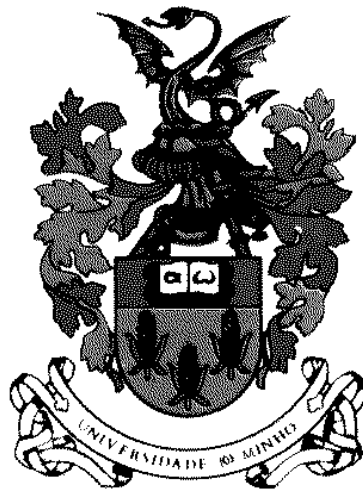


Universidade do Minho  
Escola de Engenharia  
**Departamento de Informática**

Largo do Paço • 4719 BRAGA Codex• PORTUGAL

Tel. +351-53-604470• Fax +351-53-612954



**Especificação e Processamento de Linguagens**

João Alexandre Saraiva

**Texto Pedagógico**  
**versão 0.2**

1995, Março



# **Especificação e Processamento de Linguagens**

João Alexandre Saraiva

Departamento de Informática  
Universidade do Minho  
Largo do Paço  
4719 BRAGA Codex  
PORTUGAL

1995, Março



# Índice

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Estrutura do Documento . . . . .	3
<b>2</b>	<b>Linguagens</b>	<b>5</b>
2.1	Linguagens Formais . . . . .	5
2.2	Expressões Regulares . . . . .	9
2.2.1	Álgebra de Expressões Regulares . . . . .	12
2.2.2	Expressões Regulares em UNIX . . . . .	13
2.2.3	Limitações das Expressões Regulares . . . . .	16
2.3	Gramáticas . . . . .	18
2.3.1	Definição de Gramática . . . . .	19
2.3.2	Gramáticas Independentes do Contexto . . . . .	20
2.3.3	Derivação de Frases de uma Gramática . . . . .	24
2.3.4	Gramáticas Regulares . . . . .	33
2.3.5	Conversão de Gramáticas em Expressões Regulares . . . . .	35
2.3.6	Limitações das Gramáticas . . . . .	37
<b>3</b>	<b>Reconhecimento de Linguagens</b>	<b>41</b>
3.1	Reconhecedor . . . . .	42
3.2	Reconhecimento de Linguagens Regulares . . . . .	43
3.2.1	Autómatos Finitos . . . . .	43
3.2.2	Conversão de Gramáticas Regulares em AFND . . . . .	50
3.2.3	Conversão de Expressões Regulares em AFND . . . . .	52
3.2.4	Conversão de AFND em AFD . . . . .	57
3.2.5	Conversão de Gramáticas e Expressões Regulares em AFD . . . . .	63
3.2.6	Reconhedores baseados em AFD . . . . .	63
3.2.7	Análise de um Problema . . . . .	65

3.3	Reconhecimento de Linguagens Não-Regulares Top-Down . . .	68
3.3.1	Funcionamento Geral . . . . .	69
3.3.2	Condição LL(1) . . . . .	70
3.3.3	Transformações Essenciais à Condição LL(1) . . . . .	79
3.3.4	Reconhecedor Recursivo Descendente . . . . .	83
3.3.5	Reconhecedor Dirigido por Tabela . . . . .	88
3.4	Reconhedores de Linguagens Não-Regulares Bottom-Up . . .	94
3.4.1	Reconhecedor Desloca/Reduz . . . . .	95
3.4.2	Construção das Tabelas de Parsing . . . . .	99
3.4.3	Algoritmo de Reconhecimento LR . . . . .	106
3.4.4	Conflitos no Reconhecimento LR . . . . .	110
<b>4</b>	<b>Processamento de Linguagens</b>	<b>113</b>
4.1	Autómatos Finitos com Acções Semânticas . . . . .	113
4.2	Analísadores Léxicos . . . . .	117
4.2.1	Construção de Geradores de Analísadores Léxicos . . .	122
4.2.2	Gerador de Analísadores Léxicos: lex . . . . .	124
4.3	Tradução Dirigida pela Sintaxe . . . . .	127
4.3.1	Gramática Tradutora . . . . .	129
4.3.2	Tradução Dirigida pela Sintaxe Top-Down . . . . .	131
4.3.3	Tradução Dirigida pela Sintaxe Bottom-Up . . . . .	133
4.3.4	Estudo de um Caso . . . . .	136
<b>A</b>	<b>Meta-Gramática</b>	<b>149</b>
<b>B</b>	<b>Implementação de um Reconhecedor de uma Linguagem Regular</b>	<b>151</b>
<b>C</b>	<b>Implementação de um Processador de uma Linguagem Regular</b>	<b>153</b>
<b>D</b>	<b>Implementação de um Processador Top-Down</b>	<b>157</b>
	<b>Bibliografia</b>	<b>165</b>

# Lista de Exercícios

## Linguagens

PÁGINA	NÚMERO	PÁGINA	NÚMERO
8	2.1	30	2.9
10	2.2	32	2.10
10	2.3	32	2.11
11	2.4	33	2.12
13	2.5	35	2.13
24	2.6	35	2.14
27	2.7	36	2.15
27	2.8	37	2.16

## Reconhecimento de Linguagens

PÁGINA	NÚMERO	PÁGINA	NÚMERO
47	3.1	78	3.11
56	3.2	79	3.12
62	3.3	93	3.13
62	3.4	94	3.14
63	3.5	94	3.15
68	3.6	100	3.16
73	3.7	111	3.17
74	3.8	112	3.18
77	3.9	112	3.19
78	3.10	112	3.20

## Processamento de Linguagens

PÁGINA	NÚMERO
122	4.1
127	4.2
135	4.3
147	4.4



# Capítulo 1

## Introdução

Uma *linguagem* é um "mecanismo" que permite a comunicação entre dois objectos distintos. Esta definição bastante abstracta inclui um grande número de disciplinas, desde as várias formas de arte (pintura, música, cinema, etc), até à *linguagem natural* (*i.e.*, a linguagem utilizada pelo homem).

Neste texto considera-se apenas um subconjunto destas linguagens, mais concretamente as *linguagens* formadas por um conjunto finito ou infinito de *frases* (todas com comprimento finito), construídas a partir de um conjunto finito de *símbolos* (também designado *alfabeto*). De acordo com esta definição qualquer linguagem natural, na sua forma oral ou escrita, é uma *linguagem*, pois toda a linguagem natural possui um conjunto de símbolos (designados *lexemas*) e cada frase é uma sequência finita desses símbolos, existindo um número infinito de frases. De igual modo o conjunto de frases de um sistema formal matemático é uma linguagem, pois satisfaz essas características.

Sendo as frases de uma linguagem sequências de símbolos do seu alfabeto, então existem inúmeras combinações possíveis desses mesmos símbolos. Porém, possivelmente nem todas as combinações (de símbolos da linguagem) são frases pertencentes à linguagem. Por exemplo, considerando o *Português* nem todas as combinações dos seus lexemas são frases válidas. Nesta linguagem as frases têm de obedecer a um *conjunto de regras*. Tipicamente, uma frase em Português começa pelo *sujeito* seguida do *predicado* e terminada por um *complemento*.

A *forma* como os diferentes símbolos do alfabeto se combinam numa frase constitui a sintaxe da linguagem. A sintaxe de uma linguagem pode ser especificada por um conjunto de regras, designadas *regras sintáticas*. Uma

frase que obedece às regras sintáticas de uma linguagem diz-se *sintaticamente válida* ou *bem formada*.

O *significado* de uma frase constitui a semântica da linguagem. No entanto, uma frase dita sintaticamente válida pode não ser semanticamente correcta. Isto acontece porque embora a combinação de símbolos obedeça às regras sintáticas da linguagem, o significado da frase resultante não faz sentido.

Sendo assim, verificar se uma frase  $f$  pertence a uma linguagem  $\mathcal{L}$  faz-se a dois níveis: primeiro verifica-se se  $f$  cumpre as regras sintáticas de  $\mathcal{L}$  e posteriormente verifica-se se  $f$  obedece à semântica de  $\mathcal{L}$ . A primeira tarefa diz-se *reconhecimento sintático* da linguagem (abreviadamente *reconhecimento*). Um programa de computador que efectua o reconhecimento sintático de uma linguagem designa-se *reconhecedor*. O *processamento* de uma linguagem inclui além do reconhecimento sintático a análise semântica e a *tradução* da linguagem para uma outra representação (possivelmente uma outra linguagem).

Neste texto irão ser estudadas as *linguagens de programação*, *i.e.*, linguagens usadas para interagir com um computador. Estas linguagens tem a particularidade da sintaxe e a semântica serem bastante mais simples que nas linguagens naturais e, deste modo, poderem ser descritas recorrendo a um modelo formal. Por este motivo designam-se *linguagens formais*. Sendo uma linguagem um conjunto de frases então um método para descrever esse conjunto será por enumeração dos seus elementos. Deste modo, verificar se uma frase pertence à linguagem consiste em determinar se essa frase está contida no conjunto de frases válidas da linguagem. É óbvio que este método não é viável, uma vez que as linguagens com interesse prático são constituídas por um conjunto infinito de frases. Sendo assim, é necessário estudar métodos finitos que permitam definir um conjunto infinito de frases.

Neste documento irão ser analisados dois mecanismos distintos: as *expressões regulares* e as *gramáticas*. Quer as *expressões regulares* quer as *gramáticas* permitem descrever, de um modo sucinto e bastante legível, a sintaxe de uma linguagem de programação. Porém, a simples especificação da sintaxe de uma linguagem não é suficiente para se obter um *reconhecedor* dessa linguagem. Assim, serão ainda apresentadas metodologias bem definidas que permitem converter uma *expressão regular/gramática* num *reconhecedor* da linguagem definida por essa mesma *expressão regular/gramática*. As metodologias que irão ser estudadas baseiam-se em *autómatos finitos* e *reco-*

*nhecedores top-down e bottom-up* (*i.e.*, descendentes e ascendentes, respectivamente). Posteriormente extendem-se estes métodos com acções semânticas de modo a ser possível efectuar a análise semântica e a tarefa de tradução, referidas anteriormente.

## 1.1 Estrutura do Documento

Este documento foi inicialmente elaborado com a intenção de apoiar as aulas das disciplinas de Linguagens de Programação e Técnicas de Programação das Licenciaturas em Informática de Gestão e Engenharia em Electrónica Industrial, no ano lectivo 1993/1994. Uma primeira versão do capítulo 2 deste texto foi escrita de parceria com o Eng. José Carlos Bacelar [BS93].

Este texto não tem qualquer intenção de originalidade. Basicamente, baseia-se em vários documentos produzidos no Departamento de Informática da Universidade do Minho [Val86, AB90, Oli91, Hen92] e ainda na "bíblia" da teoria da compilação, o livro [ASU86] (geralmente designado o "dragão"<sup>1</sup>).

Os exemplos analisados, os problemas resolvidos e os exercícios propostos são na sua maioria questões que foram estudadas nas aulas das disciplinas referidas e ainda questões propostas nos exames respectivos.

Os algoritmos apresentados ao longo do texto estão escritos na notação adoptada no Departamento de Informática [Mar92].

Em apêndice apresenta-se a implementação em linguagem C de alguns dos problemas analisados. O código C aí apresentado pode ser obtido via WWW no seguinte endereço:

`http://www.di.uminho.pt/~jas/teaching`

---

<sup>1</sup>Nome devido ao dragão que aparece na sua capa.



# Capítulo 2

## Linguagens

### 2.1 Linguagens Formais

Para definir o que é uma linguagem, vai-se primeiro definir alguns conceitos que lhe estão associados. O primeiro conceito é o de *vocabulário* ou *alfabeto* da linguagem, geralmente representado por  $V$ , que denota um conjunto finito de símbolos. Um exemplo típico de símbolos das linguagens são as letras e os dígitos. Um outro exemplo é o conjunto  $\{0, 1\}$  que constitui o alfabeto binário.

A uma sequência finita de símbolos de um alfabeto, chama-se *frase* ou *string* desse alfabeto. O comprimento de uma frase ou comprimento da *string* é o número de símbolos do alfabeto que ela contém. Geralmente, o comprimento de uma frase " $s$ " escreve-se por  $|s|$ . Por exemplo, o comprimento da frase "comprimento", denotado por  $|\text{comprimento}|$ , é 11. A *frase nula*, denotada por  $\epsilon$ , é uma frase especial que tem comprimento zero, *i.e.*,  $|\epsilon| = 0$ . Uma operação que usualmente se aplica a frases é a sua *concatenação*. A concatenação da frase  $\alpha$  com a  $\beta$ , denotado por  $\alpha\beta$ , é uma frase constituída pelos símbolos de  $\alpha$  seguidos pelos símbolos de  $\beta$ . Supondo  $\alpha = s_1s_2 \cdots s_n$  e  $\beta = t_1t_2 \cdots t_k$ , então  $\alpha\beta = s_1s_2 \cdots s_nt_1t_2 \cdots t_k$ . Por exemplo, sendo  $\alpha = abc$  e  $\beta = def$ , então  $\alpha\beta = abcdef$  e  $\beta\alpha = defabc$ . Facilmente se verifica que  $a\epsilon = \epsilon a = a$ , *i.e.*, a frase nula é o elemento neutro da concatenação. A concatenação obedece à propriedade associativa, pois  $(\alpha\beta)\gamma = \alpha(\beta\gamma)$ , mas não à propriedade comutativa:  $\alpha\beta \neq \beta\alpha$ . A concatenação de uma frase  $\alpha$  com

ela própria escreve-se  $\alpha\alpha$ . No entanto, para tornar mais simples a escrita é usual escrever-se  $\alpha^2 = \alpha\alpha$ . Genericamente, escreve-se  $\alpha^n$  para representar a frase que se obtém concatenando  $\alpha$   $n$  vezes. Esta operação, designada *exponenciação*, define-se do seguinte modo:  $s^0 = \epsilon$  e para  $i > 0$ ,  $s^i = ss^{i-1}$ . Assim, sendo  $a = pa$  então  $a^2 = papa$ .

Uma frase  $r$  é uma *subfrase* de  $t$  se os símbolos de  $r$  aparecem consecutivamente nos símbolos de  $t$ . Se a subfrase  $r$  aparece no início de  $t$ , então  $r$  é um *prefixo* de  $t$ . Se  $r$  aparecer no fim de  $t$ , então  $r$  é um *sufixo* de  $t$ . Por exemplo, *otorrino* é uma subfrase e um prefixo de *otorrinolaringologista*, *logista* é uma sua subfrase e um seu sufixo e *nolaringo* é apenas uma subfrase.

De notar que  $\epsilon$  é um prefixo, sufixo e subfrase de qualquer frase, enquanto toda a frase é prefixo, sufixo e subfrase de ela própria.

O conjunto de todas as frases sobre um dado alfabeto  $V$  denota-se por  $V^*$ . Uma *linguagem* é um conjunto, finito ou infinito, de frases sobre um alfabeto. Por exemplo, o conjunto  $\{\text{Isto, é, uma, linguagem}\}$  é uma linguagem constituída por quatro frases. Porém, nem todas as linguagens são conjuntos finitos. O conjunto de frases sobre  $V = \{a, b\}$  que contém o mesmo número de  $a$ 's e de  $b$ 's é um exemplo de uma linguagem com um número infinito de frases.

Uma definição mais precisa de linguagem apresenta-se a seguir.

**Definição 2.1** *Uma linguagem  $\mathcal{L}$  sobre um dado alfabeto  $V$ , é um subconjunto de  $V^*$ , i.e.,  $\mathcal{L} \subseteq V^*$ .*

Exemplo: Seja  $V$  o alfabeto formado pelos símbolos  $a$  e  $b$ , i.e.,  $V = \{a, b\}$ , então:

$$\mathcal{L} = \{\epsilon, a, b, aa, bb, aaa, bbb, \dots\}$$

é uma linguagem sobre  $V$  constituída por todas as frases que ou são nulas ou são formadas por zero ou mais repetições de um único símbolo, o símbolo  $a$  ou o  $b$ .

□

Note-se que pela definição 2.1 os seguintes conjuntos também são linguagens:

- $\emptyset$  define a linguagem vazia;
- $\{\epsilon\}$  define a linguagem cuja única frase é a frase nula;
- $\{a\}$  define a linguagem cuja única frase é a frase  $a$ , com  $V = \{a\}$ .

Uma classe de linguagens bastante importante são as linguagens regulares.

