização de certas tarefas automáticas de tal forma que, faltando a infra-estrutura computacional, seria actualmente impossível providenciar, apenas com recurso a mão-de-obra humana, o vasto leque de serviços por esta proporcionado.

Resulta desta situação o aumento sistemático da quantidade e complexidade dos problemas tratados por sistemas informáticos. Quando o número de itens processados atinge os milhões, já não bastam soluções ingénuas assentes na força-bruta do (também crescente, mas exponencialmente menos) poder computacional dos artefactos. É necessário encontrarem-se novos meios de produzir programas e de gerir eficientemente a complexidade dos problemas.

Já não são novidade os paradigmas funcional, modular, orientado a objectos, bem como geradores de código ou outras metodologias que procuram automatizar, organizar e optimizar as várias tarefas delegadas no programador.

A teoria dos autómatos sustenta muitas dessas propostas. Resumidamente, um autómato é um sistema autónomo que processa informação. A ciência da computação, como sub-disciplina da matemática, estuda modelos formais, abstractos, de tais autómatos e também dos problemas que estes podem abordar/resolver. Na sua forma mais simples, um autómato de estados finitos, AEF, (em inglês *finite state automathon, FSA*) é definido por um conjunto de estados e uma função de transição que, dado um estado e um símbolo lido, determina o próximo estado. A computação destes autómatos é então a iteração desta função a partir de um estado inicial e de uma sequência de símbolos a serem lidos.

A simplicidade conceptual, bem como um alargado conjunto de propriedades não triviais, fazem dos autómatos e da sua teoria uma das mais importantes ferramentas para a simplificação de problemas complicados. Apesar da elegância teórica, o programador praticante, em geral, preocupa-se com a construção, optimização e execução de autómatos, mas é também necessário atender a questões de complexidade, determinação, memória, tempo e aplicabilidade.

1.2 Objectivos

Procuramos resolver alguns problemas associados aos autómatos no ramo da informática, identificando os problemas existentes, e de uma forma geral, tentamos resolvê-los desenvolvendo mecanismos que permitam abranger o maior número possível de aplicações.

Pretendemos ir mais além, e permitir a construção de autómatos deterministas em ambientes interactivos e adaptativos.

A nossa estratégia para abordar estas questões passa por propor, e analisar, um novo tipo de autómato, que designamos por «Automáto Recursivo» – AR, e, com essa base teórica implementar uma biblioteca de uso geral e flexível, com vista a poder ser facilmente integrada no maior leque possível de aplicações.

Capítulo 2

Trabalho relacionado

2.1 Breve História da Teoria de Autómatos

Em 1930 Alan Turing apresenta o seu trabalho sobre máquinas abstractas (conhecidas, em sua homenagem, como máquinas de Turing em inglês; *Turing Machines*). Estas fornecem uma ferramenta teórica para lidar com o conceito de «computação». O objectivo de Turing era explorar as limitações da computação, do que resultou na classificação dos problemas «algorítmicos» como:

- decidível, onde uma máquina dá sempre resposta positiva ou negativa;
- indecidível, onde, em certos casos, a máquina entra numa computação infinita, sem resposta definida.

Entre 1940 e 1950, surgiram máquinas mais simples, conhecidas como Autómatos Finitos. Nos finais da década de 1950, o linguista Noam Chomsky inicia o estudo de gramáticas formais, e a sua relação com os autómatos, vindo mais tarde a classificá-los de acordo com a capacidade de reconhecer determinadas linguagens. Em 1969, Stephen Cook aprofundou a classificação decidível/indecidível de Turing, distinguindo como problemas que são decidíveis em tempo praticável daqueles que não o são. Estes últimos são conhecidos como problemas intratáveis.

2.2 Conceitos Envolvidos

Para melhor entendimento deste trabalho introduzimos alguns conceitos da terminologia utilizada na teoria de autómatos.

Símbolo «Figura ou imagem que representa à vista o que é puramente abstracto» [PRI];

Alfabeto conjunto finito de símbolos;

Palavra é uma sucessão finita de símbolos de um alfabeto. A maior ordem dessa sucessão (w) é o **comprimento** da palavra, denotado como $|w|$. A palavra de comprimento zero é denotada por ϵ;

Fecho de Kleene é o conjunto de todas as palavras sobre um alfabeto. Em particular, se Σ for um alfabeto, representamos por Σ^k o conjunto de todas as palavras de comprimento k e

$$\Sigma^* = \bigcup_{i=0}^{+\infty} \Sigma^i$$

e

$$\Sigma^+ = \bigcup_{i=1}^{+\infty} \Sigma^i$$

;

Concatenação se x e y forem palavras então xy denota a concatenação com x e y, que equivale à palavra x seguida de y. Ou seja, se $x = a_0a_1...a_{|x|}$ e $y = b_0b_1...b_{|y|}$, então $xy = a_0a_1...a_{|x|}b_0b_1...b_{|y|}$, $(|xy| = |x| + |y|)$;

Linguagem conjunto de palavras de algum conjunto Σ^*, onde Σ é um alfabeto em particular, isto é $\mathcal{L} \subseteq \Sigma^*$ se e só se $\mathcal{L}$ é uma linguagem sobre Σ;

Estado «Modo geral; condição, disposição, situação, posição, circunstâncias em que se está e se permanece» [PRI].

2.3 Estado da arte

Existem actualmente várias bibliotecas que produzem ou auxiliam na **construção de autómatos**. Para obtermos uma perspectiva geral consultámos a documentação das 14 principais implementações (indicadas na bibliografia).

Nesta análise foram considerados os seguintes aspectos:

- Plataformas e Linguagens;
- Interacção entre a biblioteca e o utilizador (interface);
- Autómatos suportados;
- Operações suportadas e principais características internas;
- Perfil dos autómatos produzidos;

Baseámo-nos no estudo de Chomsky (Hierarquia de Chomsky) [RSE94] para comparar os diferentes autómatos, ordenando-os de acordo com o tipo de linguagem reconhecida. Na hierarquia de Chomsky são definidas quatro classes encadeadas:

Classe	Tipo de Linguagem	Tipo de Computador
tipo 0	recursivamente enumerável	Máquinas de Turing (MT)
tipo 1	com contexto	MT não-deterministas linearmente delimitadas
tipo 2	livre de contexto	Autómato de pilha não-determinista
tipo 3	regular	Autómato finito

Tabela 2.1: Hierarquia de Chomsky

A tabela seguinte resume a informação sobre os tipos de linguagens suportadas pelas diferentes bibliotecas. Indicamos também se são (ou não) admitidas computações deterministas ou não-deterministas.

Terminamos a análise do estado da arte com uma discussão sobre as suas limitações, vantagens e desvantagens encontradas.

2.3.1 Plataformas e Linguagens

As bibliotecas consideradas são, geralmente, independentes do sistema operativo, porém na sua maioria suportam apenas uma linguagem.

Maioritariamente são usadas linguagens interpretadas (Ocaml, Haskel, Prolog, Lisp, Elan e Scheme), mas também existem algumas implementações em C, C++ e Java.

2.3.2 Interacção entre a biblioteca e o utilizador

Os serviços de uma biblioteca podem ser acedidos de formas diferentes, que agrupamos:

Implementação	Classe/Tipo				Suporte	
	3	2	1	0	determinismo	indeterminismo
LAML [LAM]	✓				✓	✓
TATA [Fil03]	✓	✓			✓	✓
FSA6.2XX [eDG03]	✓	✓			✓	✓
ELAN [Gen98]	✓	✓				✓
Timbuk [Gen01]	✓	✓			✓	✓
Automata Library for Haskell [eLS01]	✓				✓	✓
JAutomata Library [JAU]	✓				✓	✓
Visibly Pushdown Automata Library [eMS06]	✓	✓			✓	✓
NPCRE [NPC]	✓				✓	
AMoRE [OMV95]	✓				✓	✓
GENAU [Kra08]	✓				✓	✓
ASTL [Mao]	✓				✓	✓
Ijaguar [Ser02]	✓	✓	✓	✓	✓	✓
Syncroteam [eJC]	✓	✓			✓	✓
Mona [KM01]	✓					✓
Autowrite [Dur05]	✓	✓			✓	✓

Tabela 2.2: Bibliotecas estudadas e o tipo de autómatos que suportão

- pelo uso de ficheiros e representações intermédias (tais como expressões regulares ou sintaxes próprias) [NPC, Kra08];
- pelo acesso a um conjunto de funções (*API: Aplication Programming Interface*) para as operações básicas na construção de um autómato de forma directa; o programador define os estados e as transições;
- pelo uso misto de ambas formas anteriores [Ser02].

2.3.3 Tipos de autómatos suportados

As bibliotecas consideradas são, normalmente, direccionadas para o reconhecimento de expressões regulares. Daqui resulta serem usados, principalmente, os autómatos de estados finitos deterministas (AFD, em inglês *DFA, Deterministic Finite State Automata*). Para além dos AFD, algumas bibliotecas também produzem:

Autómatos Finitos Não-deterministas (AFN, em inglês *NFA, Non-deterministic Finite State Automata*) Máquinas com um número finito de estados, a função transição define, a partir de um estado de partida e símbolo de entrada, um conjunto de próximos estados possíveis, de onde será escolhido um. Uma palavra é aceite se existe um caminho (nos estados) com início no estado inicial e com fim num estado final;

Autómatos de Árvore (AA, em inglês *TrA: Tree Automata*) Lidam com estruturas em árvore em vez de palavras. Existem dois tipos; os (a) das folhas para a raiz (em ingles, *bottom up*) e (b) da raiz para as folhas (em ingles, *top*

down). Isto é importante pois os autómatos em árvore deterministas do tipo (b) são menos poderosos do que os do tipo (a). Todos os restantes, ou seja não-deterministas do tipo (a) e (b) são equivalentes;

Transdutores (AES, Autómatos de entrada e saída. Em inglês *I/O Automata*) Para além de aceitarem ou rejeitarem palavras sobre o alfabeto de entrada geram palavras sobre um alfabeto de saída;

Autómatos de Equipa (AE, em inglês *TA: Team Automata*) São usados para modelar sistemas, em que cada autómato é visto como um componente do sistema. Estes usam três alfabetos de símbolos:

1. entrada;
2. saída;
3. interno, para a sincronização de cada componente do sistema;

Autómatos de Pilha (AP, em inglês *PDA, Pushdown Automata*) Uma extensão dos Autómatos Finitos. A estes adiciona-se uma pilha/palavra sobre um alfabeto, a que se chama alfabeto da pilha. A função transição passa então a decidir o próximo estado tendo em conta o estado presente, o símbolo no topo da pilha e o símbolo de entrada. No caso da pilha estar vazia é considerado como estando no topo o símbolo ϵ. Em cada transição pode executar-se uma de duas operações sobre a pilha: inserir um símbolo no topo ou retirar um símbolo do topo. Neste caso a palavra é aceite se existir um caminho desde o estado inicial até um estado terminal, em que a pilha se encontra vazia;

Autómato de Pilha Visível (APV, *VPA: Visible Pushdown Automata*) São autómatos de pilha simplificados. As operações da pilha são apenas definidas pelos símbolos de leitura. A pilha aqui tem somente a função de contar os símbolos, resultando daí que este tipo de autómatos é menos expressivo do que os AP;

Máquinas de Turing (MT, em inglês *TM: Turing Machines*) Constituídas por um número finito de estados e uma fita potencialmente infinita. O conteúdo da fita é acedido através de uma cabeça de leitura e escrita de símbolos. Em cada transição é opcionalmente possível ler, escrever ou mover a cabeça uma posição para a direita ou esquerda. A transição é efectuada tendo em conta o estado actual e o símbolo lido da fita;

Cada biblioteca implementa apenas um tipo de autómato (algumas também as variantes não deterministas), com a notável excepção da biblioteca ijaguar que implementa todos os tipos na escala de Chomsky, a que correspondem os AFN, AFD, AP e MT.

2.3.4 Operações suportadas e principais características internas

Quase todas estas bibliotecas implementam as operações sobre autómatos mais comuns: passagem de um autómato não-determinista para o seu equivalente determinista, minimização, intersecção, união, fecho de kleene, remoção de transições ϵ, entre outras.

Algumas das bibliotecas ainda permitem condições de transição, em vez da simples leitura de símbolos. Nesses casos o cálculo de predicados ou funções sobre certos objectos podem influenciar as transições de estado.

Por último, os autómatos da biblioteca Autowrite [Dur05] têm a capacidade de rescrever as regras introduzidas.

2.3.5 Aplicações dos autómatos produzidos

Como já vimos, nestas bibliotecas os autómatos são definidos pela leitura de um ficheiro ou por uma sequência de instruções. A forma em que cada biblioteca proporciona esse autómato também varia:

- pode ser um fragmento de um programa numa certa linguagem alvo. O código produzido destina-se a ser incluído num programa maior. Geralmente a linguagem alvo coincide com a linguagem de implementação da biblioteca [Kra08, Mao];
- pode ser guardado num ficheiro especial, posteriormente interpretado por um programa dedicado [NPC];
- pode ser construído e usado em tempo de execução de um determinado programa cliente da biblioteca [Mao, LAM, eDG03, eLS01];
- os tipos do alfabeto de leitura podem ser símbolos do tipo carácter, fluxos [NPC] (de qualquer tipo de dados), objectos (Java [JAU]) e termos ou predicados (Prolog [eDG03]).

2.3.6 Conclusão

As bibliotecas avaliadas, na sua maioria, apresentam algumas características importantes, de que destacamos as operações mais comuns sobre autómatos. Podemos ainda considerar alguns aspectos interessantes:

- a simplificação automática de gramáticas, usando regras de transformação [Dur05];
- o uso de condições de transição, em vez de simples símbolos (predicados Prolog e objectos Java [JAU, eDG03]);
- a representação de fórmulas por autómatos, e a possibilidade de contra-prova [KM01, OMV95];
- a representação de autómatos em forma gráfica, usando o sistema GraphViz;
- a construção do autómato em tempo de execução;
- o uso de representações intermédias dos autómatos em XML [Ser02].

Por outro lado, não podemos deixar de observar algumas limitações:

- a linguagem alvo coincide, quase sempre, com a linguagem de implementação O uso da biblioteca fica assim limitado a uma linguagem;

- a interface disponibilizada é, na maioria das vezes, pouco sofisticada. Uma biblioteca mais avançada permitiria, por exemplo, suportar as especificações mais comuns de gramáticas, (e.g. FBN, Forma Backus-Naur, em inglês *BNF, Backus-Naur Form*);

- os tipos de autómatos produzidos são, quase sempre, autómatos de estados finitos, demasiado simples para importantes classes de linguagens, como as linguagens dependentes do contexto, por exemplo;

- os autómatos são, normalmente, reconhecedores de linguagens. Porém, para muitas situações não basta uma resposta booleana. É, frequentemente, conveniente associar acções ou sequências de acções ás palavras processadas pelo autómato;

- os códigos dos autómatos produzidos não têm nenhuma forma de modularidade. Numa situação típica o programador altera um determinado fragmento do código do autómato. Se, posteriormente, for necessário corrigir a especificação do autómato perdem-se as alterações introduzidas.

2.4 Identificação de Problemas

Estas bibliotecas estão limitadas pela linguagem de implementação. A plataforma .NET vem de alguma forma resolver esse problema, mas o facto de ser uma plataforma proprietária retira-lhe viabilidade, pois o estatuto legal e funcionalidade de ferramentas desenvolvidas para esta plataforma dependem de terceiros.

Este problema pode ser resolvido através de uma interface que permite a transformação dos resultados obtidos. Assim, uma única biblioteca define autómatos aptos a serem em várias linguagens (de programação ou de representação, e.g. para o GraphViz).

Por outro lado, o código gerado forma uma unidade (quase) indivisível. Esta falta de modularidade tem grande impacto no desenvolvimento de aplicações. Para que um programador possa associar determinadas acções à aceitação de determinadas palavras, terá de modificar fragmentos do código do autómato. Porém, se posteriormente alterar a especificação, as alterações introduzidas serão perdidas. Mas se o código gerado for modular, uma alteração local na especificação deste irá apenas alterar uma pequena parte do código.

Uma das principais vantagens dos AFD reside na sua simplicidade. No entanto, sendo basicamente reconhecedores com memória fixa, estão limitados ao reconhecimento de linguagens regulares, o que se reflecte nos perfis de grande parte das bibliotecas analisadas. Se o modelo básico dos AFD fosse estendido, de alguma forma, num compromisso entre simplicidade e expressividade, talvez passasse a ser então possível implementar uma biblioteca baseada num modelo simples de autómato, sem que essa simplicidade venha comprometer a aplicabilidade/expressividade.

Concluindo, achamos que existem poucas (ou nenhumas) bibliotecas práticas feitas a pensar no desenvolvimento de aplicações sobre autómatos, de uma forma geral e não especializada. As interfaces que transformam os dados antes e depois do processamento interno da biblioteca, não levam em consideração os problemas

encontrados pelos programadores, de forma a que o desenvolvimento de aplicações se torna pouco produtivo.

Capítulo 3

O Sistema Proposto

As secções que se seguem neste capítulo fazem de uma forma geral a apresentação do sistema e a descrição dos objectivos.

Nos capítulos posteriores é feita uma análise mais detalhada de todos os aspectos aqui retratados.

3.1 Apresentação

A teoria dos autómatos é uma ferramenta poderosa, fortemente ligada à informática e aos seus problemas práticos.

Sabemos, pelo estudo de Chomsky, que nem todos os autómatos possuem o mesmo poder, sendo estes classificados pelas linguagens que reconhecem.

Foi com base neste estudo e da análise a trabalhos relacionados [2.3] que fizemos uma apreciação do tipo de implementações existentes, o que nos permitiu não só classificar as bibliotecas conforme as suas limitações e capacidades, mas também perceber quais os tipos de autómatos mais requisitados.

Daí concluímos de que a classe associada a linguagens livres de contexto é a mais abrangente [2.3]. As linguagens livres de contexto deterministas são umas das mais requisitadas pois, devido às implicações de consumo de recursos, muitas exigem características como o determinismo e a minimização.

Da transposição da teoria para a prática, identificámos problemas relacionados com a associação de informação adicional pretendida por cada aplicação, com o objectivo de apresentar resultados contextualizados no campo onde a aplicação se enquadra, sejam eles em forma de acções ou dados.

