

Linguagens formais e autómatos:

apontamentos

(ano letivo de 2015/2016)

Artur Pereira

Fevereiro de 2016

Nota prévia

Este documento representa apenas um compilar de notas sobre a matéria teórica-prática das disciplinas “Linguagens Formais e Autómatos”, sem pretensões, por isso, de ser um texto exaustivo. A sua leitura deve apenas ser encarada como ponto de partida e nunca como único elemento de estudo.

Conteúdo

1	Linguagens	1
1.1	Exemplos de linguagens	1
1.2	Elementos básicos sobre linguagens	2
1.3	Operações sobre palavras	3
1.4	Operações sobre linguagens	4
2	Linguagens Regulares e Expressões Regulares	7
2.1	Definição de linguagem regular	7
2.2	Expressões regulares	8
2.3	Propriedades das expressões regulares	9
2.4	Simplificação notacional	10
2.5	Extensão notacional	11
2.6	Aplicação das expressão regulares	13
2.7	Exercícios	13
3	Autómatos Finitos	15
3.1	Autómatos Finitos Deterministas	16
3.1.1	Linguagem reconhecida por um AFD	17
3.1.2	Projecto de um AFD	18
3.1.3	Redução de AFD	20
3.2	Autómatos Finitos Não Deterministas	23
3.2.1	Árvore de caminhos de um AFND	24

3.2.2	Definição de autômato finito não determinista	25
3.2.3	Linguagem reconhecida por um AFND	26
3.3	Equivalência entre AFD e AFND	26
3.4	Operações sobre AFD e AFND	30
3.4.1	Reunião de autômatos	30
3.4.2	Concatenação de autômatos	33
3.4.3	Fecho de Kleene de autômatos	36
3.4.4	Complementação de autômatos	37
3.4.5	Intersecção de autômatos	38
3.4.6	Diferença de autômatos	40
4	Equivalência entre ER e AF	41
4.1	Conversão de ER em AF	41
4.1.1	Autômatos dos elementos primitivos	42
4.1.2	Algoritmo de conversão	42
4.2	Conversão de AF em ER	44
4.2.1	Algoritmo de conversão	45
5	Gramáticas	49
5.1	Definições	51
5.1.1	Definição de gramática	51
5.1.2	Derivação	53
5.2	Gramáticas regulares	54
5.2.1	Operações sobre as gramáticas regulares	54
5.2.2	Equivalência em relação aos AF e ER	55
5.3	Gramáticas independentes do contexto	58
5.3.1	Operações sobre gramáticas independentes do contexto	58
5.4	Árvore de derivação	59
5.4.1	Ambiguidade	60
5.5	Limpeza de gramáticas	62

5.5.1	Símbolos produtivos e não produtivos	62
5.5.2	Símbolos acessíveis e não acessíveis	63
5.5.3	Gramáticas limpas	64
5.6	Transformações em GIC	64
5.6.1	Eliminação de produções- λ	65
5.6.2	Eliminação de recursividade à esquerda	66
5.6.3	Factorização à esquerda	67
5.7	Os conjuntos first , follow e predict	67
5.7.1	O conjunto first	67
5.7.2	O conjunto follow	68
5.7.3	O conjunto predict	69
6	Gramática de atributos	71
6.1	Definição de gramática de atributos	71
6.1.1	Atributos herdados e atributos sintetizados	71
6.1.2	Construção de gramáticas de atributos	71
6.2	Ordem de avaliação dos atributos	72
6.2.1	Grafo de dependências	72
6.2.2	Tipos de gramáticas de atributos	72
7	Análise sintáctica descendente	73
7.1	Reconhecimento preditivo	74
7.2	Reconhecedores recursivo-descendentes	75
7.3	Reconhecedores descendentes não recursivos	76
7.4	Implementação de gramáticas de atributos	78
8	Análise sintáctica ascendente	79
8.1	Conflitos	81
8.2	Construção de um reconhecedor ascendente	83
8.2.1	Construção da coleção de conjuntos de itens	84
8.2.2	Tabela de reconhecimento	86
8.2.3	Algoritmo de reconhecimento	87

Capítulo 1

Linguagens

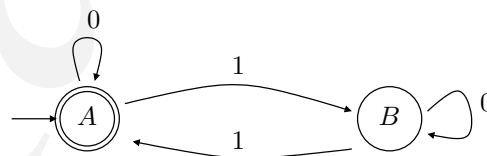
Uma linguagem é um sistema de símbolos usado para comunicar informação. Uma mensagem nessa linguagem é uma sequência de símbolos, mas nem todas as sequências são válidas. Logo, uma linguagem é caracterizada por um conjunto de símbolos e uma forma de descrever as sequências de símbolos válidas, ou seja, uma linguagem é um conjunto de sequências definidas sobre um conjunto de símbolos.

1.1 Exemplos de linguagens

Conjunto dos vocábulos em português é uma linguagem, cujos símbolos são as letras do alfabeto (mais as letras acentuadas e cedilhadas) e cujas sequências válidas são os vocábulos do português. É uma linguagem finita, pelo que as sequências válidas podem ser descritas por extenso. Poderão também ser descritas através de regras de produção. Por exemplo, as formas verbais de todos os verbos regulares terminados em ar são iguais, a menos de um prefixo. Estas palavras podem ser definidas pelo produto cartesiano dos possíveis prefixos com as possíveis terminações.

O conjunto das sequências binárias com um número par de uns é uma linguagem, cujos símbolos são os dígitos binários (0 e 1) e cujas sequências válidas são aquelas cujo número de uns é par.

Como descrever as sequências válidas? O grafo seguinte é uma forma de o fazer.



Uma sequência pertence à linguagem se atravessar o grafo e terminar no nó A.

O conjunto das sequências capicuas definidas com as letras 'a' e 'b' é uma linguagem, cujos símbolos são as letras 'a' e 'b' e cujas sequências válidas são as que se leem da mesma maneira se lidas da esquerda para a direita ou vice-versa. Esta linguagem pode ser definida por indução:

1. aa e bb são capicuas;
2. se s é capicua, também o são aSa e bSb.

O conjunto das sequências binárias começadas por '1' e terminadas em '0' é linguagem que pode ser representada pela expressão (regular) $1(0|1)^*0$. (As expressões regulares são tratadas mais à frente.)

O conjunto das sequências reconhecidas por uma máquina de calcular é uma linguagem, cujos símbolos são os dígitos, o separador decimal (ponto ou vírgula), os operadores, etc., e cujas sequências válidas são aquelas que representam expressões aritméticas válidas. Por exemplo: $10+1$ é uma expressão válida, mas $10++1$ não o é.

A **Língua portuguesa** é uma linguagem, cujos símbolos são os vocábulos do português (mais os sinais de pontuação), e cujas sequências são os textos em português. É um conjunto infinito, pelo que as sequências válidas apenas podem ser descritas através de regras de produção, a gramática do português.

1.2 Elementos básicos sobre linguagens

O **símbolo**, também designado por **letra**, é o átomo do mundo das linguagens.

O **alfabeto** é o conjunto de símbolos de suporte a uma dada linguagem. Deve ser um conjunto finito, não vazio, de símbolos. Exemplos: $A_1 = \{0, 1, \dots, 9\}$ é o alfabeto do conjunto dos números inteiros positivos; $A_2 = \{0, 1\}$ é o alfabeto do conjunto dos números binários.

A **palavra**, também designada por **string**, é uma sequência de símbolos sobre um dado alfabeto,

$$u = a_1 a_2 \cdots a_n, \quad \text{com } a_i \in A \wedge n \geq 0$$

Exemplos:

- 00011 é uma palavra sobre o alfabeto $\{0, 1\}$;
- abbbbcc é uma palavra sobre o alfabeto $\{a, b, c\}$.

O **comprimento** de uma palavra u denota-se por $|u|$ e representa o seu número de símbolos.

É habitual interpretar-se a palavra u como uma função

$$u : \{1 \cdots n\} \rightarrow A, \quad \text{com} \quad n = |u|$$

Se $u = abc$, então $u_1 = a$, $u_2 = b$ e $u_3 = c$.

A **palavra vazia** é uma sequência de 0 (zero) símbolos e denota-se por λ . Note que λ não pertence ao alfabeto.

A^* representa o conjunto de todas as palavras sobre o alfabeto A , incluindo a palavra vazia. Por exemplo, se $A = \{0, 1\}$, então $A^* = \{\lambda, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$.

Note que, dada um alfabeto A qualquer, uma **linguagem** sobre A mais não é do que um subconjunto de A^* . Note ainda que $\{\}$, $\{\lambda\}$ e A^* são subconjuntos de A e por conseguinte são linguagens sobre o alfabeto A .

1.3 Operações sobre palavras

A **concatenação**, ou **produto**, das palavras u e v denota-se por $u.v$, ou simplesmente uv , e representa a justaposição de u e v , i.e., a palavra constituída pelos símbolos de u seguidos dos símbolos de v . Note que $|u.v| = |u| + |v|$.

- A concatenação goza da propriedade **associativa**: $u.(v.w) = (u.v).w = u.v.w$.
- A concatenação goza da propriedade **existência de elemento neutro**: $u.\lambda = \lambda.u = u$.

A **potência** de ordem n , com $n \geq 0$, de uma palavra u denota-se por u^n e representa a concatenação de n réplicas de u , ou seja, $\underbrace{uu \cdots u}_{n \times}$. Note que u^0 é igual a λ , qualquer que seja o u .

Prefixo de uma palavra u é uma sequência com parte dos símbolos iniciais de u ; **sufixo** de uma palavra u é uma sequência com parte dos símbolos finais de u ; **sub-palavra** de uma palavra u é uma sequência de parte dos símbolos intermédios de u . Note que λ e u são prefixos, sufixos e sub-palavras de u .

O **reverso** de uma palavra u é a palavra, denotada por u^R , que se obtém invertendo a ordem dos símbolos de u , i.e., se $u = u_1 u_2 \cdots u_n$ então $u^R = u_n \cdots u_2 u_1$.

Denota-se por $\#(x, u)$ a função que devolve o número de ocorrências do símbolo x na palavra u .

1.4 Operações sobre linguagens

O conjunto das linguagens sobre um alfabeto A é fechado sobre as seguintes operações: reunião, intersecção, diferença, complementação, concatenação, potenciação e fecho de Kleene. Apresentam-se a seguir as definições destas operações assim como algumas propriedades de que gozam. As operações serão ilustradas com exemplos, tomando como ponto de partida as linguagens L_a e L_b definidas sobre o alfabeto $A = \{a, b, c\}$ da seguinte maneira:

$$L_a = \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\}$$

$$L_b = \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\}$$

A **reunião** de duas linguagens L_1 e L_2 denota-se por $L_1 \cup L_2$ e é definida da seguinte forma:

$$L_1 \cup L_2 = \{u \mid u \in L_1 \vee u \in L_2\}$$

EXEMPLO:

$$\begin{aligned} L_a \cup L_b &= \{u \mid u \text{ começa por } a \vee u \text{ termina com } a\} \\ &= \{xwy \mid w \in A^* \wedge (x = a \vee y = a)\} \cup \{a\} \\ &= \{w_1aw_2 \mid w_1, w_2 \in A^* \wedge (w_1 = \lambda \vee w_2 = \lambda)\} \end{aligned}$$

A **intersecção** de duas linguagens L_1 e L_2 denota-se por $L_1 \cap L_2$ e é definida da seguinte forma:

$$L_1 \cap L_2 = \{u \mid u \in L_1 \wedge u \in L_2\}$$

EXEMPLO:

$$\begin{aligned} L_a \cap L_b &= \{u \mid u \text{ começa por } a \wedge u \text{ termina com } a\} \\ &= \{awa \mid w \in A^*\} \cup \{a\} \end{aligned}$$

A **diferença** entre duas linguagens L_1 e L_2 denota-se por $L_1 - L_2$ e é definida da seguinte forma:

$$L_1 - L_2 = \{u \mid u \in L_1 \wedge u \notin L_2\}$$

EXEMPLO:

$$\begin{aligned} L_a - L_b &= \{u \mid u \text{ começa por } a \wedge u \text{ não termina com } a\} \\ &= \{awx \mid w \in A^* \wedge x \in A \wedge x \neq a\} \end{aligned}$$

A **complementação** da linguagem L_1 denota-se por $\overline{L_1}$ e é definida da seguinte forma:

$$\overline{L_1} = A^* - L_1 = \{u \mid u \notin L_1\}$$

As operações de reunião, intersecção e complementação gozam das **leis de DeMorgan**:

$$L_1 - (L_2 \cup L_3) = (L_1 - L_2) \cap (L_1 - L_3)$$

$$L_1 - (L_2 \cap L_3) = (L_1 - L_2) \cup (L_1 - L_3)$$

A **concatenação** de duas linguagens L_1 e L_2 denota-se por $L_1.L_2$ e é definida da seguinte forma:

$$L_1.L_2 = \{uv \mid u \in L_1 \wedge v \in L_2\}$$

EXEMPLO:

$$\begin{aligned} L_a.L_b &= \{uv \mid u \text{ começa por } a \wedge v \text{ termina com } a\} \\ &= \{awa \mid w \in A^*\} \end{aligned}$$

A concatenação goza das propriedades:

associativa: $L_1.(L_2.L_3) = (L_1.L_2).L_3 = L_1.L_2.L_3$

existência de elemento neutro: $L.\{\lambda\} = \{\lambda\}.L = L$

existência de elemento absorvente: $L.\emptyset = \emptyset.L = \emptyset$

distributiva em relação à reunião: $L_1.(L_2 \cup L_3) = L_1.L_2 \cup L_1.L_3$

distributiva em relação à intersecção: $L_1.(L_2 \cap L_3) = L_1.L_2 \cap L_1.L_3$

A **potência** de ordem n da linguagem L denota-se por L^n e é definida indutivamente da seguinte forma:

$$L^0 = \{\lambda\}$$

$$L^{n+1} = L^n.L$$

EXEMPLO:

$$(L_a)^0 = \{\lambda\}$$

$$(L_a)^1 = (L_a)^0.L_a = \{\lambda\}.L_a = L_a = \{aw \mid w \in A^*\}$$

$$(L_a)^2 = (L_a)^1.L_a = \{auav \mid u, v \in A^*\}$$

(Note ainda que $(L_a)^2 \neq \{(aw)^2 \mid w \in A^*\}$.)

O **fecho de Kleene** da linguagem L denota-se por L^* e é definido da seguinte forma:

$$L^* = L^0 \cup L^1 \cup L^2 \cup \dots = \bigcup_{i=0}^{\infty} L^i$$

EXEMPLO:

$$\begin{aligned}(L_a)^* &= (L_a)^0 \cup (L_a)^1 \cup (L_a)^2 \cup \dots \\ &= (L_a)^0 \cup (L_a)^1 \\ &= L_a \cup \{\lambda\}\end{aligned}$$

Para perceber a simplificação anterior note que $(L_a)^{n+1} \subset (L_a)^n$, $n > 0$.

Capítulo 2

Linguagens Regulares e Expressões Regulares

2.1 Definição de linguagem regular

A classe das **linguagens regulares** sobre o alfabeto A define-se indutivamente da seguinte forma:

1. O conjunto vazio, \emptyset , é uma linguagem regular.
2. Qualquer que seja o $a \in A$, o conjunto $\{a\}$ é uma linguagem regular.
3. Se L_1 e L_2 são linguagens regulares, então $L_1 \cup L_2$ é uma linguagem regular.
4. Se L_1 e L_2 são linguagens regulares, então $L_1.L_2$ é uma linguagem regular.
5. Se L_1 é uma linguagem regular, então $(L_1)^*$ é uma linguagem regular.
6. Nada mais é linguagem regular.

Note que $\{\lambda\}$ é uma linguagem regular, uma vez que $\{\lambda\} = \emptyset^*$.

Exemplo 2.1

Mostre que o conjunto dos números binários começados em 1 e terminados em 0 é uma linguagem regular sobre o alfabeto $A = \{0, 1\}$.

Resposta: O conjunto pretendido pode ser representado por $L = \{1\} \cdot A^* \cdot \{0\}$. Este conjunto pode ser obtido indutivamente da seguinte forma:

1. $\{0\}$ e $\{1\}$ são regulares pela regra 2.
2. $A = \{0, 1\} = \{0\} \cup \{1\}$ é regular por aplicação da regra 3.

3. Se A é regular, A^* também o é por aplicação da regra 5.
4. Finalmente, $\{1\} \cdot A^* \cdot \{0\}$ é regular por aplicação 2 vezes da regra 4.

Exemplo 2.2

Mostre que o conjunto $N = \{0, 1, 2, 3, \dots, 10, \dots\}$, conjunto dos números inteiros positivos, é uma linguagem regular sobre o alfabeto $A = \{0, 1, 2, \dots, 9\}$.

Resposta: O conjunto pretendido corresponde ao conjunto formado pela palavra 0 mais todas as sequência de 1 ou mais dígitos decimais, começadas por um dígito diferente de 0. Ou seja, $N = \{0\} \cup A' \cdot A^*$, onde $A' = \{1, 2, \dots, 9\}$. N pode ser obtido indutivamente da seguinte forma:

1. Qualquer que seja o $d \in A$, $\{d\}$ é uma linguagem regular pela regra 2.
2. $A = \{0, 1, 2, \dots, 9\}$ é uma linguagem regular, por aplicação sucessiva da regra 3.
3. $A' = \{1, 2, \dots, 9\}$ é uma linguagem regular, por aplicação sucessiva da regra 3.
4. Se A é uma linguagem regular, A^* também o é, por aplicação da regra 5.
5. Se A' e A^* são linguagens regulares, $A' \cdot A^*$, é uma linguagem regular por aplicação da regra 4.
6. Finalmente, $N = \{0\} \cup A' \cdot A^*$ é uma linguagem regular por aplicação da regra 3.

2.2 Expressões regulares

O conjunto das **expressões regulares** sobre o alfabeto A define-se indutivamente da seguinte forma:

1. $()$ é uma expressão regular que representa a linguagem regular \emptyset .
2. Qualquer que seja o $a \in A$, a é uma expressão regular que representa a linguagem regular $\{a\}$.
3. Se e_1 e e_2 são expressões regulares representando respectivamente as linguagens regulares L_1 e L_2 , então $(e_1|e_2)$ é uma expressão regular representando a linguagem regular $L_1 \cup L_2$.
4. Se e_1 e e_2 são expressões regulares representando respectivamente as linguagens regulares L_1 e L_2 , então (e_1e_2) é uma expressão regular representando a linguagem regular $L_1 \cdot L_2$.
5. Se e_1 é uma expressão regular representando a linguagem regular L_1 , então e_1^* é uma expressão regular representando a linguagem regular $(L_1)^*$.
6. Nada mais é expressão regular.

Note que é habitual representar-se por λ a expressão regular $()^*$. Esta expressão representa a linguagem regular formada apenas pela palavra vazia, $(\{\lambda\})$.

Exemplo 2.3

Obtenha uma expressão regular que represente o conjunto N do exemplo anterior (exemplo 2).

Resposta: Uma expressão regular que representa N é

$$e = 0|(((((((1|2|3|4|5|6|7|8|9)(((((((0|1|2|3|4|5|6|7|8|9)*$$

Esta expressão tem a forma $e_1|(e_2.e_3^*)$, com $e_1 = 0$, $e_2 = (((((((((1|2|3|4|5|6|7|8|9)$ e $e_3 = ((((((((((0|1|2|3|4|5|6|7|8|9)$). A expressão e_1 representa o elemento 0. A expressão $e_2.e_3^*$ representa as sequências começadas por um dígito diferente de 0. A expressão

$$e = 0|(1|2|\dots|9)(0|1|2|\dots|9)^*$$

representa o mesmo conjunto e é claramente mais legível que a anterior. No entanto, a sua utilização só é válida por causa de propriedades de que gozam os operadores das expressões regulares e que serão apresentados a seguir.

2.3 Propriedades das expressões regulares

A operação de escolha ($|$) goza das propriedades **associativa**, **comutativa**, **existência de elemento neutro** e **idempotência**.

1. As propriedades **associativa** e **comutativa** estabelecem respectivamente que

$$e_1|(e_2|e_3) = (e_1|e_2)|e_3 = e_1|e_2|e_3$$

e que

$$e_1|e_2 = e_2|e_1$$

2. A expressão vazia, representando o conjunto vazio, é o **elemento neutro** da operação de escolha

$$e_1|() = ()|e_1 = e_1$$

3. A operação de escolha goza ainda de **idempotência**

$$e_1|e_1 = e_1$$

A operação de concatenação goza das propriedades **associativa**, **existência de elemento neutro** e **existência de elemento absorvente**.

1. A propriedade **associativa** estabelece que

$$e_1(e_2e_3) = (e_1e_2)e_3 = e_1e_2e_3$$

2. A palavra vazia, representando o conjunto formado simplesmente pela palavra vazia ($\{\lambda\}$) é o **elemento neutro** da concatenação

$$e_1\lambda = \lambda e_1 = e_1$$

3. A expressão vazia, representando o conjunto vazio, é o **elemento absorvente** da concatenação

$$e_1() = ()e_1 = ()$$

Finalmente, os operadores de escolha e concatenação gozam das propriedades **distributiva à esquerda da concatenação em relação à escolha**

$$e_1(e_2|e_3) = e_1e_2|e_1e_3$$

e **distributiva à direita da concatenação em relação à escolha**

$$(e_1|e_2)e_3 = e_1e_3|e_2e_3$$

Note que em geral o fecho de Kleene não goza da propriedade distributiva. Quer isto dizer que em geral

$$(e_1|e_2)^* \neq e_1^*|e_2^*$$

e

$$(e_1e_2)^* \neq e_1^*e_2^*$$

2.4 Simplificação notacional

Na escrita de expressões regulares assume-se que a operação de fecho (*) tem precedência em relação à operação de concatenação e esta tem precedência em relação à operação de escolha (|). O uso destas precedências, aliado às propriedades apresentadas, permite a queda de alguns parêntesis e consequentemente uma notação simplificada. A expressão final do exemplo 3 é um exemplo desta notação simplificada.

Exemplo 2.4

Determine uma expressão regular que represente o conjunto das sequências binárias em que o número de 0 (zeros) é igual a 2.

Resposta: Não fazendo uso da notação simplificada apresentada atrás, obter-se-ia, por exemplo a expressão regular

$$e = (((1^*)0)(1^*))0(1^*)$$

A aplicação das simplificações resulta em

$$1^*01^*01^*$$

Exemplo 2.5

Sobre o alfabeto $A = \{a, b, c\}$ construa uma expressão regular que reconheça a linguagem L , sendo

$$L = \{w \in A^* \mid \#(a, w) = 3\}$$

Resposta: A expressão regular pretendida é

$$e = (b|c)^*a(b|c)^*a(b|c)^*a(b|c)^*$$

2.5 Extensão notacional

Mesmo com as simplificações notacionais apresentadas, por vezes, uma expressão regular pode ser difícil de entender. Isto é particularmente verdade quando os alfabetos de entrada são grandes.