**Definição 2.2** *Uma linguagem diz-se regular se puder ser obtida aplicando um número finito de operações (sobre linguagens) ao conjunto vazio e aos conjuntos contendo unicamente a frase nula ou um único símbolo do alfabeto.*

### Operações sobre Linguagens

Sendo  $\mathcal{L}_1$  e  $\mathcal{L}_2$  linguagens, então podem-se considerar as seguintes operações:

1. **Exponenciação**

$$\mathcal{L}^0 = \{\epsilon\} \text{ e } \mathcal{L}^i = \mathcal{L}\mathcal{L}^{i-1}$$

2. **Concatenação**

$$\mathcal{L}_1\mathcal{L}_2 = \{uv \mid u \in \mathcal{L}_1 \wedge v \in \mathcal{L}_2\}$$

3. **União**

$$\mathcal{L}_1 \cup \mathcal{L}_2 = \{u \mid u \in \mathcal{L}_1 \vee u \in \mathcal{L}_2\}$$

4. **Fecho Transitivo**  $\mathcal{L}^+ = \cup_{i=1}^{\infty} \mathcal{L}^i$ , i.e.,  $\mathcal{L}^+$  denota uma ou mais concatenações de  $\mathcal{L}$ ;

5. **Fecho de Kleene**:  $\mathcal{L}^* = \cup_{i=0}^{\infty} \mathcal{L}^i$ , i.e.,  $\mathcal{L}^*$  denota zero ou mais concatenações de  $\mathcal{L}$ ;

De seguida apresentam-se algumas linguagens definidas utilizando estes operadores.

- $\{a\}^* = \{a^n \mid n \geq 0\}$
- $\{a\} \cup \{b\} = \{a, b\}$
- $\{a\}\{b\} = \{ab\}$

- $\{aaa\}^* = \{a^n | n \text{ é divisível por } 3\}$
- $\{a, b\}^+$  é o conjunto de todas as frases sobre  $V = \{a, b\}$

Note-se que algumas das partes mais importantes de uma linguagem de programação são linguagens regulares, mais concretamente as palavras reservadas, os sinais de pontuação, os identificadores e os números.

Exemplo: Seja  $\mathcal{L}_1 = \{A, \dots, Z, a, \dots, z\}$  e  $\mathcal{L}_2 = \{0, \dots, 9\}$  duas linguagens<sup>1</sup>, então aplicando os operadores anteriores podemos obter novas linguagens, assim:

- $\mathcal{L}_1 \cup \mathcal{L}_2$  é uma linguagem constituída pelas frases constituídas por um símbolo, que é uma letra ou é um dígito;
- $\mathcal{L}_1 \mathcal{L}_2$  é uma linguagem constituída pelas frases formadas por uma letra, seguida por um dígito;
- $\mathcal{L}_1^4$  é uma linguagem constituída por todas as frases formadas por quatro letras;
- $\mathcal{L}_1^*$  é a linguagem constituída por todas as frases constituídas por letras, incluindo ainda a frase nula;

□

**Exercício 2.1** *Utilizando as linguagens  $\mathcal{L}_1$  e  $\mathcal{L}_2$  do exemplo anterior, defina as seguintes linguagens:*

1. *Uma linguagem constituída pelas frases com um ou mais dígitos;*
2. *Uma linguagem constituída pelas frases com letras e dígitos, mas que começam obrigatoriamente por uma letra.*

---

<sup>1</sup> $\mathcal{L}_1$  é uma linguagem constituída por um conjunto finito de frases de comprimento 1 e que representam as letras maiúsculas e minúsculas.  $\mathcal{L}_2$  é a linguagem constituída pelas frases que representam os dígitos.



## 2.2 Expressões Regulares

As expressões regulares proporcionam um modo sucinto de descrever uma linguagem regular. A sua importância reside no facto de permitirem descrever a sintaxe das linguagens de programação (mais precisamente os seus símbolos básicos) e de serem muito usadas no reconhecimento de padrões. Um outro aspecto que torna as expressões regulares bastante importantes é a sua utilização como linguagem de especificação de várias *ferramentas geradoras de analisadores léxicos*<sup>2</sup>.

As expressões regulares são analisadas de seguida.

A cada expressão regular  $e$ , sobre um alfabeto  $V$ , associa-se a linguagem  $\mathcal{L}_e$ , de acordo com as seguintes regras:

1.  $\emptyset$  é uma expressão regular que define a linguagem conjunto vazio;
2.  $\epsilon$  é uma expressão regular que representa a linguagem  $\{\epsilon\}$ , *i.e.*, a linguagem que consiste na frase nula;
3. Sendo  $a$  um símbolo do alfabeto  $V$  ( $a \in V$ ), então  $a$  é uma expressão regular que representa a linguagem  $\mathcal{L}_a$ , formada pela frase  $a$ ;  
Sendo  $p$  e  $q$  expressões regulares que representam as linguagens  $\mathcal{L}_p$  e  $\mathcal{L}_q$ , respectivamente, então:
  4.  $p + q$  é uma expressão regular representando a linguagem  $\mathcal{L}_p \cup \mathcal{L}_q$ ;
  5.  $pq$  é uma expressão regular que representa a linguagem  $\mathcal{L}_p \mathcal{L}_q$ ;
  6.  $p^*$  é uma expressão regular que representa a linguagem  $\mathcal{L}_p^*$ ;
  7.  $p^+$  é uma expressão regular que representa a linguagem  $\mathcal{L}_p^+$

De modo a minimizar o número de parêntesis durante a escrita das expressões regulares convencionou-se a seguinte hierarquia de precedências dos operadores: o fecho transitivo e o fecho de Kleene tem a maior precedência, a concatenação tem precedência intermédia e a união tem a menor precedência. Todos os operadores são associativos à esquerda.

---

<sup>2</sup>Uma *ferramenta geradora de analisadores léxicos* é um programa que a partir de uma descrição de um conjunto de símbolos básicos gera um programa, numa qualquer linguagem de programação, que identifica esses símbolos básicos num texto.

De acordo com estas convenções a expressão regular  $a + ac^*$  é interpretada como  $a + (a(c)^*)$ .

Os símbolos básicos de uma linguagem de programação definem linguagens regulares, tal como foi referido anteriormente. Sendo assim, podem ser descritos por expressões regulares. Considere-se, por exemplo, a representação dos números inteiros usual nas linguagens de programação. Geralmente, um número inteiro é constituído pelo sinal, positivo ou negativo (caso exista), seguido obrigatoriamente por um dígito, podendo, posteriormente, ter mais dígitos. O alfabeto da linguagem é  $V = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$  e esta pode ser descrita pela seguinte expressão regular

$$Inteiro = (+ + - + \epsilon)(0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)(0 + \dots + 9)^*$$

A linguagem constituída pelas palavras reservadas e pontuação de uma linguagem de programação também pode ser facilmente descrita por uma expressão regular. A seguir apresenta-se um exemplo.

$$er = 'if' + 'then' + 'else' + ',' + 'while' + 'repeat' + ';' + ')' + '('$$

**Exercício 2.2** *Diga que linguagens são representadas pelas seguintes expressões regulares sobre o alfabeto  $T = \{a, b\}$ :*

1.  $(ab)^+$ ;
2.  $a + a^*b$ ;
3.  $((\epsilon + a)b)^*$ .

**Exercício 2.3** *Escreva expressões regulares para especificar a sintaxe das seguintes linguagens:*

1. frases constituídas por um ou mais símbolos  $a$ ;
2. frases constituídas por zero ou mais símbolos  $a$  seguidos por um ou mais símbolos  $b$ ;
3. frases constituídas por um ou mais símbolos  $a$  seguidos de zero ou mais símbolos  $b$ ;

4. frases, não nulas, constituídas por símbolos  $a$  ou  $b$ .
5. frases que representam a numeração romana;
6. frases que representam os números reais, i.e., com parte inteira, parte decimal e expoente opcionais (mas tem de existir parte inteira ou parte decimal);
7. Uma frase que representa os comentários na linguagem C;

Para tornar mais clara e estruturada a escrita/leitura de uma expressão regular, é possível atribuir identificadores a expressões regulares e definir novas expressões regulares usando esses identificadores. Este processo de definir expressões regulares designa-se por *definição regular*.

Exemplo: Considere a expressão regular que define a linguagem constituída pelos números inteiros, apresentada anteriormente. Uma definição regular seria:

$$\begin{aligned} \text{senal} &\rightarrow + + - \\ \text{digito} &\rightarrow 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 \\ \text{numero} &\rightarrow (\text{senal} + \epsilon) \text{digito digito}^* \end{aligned}$$

□

**Exercício 2.4** *Reescreva a expressão regular construída no exercício 2.3.6 usando uma definição regular, definindo expressões regulares para o sinal, a mantissa e o expoente.*

A especificação de uma linguagem pode ser feita por expressões regulares diferentes. De facto, podem existir várias expressões regulares que descrevem a mesma linguagem. Considere-se a linguagem  $\mathcal{L} = \{a\}$ , então as expressões regulares  $e_1 = a$ ,  $e_2 = \epsilon a$  e  $e_3 = a \epsilon$  descrevem essa linguagem, i.e.,  $\mathcal{L}_{e_1} = \mathcal{L}_{e_2} = \mathcal{L}_{e_3} = \mathcal{L}$ . Estas expressões regulares dizem-se *equivalentes*.

Repare-se que a linguagem cujas frases são os números inteiros, que foi anteriormente descrita pela expressão regular *Inteiro*, pode também ser descrita por:

$$\text{Inteiro} = (+ + - + \epsilon)(0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)^+$$

**Definição 2.3** *Duas expressões regulares  $\alpha$  e  $\beta$  dizem-se equivalentes, e escreve-se  $\alpha \equiv \beta$ , se e só se  $\mathcal{L}_\alpha = \mathcal{L}_\beta$ , i.e., se  $\alpha$  e  $\beta$  definem a mesma linguagem.*

Esta noção de equivalência entre expressões regulares pode ser expressa por um conjunto de igualdades. Daqui resulta a seguinte álgebra de expressões regulares.

### 2.2.1 Álgebra de Expressões Regulares

Sejam  $\alpha$ ,  $\beta$  e  $\gamma$  expressões regulares, então pode-se afirmar que:

1.  $(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma)$
2.  $\alpha + \emptyset = \emptyset + \alpha = \alpha$
3.  $\alpha + \beta = \beta + \alpha$
4.  $\alpha + \alpha = \alpha$
5.  $(\alpha\beta)\gamma = \alpha(\beta\gamma)$
6.  $\alpha\epsilon = \epsilon\alpha = \alpha$
7.  $\alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma$
8.  $(\beta + \gamma)\alpha = \beta\alpha + \gamma\alpha$
9.  $\alpha^+ = \alpha\alpha^* = \alpha^*\alpha$
10.  $\alpha^* = \epsilon + \alpha^+$
11.  $(\alpha + \epsilon)^+ = (\alpha + \epsilon)^* = \alpha^*$

Este conjunto de expressões algébricas tem uma grande utilidade pois permitem manusear expressões regulares. Deste modo, é possível simplificar expressões regulares complexas, utilizando esta álgebra, em expressões regulares mais simples que definem exactamente as mesmas linguagens.

Esta simplificação de expressões regulares complexas é bastante importante pois permite uma percepção das linguagens, por elas definidas, mais fácil e rápida.

Considere a seguinte expressão regular  $a + a^+$  e que se pretende verificar se é possível simplificá-la. Utilizando a álgebra introduzida é fácil verificar que tal acontece. De seguida apresentam-se os vários passos e respectiva expressão algébrica utilizada na simplificação desta expressão regular.

$$\begin{array}{ll}
 a + a^+ & \text{(regra 9)} \\
 a + aa^* & \text{(regra 6)} \\
 a\epsilon + aa^* & \text{(regra 7)} \\
 a(\epsilon + a^*) & \text{(regra 10)} \\
 a(\epsilon + \epsilon + a^+) & \text{(regra 4)} \\
 a(\epsilon + a^+) & \text{(regra 10)} \\
 aa^* & \text{(regra 9)} \\
 a^+ &
 \end{array}$$

Deste modo a expressão regular  $a + a^+$  define a mesma linguagem que a expressão  $a^+$ , sendo assim as duas expressões são equivalentes (definição 2.3). Este facto é facilmente constatado informalmente, uma vez que a expressão regular  $a + a^+$  define uma linguagem que é a união de duas linguagens: A linguagem formada unicamente pela frase  $a$  e a linguagem cujas frases são sequências de um ou mais símbolos  $a$ . Uma vez que a frase  $a$  está contida nas sequências de um ou mais símbolos  $a$  (pois é a sequência constituída apenas por um símbolo  $a$ ), então a união das duas linguagens coincide com a linguagem cujas frases são sequências de um ou mais símbolos  $a$ . Esta linguagem é também definida pela expressão regular  $a^+$ , logo as duas expressões definem exactamente a mesma linguagem.

**Exercício 2.5** *Utilizando a álgebra de expressões regulares simplifique as seguintes expressões regulares:*

1.  $a + a^*$
2.  $a^*b + a^*bb^+$

### 2.2.2 Expressões Regulares em UNIX

As expressões regulares são bastante usadas no "mundo real", nomeadamente no sistema operativo UNIX, para descrever sequências de caracteres, também designados *padrões*. Um utilizador do sistema operativo UNIX deverá estar familiarizado com este tipo de notação, uma vez que vários comandos seus usam expressões regulares. Neste sistema operativo as expressões regulares são usadas nos seguintes programas:

- Programas que efectuem *concordância de padrões*<sup>3</sup>, como o **fgrep**, **grep**<sup>4</sup> ou **egrep**. Estes programas procuram uma frase (o **fgrep**) ou um padrão mais complexo (o **grep** e o **egrep**) num ficheiro. A frase ou padrão são especificados utilizando expressões regulares. As linhas do ficheiro que contenham uma sequência de caracteres que *concorde* com frase ou padrão são enviadas para a saída.
- Programas *editores de texto*, como o **vi** e vários outros editores de texto modernos. Estes programas permitem ao utilizador definir um padrão e "percorrer" o texto à procura de instâncias desse padrão.
- Programas que produzem *analísadores léxicos*, como o **lex**. O programa **lex** a partir de um conjunto de expressões regulares que definem os símbolos de uma linguagem, produz um programa que agrupa as sequências de caracteres de um texto num dos símbolos definidos.

O programa **lex** vai ser analisado em detalhe na secção 4.2.2.

O UNIX utiliza uma notação própria para definir expressões regulares. Um resumo desta notação apresenta-se na tabela seguinte.

Expressão Regular	Notação do UNIX
$r^*$	<b>r*</b>
$r^+$	<b>r+</b>
$r + \epsilon$	<b>r?</b>
$r + s$	<b>r s</b>
$(r)$	<b>(r)</b>
$c$	<b>c</b>

Nesta notação o fecho de Kleene e o fecho transitivo são definidos usando os caracteres usuais (o '\*' e '+', respectivamente), mas não em sobrescrito devido a limitações dos teclados. Uma outra característica desta notação é a impossibilidade de definir a frase nula. No entanto, **r?** denota zero ou uma ocorrência da expressão regular **r**. Sendo assim, na prática não é necessário definir a frase nula. A união de duas expressões regulares é denotada pelo símbolo '|'. Os parêntesis curvos têm a função usual de agrupar operadores. O caractere **c** define a linguagem composta unicamente por esse caracter.

<sup>3</sup>*Pattern-Matching* na terminologia inglesa.

<sup>4</sup>O nome **grep** é um acrónimo de *Global Regular Expression Print*.

Por exemplo, a expressão regular  $(a+bc)^+$  escreve-se em UNIX do seguinte modo `(a | bc)+`.

Frequentemente é necessário escrever expressões regulares que definem linguagens cujas frases contêm um único símbolo, em que este símbolo é um carácter. Na secção 2.2 definiu-se a expressão regular *digito* para denotar frases constituídas por um único dígito. Porém, estas expressões regulares tendem a ser bastante longas (repare-se que na expressão regular *Inteiro* definida na mesma secção abreviou-se a sua escrita devido a este facto).

O UNIX proporciona um modo mais sucinto para especificar estas linguagens. Em primeiro lugar é possível envolver os símbolos/caracteres da linguagem entre parêntesis rectos, para denotar a expressão regular que é a união desses símbolos/caracteres. Uma expressão UNIX deste tipo designa-se *classe de caracteres*. Em segundo lugar não é necessário especificar explicitamente todos os caracteres da classe. O UNIX permite definir uma classe de "caracteres consecutivos" especificando apenas o primeiro e o último carácter e separando-os pelo símbolo '-'. Por caracteres consecutivos entende-se caracteres cujos códigos ASCII sejam consecutivos.