Existe portanto interesse na conciliação entre a área da teoria dos autómatos aplicada aos problemas práticos da informática, de forma a encontrar um ponto de partida comum facilitando assim a transição de uma para a outra. Concluímos que existe a necessidade de mecanismos mais simples mas poderosos, direccionados para os problemas identificados.

Assim propomos:

- a definição de um autómato simples do tipo 2 (AR, autómato recursivo);
- definir a sua versão determinista (ARD, autómato recursivo determinista);
- permitir a associação de informação adicional (metadados);

Como este é também um trabalho prático, implementamos uma biblioteca com base nas definições, que recebe como entrada AR's e os converte internamente sempre que possível, no seu equivalente determinista (ARD). Por fim disponibilizamos uma interface que permite a sua transformação para vários formatos de saída.

3.2 O que se pretende?

Pretendemos evitar a fragmentação que existe na transposição da teoria de autómatos para a sua implementação na informática. A máxima «reinventar a roda» bem conhecida na informática, relembra-nos a não começar algo já disponível. No campo dos autómatos, esta parece estar esquecida, havendo inúmeras implementações, cada com as suas características.

Esta heterogeneidade provoca uma fragmentação neste campo, o que se reflecte na maturidade das implementações disponíveis, obrigando os programadores a escolherem soluções mais maduras fora desta área.

Pretendemos minimizar estes problemas introduzindo uma biblioteca que ofereça um grande nível de abstracção na camada de definição e manipulação dos autómatos. Esta camada oferece ao programador todas as características da teoria dos autómatos e o controlo total sobre a definição dos seus elementos abstractos assim como o acesso a todo o autómato.

Sendo esta uma biblioteca de uso geral, pode ser utilizada e reutilizada num grande número de aplicações formando uma comunidade que contribua para a sua evolução e maturação como um único projecto.

Futuramente gostaríamos de ver a sua expansão para diferentes áreas ainda não abordadas, tais como sistemas adaptativos e interactivos.

3.3 Autómatos finitos recursivos não-deterministas

Tal como referimos nas secções anteriores um novo tipo de autómato é proposto neste trabalho, assim nesta secção iremos apresentar os conceitos e as ideias que de onde emergem os autómatos finitos recursivos. Começamos por alguns conceitos fundamentais já existentes, sobre eles iremos construir a nossa proposta e uma definição formal. Por fim um estudo/analise sobre a nova definição é efectuado.

3.3.1 Conceitos fundamentais

Definição 1 *Um autómato finito não determinista, AFN, é um 5-tuplo:*

$$(Q, \Sigma, \delta, q_0, F)$$

em que

- Q *é o conjunto de* **estados de controlo** *de* M*;*
- Σ *é um alfabeto, o* **alfabeto de entrada***;*
- δ *é a função de* **transição**, *com assinatura*

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}\{Q\};$$

- $q_0 \in Q$ *é o* **estado inicial***;*
- $F \subseteq Q$, *é o conjunto de* **estados finais**.

Definição 2 *Seja* M *um AFN tal que* $M = (Q, \Sigma, \delta, q_0, F)$, *e* $w = w_0 w_1 \ldots w_n$ *uma palavra de* Σ^*. M *aceita* w *se existir em* Q *uma sequência de estados* $c = q_0, q_1, \ldots, q_m$ *em que*

1. $m \geq n$
2. $q_{i+1} \in (\delta(q_i, w_i) \vee \delta(c_i, \epsilon))$, *para* $i = 1, \ldots, m$
3. $c_m \in F$

O conjunto das palavras aceites por um autómato, M, *representamos por* $\mathcal{L}(M)$. *Um resultado elementar da teoria dos autómatos que enunciamos aqui sem demonstração, diz que a classe das linguagens reconhecidas por AFN coincide com a classe das linguagens regulares.*

Definição 3 *Uma linguagem regular sobre um alfabeto* Σ *é descrita recursivamente pelas seguintes regras:*

1. *O conjunto vazio* $\emptyset$ *é uma linguagem regular;*
2. *A linguagem* $\{\epsilon\}$ *é regular;*
3. *Para todo* $a \in \Sigma$, *a linguagem* $\{a\}$ *é regular;*
4. *Se* A *e* B *são linguagens regulares então* $A \cup B$ *(união),* $A.B$ *(concatenação),* A^* *e* B^* *são linguagens regulares;*

5. *Nenhuma outra linguagem sobre Σ é regular.*

Como exemplos de linguagens regulares temos as linguagens finitas. Outro simples exemplo seria a linguagem regular a^*b^* sobre o alfabeto $\{a, b\}$. E como exemplo do que não é uma linguagem regular temos a linguagem $\{a^n b^n : n \geq 0\}$ sobre o mesmo alfabeto $\{a, b\}$.

Definição 4 *Uma gramática é um conjunto de regras sobre um alfabeto que descrevem ou produzem uma linguagem.*

Definição 5 *Uma gramática regular (GR) é uma forma de descrever uma linguagem regular. Existem dois tipos de gramáticas regulares, à direita e à esquerda, dependendo do tipo de gramática cada produção tem que ter a seguinte forma:*

- *A e B é um símbolo não terminal, e a é um símbolo terminal.*
- $A \rightarrow a$
- $A \rightarrow \epsilon$
- *Gramática regular à direita:*

  $A \rightarrow aB$
- *Gramática regular à esquerda:*

  $A \rightarrow Ba$

Definição 6 *Uma linguagem livre de contexto equivale ao conjunto de linguagens reconhecidas por um AP.*

Definição 7 *As linguagens livres de contexto podem ser descritas através de gramáticas livres de contexto.*

O termo «livre de contexto» provém do facto dos símbolos não terminais poderem ser rescritos de modo independente. Estas gramáticas desempenham um papel fundamental na descrição e desenho de linguagens de programação e compiladores assim como na validação da sintaxe de linguagens naturais.

Seja $G = (V, T, P, S)$ uma GLC, onde

- *V, é um conjunto finito de variáveis, também conhecidas como símbolos não terminais.*
- *T, o conjunto dos símbolos terminais.*
- $T \cap V = \emptyset$.
- $A = T \cup V$.
- $P = \{(r, w) : r \in V \wedge w \in A*\}$
- *S, é o símbolo inicial, $S \in V$.*

e alguns exemplos:

Exemplo 1 $G = (\{S\}, \{a, b, c\}, P, S)$

$$P = \{\ S \rightarrow abc\ ;$$

Exemplo 2 $G = (\{S\}, \{a, b, c, d, e\}, P, S)$

$$P = \begin{cases} S \rightarrow abc \\ S \rightarrow de \end{cases};$$

Exemplo 3 $G = (\{S, B\}, \{a, b, c\}, P, S)$

$$P = \begin{cases} S \rightarrow aBc \\ B \rightarrow b \end{cases};$$

Exemplo 4 $G = (\{S\}, \{a\}, P, S)$

$$P = \begin{cases} S \rightarrow SS \\ S \rightarrow a \end{cases};$$

Exemplo 5 $G = (\{S\}, \{a, b, c\}, P, S)$

$$P = \begin{cases} S \rightarrow Sa \\ S \rightarrow aSc \\ S \rightarrow abS \end{cases};$$

3.3.2 Equivalência entre gramáticas e autómatos

Como a classe das linguagens regulares coincide com a classe das linguagens geradas pelas gramáticas regulares sabemos que a cada AFN M podemos associar uma GR, G tal que $\mathcal{L}(M) = \mathcal{L}(G)$ e vice-versa. Vejamos alguns exemplos.

Começamos por alguns exemplos de produções regulares P com início no símbolo não terminal S e respectivos autómatos finitos (M):

Exemplo 6 $P = \{S \rightarrow abcS, S \rightarrow abc\}$

$M = (\{q_0, q_1, q_2, q_3\}, \{a, b, c\}, \delta, q_0, \{q_3\})$

$$\begin{cases} \delta(q_0, a) = \{q_1\} \\ \delta(q_1, b) = \{q_2\} \\ \delta(q_2, c) = \{q_3\} \\ \delta(q_3, a) = \{q_1\} \end{cases};$$

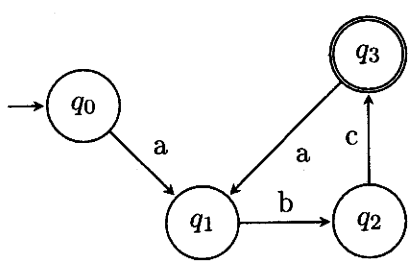

Figura 3.1: Autómato finito que reconhece a linguagem $abc(abc)^*$.

Vamos verificar a computação de algumas palavras sobre este AFN:

1. *abc é aceite pelo caminho* $q_0q_1q_2q_3$;

2. *abcabc é aceite pelo caminho* $q_0q_1q_2q_3q_1q_2q_3$;

3. *a é rejeitada pois a computação termina na sequência de estados q_0q_1, como q_1 não é final a palavra é rejeitada.*

4. *abcab é rejeitada pela mesma razão da anterior.*

Exemplo 7 $P = \{S \rightarrow abc, S \rightarrow abd\}$

$M = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{a, b, c\}, \delta, q_0, \{q_3\})$

$$\begin{cases} \delta(q_0, a) = \{q_1, q_4\} \\ \delta(q_1, b) = \{q_2\} \\ \delta(q_2, c) = \{q_3\} \\ \delta(q_4, b) = \{q_5\} \\ \delta(q_5, d) = \{q_3\} \end{cases} ;$$

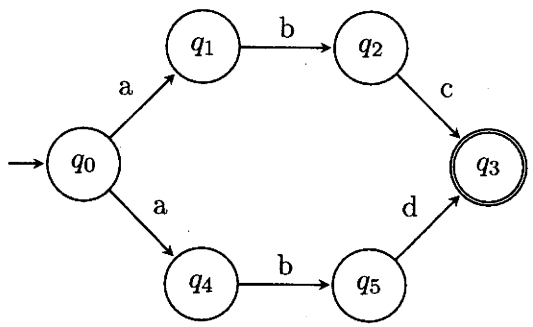

Figura 3.2: Autómato finito que reconhece a linguagem $\{abc, abd\}$.

Este AFN apenas aceita duas palavras, a palavra abc pelo caminho $q_0q_1q_2q_3$ e a palavra abd pelo caminho $q_0q_4q_5q_3$.

3.3.3 Os autómatos recursivos

O principal contributo teórico deste trabalho passa por estender um pouco a definição de AFN, de forma a permitir que as transições entre estados sejam efectuadas por «autómatos externos». Mais tarde daremos uma definição rigorosa desta ideia, mas por enquanto ficamos pela exploração informal desta ideia, através da apresentação de alguns exemplos. O segundo exemplo apresentado mostra um progresso interessante, em relação aos AFN. É que a linguagem reconhecida por este autómato **não é regular**. A linguagem dos parênteses é um dos casos que são normalmente usados para exibir uma linguagem de tipo 2 que não é de tipo 1.

As linguagens de tipo 2 são geradas pelas GLC, o nosso objectivo é reconhecer linguagens do tipo 2, daqui resulta a proposta de AR, em que adicionamos um alfabeto (Γ) de símbolos não terminais livres de contexto à definição de autómato finito.

Alguns exemplos informais:

Exemplo 8 $P = \{S \rightarrow abcS, S \rightarrow abc\}$

$M = (\{q_0, q_1, q_2, q_3\}, \{a, b, c\}, \Gamma = \{M\}, \delta, q_0, \{q_3\})$

$$\begin{cases} \delta(q_0, a) = \{q_1\} \\ \delta(q_1, b) = \{q_2\} \\ \delta(q_2, c) = \{q_3\} \\ \delta(q_3, M) = \{q_3\} \end{cases} ;$$

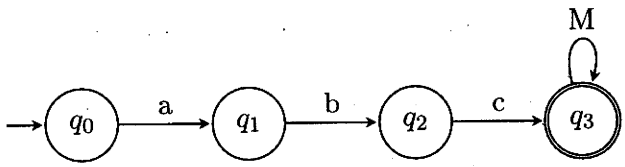

Figura 3.3: Autómato recursivo (M) com contexto $\Gamma = \{M\}$ que reconhece a linguagem $abc(abc)^*$.

Neste exemplo o autómato M aceita a linguagem $abc(abc)^*$. Iremos agora verificar passo a passo a computação de M sobre algumas palavras:

Começamos pela palavra mais simples abc, usando a definição de computação de autómato finito, podemos afirmar que a palavra abc é aceite pelo caminho $q_0q_1q_2q_3$. Portanto abc é um elemento da linguagem reconhecida por M.

Continuando, consideremos a palavra $abcabc$, começamos pelo prefixo abc em que sabemos que existe um caminho até ao estado final q_3 ficando assim por saber se existe um caminho a partir do estado q_3 que aceita o sufixo abc. No estado q_3 temos a seguinte transição $\delta(q_3, M) = \{q_3\}$, em que a transição é bem sucedida se um prefixo da palavra dada é aceite por M. No nosso exemplo temos abc que é aceite por M, então o caminho que aceita $abcabc$ é $q_0q_1q_2q_3q_3$. Então podemos afirmar que $abcabc$ é aceite por M, podemos então escrever a linguagem aceite por M da seguinte forma $\mathcal{L}(M) = abc\mathcal{L}(M) = abc(abc)^*$.

Damos agora alguns exemplos de palavras rejeitadas por M. A palavra ab é rejeitada por M. Fazendo a computação de ab verificamos que M pára na sequência de estados q_0q_1, q_1 não é um estado final logo a palavra não é aceite.

Consideramos agora a palavra $abcab$, como já vimos existe um caminho para o prefixo abc que termina no estado q_3, para que $abcab$ seja aceite M tem que aceitar ab e como já vimos no exemplo anterior isto não acontece logo $abcab$ é rejeitada.

Exemplo 9 *Linguagem $\{\epsilon, [], [[]], [[[...[]...]]], ...\}$ definida pelas produções $P = \{S \to [S], S \to \epsilon\}$*

$M = (\{q_0, q_1, q_2, q_3\}, \{a, b, c\}, \delta, q_0, \{q_0, q_3\})$

$$\begin{cases} \delta(q_0, [) & = & \{q_1\} \\ \delta(q_1, M) & = & \{q_2\} \\ \delta(q_2,]) & = & \{q_3\} \end{cases};$$

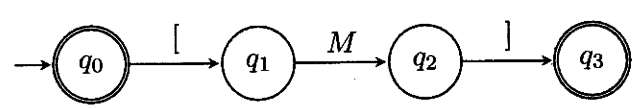

Figura 3.4: Autómato recursivo (M) com contexto $\Gamma = \{M\}$ que reconhece a linguagem $\{\epsilon, [], [[]], [[[...[]...]]], ...\}$.

Neste exemplo M aceita a linguagem $\{\epsilon, [], [[]], [[[...]]]\}$ do tipo 2.

Começamos pela palavra ϵ que é aceite pelo caminho q_0, assim $\epsilon \in \mathcal{L}(M)$ então $[]$ é aceite pelo caminho $q_0q_1q_2q_3$. Portanto temos $\mathcal{L}(M) = \{\epsilon, [\mathcal{L}(M)]\} = \{[^n]^n\}$.

Estes exemplos ilustram a ideia inspiradora dos autómatos recursivos, que iremos, explorar no resto deste capítulo.

3.3.4 Definição de Autómato Recursivo

Apresentamos agora a definição formal de AR, de forma a dar rigor aos exemplos anteriores. Ao contrario dos autómatos finitos, a computação dos AR requer um contexto. No centro da nossa proposta está a possibilidade da computação de um autómato evocar computações noutros autómatos. Esta capacidade está presente através dos contextos, que simbolizam «autómatos conhecidos». Vejamos como fica definida a computação de um AR, contemplando o seu contexto.

Definição 8 *Um* **autómato recursivo** *(AR) é um 6-tuplo:*

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

em que

- Q *é o conjunto de* **estados de controlo** *de M;*
- Σ *é um alfabeto, o* **alfabeto de entrada**;
- Γ *é um alfabeto, o* **contexto** *de M;*
- δ *é a função de* **transição**, *com assinatura*

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}((Q \times \Gamma \cup \{\theta\}))\,;$$

- $q_0 \in Q$ *é o* **estado inicial**;
- $F \subseteq Q$, *é o conjunto de* **estados finais** *de* **M**.

Definição 9 *Seja $\mathcal{C}$ um conjunto de autómatos recursivos. Para cada autómato $M \in \mathcal{C}$ supomos definida uma função $C_M : \Gamma_M \backslash \{\theta_M\} \rightarrow \mathcal{C}$.*

Sejam $w \in \Sigma^*$ uma palavra e

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

um AR. A **computação de** w **por** M *é uma sequência finita de* **passos**

$$(q_0, \epsilon, w), \ldots, (q_k, u_k, v_k)$$

em que, no passo $(q_i, u_i, a_i v_i')$, (com $i = 0, \ldots, k-1$ e $u_i, v_i' \in \Sigma^*$, $a_i \in \Sigma$),

$$\begin{aligned} u_i a_i v_i' &= w \\ (q_{i+1}, \sigma) &\in \delta(q_i, a_i) \end{aligned}$$

e

- **aceitação:** *se $q_k \in F$, $u_k = w$ e $v_k = \epsilon$ então a palavra w é aceite.*
- **terminação:** *se $u_i = w$ ou $q_i \in F$ e $\delta(q_i, a_i) = \emptyset$ então a computação termina; Caso contrário,*
- **passo local:** *se $\sigma = \theta$*

$$\begin{cases} u_{i+1} = u_i a_i \\ v_{i+1} = v_i' \end{cases};$$

- **passo externo:** *se $\sigma \neq \theta$, sendo (q', u', v') o ultimo passo da computação da palavra $a_i v_i'$ pelo AR $C_M(\sigma)$ e u' é aceite pelo AR $C_M(\sigma)$,*

$$\begin{cases} u_{i+1} = u_i u' \\ v_{i+1} = v' \end{cases}.$$

3.3.5 Exemplos

Com base na definição formal apresentamos alguns exemplos de AR, e a sua computação.