Exemplo 2.6

Repita o exemplo 5 considerando que o alfabeto de entrada é o conjunto das letras minúsculas.

Resposta: Para construir a expressão regular pretendida basta substituir na resposta do exemplo 5 cada ocorrência de $(b|c)$ por $(b|c|d|\dots|y|z)$, obtendo-se

$$e = e_1^* = ((b|c|d|\dots|y|z)^*a(b|c|d|\dots|y|z)^*a(b|c|d|\dots|y|z)^*a(b|c|d|\dots|y|z)^*)^*$$

Embora com um grau de complexidade igual ao do exemplo 5 esta expressão regular é mais difícil de ler.

Considere-se um exemplo onde esta dificuldade de leitura é ainda mais patente.

Exemplo 2.7

Considerando que o alfabeto de entrada é o conjunto das letras maiúsculas e minúsculas, construa uma expressão regular que represente o conjunto das palavras com um número par de vogais.

Resposta: A expressão regular pretendida é dada por

$$((b|c|d|f|\dots|y|z|B|C|D|F|\dots|Y|Z) \\ (a|e|i|o|u|A|E|I|O|U)(b|c|d|f|\dots|y|z|B|C|D|F|\dots|Y|Z) * (a|e|i|o|u|A|E|I|O|U))^*$$

Torna-se, por isso, necessário definir métodos de representação das expressões regulares que possuam maior poder expressivo. Um dos métodos usados faz uso de várias extensões notacionais. Ir-se-ão considerar as seguintes extensões notacionais:

1. O operador $+$ é usado para representar 1 ou mais ocorrências da expressão regular a que se aplica. Assim,

$$e+ \equiv ee^*$$

2. O operador $?$ é usado para representar zero ou uma ocorrência da expressão a que se aplica. Assim,

$$e? \equiv (e|\lambda)$$

3. O símbolo $.$ é usado para representar um símbolo qualquer do alfabeto.¹ Por exemplo, sobre o alfabeto das letras minúsculas,

$$. \equiv (a|b|c|\dots|y|z)$$

4. A expressão $[x_1x_2x_3\dots x_n]$ é usada para representar um (note bem, **um**), símbolo do conjunto $\{x_1, x_2, x_3, \dots, x_n\}$. Por exemplo, a expressão regular $[aeiouAEIOU]$ representa uma vogal.

A construção anterior permite ainda especificar gamas de símbolos usando um intervalo de valores.

A expressão $[x_i - x_f]$ representa um símbolo do alfabeto entre x_i e x_f . Por exemplo,

$$[a-z] \equiv (a|b|c|\dots|y|z)$$

É possível construir expressões juntando gamas e símbolos considerados individualmente. Por exemplo, a expressão $[a-zA-Z0-9_]$ representa uma letra, maiúscula ou minúscula, um algarismo decimal ou o símbolo ‘_’, ou seja, representa um símbolo do conjunto $\{a, b, \dots, y, z, A, B, \dots, Y, Z, 0, 1, \dots, 8, 9, _ \}$.

¹Ferramentas como o `flex` excluem do $.$ a mudança de linha.

5. A expressão $[^{\wedge}x_1x_2x_3\cdots x_n]$ é usada para representar um símbolo do conjunto complementar do conjunto $\{x_1, x_2, x_3, \cdots, x_n\}$. Por exemplo, sobre o alfabeto das letras minúsculas e maiúsculas a expressão $[^{\wedge}aeiouAEIOU]$ representa uma consoante.
6. A expressão $e\{n\}$ representa n concatenações da expressão e , ou seja, e^n .
7. A expressão $e\{n_1, n_2\}$ representa a expressão $e^{n_1}|e^{n_1+1}|\cdots|e^{n_2}$.
8. A expressão $e\{n_1, \}$ representa a expressão $e^{n_1}|e^{n_1+1}|\cdots$.

2.6 Aplicação das expressão regulares

1. Validação.
2. Procura e selecção.
3. *Tokenization* (esquadrinhamento).

2.7 Exercícios

Exercício 2.1

Determine uma expressão regular que represente as sequências binárias com um número par de uns.

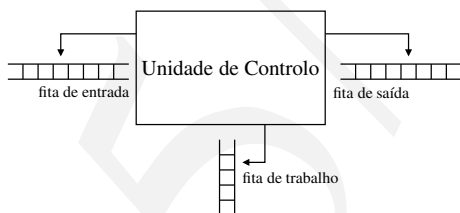
Exercício 2.2

Determine uma expressão regular que represente as constantes numéricas na linguagem C. Eis alguns exemplos de constantes numéricas em C: “10”, “10.1”, “10.”, “.12”, “10.1E+4”.

Capítulo 3

Autómatos Finitos

Um *autômato finito* é um mecanismo reconhecedor das palavras de uma linguagem. Dada uma linguagem L , definida sobre um alfabeto A , um autômato finito reconhecedor de L é um mecanismo que reconhece as palavras de A^* que pertencem a L . Genericamente um autômato finito tem a configuração representada na figura seguinte.



Uma unidade de controlo, com capacidade finita de memorização, manipula os símbolos recolhidos de uma fita de entrada, que armazena uma palavra, e produz uma resposta. Definem-se vários tipos de autómatos, dependendo da forma como a fita de entrada é acedida e da existência ou inexistência de fitas de trabalho e de fita de saída.

A fita de entrada é uma unidade só de leitura com acesso sequencial ou aleatório aos símbolos da palavra. No acesso sequencial um símbolo da palavra de entrada lido e processado não pode ser lido novamente. Se assumirmos que a fita de entrada possui uma cabeça de leitura, esta apenas pode avançar posição a posição. No acesso aleatório o mesmo símbolo pode ser lido mais que uma vez, ou seja, assume-se que a cabeça de leitura pode ser deslocada para a frente ou para trás livremente.

Em geral, a resposta dos autómatos é do tipo sim/não ou aceito/rejeito. Quer isto dizer que um autômato não possui propriamente uma fita de saída, limitando-se a produzir um **aceito** se $u \in L$ e um **rejeito** caso contrário. Neste sentido funcionam efectivamente como *reconhecedores* das palavras de uma linguagem. No entanto, há autómatos que produzem um símbolo de saída por cada símbolo de entrada processado. Nestes casos existe uma fita de saída que representa uma unidade só de escrita com acesso sequencial. Um símbolo de saída uma vez escrito não poderá ser apagado ou rescrito.

Há autómatos em que a unidade de controlo recorre a fitas de trabalho para armazenar informação que auxilie no processamento. Estas fitas têm um funcionamento tipo pilha, ou seja, a ordem de colocação de elementos na fita é contrária à de retirada.

No resto deste capítulo ir-se-á estudar a categoria de autómatos caracterizada pelo facto de possuírem uma fita de entrada com acesso sequencial e por não possuírem fitas de trabalho. Ir-se-á primeiro cobrir os autómatos finitos deterministas e não-deterministas que apenas produzem uma saída do tipo *aceito/rejeito* e depois as máquinas de Moore e as máquinas de Mealy que possuem uma fita de saída.

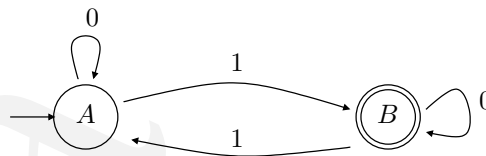
3.1 Autómatos Finitos Deterministas

Um **autômato finito determinista** (AFD) é um quintuplo $M = (A, Q, q_0, \delta, F)$, em que:

- A é o alfabeto de entrada;
- Q é um conjunto finito não vazio de estados;
- $q_0 \in Q$ é o estado inicial;
- $\delta : Q \times A \rightarrow Q$ é uma função que determina a transição entre estados; e
- $F \subseteq Q$ é o conjunto dos estados de aceitação.

Graficamente um AFD é representado por um conjunto de círculos, representando os estados, ligados por arcos, representando as transições. Os círculos correspondentes aos estados de aceitação são diferenciados usando-se riscos duplos. O estado inicial é marcado com uma seta sem origem. Os arcos estão etiquetados com elementos de A .

Considere o autômato finito M_1 representado na figura seguinte.



- A palavra 1011 faz M_1 evoluir de A para B , aceitando-a como sendo uma palavra pertencente à linguagem reconhecida por M_1 .
- A palavra 1100 faz M_1 evoluir para A , levando à sua rejeição.
- A palavra vazia, λ , deixa M_1 em A , levando à sua rejeição.

Na realidade, M_1 reconhece o conjunto das sequências binárias com um número ímpar de uns. Os estados A e B representam respectivamente *número par de uns* e *número ímpar de uns*. Inicialmente o AFD encontra-se em A — a sequência vazia tem um número par de uns. A seguir, por cada 1 que entra o AFD transita entre os estados A e B , de modo a reflectir o número de uns lidos até esse momento. A chegada de zeros não altera o estado. Quando a palavra de entrada se esgotar, se o AFD se encontrar em A é porque tinha um número par de uns e é por isso rejeitada. Se se encontrar em B é aceite.

O AFD pode também ser representado de forma textual. O aspecto mais relevante prende-se com a representação da função de transição, δ . Sendo Q e A conjuntos finitos, δ pode ser representada por um conjunto ou por uma matriz. Usando conjuntos, podemos constatar que $\delta \subseteq Q \times A \times Q$ e por isso representar δ por um conjunto de triplos da forma (q, a, q') , em que $q, q' \in Q$ e $a \in A$. Usando matrizes, podemos usar o domínio, $Q \times A$, para definir as linhas e colunas e preencher as células com as imagens de cada par $(q, a) \in Q \times A$. O AFD do exemplo da figura anterior pode ser textualmente representado por

$$A = \{0, 1\}$$

$$Q = \{A, B\}$$

$$q_0 = A$$

$$F = \{B\}$$

$$\delta = \{(A, 0, A), (A, 1, B), (B, 0, B), (B, 1, A)\}$$

A função de transição δ pode ser representada matricialmente por

	0	1
A	A	B
B	B	A

3.1.1 Linguagem reconhecida por um AFD

Diz-se que um AFD $M = (A, Q, q_0, \delta, F)$, **aceita** uma palavra $u \in A^*$ se u se puder escrever na forma $u = u_1 u_2 \cdots u_n$ e existir uma sequência de estados s_0, s_1, \cdots, s_n , que satisfaça as seguintes condições:

1. $s_0 = q_0$;
2. qualquer que seja o $i = 1, \cdots, n$, $s_i = \delta(s_{i-1}, u_i)$;
3. $s_n \in F$.

Caso contrário diz-se que M **rejeita** a sequência de entrada.

Seja $\delta^* : Q \times A^* \rightarrow Q$ a extensão de δ definida indutivamente por:

$$\delta^*(q, \lambda) = q$$

$$\delta^*(q, av) = \delta^*(\delta(q, a), v), \quad \text{com } a \in A \wedge v \in A^*$$

M aceita u se $\delta^*(q_0, u) \in F$. A linguagem reconhecida por $M = (A, Q, q_0, \delta, F)$ denota-se por $L(M)$ e é definida por

$$L(M) = \{u \in A^* \mid M \text{ aceita } u\} = \{u \in A^* \mid \delta^*(q_0, u) \in F\}$$

Exemplo 3.1

Usando δ^* verifique se a palavra $u = abab$ é aceite pelo autómato $M = (A, Q, q_0, \delta, F)$, definido por:

$$A = \{a, b, c\}$$

$$Q = \{s_1, s_2\}$$

$$q_0 = s_1$$

$$F = \{s_1\}$$

$$\delta = \{(s_1, a, s_2), (s_1, b, s_1), (s_1, c, s_1), \\ (s_2, a, s_1), (s_2, b, s_2), (s_2, c, s_2)\}$$

Resposta: Pretende-se verificar se $\delta^*(s_1, abab) \in F$.

$$\begin{aligned} \delta^*(s_1, abab) &= \delta^*(\delta(s_1, a)bab) = \delta^*(s_2, bab) \\ &= \delta^*(\delta(s_2, b)ab) = \delta^*(s_2, ab) \\ &= \delta^*(\delta(s_2, a)b) = \delta^*(s_1, b) \\ &= \delta^*(\delta(s_1, b)\lambda) = \delta^*(s_1, \lambda) \\ &= s_1 \in F. \end{aligned}$$

Logo, M aceita a palavra $abab$.

3.1.2 Projecto de um AFD

«O projecto de um autómato finito é um processo criativo; como tal não se pode reduzir a uma receita ou fórmula.» Pode, no entanto, ajudar se se seguir os seguintes passos:

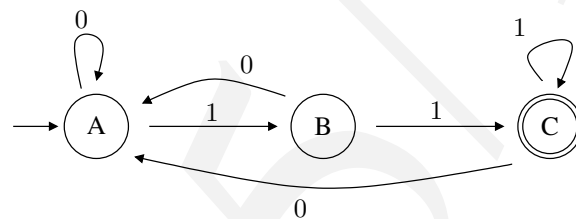
1. Identificar os estados necessários.
2. Acrescentar as transições.
3. Identificar e marcar o estado inicial.
4. Identificar e marcar os estados de aceitação.

Mas, frequentemente, é preciso iterar diversas vezes sobre estes passos antes de se chegar à solução do problema.

Exemplo 3.2

Projecte um AFD que reconheça as sequências binárias terminadas em 11.

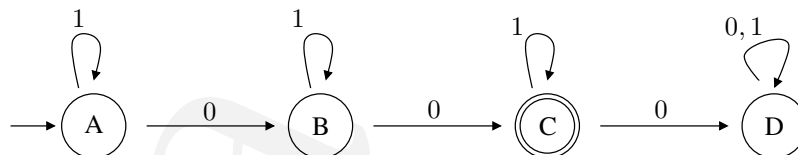
Resposta: Depois de alguma reflexão chega-se à conclusão de que são necessários 3 estados, *A*, *B* e *C*, significando respectivamente último dígito que entrou não é 1, último dígito que entrou é 1 mas o anterior não e dois últimos dígitos entrados são 1. A partir daqui chega-se facilmente ao seguinte autómato



Exemplo 3.3

Projecte um AFD que reconheça as sequências binárias com dois zeros.

Resposta: Na figura seguinte apresenta-se graficamente o AFD pretendido.

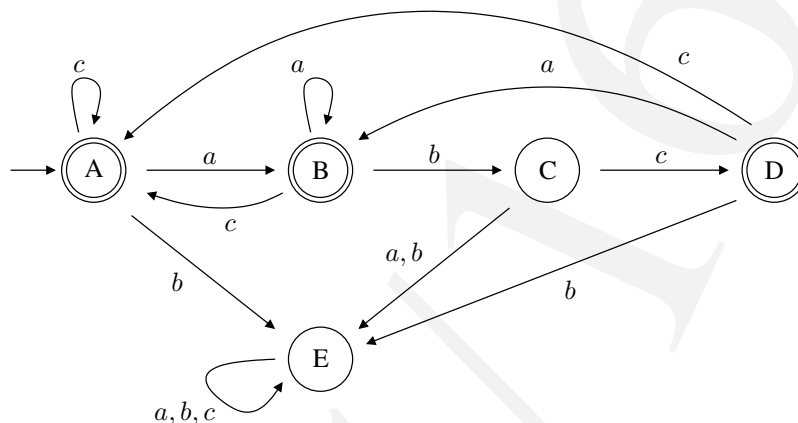


São necessários 4 estados para representar a chegada de 0, 1, 2 e mais de 2 zeros. Note que do estado *D* saem dois arcos dirigidos ao próprio estado *D*, um com a etiqueta 0 e outro com a etiqueta 1. Por uma questão de simplificação notacional, é costume fundir-se os dois arcos num só, com as etiquetas separadas por vírgulas. Note ainda que num autómato finito determinista de cada estado deve sair um arco etiquetado com cada um dos símbolos do alfabeto, mesmo que esses arcos já não possam conduzir a situações de aceitação. É o caso dos arcos saídos do estado *D*.

Exemplo 3.4

Projecte um AFD que reconheça as sequências definidas sobre o alfabeto $A = \{a, b, c\}$ que satisfazem o requisito de qualquer b ter um a imediatamente à sua esquerda e um c imediatamente à sua direita.

Resposta: À partida podemos considerar a existência de 4 estados para detectar a sequência abc . Isto corresponde aos estados A, B, C e D da figura abaixo e aos arcos entre eles indo da esquerda para a direita. Basta que um b não satisfaça o requisito imposto para que a sequência seja rejeitada. O estado E captura essas sequências.



Note que os estados A, B e D são de aceitação. O estado C não o é porque se o autômato termina aí, a sequência de entrada termina em b e, por conseguinte, existe um b que não tem um c imediatamente à sua direita. Veremos adiante que os estados A e D são equivalentes podendo ser fundidos. No entanto, isso não invalida que o AFD dado esteja correcto.

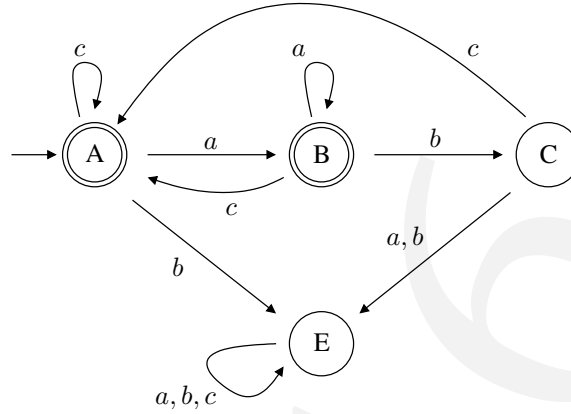
Exercício 3.1

Projecte uma AFD que reconheça sequências binárias de comprimento ímpar terminadas em 0 ou de comprimento par terminadas em 1.

3.1.3 Redução de AFD

Considere o autômato do exemplo anterior (exemplo 4). Os estados A e D são equivalentes. Por um lado, a sequência de entrada é aceite se o autômato termina nos estados A ou D . Por outro lado, se o autômato se encontra nos estados A ou D evolui, em ambos os casos, para o estado B se entra um a , para o E se entra um b e para o A se entra um c . Ou seja, enviar o autômato para o estado D é equivalente a enviá-lo para o estado A , podendo, por isso, um ser substituído pelo outro. Se desviarmos para A o único arco dirigido para D — vindo de C com etiqueta c —, o estado D deixa de ser alcançável a partir

do estado inicial, podendo ser eliminado. O autómato transforma-se então no autómato da figura abaixo, que lhe é equivalente.



Em geral, dois estados, s_i e s_j , de um autómato $M = (A, Q, q_0, \delta, F)$ são equivalentes se e só se

$$\forall u \in A^* \quad \delta^*(s_i, u) \in F \Leftrightarrow \delta^*(s_j, u) \in F$$

A notação $s_i \equiv s_j$ é usada para representar o facto de s_i e s_j serem equivalentes. O conjunto de todos os estados equivalentes a s é denotado por $[s]$ e representa uma classe de equivalência. A relação de equivalência (\equiv) entre todos os estados de um autómato determinista permite definir o seu equivalente **reduzido**. Seja $M = (A, Q, q_0, \delta, F)$ uma autómato determinista qualquer. O equivalente reduzido de M é o AFD $M' = (A, Q', q'_0, \delta', F')$ definido da seguinte maneira:

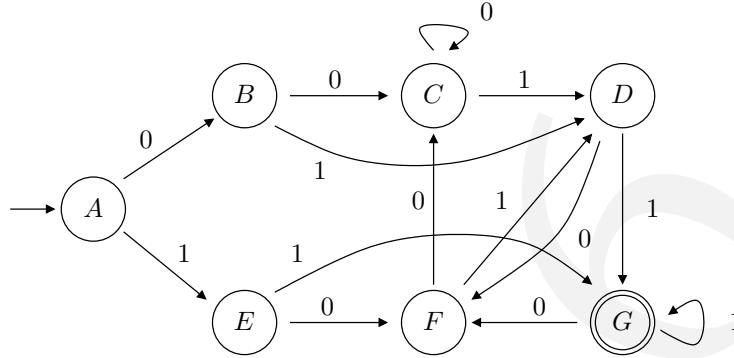
- $Q' = \{[q] \mid q \in Q\}$
- $q'_0 = [q_0]$
- $F' = \{[q] \mid q \in F\}$
- $\delta' : Q' \times A \rightarrow Q'$ tal que $\delta'([q], a) = [\delta(q, a)]$.

Algoritmo de redução de AFD

O algoritmo de redução usa um método de aproximações sucessivas a partir de uma distribuição inicial dos estados em duas classes de equivalência, uma com os estados de aceitação e outra com os restantes. Para uma dada partição, uma classe de equivalência C é válida se para qualquer elemento a do alfabeto e quaisquer dois estados q_1 e q_2 pertencentes a C , $[\delta(q_1, a)] = [\delta(q_2, a)]$. Sempre que numa dada partição uma classe viole a condição anterior desdobra-se em duas ou mais classes e repete-se o processo, até atingir uma solução, que existe sempre.

Exemplo 3.5

Considere o autômato



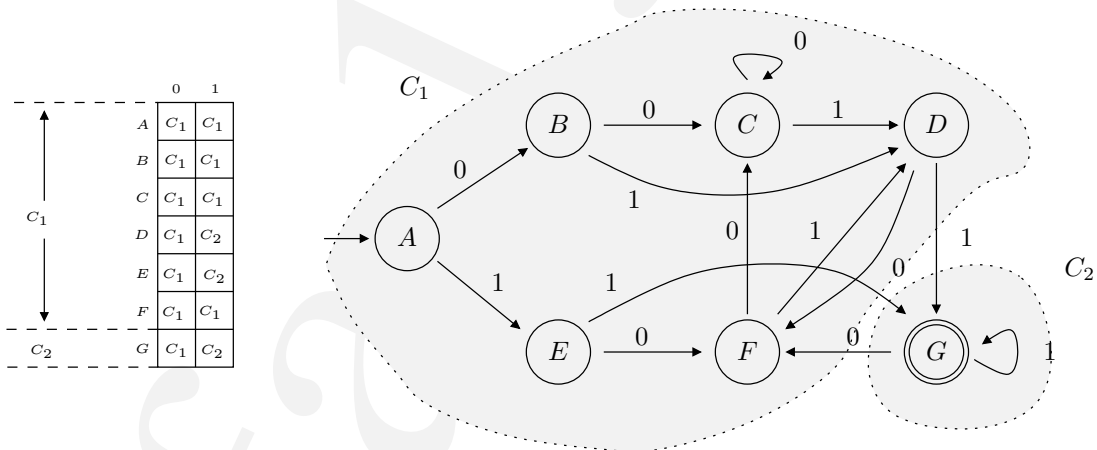
que aceita seqüências binárias terminadas em 11, sendo, por isso, equivalente ao autômato apresentado no exemplo 2. Construa-se o seu equivalente reduzido.