Por exemplo, a expressão regular *digito* = 0+1+2+3+4+5+6+7+8+9 pode ser definida pelas seguintes classes de caracteres:

$$digito = [0123456789]$$
$$digito = [0-9]$$

A notação usada pelo UNIX inclui ainda símbolos especiais para denotar o início e fim de uma linha. O símbolo '^' denota o início de uma linha e o símbolo '\$' o fim da linha.

Existe ainda um outro símbolo bastante importante nesta notação. O símbolo '.' que designa "qualquer carácter excepto o carácter *newline*". Este símbolo é designado na terminologia inglesa por *wild card symbol*.

Exemplo: Considere as expressões regulares seguintes

1. `a.*a`
2. `.*a.*e.*i.*o.*u.*`
3. `^a.*a$`

A primeira expressão regular define a linguagem cujas frases têm dois ou mais caracteres, mas que começam e terminam pelo caracter **a**. A segunda expressão regular define frases que contêm todas as vogais ordenadas. Por último, a terceira expressão regular define frases que iniciam e terminam linhas de um texto pelo caracter **a**.

□

Na notação do UNIX vários caracteres têm um significado especial, designadamente os caracteres '\$', '^', '[', ']', '(', ')', e '.'. Deste modo, não podem ser, eles próprios, especificados numa expressão regular. No entanto, o UNIX usa o caracter '\' (caracter *backslash*) para ultrapassar essa limitação. Assim, se se preceder esses caracteres pelo *backslash*, então a combinação dos dois caracteres é interpretada como o valor literal do segundo símbolo e não como o seu significado especial. Por exemplo, \. define o caracter '.' numa expressão regular do UNIX. Repare-se que o caracter '\' também tem um significado especial e portanto deve ser precedido por ele próprio, quando se pretende o seu valor literal.

Exemplo: Considere que se pretende especificar a linguagem cujas frases são sequências de dígitos com um ponto decimal, *i.e.*, números reais. Um modo de escrever essa expressão regular será:

$$[0-9]+\backslash.[0-9]*\backslash.[0-9]+$$

Note-se que o *backslash* é posto antes do caracter '.' uma vez que se pretende o seu valor literal.

□

### 2.2.3 Limitações das Expressões Regulares

As expressões regulares são um mecanismo bastante simples para descrever a sintaxe de linguagens. Esta simplicidade faz com que dêem origem a soluções simples e eficientes aos problemas em que se aplicam. No entanto, algumas linguagens não podem ser descritas utilizando expressões regulares, nomeadamente, vários construtores usuais em linguagens de programação.

Para provar esta limitação das expressão regular na descrição de algumas linguagens, vai-se utilizar o seguinte teorema:



**Teorema 2.1** *Seja  $\mathcal{L}$  uma linguagem regular, então existe um número natural  $N$  (dependente de  $\mathcal{L}$ ) tal que toda a frase  $z \in \mathcal{L}$ , com  $|z| \geq N$ , pode ser factorizada em  $z = uvw$ , satisfazendo as seguintes condições:*

- $v \neq \epsilon$
- $|uv| \leq N$
- para todo  $i \geq 0$ ,  $uv^i w \in \mathcal{L}$

A demonstração deste teorema não vai ser aqui apresentada, uma vez que não se enquadra no âmbito deste texto. Porém, esta demonstração pode ser encontrada em [FB94] (secção 4.9).

Considere-se a seguinte linguagem:

Seja  $\mathcal{L}$  uma linguagem constituída unicamente pelas frases que são formadas por  $n$  ocorrências do símbolo  $a$  seguidas por  $n$  ocorrências do símbolo  $b$ . Exemplos de frases de  $\mathcal{L}$  são:  $ab$ ,  $aabb$ ,  $aaabbb$ , ... Formalmente  $\mathcal{L}$  é o conjunto  $\{a^n b^n \mid n \geq 0\}$ . Pretende-se verificar se  $\mathcal{L}$  pode ser definida por uma expressão, *i.e.*, se  $\mathcal{L}$  é regular.

Prova: Se  $\mathcal{L}$  é regular então existe um número natural  $N$ , que satisfaz as condições do teorema 2.1. Seja  $z = a^N b^N$ , então uma vez que  $z \in \mathcal{L}$  e  $|z| \geq N$ , de acordo com o teorema 2.1, tem-se  $z = uvw$ ,  $v \neq \epsilon$ ,  $|uv| \leq N$  e para todo  $i \geq 0$ ,  $uv^i w \in \mathcal{L}$ . Uma vez que  $|uv| \leq N$  e  $v \neq \epsilon$  então  $u$  e  $v$  têm de ser subfrases de  $a^N$ . Seja  $u = a^j$ , com  $j \geq 0$ ,  $v = a^k$ , com  $k > 0$  (porque  $v \neq \epsilon$ ), e  $w = a^l b^N$ , com  $l \geq 0$ , então  $uv^0 w = a^{N-k} b^N$  (que tem menos  $a$ 's que  $b$ 's), logo  $uv^0 w \notin \mathcal{L}$  (veja-se o esquema que se apresenta a seguir). Assim, considerando  $i = 0$  contradiz-se a condição que  $uv^i w \in \mathcal{L}$ , para todo o  $i \geq 0$ . Portanto  $\mathcal{L}$  não pode ser uma linguagem regular.

□

$$\begin{aligned}
 z = uvw &= \overbrace{\underbrace{a \cdots a}_{u} \underbrace{a \cdots a}_{v} \underbrace{a \cdots a}_{a^l} \underbrace{b \cdots b}_{b^N}}^{a^N} \\
 z = uv^0 w &= \overbrace{\underbrace{a \cdots a}_{u} \underbrace{a \cdots a}_{a^{N-k}}}_{a^{N-k}} \underbrace{b \cdots b}_{b^N}
 \end{aligned}$$

Uma situação semelhante a esta surge nas linguagens de programação com o balanceamento dos parentesis. Note-se que na linguagem anterior, fazendo  $a = ($  e  $b = )$  define-se uma linguagem cujas frases tem o mesmo número de parêntesis a abrir e a fechar. Como foi provado anteriormente esta linguagem não é regular.

Uma outra limitação das expressões regulares surge na definição de estruturas aninhadas, também bastante frequentes em linguagens de programação.

Sendo assim, é necessário estudar formalismos mais poderosos que possam definir todas estas estruturas. Na secção seguinte introduz-se o conceito de *gramática* onde estas limitações das expressões regulares já não existem.

## 2.3 Gramáticas

Um outro modo de descrever a sintaxe de linguagens consiste na utilização de *gramáticas*. Uma gramática pode descrever linguagens que não podem ser descritas por expressões regulares. Um outro aspecto importante das gramáticas é o facto de permitirem especificar de uma forma bastante legível e sucinta uma linguagem de programação.

Porém, o que torna as gramáticas particularmente importantes é o facto de existirem metodologias bastante desenvolvidas que, a partir da gramática e usando processos sistemáticos, permitem construir programas que verificam se uma frase pertence ou não à linguagem. Estas metodologias atingiram um tal grau de desenvolvimento que é possível *construir/gerar automaticamente* programas que verificam se um texto está de acordo com as regras sintáticas da linguagem, definidas por uma gramática. As gramáticas são mesmo usadas por várias *ferramentas geradoras de analisadores sintáticos*<sup>5</sup> como linguagem de especificação da sintaxe da linguagem.

Por último, as gramáticas permitem especificar estruturas recursivas/aninhadas que existem em muitas linguagens de programação. Considere-se, por exemplo, a instrução *if*, usual na maioria das linguagens de programação:

$$\text{if } E \text{ then } I_1 \text{ else } I_2,$$

---

<sup>5</sup>Uma *ferramenta geradora de analisadores sintáticos* é um programa que a partir de uma descrição das regras sintáticas de uma linguagem gera automaticamente um programa, numa qualquer linguagem de programação, que verifica se um texto segue as regras sintáticas descritas.

em que  $E$  é uma expressão e  $I_1$  e  $I_2$  são instruções.

Este tipo de instrução não pode ser especificada por uma expressão regular. No entanto, pode-o ser através de uma gramática, como se verá mais à frente.

### 2.3.1 Definição de Gramática

Uma gramática é constituída por um conjunto de *símbolos terminais*, um conjunto de símbolos *não terminais*, um *símbolo inicial* e um conjunto de *produções*. Os terminais são os símbolos básicos da linguagem e definem o seu alfabeto. Os símbolos não terminais são classes (ou variáveis) sintáticas que definem conjuntos de frases. O símbolo inicial é um não terminal e representa a linguagem definida pela gramática. Por último, as produções são regras de sintaxe, em que uns símbolos são definidos em termos de outros.

De seguida, apresenta-se uma definição formal de gramática.

**Definição 2.4** *Uma gramática  $G$  é um quadruplo  $G = (T, N, S, P)$ , em que:*

- $T$  é o conjunto finito não vazio de símbolos terminais;
- $N$  é o conjunto finito não vazio de símbolos não terminais;
- $S \in N$  é o símbolo inicial, também designado por axioma da gramática;
- $P \subseteq (N \cup T)^+ \times (N \cup T)^*$  é o conjunto finito não vazio de produções ou regras de derivação. Cada produção é um par  $(\alpha, \beta)$  em que  $\alpha$  designa-se o lado esquerdo da produção e  $\beta$  o lado direito da produção; sendo  $\alpha \in (N \cup T)^+$  e  $\beta \in (N \cup T)^*$ . Geralmente, uma produção representa-se por  $\alpha \rightarrow \beta$  e lê-se " $\alpha$  deriva (imediatamente) em  $\beta$ ".

Exemplo: Considerando a instrução `if ... then ... else ...` apresentada na secção 2.3, de seguida mostra-se como seria possível especificar esta instrução utilizando-se uma gramática. Supondo que *Inst* é uma variável sintática, que representa a classe das instruções (de que a instrução `if` faz parte); e que *Expr* é uma outra variável sintática, que representa a classe das expressões; então a instrução é especificada pelas seguintes produções de uma gramática:

$$\begin{aligned} Inst &\rightarrow \text{if } Expr \text{ then } Inst \text{ else } Inst \\ Expr &\rightarrow \dots \end{aligned}$$

Em que os símbolos *Inst* e *Expr* são símbolos não terminais e **if**, **then** e **else** são símbolos terminais.

□

### 2.3.2 Gramáticas Independentes do Contexto

Geralmente definem-se classes de gramáticas mais restritivas, do que a apresentada na definição 2.4, mas com maior interesse prático. Uma das classes de gramáticas mais usadas, nomeadamente na definição de linguagens de programação, são as *gramáticas independentes do contexto* — GIC —, também chamadas *gramáticas de contexto livre*.

**Definição 2.5** *Uma gramática diz-se independente do contexto se todas as produções da gramática têm a forma:*

$$A \rightarrow \alpha \text{ tal que } A \in N \text{ e } \alpha \in (N \cup T)^*$$

i.e., quando o lado esquerdo das produções têm apenas um símbolo não terminal.

Por independência de contexto entende-se que durante o processo de derivação de uma frase todo o símbolo não terminal pode ser substituído sem ter em conta o contexto em que ocorre. Caso se admitissem produções da forma  $\gamma A \delta \rightarrow \alpha$  (de acordo com a definição 2.4), com  $\gamma, \delta, \alpha \in (N \cup T)^*$  e  $A \in N$ , então a substituição de  $A$  por  $\alpha$  só poderia ser feita quando a sua ocorrência fosse "envolvida" por  $\gamma$  e  $\delta$ . Deste modo, a gramática seria *sensível ao contexto*.

De seguida apresenta-se um exemplo de uma gramática independente do contexto.

*Exemplo:* Seja  $G = (T, N, S, P)$  uma gramática, com  $T = \{a, b, c\}$ ,  $N = \{A, B, C\}$ ,  $S = A$  e  $P = \{A \rightarrow aA, A \rightarrow Bc, A \rightarrow bC, B \rightarrow b, C \rightarrow c\}$ ; então  $G$  é uma GIC uma vez que todas as suas produções contêm um único símbolo não terminal do lado esquerdo.

□

As gramáticas apresentadas anteriormente estão escritas usando uma notação ou formalismo geralmente utilizado para o efeito e que irá ser adoptado neste texto. Este formalismo designa-se por *Backus Naur Form* — BNF<sup>6</sup> — e é universalmente aceite para escrever gramáticas independentes do contexto.

Ao longo deste texto vai-se utilizar uma série de convenções para tornar mais compacta a escrita de gramáticas e mais legível a sua leitura. Assim,

1. Os símbolos terminais escrevem-se utilizando letras minúsculas;
2. Os símbolos não terminais escrevem-se utilizando letras maiúsculas e minúsculas;
3. Quando existem várias produções com o mesmo lado esquerdo, do tipo  $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \dots, \alpha \rightarrow \beta_n$ , então escreve-se  $\alpha \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$ ;
4. Se o lado direito de uma produção  $\alpha \rightarrow \beta$  não contém nenhum símbolo, *i.e.*,  $\beta$  é um conjunto vazio, então escreve-se  $\alpha \rightarrow \epsilon$ .

No apêndice A apresenta-se uma gramática independente do contexto que define a sintaxe da linguagem que será usada ao longo deste texto para escrever gramáticas.

Voltando ao exemplo anterior, o conjunto de produções da gramática pode ser escrito do seguinte modo:

$$P = \{A \rightarrow aA \mid Bc \mid bC, B \rightarrow b, C \rightarrow c\}$$

Um outro modo de escrever as produções de uma gramática consiste em utilizar *grafos de sintaxe*. Estes grafos apresentam-se de seguida.

### Grafos de Sintaxe

A sintaxe de uma linguagem pode ainda ser definida através de *grafos de sintaxe*. Estes grafos permitem representar as produções  $P \subseteq N \times (N \cup T)^*$  da gramática  $G = (T, N, S, P)$  graficamente. A representação gráfica

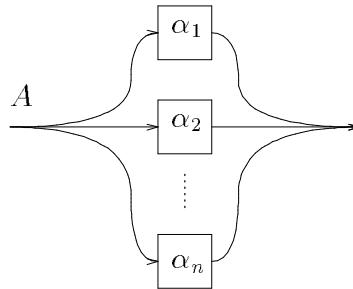
---

<sup>6</sup>Nome devido aos autores que definiram o formalismo.

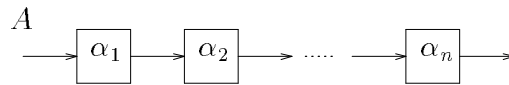
das produções permite uma melhor percepção da sintaxe da linguagem e do processo de *parsing* de uma frase.