Exemplo 10 *Seja M um AR, tal que:*

1. $M = (\{q_0, q_1\}, \{a\}, \delta, \{M'\}, q_0, \{q_1\})$
2. $M = C(M')$
3. $\delta(q_0, a) = \{(M', q_1)\}$

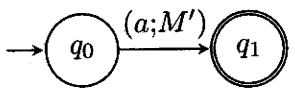

Figura 3.5: Autómato recursivo (M) que reconhece a linguagem $\{a\}$.

Olhando para a definição de M sabemos que $\mathcal{L}(M) \subset a^*$. Pela definição de computação de AR verificamos que M não aceita ϵ pois a computação termina no estado não final q_0. Quanto as restantes palavras aa^* podemos verificar que também são rejeitadas por M, pois o prefixo a é de imediato rejeitado por M por duas razões:

- *não existe nenhum caminho finito em M que aceite aa^*, pois a transição $\delta(q_0, a) = \{(M', q_1)\}$ é circular;*
- *o passo externo não pode ser executado pois M não reconhece nenhuma palavra da linguagem aa^*.*

Exemplo 11 *Seja M um AR, tal que:*

1. $M = (\{q_0, q_1, q_2\}, \{a\}, \delta, \{M'\}, q_0, \{\})$
2. $M = C(M')$
3. $\delta(q_0, a) = \{(M', q_1)\}$
4. $\delta(q_0, a) = \{(a, q_2)\}$

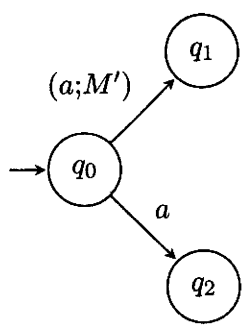

Figura 3.6: Autómato recursivo (M) que reconhece a linguagem $\{a\}$.

Neste caso M não possui estados finais, rejeitando qualquer palavra.

Exemplo 12 *Seja M um AR, tal que:*

1. $M = (\{q_0, q_1, q_2\}, \{a\}, \delta, \{M'\}, q_0, \{q_1, q_2\})$
2. $M = C(M')$
3. $\delta(q_0, a) = \{(M', q_1)\}$
4. $\delta(q_0, a) = \{(a, q_2)\}$

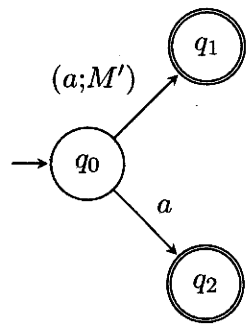

Figura 3.7: Autómato recursivo (M) que reconhece a linguagem $\{a\}$.

Este exemplo é bastante parecido com o anterior com a diferença que desta vez q_1 é um estado final e assim M aceita a pelo caminho q_0q_1 (passo externo) ou por q_0q_2 (passo local).

Exemplo 13 *Seja M um AR, tal que:*

1. $M = (\{q_0, q_1, q_2, q_3\}, \{a, b, c\}, \delta, q_0, \{q_0, q_3\})$
2. $M = C(M')$
3. $\begin{cases} \delta(q_0, a) = \{q_1\} \\ \delta(q_1, M) = \{q_2\} \\ \delta(q_2, b) = \{q_3\} \end{cases}$;

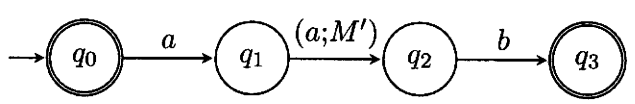

Figura 3.8: Autómato recursivo (M) que reconhece a linguagem a^nb^n.

Neste caso M reconhece a linguagem a^nb^n.

Começamos pela palavra ϵ que é aceite pelo caminho q_0, assim $\epsilon \in \mathcal{L}(M)$ então ab é aceite pelo caminho $q_0q_1q_2q_3$. Portanto temos $\mathcal{L}(M) = \{\epsilon, a\mathcal{L}(M)b\} = \{a^nb^n\}$.

3.3.6 Autómatos Recursivos e Gramáticas Livres de Contexto

Como os exemplos anteriores sugerem, os AR parecem reconhecer linguagens de tipo 2, geradas pelas GLC, ou reconhecidas por autómatos de pilha não-deterministas. Assim pretendemos nesta secção definir um método que nos permita a partir de uma GLC G obter um AR M equivalente ($\mathcal{L}(G) = \mathcal{L}(M)$). Exploramos alguns exemplos que, esperamos, ilustram as ideias principais do método que iremos propor.

Consideramos alguns exemplos de GLC (G) e alguns AR (M) equivalentes:

Exemplo 14 $G = (\{S\}, \{\}, \{S \rightarrow S\}, S)$

- $\mathcal{C}(M') = M$
- $M = (\{q_0, q_1\}, \{\}, \delta, \{M'\}, q_0, \{q_1\})$

$$\delta(q_0, \epsilon) = \{(q_1, M')\}$$

De um modo simples o método funciona fazendo corresponder a cada símbolo não terminal um AR. Neste exemplo o símbolo não terminal S é representado pelo AR M. O próximo passo é simular na função δ caminhos em M equivalentes a cada produção de S. No nosso exemplo temos $S \rightarrow S$ representada pela transição $\delta(q_0, \epsilon) = \{(q_1, M')\}$ e $\mathcal{C}(M') = M$ em que o caminho q_0q_1 aceita palavras de $\mathcal{L}(M) = \mathcal{L}_G(S)$. Visto que S não produz símbolos terminais então M não contém símbolos terminais à esquerda logo ϵ é usado na definição de δ. Como podemos ver o contexto de M (Γ) é um mapeamento quase directo dos símbolos não terminais em G e o alfabeto de entrada de M é exactamente o mesmo que o conjunto de símbolos terminais de G (T).

Por fim o AR equivalente ao símbolo inicial de G reconhece as mesmas linguagens que G, assim $\mathcal{L}(M) = \mathcal{L}_G(S) = \mathcal{L}(G)$.

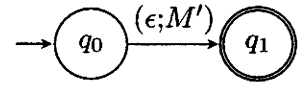

Figura 3.9: AR que reconhece a linguagem $\emptyset$.

Exemplo 15 $G = (\{S\}, \{a\}, \{S \rightarrow S, S \rightarrow a\}, S)$

- $\mathcal{C}(M') = M$
- $M = (\{q_0, q_1, q_2\}, \{a\}, \delta, \{M'\}, q_0, \{q_1, q_2\})$

$$\begin{aligned} \delta(q_0, a) &= \{(q_1, M')\} \\ \delta(q_0, a) &= \{(q_2, \theta)\} \end{aligned}$$

Neste exemplo temos a produção $S \rightarrow a$ como o símbolo a é terminal então definimos a transição $\delta(q_0, a) = \{(q_2, \theta)\}$ usando θ. No exemplo anterior vimos $S \rightarrow S$, em que a transição equivalente fazia uso de ϵ porque S não produzia qualquer palavra, agora S produz a linguagem a portanto definimos a transição equivalente assim $\delta(q_0, a) = \{(q_1, M')\}$.

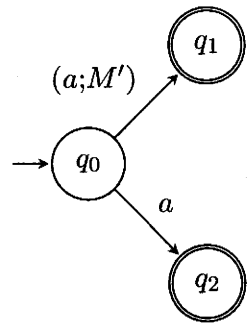

Figura 3.10: AR que reconhece a linguagem $\{a\}$.

Exemplo 16 $G = (\{S\}, \{a, b\}, \{S \rightarrow aS, S \rightarrow Sb, S \rightarrow c\}, S)$

- $\mathcal{C}(M') = M$
- $M = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{a, b, c\}, \delta, \{M'\}, q_0, \{q_1, q_2, q_5\})$

$$\begin{aligned} \delta(q_0, a) &= \{(q_1, \theta)\} \\ \delta(q_1, a) &= \{(q_2, M')\} \\ \delta(q_0, a) &= \{(q_3, M')\} \\ \delta(q_3, b) &= \{(q_4, \theta)\} \\ \delta(q_0, c) &= \{(q_5, \theta)\} \end{aligned}$$

Já vimos como traduzir os alfabetos da GLC para o AR, e vimos também como devemos fazer gerar uma transição de uma produção com um símbolo não terminal e com um símbolo terminal. Vamos agora introduzir algumas regras na geração de caminhos através de produções.

Cada produção de um símbolo não terminal é gerada como um caminho único na máquina correspondente, esse caminho tem inicio no estado inicial e fim num estado final. Consideramos a produção $S \rightarrow w$, o caminho gerado por $w = w_0 w_1 \ldots w_{|w|}$ satisfaz as seguintes regras para todo $i = 0 \ldots |w| - 1$:

1. *Se w_i é terminal então $(q_{i+1}, \theta) \in \delta(q_i, w_i)$*
2. *Senão $(q_{i+1}, c) \in \delta(q_i, a)$, em que $c \in \Gamma$ e corresponde ao símbolo não terminal w_i.*

Aplicando as regras ao nosso exemplo iremos obter as transições em cima, vamos exemplificar passo a passo com a primeira produção $S \rightarrow aS$:

1. *$w = aS$*
2. *para $i = 0, 1$:*
3. *$w_0 = a$ é terminal? Sim, então $(q_1, \theta) \in \delta(q_0, a)$;*
4. *$w_1 = S$ é terminal? Não, então $(q_2, M') \in \delta(q_1, a)$;*
5. *Como todos os caminhos terminam num estado final $q_2 \in F$.*

Continuando o processo para todas as produções de S obtemos o AR M.

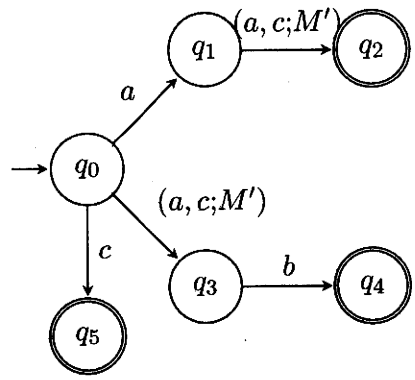

Figura 3.11: AR que reconhece a linguagem a^*cb^*.

Exemplo 17 $G = (\{S, A\}, \{0, 1\}, \{S \rightarrow A, S \rightarrow 01, A \rightarrow 0S1\}, S)$

- $C(A') = A$
- $M = (\{q_0, q_1, q_2, q_3\}, \{a\}, \delta, \{A'\}, q_0, \{q_1, q_3\})$

$$\begin{aligned} \delta(q_0, 0) &= \{(q_1, A')\} \\ \delta(q_0, 0) &= \{(q_2, \theta)\} \\ \delta(q_2, 1) &= \{(q_3, \theta)\} \end{aligned}$$

- $C(A') = A$
- $A = (\{q_0, q_1, q_2, q_3\}, \{a\}, \delta, \{M'\}, q_0, \{q_3\})$

$$\begin{aligned} \delta(q_0, 0) &= \{(q_1, \theta)\} \\ \delta(q_1, 0) &= \{(q_2, M')\} \\ \delta(q_2, 1) &= \{(q_3, \theta)\} \end{aligned}$$

Mais um exemplo, desta vez temos mais que um símbolo não terminal o que significa que temos que criar AR's para cada uma delas. O processo continua o mesmo, apenas temos que fazer corresponder cada produção ao AR correcto.

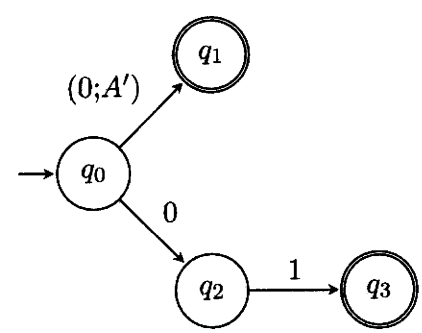

Figura 3.12: AR que reconhece a linguagem 00^*1^*1.

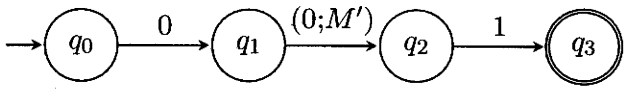

Figura 3.13: AR que reconhece a linguagem 000*1*11.

Vamos agora formalmente definir os passos necessários para construir um AR M a partir de uma GLC G.

Definição 10 *Dada uma gramática $G = (V, T, P, S)$, para cada $v \in V$ definimos*

1. $\texttt{prod}_G(v) = \{p : (v, p) \in P\}$;
2. *o conjunto de todos os primeiros símbolos terminais que aparecem numa derivação à esquerda de um símbolo não terminal, dado por:*

$$\texttt{first}_G(v) = \{a : v \Rightarrow aw \wedge a \in T\}$$

3. $\overline{p} = \left\{\overline{p}_1, \ldots, \overline{p}_{|p|}\right\}$ *para cada* $p \in \texttt{prod}_G(v)$;
4. $A = T \cup V$
5. *para cada* $x \in A$,

$$x^\theta = \begin{cases} (\{x\}, \theta) & se \quad x \in T \\ (\texttt{first}_G(x), x) & se \quad x \in V \end{cases}$$

6. *um NAR* $R_G(v) = (Q_v, \Sigma_v, \Gamma_v, q_{v0}, \delta_v, F_v)$ *em que*
   (a) $Q_v = \{q_{v0}\} \cup \bigcup_{p \in \texttt{prod}_G(v)} \overline{p}$;
   (b) $\Sigma_v = T$;
   (c) $\Gamma_v = \{v' : v \in V \wedge C(v') = v\}$;
   (d) $F_v = \left\{\overline{p}_{|p|} : p \in \texttt{prod}_G(v)\right\} \cup \{q_{v0} : \epsilon \in \texttt{prod}_G(v)\}$;

   e a transição δ_v é definida da seguinte forma:

$$\delta_v = \bigcup_{p \in \texttt{prod}_G(v)} \delta_{vp}$$

   sendo, para cada $p \in \texttt{prod}_G(v)$,

$$\delta_{vp} = \left\{\bigcup_{a \in s} ((q_{v0}, a), \{(\overline{p}_1, c)\}) : (s, c) \in p_1^\theta\right\}$$
$$\cup \left\{\bigcup_{a \in s} ((\overline{p}_i, a), \{(\overline{p}_{i+1}, c)\}) : (s, c) \in p_{i+1}^\theta \wedge 1 \leq i < |p|\right\}$$

Seguindo os passos acima, dada uma GLC G definimos um certo AR M que esperamos que seja equivalente a G.

3.4 Autómatos recursivos deterministas

A definição de autómato recursivo determinista corresponde, intuitivamente, aos autómatos finitos não-deterministas. Nestes, cada configuração (estado, símbolo) define um conjunto de possíveis estados futuros enquanto que, nos autómatos finitos deterministas uma configuração define exactamente um único estado futuro. No caso autómatos recursivos seguimos uma abordagem idêntica.

Definição 11 *Um* **autómato recursivo** *(AR)*

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

é **determinista** *(ARD) se*

- δ *é a função de* **transição**, *com assinatura*

$$\delta : Q \times \Sigma \rightarrow Q \times \Gamma \cup \{\theta\}$$

Dado que os ARD são um caso particular dos ARN, a noção de computação antes definida, assim como todas as noções associadas (palavra aceite, passo local, etc) podem ser re-aplicadas, sem qualquer alteração.

3.4.1 Transformações entre Autómatos Recursivos Deterministas e não-deterministas

Continuando um percurso paralelo, dispomos agora de duas noções de autómato recursivo. A primeira (ARN) permite transições não-deterministas e a segunda (ARD), mais restrita, especifica que cada configuração (estado, símbolo) define um único estado seguinte. Tal como acontece no estudo dos AFD e AFN, estamos agora interessados em explorar a relação entre ARN e ARD. Nomeadamente:

1. qualquer linguagem $\mathcal{L}$ reconhecida por um AR determinista também é reconhecida por algum AR não determinista?

2. qualquer linguagem $\mathcal{L}$ reconhecida por um AR não determinista também é reconhecida por algum AR determinista?

A primeira pergunta é simples de responder, pois os ARD são, por definição sub-casos dos ARN.

A segunda questão é um pouco mais complicada e muito mais interessante.

Nas secções seguintes não vamos responder a essa questão mas apenas provar que a classe de linguagens reconhecidas por um ARD:

1. contém todas as linguagens regulares;

2. está contida na classe das linguagens livres de contexto.

Iremos também propor um algoritmo que em certos casos transforma um ARN num ARD de forma a que ambos reconhecem a mesma linguagem. Infelizmente esse algoritmo não pode ser aplicado a qualquer ARN, o que deixa em aberto a questão de saber se estes dois modelos são equivalentes (o que acontece com os autómatos finitos) ou se as computações não-deterministas permitem reconhecer mais linguagens do que as deterministas (como é o caso dos autómatos de pilha).

3.4.2 Linguagens regulares

Vamos então provar a seguinte afirmação:

Qualquer linguagem regular é reconhecida por um ARD.

Ora, sabendo que a classe das linguagens regulares coincide com a classe das linguagens reconhecidas pelos AFD, basta mostrar que, para cada AFD dado, digamos $A = (Q, \Sigma, \delta, q_0, F)$, existe um ARD, $M = (Q, \Sigma, \delta, \Gamma, q_0, F)$ tal que $\mathcal{L}(A) = \mathcal{L}(M)$.

Considerando as duas definições:

- ARD $M = (Q, \Sigma, \delta, \Gamma, q_0, F)$;
  $\delta : Q \times \Sigma \rightarrow Q \times \Gamma \cup \{\theta\}$;
- AFD $A = (Q, \Sigma, \delta, q_0, F)$;
  $\delta : Q \times \Sigma \rightarrow Q$;

Se considerarmos o sub-conjunto D de máquinas ARD com $\Gamma = \{\}$ obtemos:

- ARD $M = (Q, \Sigma, \delta, \{\}, q_0, F)$;
  $\delta : Q \times \Sigma \rightarrow (Q \times \{\theta\})$;

Facilmente provamos que o conjunto D reconhece todas as linguagens regulares, provando que todas as máquinas em D são equivalentes a qualquer AFD.

Considerando apenas o conjunto D, $\Gamma \cup \{\theta\}$ passa a ser constante sem qualquer tipo de funcionalidade podendo ser ignorado, obtendo assim:

- ARD $M = (Q, \Sigma, \delta, q_0, F) \in D$;
  $\delta : Q \times \Sigma \rightarrow Q$;

Que é a definição de AFD. Como todos os ARD do conjunto D não possuem referencias a outras a máquinas a sua computação é sempre feita usando o passo local que equivale à computação feita por um AFD, assim podemos concluir que os ARD reconhecem pelo menos as mesmas linguagens que um AFD.