Resposta: Primeiro, definem-se duas classes de equivalência, uma com os estados de aceitação e outra com os restantes

$$C_1 = Q - F = \{A, B, C, D, E, F\}$$

$$C_2 = F = \{G\}$$

e constroi-se um grafo cujo conjunto de nós Q' é o conjunto das classes de equivalência e cujos arcos definem a função $Q \times A \rightarrow Q'$. Esta função mapeia cada par $(q, a) \in Q \times A$ na classe de equivalência que contém $\delta(q, a)$.



Neste grafo, as classes com arcos com a mesma etiqueta a dirigirem-se a classes diferentes têm de ser desdobradas, porque estão mal definidas. No caso do exemplo, o conjunto C_1 precisa de ser desdobrado: há arcos com etiqueta 1 que se dirigem à própria classe e outro que vai para a classe C_2 . Olhando para a tabela verifica-se que as linhas referentes a C_1 caem em duas categorias, uma

com os estados A, B, C e F e outra com os estados D e E . Dividimos então a classe C_1 em duas classes que representam estas duas categorias, passando a ter 3 classes de equivalência

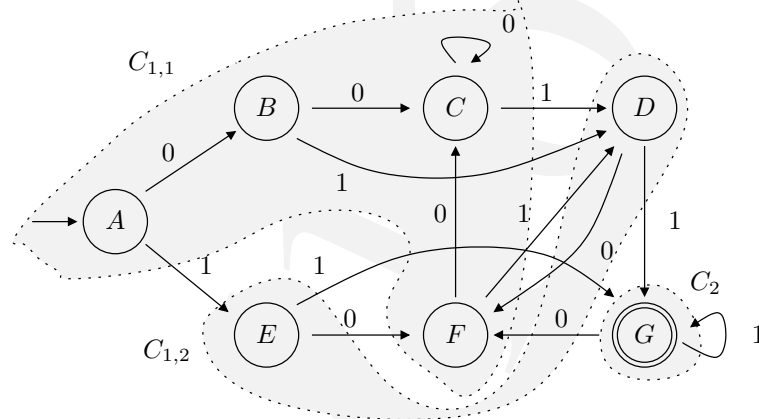
$$C_{1.1} = \{A, B, C, F\}$$

$$C_{1.2} = \{D, E\}$$

$$C_2 = \{G\}$$

Reconstruímos a tabela e o grafo

	0	1
$C_{1.1}$		
A	$C_{1.1}$	$C_{1.2}$
B	$C_{1.1}$	$C_{1.2}$
C	$C_{1.1}$	$C_{1.2}$
F	$C_{1.1}$	$C_{1.2}$
$C_{1.2}$		
E	$C_{1.1}$	C_2
D	$C_{1.1}$	C_2
C_2		
G	$C_{1.1}$	C_2

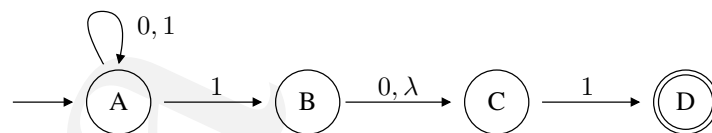


obtendo-se um grafo que representa o autômato equivalente reduzido pretendido. Se ainda houvesse classes mal definidas repetir-se-ia o processo até se atingir uma situação correcta.

O algoritmo encontra sempre uma solução. Se se tentar reduzir um autômato finito determinista já reduzido, o processo de redução conduzirá ao próprio autômato.

3.2 Autômatos Finitos Não Deterministas

Considere o seguinte autômato sobre o alfabeto $A = \{0, 1\}$.



Possui 3 características não válidas nos autômatos finitos definidos anteriormente:

- Não há nenhum arco etiquetado com 1 a sair dos estados B e D , nem nenhum arco etiquetado com 0 a sair dos estados C e D .
- Há dois arcos etiquetados com 1 a sair do estado A , um para A e outro para B .

- Há um arco etiquetado com λ , a palavra vazia, a sair do estado B .

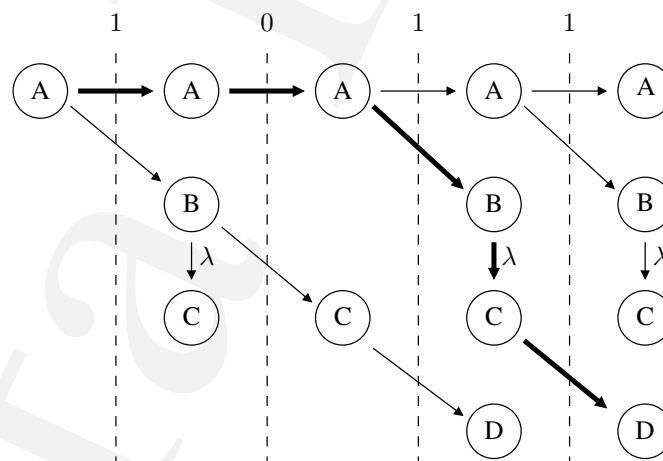
Estes arcos violam a definição de autómato finito determinista, mas são aceites pelos autómatos finitos não deterministas (AFND), que permitem que de cada estado possam sair zero ou mais arcos etiquetados com cada um dos símbolos do alfabeto de entrada ou com a palavra vazia. A multiplicidade permite que para fazer aceitar uma palavra se possa escolher entre caminhos alternativos. Ou seja, uma palavra é aceite pelo autómato se houver um caminho que conduza do estado inicial a um estado de aceitação. Assim, perante a entrada 1011 o autómato anterior pode realizar 4 caminhos alternativos:

$$\begin{aligned} A &\xrightarrow{1} A \xrightarrow{0} A \xrightarrow{1} A \xrightarrow{1} A \\ A &\xrightarrow{1} A \xrightarrow{0} A \xrightarrow{1} A \xrightarrow{1} B \\ A &\xrightarrow{1} A \xrightarrow{0} A \xrightarrow{1} A \xrightarrow{1} B \xrightarrow{\lambda} C \\ A &\xrightarrow{1} A \xrightarrow{0} A \xrightarrow{1} B \xrightarrow{\lambda} C \xrightarrow{1} D \end{aligned}$$

um dos quais, o último, conduz ao estado de aceitação. Logo, a palavra 1011 é aceite. Este autómato reconhece todas as sequências binárias terminadas em 11 ou 101.

3.2.1 Árvore de caminhos de um AFND

Uma forma simples de avaliar os diversos caminhos (percursos) que um autómato não determinista pode realizar em resposta a uma dada palavra à entrada obtém-se traçando a *árvore de percursos*. Corresponde a um grafo acíclico, em forma de árvore, tendo como raiz o estado inicial e onde se representam todos os estados alcançáveis por ocorrência da palavra de entrada. Na figura seguinte mostra-se a árvore de caminhos que representa a resposta do autómato anterior à palavra 1011.



A árvore de caminhos é uma estrutura em camadas alcançadas por força da ocorrência de cada símbolo da palavra de entrada. Uma palavra é aceite se o conjunto de estados da última camada contiver pelo menos um estado de aceitação.

3.2.2 Definição de autómato finito não determinista

Um **autómato finito não determinista** (AFND) é um quintuplo $M = (A, Q, q_0, \delta, F)$, em que:

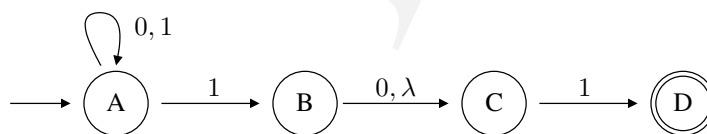
- A é o alfabeto de entrada;
- Q é um conjunto finito não vazio de estados;
- $q_0 \in Q$ é o estado inicial;
- $\delta \subseteq (Q \times A_\lambda \times Q)$ é a relação de transição entre estados, com $A_\lambda = A \cup \{\lambda\}$; e
- $F \subseteq Q$ é o conjunto dos estados de aceitação.

A diferença relativamente à definição de autómato finito determinista verifica-se na relação δ . Por um lado, permite que os arcos (transições) sejam etiquetados com λ . Por outro lado, pelo facto de δ ser uma relação e não uma função, permite que o mesmo par $(q, a) \in Q \times A_\lambda$ possa ter 0 ou mais imagens.

Alternativamente, é possível definir-se δ como sendo uma função que aplica o conjunto $Q \times A_\lambda$ em $\wp(Q)$, onde $\wp(Q)$ representa o conjunto dos subconjuntos de Q .

Exemplo 3.6

Represente analiticamente o AFND



Resposta: O AFND dado pode-se representar textualmente por

$$Q = \{A, B, C, D\}$$

$$q_0 = A$$

$$F = \{D\}$$

$$\delta = \{(A, 0, A), (A, 1, A), (A, 1, B), (B, \lambda, C), (B, 0, B), (C, 1, D)\}$$

Note que a existência dos elementos $(A, 1, A)$ e $(A, 1, B)$ faz com que δ não seja uma função.

Se δ fosse definida na forma de função ($\delta : Q \times A_\lambda \rightarrow \wp(Q)$), ter-se-ia:

$$\delta = \{ \begin{array}{lll} (A, 0) \rightarrow \{A\}, & (A, 1) \rightarrow \{A, B\}, & (A, \lambda) \rightarrow \emptyset, \\ (B, 0) \rightarrow \{B\}, & (B, 1) \rightarrow \emptyset, & (B, \lambda) \rightarrow \{C\}, \\ (C, 0) \rightarrow \emptyset, & (C, 1) \rightarrow \{D\}, & (C, \lambda) \rightarrow \emptyset, \\ (D, 0) \rightarrow \emptyset, & (D, 1) \rightarrow \emptyset, & (D, \lambda) \rightarrow \emptyset \end{array} \}$$

Note que sendo δ definida como uma função é preciso apresentar as imagens para todos os elementos de $Q \times A_\lambda$, mesmo que sejam o conjunto vazio. Em geral, torna-se mais clara a representação usando a relação de transição.

3.2.3 Linguagem reconhecida por um AFND

Diz-se que um AFND $M = (A, Q, q_0, \delta, F)$, **aceita** uma palavra $u \in A^*$ se u se puder escrever na forma $u = u_1 u_2 \cdots u_n$, com $u_i \in A_\lambda$, e existir uma sequência de estados s_0, s_1, \dots, s_n , que satisfaça as seguintes condições:

1. $s_0 = q_0$;
2. qualquer que seja o $i = 1, \dots, n$, $(s_{i-1}, u_i, s_i) \in \delta$;
3. $s_n \in F$.

Caso contrário diz-se que M **rejeita** a entrada. Note que n pode ser maior que $|u|$, porque alguns dos u_i podem ser λ .

A linguagem reconhecida por M denota-se por $L(M) = \{u \mid M \text{ aceita } u\}$. Usar-se-á a notação $q_i \xrightarrow{u} q_j$ para representar a existência de uma palavra u que conduza do estado q_i ao estado q_j . Usando esta notação tem-se $L(M) = \{u \mid q_0 \xrightarrow{u} q_f \wedge q_f \in F\}$.

3.3 Equivalência entre AFD e AFND

A definição de AFND incorpora a definição de AFD, pelo que qualquer AFD é por definição um AFND. Na verdade, na definição de AFD, δ é uma função que aplica $Q \times A$ em Q , logo $\delta \subset (Q \times A \times Q)$. Mas, $(Q \times A \times Q) \subset (Q \times A_\lambda \times Q)$, pelo que a função δ dos AFD é um subconjunto da relação δ dos AFND.

Prova-se também que dado um qualquer AFND é possível construir um AFD que reconhece exactamente a mesma linguagem. Olhando para uma árvore de caminhos, constata-se que por acção de um símbolo à entrada o autómato não determinista avança de um subconjunto de estados para outro, sendo que

inicialmente se encontra no subconjunto constituído por apenas o seu estado inicial. É assim possível transformar um AFND num AFD cujos estados são subconjuntos de estados do AFND. Se o AFND tiver n estados, haverá 2^n subconjuntos de estados, pelo que o AFD equivalente terá no máximo 2^n estados. Os estados de aceitação do AFD serão todos aqueles que contenham estados de aceitação do AFND.

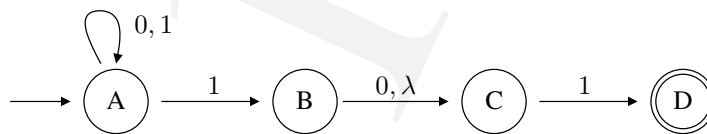
A estrutura de construção da árvore de caminhos permite obter a resposta de um AFND a cada símbolo do alfabeto de entrada, considerando que o AFND se encontra hipoteticamente num dado subconjunto de estados. É assim possível, dado um AFND $M = (A, Q, q_0, \delta, F)$, definir-se a função $\Delta : \wp(Q) \times A \rightarrow \wp(Q)$, que representa as transições do autómato em termos de subconjuntos de estados. O autómato finito determinista $M' = (A, Q', q'_0, \delta', F')$ onde:

1. $Q' = \wp(Q)$;
2. $\delta' = \Delta$;
3. q'_0 é o subconjunto de Q constituído pelo estado inicial de M mais todos os alcançáveis a partir dele por ocorrências de palavras vazias (λ); e
4. F' é o conjunto dos subconjuntos de Q que contêm elementos de F , isto é, $f' \in \wp(Q)$ é um elemento de F' se e só se $f' \cap F \neq \emptyset$.

é um autómato finito determinista equivalente a M .

Exemplo 3.7

Usando o método anterior, construa-se um AFD equivalente ao AFND apresentado no exemplo 6, cuja representação gráfica se repete aqui por comodidade.



Resposta: Seja $M' = (A, Q', q'_0, \delta', F')$ o autómato pretendido. Tem-se

$$Q' = \{X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}\}$$

onde

$$\begin{array}{llll}
 X_0 = \{\} & X_1 = \{A\} & X_2 = \{B\} & X_3 = \{A, B\} \\
 X_4 = \{C\} & X_5 = \{A, C\} & X_6 = \{B, C\} & X_7 = \{A, B, C\} \\
 X_8 = \{D\} & X_9 = \{A, D\} & X_{10} = \{B, D\} & X_{11} = \{A, B, D\} \\
 X_{12} = \{C, D\} & X_{13} = \{A, C, D\} & X_{14} = \{B, C, D\} & X_{15} = \{A, B, C, D\}
 \end{array}$$

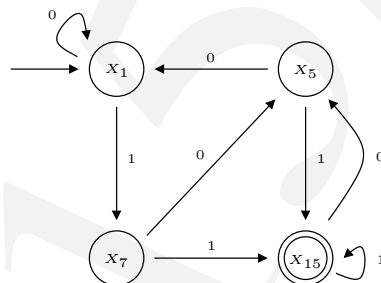
Tem-se ainda $q'_0 = X_1$ e

$$F' = \{X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}\}$$

Finalmente, a função δ' é definida pela tabela seguinte

estado	0	1	estado	0	1
X_0	X_0	X_0	X_1	X_1	X_7
X_2	X_4	X_0	X_3	X_5	X_7
X_4	X_0	X_8	X_5	X_1	X_{15}
X_6	X_4	X_8	X_7	X_5	X_{15}
X_8	X_0	X_0	X_9	X_1	X_7
X_{10}	X_4	X_0	X_{11}	X_5	X_7
X_{12}	X_0	X_8	X_{13}	X_1	X_{15}
X_{14}	X_4	X_8	X_{15}	X_5	X_{15}

É fácil constatar que a partir do estado inicial X_1 apenas os estados X_7 , X_5 e X_{15} são alcançáveis, pelo que os restantes podem ser retirados do autómato. Obtém-se assim um AFD, representado graficamente abaixo, com apenas 4 estados, em vez dos 16 inicialmente considerados.



A obtenção de um AFD equivalente sem estados não alcançáveis pode fazer-se por um método construtivo, tendo por base o estado inicial e a função Δ definida anteriormente. A ideia é ir contruindo a função de transição δ' e o conjunto de estados Q' a partir de um conjunto contendo inicialmente apenas o estado inicial do AFD pretendido. Calculando a imagem dos elementos de Q' para cada símbolo de A vão aparecendo novos elementos que são acrescentados a Q' . O processo repete-se até que todos os elementos de Q' estejam tratados e nenhum novo surja. Clarifica-se o exposto através de um exemplo.

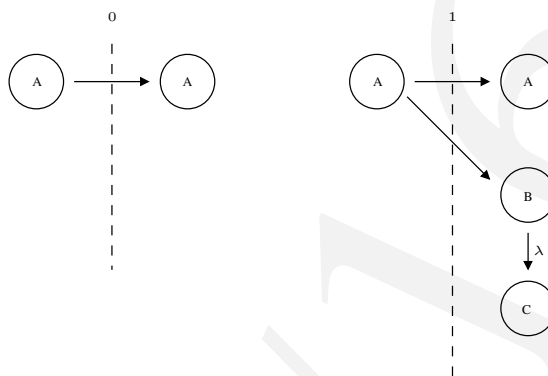
Exemplo 3.8

Usando o método construtivo anterior, construa-se um AFD equivalente ao AFND apresentado no exemplo 6.

Resposta: Define-se primeiro o estado inicial do AFD pretendido.

$$X_1 = \{A\}$$

Não se dispõe da função Δ . Por isso, para determinar os estados alcançáveis a partir de X_1 por ocorrência dos símbolos do alfabeto de entrada, calculam-se as árvores de caminhos do AFND às palavras 0 e 1. Obtém-se



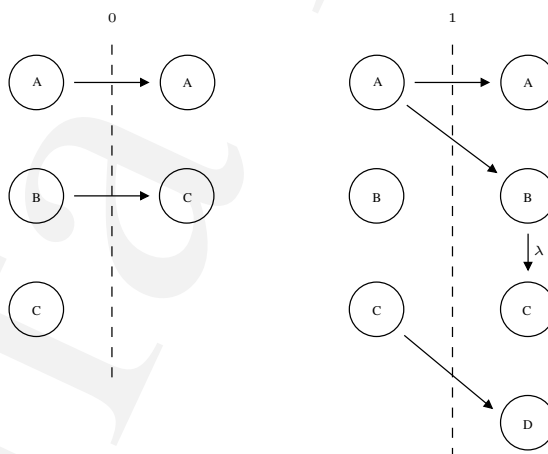
o que resulta em

estado	0	1
X_1	X_1	$X_7 = \{A, B, C\}$

o que introduz um novo estado

$$X_7 = \{A, B, C\}$$

Calculando as árvores para X_7 obtém-se

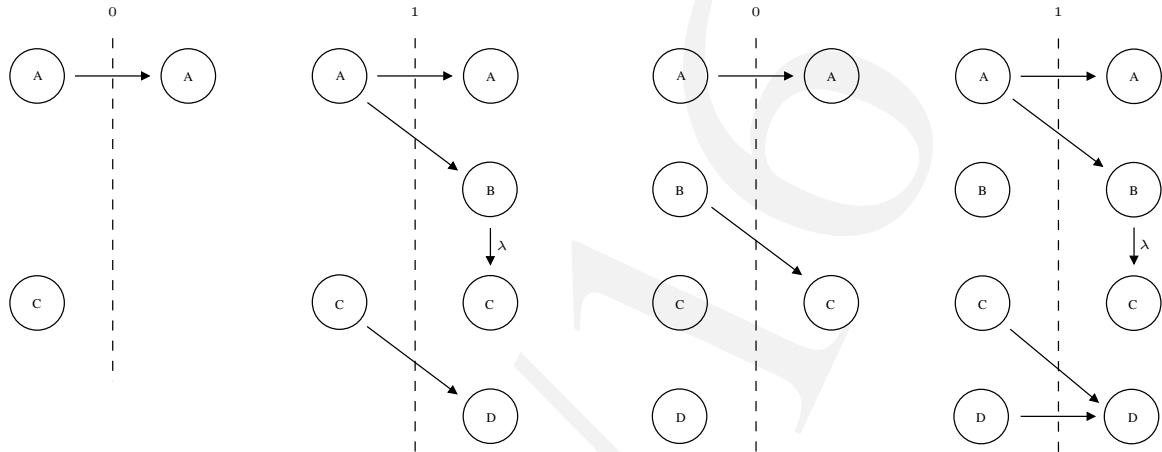


o que introduz os estados

$$X_5 = \{A, C\}$$

$$X_{15} = \{A, B, C, D\}$$

Repetindo para X_5 e X_{15} obtém-se as árvores de caminhos



que não introduzem nenhum novo estado. Tem-se, portanto, um AFD com 4 estados. Os estados de aceitação são aqueles que contêm D , logo, apenas X_{15} . Pode constatar que o AFND obtido corresponde ao apresentado no fim do exemplo anterior.

3.4 Operações sobre AFD e AFND

A classe das linguagens regulares é fechada sob as operações de **reunião**, **concatenação**, **fecho de Kleene**, **complementação**, **diferença** e **intersecção**. Quer isto dizer que se M_1 e M_2 são dois autómatos quaisquer, reconhecendo respectivamente as linguagens $L(M_1)$ e $L(M_2)$, existem autómatos para reconhecer as linguagens $L(M_1) \cup L(M_2)$, $L(M_1)L(M_2)$, $L(M_1)^*$, $\overline{L(M_1)}$, $L(M_1) - L(M_2)$ e $L(M_1) \cap L(M_2)$.

3.4.1 Reunião de autómatos

Dados dois autómatos M_1 e M_2 construa-se um autômato M tal que $L(M) = L(M_1) \cup L(M_2)$, i.e., que a linguagem reconhecida por M seja a **reunião** das linguagens reconhecidas por M_1 e M_2 .

Seja $M_1 = (A, Q_1, q_1, \delta_1, F_1)$ e $M_2 = (A, Q_2, q_2, \delta_2, F_2)$ dois autómatos (AFD ou AFND) quaisquer.

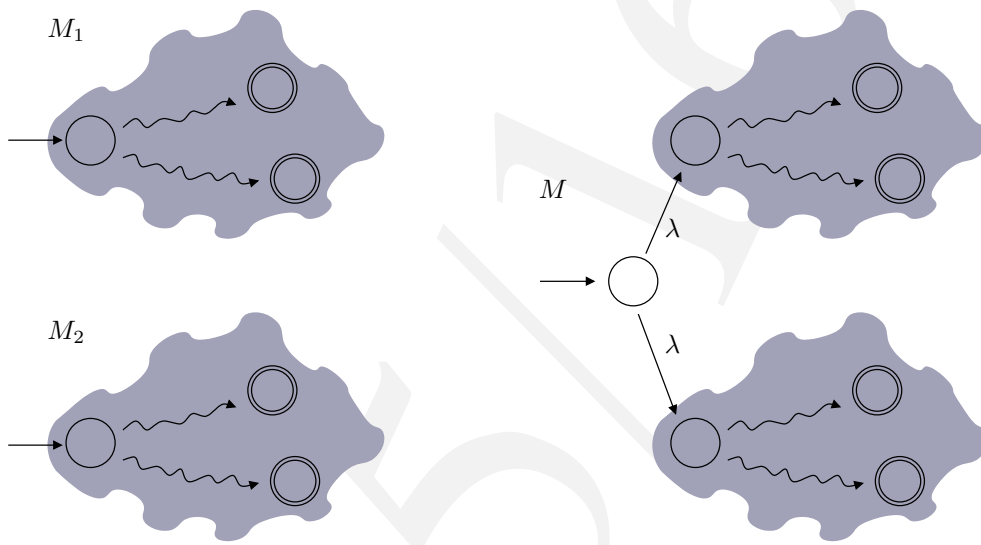
O AFND $M = (A, Q, q_0, \delta, F)$, onde

$$Q = Q_1 \cup Q_2 \cup \{q_0\}, \quad \text{com } q_0 \notin Q_1 \wedge q_0 \notin Q_2$$

$$F = F_1 \cup F_2$$

$$\delta = \delta_1 \cup \delta_2 \cup \{(q_0, \lambda, q_1), (q_0, \lambda, q_2)\}$$

satisfaz o objectivo pretendido. Antes de se provar a afirmação anterior, veja-se graficamente a construção de M a partir de M_1 e M_2 .