Os grafos de sintaxe são construídos de acordo com cinco regras que permitem transformar BNF num grafo de sintaxe correspondente [Wir76]. As regras são:

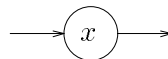
1. Produções da forma  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$  representam-se pelo grafo:



2. As produções da forma  $A \rightarrow \alpha_1 \alpha_2 \dots \alpha_n$  representam-se por:

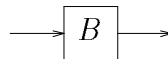


3. Os símbolos terminais  $x \in T$  em cada  $\alpha_i$  representam-se por:



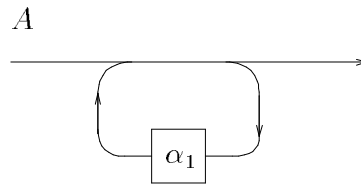
e corresponde a aceitar o símbolo  $x$  e avançar um símbolo na frase a reconhecer;

4. Os símbolos não terminais  $B \in N$  em cada  $\alpha_i$  representam-se por:



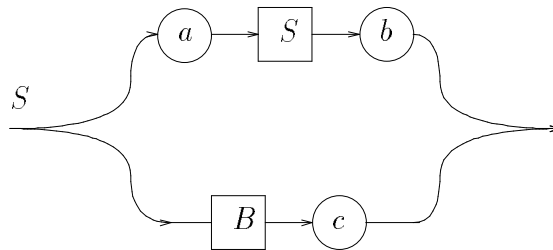
e corresponde a activar o subgrafo associado ao símbolo  $B$ ;

5. Produções da forma  $A \rightarrow \epsilon \mid \alpha_1 A$  representam-se pelo grafo:

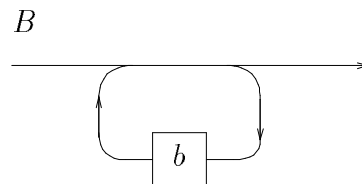


Exemplo: Considere-se a linguagem  $\mathcal{L}_1$  definida pela gramática  $G_1 = (T, N, S, P)$  com  $T = \{a, b, c\}$ ,  $N = \{S, B\}$ , e  $P = \{S \rightarrow aSb \mid Bc, B \rightarrow bB \mid \epsilon\}$ . Então utilizando as regras anteriores  $\mathcal{L}_1$  pode ser definida pelos grafos de sintaxe seguintes:

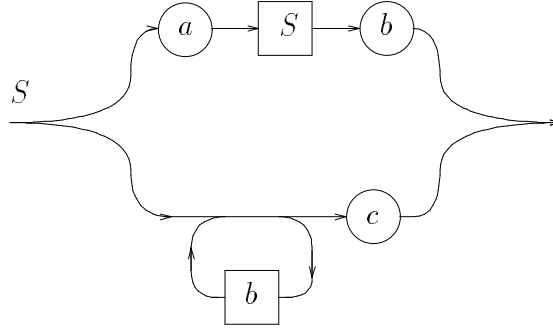
Para as produções  $S \rightarrow aSb \mid Bc \in P$  obtém-se:



e para as produções  $B \rightarrow bB \mid \epsilon \in P$  obtém-se



Compondo os dois grafos obtém-se o grafo de sintaxe  $g_1$  que define  $\mathcal{L}_1$ :



□

**Exercício 2.6** Construa o grafo de sintaxe  $g$  para a linguagem definida pela seguinte gramática  $G = (T, N, S, P)$  com  $T = \{x, y, z\}$ ,  $N = \{A, B, C\}$ ,  $S = A$  e  $P = \{A \rightarrow xB \mid \epsilon, B \rightarrow C \mid yB, C \rightarrow zC \mid \epsilon\}$ .

### 2.3.3 Derivação de Frases de uma Gramática

Após se ter escrito uma gramática, é necessário saber qual a linguagem gerada pela gramática, *i.e.*, quais são as suas frases válidas. Um modo de se determinar se uma dada frase pertence a uma linguagem, definida por uma gramática, consiste em determinar se essa frase pode ser *derivável* a partir do seu símbolo inicial. De seguida, define-se este novo conceito de *derivação*.

A derivação de uma frase de uma linguagem, definida por uma gramática, consiste em começar pelo seu símbolo inicial e consecutivamente substituir (reescrever) os símbolos não terminais pelo lado direito de uma produção, que tem esse não terminal do lado esquerdo. Este processo termina quando não existir mais nenhum símbolo não terminal na frase a reconhecer. A frase obtida no fim deste processo contém unicamente símbolos terminais (*i.e.*, símbolos do alfabeto) e é uma frase válida da linguagem.

Sendo  $G = (T, N, S, P)$  uma gramática, podem-se definir as seguintes relações:

1. **Derivação imediata:** Seja  $\alpha \in (N \cup T)^+$  e  $\beta \in (N \cup T)^*$ . Diz-se que  $\alpha$  deriva imediatamente em  $\beta$  e escreve-se  $\alpha \Rightarrow \beta$ , sse

$$\exists_{\alpha_1, \alpha_2 \in (N \cup T)^*, (a \rightarrow b) \in P} : \alpha = \alpha_1 a \alpha_2 \wedge \beta = \alpha_1 b \alpha_2$$



*i.e.*,  $\alpha A \beta \Rightarrow \alpha B \beta$  se  $A \rightarrow B$  é uma produção da gramática e  $\alpha$  e  $\beta$  são frases da linguagem. Neste caso diz-se que a produção  $A \rightarrow B$  foi *aplicada* ao símbolo não terminal  $A$  em  $\alpha A \beta$  e que  $A$  foi *reescrito*.

Exemplo: Seja  $G = (T, N, S, P)$  uma gramática, em que  $T = \{a, b, c\}$ ,  $N = \{S, A\}$ ,  $S = S$  e o conjunto de produções é  $P = \{S \rightarrow aAc, S \rightarrow \epsilon, A \rightarrow b\}$ , então teremos a seguinte derivação imediata:  $S \Rightarrow aAc$  em que  $\alpha_1 = \alpha_2 = \epsilon$  e a produção  $a \rightarrow b \in P$  é  $S \rightarrow aAc$ . Por sua vez  $aAc$  deriva em  $abc$  (*i.e.*,  $aAc \Rightarrow abc$ ) sendo  $\alpha_1 = a$ ,  $\alpha_2 = c$  e a produção usada é  $A \rightarrow b$ .

□

Como se pode constatar no exemplo anterior, qualquer produção da gramática  $\alpha \rightarrow \beta$  origina que se verifique  $\alpha$  deriva imediatamente em  $\beta$ , *i.e.*,  $\alpha \Rightarrow \beta$  (por exemplo, isto aconteceu na primeira derivação do exemplo).

2. **Deriva em um ou mais passos:** Dizemos que  $\alpha$  deriva em um ou mais passos em  $\beta$ , e escreve-se  $\alpha \Rightarrow^+ \beta$  sse

$$\alpha \Rightarrow \beta \vee \exists_{\delta \in (N \cup T)^*} : \alpha \Rightarrow \delta \wedge \delta \Rightarrow^+ \beta$$

*i.e.*,  $\alpha$  deriva em um ou mais passos em  $\beta$  se  $\alpha$  deriva imediatamente em  $\beta$  ou se  $\alpha$  deriva imediatamente em  $\delta$  que por sua vez deriva em uma ou mais vezes em  $\beta$ ;

3. **Deriva em zero ou mais passos:** Dizemos que  $\alpha$  deriva em zero ou mais passos em  $\beta$ , e escreve-se  $\alpha \Rightarrow^* \beta$  sse

$$\alpha = \beta \vee \alpha \Rightarrow^+ \beta$$

*i.e.*,  $\alpha$  deriva em zero ou mais passos em  $\beta$  se  $\alpha$  é igual a  $\beta$  ou se  $\alpha$  deriva em um ou mais passos em  $\beta$ .

Estamos agora em condições de definir a linguagem gerada por uma gramática.

**Definição 2.6** Sendo  $G = (T, N, S, P)$  uma gramática, então a linguagem  $\mathcal{L}_G$  gerada por  $G$  é um conjunto de frases sobre  $T$  dado por:

$$\mathcal{L}_G = \{\mu \in T^* \mid S \Rightarrow^* \mu\},$$

i.e., uma frase  $\mu$  pertence à linguagem se e só se fôr derivável em zero ou mais passos a partir de  $S$  e através da gramática  $G$ .

*Exemplo:* Seja  $G_1 = (T, N, S, P)$  uma gramática, em que  $T = \{a, b\}$ ,  $N = \{S, A, B\}$ ,  $S = S$  e o conjunto de produções é  $P = \{S \rightarrow bA, S \rightarrow aAB, A \rightarrow aA, A \rightarrow a, B \rightarrow b\}$ , então  $\mu = aaab$  é uma frase válida de  $\mathcal{L}_{G_1}$  (a linguagem gerada por  $G$ ), uma vez que  $\mu$  é derivável a partir do axioma de  $G_1$ , como se pode verificar na seguinte sequência de derivações imediatas:

$$S \Rightarrow aAB \Rightarrow aaAB \Rightarrow aaaB \Rightarrow aaab$$

□

Uma derivação que termina com uma frase sobre  $T$  diz-se uma *derivação completa* (este é o caso do exemplo anterior).

**Definição 2.7** *Uma linguagem diz-se independente do contexto se puder ser especificada por uma gramática independente do contexto.*

Existem duas estratégias distintas para efectuar a derivação de uma frase: *derivação pela esquerda* e *derivação pela direita*. Numa derivação pela esquerda (direita) é sempre o símbolo não terminal mais à esquerda (direita) que é substituído. De seguida apresentam-se as suas definições formais.

**Definição 2.8** *Uma derivação pela esquerda<sup>7</sup> é toda a derivação*

$$S = f_0 \Rightarrow f_1 \Rightarrow \dots \Rightarrow f_{n-1} \Rightarrow f_n = f$$

*em que  $f_i$  obtém-se de  $f_{i-1}$  substituindo o não terminal mais à esquerda em  $f_{i-1}$ .*

**Definição 2.9** *Uma derivação pela direita<sup>8</sup> é toda a derivação*

$$S = f_0 \Rightarrow f_1 \Rightarrow \dots \Rightarrow f_{n-1} \Rightarrow f_n = f$$

*em que  $f_i$  obtém-se de  $f_{i-1}$  substituindo o não terminal mais à direita em  $f_{i-1}$ .*

---

<sup>7</sup>*Leftmost Derivation* na terminologia inglesa.

<sup>8</sup>*Rightmost Derivation* na terminologia inglesa.

Na derivação apresentada anteriormente para a frase  $\mu = aaab$  foi efectuada uma derivação pela esquerda. Uma derivação pela direita seria:

$$S \Rightarrow aAB \Rightarrow aAb \Rightarrow aaAb \Rightarrow aaab$$

Após se ter introduzido o conceito de derivação de uma frase é possível agora apresentar três novos conceitos que serão necessários para o reconhecimento de linguagens: o de símbolo anulável e de gramáticas recursivas à esquerda ou à direita. As suas definições apresentam-se de seguida.

**Definição 2.10** *Um símbolo  $A \in N$  diz-se anulável se existe*

$$A \Rightarrow^* \epsilon$$

*i.e., a partir de  $A$  consegue derivar-se a frase nula.*

**Definição 2.11** *Uma gramática  $G = (T, N, S, P)$  diz-se recursiva à esquerda se existir um  $A \in N$  tal que existe uma derivação  $A \Rightarrow^* A\alpha$  para alguma frase  $\alpha$ .*

Uma gramática recursiva à direita define-se de modo semelhante. Assim, tem-se:

**Definição 2.12** *Uma gramática  $G = (T, N, S, P)$  diz-se recursiva à direita se existir um  $A \in N$  tal que existe uma derivação  $A \Rightarrow^* \alpha A$  para alguma frase  $\alpha$ .*

**Exercício 2.7** *Escreva uma gramática  $G_1$  para descrever a sintaxe da linguagem  $\mathcal{L}_1$  constituída por strings com um ou mais símbolos  $a$ .*

*Prove que  $aaa$  é uma string válida de  $\mathcal{L}_1$ .*

**Exercício 2.8** *Escreva gramáticas para especificar a sintaxe das seguintes linguagens:*

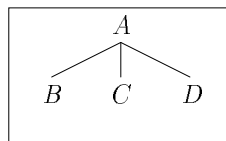
1. *frase constituídas por um ou mais símbolos  $a$  seguidos de zero ou mais símbolos  $b$ ;*
2. *frases constituídas por zero ou mais símbolos  $a$  seguidos de zero ou mais símbolos  $b$ , sendo o número de símbolos  $a$  igual ao número de símbolos  $b$ ;*

3. frases constituídas por zero ou mais símbolos  $a$  seguidos de zero ou mais símbolos  $c$  e de zero ou mais símbolos  $c$ , sendo o número de símbolos  $a$  mais o de  $b$  igual ao número de símbolos  $c$ ;
4. frases que representam os números inteiros;
5. frases que representam os números reais, i.e., com parte inteira, parte decimal e expoente opcionais (mas tem de existir parte inteira ou parte decimal);
6. frases que representam os identificadores de uma linguagem de programação (considere que um identificador tem de começar obrigatoriamente por uma letra);
7. frases que representam expressões aritméticas simples.

### Árvore de Derivação

Uma árvore de derivação pode ser vista como uma representação gráfica (em forma de árvore) da sequência de derivação. Deste modo, uma árvore de derivação mostra como a partir de um símbolo a gramática deriva uma frase da linguagem.

A representação em árvore do processo de derivação de uma frase da linguagem é feito do seguinte modo: para cada produção da gramática  $A \rightarrow BCD$  usada na derivação da frase a reconhecer, existirá um nodo (interior) na árvore, associado ao símbolo  $A$ , nodo este que terá três nodos descendentes associados aos símbolos  $B$ ,  $C$  e  $D$ , representados da esquerda para a direita, pelos quais o não terminal  $A$  foi substituído nessa derivação. Graficamente será:



As folhas da árvore de derivação, agrupadas da esquerda para a direita, constituem o que se designa por *fronteira* da árvore. De notar que os símbolos da gramática aos quais estão associados os nodos da fronteira de uma árvore de derivação coincidem com a frase representada por essa mesma árvore.

Formalmente, uma árvore de derivação define-se do seguinte modo:

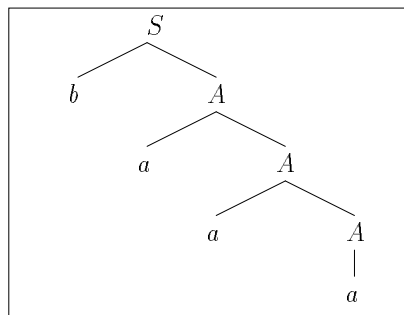
**Definição 2.13** *Seja  $G = (T, N, S, P)$  uma gramática independente do contexto. Se  $A \rightarrow \alpha_1 \cdots \alpha_n \in P$ , então existe uma árvore de derivação cuja raiz é  $A$  e cujas subárvores são  $S_1, \dots, S_n$  (nesta ordem) em que:*

- $S_i$  consiste num único nodo associado ao símbolo  $\alpha_i$ , se  $\alpha_i \in T$ ;
- $S_i$  é uma árvore de derivação cuja raiz é  $\alpha_i$ , se  $\alpha_i \in N$ .

De notar que as folhas de uma árvore de derivação estão sempre associadas a símbolos terminais enquanto os seus nodos interiores estão sempre associados a símbolos não terminais.

Uma árvore de derivação é construída do seguinte modo: para cada derivação imediata do tipo  $\alpha X \beta \Rightarrow \alpha X_1 X_2 \dots X_n \beta$ , de uma sequência de derivações, em que se usou a produção  $X \rightarrow X_1 X_2 \dots X_n$ , acrescenta-se ao nodo (da árvore de derivação) associado ao símbolo  $X$   $n$  nodos descendentes associados, da esquerda para a direita, aos símbolos  $X_1 X_2 \dots X_n$ . No caso particular da primeira derivação cria-se um nodo, que será a raiz da árvore de derivação, associado ao símbolo pelo qual se inicia a sequência de derivação.

Exemplo: Considere-se de novo o exemplo apresentado na página 26, em que a partir de uma gramática  $G$  se apresentou a sequência de derivações para a frase  $\mu = baaa$ . A árvore de derivação que se obtém ao derivar esta frase é a seguinte:



Como se pode ver nesta figura as folhas da árvore de derivação (*i.e.*, a fronteira da árvore) agrupadas da esquerda para a direita coincidem com a frase  $\mu$ .

□

**Exercício 2.9** *Construa uma árvore de derivação associada ao símbolo inicial da gramática do exercício 2.8.2 cuja fronteira é  $f = aabb$ .*

### Gramáticas Ambíguas

A ambiguidade é uma característica frequente nas linguagens naturais. Nas linguagens formais, embora em menor grau, também existe ambiguidade. A ambiguidade manifesta-se numa gramática independente do contexto quando é possível utilizarem-se diferentes produções da gramática para derivar uma mesma frase.

Considere-se o exemplo seguinte:

*Exemplo:* Seja  $G_1 = (T, N, S, P)$  uma gramática, em que  $T = \{a, b\}$ ,  $N = \{S, A, B\}$ , e o conjunto de produções é  $P = \{S \rightarrow bA, S \rightarrow BA, A \rightarrow aA, A \rightarrow a, B \rightarrow bB, B \rightarrow b\}$ , então a frase  $\mu = baa$  é uma frase válida da linguagem definida por  $G_1$ , uma vez que existe uma derivação

$$S \Rightarrow bA \Rightarrow baA \Rightarrow baa$$

que a partir do axioma deriva na frase  $\mu$  (i.e.,  $S \Rightarrow^+ \mu$ ). No entanto, a seguinte sequência de derivação

$$S \Rightarrow BA \Rightarrow bA \Rightarrow baA \Rightarrow baa$$

também deriva a frase  $\mu$ .

Sendo assim,  $G_1$  diz-se uma gramática ambígua.