Já provámos que a classe das linguagens reconhecidas por um AFD está contida na classe das linguagens reconhecida por ARD. Agora vamos ver que esta inclusão é própria, isto é, existe uma linguagem reconhecida por um ARD que não é reconhecida por nenhum AFD. Sabemos que a linguagem $a^n b^n$ não é reconhecida por nenhum AFD. Demonstramos que esta linguagem é reconhecida por pelo menos por um ARD, construindo M que a reconhece:

- $M = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, \{M'\}, q_0, \{q_2\}) \in D$
- $C(M') = M$
  $\delta(q_0, a) = (q_1, M')$
  $\delta(q_1, b) = (q_2, \theta)$

O ARD M reconhece a linguagem $a^n b^n$. Vejamos porquê. Começamos pela palavra ϵ que é aceite pelo caminho q_0, logo $\epsilon \in \mathcal{L}(M)$ então ab é aceite pelo caminho $q_0 q_1 q_2 q_3$. Portanto temos $\mathcal{L}(M) = \{\epsilon\} \cup a\mathcal{L}(M)b\} = \{a^n b^n\}$.

Assim demonstramos que os ARD reconhecem mais linguagens que os AFD.

3.4.3 ARN para ARD

Sabemos que o conjunto de linguagens regulares está contido nas linguagens que os ARD reconhecem [3.4.2], contudo, desconhecemos qual o conjunto de linguagens que este reconhece.

Acreditamos que a classe das linguagens reconhecidas pelos AR deterministas é um subconjunto próprio das linguagens reconhecidas pelos AR não-deterministas. Isto é, estamos convencidos que existe pelo menos um AR não-determinista que reconhece uma linguagem que nenhum AR determinista reconhece. Porém, não conseguimos produzir uma prova desta conjectura. No entanto, temos a seguinte indicação da sua validade: Se tentarmos simular um AR não-determinista por um AR determinista, em certos casos não somos capazes.

Ainda assim temos um método, que funciona em certos casos, que permite converter um AR não-determinista num AR determinista equivalente. De seguida apresentamos esse método, precedido de alguns exemplos.

Exemplo 18 $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b, c\}, \delta_M, \{\}, q_0, \{q_3\})$

$$\begin{aligned} \delta_M(q_0, a) &= \{(q_1, \theta), (q_4, \theta)\} \\ \delta_M(q_1, b) &= \{(q_2, \theta)\} \\ \delta_M(q_2, c) &= \{(q_3, \theta)\} \\ \delta_M(q_4, c) &= \{(q_3, \theta)\} \end{aligned}$$

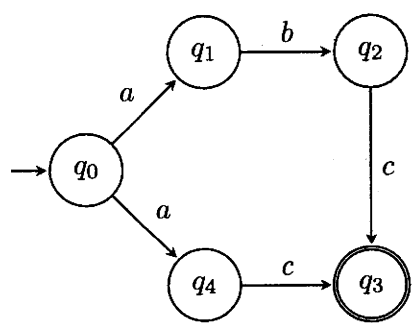

Figura 3.14: ARN que reconhece a linguagem $\{abc, ac\}$.

$N = (\{q_0, q_{1,4}, q_2, q_3\}, \{a, b, c\}, \delta_N, \{\}, q_0, \{q_3\})$

$$\begin{aligned} \delta_N(q_0, a) &= (q_{1,4}, \theta) \\ \delta_N(q_{1,4}, b) &= (q_2, \theta) \\ \delta_N(q_{1,4}, c) &= (q_3, \theta) \\ \delta_N(q_2, c) &= (q_3, \theta) \end{aligned}$$

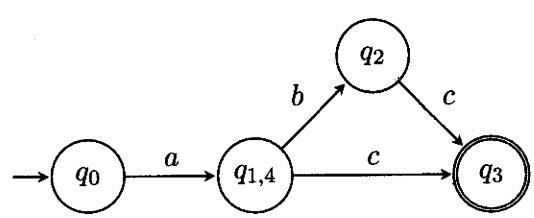

Figura 3.15: ARD que reconhece a linguagem $\{abc, ac\}$.

Exemplo 19 $M = (\{q_0, q_1\}, \{a\}, \delta_M, \{M'\}, q_0, \{q_1\})$
$\delta_M(q_0, a) = \{(q_1, \theta), (q_1, M')\}$

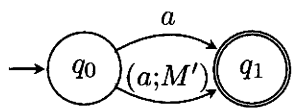

Figura 3.16: ARN que reconhece a linguagem $\{a\}$.

$N = (\{q_0, q_1\}, \{a\}, \delta_N, \{\}, q_0, \{q_1\})$
$\delta_M(q_0, a) = \{(q_1, \theta)\}$

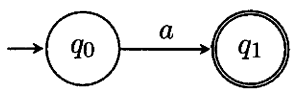

Figura 3.17: ARN que reconhece a linguagem $\{a\}$.

Os casos em que o não-determinismo pode ocorrer são descritos a seguir, juntamente com uma descrição de como esse não-determinismo pode ser evitado. Assim, assumindo que temos transições com o mesmo símbolo, o mesmo estado de partida e:

1. vários estados de chegada diferentes em que todas as referencias a máquinas são iguais. Este é um simples caso de indeterminismo. Podemos adoptar a mesma estratégia dos autómatos finitos; juntando todos os estados de chegada em apenas um, copiando as suas transições e aplicando as regras aqui descritas, de modo a criar um novo estado apenas com transições deterministas.

   Exemplo 20 $N = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b\}, \delta_N, \Gamma, q_0, \{q_2, q_3, q_4\})$
   $\delta_N(q_0, a) = \{(q_1, m), (q_3, m)\}$
   $\delta_N(q_1, b) = \{(q_2, n)\}$
   $\delta_N(q_3, b) = \{(q_4, n)\}$

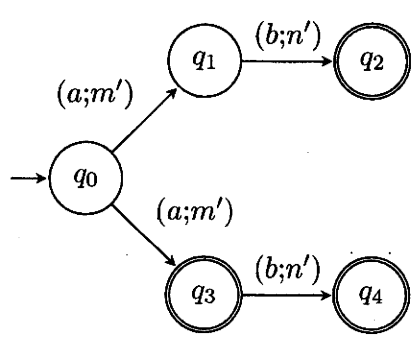

Figura 3.18: ARN com transições iguais estados de chegada diferentes.

A máquina equivalente determinista é definida por:
$N' = (\{q_0, q_{1,3}, q_{2,4}\}, \{a, b\}, \delta'_N, \Gamma, q_0, \{q_{2,4}, q_{1,3}\})$

$$\begin{aligned}\delta'_N(q_0, a) &= (q_{1,3}, m)\\ \delta'_N(q_{1,3}, b) &= (q_{2,4}, n)\end{aligned}$$

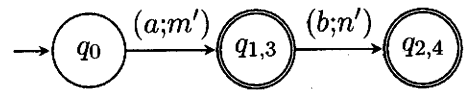

Figura 3.19: ARD equivalente ao ARN.

2. transições ϵ sem referencias a máquinas. Neste caso copiamos todas as transições do estado de chegada para o estado de partida e eliminamos a transição ϵ. Como todas as transições ϵ sem referencias a máquinas são eliminadas deixam de existir também transições ϵ com referencias a máquinas.

Exemplo 21 $N = (\{q_0, q_1, q_2\}, \{a, b\}, \delta_N, \Gamma, q_0, \{q_2\})$

$$\begin{aligned}\delta_N(q_0, \epsilon) &= \{(q_1, \theta)\}\\ \delta_N(q_0, b) &= \{(q_1, \theta)\}\\ \delta_N(q_1, a) &= \{(q_2, \theta)\}\end{aligned}$$

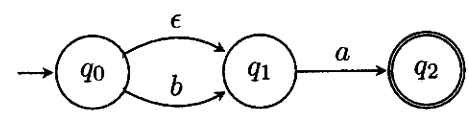

Figura 3.20: ARN com transições ϵ.

A máquina equivalente determinista é definida por:

$N' = (\{q_0, q_1, q_2\}, \{a, b\}, \delta_N, \Gamma, q_0, \{q_2\})$

$$\begin{aligned}\delta'_N(q_0, a) &= (q_2, \theta)\\ \delta'_N(q_0, b) &= (q_1, \theta)\\ \delta'_N(q_1, a) &= (q_2, \theta)\end{aligned}$$

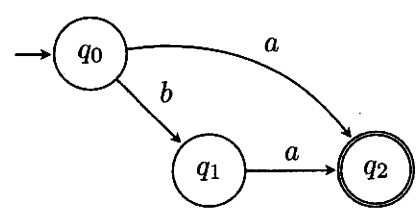

Figura 3.21: ARN com transições ϵ.

3. recursividade à esquerda (transições recursivas com partida no estado inicial). Os ARD não suportam recursividade à esquerda, porque ao se autoreferenciarem no estado inicial terão sempre que definir transições indeterministas. Para simular recursividade à esquerda optamos por juntar o estado de chegada da transição recursiva a todos os estados finais da máquina.

Exemplo 22 $N = (\{q_0, q_1, q_2\}, \{a, b\}, \delta_N, \Gamma, q_0, \{q_2\})$

$C(N) = n$

$$\begin{aligned} \delta_N(q_0, a) &= \{(q_1, n), (q_2, \theta)\} \\ \delta_N(q_1, b) &= \{(q_2, \theta)\} \end{aligned}$$

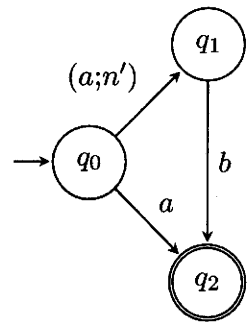

Figura 3.22: ARN recursividade à esquerda.

A máquina equivalente determinista é definida por:

$N' = (\{q_0, q_{1,2}\}, \{a, b\}, \delta_N, \Gamma, q_0, \{q_{1,2}\})$

$C(N') = n'$

$$\begin{aligned} \delta'_N(q_0, a) &= (q_2, \theta) \\ \delta'_N(q_{1,2}, b) &= (q_{1,2}, \theta) \end{aligned}$$

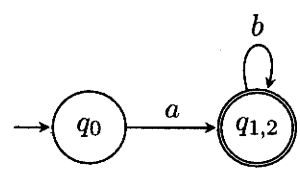

Figura 3.23: ARD resolução de recursividade à esquerda.

4. referencias a máquinas diferentes. Este é um caso específico dos AR, em que a solução será «estender» todas as máquinas diferentes desta transição até que o indeterminismo fique resolvido. Neste caso estender significa substituir as transições em conflito por uma porção, a partir do estado inicial, da máquina referenciada. Como esta operação copia parte da máquina referenciada é necessário criar novas máquinas que contenham as porções que não foram copiadas de modo a que sejam referenciadas nos estados apropriados.

Exemplo 23 $N = (\{q_0, q_1, q_2\}, \{a\}, \delta_N, \{n_1, n_2\}, q_0, \{q_1, q_2\})$

$C(n_1) = N_1$ *e* $C(n_2) = N_2$

$$\begin{aligned} \delta_N(q_0, a) &= \{(q_1, n_1)\} \\ \delta_N(q_0, a) &= \{(q_2, n_2)\} \end{aligned}$$

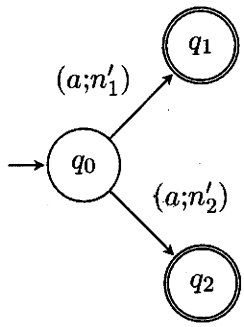

Figura 3.24: ARN, transições apenas com símbolos iguais.

$N_1 = (\{q_0, q_1, q_2\}, \{a, b\}, \delta_{N1}, \{\}, q_0, \{q_2\})$

$$\begin{aligned} \delta_{N1}(q_0, a) &= \{(q_1, \theta)\} \\ \delta_{N1}(q_1, b) &= \{(q_2, \theta)\} \end{aligned}$$

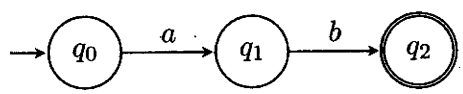

Figura 3.25: ARD que reconhece a linguagem $\{ab\}$.

$N_2 = (\{q_0, q_1, q_2\}, \{a, c\}, \delta_{N2}, \{\}, q_0, \{q_2\})$

$$\begin{aligned} \delta_{N2}(q_0, a) &= \{(q_1, \theta)\} \\ \delta_{N2}(q_1, c) &= \{(q_2, \theta)\} \end{aligned}$$

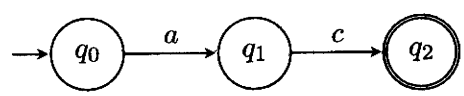

Figura 3.26: ARD que reconhece a linguagem $\{ac\}$.

A máquina equivalente determinista é definida por:

$N' = (\{q_0, q_1, q_2, q_3\}, \{a, b, c\}, \delta'_N, \{n'_1, n'_2\}, q_0, \{q_1, q_2\})$

$C(n_1) = N'_1$ *e* $C(n_2) = N'_2$

$$\begin{aligned} \delta'_N(q_0, a) &= (q_3, \theta) \\ \delta'_N(q_3, b) &= (q_1, n'_1) \\ \delta'_N(q_3, c) &= (q_2, n'_2) \end{aligned}$$

$N_1 = (\{q_0, q_1, q_2\}, \{a, b\}, \delta_{N1}, \{\}, q_1, \{q_2\})$

$N_2 = (\{q_0, q_1, q_2\}, \{a, c\}, \delta_{N2}, \{\}, q_1, \{q_2\})$

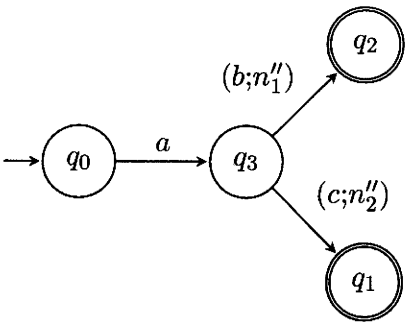

Figura 3.27: ARD N', que reconhece a linguagem $\{ab, ac\}$.

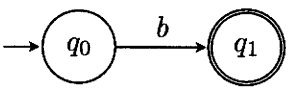

Figura 3.28: ARD, N_1' que reconhece a linguagem $\{b\}$.

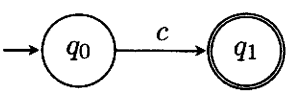

Figura 3.29: ARD, N_2' que reconhece a linguagem $\{c\}$.

5. estados de chegada (retorno) em conflito com estados finais da máquina referenciada.

Um AR determinista apenas pode percorrer um caminho, caso o caminho escolhido falhe, toda a computação falha e a palavra é rejeitada. Por outro lado a computação de um AR não-determinista apenas falha se todos os caminhos falham, apenas nesse caso uma palavra é rejeita. No caso das chamadas a outras máquinas, nos estados finais da máquina referenciada o ARN pode retornar ou continuar nessa mesma máquina. Ora como o ARD apenas tem a possibilidade de continuar na máquina chamada o comportamento do ARN nestes casos terá que ser simulado. Tentaremos explicar melhor estes conceitos com um exemplo. Seja M um AR determinista e N um AR não-determinista. Supondo que M reconhece a linguagem $\mathcal{L}_M = \{v, vw\}$ e que N reconhece a linguagem definida por $\mathcal{L}_M w$, ou seja $\mathcal{L}_N = \{vw, vww\}$. Como a computação do ARD apenas pode tomar um caminho então assim a única linguagem aceite por N seria $\{vww\}$, isto porque após reconhecermos v estamos na máquina M e qualquer w é reconhecido na máquina M. Isto significa que iremos retornar para N, onde se segue de novo outro w. No caso do ARN em que este pode retornar da máquina M no primeiro símbolo v a máquina N aceitará também a palavra vw. Assim fica claro que temos que ter alguma maneira de contornar este problema. Uma das soluções seria «estender» a máquina M em N até aos estados de retorno que se encontram em conflito.

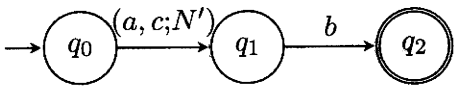

Figura 3.30: ARN M, estados de retorno com transições em conflito.

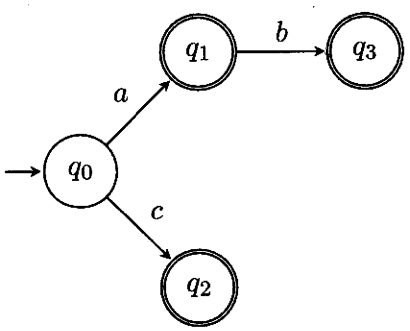

Figura 3.31: ARN/ARD N.

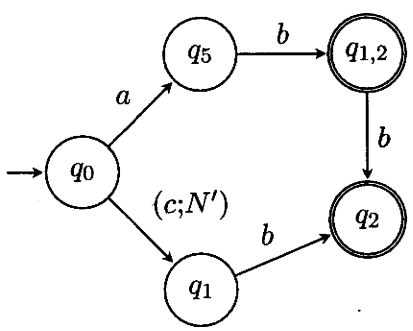

Figura 3.32: ARD M

Depois de termos explorado os principais casos de não-determinismo, e de termos visto como podem ser simulados numa computação determinista, apresentamos um algoritmo para converter alguns ARN para o seu equivalente determinista (ARD).

Dado um ARN N definimos o seu equivalente determinista M, onde iremos usar a notação $Det(N) = M$ para exprimir que M foi gerado com base em N pelos seguintes passos:

1. $N = (Q_N, \Sigma_N, \delta_N, \Gamma_N, q_{N0}, F_N)$;
2. $M = (Q_M, \Sigma_M, \delta_M, \Gamma_M, q_{M0}, F_M)$;
3. $\Sigma_N \subseteq \Sigma_M$;
4. $q_{M0} = q_{N0}$;
5. $Q_N \subseteq Q_M$;
6. $F_N \subseteq F_M$;

7. Definimos $r : Q_M \rightarrow \mathcal{P}(Q_M)$ que vamos usar para identificar cada estado no algoritmo;

8. $\forall q(q \in Q_N \wedge r(q) = \{q\})$;

9. Definimos também $C_{det} : \Gamma_M \cup \Gamma_N \cup \{\theta\} \rightarrow \Gamma_M$

$$C_{det}(x) \begin{cases} x & se \quad x \in \Gamma_M \vee x = \theta \\ D & se \quad x \in \Gamma_N \wedge \mathcal{C}(D) = Det(\mathcal{C}(X)) \end{cases} ;$$

10. Os restantes estados e transições de M são definidos pelo método t. Para cada transição $(q', C) \in \delta_N(q, s)$ invocamos $t(N, M, q, s, C_{det}(C), q')$.