Por um lado, é intuitivo que qualquer sequência u aplicada a M , segue não deterministicamente os caminhos de M_1 e de M_2 . Logo, se u é aceite por um deles também o é por M . Por outro lado, também é intuitivo que M só aceita sequências que sejam aceites por M_1 ou por M_2 . Mas, prove-se formalmente esta equivalência.

Pretende-se provar que $L(M) = L(M_1) \cup L(M_2)$. Para isso basta provar que

$$L(M) \subseteq L(M_1) \cup L(M_2)$$

e que

$$L(M_1) \cup L(M_2) \subseteq L(M)$$

A primeira condição corresponde a provar que

$$u \in L(M) \Rightarrow (u \in L(M_1) \vee u \in L(M_2))$$

o que se obtém do seguinte desenvolvimento

$$\begin{aligned}
 u \in L(M) &\Rightarrow q_0 \xrightarrow{u} q_f, \quad \text{com } q_f \in F \\
 &\Rightarrow q_0 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_1 \cup F_2 \\
 &\Rightarrow (q_0 \xrightarrow{\lambda} q_1 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_1) \vee (q_0 \xrightarrow{\lambda} q_2 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_2) \\
 &\Rightarrow (q_1 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_1) \vee (q_2 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_2) \\
 &\Rightarrow u \in L(M_1) \vee u \in L(M_2)
 \end{aligned}$$

A segunda condição corresponde a provar, para $i = 1, 2$, que

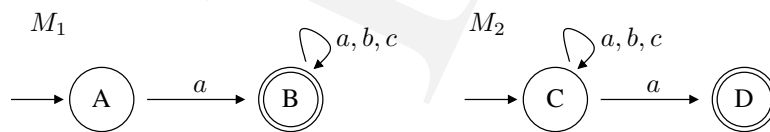
$$u \in L(M_i) \Rightarrow u \in L(M)$$

o que se obtém do seguinte desenvolvimento

$$\begin{aligned}
 u \in L(M_i) &\Rightarrow q_i \xrightarrow{u} q_f, \quad \text{com } q_f \in F_i \\
 &\Rightarrow q_0 \xrightarrow{\lambda} q_i \xrightarrow{u} q_f, \quad \text{com } q_f \in F_i \\
 &\Rightarrow q_0 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_i \\
 &\Rightarrow q_0 \xrightarrow{u} q_f, \quad \text{com } q_f \in F \\
 &\Rightarrow u \in L(M)
 \end{aligned}$$

Exemplo 3.9

A figura



representa graficamente os autômatos M_1 e M_2 , definidos sobre o alfabeto $A = \{a, b, c\}$, que reconhecem respectivamente as linguagens

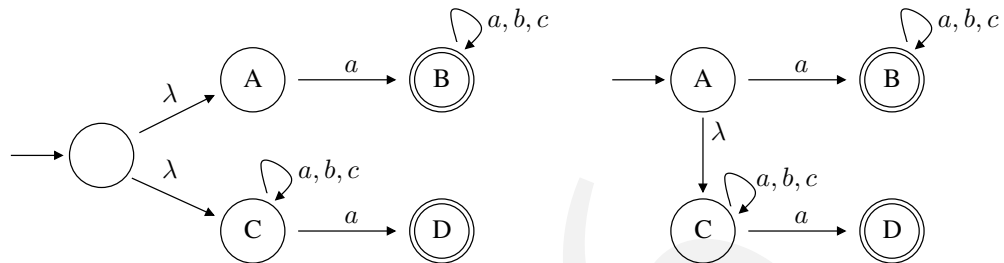
$$L_1 = \{aw | w \in A^*\} = \{w \in A^* | w \text{ começa por } a\}$$

e

$$L_2 = \{wa | w \in A^*\} = \{w \in A^* | w \text{ termina por } a\}$$

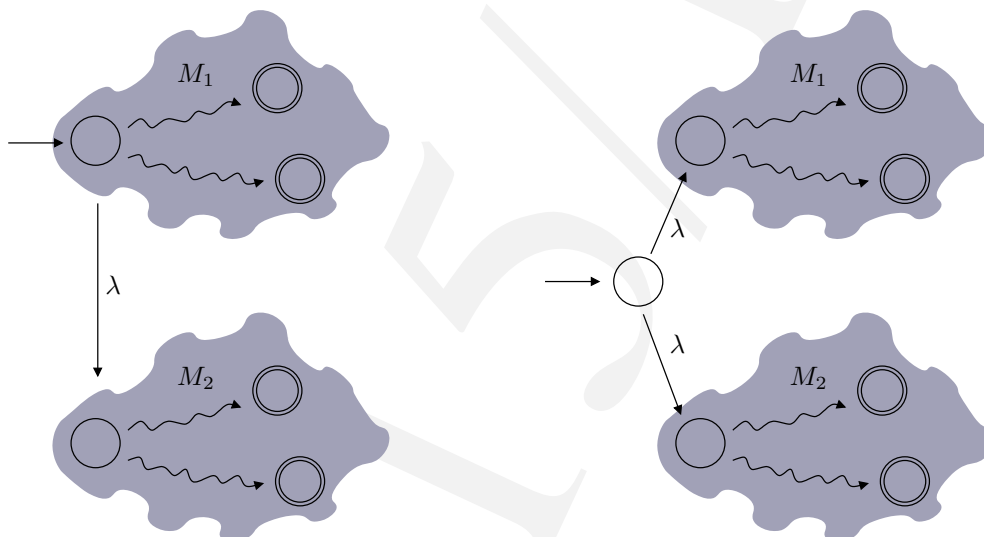
Usando a operação de reunião sobre autômatos determine o autômato M que reconhece a linguagem das palavras começadas ou terminadas por a .

Resposta: A aplicação directa do algoritmo de reunião resulta no autómato da figura abaixo, lado esquerdo. Alguma manipulação permite transformá-lo no da direita.



Exercício 3.2

Mostre que as duas construções definidas à volta dos autómatos M_1 e M_2 (a sombreado) apresentadas na figura seguinte só são equivalente se o estado inicial de M_1 não tiver arcos de chegada.



Considere que M_1 e M_2 são autómatos definidos sobre o alfabeto binário, reconhecendo respectivamente as sequências com número par de uns e ímpar de zeros. Mostre que a construção da esquerda reconhece as sequências com número ímpar de uns e par de zeros, que são rejeitadas pela construção da direita. Qual é a linguagem reconhecida pela construção da esquerda?

3.4.2 Concatenação de autómatos

Dados dois autómatos M_1 e M_2 construa-se um autómato M tal que $L(M) = L(M_1).L(M_2)$, i.e., que a linguagem reconhecida por M seja a **concatenação** da linguagem reconhecida por M_1 com a linguagem reconhecida por M_2 .

Seja $M_1 = (A, Q_1, q_1, \delta_1, F_1)$ e $M_2 = (A, Q_2, q_2, \delta_2, F_2)$ dois autómatos (AFD ou AFND) quaisquer. O AFND $M = (A, Q, q_0, \delta, F)$, onde

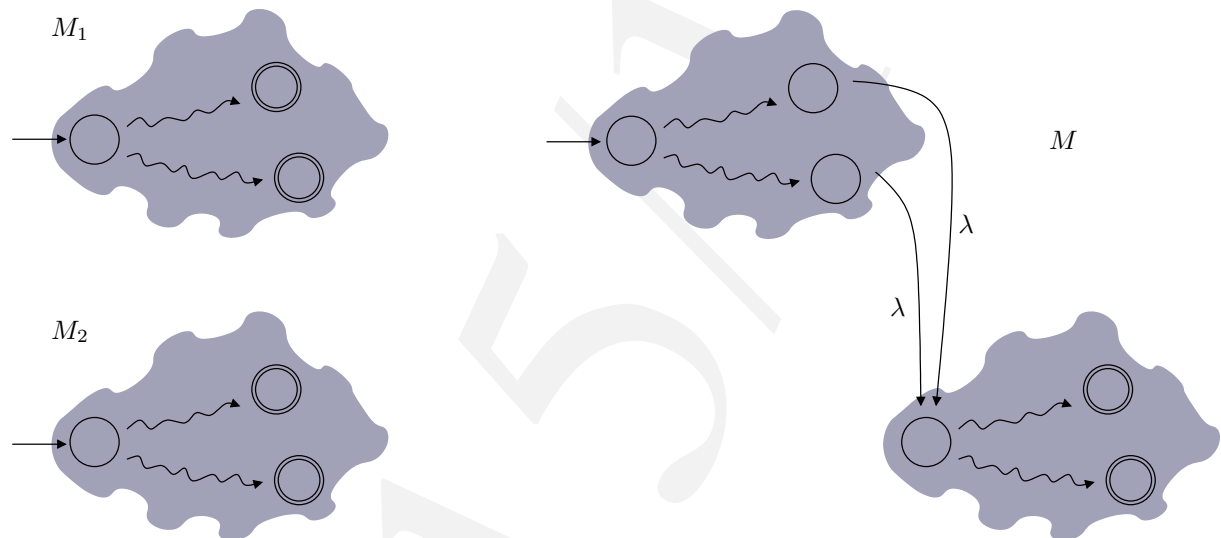
$$Q = Q_1 \cup Q_2$$

$$q_0 = q_1$$

$$F = F_2$$

$$\delta = \delta_1 \cup \delta_2 \cup (F_1 \times \{\lambda\} \times \{q_2\})$$

satisfaz o objectivo pretendido. Antes de se provar a afirmação anterior, veja-se graficamente a construção de M a partir de M_1 e M_2 .



Para que M aceite uma palavra u é necessário percorrer um caminho que vá desde o seu estado inicial até um dos seus estados de aceitação. Ora, isto significa ir do estado inicial até um estado de aceitação de M_1 , passando-se, de seguida e por efeito de um λ , para o estado inicial de M_2 e finalmente indo deste até um dos estados de aceitação de M_2 . Ou seja, uma palavra u é aceite por M se se puder decompor u em duas partes v e w ($u = vw$), sendo v aceite por M_1 e w por M_2 . Prove-se formalmente esta equivalência.

Pretende-se provar que $L(M) = L(M_1)L(M_2)$. Para isso basta provar que

$$L(M) \subseteq L(M_1)L(M_2)$$

e que

$$L(M_1)L(M_2) \subseteq L(M)$$

A primeira condição corresponde a provar que

$$u \in L(M) \Rightarrow (u \in L(M_1)L(M_2))$$

o que se obtém do seguinte desenvolvimento

$$u \in L(M) \Rightarrow q_0 \xrightarrow{u} q_f, \quad \text{com } q_f \in F$$

$$u \in L(M) \Rightarrow q_0 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_2$$

$$\Rightarrow q_0 \xrightarrow{v} q_v \xrightarrow{\lambda} q_2 \xrightarrow{w} q_f, \quad \text{com } u = vw \wedge q_v \in F_1 \wedge q_f \in F_2$$

$$\Rightarrow v \in L(M_1) \wedge w \in L(M_2), \quad \text{com } u = vw$$

$$\Rightarrow u \in L(M_1)L(M_2)$$

Exercício 3.3

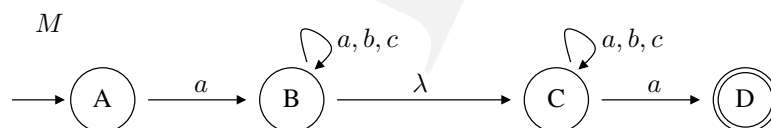
Prove a segunda condição, isto é, prove que

$$u \in L(M_1)L(M_2) \Rightarrow u \in L(M)$$

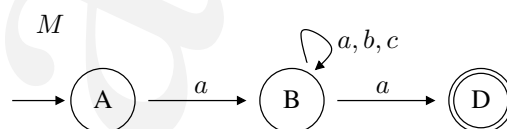
Exemplo 3.10

Usando o algoritmo apresentado determine o autómato M que reconhece a linguagem $L(M_1).L(M_2)$, considerando os autómatos M_1 e M_2 do exemplo 9.

Resposta: A aplicação directa do algoritmo de concatenação resulta no autómato da figura abaixo



que pode ser simplificado para o seguinte



Exercício 3.4

Considerando os autómatos do exemplo anterior determine o autômato que reconhece a linguagem $L(M_2).L(M_1)$.

3.4.3 Fecho de Kleene de autómatos

Dado um autômato M_1 construa-se um autômato M tal que $L(M) = (L(M_1))^*$, i.e., que a linguagem reconhecida por M seja o **fecho de Kleene** da linguagem reconhecida por M_1 .

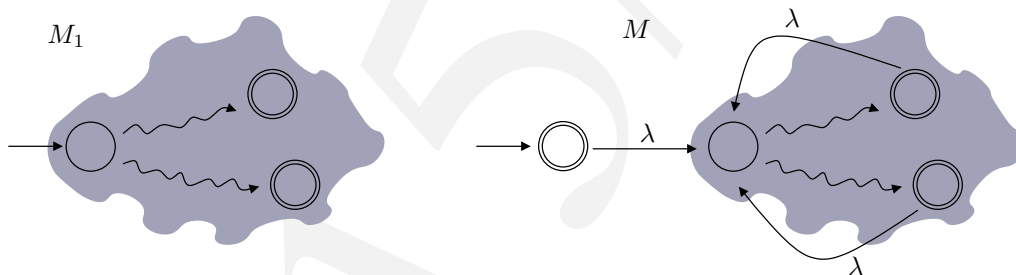
Seja $M_1 = (A, Q_1, q_1, \delta_1, F_1)$ um autômato (AFD ou AFND) qualquer. O AFND $M = (A, Q, q_0, \delta, F)$, onde

$$Q = Q_1 \cup \{q_0\}$$

$$F = F_1 \cup \{q_0\}$$

$$\delta = \delta_1 \cup (F_1 \times \{\lambda\} \times \{q_1\}) \cup \{(q_0, \lambda, q_1)\}$$

satisfaz o objectivo pretendido. Antes de se provar a afirmação anterior, veja-se graficamente a construção de M a partir de M_1 .



É claro da composição gráfica que M aceita λ ou qualquer palavra que percorra o caminho

$$q_0 \xrightarrow{\lambda} q_1 \xrightarrow{u_1} q_{f_1} \xrightarrow{\lambda} q_1 \xrightarrow{u_2} \dots \xrightarrow{\lambda} q_1 \xrightarrow{u_n} q_{f_n}, \quad \text{com } q_{f_i} \in F_1$$

o que corresponde a zero ou mais concatenações de palavras de $L(M_1)$.

Exercício 3.5

Prove-se formalmente esta equivalência.

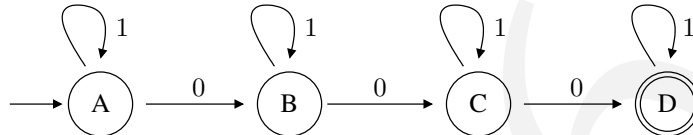
Exemplo 3.11

Sobre o alfabeto $A = \{0, 1\}$ construa um autômato que reconhece a linguagem

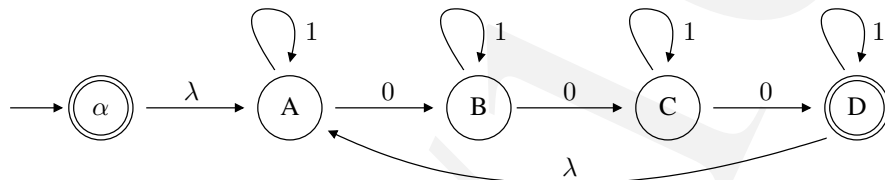
$$L = \{w \in A^* \mid \#(0, w) = 3\}$$

Com base nele construa o autômato que reconhece a linguagem L^* .

Resposta: A resposta à primeira questão é dada pelo autômato seguinte.

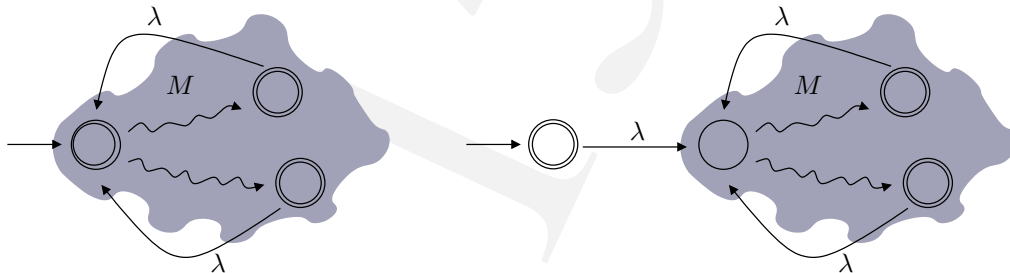


Aplicando-lhe o algoritmo de fecho obtém-se



Exercício 3.6

Mostre que as duas construções definidas com base no autômato M (a parte a sombreado) apresentadas na figura seguinte são equivalentes se o estado inicial de M for de aceitação.



Considere que M é um autômato definido sobre o alfabeto $A = \{a, b, c\}$, reconhecendo as seqüências começadas por a . Mostre que as linguagens reconhecidas pelas duas construções são a mesma. Serão equivalentes as duas construções para um M qualquer desde que o seu estado inicial não tenha arcs de chegada?

3.4.4 Complementação de autômatos

Dado um autômato M_1 construa-se um autômato M tal que $L(M) = \overline{L(M_1)}$, i.e., que a linguagem reconhecida por M seja o complemento da linguagem reconhecida por M_1 .

Seja $M = (A, Q_1, q_1, \delta_1, F_1)$ um AFD (determinista!) qualquer. M pode ser obtido complementando o conjunto dos estados de aceitação de M_1 . Ou seja, $M = (A, Q_1, q_1, \delta, Q_1 - F_1)$. Na realidade

$$u \in L(M) \Rightarrow \delta^*(q_0, u) \in F \Rightarrow \delta^*(q_0, u) \notin Q - F \Rightarrow u \notin L(M')$$

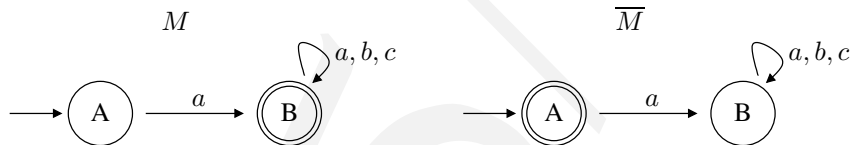
e

$$u \in L(M') \Rightarrow \delta^*(q_0, u) \in Q - F \Rightarrow \delta^*(q_0, u) \notin F \Rightarrow u \notin L(M)$$

A complementação dos estados de aceitação não funciona para AFND (não deterministas!). Se se quiser complementar um AFND deve-se primeiro convertê-lo para AFD e depois complementar este.

Exercício 3.7

Mostre que os dois autómatos da figura seguinte não são complementares um do outro. Para isso, apresente uma palavra que ambos reconheçam ou ambos rejeitem.



Descreva a linguagem reconhecida pelo autômato da direita.

3.4.5 Intersecção de autómatos

Dados dois autómatos M_1 e M_2 construa-se um autômato M tal que $L(M) = L(M_1) \cap L(M_2)$, i.e., que a linguagem reconhecida por M seja a **intersecção** das linguagens reconhecidas por M_1 e M_2 .

Com base nas operações apresentadas anteriormente é possível provar-se a existência de tal autômato. Na realidade

$$L(M_1) \cap L(M_2) = \overline{\overline{L(M_1)} \cup \overline{L(M_2)}}$$

Ora, já se provou como é que se constroem autómatos para o complemento e para a reunião.

Alternativamente, seja $M_1 = (A, Q_1, q_1, \delta_1, F_1)$ e $M_2 = (A, Q_2, q_2, \delta_2, F_2)$ dois autómatos (AFD ou

AFND) quaisquer. O AFND $M = (A, Q, q_0, \delta, F)$, em que

$$Q = Q_1 \times Q_2$$

$$q_0 = (q_1, q_2)$$

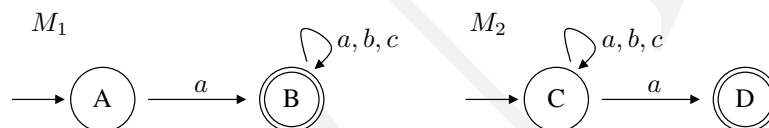
$$F = F_1 \times F_2$$

$$\delta \subseteq (Q_1 \times Q_2) \times A_\lambda \times (Q_1 \times Q_2)$$

sendo δ definido de modo que $((q_i, q_j), a, (q'_i, q'_j)) \in \delta$ se e só se $(q_i, a, q'_i) \in \delta_1$ e $(q_j, a, q'_j) \in \delta_2$, satisfaz o objectivo pretendido.

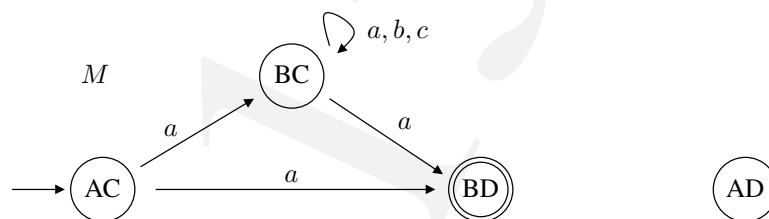
Exemplo 3.12

Sobre o alfabeto $A = \{a, b, c\}$ considere os autómatos M_1 e M_2 , graficamente apresentados a seguir, que reconhecem respectivamente as linguagens das palavras começadas e terminadas por a .



Construa o autómato que reconhece $L = L(M_1) \cap L(M_2)$.

Resposta: A figura seguinte mostra o autómato obtido pela aplicação do mecanismo de construção definido acima.

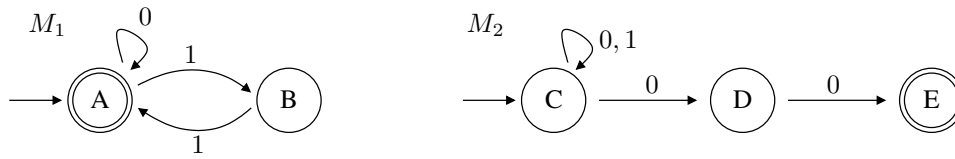


O estado AD não é alcançável a partir do estado inicial, pelo que pode ser eliminado da resposta, ficando-se com um autómato com 3 estados.