□

Formalmente a ambiguidade define-se do seguinte modo:

**Definição 2.14** *Uma gramática independente do contexto  $G = (T, N, S, P)$  diz-se ambígua se existirem duas sequências de derivações diferentes que a partir do seu símbolo inicial  $S$  derivam a mesma frase  $\mu \in T^*$ .*

É óbvio que se podemos usar duas sequências de derivações diferentes para derivar a mesma frase, então também existirão árvores de derivação

diferentes para essa frase. Um caso frequente de ambigüidade em linguagens de programação surge na definição da sintaxe de expressões aritméticas.

Exemplo: Considere-se a seguinte gramática  $G = (T, N, S, P)$ , que define a sintaxe de expressões aritméticas simples, com:

$$\begin{aligned} T &= \{+, -, *, /, (, ), num, var\} & P &= \{Expr \rightarrow Expr \ Oper \ Expr \\ N &= \{Expr, Oper\} & &| ' ( Expr ) ' \\ S &= Expr & &| num \\ & & &| var \\ & & &Oper \rightarrow ' + ' | ' - ' | ' * ' | ' / ' \\ & & &\} \end{aligned}$$

em que os símbolos não terminais  $num$  e  $var$  representam qualquer número ou identificador de uma variável, respectivamente. Supondo que se pretendia verificar se a frase  $\mu = x + 3 * 5$  é uma frase válida da linguagem.

Nesta gramática existem duas derivações possíveis para derivar a frase  $\mu$ , nomeadamente:

$$\begin{aligned} \text{a)} Expr &\Rightarrow Expr \ Oper \ Expr \Rightarrow var \ ' + ' \ Expr \ Oper \ Expr \Rightarrow \\ &var \ ' + ' \ num \ ' * ' \ num \end{aligned}$$

$$\begin{aligned} \text{b)} Expr &\Rightarrow Expr \ Oper \ Expr \Rightarrow Expr \ Oper \ Expr \ ' * ' \ num \Rightarrow \\ &var \ ' + ' \ num \ ' * ' \ num \end{aligned}$$

Deste modo, a gramática  $G$  é uma gramática ambígua.

Estas duas derivações dão origem a duas árvores de derivação diferentes (construídas utilizando a estratégia definida na subsecção 2.3.3), sendo, no entanto, as suas fronteiras iguais. Estas duas árvores estão representadas na figura seguinte.





*A gramática  $G$  é ambígua? Em caso afirmativo indique uma frase da linguagem que prove esse facto.*

**Exercício 2.12** *Escreva uma gramática  $G = (T, N, S, P)$  não ambígua, equivalente à do exercício anterior.*

### 2.3.4 Gramáticas Regulares

Uma outra classe de gramáticas bastante importante são as chamadas *gramáticas regulares*. Uma gramática regular define-se do seguinte modo:

**Definição 2.15** *Uma gramática independente do contexto é uma gramática regular se e só se for regular à direita ou regular à esquerda.*

De seguida, apresenta-se a definição de *gramática regular à direita* e à esquerda.

**Definição 2.16** *Uma gramática independente do contexto  $G = (T, N, S, P)$  é uma gramática regular à direita quando todas as produções têm a forma:*

$$A \rightarrow \mu \vee A \rightarrow \mu B,$$

*com  $A, B \in N$  e  $\mu \in T^*$ , i.e., todas as produções têm apenas um símbolo não terminal do lado direito, que é o símbolo mais à direita da produção.*

**Definição 2.17** *Uma gramática independente do contexto  $G = (T, N, S, P)$  é uma gramática regular à esquerda quando todas as produções têm a forma:*

$$A \rightarrow \mu \vee A \rightarrow B\mu,$$

*com  $A, B \in N$  e  $\mu \in T^*$ , i.e., todas as produções têm apenas um símbolo não terminal do lado direito, que é o símbolo mais à esquerda da produção.*

De notar que as gramáticas regulares além das restrições no lado esquerdo das produções (devido ao facto de serem gramáticas independentes do contexto) têm também restrições nos lados direitos. Estas restrições diminuem o poder expressivo da gramática, reduzindo-se deste modo o número de casos a que se aplicam. Assim, usam-se unicamente na definição de *linguagens regulares*. Surge agora o problema de saber o que é uma linguagem regular. A sua definição apresenta-se de seguida.

**Definição 2.18** *Uma linguagem diz-se uma linguagem regular se fôr possível ser definida por uma expressão ou gramática regular.*

Obviamente que uma linguagem que não é regular diz-se *não regular* e define-se da seguinte forma:

**Definição 2.19** *Uma linguagem diz-se uma linguagem não regular se não fôr possível ser definida por uma expressão ou gramática regular.*

Considere que se pretende escrever uma gramática regular à direita e uma outra regular à esquerda para descrever a sintaxe da linguagem definida pela expressão regular  $e = ab^+c^*d^+$ . A primeira gramática (não regular...) que ocorre escrever é constituída pelo seguinte conjunto de produções:

$$P = \left\{ \begin{array}{l} S \rightarrow a B C D \\ B \rightarrow bB \mid b \\ C \rightarrow cC \mid \epsilon \\ D \rightarrow dD \mid d \end{array} \right.$$

Como facilmente se verifica esta gramática não é regular (na produção  $S \rightarrow a B C D$  existe mais do que um símbolo não terminal do lado direito). Para se obterem as gramáticas regulares à esquerda e direita tem de se alterar estas produções de modo a cada uma delas obdecer à respectiva definição. As respectivas gramáticas regulares (definidas pelas suas produções) que se obtém apresentam-se de seguida:

Gramática regular à direita:

$$P = \left\{ \begin{array}{l} S \rightarrow abB \\ B \rightarrow bB \mid C \\ C \rightarrow cC \mid D \\ D \rightarrow dD \mid d \end{array} \right.$$

Gramática regular à esquerda:

$$P = \left\{ \begin{array}{l} S \rightarrow Cd \mid Sd \\ C \rightarrow B \mid Cc \\ B \rightarrow Bb \mid ab \end{array} \right.$$

**Exercício 2.13** *Escreva gramáticas regulares à esquerda e à direita que descrevam a linguagem constituída por frases com um ou mais símbolos a seguidos de zero ou mais símbolos  $b$ .*

**Exercício 2.14** *Escreva gramáticas regulares à esquerda e à direita que descrevam a linguagem constituída por frases que representam os números inteiros.*

### 2.3.5 Conversão de Gramáticas em Expressões Regulares

Uma das vantagens das expressões regulares quando comparadas com as gramáticas, nomeadamente as GIC, é a possibilidade de se poder construir mais facilmente reconhecedores das frases da linguagem que elas definem. Sendo assim, é vantajoso converter as gramáticas em expressões regulares para se implementar o respectivo reconhecedor. Porém, nem todas as GIC podem ser convertidas em expressões regulares, uma vez que as gramáticas são mais poderosas na descrição da sintaxe de linguagens e portanto pode surgir a situação de uma linguagem definida por uma gramática não poder ser definida por uma expressão regular (por exemplo, como foi dito anteriormente, uma linguagem com estruturas recursivas pode ser definida por uma gramática e não por uma expressão regular). No entanto, as gramáticas regulares podem sempre ser convertidas em expressões regulares. De seguida apresenta-se um método para converter gramáticas regulares em expressões regulares.

O método para converter gramáticas em expressões regulares divide-se em duas etapas:

1. Para cada símbolo não terminal  $X$  cujas produções em que aparece do lado esquerdo são da forma

$$X \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

escreve-se uma equação da forma

$$A = \Phi_1 + \Phi_2 + \dots + \Phi_n$$

onde  $\Phi_i$  é a expressão regular correspondente a  $\alpha_i$ , com  $1 \leq i \leq n$ . Obtendo-se, deste modo, um sistema de equações regulares;

2. Resolve-se este sistema em ordem a  $S$  (símbolo inicial da gramática) obtendo-se, no caso do sistema ter solução, a expressão regular pretendida. Se o sistema não tiver solução é porque a gramática não pode ser convertida numa expressão regular.

Na resolução destes sistemas de equações regulares são úteis os seguintes resultados, que completam a álgebra de expressões regulares, apresentada na secção 2.2.1:

$$12. \text{ Se } X = \beta + \alpha X \text{ então } X = \alpha^* \beta$$

$$13. \text{ Se } X = \beta + X\alpha \text{ então } X = \beta\alpha^*$$

*Exemplo:* Considere-se a gramática regular à esquerda do exercício 2.13. Seja  $G = (T, N, S, P)$  essa gramática, em que  $T = \{a, b\}$ ,  $N = \{S, Y\}$ ,  $S = S$  e  $P = \{S \rightarrow Y|Sb, Y \rightarrow a|Ya\}$ , então a conversão desta gramática regular numa expressão regular é feita do seguinte modo:

$$\begin{aligned} \begin{matrix} S \rightarrow Y|Sb \\ Y \rightarrow a|Ya \end{matrix} & \Rightarrow^9 \begin{cases} S = Y + Sb \\ Y = a + Ya \end{cases} \Rightarrow^{10} \begin{cases} S = Y + Sb \\ Y = aa^* \end{cases} \Rightarrow^{11} \\ & \Rightarrow \begin{cases} S = Y + Sb \\ Y = a^+ \end{cases} \Rightarrow^{12} S = a^+ + Sb \Rightarrow^{13} a^+b^* \end{aligned}$$

Deste modo, a linguagem definida pela gramática é exactamente a mesma que é definida pela expressão regular  $a^+b^*$  (ver exercício 2.3.3).

□

**Exercício 2.15** Considere a seguinte gramática  $G = (T, N, S, P)$  com:

$$\begin{array}{ll} T = \{a, b, c, d\} & P = \{ \begin{array}{l} S \rightarrow aA \mid B \\ A \rightarrow abS \mid bB \\ B \rightarrow b \mid cC \\ C \rightarrow bB \mid d \end{array} \\ N = \{S, A, B, C\} & \end{array}$$

*Determine uma expressão regular equivalente.*

---

<sup>9</sup>Conversão da gramática no sistema de expressões regulares (passo 1).

<sup>10</sup>Aplicação da regra 13.

<sup>11</sup>Aplicação da regra 9.

<sup>12</sup>Substituição.

<sup>13</sup>Aplicação da regra 13.

**Exercício 2.16** *Considere as produções de uma gramática  $G = (T, N, S, P)$  com:*

$$P = \{ \begin{array}{l} S \rightarrow Sinal \ RealSemSinal \\ Sinal \rightarrow + \mid - \mid \epsilon \\ RealSemSinal \rightarrow Inteiro \ ParteDec \\ Inteiro \rightarrow digito \mid digito \ Inteiro \\ ParteDec \rightarrow ' \cdot ' \ Inteiro \mid \epsilon \end{array} \}$$

*Determine uma expressão regular equivalente.*

### 2.3.6 Limitações das Gramáticas

As gramáticas independentes de contexto são um mecanismo poderoso para descrever a sintaxe das linguagens de programação. Porém, nem todos os requisitos usuais numa linguagem de programação podem ser especificadas usando unicamente GIC's. Nesta secção vão ser analisados alguns exemplos que demonstram este facto.

Na secção 2.2.3 apresentou-se um teorema que permitiu demonstrar que certas linguagens não são regulares. Em particular provou-se que a linguagem  $\mathcal{L}_1 = \{a^n b^n \mid n \geq 0\}$  não é regular. Porém, esta linguagem<sup>14</sup> é independente do contexto uma vez que pode ser definida pela GIC  $G = (\{a, b\}, \{S\}, S, P)$ , com

$$P = \{S \rightarrow aSb \mid \epsilon\}$$

Do mesmo modo, é desejável a existência de um teorema semelhante para provar que certas linguagens não podem ser descritas por uma gramática independente do contexto. Este teorema apresenta-se a seguir.

**Teorema 2.2** *Seja  $\mathcal{L}$  uma linguagem independente do contexto, então existe um número natural  $N$  (dependente de  $\mathcal{L}$ ) tal que toda a frase  $z \in \mathcal{L}$ , com  $|z| \geq N$ , pode ser factorizada em  $z = uvwxy$ , satisfazendo as seguintes condições:*

$$\bullet \ v \neq \epsilon \vee x \neq \epsilon$$

---

<sup>14</sup>Repare-se que esta linguagem é a mesma do exercício 2.8.2.

- $|vwx| \leq N$
- para todo  $i \geq 0$ ,  $uv^iwx^iy \in \mathcal{L}$

A demonstração deste teorema não vai ser aqui apresentada, pois (tal como no caso do teorema 2.1) ela não se enquadra no âmbito deste texto. A sua demonstração pode ser encontrada em [FB94] (secção 5.8).

Considere-se a seguinte linguagem:

$$\mathcal{L}_2 = \{a^ib^jc^j \mid i \geq 0\}$$

Pretende-se verificar se  $\mathcal{L}_2$  é independente de contexto.

*Prova:* Supondo que  $\mathcal{L}_2$  é uma linguagem independente de contexto, então, de acordo com o teorema 2.2, existe um número  $N$  que satisfaz as suas condições. Seja  $z = a^Nb^Nc^N$ , então, pelo teorema 2.2,  $z$  pode ser factorizado em  $z = uvwxy$ , tal que  $v$  e  $x$  não são simultaneamente vazios,  $|vwx| \leq N$  e  $uv^iwx^iy \in \mathcal{L}_2$  para todo o  $i \geq 0$ . Se  $v$  contém um número positivo de  $a$ 's e um número positivo de  $b$ 's, então  $uv^2wx^2y$  contém um  $b$  seguido de um  $a$ , logo  $uv^2wx^2y \notin \mathcal{L}_2$  (ver esquema seguinte). Assim, a hipótese de que se partiu é contrariada, logo  $\mathcal{L}_2$  não é independente do contexto.

□

$$\begin{aligned} z = uvwxy &= \underbrace{a \cdots a}_u \underbrace{ab}_v \underbrace{b \cdots b}_w \underbrace{b \cdots b}_x \underbrace{bc \cdots c}_y \in \mathcal{L}_2 \\ z = uv^2wx^2y &= \underbrace{a \cdots a}_u \underbrace{ab}_v \underbrace{ab}_v \underbrace{b \cdots b}_w \underbrace{b \cdots b}_x \underbrace{b \cdots b}_x \underbrace{bc \cdots c}_y \notin \mathcal{L}_2 \end{aligned}$$

Esta estrutura não é muito frequente numa linguagem de programação. No entanto, existem várias outras estruturas de uma linguagem que não podem ser descritas por uma GIC. Por exemplo, a linguagem

$$\mathcal{L}_3 = \{a^n b^m c^n d^m \mid n \geq 0 \wedge m \geq 0\},$$

que descreve abstratamente o problema de verificar se o número de parâmetros formais de uma função é o mesmo que o número de parâmetros actuais na sua invocação, também não é independente do contexto. Usando o teorema 2.2, facilmente se prova que  $\mathcal{L}_3$  não é independente de contexto. Esta demonstração é deixada como exercício.

Informalmente, diz-se que uma expressão regular "não sabe contar", uma vez que não consegue descrever linguagens como  $\mathcal{L}_1$ , pois precisaria de contar o número de  $a$ 's para ver se é igual ao número de  $b$ 's. De modo semelhante, diz-se que uma gramática independente do contexto "consegue contar dois *items*, mas não três", pois consegue descrever  $\mathcal{L}_1$ , mas não  $\mathcal{L}_2$  e  $\mathcal{L}_3$ .

A solução para especificar completamente estas linguagens consiste na utilização de atributos e condições contextuais. A extensão das GIC com atributos e condições contextuais designam-se por *Gramáticas de Atributos*<sup>15</sup> [Hen92].

---

<sup>15</sup>Embora no livro *dragão* [ASU86], sem se perceber bem porquê, sejam designadas por *Definições Dirigidas pela Sintaxe* (em inglês *Syntax Directed-Definitions*)!





## Capítulo 3

# Reconhecimento de Linguagens

As expressões regulares e as gramáticas são mecanismos bastante apropriados para descrever a sintaxe das linguagens de programação, tal como foi referido no capítulo 2. No entanto, após se definir a sintaxe de uma linguagem, através de um desses mecanismos, surge a necessidade de verificar se uma dada frase pertence a essa mesma linguagem, *i.e.*, se obedece às suas regras sintáticas. Esta tarefa designa-se por *reconhecimento da linguagem*, ou na terminologia inglesa *parsing*.

Neste capítulo serão apresentadas várias técnicas que permitem construir programas que efectuem o reconhecimento de uma linguagem, definida por um dos mecanismos estudados. Estes programas designam-se por *reconhecedores*, ou *parsers*, da linguagem.