O método $t(N, M, p, s, C, c)$ é definido pelos seguintes passos:

1. $M = (Q_M, \Sigma_M, \delta_M, \Gamma, q_{M0}, F_M)$

2. $s \in \Sigma_M$

3. Se $s = \epsilon \wedge C = \theta$ então

   $\forall s'((\delta_M(c, s') = (c', C')) \rightarrow t(N, M, p, s', C', c'))$;

   termina.

4. Se $p = q_{M0} \wedge M = \mathcal{C}(C)$ então

   $\forall q((q \in F_M) \rightarrow t(M, q, \epsilon, \theta, c))$;

   termina.

5. Se $\delta_M(p, s)$ não existe então $\delta_M(p, s) = (c, C)$, invocamos $ret(M, p)$, $ret(M, c)$ e termina.

6. Senão redefinir $\delta_M(p, s) = (q, A)$ como $\delta_M(p, s) = (q', A')$ em que:

   (a) Se $A = C \wedge c \notin r(q)$ então

   $A' = C = A$;

   $r(q') = r(q) \cup r(c)$;

   $q' \in Q_M$;

   Se $q \in F_M \vee c \in F_M$ então $q' \in F_M$;

   $\forall i(i \in r(q') \wedge i \in Q_N \wedge \forall s'(\delta_N(i, s') = (c', C') \rightarrow t(N, M, q', s', C', c')))$

   $ret(N, M, p)$;

   $ret(N, M, q')$;

   termina.

   (b) Se $A \neq C \wedge C = \theta$ então

   $q' = c$;

   $A' = C$;

   $t(N, M, q, s, A, q')$;

   termina.

(c) Senão

$C = (Q, \Sigma, \delta_C, \Gamma, q_{C0}, F)$

$\delta_C(q_{C0}, s) = (q_c, C')$;

definimos a máquina $Cn = (Q, \Sigma, \Gamma, \delta_C, q_c, F)$;

$C_{det}(Cn) \in \Gamma_M$

$\delta_M(q', x) = (c, C_{det}(Cn))$, em que x é algum simbolo das transições iniciais de Cn;

$ret(N, M, q')$;

$ret(N, M, c)$;

$\delta_M(p, s) = (q', C')$;

$ret(N, M, p)$;

$ret(N, M, q')$;

$t(N, M, p, s, A, q)$;

termina.

O método que iremos descrever, $ret(N, M, c)$ onde N é um ARN, M é um ARD e c um estado de M, verifica e resolve o caso em que existem estados de chegada (retorno) em conflito com estados finais da máquina referenciada.

O método $ret(N, M, c)$ é definido pelos seguintes passos:

1. Para cada transição $\delta_M(p, s) = (c, C)$ em que $C \neq \theta$:

   Seja $\mathcal{C}(C) = C' = (Q_{C'}, \Sigma_{C'}, \delta_{C'}, \Gamma, q_{C'0}, F_{C'})$ e para cada $f \in F_{C'}$ se existe $\delta_{C'}(f, a) = (q, A)$ e $\delta_M(c, a) = (c', B)$ então:

   (a) definimos a máquina $Cn = (Q_{C'}, \Sigma_{C'}, \Gamma_{C'}, \delta_{C'}, f, F_{C'})$;

   (b) $C_{det}(Cn) \in \Gamma_M$

   (c) $t(N, M, c, x, C_{det}(Cn), c)$, em que x é algum símbolo das transições iniciais de Cn;

   (d) Para cada q_i, q_j onde existe um caminho de q_i, q_j até f, definimos dois novos estados q_i', q_j' identificados por:

   $r(q_i') = r(q_i) \cup \{q_i'\}$;

   $r(q_j') = r(q_j) \cup \{q_j'\}$;

   e aplicamos os seguintes passos a cada transição $\delta_{C'}(q_i, x) = (q_j, X)$:

   i. $t(N, M, q_i', x, X, q_j')$;

   ii. Se $q_i = q_{C'0}$ então $t(M, c, \epsilon, \theta, q_i')$;

   iii. Se $q_j = f$ então $t(M, c, \epsilon, \theta, q_j')$;

3.4.4 Conclusões

Neste capítulo apresentámos uma definição formal de autómato recursivo não-determinista e determinista. Exploramos também a relação dos ARN com gramáticas livres de contexto, propusemos um algoritmo para converter uma GLC para o ARN equivalente. Deixámos para trabalho futuro a identificação das linguagens reconhecidas por um ARN e a validação do algoritmo da construção de um ARN através de uma de uma GLC. Propusemos também um método para converter um AR indeterminista

para o seu equivalente determinista. Acreditamos que nem todos os AR indeterministas tem um equivalente determinista, mas o contrario é verdade, o que revela que o primeiro reconhece potencialmente mais linguagens. Vimos também que os AR deterministas reconhecem pelo menos todas as linguagens regulares, o que por acréscimo todos os AR indeterministas reconhecem também essas mesmas linguagens.

Capítulo 4

Aplicações

Como referido anteriormente, a teoria dos autómatos tem, na informática e noutras áreas, imensas aplicações. Na linha de trabalho que nos motivou a considerar a possibilidade teórica dos AR, vamos agora explorar um meio de associar, a cada computação de um AR, informação mais rica de que uma simples «palavra aceite» ou «palavra rejeitada». O mecanismo que vamos empregar é conhecido como «metadados», de que falaremos de seguida. Depois vamos também estudar «acções» como um caso particular dos metadados.

4.1 Metadados e Acções

4.1.1 Introdução

No inicio deste trabalho referimos várias características dos autómatos, e como estes são usados na pratica. Através da analise de implementações já existentes neste ramo e das necessidades dos programas actuais, acreditamos que a capacidade de associar e extrair dados independentes ou dependentes de contexto num autómato aumenta a sua versatilidade e utilidade. Assim, nesta secção, é sugerido um método para alcançar esse objectivo para os autómatos recursivos indeterministas e deterministas, sendo estes últimos os mais abordados, pois é aqui que se encontra o maior grau de dificuldade, mas também porque são eles a base do trabalho e da implementação.

4.1.2 Metadados

Em teoria da informação, o termo metadados é descrito como «dados sobre outros dados», com vista a facilitar o entendimento das relações e a sua utilidade. No contexto deste trabalho designam qualquer tipo de informação extra associada às relações ou estados do autómato, e que fornecem um melhor entendimento do mesmo, tanto seja por um individuo como por um processo automatizado. São por esta razão de natureza abstracta. Assim, o seu conteúdo não é importante, mas apenas a forma de como estes são associados e mantidos. Podemos pensar nestes como notas no nosso autómato.

O nosso objectivo na implementação de metadados é que possamos através de uma palavra aceite extrair uma lista de metadados que está directamente relacionado com a palavra. A interpretação da lista de metadados extraída e a relação com a palavra aceite é deixada ao cargo de quem define os metadados.

Os metadados não fazem parte da definição de AR, portanto a associação de metadados a um AR é definida por uma função, para facilitar a definição desta função optamos por apenas permitir associar um metadado por estado ou transição. Como os AR são independentes da função de metadados então todas as operações sobre um AR continuam validas sem qualquer tipo de alteração. Porem ao associarmos metadados a um AR M e ao aplicarmos uma operação a M, obtemos um novo AR N que não possui metadados, neste caso podemos definir novos metadados para N através de um processo de migração dos metadados de M.

Onde propomos o seguinte método de migração de metadados:

1. Se um estado s de N resulta de vários estados $e_1, \ldots e_n$ de M onde em que para cada e_i está associado um metadado m_i então o metadado m' representa a união de todos os m_i e está associado a s.

2. Se uma transição t de N resulta de varias transições $e_1, \ldots e_n$ de M onde em que para cada e_i está associado um metadado m_i então o metadado m' representa a união de todos os m_i e está associado a t.

Como associar os metadados? Para clarificar esta ideia, imaginemos a linguagem que contém algumas palavras do português. Esta linguagem é reconhecida por um autómato mas este não nos diz muito sobre cada palavra, como por exemplo se estão no «singular» ou no «plural». Para tal podemos usar os metadados para rotular cada palavra. De imediato, temos de saber responder à seguinte questão: vamos considerar duas possibilidades que ocorrem num primeiro pensamento, os metadados estão associados aos estados do autómato ou em alternativa, estão associados às transições? Para o nosso exemplo de análise associamos três metadados «singular», «plural», e «nenhum», primeiro aos estados de um autómato e depois às transições.

Análise do caso «associar metadados apenas a estados» Como primeira abordagem escolhemos associar «singular» e «plural» a estados finais, e «nenhum» a qualquer outro estado, e a todas as relações. Todas as palavras em plural acabam em estados finais rotulados como «plural» e todas as outras em estados rotulados como «singular».

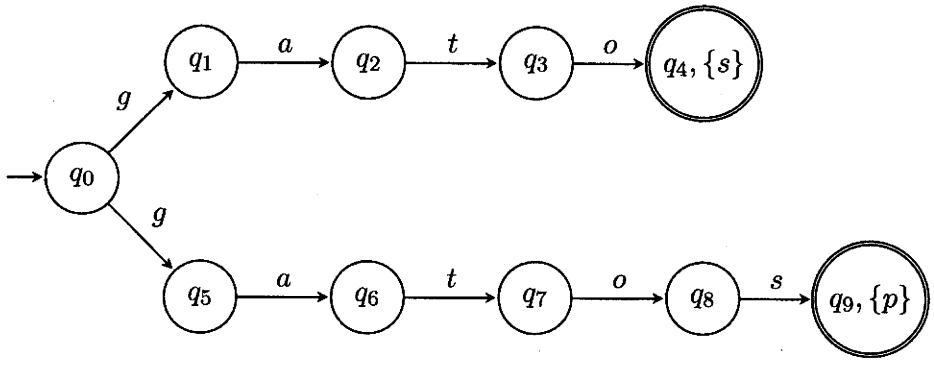

Figura 4.1: Palavras gato e gatos, com associação de metadados s=«singular» e p=«plural» a estados.

O problema com esta abordagem, resulta de termos de garantir que as palavras acabam num estado «plural» ou «singular», e como os autómatos não tem qualquer conhecimento sobre os metadados, isto torna-se um problema, pois existem autómatos equivalentes onde não nos é possivel usar o estado final para classificar cada palavra como «singular» ou «plural», um dos exemplos seria todas as palavras acabarem num único estado final. Outra solução seria associar os metadados a estados únicos de cada palavra, mas nem todos os autómatos possuem estados exclusivos para cada palavra que aceitam, logo esta não é uma solução viável para este tipo de problema. Por fim, se usarmos o algoritmo de determinismo de um AR com o algoritmo de migração de metadados é possível que este afecte o significado de metadados neste tipo de problemas, tornando-se assim inútil.

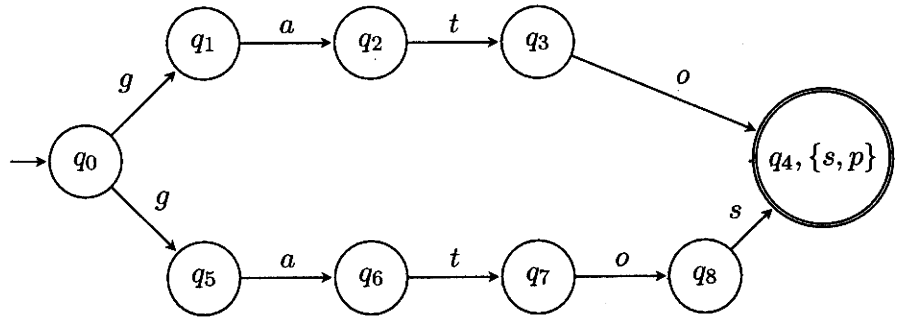

Figura 4.2: Palavras gato e gatos com associação de metadados s=«singular» e p=«plural» a estados.

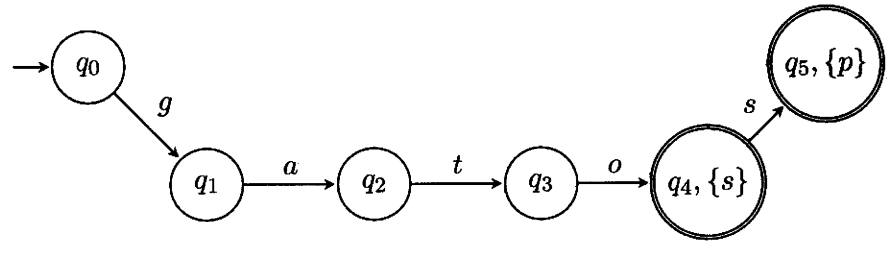

Figura 4.3: Palavras gato e gatos com associação de metadados s=«singular» e p=«plural» a estados.

Análise do caso «associar metadados apenas as transições» Como segunda hipótese podemos associar os metadados «plural» e «singular» apenas às relações de entrada dos estados finais, ou seja, à última transição da palavra, com os respectivos metadados.

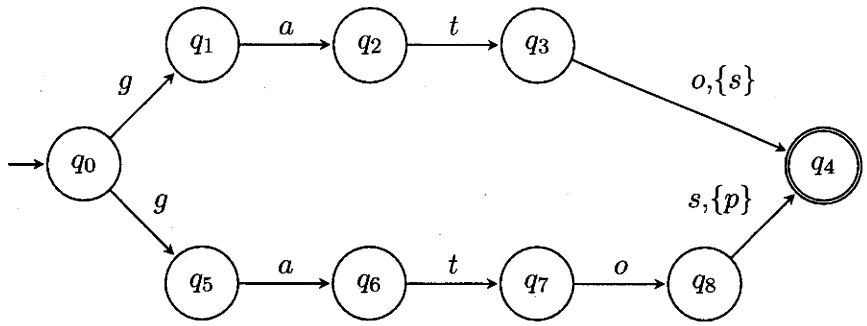

Figura 4.4: Palavras gato e gatos com associação de metadados s=«singular» e p=«plural» a relações.

Isto também não é solução. Mais uma vez nada nos garante que cada transição está associada a palavras apenas no plural ou singular. Por exemplo, todas as palavras terminadas em «s» que podem estar no plural ou no singular (como exemplo: filhós, no singular, e filhoses, no plural).

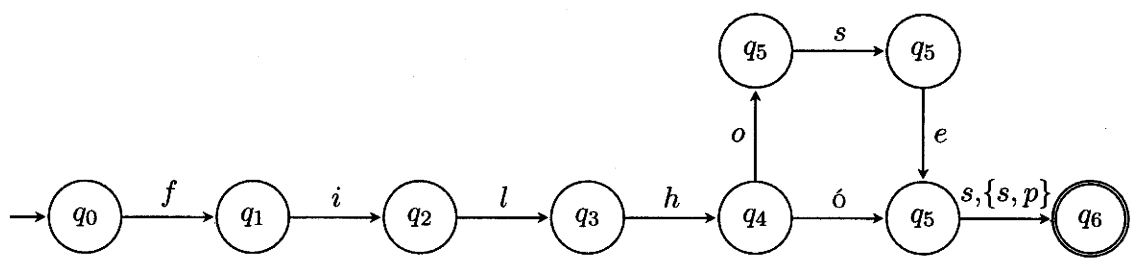

Figura 4.5: Palavras filhós e filhoses com associação de metadados s=«singular» e p=«plural» a relações.

Antes de partirmos para as conclusões vejamos um dos exemplos de uma das utilizações de metadados.

Exemplo 24 *Queremos definir alguns conjuntos de números inteiros, sendo* $a = \{1,2,3\}$, $b = \{1,2,3\}$ *e* $c = \{1,4\}$.

A linguagem que aceita todos os subconjuntos de a, b e c pode ser definida pela

seguinte GLC $G = (\{conj, a, b, c\}, \{1, 2, 3, 4\}, P, conj)$

$$P = \begin{cases} conj & \rightarrow & a \\ conj & \rightarrow & b \\ conj & \rightarrow & c \\ a & \rightarrow & 1a \\ a & \rightarrow & 2a \\ a & \rightarrow & 3a \\ a & \rightarrow & \epsilon \\ b & \rightarrow & 1b \\ b & \rightarrow & 2b \\ b & \rightarrow & 3b \\ b & \rightarrow & \epsilon \\ c & \rightarrow & 1c \\ c & \rightarrow & 4c \\ c & \rightarrow & \epsilon \end{cases};$$

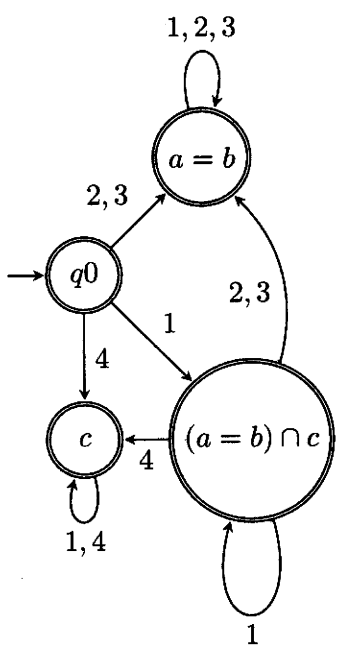

Figura 4.6: Conjuntos $a = b = \{1, 2, 3\}$ e $c = \{1, 4\}$

Definimos o conjunto $C' = \{a, b, c\}$, *e os metadados* $M = \{x : x \subset C'\}$, *e* S *é o autómato que reconhece a linguagem* $\{1, 2, 3\}^* \cup \{1, 4\}^*$. *A todos os estados de* S *com relações de entrada com símbolo* s, *associamos o metadado* $x \in M : e \in x \wedge s \in e$

Para qualquer subconjunto aceite por S, *acedendo aos metadados associados ao estado final que o aceita podemos extrair quais os conjuntos originais a que este pertence. Como este problema esta modelado na forma de um autómato, um conjunto* $\{a_0, a_1, \ldots, a_i\}$ *equivale à palavra* $a_0 a_1 \ldots a_i$, *por exemplo o conjunto* $\{1, 2, 3, 3\}$ *equivale à palavra* 1233. *A palavra* 1233 *é aceite pelo autómato no estado* $a = b$ *o que significa que* 1233 *é um subconjunto de* $a = b = \{1, 2, 3\}$. *Se tentarmos a palavra* 1

iremos ter a resposta que o conjunto $\{1\}$ *é subconjunto de a, b e c. Por ultimo palavras que não formam subconjuntos de a, b e c são rejeitadas, por exemplo* $\{1, 2, 3, 4\}$ *não é um subconjunto de a, b ou c e a palavra* 1234 *é rejeitada.*

Neste exemplo qualquer manipulação ao autómato não afecta o significado dos metadados, pois estados equivalentes e relações equivalentes apenas implicam elementos e subconjuntos iguais.