O exemplo anterior mostra que a aplicação do mecanismo de construção da intersecção de autómatos pode conduzir à introdução de estados não alcançáveis a partir do estado inicial. Isto propõe que a construção deve ser feita a partir do estado inicial, apenas se considerando os estados alcançáveis.

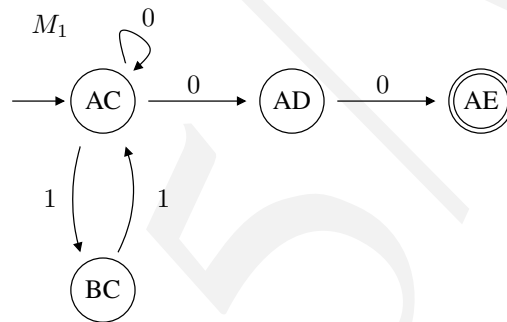
Exemplo 3.13

Sobre o alfabeto binário considere os autómatos M_1 e M_2 , graficamente apresentados a seguir



e determine a sua intersecção.

Resposta: O estado inicial do novo autômato é o estado $AC = (A, C)$. (Designar-se-á, no resto do exemplo, por XY o estado do autômato intersecção correspondente ao par (X, Y) .) Há uma transição a sair de A etiquetada com 0 — a transição $(A, 0, A)$ — e duas a sair de C — as transições $(C, 0, C)$ e $(C, 0, D)$. Logo, haverá duas transições a sair de AC etiquetadas com 0 — as transições $(AC, 0, AC)$ e $(AC, 0, AD)$. Etiquetadas com 1 há uma transição a sair de A , dirigida a B , e uma a sair de C , dirigida a C , pelo que o autômato intersecção terá um único arco etiquetado com 1 a sair de AC — o arco $(AC, 1, BC)$. Com estes arcos foram alcançados os estados AD e BC . Procedendo da mesma forma com estes estados e com outros que, eventualmente, se alcancem, obtém-se o autômato desenhado na figura seguinte, que possui apenas 4 estados e não os 6 do produto cartesiano.



3.4.6 Diferença de autômatos

Dados dois autômatos M_1 e M_2 construa-se um autômato M tal que $L(M) = L(M_1) - L(M_2)$, i.e., que a linguagem reconhecida por M seja a **diferença** das linguagens reconhecidas por M_1 e M_2 , respectivamente.

Com base nas operações apresentadas anteriormente é possível provar-se a existência de um tal autômato. Na realidade

$$L(M_1) - L(M_2) = L(M_1) \cap \overline{L(M_2)}$$

Ora, já se provou como é que se constroem autômatos para o complemento e para a intersecção.

Capítulo 4

Equivalência entre Expressões Regulares e Autómatos Finitos

As expressões regulares e os autómatos finitos são equivalentes, no sentido em que descrevem o mesmo conjunto de linguagens, as linguagens regulares. Assim sendo, é possível converter uma expressão regular dada num autómato finito que represente a mesma linguagem. Inversamente, é também possível converter um autómato finito dado numa expressão regular descrevendo a mesma linguagem.

4.1 Conversão de uma expressão regular num autómato finito

Como vimos na sua definição, uma expressão regular é contruída à custa de elementos primitivos e dos operadores escolha ($|$), concatenação ($.$) e fecho ($*$). Os elementos primitivos correspondem a expressão $()$, representando o conjunto vazio, e às expressões do tipo a com $a \in A$, sendo A o alfabeto. Podemos ainda considerar a expressão λ como um elemento primitivo, embora se saiba que $\lambda = ()^*$.

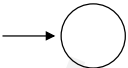
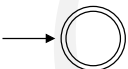
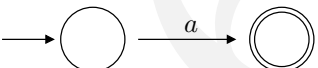
É possível decompor uma expressão regular num conjunto de elementos primitivos interligados pelos operadores referidos acima. Dada uma expressão regular qualquer ela é:

1. um elemento primitivo;
2. uma expressão do tipo e^* , sendo e uma expressão regular qualquer;
3. uma expressão do tipo $e_1.e_2$, sendo e_1 e e_2 duas expressões regulares quaisquer;
4. uma expressão do tipo $e_1|e_2$, sendo e_1 e e_2 duas expressões regulares quaisquer;

Se se identificar os autómatos equivalentes das expressões primitivas, tem-se o problema da conversão de uma expressão regular para um autómato finito resolvido, visto que se sabe como fazer a reunião, concatenação e fecho de autómatos (ver secção 3.4).

4.1.1 Autômatos dos elementos primitivos

A tabela seguinte mostra os autômatos correspondentes às expressões regulares $()$, λ e a , com $a \in A$.

expressão regular	autômato finito
$()$	
λ	
a	

Na realidade, o autômato referente a λ pode ser obtido aplicando o fecho ao autômato de $()$.

4.1.2 Algoritmo de conversão

A transformação de uma expressão regular num autômato finito pode ser feita aplicando os seguintes passos:

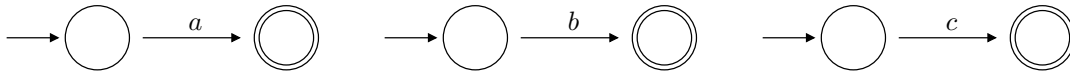
1. Se a expressão regular é o do tipo primitivo, o autômato correspondente pode ser obtido da tabela anterior.
2. Se é do tipo e^* , aplica-se este mesmo algoritmo na obtenção de um autômato equivalente à expressão regular e e, de seguida, aplica-se o fecho de autômatos descrita na secção 3.4.
3. Se é do tipo $e_1.e_2$, aplica-se este mesmo algoritmo na obtenção de autômatos para as expressões e_1 e e_2 e, de seguida, aplica-se a concatenação de autômatos descrita na secção 3.4.
4. Finalmente, se é do tipo $e_1|e_2$, aplica-se este mesmo algoritmo na obtenção de autômatos para as expressões e_1 e e_2 e, de seguida, aplica-se a reunião de autômatos descrita na secção 3.4.

Exemplo 4.1

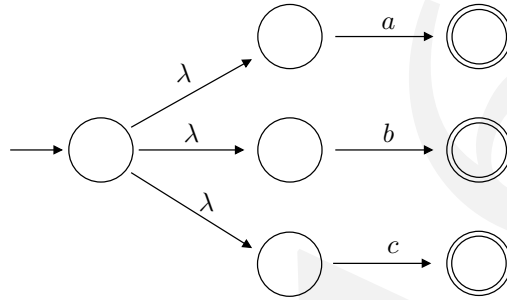
Construa um autômato equivalente à expressão regular $e = a|a(a|b|c)^*a$.

Resposta: A expressão regular e é do tipo $e_1|e_2$, com $e_1 = a$ e $e_2 = a(a|b|c)^*a$, pelo que se deve aplicar a regra 4 do algoritmo. e_1 é do tipo primitivo, pelo que se pode aplicar a tabela para obter o autômato A expressão e_2 resulta da concatenação de 3 expressões regulares, para as quais temos de obter autômatos. Apenas a segunda expressão da concatenação tem de ser considerada, visto já se ter os autômatos das outras. Tem-se então que converter a expressão regular $e_3 = (a|b|c)^*$, que se obtém do fecho sobre o autômato que represente a expressão $e_4 = a|b|c$. Este, por sua vez, resulta da reunião dos autômatos das expressões primitivas a , b e c .

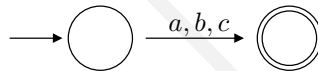
Uma vez definida a decomposição a construção faz-se dos elementos primitivos para a expressão global. Os autómatos para as expressões a , b e c são



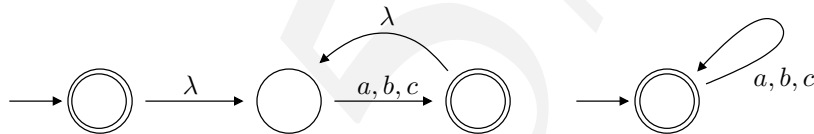
Fazendo a sua reunião obtém-se o autômato para e_4



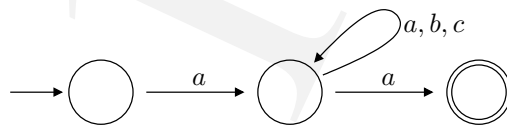
que pode ser simplificado para



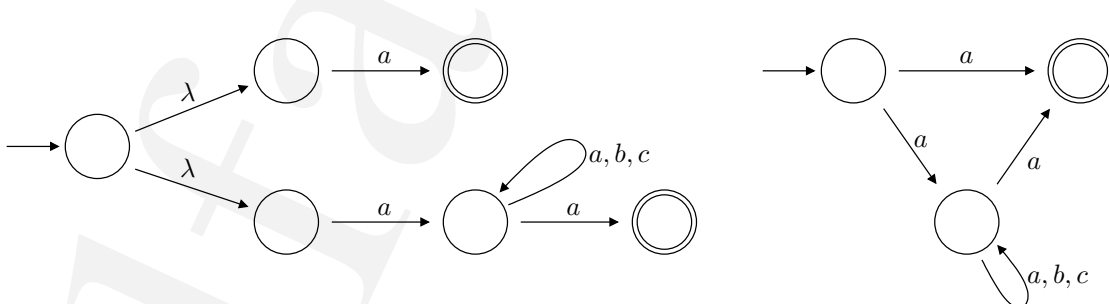
Aplicando o fecho a este autômato obtém-se o autômato para a expressão e_3 , que se apresenta a seguir tal como resulta do fecho (lado esquerdo) e após simplificação (lado direito).



O autômato para e_2 resulta da concatenação dos autómatos para a , e_3 e a , obtendo-se, após simplificação, o autômato seguinte



Finalmente, o autômato para e obtém-se da reunião dos autómatos para e_2 e a , o que resulta no autômato seguinte, lado esquerdo (o lado direito representa o mesmo autômato após simplificação).



4.2 Conversão de um autômato finito numa expressão regular

A conversão de autômatos finitos em expressões regulares baseia-se num novo tipo de autômatos, designados **autômatos finitos generalizados**. Estes autômatos são semelhantes aos autômatos finitos não-deterministas mas em que as etiquetas dos arcos são expressões regulares.

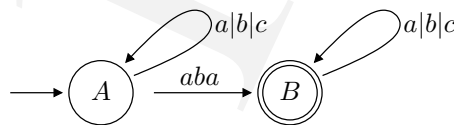
Um **autômato finito generalizado** (AFG) é um quintuplo $M = (A, Q, q_0, \delta, F)$, em que:

- A é o alfabeto de entrada;
- Q é um conjunto finito não vazio de estados;
- $q_0 \in Q$ é o estado inicial;
- $\delta \subseteq (Q \times E \times Q)$ é a relação a transição entre estados, sendo E o conjunto das expressões regulares definidas sobre A ; e
- $F \subseteq Q$ é o conjunto dos estados de aceitação.

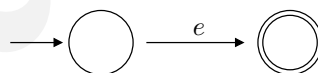
Note que com base nesta definição os autômatos finitos deterministas e não deterministas são autômatos finitos generalizados.

Exemplo 4.2

O autômato finito generalizado representado graficamente abaixo representa o conjunto das palavras, definidas sobre o alfabeto $A = \{a, b, c\}$, que contêm a sub-palavra aba .



Imagine que consegue transformar um autômato finito generalizado dado — pense, por exemplo, no autômato da figura — num outro com a configuração dada pela figura seguinte



sendo e uma expressão regular qualquer. Então, e é uma expressão regular equivalente ao autômato dado. Um autômato como o da figura designa-se por **autômato finito generalizado reduzido**.

4.2.1 Algoritmo de conversão

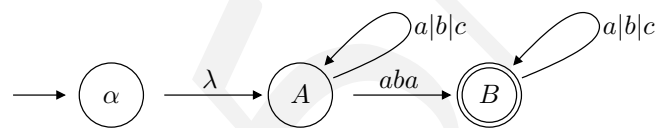
Assim sendo, a obtenção de uma expressão regular equivalente a um autômato qualquer dado, resume-se à transformação desse autômato num autômato finito generalizado reduzido. A redução de um AFG pode ser feita aplicando o seguinte procedimento:

1. transformação de um AFG noutra cujo estado inicial não tenha arcos de chegada;
2. transformação de um AFG noutra com um único estado de aceitação que não tenha arcos de saída;
3. eliminação, um a um, dos restantes estados.

Se o estado inicial de um AFG possui arcos de chegada, a transformação definida pelo ponto 1 do procedimento anterior faz-se acrescentando um novo estado, que passa a ser o estado inicial do AFG transformado, e um arco etiquetado com λ ligando este estado ao antigo estado inicial.

Exemplo 4.3

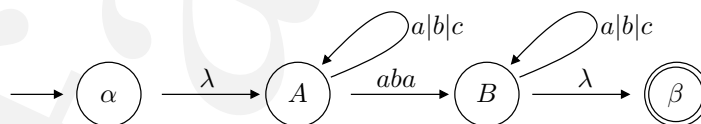
O AFG do exemplo 2 tem o estado inicial com arcos de chegada. O AFG representado abaixo, é-lhe equivalente e o estado inicial apenas possui arcos de saída.



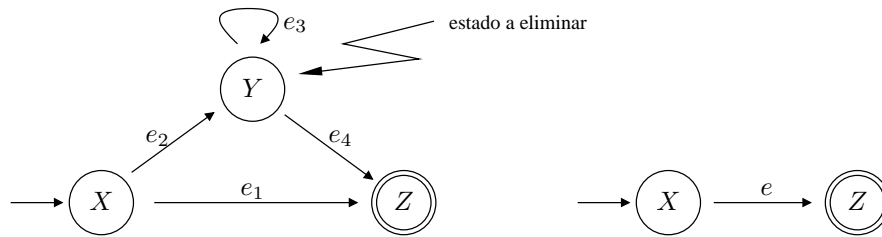
Se um AFG possui mais que um estado de aceitação ou se possui apenas um mas com arcos de saída, a transformação definida pelo ponto 2 do procedimento de transformação faz-se acrescentando um novo estado, que passa a ser o único estado de aceitação do AFG transformado, e arcos etiquetados com λ ligando os antigos estados de aceitação ao novo.

Exemplo 4.4

O AFG do exemplo 3 tem um único estado de aceitação mas com arcos de saída. O AFG representado abaixo, é-lhe equivalente e o seu estado de aceitação apenas possui arcos de chegada.



A eliminação de estados preconizada pelo ponto 3 do procedimento de transformação está esquematizada na figura seguinte



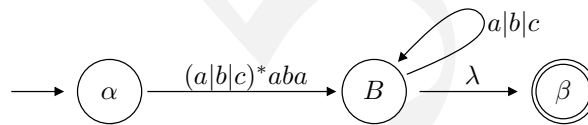
A situação inicial é representada pelo autômato da esquerda, sendo Y o estado que se pretende eliminar. Após a eliminação de Y os estados X e Z devem ser ligados por um arco etiquetado com uma expressão regular que represente o conjunto de palavras que permitiam evoluir de X para Z na situação inicial. Assim,

$$e = e_1 | e_2 e_3^* e_4$$

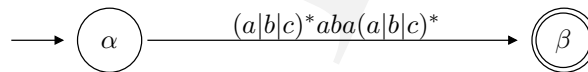
Exemplo 4.5

Remova os estados A e B do AFG do exemplo 4.

Resposta: Comece-se por remover o estado A . Os estados α , A e β desempenham respectivamente os papéis dos estados X , Y e Z no esquema relativo ao ponto 3 do procedimento de transformação. Fazendo a correspondência tem-se $e_1 = ()$, $e_2 = \lambda$, $e_3 = a|b|c$ e $e_4 = aba$, pelo que o AFG após supressão do estado A se transforma no AFG seguinte, onde $e = ()|\lambda(a|b|c)^*aba = (a|b|c)^*aba$.



De seguida elimina-se o estado B . Neste caso os estados α , B e β desempenham respectivamente os papéis dos estados X , Y e Z , resultando em



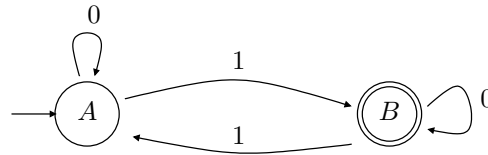
pelo que a expressão pretendida é

$$e = (a|b|c)^*aba(a|b|c)^*$$

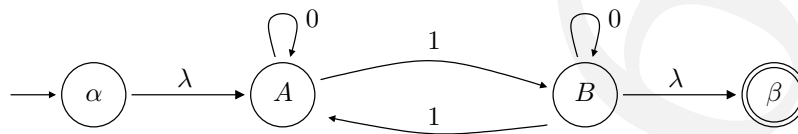
O exemplo anterior não deixa transparecer alguma complexidade que pode aparecer no processo de remoção de estados. O estado a remover pode participar em vários triângulos $X - Y - Z$, o que obriga a calcular várias expressões regulares por cada estado a remover.

Exemplo 4.6

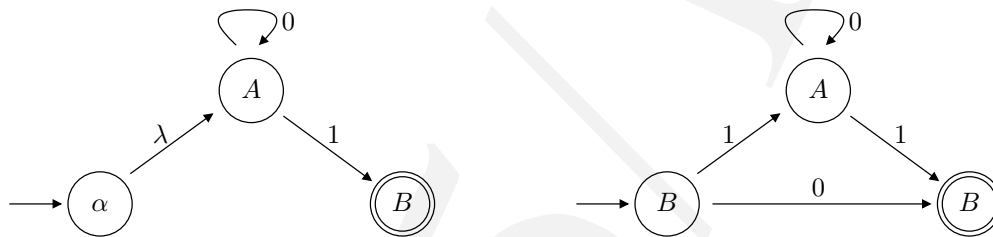
Considere o autômato da figura seguinte é obtenha uma expressão regular equivalente



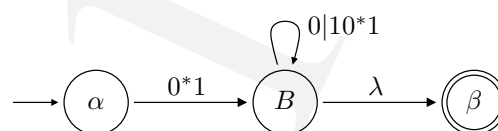
Resposta: Comece-se por acrescentar novos estados inicial e de aceitação em consequência da aplicação dos pontos 1 e 2 do algoritmo de transformação. Obtém-se



A seguir, suprima-se o estado A . Há dois triângulos $X - Y - Z$ a considerar, um formado pelos estados α , A e B e outro pelos estados B , A e B . Estes triângulos estão ilustrados na figura seguinte.



Da supressão de A resultam um arco entre os estados α e B e outro de B para ele próprio. O primeiro terá a etiqueta $e_1 = 0^*1$ e o segundo a etiqueta $e_2 = 0|10^*1$. Note que o arco original de B para B desaparece em consequência da supressão de A , sendo substituído pelo novo com etiqueta e_2 . Após a supressão obtém-se o AFG seguinte

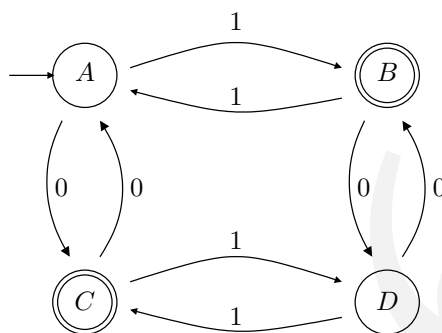


Suprimindo B obtém-se facilmente a expressão pretendida

$$e = 0^*1(0|10^*1)^*$$

Exercício 4.1

Determine uma expressão regular equivalente do autômato finito seguinte



Capítulo 5

Gramáticas

NOTA PRÉVIA: *Este capítulo não está completo e não foi devidamente revisto, pelo que, por um lado, há partes omissas e, por outro lado, pode conter falhas.*

Considere uma estrutura G , definido sobre o alfabeto $T = \{a, b\}$, com as regras de rescrita

$$S \rightarrow b$$

$$S \rightarrow a S$$

significando que em qualquer sequência onde apareça o símbolos S , ele pode ser substituído (rescrito) por b ou por $a S$. Que palavras apenas constituídas por símbolos do alfabeto T se podem gerar a partir de S ? Substituindo S por b obtém-se b que pertence a T^* . Isto pode-se representar por

$$S \Rightarrow b$$

Substituindo S por aS e o novo S por b , obtém-se ab , escrevendo-se

$$S \Rightarrow a S \Rightarrow a b$$

Na realidade, começando em S podem-se gerar as palavras b , ab , aab , $aa \cdots ab$, ou seja, todas as palavras da linguagem L dada por

$$L = \{a^n b \mid n \geq 0\}$$

ou, equivalentemente, as palavras descritas pela expressão regular a^*b .

A estrutura G é uma **gramática** que permite gerar a linguagem L . T é o alfabeto de entrada, também designado por conjunto de símbolos terminais. S é um símbolo não-terminal, ponto de partida de todas

as palavras. É designado **símbolo inicial da gramática**. As regras de rescrita são designadas **produções**. O conjunto das palavras que se podem gerar a partir de S é a linguagem descrita por G .

Sobre o alfabeto $T = \{a, b, c\}$, considere as seguintes regras de rescrita.

$$S \rightarrow \lambda$$

$$S \rightarrow X S$$

$$X \rightarrow a$$

$$X \rightarrow c$$

$$X \rightarrow a b c$$

Que palavras se podem gerar a partir de S ? Podem-se gerar 0 ou mais concatenações de X , sendo cada X substituível por a , c ou abc . Ou seja, pode-se gerar a mesma linguagem que a descrita pela expressão regular $e = (a|c|abc)^*$. Esta estrutura é uma gramática que tem dois símbolos não terminais — S e X —, sendo que S é o símbolo inicial.

Nos dois exemplos anteriores as gramáticas descrevem linguagens regulares. Mas o poder expressivo das gramáticas é superior ao das expressões regulares, como se mostra com os exemplos seguintes.

Seja

$$S \rightarrow \lambda$$

$$S \rightarrow a S b$$

uma gramática G definida sobre o alfabeto $T = \{a, b\}$. Qual é a linguagem L descrita pela gramática G ? A partir de S podem-se gerar as palavras λ , ab , $aabb$, \dots , ou seja $L = \{a^n b^n \mid n \geq 0\}$.

Exemplo 5.1

A gramática seguinte, definida sobre o alfabeto $T = \{a, b\}$, gera a linguagem $L = \{a^m b^n \mid n > 0 \wedge m > n\}$.

$$S \rightarrow A X$$

$$A \rightarrow a$$

$$A \rightarrow a A$$

$$X \rightarrow a b$$

$$X \rightarrow a X b$$

Exemplo 5.2

A gramática seguinte, definida sobre o alfabeto $T = \{a, b, c\}$, gera a linguagem $L = \{wcw^R \mid w \in \{a, b\}^*\}$, onde w^R representa a palavra w com os símbolos tomados por ordem inversa.

$$S \rightarrow c$$

$$S \rightarrow a S a$$

$$S \rightarrow b S b$$

As linguagens descritas pelas 3 últimas gramáticas não são regulares, não podendo, por isso, ser descritas por expressões regulares. Mas são-no por autómatos de pilha. São designadas **independentes do contexto**. Esta designação resulta de todas as produções serem da forma $A \rightarrow \alpha$, em que A é um símbolo não terminal e α é uma expressão qualquer constituída por símbolos terminais ou não terminais.