Na secção seguinte apresenta-se mais detalhadamente esta noção de reconhecedor de uma linguagem. Na secção 3.2 analisa-se o problema do reconhecimento de linguagens regulares, introduzindo-se o conceito de *autómato finito* (secção 3.2.1) e indicando-se como um autómato finito pode ser usado no reconhecimento de uma linguagem regular. Posteriormente analisam-se técnicas para converter gramáticas regulares e expressões regulares em autómatos finitos (secções 3.2.2 e 3.2.3, respectivamente). A secção 3.2 termina com a análise de um problema concreto (secção 3.2.7).

Nas secções 3.3 e 3.4 analisa-se o reconhecimento das linguagens não regulares. Nessas secções analisam-se duas estratégias distintas: reconhecimento *top-down* (secção 3.3) e o reconhecimento *bottom-up* (secção 3.4).

### 3.1 Reconhecedor

Genericamente, um reconhecedor de uma linguagem pode ser visto como um programa que para cada frase  $\alpha$  que lhe é submetida responde "ok" se  $\alpha$  é uma frase válida da linguagem e "erro" caso contrário.

Formalmente, um reconhecedor pode ser definido do seguinte modo:

**Definição 3.1** *Seja  $\mathcal{L}$  uma linguagem, tal que  $\mathcal{L} \subseteq \mathcal{V}^*$ , supondo  $\mathcal{R}_{\mathcal{L}}$  o reconhecedor de  $\mathcal{L}$  e  $l \in \mathcal{V}^*$  a frase a reconhecer, então*

$$\mathcal{R}_{\mathcal{L}}(l) = \begin{cases} aceita & \Leftarrow l \in \mathcal{L} \\ erro & \Leftarrow l \notin \mathcal{L} \end{cases}$$

Assim, é necessário estudar técnicas que, a partir dos mecanismos que especificam a sintaxe da linguagem, permitam construir a função  $\mathcal{R}_{\mathcal{L}}$ .

Porém, um reconhecedor de uma linguagem não tem a única função de indicar se uma frase pertence ou não à linguagem. O reconhecedor tem ainda a importante função de *estruturar uma representação linear* da frase, de acordo com as regras sintáticas da linguagem. A representação linear da frase a reconhecer é geralmente um texto constituído por uma sequência de caracteres. A estruturação consiste em agrupar os caracteres da frase em símbolos do alfabeto da linguagem e na construção de uma árvore, designada *árvore sintática*, que representa a estrutura sintática dessa frase. Nesta árvore cada nodo representa um construtor da linguagem e os seus filhos representam os componentes desse construtor. As folhas, tal como nas árvores de derivação (*c.f.* secção 2.3.3), representam os símbolos básicos da linguagem.

Note-se que uma árvore de derivação não é o mesmo que uma árvore sintática. A primeira representa a sequência de derivação usada para reconhecer uma frase, enquanto a segunda representa a estrutura sintática da frase. No entanto, podem surgir situações em que estas árvores são iguais.

Na secção seguinte analisa-se o reconhecimento de linguagens regulares, que é o caso mais simples. Posteriormente analisa-se o reconhecimento das linguagens não regulares.

## 3.2 Reconhecimento de Linguagens Regulares

A construção de um reconhecedor de uma linguagem regular é geralmente feito utilizando um mecanismo bastante conhecido: os *autómatos finitos*. Os autómatos finitos são utilizados no reconhecimento de linguagens regulares devido a várias características, nomeadamente:

- todo o autómato finito define uma linguagem regular;
- toda a linguagem regular pode ser definida por um autómato finito;
- existem métodos para converter gramáticas e expressões regulares em autómatos finitos; e por último
- existem técnicas bem definidas para converter autómatos finitos (determinísticos) em programas de computador.

### 3.2.1 Autómatos Finitos

Formalmente um *autómato finito* —AF— define-se do seguinte modo:

**Definição 3.2** Um *autómato finito*  $\mathcal{A}$  é um *quintúplo*  $\mathcal{A} = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$ , em que:

- $\mathcal{T}$  é o vocabulário, um conjunto finito de símbolos;
- $\mathcal{Q}$  é um conjunto finito não vazio de estados;
- $\mathcal{S} \subseteq \mathcal{Q}$  é um conjunto não vazio de estados iniciais;
- $\mathcal{Z} \subseteq \mathcal{Q}$  é um conjunto não vazio de estados finais;
- $\delta \subseteq (\mathcal{Q} \times (\mathcal{T} \cup \{\epsilon\})) \times \mathcal{Q}$  é um conjunto de transições de estados.

Uma transição  $((q, s), q') \in \delta$ , com  $q, q' \in \mathcal{Q}$  e  $s \in (\mathcal{T} \cup \{\epsilon\})$ , deve ler-se "o autómato  $\mathcal{A}$  transita do estado  $q$  para o estado  $q'$  através do reconhecimento do símbolo  $s$ ". As transições pelo símbolo  $\epsilon$ , *i.e.* do tipo  $((q, \epsilon), q')$ , designam-se *transições- $\epsilon$* .

De seguida apresenta-se um exemplo de um AF  $\mathcal{A}_1 = (\mathcal{T}_1, \mathcal{Q}_1, \mathcal{S}_1, \mathcal{Z}_1, \delta_1)$ , cujo vocabulário é formado pelos símbolos  $+, -, ., 1$ , e é constituído por 5 estados  $A, B, C, D, E$  cujo único estado inicial é o estado  $A$  e o final é o  $E$ . Este autómato define-se do seguinte modo:

$$\begin{aligned}\mathcal{T}_1 &= \{+, -, ., 1\} \\ \mathcal{Q}_1 &= \{A, B, C, D, E\} \\ \mathcal{S}_1 &= \{A\} \\ \mathcal{Z}_1 &= \{B\} \\ \delta_1 &= \{((A, +), B), ((A, -), B), ((A, \epsilon), B), ((B, 1), C), ((C, \epsilon), B), \\ &\quad ((C, .), D), ((C, \epsilon), E), ((D, 1), E), ((E, \epsilon), D)\}\end{aligned}$$

Geralmente o conjunto de transições de um autómato finito representa-se numa tabela, designada por *tabela de transições de estados*. Nesta tabela a cada linha associa-se um estado do autómato. As colunas associam-se aos símbolos que constituem o seu vocabulário e ainda ao símbolo  $\epsilon$ . As entradas da tabela são construídas do seguinte modo: para cada transição  $((q, s), q') \in \delta$  cria-se uma entrada na linha  $q$  (associada ao estado  $q$ ) e coluna  $s$  (associada ao símbolo  $s$ ) com o valor  $q'$ .

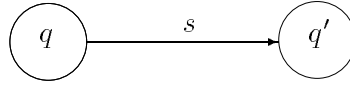
Representando o conjunto de transições do autómato anterior numa tabela de transição de estados obtém-se:

$$\delta =$$

$\mathcal{Q}_1 \setminus \mathcal{T}_1 \cup \{\epsilon\}$	$+$	$-$	$.$	$1$	$\epsilon$
A	B	B			B
B				C	
C			D		{ B,E }
D				E	
E					D

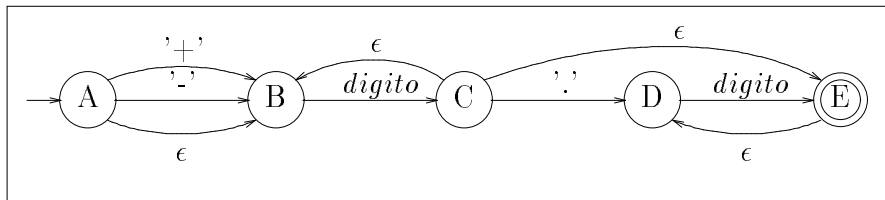
Esta tabela permite uma melhor visualização das transições entre os diferentes estados dos autómatos. No entanto continua a ser pouco "*claro*" qual a linguagem, ou melhor quais as frases da linguagem, que um dado autómato reconhece. Para resolver este problema é usual representar um autómato graficamente, uma vez que a tabela de transição de estados pode ser vista como um *grafo pesado*. Neste grafo os nodos correspondem aos estados do autómato e os ramos às transições de estados.

Deste modo, a cada transição  $((q, s), q') \in \delta$  associa-se um ramo do grafo com origem no nodo  $q$ , destino no nodo  $q'$  e peso  $s$ . Assim, obtém-se o seguinte subgrafo:



O grafo final, que representa o autómato, obtém-se compondo os vários subgrafos associados às transições. Neste grafo é usual diferenciarem-se os nodos que correspondem aos estados iniciais e finais do autómato. Geralmente os nodos do grafo associados aos estados iniciais são marcados com uma *seta* e os estados finais são representados por um *circulo duplo*.

Voltando ao autómato  $\mathcal{A}_1$  que tem vindo a ser considerado, a sua representação gráfica será:

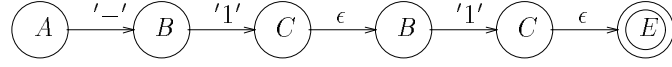


Após se ter definido o que é um autómato e as diferentes formas de o representar é importante agora analisar como é que um autómato pode efectuar o reconhecimento de uma linguagem regular.

Para um autómato finito ser um reconhecedor de uma linguagem tem de considerar *aceites* as frases válidas da linguagem e não aceites todas as outras, tal como foi definido na secção 3.1. O reconhecimento de uma frase por um autómato pode ser visto como o acto de caminhar ao longo do grafo (que o representa graficamente) a partir de um estado inicial até terminar num estado final. Assim, um autómato reconhece uma frase do seguinte modo:

**Definição 3.3** Sendo  $\mathcal{A} = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$  um autómato finito e  $\gamma \in \mathcal{T}^*$ , diz-se que  $\mathcal{A}$  aceita  $\gamma$  se e só se existir um caminho de um estado inicial para um estado final tal que a frase que se obtém concatenando os pesos dos ramos do grafo é igual a  $\gamma$ .

Considerando de novo o autómato  $\mathcal{A}_1$  e a frase  $\mu = -11$ , pretende-se verificar se  $\mathcal{A}_1$  aceita  $\mu$ . Facilmente se constata através da representação gráfica de  $\mathcal{A}_1$  que a frase é aceite, pois existe um caminho (embora não seja o único...) do estado inicial  $A$  para o estado final  $E$  cuja concatenação dos ramos corresponde à frase  $\mu$ . Um dos caminhos possíveis é:



Note-se que  $\epsilon$  é o elemento neutro da concatenação (*c.f.* secção 2.1) e portanto não aparece nas concatenações dos símbolos do vocabulário.

É óbvio que um reconhecedor baseado num AF não aceita uma frase "olhando" para a sua representação gráfica para determinar se existe tal caminho. Assim, é necessário *guiar* o reconhecedor ao longo do AF. Isto é feito usando o estado actual (que no início do processo é um dos estados iniciais) e o símbolo (da frase) a processar para *consultar* o conjunto de transições de estados e determinar para qual estado se transita. Sempre que a transição se efectuar pelo símbolo que se está a processar, então esse símbolo já foi reconhecido e passa-se para o símbolo seguinte. No caso de não existir nenhuma transição pelo símbolo a processar, então transita-se por  $\epsilon$  (caso existam transições- $\epsilon$  nesse estado) para um estado que tenha posteriormente uma transição pelo símbolo pretendido. Se esta situação também não se verificar então a frase não é reconhecida pelo AF, tal como veremos mais à frente. O processo de reconhecimento termina quando já não existir nenhum símbolo da frase por reconhecer e se atingir o estado final. Na tabela seguinte apresentam-se os passos do reconhecedor necessários para processar a frase  $\mu$  pelo autómato  $\mathcal{A}_1$ .

estado	$\mu$	transição	
$q_0 = A$	-11	$((A, ' -'), B)$	(1)
$q_1 = B$	11	$((B, ' 1'), C)$	(2)
$q_2 = C$	1	$((C, \epsilon), B)$	(3)
$q_3 = B$	1	$((B, ' 1'), C)$	(4)
$q_4 = C$	$\epsilon$	$((C, \epsilon), E)$	(5)
$q_5 = E$	$\epsilon$		<i>aceite</i>

No entanto, existem frases que não são reconhecidas pelo autómato  $\mathcal{A}_1$ . Considere-se por exemplo a frase  $\alpha = .11$ . Esta frase não é aceite por  $\mathcal{A}_1$ , pois embora a partir do estado inicial se possa transitar por  $\epsilon$  para o estado  $B$ , posteriormente só é possível transitar de estado pelo símbolo 1. Deste modo, não é possível *sair* deste estado e portanto atinge-se uma situação de *erro*.

Um reconhecedor, baseado num autómato finito  $\mathcal{A} = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$ , não aceita uma frase  $\mu = a_0 a_1 \dots a_n$  sempre que num estado  $q \in \mathcal{Q}$  se está a tentar reconhecer um símbolo  $a_i$  e  $((q, a_i), q'), ((q, \epsilon), q'') \notin \delta$ , com  $q', q'' \in \mathcal{Q}$  e  $0 \leq i \leq n$ .

O conjunto de todas frases aceites por um reconhecedor baseado num autómato finito constitui a linguagem que ele define. De seguida apresenta-se uma definição formal de um linguagem definida por um autómato finito.

**Definição 3.4** *A linguagem  $\mathcal{L}_{\mathcal{A}}$  definida por um autómato finito  $\mathcal{A}$  é o conjunto de todas as frases  $a_1 \dots a_n \in \mathcal{T}^*$ , tal que existe  $q_0 \dots q_n \in \mathcal{Q}$ , com  $q_0 \in \mathcal{S}$  e  $q_n \in \mathcal{Z}$ , e  $((q_{i-1}, a_i), q_i) \in \delta$  para todo  $1 \leq i \leq n$ .*

Assim, um reconhecedor baseado num autómato  $\mathcal{A}$  aceita uma frase  $\mu$  se e só se  $\mu \in \mathcal{L}_{\mathcal{A}}$ .

Considere-se de novo o reconhecimento da frase  $\mu = -11$  pelo autómato  $\mathcal{A}_1$  apresentado anteriormente. Facilmente se verifica que a sequência de símbolos que constituem a frase  $\mu = -11 = -1\epsilon 1\epsilon$  são exactamente os mesmos símbolos das transições  $a_1 a_2 a_3 a_4 a_5$  utilizadas durante o reconhecimento de  $\mu$ , partindo de um estado inicial  $q_0 = A$  até um estado final  $q_5 = E$ . Logo  $-11 \in \mathcal{L}_{\mathcal{A}_1}$ .

No início desta secção 3.2 foi dito que toda a linguagem regular pode ser definida por um autómato finito. Este facto é verdade, no entanto, tal como veremos nas secções seguintes, podem existir vários autómatos finitos diferentes que definem exactamente a mesma linguagem regular. Nesta situação os autómatos dizem-se *equivalentes*. De seguida apresenta-se a sua definição formal.

**Definição 3.5** *Dois autómatos finitos  $\mathcal{A}_1$  e  $\mathcal{A}_2$  dizem-se equivalentes, e representa-se por  $\mathcal{A}_1 \equiv \mathcal{A}_2$ , se e só se  $\mathcal{L}_{\mathcal{A}_1} = \mathcal{L}_{\mathcal{A}_2}$ .*

Os autómatos finitos dividem-se em *determinístico* ou *não determinístico*, podendo qualquer um deles reconhecer uma linguagem regular. De seguida apresentam-se estes dois tipos de autómatos finitos.

**Exercício 3.1** *Suponha que um dado autómato finito  $\mathcal{A}$  reconhece uma linguagem regular  $\mathcal{L}$  e que  $\mathcal{L} \supset \mathcal{L}'$ , onde  $\mathcal{L}'$  é outra linguagem regular. Será que  $\mathcal{A}$  pode ser considerado um reconhecedor de  $\mathcal{L}'$ ? Justifique convenientemente a resposta.*

### Autômatos Finitos Não Determinísticos

Um autômato finito não determinístico — AFND — define-se do seguinte modo:

**Definição 3.6** *Um autômato finito  $\mathcal{A} = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$  diz-se não determinístico se:*

- *existe pelo menos um  $q \in \mathcal{Q}$  e um  $a \in \mathcal{T}$ , tal que  $((q, a), q'), ((q, a), q'') \in \delta$ , com  $q', q'' \in \mathcal{Q}$ , ou*
- *existe pelo menos uma transição- $\epsilon$ .*

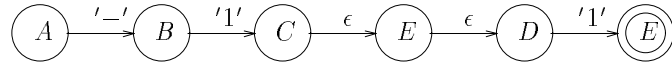
Informalmente são todos os autômatos finitos que ou têm pelo menos um estado com mais do que uma transição pelo mesmo símbolo, ou têm pelo menos uma transição- $\epsilon$ . Deste modo, a tabela de transição de estados é uma função  $\delta$  definida do seguinte modo:

$$\delta : \mathcal{Q} \times (\mathcal{T} \cup \{\epsilon\}) \rightarrow 2^{\mathcal{Q}}$$

Esta função dado um estado e um símbolo do vocabulário ou  $\epsilon$  dá como resultado um conjunto de estados.