Concluindo, estados e relações equivalentes devem conter metadados equivalentes. Todos os aspectos de como os metadados são definidos, interpretados, e associados ou não a estados e relações, assim como propriedades de equivalência, são deixados ao cargo do programador.

No caso do nosso primeiro exemplo, foram dadas duas abordagens em que ambas não são solução, isto porque tal problema requer um contexto. Uma das soluções que nos parece mais adequada, e que é usada na secção seguinte para definir um tipo especifico de metadados, chamado acções, baseia-se em definir metadados no contexto das palavras aceites.

Portanto, para que se possa determinar o contexto dos metadados em AR, fica claro que tem que existir informação adicional. O problema está onde deverá ficar essa informação, se associar-mos dados específicos aos autómatos sobre o seu contexto e metadados estaríamos a ir contra a sua definição teórica, logo está não é a solução. Se adicionarmos informação especifica do contexto, fora ou dentro dos metadados, deixamos de ter a componente abstracta, que nos dá uma maior capacidade de adaptação as necessidades de cada implementação, pois nem todos os problemas necessitam de metadados, ou de metadados dependentes de contexto.

Na secção seguinte propomos uma solução para este problema. Podem existir outras soluções e tal como já dissemos cada caso é um caso, e várias optimizações podem ser exploradas.

4.1.3 Acções

São um caso especifico de metadados. Definem-se com um objectivo puramente pratico, pois é natural esperar de um sistema informático que este, através de valores de entrada, não apenas os valide (no caso do autómato os aceite), mas os processe e devolva resultados úteis (Ex. analisadores sintácticos e lexicais).

De um ponto de vista prático, um autómato é implementado ou gerado como um programa. Este é realizado de maneira a controlar e interpretar valores de entrada, a processa-los e a apresentar os valores de saída. É portanto, neste contexto, que inserimos as acções. Estas são meras instruções contidas no programa e que apenas são executadas se determinadas condições forem satisfeitas, ou seja, quando uma palavra é aceite.

Contextualização

Um autómato aceita ou rejeita sequências de símbolos (palavras), as palavras aceites dependendo do contexto podem ser divididas em blocos lógicos. Se considerar-mos a

linguagem natural, uma palavra aceite pode ser dividida em frases, palavras, pontuação, expressões e etc. Se as acções apenas guardam informação relativa a um estado ou transição então estes não possuem contexto, e são sempre executadas. Mas se adicionar-mos informação suficiente para que se possa determinar em quais os caminhos ou palavras em que as acções podem ser executadas então a acção é apenas executada nesse contexto.

É por isso que faz sentido falarmos em associar acções a símbolos (associação de acções a transições sem chamadas a outras máquinas) palavras (associação de acções a caminhos) ou linguagens (associação de acções a transições com chamadas a máquinas).

Exemplo 25 • expressão =

«segunda» dia_semana_segunda «-feira»

/ «segunda» segunda_classe «classe»

/ «quarta» dia_semana_quarta «-feira»

/ «quarta» quarta_classe «classe»

Pelo exemplo, qualquer pessoa de imediato espera certos comportamentos em resultado das acções associadas a cada palavra, isto é, que ao aceitar «segunda-feira» a acção a ser escolhida seja unicamente dia_semana_segunda, e não segunda_classe.

Assim, o objectivo, ao definir as acções é manter a integridade das intenções iniciais, tanto em AR indeterministas, como no determinista.

Na secção de metadados concluímos que, em cada problema, estes podem ser mais úteis associados a relações ou a estados. Vimos também que os estados não nos permitem associar metadados apenas a uma palavra, e que as relações nos permitem fazer isso ao nível dos símbolos e das suas posições na palavra/linguagem, o que as torna de imediato a opção lógica.

É necessário de alguma forma preservar a informação da versão original, e guardá-la nas acções, de modo a determinar em que contexto se inserem, dada uma palavra.

4.1.4 Algoritmo

Existem dois tipos de máquinas nesta abordagem, a que chamaremos as máquinas de símbolos e a máquina de acções. A cada máquina de símbolos corresponde uma de acções. As máquinas de acções, reconhecem linguagens sobre um alfabeto de acções.

Às máquinas de símbolos associamos a cada transição metadados definidos como conjuntos de referencias a transições na máquina de acções correspondente.

Sempre que uma palavra é aceite na máquina de símbolos gera uma lista de metadados. Esses metadados contêm informação sobre o AR de resultados, em que um caminho é apenas válido se terminar num estado final.

Na prática, o algoritmo funciona da seguinte forma: seja A um conjunto de acções e $M = (Q, \Sigma, \delta, \Gamma, q_0, F)$ um AR, e $M' = (Q', A, \delta', \Gamma, q_0, F)$ o AR de acções associado a M. Para cada transição:

1. $m : (Q, \Sigma, Q, \Gamma) \rightarrow \mathcal{P}((Q, A, Q, \Gamma))$, é a função que faz correspondência entre as transições da máquina de símbolos para a máquina de acções.

2. $\Phi \in A$, em que Φ corresponde à acção nula.

3. $((q', \theta) \in \delta(q'', s)) \rightarrow (((q', \theta) \in \delta'(q'', a)) \wedge (m(q'', s, q', \theta) = \{(q'', a, q', \theta)\}))$, ou seja associamos a acção a ao símbolo s, desta transição.

4. $(N \neq \theta) \wedge ((q', N) \in \delta(q, s)) \rightarrow ((q'', N') \in \delta'(q, x)) \wedge ((q', \theta) \in \delta'(q'', a)) \wedge (q'' \notin Q) \wedge (m(q, s, q', N) = \{(q, \Phi, q'', N'), (q'', a, q', \theta)\})$

A cada transição do autómato de símbolos associamos os metadados que contêm a informação que faz correspondência a varias transições no autómato de acções.

Ambos os autómatos podem ser manipulados normalmente e convertidos para deterministas, o que implica actualização da informação contida nos metadados. Sempre que uma ou mais transições sejam unificadas, também os metadados terão que ser unificados. Neste algoritmo optámos por definir os metadados como conjuntos de referencias a transições do AR de acções e usar a união de conjuntos para a operação de unificação de metadados.

Concluindo, dada uma palavra aceite obtemos uma sequência de metadados que nos permite determinar um AR de acções, a linguagem do AR obtido define a sequência de todas as acções a serem executadas.

Exemplo 26 *Supomos que temos a linguagem* $\{ab, abc\}$ *e queremos associar a acção* AB *à palavra* ab *e* ABC *a* abc. *Começamos por construir o ARN que reconhece* $\{ab, abc\}$.

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b, c\}, \delta, \{\}, q_0, \{q_2\})$$

$$\begin{aligned} \delta(q_0, a) &= \{(q_1, \theta)\} \\ \delta(q_1, b) &= \{(q_2, \theta)\} \\ \delta(q_0, a) &= \{(q_3, \theta)\} \\ \delta(q_3, b) &= \{(q_4, \theta)\} \\ \delta(q_4, c) &= \{(q_2, \theta)\} \end{aligned}$$

Defini-mos a função m_M *para associar as acções a cada transição de* M:

$$\begin{aligned} m_M(q_0, a, q_1, \theta) &= \{(q_0, \Phi, q_1, \theta)\} \\ m_M(q_1, b, q_2, \theta) &= \{(q_1, AB, q_2, \theta)\} \\ m_M(q_0, a, q_3, \theta) &= \{(q_0, \Phi, q_3, \theta)\} \\ m_M(q_3, b, q_4, \theta) &= \{(q_3, \Phi, q_4, \theta)\} \\ m_M(q_4, c, q_2, \theta) &= \{(q_4, ABC, q_2, \theta)\} \end{aligned}$$

O contradomínio de m_M *corresponde a transições do autómato de acções associadas e* M *que chamaremos de* N, *e está definido do seguinte modo:*

$N = (\{q_0, q_1, q_2, q_3, q_4\}, \{AB, ABC, \Phi\}, \delta, \{\}, q_0, \{q_2\})$

$$\begin{aligned}
\delta(q_0, \Phi) &= \{(q_1, \theta)\} \\
\delta(q_1, AB) &= \{(q_2, \theta)\} \\
\delta(q_0, \Phi) &= \{(q_3, \theta)\} \\
\delta(q_3, \Phi) &= \{(q_4, \theta)\} \\
\delta(q_4, ABC) &= \{(q_2, \theta)\}
\end{aligned}$$

Transformando o ARN M para o seu equivalente ARD M' obtemos:

$M' = (\{q_0, q_{1,3}, q_2, q_{2,4}\}, \{a, b, c\}, \delta, \{\}, q_0, \{q_2, q_{2,4}\})$

$$\begin{aligned}
\delta(q_0, a) &= (q_{1,3}, \theta) \\
\delta(q_{1,3}, b) &= (q_{2,4}, \theta) \\
\delta(q_{2,4}, c) &= (q_2, \theta)
\end{aligned}$$

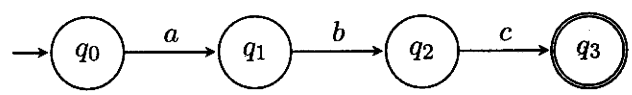

Figura 4.7: ARD M', associação de acções

e actualizamos a função de acções:

$$\begin{aligned}
m_{M'}(q_0, a, q_{1,3}, \theta) &= \{(q_0, \Phi, q_1, \theta), (q_0, \Phi, q_3, \theta)\} \\
m_{M'}(q_{1,3}, b, q_{2,4}, \theta) &= \{(q_1, AB, q_2, \theta), (q_3, \Phi, q_4, \theta)\} \\
m_{M'}(q_{2,4}, c, q_2, \theta) &= \{(q_4, ABC, q_2, \theta)\}
\end{aligned}$$

O próximo passo será converter o ARN N de acções para o seu equivalente determinista N':

$N' = (\{q_0, q_2, q_{1,3}, q_4\}, \{AB, ABC, \Phi\}, \delta, \{\}, q_0, \{q_2\})$

$$\begin{aligned}
\delta(q_0, \Phi) &= \{(q_{1,3}, \theta)\} \\
\delta(q_{1,3}, AB) &= \{(q_2, \theta)\} \\
\delta(q_{1,3}, \Phi) &= \{(q_4, \theta)\} \\
\delta(q_4, ABC) &= \{(q_2, \theta)\}
\end{aligned}$$

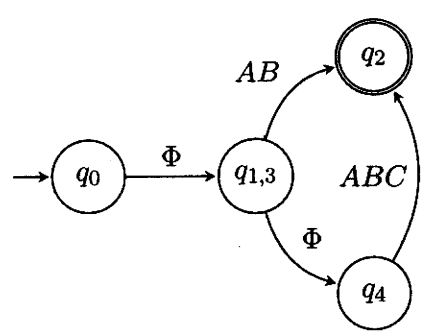

Figura 4.8: ARD N' de acções associadas a M

Por fim actualizamos mais uma vez a função de acções para $m_{M''}$:

$$m_{M''}(q_0, a, q_{1,3}, \theta) = \{(q_0, \Phi, q_{1,3}, \theta)\}$$
$$m_{M''}(q_{1,3}, b, q_{2,4}, \theta) = \{(q_{1,3}, AB, q_2, \theta), (q_{1,3}, \Phi, q_4, \theta)\}$$
$$m_{M''}(q_{2,4}, c, q_2, \theta) = \{(q_4, ABC, q_2, \theta)\}$$

Para finalizar este exemplo iremos testar as duas palavras da nossa linguagem:

1. *Palavra ab, aceite por M' pelo caminho $q_0 q_{1,3} q_{2,4}$;*

   O caminho de acções associados ao caminho anterior é

   $$\{(q_0, \Phi, q_{1,3}, \theta)\}\{(q_{1,3}, AB, q_2, \theta), (q_{1,3}, \Phi, q_4, \theta)\}$$

   ;

   Da sequência de acções escolhemos apenas os caminhos que terminam em estados finais no ultimo elemento da sequência de acções. Neste caso q_4 não é estado final então retiramos o caminho $q_0 q_{1,3} q_4$ do espaço de resultados, onde, ficamos com $q_0 q_{1,3} q_2$.

   Portanto a sequência de acções a ser executada para a palavra ab é ΦAB.

   Por outro lado podemos obter a solução, preferível se a linguagem de resultados conter um numero exponencial de palavras, em forma de AR

   $S' = (\{q_0, q_2, q_{1,3}\}, \{AB, \Phi\}, \delta, \{\}, q_0, \{q_2\})$

   $$\begin{aligned} \delta(q_0, \Phi) &= \{(q_{1,3}, \theta)\} \\ \delta(q_{1,3}, AB) &= \{(q_2, \theta)\} \end{aligned}$$

   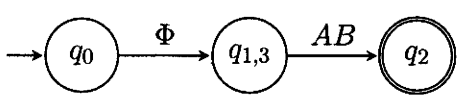

Figura 4.9: ARD S' de acções associadas a M quando a palavra ab é aceite.

2. *Palavra abc, aceite por M' pelo caminho $q_0 q_{1,3} q_{2,4} q_2$*

   O caminho de acções que resulta desta sequência é o seguinte

   $$\{(q_0, \Phi, q_{1,3}, \theta)\}\{(q_{1,3}, AB, q_2, \theta), (q_{1,3}, \Phi, q_4, \theta)\{(q_4, ABC, q_2, \theta)\}$$

   Retirando os caminhos que não terminam no ultimo elemento da sequência de acções obtemos $q_0 q_{1,3} q_4 q_2$ então a sequência de acções a ser executada para a palavra abc é $\Phi\Phi ABC$.

   Por outro lado podemos obter o nosso resultado na forma de um AR, em que a solução neste caso seria o AR

Exemplo 27 *Neste exemplo iremos ilustrar o funcionamento de metadados associados a uma transição com referencia a uma máquina. Supomos que temos o seguinte AR M definido por:*

$M = (\{q_0, q_1, q_2, q_3\}, \{a\}, \delta, \{X'\}, q_0, \{q_3\})$
$\mathcal{C}(X') = X$

$$\begin{aligned} \delta(q_0, a) &= \{(q_1, \theta)\} \\ \delta(q_1, x) &= \{(q_2, X'\} \\ \delta(q_2, a) &= \{(q_3, \theta)\} \end{aligned}$$

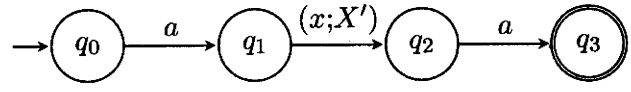

Figura 4.10: AR M, associação de acções a máquinas

Queremos associar a $\mathcal{L}(X)$ a acção A, então definimos o AR de acções N correspondente a M:

$N = (\{q_0, q_1, q_2, q_3, q_4\}, \{a\}, \delta, \{X''\}, q_0, \{q_3\})$
X'' é uma referencia à maquina de acções que corresponde à máquina de símbolos X

$$\begin{aligned} \delta(q_0, \Phi) &= \{(q_1, \theta)\} \\ \delta(q_1, x) &= \{(q_4, X'')\} \\ \delta(q_4, A) &= \{(q_2, \theta)\} \\ \delta(q_2, \Phi) &= \{(q_3, \theta)\} \end{aligned}$$

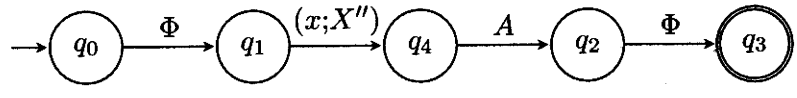

Figura 4.11: AR N de acções associado ao AR M

Por fim a função de acções é definida por:

$$\begin{aligned} m_M(q_0, a, q_1, \theta) &= \{(q_0, \Phi, q_1, \theta)\} \\ m_M(q_1, b, q_2, X') &= \{(q_1, x, q_4, X''), (q_1, A, q_2, \theta)\} \\ m_M(q_2, a, q_3, \theta) &= \{(q_2, \Phi, q_3, \theta)\} \end{aligned}$$

Construído o autómato determinista, vamos testa-lo com a linguagem $a\mathcal{L}(X)a$.

1. *Seja w uma palavra da linguagem $a\mathcal{L}(X)a$, então w é aceite por M pelo caminho q_0, q_1, q_2, q_3;*
2. *O caminho na máquina de acções N associado a w é q_0, q_1, q_4, q_2, q_3;*
3. *$w = axa$, em que x é uma palavra aceite por X então existe uma função de metadados que associa x a uma linguagem de acções S;*

4. Logo temos que a linguagem de acções associada à palavra w é $\Phi SA\Phi$.

4.1.5 Vantagens

Uma das maiores vantagens em definir todas as relações das acções como um autómato determinista é que herdamos todas as suas características, sendo a mais importante o facto de termos definida toda a linguagem de resultados de uma maneira estática e finita. Podemos também optimizar esse mesmo autómato usando minimização, ou outras técnicas.

Outra das vantagens é que todas as acções executadas produzem resultados úteis, porque a sua validação já foi feita. Assim todas elas são intencionalmente executadas ao contrário de outros sistemas, que executam todas as acções potencialmente úteis e que posteriormente as anulam quando estas se tornam inválidas.

Como todas as acções estão expostas como um AR determinista, é mais fácil controlar a sua execução, sendo possível criar optimizações como manter em memoria resultados de cálculos duplicados ou mesmo cortar caminhos ou fracções de caminhos que já foram executados e que produzem resultados duplicados.