No exemplo seguinte apresenta-se uma gramática **dependente do contexto**.

Exemplo 5.3

A gramática seguinte, definida sobre o alfabeto $T = \{a, b\}$, gera a linguagem $L = \{a^n b^n a^n \mid n > 0\}$.

$$S \rightarrow a S B A$$

$$S \rightarrow a b A$$

$$A B \rightarrow B A$$

$$b B \rightarrow b b$$

$$a A \rightarrow a a$$

$$b A \rightarrow b a$$

5.1 Definições**5.1.1 Definição de gramática**

Dos exemplos anteriores constata-se que uma gramática é uma estrutura definida com base num conjunto de símbolos terminais (o alfabeto), num conjunto de símbolos não terminais (também designados variáveis), num conjunto de produções e no estabelecimento do símbolo não terminal inicial. Formalmente, uma gramática é um quádruplo $G = (T, N, P, S)$, onde

- T é um conjunto finito não vazio de símbolos terminais;
- N , sendo $N \cap T = \emptyset$, é um conjunto finito não vazio de símbolos não terminais;
- P é um conjunto de produções, cada uma da forma $\alpha \rightarrow \beta$; e
- $S \in N$ é o símbolo inicial.

Nas produções, α e β são designados por **cabeça da produção** e **corpo da produção**, respectivamente.

Viu-se na secção anterior que as gramáticas são mais expressivas que as expressões regulares ou os autómatos finitos, no sentido em que permitem descrever mais linguagens. Dos 6 exemplos apresentados anteriormente, os 2 primeiros representam linguagens regulares — descritíveis por expressões regulares ou autómatos —, os 3 seguintes representam linguagens independentes do contexto — descritíveis por autómatos de pilha — e o último representa uma linguagem dependente do contexto.

Na realidade o tipo de linguagem descritível pelas gramáticas depende da forma como P é definido. Consideram-se 4 casos.

1. Se todos os elementos de P têm a forma $\alpha \rightarrow \beta$, com $\alpha \in N$ e $\beta \in (T^* \cup T^* \cdot N)$, a gramática diz-se **regular** e prova-se que o seu poder descritivo é equivalente ao das expressões regulares e autómatos finitos.
2. O mesmo acontece quando todos os elementos de P têm a forma $\alpha \rightarrow \beta$, com $\alpha \in N$ e $\beta \in (T^* \cup N \cdot T^*)$. No entanto, daqui em diante será usada sempre a primeira forma.
3. Se todos os elementos de P têm a forma $\alpha \rightarrow \beta$, com $\alpha \in N$ e $\beta \in (T \cup N)^*$ a gramática diz-se **independente do contexto** e prova-se que seu poder descritivo é equivalente aos dos autómatos de pilha.
4. Se todos os elementos de P têm a forma $\alpha \rightarrow \beta$, com $\alpha \in (N \cup T)^* \cdot N \cdot (T \cup N)^*$ e $\beta \in (N \cup T)^*$, a gramática diz-se **dependente do contexto**. Esta categoria não será tratada no âmbito deste curso.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.2.1, "The formal definition of a context-free grammar".

Simplificação notacional

Na descrição de uma gramática o operador $|$ é usado como elemento simplificador. Se $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_n$, são produções de uma gramática, a notação $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ representa o mesmo conjunto de produções. Por exemplo, a gramática

$$S \rightarrow c | a S a | b S b$$

é a mesma que a do exemplo 5.2.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.2.2, "Notational conventions".

5.1.2 Derivação

D Dada uma produção $u \rightarrow v$ e uma palavra $\alpha u \beta$, chama-se **derivação direta** à rescrita de $\alpha u \beta$ em $\alpha v \beta$, denotando-se

$$\alpha u \beta \Rightarrow \alpha v \beta$$

Em algumas situações, será usada o termo **passo de derivação** como sinónimo de derivação directa.

D Chama-se **derivação** a uma sucessão de zero ou mais derivações diretas, denotando-se

$$\alpha \Rightarrow^* \beta$$

ou, equivalentemente,

$$\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_n = \beta$$

onde n é o comprimento da derivação.

- A notação $\alpha \Rightarrow^n \beta$ é usada para representar uma derivação de comprimento n
- A notação $\alpha \Rightarrow^+ \beta$ é usada para representar uma derivação de comprimento não nulo

D Dada uma gramática $G = (T, N, P, S)$ e uma palavra $u \in (T \cup N)^+$, o conjunto das **palavras derivadas** a partir de u é representado por

$$D(u) = \{v \in T^* : u \Rightarrow^* v\}$$

D A **linguagem gerada pela gramática** $G = (T, N, P, S)$ é representada por

$$L(G) = D(S) = \{v \in T^* : S \Rightarrow^* v\}$$

5.2 Gramáticas regulares

Tal como foi definido na secção 5.1.1, uma gramática $G = (T, N, P, S)$ diz-se regular se todas as suas produções têm a forma $\alpha \rightarrow \beta$, com $\alpha \in N$ e $\beta \in (T^* \cup T^* \cdot N)$.

5.2.1 Operações sobre as gramáticas regulares

As gramáticas regulares são fechadas sob as operações de reunião, concatenação, fecho, intersecção e complementação.

Reunião

D Sejam $G_1 = (T_1, N_1, P_1, S_1)$ e $G_2 = (T_2, N_2, P_2, S_2)$ duas gramáticas regulares quaisquer, com $N_1 \cap N_2 = \emptyset$. A gramática $G = (T, N, P, S)$ onde

$$\begin{aligned} T &= T_1 \cup T_2 \\ N &= N_1 \cup N_2 \cup \{S\} \quad \text{com } S \notin (N_1 \cup N_2) \\ P &= \{S \rightarrow S_1, S \rightarrow S_2\} \cup P_1 \cup P_2 \end{aligned}$$

é regular e gera a linguagem $L = L(G_1) \cup L(G_2)$.

As produções de G_1 e G_2 são transferidas, inalteradas, para a gramática G e são acrescentadas duas novas. As novas produções, $S \rightarrow S_1$ e $S \rightarrow S_2$, permitem que a nova gramática gere as palavras que as gramáticas G_1 e G_2 geravam, concretizando dessa forma a reunião.

Concatenação

D Sejam $G_1 = (T_1, N_1, P_1, S_1)$ e $G_2 = (T_2, N_2, P_2, S_2)$ duas gramáticas regulares quaisquer, com $N_1 \cap N_2 = \emptyset$. A gramática $G = (T, N, P, S)$ onde

$$\begin{aligned} T &= T_1 \cup T_2 \\ N &= N_1 \cup N_2 \\ P &= \{A \rightarrow \omega S_2 : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^*\} \\ &\quad \cup \{A \rightarrow \omega : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^* N_1\} \\ &\quad \cup P_2 \\ S &= S_1 \end{aligned}$$

é regular e gera a linguagem $L = L(G_1) \cdot L(G_2)$.

As produções de G_2 são transferidas inalteradas para G . As produções de G_1 cujos corpos terminam em não terminais também são transferidas inalteradas para G . As restantes, i.e., as cujos corpos só têm terminais, são transferidas para G , sendo-lhes acrescentado o símbolo inicial de G_2 no fim. O símbolo inicial de G é o símbolo inicial de G_1 .

Fecho de Kleene

D Seja $G_1 = (T_1, N_1, P_1, S_1)$ uma gramática regular qualquer. A gramática $G = (T, N, P, S)$ onde

$$\begin{aligned} T &= T_1 \\ N &= N_1 \cup \{S\} \quad \text{com} \quad S \notin N_1 \\ P &= \{S \rightarrow \lambda, S \rightarrow S_1\} \\ &\quad \cup \{A \rightarrow \omega S : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^*\} \\ &\quad \cup \{A \rightarrow \omega : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^* N_1\} \end{aligned}$$

é regular e gera a linguagem $L = (L(G_1))^*$.

As produções de G_1 cujos corpos terminam em não terminais são transferidas inalteradas para G . As restantes, i.e., as cujos corpos só têm terminais, são transferidas para G , sendo-lhes acrescentado o símbolo inicial de G_1 no fim. A nova produção $S \rightarrow \lambda$, per si, garante que $(L(G_1))^0$ está contido em $L(G)$. Essa produção em conjunto com a outra nova $S \rightarrow S_1$ garante que $(L(G_1))^n \subseteq L(G)$, para qualquer $n \geq 1$.

5.2.2 Equivalência em relação aos AF e ER

A linguagem gerada por uma gramática regular é regular. Logo, é possível converter uma gramática regular num autómato finito ou numa expressão regular que represente a mesma linguagem e vice-versa.

Conversão de uma GR num AFG

D Seja $G = (T, N, P, S)$ uma gramática regular qualquer. O autómato finito generalizado $M =$

(A, Q, q_0, δ, F) onde

$$\begin{aligned} A &= T \\ Q &= N \cup \{q_f\} \quad \text{com } q_f \notin N \\ q_0 &= S \\ \delta &= \{(A, \omega, B) : (A \rightarrow \omega B) \in P \wedge B \in N\} \\ &\quad \cup \{(A, \omega, q_f) : (A \rightarrow \omega) \in P \wedge \omega \in T^*\} \\ F &= \{q_f\} \end{aligned}$$

reconhece a linguagem $L = L(G)$.

Os símbolos não terminais de G dão origem aos estados de não aceitação de M . É acrescentado um estado de aceitação. As produções de G cujos corpos terminam num símbolo não terminal dão origem às transições entre estados de não aceitação. As produções cujos corpos apenas têm símbolos terminais dão origem a transições para o estado de aceitação.

Conversão de uma AF num GR

D Seja $M = (A, Q, q_0, \delta, F)$ um autômato finito, não generalizado, qualquer. A gramática regular $G = (T, N, P, S)$ onde

$$\begin{aligned} T &= A \\ N &= Q \\ P &= \{q_i \rightarrow a q_j : (q_i, a, q_j) \in \delta\} \\ &\quad \cup \{q \rightarrow \lambda : q \in F\} \\ S &= q_0 \end{aligned}$$

reconhece a linguagem $L = L(G)$.

Os estados de M convertem-se nos símbolos não terminais de G . As transições de M convertem-se em produções com um não terminal no fim do corpo. Os estados de aceitação convertem-se em produções- λ . O símbolo inicial de G é o correspondente ao estado inicial de M .

Conversão de uma GR num ER

A conversão de uma gramática regular numa expressão regular que represente a mesma linguagem faz-se por um processo de transformação de equivalência semelhante ao usado para a conversão de um autômato finito generalizado numa expressão regular.

Dada uma gramática $G = (T, N, P, S)$, primeiro gera-se o conjunto de triplos seguinte:

$$\begin{aligned}\mathcal{E} &= \{(E, \lambda, S)\} \\ &\cup \{(A, \omega, B) : (A \rightarrow \omega B) \in P \wedge B \in N\} \\ &\cup \{(A, \omega, \lambda) : (A \rightarrow \omega) \in P \wedge \omega \in T^*\}\end{aligned}$$

com $E \notin N$. Desta forma garante-se que E não aparece no lado direito de qualquer triplo. De seguida, removem-se, por transformações de equivalência, todos os triplos que contêm símbolos de N . No fim, obtém-se um único triplo da forma (E, e, λ) , sendo e a expressão regular pretendida.

O procedimento de remoção de todos os triplos de \mathcal{E} que contêm um dado símbolo B de N desenvolve-se nos seguintes passos:

1. Primeiro, juntam-se num só todos os triplos que possuem o primeiro e o terceiro elementos iguais. Isto é, todos os triplos da forma (X, e_i, Y) , com $i = 1, 2, \dots, n$, são substituídos por um único triplo (X, e, Y) , com $e = e_1|e_2|\dots|e_n$.
2. A seguir, por cada triplo de triplos da forma $((A, e_1, B), (B, e_2, B), (B, e_3, C))$ pertencente a \mathcal{E} acrescenta-se o triplo (A, e, B) , com $e = e_1(e_2)^*e_3$.
3. Por fim, todos os triplos com B são removidos de \mathcal{E} .

Note que, para um dado B , pode não existir em \mathcal{E} um triplo da forma (B, e_2, B) . Mas, dizer que o triplo (B, e_2, B) não existe é equivalente a dizer que existe com $e_2 = ()$, pelo que a substituição referida acima pode aplicar-se. Note ainda que $()^* = \lambda$.

Conversão de uma ER numa GR

Qualquer expressão regular pode ser obtida através da aplicação dos operadores regulares — escolha, concatenação e fecho de Kleene — às expressões regulares primitivas. Para obter uma gramática regular equivalente a uma expressão regular dada, basta obter gramáticas regulares para as expressões regulares primitivas e aplicar as operações regulares sobre gramáticas regulares.

Em relação às expressões regulares primitivas tem-se o seguinte:

- A GR para a ER λ é dada por

$$S \rightarrow \lambda$$

- A GR para a ER a , qualquer que seja o a , é dada por

$$S \rightarrow a$$

5.3 Gramáticas independentes do contexto

Nas gramáticas independentes do contexto, em geral, em cada passo de uma derivação estão envolvidos vários símbolos não terminais. Como as produções têm a forma $\alpha \rightarrow \beta$, com $\alpha \in N$ e $\beta \in (T \cup N)^*$, β pode conter vários símbolos não terminais, que serão introduzidos na derivação se a produção for utilizada.

A gramática seguinte, definida sobre o alfabeto $T = \{a, b\}$, é independente do contexto e gera a linguagem $L = \{w \in T^* \mid \#(a, w) = \#(b, w)\}$.

$$S \rightarrow a B \mid b A$$

$$A \rightarrow a \mid a S \mid b A A$$

$$B \rightarrow b \mid b S \mid a B B$$

A palavra $aabbab$ tem 4 derivações possíveis, das quais são apresentadas duas.

1. $\underline{S} \Rightarrow a\underline{B} \Rightarrow aa\underline{B}B \Rightarrow aab\underline{B} \Rightarrow aabb\underline{S} \Rightarrow aabba\underline{B} \Rightarrow aabbab$
2. $\underline{S} \Rightarrow a\underline{B} \Rightarrow aa\underline{B}B \Rightarrow aaBb\underline{S} \Rightarrow aaBba\underline{B} \Rightarrow aa\underline{B}bab \Rightarrow aabbab$

Usou-se o sublinhado para destacar o símbolo não terminal expandido em cada derivação directa.

A derivação 1 designa-se por **derivação à esquerda**, porque em cada passo se expande o símbolo não-terminal mais à esquerda. A derivação 2 designa-se por **derivação à direita**, porque em cada passo se expande o símbolo não-terminal mais à direita.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.2.3, "Derivations".

5.3.1 Operações sobre gramáticas independentes do contexto

A classe das gramáticas independentes do contexto é fechada sobre as operações de reunião, concatenação e fecho, mas não o é sobre as operações de intersecção e complementação.

Reunião

Sejam $G_1 = (T, N_1, P_1, S_1)$ e $G_2 = (T, N_2, P_2, S_2)$ duas gramáticas independentes do contexto que geram as linguagens L_1 e L_2 , respectivamente. A gramática $G = (T, N, P, S)$, onde

$$S \notin (T \cup N_1 \cup N_2);$$

$$N = N_1 \cup N_2 \cup \{S\}$$

$$P = P_1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}$$

gera a linguagem $L = L_1 \cup L_2$.

Concatenação

Sejam $G_1 = (T, N_1, P_1, S_1)$ e $G_2 = (T, N_2, P_2, S_2)$ duas gramáticas independentes do contexto que geram as linguagens L_1 e L_2 , respectivamente. A gramática $G = (T, N, P, S)$, onde

$$S \notin (T \cup N_1 \cup N_2);$$

$$N = N_1 \cup N_2 \cup \{S\}$$

$$P = P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}$$

gera a linguagem $L = L_1 \cdot L_2$.

Fecho de Kleene

Seja $G_1 = (T, N_1, P_1, S_1)$ uma gramática independente do contexto que gera a linguagem L_1 . A gramática $G = (T, N, P, S)$, onde

$$S \notin (T \cup N_1);$$

$$N = N_1 \cup \{S\}$$

$$P = P_1 \cup \{S \rightarrow \lambda, S \rightarrow S_1 S\}$$

gera a linguagem $L = L_1^*$.

Intersecção e complementação

Se L_1 e L_2 são duas linguagens independentes do contexto, e, por conseguinte, descritíveis por gramáticas independentes do contexto, a sua intersecção pode não o ser. Já se disse atrás que a linguagem

$$L = \{a^i b^i c^i \mid i \geq 0\}$$

não é independente do contexto. No entanto, ela pode ser obtida por intersecção das linguagens $L_1 = \{a^i b^i c^j \mid i, j \geq 0\}$ e $L_2 = \{a^i b^j c^j \mid i, j \geq 0\}$ que o são.

Sabe-se que $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. Então, se a intersecção de linguagens independentes do contexto pode resultar numa linguagem que não o é, o mesmo pode acontecer com a complementação.

5.4 Árvore de derivação

Será que as várias derivações de uma mesma palavra são diferentes? Poderão ter interpretações diferentes? Veja-se com um exemplo que de facto podem. Considere a gramática

$$S \rightarrow S + S \mid S \cdot S \mid \neg S \mid (S) \mid 0 \mid 1$$

e compare as duas derivações esquerdas seguintes da palavra $1+1 \cdot 0$.

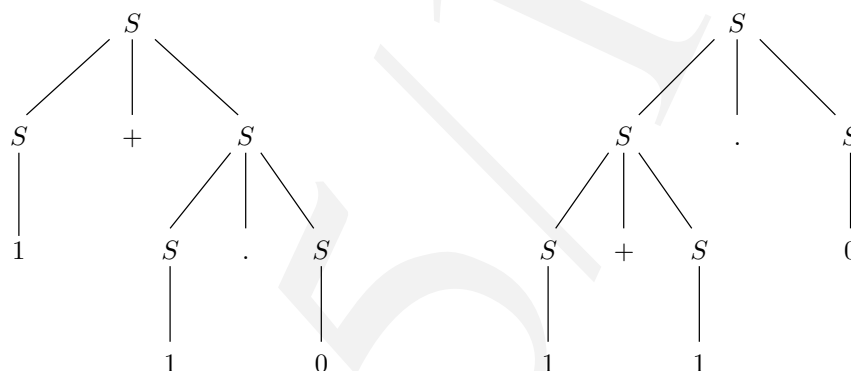
Derivação 1:

$$\underline{S} \Rightarrow \underline{S}+S \Rightarrow 1+\underline{S} \Rightarrow 1+\underline{S} \cdot S \Rightarrow 1+1 \cdot \underline{S} \Rightarrow 1+1 \cdot 0$$

Derivação 2:

$$\underline{S} \Rightarrow \underline{S} \cdot S \Rightarrow \underline{S}+S \cdot S \Rightarrow 1+\underline{S} \cdot S \Rightarrow 1+1 \cdot \underline{S} \Rightarrow 1+1 \cdot 0$$

Serão estas derivações equivalentes? Para responder a esta pergunta vão-se representar as derivações usando um outro formalismo. A **árvore de derivação** (*parse tree*) é um mecanismo de representação de uma derivação que capta a interpretação dada nessa derivação. Veja-se as duas derivações anteriores representadas de forma arbórea.



A árvore da esquerda corresponde à derivação 1 e a da direita à 2. Vê-se claramente que as duas árvores têm interpretações distintas: a da esquerda é equivalente a ter-se $1+(1 \cdot 0)$, enquanto que a direita é equivalente a $(1+1) \cdot 0$. Em termos de álgebra booleana as duas expressões são bastante diferentes.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.2.4, "Parse trees and derivations".

5.4.1 Ambiguidade

Diz-se que uma palavra é derivada **ambiguamente** se possuir duas ou mais árvores de derivação distintas. Na gramática anterior a palavra $1+1 \cdot 0$ é gerada ambiguamente. Diz-se que uma gramática é **ambígua** se possuir pelo menos uma palavra gerada ambiguamente. A gramática anterior é ambígua.

Frequentemente é possível definir-se uma gramática não ambígua que gere a mesma linguagem que uma

ambígua. A gramática anterior pode ser reescrita por

$$S \rightarrow T \mid S + T$$

$$T \rightarrow F \mid T \cdot F$$

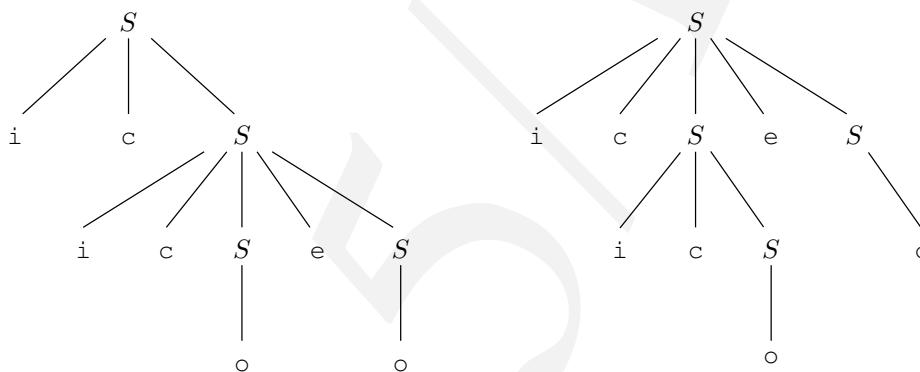
$$F \rightarrow K \mid \neg K$$

$$K \rightarrow 0 \mid 1 \mid (S)$$

que não é ambígua e gera exactamente a mesma linguagem. A gramática

$$S \rightarrow o \mid i \text{ c } S \mid i \text{ c } S \text{ e } S$$

é uma abstracção da instrução `if-then-else` e possui ambiguidade. A palavra `icicoeo` tem duas árvores de derivação possíveis, representadas abaixo.



A árvore da esquerda corresponde à interpretação dada na linguagem C, em que o e (*else*) está associado ao i (*if*) mais à direita. A árvore da direita associa o e ao i mais à esquerda.

É possível por manipulação gramatical obter-se uma gramática equivalente sem ambiguidade. A gramática seguinte não é ambígua e descreve a mesma linguagem que a anterior.

$$S \rightarrow o \mid i \text{ c } S \mid i \text{ c } S' \text{ e } S$$

$$S' \rightarrow o \mid i \text{ c } S' \text{ e } S'$$

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.2.5, "Ambiguity".