Analisando o AF  $\mathcal{A}_1$ , apresentado na página 43, facilmente se verifica que  $\mathcal{A}_1$  é um AFND, pois contém transições- $\epsilon$ .

A aceitação de uma frase por parte de um AFND pode ser feita utilizando mais que um caminho (de um estado inicial para um final) diferente. Considere-se, por exemplo, o reconhecimento da frase  $\mu = -11$ , utilizando o autômato  $\mathcal{A}_1$ , apresentado graficamente na página 46. Como foi referido na altura esse não é o único caminho para se efectuar o reconhecimento de  $\mu$ . Por exemplo, existiria ainda o seguinte caminho:



Embora um autômato finito não determinístico defina uma linguagem regular, não é imediata a sua implementação num programa de computador. Isto acontece devido ao não determinismo inerente ao próprio autômato, que



origina que o reconhecedor produzido não seja determinístico e portanto terá de ser capaz de *recuperar decisões erradas*.

Apesar de ser possível implementar tal reconhecedor, vamos de seguida analisar os autómatos finitos determinísticos, nos quais este problema já não surge, sendo portanto mais simples construir um reconhecedor baseado nestes autómatos.

### Autómatos Finitos Determinísticos

Um autómato finito determinístico —AFD— define-se do seguinte modo:

**Definição 3.7** *Um autómato finito  $\mathcal{A} = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$  diz-se determinístico se:*

- $|\mathcal{S}| = 1$ ;
- para todo  $q \in \mathcal{Q}$  e  $a \in \mathcal{T}$ , existe no máximo um  $q' \in \mathcal{Q}$  tal que  $((q, a), q') \in \delta$  e
- o conjunto de transições é  $\delta \subseteq (\mathcal{Q} \times \mathcal{T}) \times \mathcal{Q}$ , i.é, não existem transições- $\epsilon$ .

A tabela de transição de estados é uma função  $\delta$  definida do seguinte modo:

$$\delta : \mathcal{Q} \times \mathcal{T} \rightarrow 2^{\mathcal{Q}}$$

Esta função dado um estado e um símbolo do vocabulário dá como resultado um estado. A função  $\delta$  é uma função parcial pois  $\exists_{(q,t) \in \mathcal{Q} \times \mathcal{T}} : \delta(q, t) = \perp$ .

Nos autómatos finitos determinísticos (ao contrário dos AFND) a aceitação de uma frase  $\mu$  é feita percorrendo o *único caminho possível*, do estado inicial para o estado final, para reconhecer os símbolos de  $\mu$ . Nestes autómatos nunca existe ambiguidade sobre qual a transição a efectuar (daí o seu determinismo e a sua utilidade prática).

Deste modo, os autómatos finitos determinísticos são mais fáceis de implementar e produzem reconhecedores bastante mais rápidos que os não determinísticos. No entanto, um autómato determinístico que define uma linguagem regular pode ser muito maior (em número de estados e de transições) que um não determinístico que define exactamente a mesma linguagem. Deste modo, quando se constrói um reconhecedor baseado em autómatos finitos é

necessário ter estes aspectos em conta. Na prática utilizam-se geralmente os autómatos finitos determinísticos devido à sua facilidade de implementação e rapidez.

Como foi referido anteriormente, existem métodos para converter gramáticas e expressões regulares em autómatos finitos. De seguida vão ser apresentados estes métodos, começando pela conversão num autómato não determinístico pois é o mais imediato.

### 3.2.2 Conversão de Gramáticas Regulares em AFND

Nesta secção é apresentada uma técnica que permite *converter* uma gramática regular num autómato finito não determinístico. Pelo termo *converter* entende-se construir um AFND que define a mesma linguagem que a gramática regular.

Deste modo, dada uma gramática  $G = (T, N, S, P)$  pretende-se obter um AFND  $\mathcal{A} = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$ , tal que  $\mathcal{L}_G = \mathcal{L}_{\mathcal{A}}$ .

A conversão de uma gramática regular num autómato finito não determinístico é feito seguindo um conjunto de regras. Estas regras são diferentes consoante a gramática seja regular à esquerda ou à direita. De seguida analisa-se o caso da gramática ser regular à direita e posteriormente o caso da gramática ser regular à esquerda.

Dada uma gramática  $G = (T, N, S, P)$  regular à direita então a sua conversão num autómato finito  $\mathcal{A} = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$  não determinístico é feita segundo as seguintes regras:

1.  $\mathcal{T} = T$ ;
2. A cada não terminal  $X \in N$  corresponde um estado  $X \in \mathcal{Q}$ . No caso particular do símbolo inicial da gramática  $S$  corresponde o estado inicial  $S \in \mathcal{S}$ ;
3. A cada produção da forma  $X \rightarrow \alpha_1 \dots \alpha_n Y$ , com  $X, Y \in N$  e  $\alpha_1 \dots \alpha_n \in T$  corresponde o seguinte conjunto de transições:

$$((X, \alpha_1), X_1), ((X_1, \alpha_2), X_2), \dots, ((X_{n-1}, \alpha_n), Y) \in \delta,$$

em que  $X_1, X_2, \dots, X_{n-1} \in \mathcal{Q}$  são novos estados;

4. A cada produção da forma  $X \rightarrow \alpha_1 \dots \alpha_n$ , com  $X \in N$  e  $\alpha_1 \dots \alpha_n \in T$  corresponde o seguinte conjunto de transições:

$$((X, \alpha_1), X_1), ((X_1, \alpha_2), X_2), \dots, ((X_{n-1}, \alpha_n), Z_0) \in \delta,$$

em que  $X_1, X_2, \dots, X_{n-1}, Z_0 \in \mathcal{Q}$  são novos estados e  $Z_0 \in \mathcal{Z}$  é um novo estado final;

5. A cada produção da forma  $X \rightarrow Y$  corresponde a seguinte transição  $((X, \epsilon), Y) \in \delta$ .

A conversão de uma gramática regular à esquerda é feita de modo semelhante.

Dada uma gramática  $G = (T, N, S, P)$  regular à esquerda então a sua conversão num autómato finito  $\mathcal{A} = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$  não determinístico é feita segundo as seguintes regras:

1.  $\mathcal{T} = T$ ;
2. A cada não terminal  $X \in N$  corresponde um estado  $X \in \mathcal{Q}$ . No caso particular do símbolo inicial da gramática  $S$  corresponde o estado final  $S \in \mathcal{S}$ ;
3. A cada produção da forma  $X \rightarrow Y\alpha_1 \dots \alpha_n$ , com  $X, Y \in N$  e  $\alpha_1 \dots \alpha_n \in T$  corresponde o seguinte conjunto de transições:

$$((Y, \alpha_1), X_1), ((X_1, \alpha_2), X_2), \dots, ((X_{n-1}, \alpha_n), X) \in \delta,$$

em que  $X_1, X_2, \dots, X_{n-1} \in \mathcal{Q}$  são novos estados;

4. A cada produção da forma  $X \rightarrow \alpha_1 \dots \alpha_n$ , com  $X \in N$  e  $\alpha_1 \dots \alpha_n \in T$  corresponde o seguinte conjunto de transições:

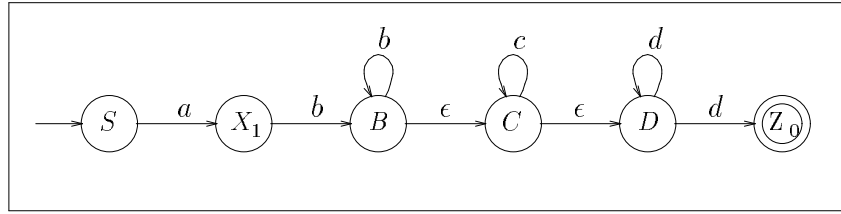
$$((S_0, \alpha_1), X_2), ((X_2, \alpha_2), X_3), \dots, ((X_{n-1}, \alpha_n), X) \in \delta,$$

em que  $X_1, X_2, \dots, X_{n-1}, Z_0 \in \mathcal{Q}$  são novos estados e  $S_0 \in \mathcal{S}$  é um novo estado inicial;

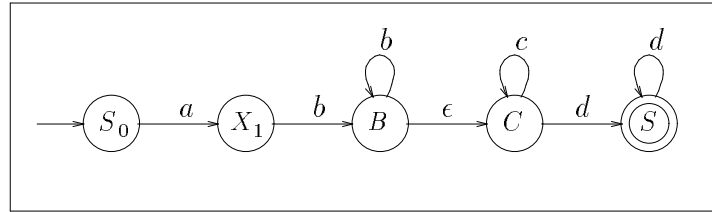
5. A cada produção da forma  $X \rightarrow Y$  corresponde a seguinte transição  $((Y, \epsilon), X) \in \delta$ .

Considere as gramáticas regulares definidas na secção 2.3.4 (página 34) e que definem a mesma linguagem (pois definem a mesma linguagem que a expressão regular  $e = ab^+c^*d^+$ ). Considere ainda que se pretende obter os respectivos autómatos finitos não determinísticos. Como foi referido, a conversão de gramáticas regulares em AFND é feito aplicando um conjunto de regras, definidas nesta secção.

Assim, no caso da gramática regular à direita obtém-se o seguinte AFND  $\mathcal{A}_1$ :



Para a gramática regular à esquerda obtém-se o AFND  $\mathcal{A}_2$  seguinte:



Como facilmente se verifica os dois AFND não são iguais (independente da diferença nos identificadores dos estados), pois o AFND obtido a partir da gramática regular à direita tem mais um estado que o obtido a partir da gramática regular à esquerda. No entanto,  $\mathcal{A}_1 \equiv \mathcal{A}_2$ , i.é. os dois autómatos são equivalentes, pois definem a mesma linguagem como se pode constatar pela definição 3.4.

### 3.2.3 Conversão de Expressões Regulares em AFND

Na secção anterior vimos como é possível construir um AFND a partir de uma gramática regular. Nesta secção apresenta-se um método para construir um AFND mas agora a partir de uma expressão regular.

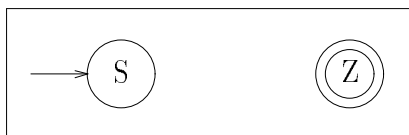
Formalmente o que se pretende é dada uma expressão regular  $p$  obter um AFND  $\mathcal{A} = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$ , tal que  $\mathcal{L}_p = \mathcal{L}_{\mathcal{A}}$ .

Facilmente se verifica pela definição de expressão regular (secção 2.2) que o vocabulário  $\mathcal{T}$  do AFND  $\mathcal{A}$  é constituído pelos símbolos do vocabulário  $V$  que compõem a expressão regular  $p$ . De seguida apresenta-se um método para determinar os outros elementos que constituem um autómato finito não determinístico.

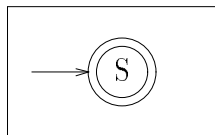
A conversão de expressões regulares em AFND é feita seguindo um conjunto de regras. Para cada uma das formas básicas de expressões regulares definidas (ver secção 2.2) associa-se um autómato finito não determinístico que define a mesma linguagem.

As regras de conversão são:

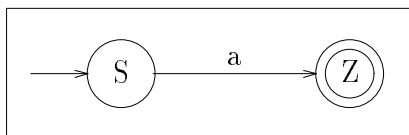
1.  $e = \emptyset$  converte em



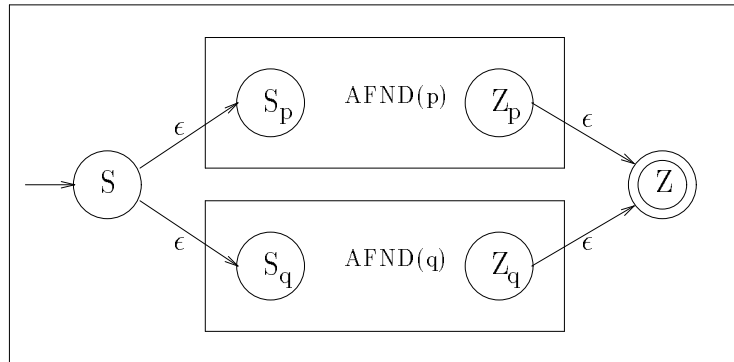
2.  $e = \epsilon$  converte em



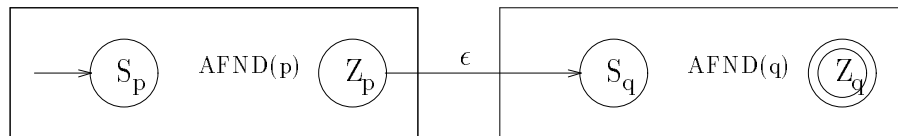
3.  $e = a$  converte em



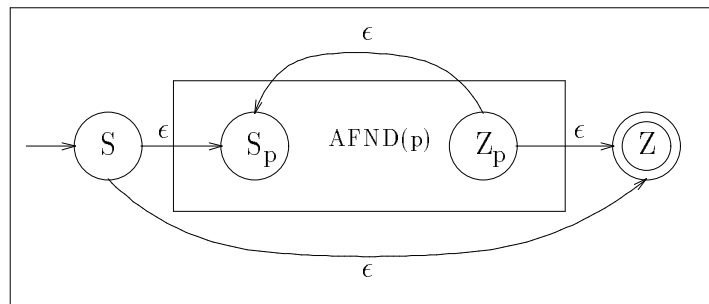
4.  $e = p + q$  converte em



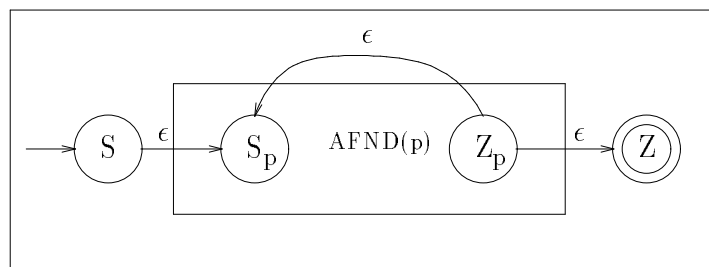
5.  $e = pq$  converte em



6.  $e = p^*$  converte em



7.  $e = p^+$  converte em



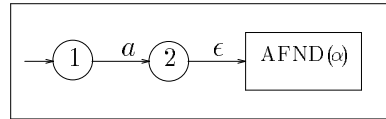
A conversão de uma expressão regular composta  $p$  (i.é, constituída por mais que uma das formas básicas definidas) é feita decompondo  $p$  nas subexpressões que a constituem. Posteriormente, com base na estrutura sintática de  $p$ , combinam-se os vários AFND indutivamente (usando as últimas quatro regras de conversão) até se obter o AFND para toda a expressão regular.

Considere que se pretende construir o AFND  $\mathcal{A}_1$  que define a mesma linguagem  $\mathcal{L}$  que a expressão regular  $e = a(bc^* + d)^+$ . O primeiro passo a efectuar é decompor  $e$  nas várias subexpressões que a constituem, tal como foi dito anteriormente.

Na primeira situação tem-se uma expressão regular do tipo  $e = pq$ , com  $p = a$  e  $q = (bc^* + d)^+$ . Considerando a seguinte decomposição de  $e$

$$e = a \underbrace{(bc^* + d)^+}_{\alpha}$$

e usando a regra 5 obtém-se o seguinte AFND:

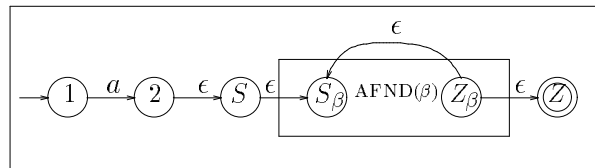


em que  $\text{AFND}(\alpha)$  representa o AFND que se irá associar a  $\alpha$ .