Do ponto de vista da implementação estamos a reutilizar código, o que é sempre uma grande vantagem.

Por último, os resultados, que em alguns casos são de uma magnitude exponencial, podem ser apresentados na forma de um AR determinista, ao contrario de serem apresentadas todas as soluções uma a uma. Isto também possibilita operações nos resultados antes destes serem executados, caso seja essa a intenção.

4.1.6 Desvantagens

Uma das maiores desvantagens é que estamos restringidos a uma linguagem de resultados reconhecida pelos AR deterministas.

A construção e sincronização do autómato de símbolos com o autómato de acções é mais complexa do que em outros sistemas.

É necessário manter uma lista de referências para os metadados fornecidos pelo autómato de símbolos até que a palavra seja aceite ou rejeitada pelo mesmo e, no caso de aceitação, extrair quais as acções válidas a serem executadas. Isto pode ter um grande impacto na memória.

4.1.7 Conclusões

Apesar de não ser um sistema perfeito, com várias restrições tentámos balancear os vários prós e contras. Uma vez que esta secção se relaciona fundamentalmente com a implementação prática do algoritmo, levámos em conta factores como a complexidade, a velocidade e a memória.

A a memória aumenta proporcionalmente em relação ao autómato de símbolos a que as acções estão associadas, assim como a lista de metadados usados para validar acções é proporcional ao tamanho da palavra dada. Todas as operações na aceitação de uma palavra de comprimento n, e na extracção de acções são aproximadamente O(n) para cada uma delas. Em termos de memória, apenas a lista de metadados cresce dinamicamente. O número de problemas que podem ser resolvidos com este tipo de estratégia está limitado pelas linguagens que o autómato determinista reconhece. No entanto, obtemos outras vantagens, como, conhecimento do espaço de resultados de uma forma determinista, capacidade de usar todas as características e operações dos ARD e por fim um maior controlo sobre os resultados obtidos.

Capítulo 5

Implementação

Os capítulos anteriores descrevem um modelo teórico de autómato. Aqui começamos a descrição da implementação desse modelo. Esta é constituída por duas componentes principais: uma biblioteca de uso geral, que dá forma ao modelo teórico dos autómatos recursivos e, com base nessa biblioteca, a título de ilustração/aplicação, um gerador de código de analisador sintáctico e lexical através de especificações FBNA (Forma Backus–Naur Aumentada, em inglês *Augmented Backus–Naur Form*) [Net08]. Com esta última aplicação emergiram máquinas tipo, novas implementações de interfaces de entrada e saída, sendo os mais relevantes; máquinas com associação de acções, geradores de gráficos em GraphViz, geradores de código C++ e Run.

Nas secções que se seguem discutimos os vários aspectos da implementação, as escolhas tomadas, a arquitectura e finalmente uma análise global dos resultados em comparação com a realidade actual.

A biblioteca, desenvolvida com licença de código livre GPL (GPL, *GNU public license* [Fre07]) é implementada na linguagem C++ e tem como principais características:

- portabilidade: pode ser compilada e usada nas plataformas que suportem C++ e STL (*Standard Template Library*);
- uso em tempo de execução: a biblioteca pode ser ligada a um programa que defina e use autómatos em tempo de execução.
- metadados: os estados e transições de um autómato podem ser associados a metadados, o processamento, por um autómato, de uma palavra produz uma sequência de metadados, que podem ser processados numa fase posterior.
- símbolos e metadados abstractos: os símbolos assim como os metadados são definidos pelo programador, podendo este escolher o tipo de dados que mais lhe convém.
- interface para introduzir produções no formato de AR não-determinista (interface de entrada): permite ao programador abstrair-se do sistema que gere e constrói os autómatos, preocupando-se apenas com o AR que quer definir.
- interface para manipular os resultados e acesso à representação interna do AR determinista (interface de saída): esta camada permite abstrair o sistema

mapeando-o para os conceitos de AR tais como transições, símbolos, máquinas, computação, metadados, alfabetos, estados, estados finais... O programador pode facilmente aceder a todos estes elementos de uma forma transparente e aleatória.

As características aqui enunciadas, tais como a definição de símbolos e metadados, permitem aos programadores estenderem a biblioteca de acordo com as suas necessidades. A contribuição de novos símbolos e metadados faz com que seja possível obter varias máquinas tipo e criar novas combinações que podem ser reutilizadas por outros. O mesmo acontece com as contribuições efectuadas para as interfaces de entrada e saída.

É importante considerar que no estudo teórico, não nos foi possível provar, nem refutar, que qualquer ARN pode ser simulado por um ARD. Ainda assim, para certos ARN conseguimos definir um ARD equivalente. Ora, o funcionamento desta biblioteca assenta no processamento de uma palavra por um ARD, embora o autómato possa ser definido como um ARN. Portanto, nesta fase, o suporte que a biblioteca fornece a computações não-deterministas é, ainda, restrito aos casos que podem ser automaticamente convertidos em ARD equivalentes. O esclarecimento dos restantes ARN tem que ficar adiado para trabalho futuro.

5.1 Plataforma

Tendo em vista a maior portabilidade desta biblioteca, esta está implementada em C++, com uso da STL. Além disso, a escolha desta linguagem assenta nas seguintes razões:

1. facilidade de integração em outras linguagens, pois muitas delas possuem suporte para a ligação de bibliotecas em C/C++;

2. eficiência no código gerado;

3. estabilidade; em que o código gerado é executado de maneira previsível para arquitecturas equivalentes. O mesmo não acontece para linguagens interpretadas ou emuladas (máquinas virtuais) em que os comportamentos dependem da implementação, tornando difícil ou impossível a correcção de erros;

4. fiabilidade; os erros mais comuns são reportados no processo de compilação e ligação, ao contrario de linguagens interpretadas que só reconhecem muitos dos erros quando estes ocorrem;

5. tem à sua disposição uma grande escolha de bibliotecas de grande qualidade;

6. documentação de qualidade;

7. possui uma enorme comunidade de utilizadores e suporte, o que nos leva a acreditar que a biblioteca terá mais utilidade e sucesso.

5.2 Arquitectura actual

A arquitectura é constituída por um componente parcialmente definido, este é o núcleo da biblioteca (librfa, *recursive finite automata library*). A librfa gere internamente toda a construção dos AR deterministas, deixando por definir a representação de símbolos e metadados. Quando completada a sua definição, a librfa pode ser acedida por uma interface de entrada e de saída.

Apesar da definição de símbolos e metadados ser livre, a librfa impõe algumas condições necessárias: no caso dos símbolos apenas se impõe que estes sejam ordenáveis/enumeráveis; os metadados para além de serem enumeráveis têm que ter definida a operação união, em que a união de metadados resulta em um único metadado. Estes disponibilizam também uma interface de notificações especificas transmitidas pela librfa, que reflecte as propriedades das transições ou estados em que estes estão associados.

Definidos os metadados e os símbolos, obtemos um tipo de máquina AR, podendo assim criar-se máquinas desse mesmo tipo. Cada máquina disponibiliza uma interface de entrada e saída, onde podemos declarar transições, estados finais e associar metadados a estes últimos de uma maneira indeterminista e através da interface de saída aceder à sua representação determinista.

As duas interfaces de entrada e saída possibilitam a transformação dos dados antes e depois destes serem processados pela librfa. Isto permitiu-nos adicionar outros componentes, considerados externos, mas de grande importância, como por exemplo, as máquinas com associação de acções.

Por fim existe um componente externo que gere um grupo de máquinas. O gestor de máquinas tem apenas a finalidade de auxiliar o programador a gerir e identificar os grupos e máquinas com que está a trabalhar.

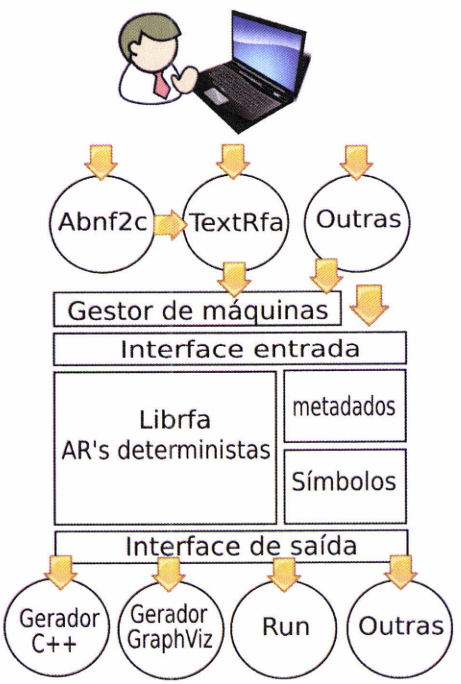

Figura 5.1: Arquitectura da implementação actual

5.2.1 Interfaces de entrada

A interface de entrada é o conjunto de métodos que de alguma maneira modificam a representação interna de um AR determinista. É considerado uma interface de entrada qualquer código que define a máquina, que a constrói e que é normalmente um tradutor entre os valores de entrada para interface base de entrada da librfa.

Ex. TextRfa , é uma interface de entrada que transforma uma especificação de uma máquina em texto para instruções de entrada da librfa.

Interfaces de saída

A interface de saída é o conjunto de métodos que acedem à representação interna de um AR determinista. É a camada que apresenta ou transforma os resultados de uma maneira contextual.

Ex. GraphViz , traduz uma ou mais máquinas para a linguagem dot do GraphViz, gerando um ou mais ficheiros .dot. Os ficheiros podem ser convertidos com a ferramenta dot para a sua representação gráfica nos formatos mais comuns, tal como svg, png, bmp ...

Ex. CPP , gera em código C/C++ o equivalente ao conjunto de máquinas ou máquina fornecida. Este pode ser depois compilado e executado. As máquinas compiladas são estáticas, retendo apenas as seguintes funcionalidades mais importantes:

- obtenção de símbolos através de qualquer fonte;
- análise de varias palavras ao mesmo tempo;
- aceitação, rejeição, falha e recuperação;
- acesso de forma indiscriminada a qualquer máquina.

Ex. Run , esta interface de saída equivale ao anterior (CPP), com a diferença de que o código não é gerado. A analise das palavras é feita sobre o AR na sua versão original/dinâmica logo todas as suas funcionalidades são mantidas, incluído a sua posterior modificação.

5.3 A biblioteca

5.3.1 Utilização

A biblioteca permite construir máquinas de autómatos recursivos deterministas, através de definições indeterministas. Existem dois tipos de objectos principais e é por eles que passam todas as operações. Estes são a máquina rfa (RFA) e o gestor de máquinas (RFAs). Ambos podem ser acedidos pelas interfaces de saída e entrada.

5.3.2 Construção dos Autómatos Recursivos Deterministas

Tal como já foi dito, a librfa constrói internamente um autómato recursivo determinista a partir de especificações indeterministas. Para converter as especificações indeterministas em deterministas usamos o algoritmo apresentado na parte teórica, mas introduzimos algumas optimizações.

Como na implementação, as transições são inseridas uma a uma. Isto dá-nos a oportunidade de manter internamente sempre um autómato determinista. Assim, qualquer nova transição pode ou não quebrar o determinismo. No caso de quebrar o determinismo, sabemos que apenas existe uma transição já existente nessa máquina que tem conflitos com a nova transição. Deste modo, em comparação com o que existe na parte teórica, não necessitamos de nos preocupar com n transições em conflito mas apenas com duas, reduzindo a sua complexidade.

A resolução de indeterminismo implica transformações que criam novos estados e inutilizam outros, portanto, estes necessitam de ser geridos de forma eficiente.

Toda a gestão de estados é feita internamente, e estes dividem-se em duas categorias:

- os estados de utilizador, que são sempre mantidos, pois estes podem vir a ser utilizados posteriormente;
- e os estados de sistema, que são libertados ou reutilizados sempre que sejam considerados inúteis (sem relações de entrada).

Um estado de sistema é único e é definido/identificado por um conjunto de estados de utilizador. Esta gestão visa minimizar a sua duplicação, e por consequência obter um melhor desempenho.

Sempre que é efectuada ou modificada uma relação seja interna ou através da API de entrada, é feita a actualização em todos os estados que unificam o estado de partida.

Deste modo mantemos autómatos recursivos deterministas pronto a ser utilizado sem comprometer a performance.

5.3.3 Metadados e Acções

Como dissemos na secção sobre a arquitectura, os metadados podem ser definidos pelo programador. Para um melhor desempenho os metadados estão associados a cada transição ou estado não como uma função, mas definidos como um modelo (em inglês *template*) na própria classe do objecto transição/estado. Sempre que possível, tentamos optimizar o uso de memoria evitando duplicações, fazendo uso de objectos estáticos, referencias e reciclagem/reutilização.

Nesta secção referimos também um tipo de metadados mais especifico a que chamamos acções. As acções são metadados que permitem a execução de código externo associado ao contexto da palavra aceite.

Consideramos que associar acções é uma questão importante e, portanto, o algoritmo apresentado em outros capítulos foi também implementado.

Neste trabalho usamos as acções para construir as nossas aplicações de demonstração, mas estas podem ser utilizadas em qualquer outra aplicação.

Capítulo 6

Exemplos

Para demonstrar algumas das utilidades da biblioteca, foram realizadas dois programas com base na mesma.

Apesar de nos focarmos nos programas seguintes como demonstração da aplicação mais directa da biblioteca, esta revelou-se bastante útil na resolução de vários problemas internos, tanto da própria biblioteca como das aplicações seguintes.

Podemos então afirmar que a própria biblioteca já é só por si uma das demonstrações da sua utilidade.

As secções que se seguem descrevem estas aplicações:

- TextRfa;
- Abnf2c.

6.1 TextRfa

A aplicação Textrfa lê um AR em forma de texto e constrói-o usando a biblioteca, que de seguida gera para o formato C++ e GraphViz.

A sua arquitectura baseia-se numa máquina com símbolos definidos como caracteres de texto e acções como metadados.

6.2 Gerador FBNA

FBNA (em inglês, *Augmented Backus–Naur Form*), surgiu ao longo dos anos como uma necessidade de especificar as questões técnicas da Internet. É pratica comum os seus autores usarem a notação que mais lhes convém, dando origem a várias extensões e modificações às gramáticas FBN, que se popularizaram com o nome de ABNF (*Augmented Backus–Naur Form*). Presentemente existem algumas tentativas de a tornar num modelo a seguir (em inglês, *standard*), e podemos considerar dois documentos como os mais importantes, estando ambos em discussão, e tendo eles especificações e objectivos diferentes:

- o grupo W3C(World Wide Web Consortium), que a usa na construção de linguagens de forma a serem usados em reconhecedores de fala;

- o RFC 5234 (em inglês, *request for comments*) [Net08] que é a base desta aplicação. Surge da própria comunidade da Internet, e contém várias contribuições desta. FBNA pode ser encontrada em varias especificações como o endereço de correio electrónico, protocolos e endereços de Internet, ...

Por questões técnicas (alguns erros de implementação que ainda não foram solucionados) e de utilidade, modificámos a gramática original, acrescentando-lhe acções. A aplicação permite através de especificações FBNA modificadas construir automaticamente analisadores lexicais e sintácticos.

Antes de passarmos aos exemplos faremos uma pequena introdução informal aos elementos de uma gramática FBNA.

- FBNA é uma lista de uma ou mais regras;
- cada regra é identificada por um nome;
- o sinal = é usado na declaração de uma nova regra;

  ex. regra = elementos;
- o sinal / representa as varias alternativas;
- o sinal =/ é usado para adicionar alternativas a uma regra já existente;
- os espaços entre elementos representam a sua concatenação;
- um elemento pode ser uma palavra, grupo, opção, um valor numérico ou um intervalo de valores numéricos;

  palavra = sequência de caracteres visíveis delimitada por «aspas» (ex. "...");

  grupo = '(' elementos ')';

  opção = '['elementos']', equivalente ao grupo ((elemento) / ϵ);
- as repetições tem a forma de min*max elemento, em que o min e o max são valores naturais e representam o número min e max de vezes que o elemento deve aparecer. O min e max são opcionais, e os seus valores por defeito são, min=0 e max=infinito;

  ex. *"a", 0 ou mais "a"'s;

  ex. 2*"a", 2 ou mais "a"'s;

  ex. 0*1"a", equivalente a ["a"];
- os elementos são constituídos, por alternativas, concatenações e repetições de elementos.
- Introduzimos as acções, em que cada acção é identificada por $

Programa 1 Gramática de Textrfa (formato Abnf2c)

```
#
#       This file is part of Abnf2c.
#
#   Abnf2c is free software: you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation, either version 3 of the License, or
#   (at your option) any later version.
#
#   Abnf2c is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
#   You should have received a copy of the GNU General Public License
#   along with Abnf2c.  If not, see <http://www.gnu.org/licenses/>.
#

textrfa = [spaces] rule *(spaces [rule]);


name = ( %x41-5A$saveChar / %x61-7A$saveChar)
                *(        %x41-5A$saveChar
                        / %x61-7A$saveChar
                        / "-"$saveUnderscore
                        / %x30-39$saveChar
                )
;


spaces = (
                        %x20$space / %x09$tab / %x0a$newline
                        / "#"$comment *(%x20-7e$space / %x09 ) %x0a$newline
                )
                *(
                        %x20$space / %x09$tab / %x0a$newline
                        / "#"$comment *(%x20-7e$space / %x09$tab ) %x0a$newline
                )
;

rule = name$declareRfa [spaces] "," [spaces]
        (
           number$stateFrom [spaces]
           "," [spaces] condition [spaces]
           "," [spaces] number$stateTo [spaces]
           "," [spaces] name$action
         /
           "f"$beginFinal "inal" spaces number$stateFinal [spaces] "."
        )
;
```

```
condition =
         "rfa" spaces name$call
        / symbol$minSymbol ["."$range "." symbol$maxSymbol]
;

symbol = number$beginSymbol
                / "'"$beginSymbol %x20-7e$charSymbol "'"
                / "n"$newlineSymbol "ewline"
                / "t"$tabSymbol "ab"
;


number = %x30-39$digit *(%x30-39$digit);
```