Linguagens inerentemente ambíguas

Há linguagens **inerentemente ambíguas**, no sentido em que é impossível definir-se uma gramática não ambígua que gere essa linguagem. É, por exemplo, o caso da linguagem

$$L = \{a^i b^j c^k \mid i = j \vee j = k\}$$

5.5 Limpeza de gramáticas

5.5.1 Símbolos produtivos e não produtivos

Seja $G = (T, N, P, S)$ uma gramática qualquer. Um símbolo não terminal A diz-se **produtivo** se for possível transformá-lo num expressão contendo apenas símbolos terminais. Ou seja, A é produtivo se

$$A \Rightarrow^* u \quad \wedge \quad u \in T^*$$

Caso contrário, diz-se que A é **improdutivo**. Uma gramática é improdutiva se o seu símbolo inicial for improdutivo.

Sobre o alfabeto $T = \{a, b, c\}$, considere a gramática

$$\begin{aligned} S &\rightarrow a b \mid a S b \mid X \\ X &\rightarrow c X \end{aligned}$$

S é produtivo, porque

$$S \Rightarrow ab \quad \wedge \quad ab \in T^*$$

Em contrapartida, X é improdutivo.

$$X \Rightarrow cX \Rightarrow ccX \Rightarrow^* c \cdots cX$$

Por mais que se tente é impossível transformar X numa sequência de símbolos terminais.

Seja $G = (T, N, P, S)$ uma gramática qualquer. O conjunto dos seus símbolos produtivos, N_p , pode ser obtido por aplicação das seguintes regras construtivas

$$\begin{aligned} \text{if } (A \rightarrow \alpha) \in P \text{ and } \alpha \in T^* \text{ then } A &\in N_p \\ \text{if } (A \rightarrow \alpha) \in P \text{ and } \alpha \in (T \cup N_p)^* \text{ then } A &\in N_p \end{aligned}$$

A 1ª regra é um caso particular da 2ª, pelo que poderia ser retirada. Optou-se por a incluir porque torna a leitura mais fácil.

Começando com um N_p igual ao resultado da aplicação da 1ª regra a todas as produções da gramática e extendendo depois esse conjunto por aplicação sucessiva da 2ª regra obtém-se o conjunto de todos os símbolos produtivos de G . O algoritmo seguinte executa esse procedimento.

Algoritmo 5.1 (Cálculo dos símbolos produtivos)

```

let  $N_p = \emptyset, P_p = P$ 
repeat
  nothingAdded = TRUE
  foreach  $(A \rightarrow \alpha) \in P_p$  do
    if  $\alpha \in (T \cup N_p)^*$  then
      if  $A \notin N_p$  then
         $N_p = N_p \cup \{A\}$ 
        nothingAdded = FALSE
         $P_p = P_p \setminus \{A \rightarrow \alpha\}$ 
until nothingAdded or  $N_p = N$ 

```

Nele, N_p representa o conjunto dos símbolos produtivos já identificados e P_p o conjunto das produções contendo símbolos ainda não identificados como produtivos. Se numa iteração nenhum símbolo for marcado como produtivo o algoritmo pára, sendo o conjunto dos símbolos produtivos o conjunto N_p tido nesse momento. Obviamente que o algoritmo também pára, se no fim de uma iteração, $N_p = N$, isto é, se todos os símbolos foram marcados como produtivos.

5.5.2 Símbolos acessíveis e não acessíveis

Seja $G = (T, N, P, S)$ uma gramática qualquer. Um símbolo terminal ou não terminal x diz-se **acessível** se for possível transformar S (o símbolo inicial) numa expressão que contenha x . Ou seja,

$$S \Rightarrow^* \alpha x \beta$$

Caso contrário, diz-se que x é **inacessível**.

Considere a gramática

$$S \rightarrow \lambda \mid a S b \mid c C c$$

$$C \rightarrow c S c$$

$$D \rightarrow d X d$$

$$X \rightarrow C C$$

É impossível transformar S numa expressão que contenha D , d , ou X , pelo que estes símbolos são inacessíveis. Os restantes são acessíveis.

Seja $G = (T, N, P, S)$ uma gramática qualquer. O conjunto dos seus símbolos acessíveis, V_A , pode ser obtido por aplicação das seguintes regras construtivas

```

 $S \in V_A$ 
if  $A \rightarrow \alpha B \beta \in P$  and  $A \in V_A$  then  $B \in V_A$ 

```

Começando com $V_A = \{S\}$ e aplicando sucessivamente a 2ª regra até que ela não acrescente nada a V_A obtém-se o conjunto dos símbolos acessíveis. O algoritmo seguinte executa esse procedimento. Nele, V_A representa o conjunto dos símbolos acessíveis já identificados e N_X o conjunto dos símbolos não terminais acessíveis já identificados mas ainda não processados. No fim, quando N_X for o conjunto vazio, V_A contém todos os símbolos acessíveis.

Algoritmo 5.2 (Cálculo dos símbolos acessíveis)

```

let  $V_A = \{S\}$ ,  $N_X = V_A$ 
repeat
    let  $A =$  um elemento de  $N_X$ 
     $N_X = N_X \setminus \{A\}$ 
    foreach  $(A \rightarrow \alpha) \in P$  do
        foreach  $x$  in  $\alpha$  do
            if  $x \notin V_A$  then
                 $V_A = V_A \cup \{A\}$ 
            if  $x \in N$  then
                 $N_X = N_X \cup \{A\}$ 
until  $N_X = \emptyset$ 

```

5.5.3 Gramáticas limpas

Numa gramática os símbolos inacessíveis e os símbolos improdutivos são **símbolos inúteis**, porque não contribuem para as palavras que a gramática pode gerar. Se tais símbolos forem removidos obtém-se uma gramática equivalente, em termos da linguagem que descreve. Diz-se que uma gramática é **limpa** se não possuir símbolos inúteis.

Para limpar uma gramática deve-se começar por a expurgar dos símbolos improdutivos. Só depois se devem remover os inacessíveis.

5.6 Transformações em gramáticas independentes do contexto

Em muitas situações práticas — algumas serão abordadas nos capítulos seguintes — é necessário transformar uma gramática numa outra que seja equivalente e goze de determinada propriedade. Apresentam-se a seguir algumas dessas transformações.

5.6.1 Eliminação de produções- λ

Uma **produção- λ** é uma produção do tipo $A \rightarrow \lambda$, para um qualquer símbolo não terminal A . Se L é uma linguagem independente do contexto tal que $\lambda \notin L$, é possível descrever L por uma gramática independente do contexto sem produções- λ . Se assim é então tem de ser possível transformar uma gramática que descreva uma linguagem L e possua produções- λ numa outra equivalente que as não possua.

Considere a gramática

$$\begin{aligned} I &\rightarrow 0 \mid I \mid 1 \mid P \\ P &\rightarrow \lambda \mid 0 \mid P \mid 1 \mid I \end{aligned}$$

que descreve a linguagem L formada pelas palavras definidas sobre o alfabeto $\{0, 1\}$, com número ímpar de 1s. Claramente, a palavra vazia não pertence a L porque não tem número ímpar de uns. Mas, a gramática contém a produção $P \rightarrow \lambda$. Então, de acordo com o dito anteriormente, existe uma gramática equivalente que não tem produções- λ .

A existência de tal produção na gramática anterior significa que as produções $I \rightarrow 1P$ e $P \rightarrow 0P$ podem produzir as derivações $I \Rightarrow 1$ e $P \Rightarrow 0$, respectivamente. Mas, estas derivações podem ser contempladas acrescentando as produções $I \rightarrow 1$ e $P \rightarrow 0$ à gramática, tornando desnecessária a produção- λ . Na realidade a gramática

$$\begin{aligned} I &\rightarrow 0 \mid I \mid 1 \mid P \mid 1 \\ P &\rightarrow 0 \mid P \mid 0 \mid 1 \mid I \end{aligned}$$

é equivalente à anterior e não possui produções- λ .

Em geral, o papel da produção $A \rightarrow \lambda$ sobre uma produção $B \rightarrow \alpha A \beta$ pode ser representado pela inclusão da produção $B \rightarrow \alpha \beta$. Assim a eliminação das produções- λ de uma gramática pode ser obtido por aplicação do algoritmo seguinte

Algoritmo 5.3 (Eliminação de produções- λ , 1ª versão)

```
foreach  $A \rightarrow \lambda$  do
  foreach  $B \rightarrow \alpha A \beta$  do
    add  $B \rightarrow \alpha \beta$  to  $P$ .
  remove  $A \rightarrow \lambda$  from  $P$ .
```

O algoritmo anterior pode introduzir novas produções- λ na gramática. Se $B \rightarrow A$ for uma produção da gramática, a eliminação da produção $A \rightarrow \lambda$ introduz a produção $B \rightarrow \lambda$. A algoritmo deve ser alterado de modo a acautelar estas situações.

...

5.6.2 Eliminação de recursividade à esquerda

Diz-se que uma gramática é **recursiva à esquerda** se possuir um símbolo não terminal A que admita uma derivação do tipo $A \Rightarrow^+ A\gamma$, ou seja, que seja possível, em um ou mais passos de derivação, transformar A numa expressão que tem A no início.

A gramática seguinte é recursiva à esquerda.

$$\begin{aligned} E &\rightarrow X \ T \\ X &\rightarrow \lambda \mid E \ + \\ T &\rightarrow a \mid b \mid (\ E \) \end{aligned}$$

A derivação $E \Rightarrow X \ T \Rightarrow E \ + \ T$ mostra que é possível transformar E numa expressão com E à esquerda. Logo, esta gramática tem recursividade à esquerda associada ao símbolo não terminal E .

Se a obtenção da expressão começada por A se faz em apenas um passo de derivação, então diz-se que a recursividade é **imediate**. Esta última situação só ocorre se a gramática possuir uma ou mais produções do tipo $A \rightarrow A \ \alpha$.

No gramática seguinte a recursividade à esquerda é imediata.

$$\begin{aligned} E &\rightarrow T \mid E \ + \ T \\ T &\rightarrow a \mid b \mid (\ E \) \end{aligned}$$

A eliminação de recursividade imediata à esquerda fazer-se com um algoritmo bastante simples. Considere que $A \rightarrow \beta$ e $A \rightarrow A \ \alpha$, onde A é um símbolo não terminal e α e β sequências de zero ou mais símbolos terminais ou não terminais, são duas produções de uma gramática qualquer. Será possível substituir as duas produções por outras que não possuam recursividade à esquerda e produzam uma gramática equivalente? Para responder a esta pergunta observem-se as palavras que se podem obter a partir de A . Numa derivação de um passo obtem-se $A \Rightarrow \beta$. Numa de dois passos obtem-se $A \Rightarrow A\alpha \Rightarrow \beta\alpha$. Numa de n passos, com $n > 0$, obtem-se $A \Rightarrow \beta\alpha^{n-1}$. Mas estas palavras também podem ser obtidas com as produções

$$\begin{aligned} A &\rightarrow \beta \ X \\ X &\rightarrow \lambda \mid \alpha \ X \end{aligned}$$

que não possui recursividade à esquerda. A nova gramática continua a ser recursiva. Na realidade, não pode deixar de o ser, a recursividade passou para à direita.

O algoritmo anterior pode ser facilmente adaptado a situações em que possa haver mais do que uma produção a introduzir a recursividade imediata à esquerda. Considere que

$$A \rightarrow \beta_1 \mid \beta_2 \mid \cdots \beta_m \mid A \ \alpha_1 \mid A \ \alpha_2 \mid \cdots \mid A \ \alpha_n$$

são as produções de uma gramática com A à cabeça. As palavras que se podem gerar com estas produções são as mesmas que se podem gerar com as produções seguintes e que não possuem recursividade à esquerda.

$$A \rightarrow \beta_1 X \mid \beta_2 X \mid \cdots \beta_m X$$

$$X \rightarrow \lambda \mid \alpha_1 X \mid \alpha_2 X \mid \cdots \mid \alpha_n X$$

Se a recursividade à esquerda não é imediata o algoritmo de eliminação é um pouco mais complexo.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.3.3, "Elimination of left recursion".

5.6.3 Factorização à esquerda

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.3.4, "Left factoring".

5.7 Os conjuntos **first**, **follow** e **predict**

A construção de reconhecedores sintáticos (*parsers*) — apresentados nos capítulos seguintes — é auxiliada por duas funções associadas às gramáticas. Estas funções são os conjuntos **first**, **follow** e **predict**.

5.7.1 O conjunto **first**

Seja $G = (T, N, P, S)$ uma gramática qualquer e α uma sequência de símbolos terminais e não terminais, isto é, $\alpha \in (T \cup N)^*$. O conjunto **first**(α) contém todos os símbolos terminais que podem aparecer no início de sequências obtidas a partir de α por aplicação de produções da gramática. Ou seja, um símbolo terminal x pertence a **first**(α) se e só se $\alpha \Rightarrow^* x\beta$, com β qualquer. Adicionalmente, considera-se que λ pertence ao conjunto **first**(α) se $\alpha \Rightarrow^* \lambda$.

Algoritmo de cálculo do conjunto **first**

```
first( $\alpha$ ) {
  if ( $\alpha == \lambda$ ) then
    return { $\lambda$ }
  else if ( $\alpha == a$  and  $a \in T$ ) then
```

```

    return {a}
else if ( $\alpha == B$  and  $B \in N$ ) then
     $M = \{\}$ 
    foreach ( $B \rightarrow \gamma \in P$ )
         $M = M \cup \text{first}(\gamma)$ 
    return  $M$ 
else /*  $|\alpha| > 1$  */
     $x = \text{head}(\alpha)$  /* the first symbol */
     $\beta = \text{tail}(\alpha)$  /* all but the first symbol */
     $M = \text{first}(x)$ 
    if  $\lambda \notin M$  then
        return  $M$ 
    else
        return  $(M - \{\lambda\}) \cup \text{first}(\beta)$ 
}

```

5.7.2 O conjunto follow

O conjunto **follow** está relacionado com os símbolos não terminais de uma gramática. Seja $G = (T, N, P, S)$ uma gramática e A um elemento de N ($A \in N$). O conjunto **follow**(A) contém todos os símbolos terminais que podem aparecer imediatamente à direita de A num processo derivativo qualquer. Formalmente, **follow**(A) = $\{a \in T \mid S \Rightarrow^* \gamma A a \psi\}$, com α e β quaisquer.

O cálculo dos conjuntos **follow** dos símbolos não terminais da gramática $G = (T, N, P, S)$ baseia-se na aplicação das 4 regras seguintes, onde \supseteq significa “contém”.

1. $\$ \in \text{follow}(S)$.
2. se $(A \rightarrow \alpha B) \in P$, então **follow**(B) \supseteq **follow**(A).
3. se $(A \rightarrow \alpha B \beta) \in P$ e $\lambda \notin \text{first}(\beta)$, então **follow**(B) \supseteq **first**(β).
4. se $(A \rightarrow \alpha B \beta) \in P$ e $\lambda \in \text{first}(\beta)$, então **follow**(B) $\supseteq ((\text{first}(\beta) - \{\lambda\}) \cup \text{follow}(A))$.

A primeira regra é óbvia. Sendo o símbolo inicial da gramática, S representa as palavras da linguagem. Logo $\$$ vem a seguir.

A segunda regra diz que se $A \rightarrow \alpha B$ é uma produção da gramática e $x \in \text{follow}(A)$, então $x \in \text{follow}(B)$. Considere, por hipótese, que $x \in \text{follow}(A)$. Então, pela definição de conjunto **follow**, $S \Rightarrow^* \gamma A x \psi$. Logo, sendo $A \rightarrow \alpha B$ uma produção da gramática, tem-se que $S \Rightarrow^* \gamma \alpha B x \psi$, ou seja, $x \in \text{follow}(B)$.

A terceira regra diz que se $A \rightarrow \alpha B \beta$ é uma produção da gramática, com $\lambda \notin \mathbf{first}(\beta)$, e $x \in \mathbf{first}(\beta)$, então $x \in \mathbf{follow}(B)$. Considere, por hipótese, que $x \in \mathbf{first}(\beta)$. Então, pela definição de conjunto **first**, $\beta \Rightarrow^* x \gamma$ e, conseqüentemente, $A \Rightarrow^* \alpha B x \gamma$, ou seja, $x \in \mathbf{follow}(B)$.

Finalmente, a quarta e última regra diz que se $A \rightarrow \alpha B \beta$ é uma produção da gramática, com $\lambda \in \mathbf{first}(\beta)$, e $x \in (\mathbf{first}(\beta) - \{\lambda\}) \cup \mathbf{follow}(A)$, então $x \in \mathbf{follow}(B)$. Esta regra pode ser entendida cruzando as duas regras anteriores. Considere-se os elementos de **first**(β) diferentes de λ . Pela regra 3, pertencem ao **follow**(B). Se β se transforma em λ , então $A \Rightarrow^* \alpha B$ e, pela regra 2, se $x \in \mathbf{follow}(A)$, então $x \in \mathbf{follow}(B)$.

5.7.3 O conjunto **predict**

O conjunto **predict** aplica-se às produções de uma gramática e envolve os conjuntos **first** e **follow**. É dado pela seguinte equação.

$$\mathbf{predict}(A \rightarrow \alpha) = \begin{cases} \mathbf{first}(\alpha) & \lambda \notin \mathbf{first}(\alpha) \\ (\mathbf{first}(\alpha) - \{\lambda\}) \cup \mathbf{follow}(A) & \lambda \in \mathbf{first}(\alpha) \end{cases}$$

Capítulo 6

Gramática de atributos

NOTA PRÉVIA: *Este capítulo é apenas um enumerado das secções do livro de referência ([Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition;](#)) que cobrem a matéria dada sobre gramáticas de atributos.*

6.1 Definição de gramática de atributos

No contexto destes apontamentos considera-se *gramáticas de atributos* o que no livro de referência se designa por *syntax-directed definitions*.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.1, "Syntax-directed definitions".

6.1.1 Atributos herdados e atributos sintetizados

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.1.1, "Inherited and synthesized attributes".

6.1.2 Construção de gramáticas de atributos

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.1.2, "Evaluating an SDD at the nodes of a parse tree".

6.2 Ordem de avaliação dos atributos

6.2.1 Grafo de dependências

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.1, "Dependency graphs".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.2, "Ordering the evaluation of attributes".

6.2.2 Tipos de gramáticas de atributos

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.3, "S-attributed definitions".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.4, "L-attributed definitions".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.5, "Semantic rules with controlled side effects".

Capítulo 7

Análise sintáctica descendente

NOTA PRÉVIA: *Este capítulo não está completo e não foi devidamente revisto, pelo que, por um lado, há partes omissas e, por outro lado, pode conter falhas.*

Dada uma gramática $G = (T, N, P, S)$ e dada uma palavra $u \in T^*$, $u \in L(G)$ se e só se existir uma derivação que produza u a partir de S , isto é, se $S \Rightarrow^* u$. Um **reconhecedor sintáctico** da gramática G é um mecanismo que responde à pergunta “ $u \in L(G)$?”, tentando produzir a derivação anterior. Para o fazer, o reconhecedor pode partir de S e tentar chegar a u ou partir de u e tentar chegar a S . No primeiro caso diz-se que o reconhecedor é **descendente**, porque o seu procedimento corresponde à geração da árvore de derivação da palavra u de cima (raiz) para baixo (folhas).

O papel da análise sintáctica é definir procedimentos que permitam contruir reconhecedores sintácticos a partir da gramática. Por exemplo, considere a linguagem L descrita pela gramática G seguinte.

$$S \rightarrow a S b \mid c S \mid \lambda$$

Será que a palavra $acacbb \in L$? Pertencerá se $S \Rightarrow^* acacbb$. Na verdade pertence porque

$$S \Rightarrow aSb \Rightarrow acSb \Rightarrow acaSbb \Rightarrow acacSbb \Rightarrow acacbb$$

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.4, "Top-down parsing".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.4.3, "LL(1) grammars".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.4.1, "Predictive parsing".

7.1 Reconhecimento preditivo

Mas como deterministicamente produzir a derivação anterior se houver duas ou mais produções com o mesmo símbolo à cabeça? Por exemplo, considerando o caso anterior, como saber que se deve optar por expandir o S usando $S \rightarrow a S b$ e não $S \rightarrow c S$ ou $S \rightarrow \lambda$? A solução baseia-se na observação antecipada dos próximos símbolos à entrada (*lookahead*). No exemplo, se o próximo símbolo à entrada for um a expande-se usando a produção $S \rightarrow a S b$; se for um c usa-se $S \rightarrow c S$; se for b usa-se $S \rightarrow \lambda$. Se a entrada se esgotou, o que é representado considerando que a próxima entrada é um $\$$, também se expande usando a produção $S \rightarrow \lambda$.

Pode-se então definir uma tabela que para cada símbolo não terminal da gramática e para cada valor do *lookahead* indica qual a produção da gramática que deve ser usada na expansão. O profundidade da observação antecipada (número de símbolos de *lookahead*) pode ser qualquer, embora apenas a profundidade 1 será usada neste documento. Para o exemplo anterior a tabela assume a forma

	<i>lookahead</i>			
symbol	a	b	c	\$
S	$S \rightarrow a S b$	$S \rightarrow \lambda$	$S \rightarrow c S$	$S \rightarrow \lambda$

Na tabela anterior, o preenchimento das colunas a e c são óbvias, visto que as produções associadas começam pelo próprio símbolo do *lookahead*. Nas colunas b e $\$$ tal não acontece. O preenchimento da tabela de reconhecimento preditivo baseia-se nos conjuntos **first**, **follow** e **predict**, apresentados na secção 5.7, e faz-se usando o algoritmo seguinte:

Algoritmo 7.1 (Preenchimento da tabela de reconhecimento preditivo)

```

foreach  $(A \rightarrow \alpha) \in P$ 
    foreach  $a \in \text{predict}(A \rightarrow \alpha)$ 
        add  $(A \rightarrow \alpha)$  to  $T[A, a]$ 

```

As células da tabela que fiquem vazias representam situações de erro sintático. As células da tabela que fiquem com dois ou mais produções representam situações de não determinismo: com base no *lookahead* usado não é possível escolher que produção usar na expansão. Uma gramática diz-se **LL(1)** se na tabela de reconhecimento, para um *lookahead* de profundidade 1, não houver células com mais que uma produção. Equivalentemente, uma gramática diz-se LL(1) se para todas as produções com o mesmo símbolo à cabeça os seus conjuntos **predict** são disjuntos entre si.