Programa 2 Gramática de Abnf2c (formato Abnf2c)

```
#
#       This file is part of Abnf2c.
#
#   Abnf2c is free software: you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation, either version 3 of the License, or
#   (at your option) any later version.
#
#   Abnf2c is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
#   You should have received a copy of the GNU General Public License
#   along with Abnf2c.  If not, see <http://www.gnu.org/licenses/>.
#


rulelist =  *( [spaces]
                                rulename$declareRulename
                                [spaces] "="$equal elements$elements
                                ";"$endRule
                    ) [spaces];

rulename = ( %x41-5A$saveChar / %x61-7A$saveChar)
                *( %x41-5A$saveChar
                        / %x61-7A$saveChar
                        / "-"$saveChar
                        / %x30-39$saveChar
                )
;

spaces = (
                 %x20$space / %x09$tab / %x0a$newline
                 / "#"$comment *(%x20-7e$space / %x09 )
                 %x0a$newline
                )
                *(
                 %x20$space / %x09$tab / %x0a$newline
                 / "#"$comment *(%x20-7e$space / %x09$tab )
                 %x0a$newline
                )
;
```

```
elements = [spaces] repeat$repeat
                        *(
                         spaces
                                [
                                        repeat$repeat
                                /
                                        "/"$alternation [spaces]
                                        repeat$repeat
                                ]
                        /
                         "/"$alternation [spaces] repeat$repeat
                        )
;


repeat = ["*"$repeatInf] element$element;


element =
        (
                rulename$callRulename
                / "["$optionStart elements$option "]"$optionEnd
                / "("$groupStart elements$group ")"$groupEnd
                / char-val
                / prose-val
                / num-val
        ) ["$"$startAction rulename$action]
;

char-val = %x22$charStringStart
                *(%x20-21$saveSymbolChar / %x23-7E$saveSymbolChar)
                   %x22$charStringEnd
;


prose-val = "<"$charProseStart
                *(%x20-3d$saveSymbolChar / %x3f-7E$saveSymbolChar)
                   ">"$charProseEnd
;


num-val = "%"$numStart (bin-val$bin / dec-val$dec / hex-val$hex );
```

```
bin-val = "b"$binStart
                        %x30-31$saveBin
                        *(%x30-31$saveBin)
                        [
                                *("."$concatBin
                                        %x30-31$saveBin
                                        *(%x30-31$saveBin)
                                )
                        /
                                "-"$binRange
                                        %x30-31$saveBin
                                        *(%x30-31$saveBin)
                        ]
;

dec-val = "d"$decStart
                        %x30-39$saveDec
                        *(%x30-39$saveDec)
                        [
                                *("."$concatDec
                                        %x30-39$saveDec
                                        *(%x30-39$saveDec)
                                )
                        /
                                "-"$decRange
                                %x30-39$saveDec
                                *(%x30-39$saveDec)
                        ]
;
```

```
hex-val = "x"$hexStart
        ( %x30-39$saveHexDec
                / %x41-46$saveHexUpperLetter
                / %x61-66$saveHexLowerLetter )
                *( %x30-39$saveHexDec
                        / %x41-46$saveHexUpperLetter
                        / %x61-66$saveHexLowerLetter )
                [
                *("."$concatHex
                        ( %x30-39$saveHexDec
                                / %x41-46$saveHexUpperLetter
                                / %x61-66$saveHexLowerLetter )
                        *( %x30-39$saveHexDec
                                / %x41-46$saveHexUpperLetter
                                / %x61-66$saveHexLowerLetter )
                        )
                /
                        "-"$hexRange
                        ( %x30-39$saveHexDecMax
                                / %x41-46$saveHexUpperLetterMax
                                / %x61-66$saveHexLowerLetterMax )
                        *( %x30-39$saveHexDecMax
                                / %x41-46$saveHexUpperLetterMax
                                / %x61-66$saveHexLowerLetterMax )
                ]
;
```

Ambos os programas Abnf2c e Textrfa são definidos em FBNA modificada. O programa Abnf2c gera os AR correspondentes em formato de texto, que por fim serão processados pelo Textrfa.

Os metadados/acções são definidos por um identificador, que cabe às interfaces de saída resolver.

No caso do GraphViz, o identificador é utilizado na sua forma original, para se auto-referenciar.

No backend C++ o identificador refere-se a um método compatível com a linguagem C++.

Para clarificar todos este conceitos iremos exemplificar todo este processo construindo uma calculadora infix. Efectuaremos todos os passos necessários para construir e correr a nossa calculadora.

Começamos por definir em FBNA modificada a gramática de uma simples calculadora infix que respeita a prioridade dos operadores.

Programa 3 Gramática de calculadora (formato Abnf2c)

```
exp = exp-2$a [("+" exp$add-b / "-" exp$sub-b )];
exp-2 = exp-1$a ["/" exp-2$div-b];
exp-1 = exp-value$a ["*" exp-1$mul-b];


exp-value =
        [spaces]
        (       number$number
                /
                "("$begin exp$exp ")"$end
        )
        [spaces]
;

spaces = (%x20$space / %x09$tab / %x0a$newline )
                *(%x20$space / %x09$tab / %x0a$newline )
;


number = %x30-39$startDecDigit *(%x30-39$decDigit);
```

Processando a gramática da calculadora com o programa Abnf2c obtemos o AR em modo de texto, pronto para ser construído pelo programa Textrfa. Este irá gerar os ficheiros C++ correspondentes ao AR dado.

Programa 4 AR de calculadora em formato de texto (Gerado com Abnf2c)

```
exp, 0, rfa exp-2, 1, a
exp, 1, '+', 2, nop
exp, 2, rfa exp, 3, add-b
exp, 1, '-', 4, nop
exp, 4, rfa exp, 3, sub-b
exp, final 1.
exp, final 3.
exp-2, 0, rfa exp-1, 1, a
exp-2, 1, '/', 2, nop
exp-2, 2, rfa exp-2, 3, div-b
exp-2, final 1.
exp-2, final 3.
exp-1, 0, rfa exp-value, 1, a
exp-1, 1, '*', 2, nop
exp-1, 2, rfa exp-1, 3, mul-b
exp-1, final 1.
exp-1, final 3.
exp-value, 0, rfa spaces, 1, nopCall
exp-value, 0, rfa number, 2, number
exp-value, 1, rfa number, 2, number
exp-value, 0, '(', 3, begin
exp-value, 1, '(', 3, begin
exp-value, 3, rfa exp, 4, exp
exp-value, 4, ')', 5, end
exp-value, 2, rfa spaces, 6, nopCall
exp-value, 5, rfa spaces, 6, nopCall
exp-value, final 2.
exp-value, final 5.
exp-value, final 6.
spaces, 0, ' ', 1, space
spaces, 0, tab, 1, tab
spaces, 0, newline, 1, newline
spaces, 1, ' ', 1, space
spaces, 1, tab, 1, tab
spaces, 1, newline, 1, newline
spaces, final 1.
number, 0, '0'..'9', 1, startDecDigit
number, 1, '0'..'9', 1, decDigit
number, final 1.
```

O passo seguinte será escrever o resto do programa, ou seja, as acções e o programa principal (*«main»*).

Começamos então pelas acções. Cada acção é executada pela ordem da sequência de caracteres a que corresponde. Apenas acções válidas são executadas, isto é, apenas as acções correspondentes a caminhos aceites são executadas.

Programa 5 Acções da calculadora (C++)

```
#include "calc.hpp"

Calc::Calc(){
        value=0;
}

bool Calc::a(unsigned int){
}

void Calc::push(){
        values.push_front(value);
}

float Calc::pop(){
        float ret = values.front();
        values.pop_front();
        return ret;
}

bool Calc::add_b(unsigned int){
        float b = pop();
        float a = pop();
        value = a+b;
        push();
        return true;
}

bool Calc::sub_b(unsigned int){
        float b = pop();
        float a = pop();
        value = a-b;
        push();
        return true;
}

bool Calc::div_b(unsigned int){
        float b = pop();
        float a = pop();
        value = a/b;
        push();
        return true;
}

bool Calc::mul_b(unsigned int){
        float b = pop();
        float a = pop();
        value = a*b;
        push();
        return true;
}
```

```
bool Calc::number(unsigned int){
        push();
        return true;
}

bool Calc::space(unsigned int){
        // ignore
        return true;
}

bool Calc::tab(unsigned int){
        // ignore
        return true;
}

bool Calc::newline(unsigned int){
        // ignore
        return true;
}

bool Calc::decDigit(unsigned int symbol){
        value = symbol - '0' + value*10;
        return true;
}

bool Calc::startDecDigit(unsigned int symbol){
        value = symbol -'0';
        return true;
}

bool Calc::nop(unsigned int){
        return true;
}

bool Calc::nopCall(unsigned int){
        return true;
}

bool Calc::begin(unsigned int){
        return true;
}

bool Calc::exp(unsigned int){
        return true;
}
```

```
bool Calc::end(unsigned int){
        return true;
}

float Calc::getResult(){
        return value;
}

void Calc::end(){
        pop();
}
```

E por fim, o ficheiro principal («main») que irá usar todos os outros para construir o programa final. Não entrando em pormenores demasiado técnicos, o rfa gerado corresponde na realidade a todos os estados de cada rfa transformados numa função. A função recebe um objecto a que chamaremos «parser». Este contém toda a informação do estado actual do analisador. O nome de cada estado é criado de uma forma automática, sendo o estado inicial de cada máquina o próprio nome, como prefixo, e «_start» como terminação. Dado o estado inicial e uma palavra, o programa irá rejeitar ou aceitar a palavra. No caso de aceitação, as acções associadas à palavra são executadas.

Programa 6 Calculadora programa principal (main, C++)

```
#include "charrfa.hpp"
#include "calc.hpp"

typedef CharRfa<Calc> CharRfaT;

extern void (*exp_start)(CharRfaT::BacktrackParser *parser);

int main(int argc, char *argv[]){

        if( (argc!=2) || (argc > 6)){
                cout << argv[0] << "expression\n";
                cout << "ex: " << argv[0] << "'1 + 2 * (3+4)'\n";
                return 0;
        }

        string expression = argv[1];
        stringstream ss;
        ss.str(expression);

        CharRfaT charRfa(exp_start);

        Calc calc;
        CharRfaT::Error error;
        error = charRfa.run(&ss, &calc );

        if(error.isError()){

                switch(error.getReason()){
                        case CharRfaT::Error::ERROR_SINTAX:
                                cout << "sintax error, unexpected symbol "
                                                << error.getChar();
                                break;
                        default:
                                cout << "unknown";
                }

                cout << " at line " << error.getLine()
                                << " column " << error.getColumn() << ".\n";
                }else{
                        cout << calc.getResult() << "\n";
                }

                return 0;

}
```

No final apenas temos que compilar todos os ficheiros C++ num único programa e executar. Este processo é análogo para todos os programas deste tipo, tal como Abnf2c e Textrfa.

Capítulo 7

Conclusões

Neste capitulo faremos o balanço do trabalho realizado. Identificaremos os objectivos alcançados, as diferenças significativas a trabalhos relacionados e por fim que aspectos que se podem melhorar.

7.1 Pontos Negativos

Dos objectivos a que nos propusemos nem todos foram completamente alcançados. Consideramos que existem vários pontos incompletos, negativos mas também positivos.

Dos pontos incompletos consideramos que não foram provados:

- as transformações de um AR não-determinista para determinista [3.4.3];
- os algoritmos relacionados com as acções [4.1.3].

Para além destes também não foi determinada a linguagem reconhecida por um AR. Apenas acreditamos que se localizam entre as linguagens regulares e as linguagens livres de contexto [3.4.2].

Identificamos dois pontos negativos no trabalho. O primeiro, o sistema de acções é ainda pouco sofisticado e limitado. O segundo, a implementação, que:

- encontra-se presentemente num estado instável;
- aceita todos os AR não-deterministas, mesmo aqueles que não possuem equivalente determinista (consequência da sua base teórica);

Apesar disso consideramos o balanço positivo, pois vários objectivos foram alcançados com sucesso.

7.2 Objectivos Alcançados

Dos objectivos alcançados, a que nos propusemos, consideremos como sendo os mais importantes:

- a definição de AR, como um modelo de autómato simplificado, inserido na classe das linguagens livres de contexto;
- a construção de AR deterministas em tempo real;
- associação de acções/metadados independentes da base teórica dos AR.
- uma biblioteca facilmente extensível, com interfaces intuitivas e muito flexíveis;

A partir dos pontos acima referidos foi-nos possível criar diversificadas ferramentas de demonstração, tais como abnf2c [6.2], uma calculadora [6.2], gerador de código em C++ e um gerador de autómatos na sua forma gráfica [6.1].

7.3 Discussão e comparações com trabalhos relacionados

Em comparação com outras implementações, pensamos que a biblioteca implementada é muito mais flexível e fácil de ser integrada em varias aplicações podendo ser estendida de acordo com as necessidades de cada uma. Pelo contrario, as analisadas apenas se centram em problemas específicos, restringindo o número de aplicações em que podem ser utilizadas.

Uma das características únicas da biblioteca é que o autómato determinista é construído em tempo real e utilizado em tempo real o que abre a possibilidade de ser utilizada por aplicações que requerem estas características.

Nas aplicações analisadas, a grande maioria é especifica apenas para uma linguagem, seja a própria biblioteca ou o código gerado, e para muitas delas é impossível ou difícil fazerem-se ligações para outras. No caso da biblioteca é mais fácil fazer ligações para C/C++ e interfaces que geram código para outras linguagens.

Dividindo as aplicações em dois grupos em comparação com a biblioteca, existem as que implementam autómatos (in)deterministas mais poderosos e as que não o fazem. No primeiro caso, a biblioteca está limitada apenas a um subconjunto dos problemas que essas aplicações podem resolver, sendo por isso mais limitada. Para as restantes, todos os problemas resolvidos por estas são possíveis de ser resolvidos pela biblioteca, o que a torna mais poderosa.

Concluindo, cada biblioteca tem os seus prós e contras; algumas resolvem mais problemas, outras resolvem um só tipo de problemas de uma maneira mais eficaz ou são mais simples/disponíveis para determinada linguagem.

Acreditamos que, para os problemas que a biblioteca resolve, se constitui como uma boa escolha, uma vez que proporciona mais possibilidades que as restantes.

7.4 Limitações

Os AR's deterministas, reconhecem um conjunto de linguagens. Estas estão associadas aos problemas a que podem ser aplicados, logo os AR's deterministas estão limitados a esse conjunto de problemas.

Tal como já foi referido neste trabalho, o algoritmo proposto para converter um AR não-determinista para determinista é incapaz de reconhecer quais os AR não-deterministas que possuem equivalente determinista.

Por fim ainda é desconhecida a veracidade do algoritmo proposto.

7.5 Trabalho Futuro

Como principal prioridade consideramos terminar o estudo sobre as limitações referidas. Melhorar o sistema de acções, de modo a torna-lo mais eficiente e sofisticado.

Os autómatos tem inúmeras aplicações. Acreditamos que a biblioteca de AR's deterministas será uma delas, portanto no futuro, mais componentes da mesma serão implementados sobre esta, tendo como principal objectivo torna-la mais fiável. Do ponto de vista do utilizador é necessário melhorar e simplificar cada vez mais as interfaces e alargar o número de operações possíveis sobre os Autómatos, tais como minimização e produto, criar novos interfaces de entrada, saída, novos símbolos e metadados. Por fim, a biblioteca deverá ser avaliada e testada por vários utilizadores e receber as suas criticas e opiniões para a tornar mais fiável e produtiva.

Bibliografia

[Dur05] Irène Durand. Autowrite: A tool for term rewrite systems and tree automata. *Electronic Notes in Theorical Computer Science*, 124:29–49, 2005.

[eDG03] Gertjan van Noord e Dale Gerdemann. Finite state transducers with predicates and identities, 2003.

[eJC] Josep Carmona e Jonas Casanova. Syncroteam, http://synchroteam.sourceforge.net/.

[eLS01] G.-Q. Zhang e L. Smith. A collection of functional libraries for theory of computation. *Proceedings of the 39th ACM-SE Conference, ACM PRESS*, May 2001.

[eMS06] Dong Ha Nguyen e Mario Sdholt. Vpa-based aspects: better support for aop over protocols. Technical report, Ecole des Mines de Nantes - INRIA, LINA, June 2006.

[Fil03] Emmanuel Filiot. Automates d'arbres à arités arbitraires. Technical report, Ecole Normale Supérieure de Lyon, June 2003.

[Fre07] Inc Free Software Foundation. Gnu general public license v3.0, http://www.gnu.org/licenses/gpl-3.0-standalone.html, 2007.

[Gen98] Thomas Genet. Decidable approximations of sets of descendants and sets of normal forms. *roceedings 9th Conference on Rewriting Techniques and Applications, Tsukuba (Japan)*, volume 1379 of Lecture Notes in Computer Science:151–165, 1998.

[Gen01] Thomas Genet. Reachability analysis of term rewriting systems with timbuk. *Proc. of the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LNAI, Springer-Verlag*, 2250:691–702, 2001.

[JAU] Jautomata library, http://jautomata.sourceforge.net/.

[KM01] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual.* BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.

[Kra08] Dirk Krause. genau, http://genau.sourceforge.net/, 2008.

[LAM] Laml, finite state automaton library.

[Mao] Vincent Le Maout. *Expérience de programmation générique sur des structures non-séquentielles : les automates.* PhD thesis, Université de Marne La Vallée.

[Net08] Network Working Group. *Augmented BNF for Syntax Specifications: ABNF.* The IETF Trust, January 2008.

[NPC] Npcre project, http://dfa.sourceforge.net/.

[OMV95] A. Potthoff e W. Thomas O. Matz, A. Miller and E. Valkema. Report on the program amore. Technical Report 9507, Inst. für Informatik u. Prakt. Mathematik, 1995.

[PRI] Dicionário priberam da língua portuguesa, http://www.priberam.pt.

[RM01] Jeffrey D. Ullman e John E. Hopcroft Rajeev Motwani. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, segunda edição edition, 2001.

[RSE94] J. Weyuker e Martin Davis Ron Sigal Elaine. *Computability, complexity, and languages: fundamentals of theoretical Science.* Elsevier Science, segunda edição edition, 1994.

[Ser02] Ivan Hernandez Serrano. Ijaguar, http://ijaguar.sourceforge.net/, August 2002.