Exemplo 7.1

Calcule a tabela de reconhecimento de um reconhecedor preditivo para a gramática seguinte.

$$S \rightarrow A B$$

$$A \rightarrow \lambda \mid a A$$

$$B \rightarrow \lambda \mid b B$$

Resposta:

(Deixo ao cuidado do leitor o cálculo dos conjuntos **predict**.)

$$\text{predict}(S \rightarrow A B) = \{a, b, \$\}$$

$$\text{predict}(A \rightarrow \lambda) = \{b, \$\}$$

$$\text{predict}(A \rightarrow a A) = \{a\} \quad l$$

$$\text{predict}(B \rightarrow \lambda) = \{\$\}$$

$$\text{predict}(B \rightarrow b B) = \{b\}$$

	<i>lookahead</i>		
symbol	a	b	\$
<i>S</i>	$S \rightarrow A B$	$S \rightarrow A B$	$S \rightarrow A B$
<i>A</i>	$A \rightarrow a A$	$A \rightarrow \lambda$	$A \rightarrow \lambda$
<i>B</i>		$B \rightarrow b B$	$B \rightarrow \lambda$

7.2 Reconhecedores recursivo-descendentes

O reconhecimento preditivo, sintetizado na tabela de reconhecimento apresentada acima, permite construir programas de reconhecimento, reconhecedores (ou *parsers* na terminologia em inglês). Uma solução para a construção do reconhecedor é baseada numa estrutura na qual cada símbolo não terminal da gramática dá origem a uma função, possivelmente recursiva.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 2.4.2, "Recursive-descent parsing".

Exemplo 7.2

Considere que com base na gramática e na tabela de reconhecimento do exemplo anterior se contrói o programa seguinte.

```
function match(token)
  if lookahead = token then
    lookahead := nextToken().
  else
    REJECT.

function S()
  A(), B().

function A()
  case lookahead in
    a : match(a), A(), return .
    b, $ : return .

function B()
  case lookahead in
    a : REJECT.
    b : match(b), B(), return .
    $ : return .

program Parser()
  lookahead := nextToken(). S().
  if lookahead = $ then
    ACCEPT.
  else
    REJECT.
```

Se executar o programa `Parser` quando a entrada é `aab` verificará que após o retorno da função `S` o `lookahead` é igual a `$`, indicando que a palavra é válida. Mas a palavra `aba` é rejeitada, porque durante a execução a função `B` vai ser invocada numa altura em que o `lookahead` é igual a `a`. (Confirme.)

7.3 Reconhedores descendentes não recursivos

Uma solução alternativa para implementar o reconhecedor preditivo usa uma pilha (*stack*) para reter o estado no processo de reconhecimento e usa a tabela de reconhecimento preditivo para decidir como evoluir no processo de reconhecimento.

Seja $G = (T, N, P, S)$ uma gramática independente do contexto, que se assume seja LL(1). Seja M a tabela de reconhecimento preditivo de G para um *lookahead* de profundidade 1. Finalmente, considere

que dispõe de uma pilha onde pode armazenar elementos do conjunto $Z = T \cup N \cup \{\$\}$ e que pode ser manipulada com as funções **push**, **pop** e **top**, que, respetivamente, coloca uma sequência de símbolos na pilha, retira o símbolo no topo da pilha e mostra qual o símbolo no topo sem o retirar. O programa seguinte é um reconhecedor das palavras da gramática G

```

program Parser()
  push(S $) .
  lookahead = getToken() .
  forever
     $z := \text{top}()$  .
    if  $z \in T$  then
      if  $z \neq \text{lookahead}$  then
        REJECT .
      elseif  $z = \$$  then
        ACCEPT .
      else ( $* z = \text{lookahead} \wedge z \neq \$ *$ )
        pop(), lookahead = getToken() .
    else ( $* z \in N *$ )
       $\alpha := M(z, \text{lookahead})$  .
      if  $\alpha = \emptyset$  then
        REJECT .
      else
        pop(), push( $\alpha$ ) .

```

A evolução do programa anterior no processo de reconhecimento pode ser captado por uma tabela onde se mostre a cada passo os estados da pilha e da entrada e a ação tomada. Se se considerar a gramática e a tabela de reconhecimento do exemplo 1, a execução do programa anterior sobre a palavra *aab* resulta na seguinte tabela. Na coluna da pilha, o símbolo mais à direita é o que está no topo e, na coluna entrada, o símbolo mais à esquerda é o *lookahead*.

Pilha	Entrada	Ação
\$ S	a a b \$	pop() , push(A B)
\$ B A	a a b \$	pop() , push(a A)
\$ B A a	a a b \$	pop() , lookahead = getToken()
\$ B A	a b \$	pop() , push(a A)
\$ B A a	a b \$	pop() , lookahead = getToken()
\$ B A	b \$	pop()
\$ B	b \$	pop() , push(b B)
\$ B b	b \$	pop() , lookahead = getToken()
\$ B	\$	pop()
\$	\$	ACCEPT

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.4.4, "Nonrecursive predictive parsing".

7.4 Implementação de gramáticas de atributos

Os reconhecedores recursivo-descendentes (ver secção 7.2) permitem implementar facilmente gramáticas de atributos do tipo L (ver secção 6.2.2). As funções recursivas podem ter parâmetros de entrada e de saída. Os primeiros permitem passar informação da função chamadora para a função chamada, o que corresponde, na árvore de derivação, a definir atributos herdados dos nós filhos com base em atributos do nó pai. Os segundos permitem passar informação da função chamada para a função chamadora, o que corresponde ao suporte de atributos sintetizados e de atributos herdados de nós filhos com base em atributos de nós filhos à esquerda na árvore de derivação.

Capítulo 8

Análise sintáctica ascendente

NOTA PRÉVIA: *Este capítulo não está completo e não foi devidamente revisto, pelo que, por um lado, há partes omissas e, por outro lado, pode/deve conter falhas.*

Considere a gramática

$$D \rightarrow T L ;$$

$$T \rightarrow i \mid r$$

$$L \rightarrow v \mid L , v$$

que representa uma declaração de variáveis *a la* C. Como reconhecer a palavra “ $u = i \ v , v ;$ ” como pertencente à linguagem gerada pela gramática dada? Se u pertence à linguagem gerada pela gramática, então $D \Rightarrow^* u$. Tente-se chegar lá andando no sentido contrário ao de uma derivação, ie. de u para D .

$$i \ v , v ;$$

$$\Leftarrow T \ v , v ; \quad (\text{por aplicação da regra } T \rightarrow i)$$

$$\Leftarrow T \ L , v ; \quad (\text{por aplicação da regra } L \rightarrow v)$$

$$\Leftarrow T \ L ; \quad (\text{por aplicação da regra } L \rightarrow L , v)$$

$$\Leftarrow D \quad (\text{por aplicação da regra } D \rightarrow T \ L ;)$$

Colocando ao contrário

$$D \Rightarrow T \ L ; \Rightarrow T \ L , v ; \Rightarrow T \ v , v ; \Rightarrow i \ v , v ;$$

vê-se que corresponde a uma derivação à direita. A tabela seguinte mostra como, na prática, se realiza esta (retro)derivação.

pilha	entrada	ação
\$	i v , v ; \$	deslocamento
\$ i	v , v ; \$	redução por $T \rightarrow i$
\$ T	v , v ; \$	deslocamento
\$ T v	, v ; \$	redução por $L \rightarrow v$
\$ T L	, v ; \$	deslocamento
\$ T L ,	v ; \$	deslocamento
\$ T L , v	; \$	redução por $L \rightarrow L , v$
\$ T L	; \$	deslocamento
\$ T L ;	\$	redução por $D \rightarrow T L ;$
\$ D	\$	aceitação

Inicialmente, o topo da pilha apenas possui um símbolo especial, representado por um \$, que indica, quando no topo, que a pilha está vazia. A entrada possui a palavra a reconhecer seguida também de um símbolo especial, aqui também representado por um \$, que indica fim da entrada. Em cada ciclo realiza-se, normalmente, uma operação de deslocamento ou de redução. A operação de **deslocamento** (no inglês, *shift*) transfere o símbolo não terminal da entrada para o topo da pilha. A operação de **redução** (no inglês, *reduce*) substitui os símbolos do topo da pilha que correspondem ao corpo de uma produção da gramática pela cabeça dessa regra.

Se se atingir uma situação em que a entrada apenas possui o símbolo \$ e a pilha apenas possui o símbolo \$ e o símbolo inicial da gramática, tal como acontece na tabela anterior, a palavra é reconhecida como pertencendo à linguagem descrita pela gramática. Caso contrário, a palavra é rejeitada.

Veja-se a reação deste procedimento a uma entrada errada, por exemplo a palavra i v v ; .

pilha	entrada	ação
\$	i v v ; \$	deslocamento
\$ i	v v ; \$	redução por $T \rightarrow i$
\$ T	v v ; \$	deslocamento
\$ T v	v ; \$	rejeição

Com $T v$ na pilha e v na entrada é impossível chegar-se à aceitação. Porque se se reduzir v para L ficar-se-ia com um $T L$ na pilha e v na entrada, que não pertence ao conjunto **follow**(L). Mais à frente voltaremos a este assunto.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.5.1, "Reductions".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.5.2, "Handle pruning".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.5.3, "Shift-reduce parsing".

8.1 Conflitos

O procedimento acabado de descrever pode acarretar situações ambíguas chamadas **conflitos**. Considere a gramática

$$\begin{aligned} S &\rightarrow i \ c \ S \\ &| i \ c \ S \ e \ S \\ &| a \end{aligned}$$

e execute o procedimento de reconhecimento com a palavra $i \ c \ i \ c \ a \ e \ a$. Obtém-se

pilha	entrada	ação
\$	$i \ c \ i \ c \ a \ e \ a \ \$$	deslocamento
\$ i	$c \ i \ c \ a \ e \ a \ \$$	deslocamento
\$ i c	$i \ c \ a \ e \ a \ \$$	deslocamento
\$ i c i	$c \ a \ e \ a \ \$$	deslocamento
\$ i c i c	$a \ e \ a \ \$$	deslocamento
\$ i c i c a	$e \ a \ \$$	redução por $S \rightarrow a$
\$ i c i c S	$e \ a \ \$$	conflito: redução usando $S \rightarrow i \ c \ S$ ou deslocamento para tentar $S \rightarrow i \ c \ S \ e \ S$?

Na última linha é possível reduzir-se por aplicação da regra $S \rightarrow i \ c \ S$ ou deslocar-se o e para tentar posteriormente a redução pela regra $S \rightarrow i \ c \ S \ e \ S$. Trata-se de um conflito deslocamento-redução (*shift-reduce conflict*). Perante este tipo de conflitos, ferramentas como o *bison* optam pelo deslocamento, mas pode não ser a mais adequada.

Também pode haver conflitos entre reduções (*reduce-reduce conflict*). Considere a gramática

$$\begin{aligned} S &\rightarrow A \\ &\quad | B \\ A &\rightarrow c \\ &\quad | A a \\ B &\rightarrow c \\ &\quad | B a \end{aligned}$$

e a palavra c . O procedimento de reconhecimento produz

pilha	entrada	ação
\$	c \$	deslocamento
\$ c	\$	conflito: redução usando $A \rightarrow c$ ou $B \rightarrow c$?

Na última linha é possível reduzir-se por aplicação das regras $A \rightarrow c$ ou $B \rightarrow c$. Perante este tipo de conflitos, ferramentas como o *bison* optam pela produção que aparece primeiro. Neste caso é irrelevante, mas pode não ser o adequado.

Veja-se agora a situação de um falso conflito. Considere a gramática

$$\begin{aligned} S &\rightarrow a \mid (S) \mid a P \mid (S) S \\ P &\rightarrow (S) \mid (S) S \end{aligned}$$

e reconheça-se a palavra $a (a) a$.

pilha	entrada	ação
\$	$a (a) a$ \$	deslocamento
\$ a	$(a) a$ \$	redução usando $S \rightarrow a$
		deslocamento para tentar $S \rightarrow a P$?

Considerar a redução corresponde a realizar a retro-derivação

$$a (a) a \leftarrow S (a) a$$

que não faz sentido porque $(\notin \text{follow}(S)$. Não há, portanto, conflito, sendo realizado o deslocamento.

pilha	entrada	ação
\$	a (a) a \$	deslocamento
\$ a	(a) a \$	deslocamento
\$ a (a) a \$	deslocamento
\$ a (a) a \$	redução por $S \rightarrow a$
\$ a (S) a \$	deslocamento
\$ a (S)	a \$	deslocamento, porque $a \notin \text{follow}(S), \text{follow}(P)$
\$ a (S) a	\$	redução por $S \rightarrow a$
\$ a (S) S	\$	redução por $P \rightarrow (S) S$
\$ a P	\$	redução por $S \rightarrow a P$
\$ S	\$	aceitação

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.5.4, "Conflicts during shift-reduce parsing".

Pode-se alterar a gramática de modo a eliminar a fonte de conflito. Considerando que se pretendia optar pelo deslocamento, a gramática seguinte gera a mesma linguagem e está isenta de conflitos.

$$\begin{aligned}
 &S \rightarrow a \\
 &\quad | i \text{ c } S \\
 &\quad | i \text{ c } S' \text{ e } S \\
 &S' \rightarrow a \\
 &\quad | i \text{ c } S' \text{ e } S'
 \end{aligned}$$

8.2 Construção de um reconhecedor ascendente

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.6, "Introduction to LR parsing: Simple LR".

O procedimento de reconhecimento apresentado atrás é um mecanismo iterativo. Em cada passo, a operação a realizar — deslocamento, redução, aceitação ou rejeição — depende da configuração nesse momento. Uma *configuração* é formada pelo conteúdo da pilha mais a parte da entrada ainda não processada. A pilha é conhecida — na realidade, é preenchida pelo procedimento de reconhecimento —,

mas a entrada é desconhecida, conhecendo-se apenas o próximo símbolo (*lookahead*). Então a decisão a tomar só pode basear-se no conteúdo da pilha e no *lookahead*.

Mas, se se quiser construir um reconhecedor apenas com capacidade de observar o topo da pilha, uma pilha onde se guardam os símbolos terminais e não terminais tem pouco interesse. Deve-se guardar um símbolo que represente tudo o que está para trás.

Como definir esses símbolos?

A associação de um símbolo diferente por cada configuração da pilha não serve porque a pilha pode, em geral, crescer de forma não limitada. Os símbolos a colocar na pilha devem representar estados no processo de deslocamento-redução. O alfabeto da pilha representa assim o conjunto de estados nesse processo de reconhecimento.

Cada estado representa um conjunto de itens. Um item de uma gramática é uma produção com um ponto (*dot*) numa posição do seu corpo. Por exemplo, a produção $A \rightarrow B_1 B_2 B_3$, produz 4 itens, a saber

$$A \rightarrow \cdot B_1 B_2 B_3$$

$$A \rightarrow B_1 \cdot B_2 B_3$$

$$A \rightarrow B_1 B_2 \cdot B_3$$

$$A \rightarrow B_1 B_2 B_3 \cdot$$

A produção $A \rightarrow \varepsilon$ produz um único item, $A \rightarrow \cdot$.

Um item representa o quanto de uma produção já foi obtido e, simultaneamente, o quanto falta obter. Por exemplo, $A \rightarrow B_1 \cdot B_2 B_3$, significa que já foi obtido algo correspondente a B_1 , faltando obter o correspondente a $B_2 B_3$. Se o ponto(·) se encontra à direita, então poder-se-á reduzir $B_1 B_2 B_3$ a A .

8.2.1 Construção da coleção de conjuntos de itens

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.6.2, "Items and the LR(0) automaton".

Considere a gramática

$$S \rightarrow E$$

$$E \rightarrow a \mid (E)$$

O estado inicial (primeiro elemento da coleção de conjunto de itens) contém o item

$$Z_0 = \{S \rightarrow \cdot E \$\}$$

Este conjunto tem de ser fechado. O facto de o ponto (\cdot) se encontrar imediatamente à esquerda de um símbolo não terminal, significa que para se avançar no processo de reconhecimento é preciso obter esse símbolo. Isso é considerado juntando ao conjunto Z_0 os itens iniciais das produções cuja cabeça é E . Fazendo-o, Z_0 passa a

$$Z_0 = \{ S \rightarrow \cdot E \$ \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (E) \}$$

Se nos novos elementos adicionados ao conjunto voltasse à acontecer de o ponto (\cdot) ficar imediatamente à esquerda de outros símbolos não terminais o processo deve ser repetido para esses símbolos.

O estado Z_0 pode evoluir por ocorrência de um E , um a ou um $($. Correspondem aos símbolos que aparecem imediatamente à direita do ponto (\cdot), e produzem 3 novos estados

$$Z_1 = \delta(Z_0, E) = \{ S \rightarrow E \cdot \$ \}$$

$$Z_2 = \delta(Z_0, a) = \{ E \rightarrow a \cdot \}$$

$$Z_3 = \delta(Z_0, () = \{ E \rightarrow (\cdot E) \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (E) \}$$

Note que Z_3 foi extendido pela função de fecho, uma vez que o ponto ficou imediatamente à esquerda de um símbolo não terminal (E). Z_1 representa um situação de aceitação se o símbolo à entrada (*lookahead*) for igual a $\$$ e de erro caso contrário. Z_2 representa uma possível situação de redução pela regra $E \rightarrow a$. Esta redução só faz sentido se o símbolo à entrada (*lookahead*) for um elemento do conjunto **follow**(E). Caso contrário corresponde a uma situação de erro. Finalmente, Z_3 pode evoluir por ocorrência de um E , um a ou um $($, que correspondem aos símbolos que aparecem imediatamente à direita do ponto (\cdot). Estas evoluções são indicadas a seguir

$$Z_4 = \delta(Z_3, E) = \{ E \rightarrow (E \cdot) \}$$

$$\delta(Z_3, a) = Z_2$$

$$\delta(Z_3, () = Z_3$$

Apenas um novo estado foi gerado (Z_4). Este estado apenas evolui por ocorrência de $)$.

$$Z_5 = \delta(Z_4,) = \{ E \rightarrow (E) \cdot \}$$

Pondo tudo agrupado, a coleção de conjunto de itens é

$$Z_0 = \{ S \rightarrow \cdot E \$ \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (E) \}$$

$$Z_1 = \delta(Z_0, E) = \{ S \rightarrow E \cdot \$ \}$$

$$Z_2 = \delta(Z_0, a) = \{ E \rightarrow a \cdot \}$$

$$Z_3 = \delta(Z_0, () = \{ E \rightarrow (\cdot E) \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (E) \}$$

$$Z_4 = \delta(Z_3, E) = \{ E \rightarrow (E \cdot) \}$$

$$Z_5 = \delta(Z_4,) = \{ E \rightarrow (E) \cdot \}$$

8.2.2 Tabela de reconhecimento

A coleção de conjuntos de itens (conjunto de estados) fornece a base para a construção de uma tabela usada no algoritmo de reconhecimento. A tabela de reconhecimento é uma matriz dupla, em que as linhas são indexadas pelo alfabeto da pilha (coleção de conjuntos de itens) e as colunas são indexadas pelos símbolos terminais e não terminais da gramática. Representa simultaneamente duas funções, designadas ACTION e GOTO. A função ACTION tem como argumentos um estado (símbolo da pilha) e um símbolo terminal (incluindo o \$) e define a ação a realizar. Pode ser uma de *shift*, *reduce*, *accept* ou *error*. A função GOTO mapeia um estado e um símbolo não terminal num estado. É usada após uma operação de redução.

Veja-se um exemplo. Considerando a gramática e a coleção de conjunto de itens anteriores, obtem-se a seguinte tabela de reconhecimento.

	a	()	\$	E
Z_0	shift, Z_2	shift, Z_3			Z_1
Z_1				accept	
Z_2			reduce, $E \rightarrow a$	reduce, $E \rightarrow a$	
Z_3	shift, Z_2	shift, Z_3			Z_4
Z_4			shift, Z_5		
Z_5			reduce, $E \rightarrow (E)$	reduce, $E \rightarrow (E)$	

- $Z = \{Z_0, Z_1, Z_2, Z_3, Z_4, Z_5\}$ é o alfabeto da pilha e foi obtida calculando a coleção de conjuntos de itens.
- shift, Z_i , com $Z_i \in Z$, representa um deslocamento, no qual é consumido o símbolo à entrada e é feito o empilhamento do símbolo Z_i .
- reduce, $A \rightarrow \alpha$, onde $A \rightarrow \alpha$ é uma produção da gramática, representa uma redução, na qual são retirados da pilha tantos símbolo quantos os símbolo do corpo da regra.
- Os símbolos $Z_i \in Z$ na última coluna, representam os símbolos a empilhar após uma redução.
- accept representa a aceitação.
- As células vazias representam situações de erro.

8.2.3 Algoritmo de reconhecimento

O algoritmo seguinte mostra como se usa a tabela anterior. Nele, `top`, `push` e `pop` são funções de manipulação da pilha, com os significados habituais, e `lookahead` e `adv` são funções de manipulação da entrada que, respetivamente, devolve o próximo símbolo terminal à entrada e consome um símbolo.

Algoritmo 8.1

```

push( $Z_0$ )
forever
  if ( $\text{top}() == Z_1 \ \&\& \ \text{lookahead}() == \$$ )
    aceita a entrada como pertencendo à linguagem.
   $\text{acc} = \text{table}[\text{top}(), \text{lookahead}()]$ 
  if ( $\text{acc}$  is shift  $Z_i$ )
     $\text{adv}(); \text{push}(Z_i);$ 
  else if ( $\text{acc}$  is reduce  $A \rightarrow \alpha$ )
     $\text{pop}(|\alpha| \text{ símbolos}; \text{push}(\text{table}[\text{top}(), A]);$ 
  else
    rejeita a entrada

```

A aplicação deste algoritmo à palavra $((a))$ resulta na tabela seguinte. No preenchimento dessa tabela, optou-se por separar em duas linhas as operações de *pop* e *push* das ações de redução. Desta forma fica mais claro que o símbolo a empilhar resulta do símbolo no topo da pilha após os *pops*.

pilha	entrada	ação
Z_0	$((a)) \$$	shift Z_3
$Z_0 \ Z_3$	$(a)) \$$	shift Z_3
$Z_0 \ Z_3 \ Z_3$	$a)) \$$	shift Z_2
$Z_0 \ Z_3 \ Z_3 \ Z_2$	$)) \$$	reduce $E \rightarrow a$
$Z_0 \ Z_3 \ Z_3$	$)) \$$	goto Z_4
$Z_0 \ Z_3 \ Z_3 \ Z_4$	$)) \$$	shift Z_5
$Z_0 \ Z_3 \ Z_3 \ Z_4 \ Z_5$	$) \$$	reduce $E \rightarrow (E)$
$Z_0 \ Z_3$	$) \$$	goto Z_4
$Z_0 \ Z_3 \ Z_4$	$) \$$	shift Z_5
$Z_0 \ Z_3 \ Z_4 \ Z_5$	$\$$	reduce $E \rightarrow (E)$
Z_0	$\$$	goto Z_1
$Z_0 \ Z_1$	$\$$	accept

Na redução com a produção $E \rightarrow a$ foi feito o `pop` de 1 símbolo (número de símbolos no corpo da produção), ficando, em consequência, um Z_3 no topo da pilha. O Z_4 que foi empilhado logo a seguir corresponde a `table`[Z_3, E]. Nas duas reduções com a produção $E \rightarrow (E)$ são feitos o `pop` de 3 símbolos, ficando, em consequência, um Z_3 no topo da pilha, no primeiro caso, e um Z_0 no segundo.