Prosseguindo a decomposição de  $e$  e considerando agora o operador de fecho transitivo temos:

$$e = a \underbrace{(bc^* + d)^+}_{\beta}$$

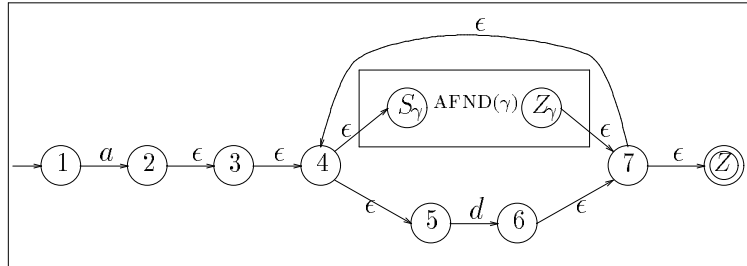
Usando a regra 7 obtém-se o AFND:



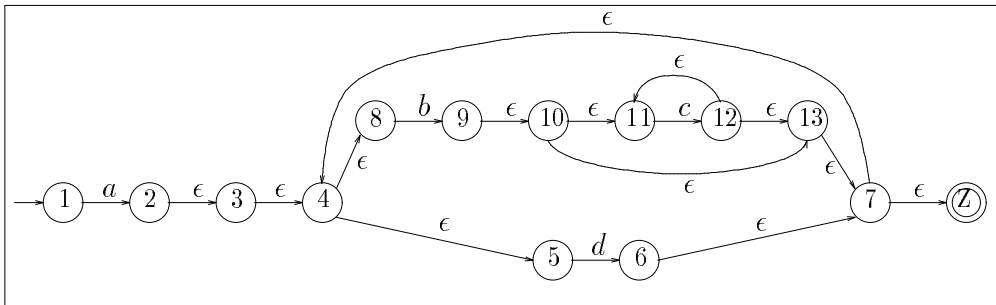
Continuando temos agora uma expressão da forma  $e = p + q$ , com  $p = bc^*$  e  $q = d$ . Considerando a decomposição

$$a(\underbrace{bc^*}_{\gamma} + d)^+$$

e a regra 4 obtém-se o AFND seguinte:



Por último, e avançando o passo em que se considerava a decomposição  $e = a(b(\underbrace{c^*}_{\delta} + d)^+)$ , obtém-se o AFND que define a linguagem  $\mathcal{L}$



**Exercício 3.2** Considere a seguinte expressão regular

$$e = (ab^*a) + a$$

Determine um autómato finito não determinístico  $\mathcal{A}$ , tal que  $\mathcal{L}_e = \mathcal{L}_{\mathcal{A}}$ .



### 3.2.4 Conversão de AFND em AFD

Os autómatos finitos não determinísticos, embora definam uma linguagem regular, não são geralmente utilizados como reconhecedores de uma linguagem regular devido às seguintes razões:

- É difícil simular num programa de computador situações em que existem várias transições, a partir de um estado, etiquetadas pelo mesmo símbolo. Nesta situação o programa não "*sabe*" para qual dos estados ir, podendo assim tomar uma decisão errada. Posteriormente terá de saber recuperar dessa decisão. Um outro facto que causa ambiguidade é a existência de transições- $\epsilon$ . Neste caso o programa pode optar por transitar de estado pela transição- $\epsilon$  ou por uma outra transição etiquetada pelo símbolo a reconhecer.
- A segunda razão tem a ver com questões de eficiência e os reconhecedores baseados em AFNDs são considerados bastante ineficientes.

Apesar de ser possível implementar um reconhecedor baseado num AFND, na prática utilizam-se antes os reconhecedores baseados em AFD, pois nestes autómatos estas características indesejáveis não se verificam. Deste modo, uma solução para resolver este problema consiste em converter os AFNDs em AFDs que *definem exactamente as mesmas linguagens*. Posteriormente, o reconhecedor da linguagem é implementado baseado no AFD. Sendo assim, é necessário estudar métodos de conversão entre estes dois tipos de autómatos finitos.

De seguida apresenta-se um método para converter um AFND num AFD que define a mesma linguagem.

A ideia geral do método de conversão de um AFND num AFD é a seguinte: cada estado do AFD corresponde a um conjunto de estados do AFND. Isto acontece porque, tal como foi referido na secção 3.2.1, o reconhecimento de uma frase por um AFND pode ser feito utilizando caminhos diferentes (e consequentemente sequências de estados diferentes). Assim, cada estado de um AFD contém a informação (i.é., os estados) que se podem alcançar a partir de um estado do AFND transitando por um dado símbolo. Cada estado do AFD inclui ainda todos os estados que se alcançam no AFND por transições- $\epsilon$ , pois, como é óbvio, qualquer um desses estados pode também ser atingido.

Neste método, um reconhecedor baseado num AFD após processar uma frase constituída pelos símbolos  $a_1a_2 \dots a_n$  está num estado  $q$ , que representa o conjunto de estados do AFND que se podem alcançar a partir dos seus estados iniciais e percorrendo o caminho  $a_1a_2 \dots a_n$ . Daqui resulta que o número de estados do AFD pode ser muito superior ao do AFND. Na pior situação pode ser exponencialmente superior, no entanto na prática esta situação é muito rara.

Basicamente o que se pretende é: dado um AFND  $\mathcal{A}_{nd} = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$  obter um AFD  $\mathcal{A}_d = (\mathcal{T}', \mathcal{Q}', \mathcal{S}', \mathcal{Z}', \delta')$ , tal que  $\mathcal{A}_{nd} \equiv \mathcal{A}_d$ .

Antes de se apresentar formalmente como isto pode ser feito, define-se de seguida uma função auxiliar —  $\epsilon$ -*fecho* — que será usada na conversão dos autómatos finitos. Informalmente o que esta função "*diz*" é que o *fecho* de um conjunto de estado  $X$  é o conjunto de estados que se atingem partindo dos estados de  $X$  através de transições- $\epsilon$ , incluindo ainda o próprio  $X$ . Formalmente esta função define-se do seguinte modo:

$$\begin{aligned} \epsilon\text{-fecho}: 2^{\mathcal{Q}} &\rightarrow 2^{\mathcal{Q}} \\ \epsilon\text{-fecho}(X) &= X \cup \bigcup_{x \in X} \epsilon\text{-fecho}(\delta(x, \epsilon)) \end{aligned}$$

Considerando os autómatos finitos  $\mathcal{A}_{nd}$  e  $\mathcal{A}_d$  referidos anteriormente, a conversão é feita do seguinte modo:

- $\mathcal{T}' = \mathcal{T}$
- $\mathcal{S}' = \epsilon\text{-fecho}(\{\mathcal{S}\})$
- $\delta'(X, t) = \epsilon\text{-fecho}(\bigcup_{q \in X} \delta(q, t))$
- $\mathcal{Q}'$  define-se recursivamente por:
  - $\mathcal{S}' \in \mathcal{Q}'$
  - $X \in \mathcal{Q}' \wedge \delta'(X, t) \neq \emptyset \Rightarrow \delta'(X, t) \in \mathcal{Q}'$
- $\mathcal{Z}' = \{X \in \mathcal{Q}' \mid X \cap \mathcal{Z} \neq \emptyset\}$

Informalmente estas regras dizem o seguinte: o vocabulário do autómato finito determinístico  $\mathcal{T}'$  é o mesmo que o do não determinístico. O seu estado inicial  $\mathcal{S}'$  é o conjunto formado pelos estados que se alcançam a partir do

estado inicial do AFND transitando por  $\epsilon$ . O conjunto de transições  $\delta'$  obtém-se do seguinte modo: para cada estado  $X$  do AFD e para cada símbolo do vocabulário  $t$  obtém-se, caso exista, uma transição para um novo estado. Este novo estado do AFD é obtido através do  $\epsilon$ -*fecho* da união dos estados que se alcançam no AFND a partir dos estados que constituem  $X^1$ , transitando por  $t$ . O conjunto de estados  $\mathcal{Q}'$  do AFD é constituído pelo seu estado inicial e ainda por todos os estados que se alcançam em transições de um estado de  $\mathcal{Q}'$  por um símbolo do vocabulário. O conjunto de estados finais  $\mathcal{Z}'$  é formado por todos estados do AFD que incluem algum estado final do AFND.

Considere o AFND  $\mathcal{A}_1 = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$  construído na secção 3.2.4 (apresentado graficamente na página 56) e que se pretende obter um AFD  $\mathcal{A}_2 = (\mathcal{T}', \mathcal{Q}', \mathcal{S}', \mathcal{Z}', \delta')$  equivalente.

Utilizando o método de conversão referido anteriormente têm-se:

- $\mathcal{T}' = \{a, b, c, d\}$
- $\mathcal{S}' = \epsilon\text{-fecho}(\{1\}) = \{1\} \cup \epsilon\text{-fecho}(\emptyset) = \{1\}$ ,  
logo  $\{1\} \in \mathcal{Q}'$ .
- $\delta'(\{1\}, a) = \epsilon\text{-fecho}(\delta(1, a)) = \epsilon\text{-fecho}(\{2\}) = \{2\} \cup \{3, 4, 5, 8\} = \{2, 3, 4, 5, 8\}$ ,

pela terceira regra  $\{2, 3, 4, 5, 8\} \in \mathcal{Q}'$ . Este é o único estado novo do AFD que se obtém transitando a partir do estado inicial, pois todas as restantes transições por símbolos do vocabulário não existem, como se prova de seguida:

$$\delta'(\{1\}, b) = \epsilon\text{-fecho}(\delta(1, b)) = \epsilon\text{-fecho}(\emptyset) = \emptyset$$

$$\delta'(\{1\}, c) = \epsilon\text{-fecho}(\delta(1, c)) = \epsilon\text{-fecho}(\emptyset) = \emptyset$$

$$\delta'(\{1\}, d) = \epsilon\text{-fecho}(\delta(1, d)) = \epsilon\text{-fecho}(\emptyset) = \emptyset,$$

Vão-se agora calcular os estados que se atingem a partir do novo estado obtido  $\{2, 3, 4, 5, 8\}$ .

$$\delta'(\{2, 3, 4, 5, 8\}, a) = \delta'(\{2, 3, 4, 5, 8\}, c) = \emptyset$$

---

<sup>1</sup>Recorde-se que  $X$ , i.é., um estado de um AFD, é formado por um ou mais estados do AFND.

$$\begin{aligned}\delta'(\{2, 3, 4, 5, 8\}, b) &= \epsilon\text{-fecho}(\{\delta(2, b), \delta(3, b), \delta(4, b), \delta(5, b), \delta(8, b)\}) = \\ &= \epsilon\text{-fecho}(\{9\}) = \{9\} \cup \{10, 11, 13, 7, 4, 8, 5, Z\} = \\ &= \{4, 5, 7, 8, 9, 10, 11, 13, Z\}\end{aligned}$$

$$\begin{aligned}\delta'(\{2, 3, 4, 5, 8\}, d) &= \epsilon\text{-fecho}(\{\delta(2, d), \delta(3, d), \delta(4, d), \delta(5, d), \delta(8, d)\}) = \\ &= \epsilon\text{-fecho}(\{6\}) = \{6\} \cup \{7, 4, 8, 5, Z\} = \\ &= \{4, 5, 6, 7, 8, Z\}\end{aligned}$$

Obtendo-se assim dois novos estados:

$$\{4, 5, 7, 8, 9, 10, 11, 13, Z\}, \{4, 5, 6, 7, 8, Z\} \in \mathcal{Q}'$$

Prosseguindo o cálculo do conjunto transições, e apresentando apenas os cálculos das transições que realmente existem no AFD, vem:

$$\delta'(\{4, 5, 7, 8, 9, 10, 11, 13, Z\}, b) = \epsilon\text{-fecho}(\{9\}) = \{4, 5, 7, 8, 9, 10, 11, 13, Z\}$$

$$\delta'(\{4, 5, 7, 8, 9, 10, 11, 13, Z\}, c) = \epsilon\text{-fecho}(\{12\}) = \{4, 5, 7, 8, 11, 12, 13, Z\}$$

$$\delta'(\{4, 5, 7, 8, 9, 10, 11, 13, Z\}, d) = \epsilon\text{-fecho}(\{6\}) = \{4, 5, 6, 7, 8, Z\}$$

$$\delta'(\{4, 5, 6, 7, 8, Z\}, b) = \epsilon\text{-fecho}(\{9\}) = \{4, 5, 7, 8, 9, 10, 11, 13, Z\}$$

$$\delta'(\{4, 5, 6, 7, 8, Z\}, d) = \epsilon\text{-fecho}(\{6\}) = \{4, 5, 6, 7, 8, Z\}$$

O único estado novo que se obtém é  $\{4, 5, 7, 8, 12, 13, Z\} \in \mathcal{Q}'$ . Determinando as transições que lhe estão associadas têm-se:

$$\delta'(\{4, 5, 7, 8, 11, 12, 13, Z\}, b) = \epsilon\text{-fecho}(\{9\}) = \{4, 5, 7, 8, 9, 10, 11, 13, Z\}$$

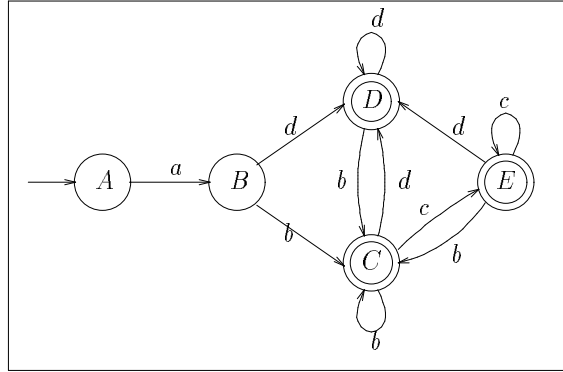
$$\delta'(\{4, 5, 7, 8, 11, 12, 13, Z\}, c) = \epsilon\text{-fecho}(\{12\}) = \{4, 5, 7, 8, 11, 12, 13, Z\}$$

$$\delta'(\{4, 5, 7, 8, 11, 12, 13, Z\}, d) = \epsilon\text{-fecho}(\{6\}) = \{4, 5, 6, 7, 8, Z\}$$

Obtendo-se deste modo o conjunto de transições  $\delta'$ .

- $\mathcal{Q}' = \{ \{1\}, \{2, 3, 4, 5, 8\}, \{4, 5, 6, 7, 8, Z\}, \{4, 5, 7, 8, 9, 10, 11, 13, Z\}, \{4, 5, 7, 8, 11, 12, 13, Z\} \}$
- $\mathcal{Z}' = \{ \{4, 5, 6, 7, 8, Z\}, \{4, 5, 7, 8, 9, 10, 11, 13, Z\}, \{4, 5, 7, 8, 11, 12, 13, Z\} \}$

Fazendo  $A = \{1\}$ ,  $B = \{2, 3, 4, 5, 8\}$ ,  $C = \{4, 5, 7, 8, 9, 10, 11, 13, Z\}$ ,  $D = \{4, 5, 6, 7, 8, Z\}$  e  $E = \{4, 5, 7, 8, 11, 12, 13, Z\}$ , obtém-se a seguinte representação gráfica de  $\mathcal{A}_2$ :



Como facilmente se verifica no exemplo anterior, este processo de conversão é relativamente simples. No entanto, tal como foi efectuado/apresentado é muito pouco perceptível para quem o realiza. Sendo assim, as probabilidades de um engano aumentam.

Uma técnica bastante mais sistemática para efectuar a conversão entre um AFND e um AFD consiste na utilização de uma *tabela* que auxilia o processo de conversão. Esta tabela é constituída por várias colunas. A primeira coluna está associada aos estados do AFD e as restantes colunas estão associadas a cada um dos símbolos do vocabulário. Cada linha da tabela define as transições associadas ao estado que consta da sua primeira coluna.

Esta tabela é preenchida do seguinte modo: a primeira coluna da primeira linha preenche-se com o estado inicial do AFD (calculado através do  $\epsilon$ -fecho dos estados iniciais do AFND). Cada uma das colunas restantes preenchem-se com  $\epsilon$ -fecho dos estados que se alcançam partindo do estado inicial e transitando por ramos com o peso do símbolo associado à coluna. Cada um destes conjuntos de símbolos encontrados corresponderá um novo estado do AFD. Para cada novo estado proceder-se-á de igual modo, i.é, acrescentando uma linha na tabela, pondo esse estado na primeira coluna e determinando as suas transições.

Vai-se agora aplicar esta técnica ao AFND  $\mathcal{A}_1 = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$  definido na secção 3.2.1 (apresentado graficamente na página 45), para obter um AFD

$\mathcal{A}_2 = (\mathcal{T}', \mathcal{Q}', \mathcal{S}', \mathcal{Z}', \delta')$  equivalente.

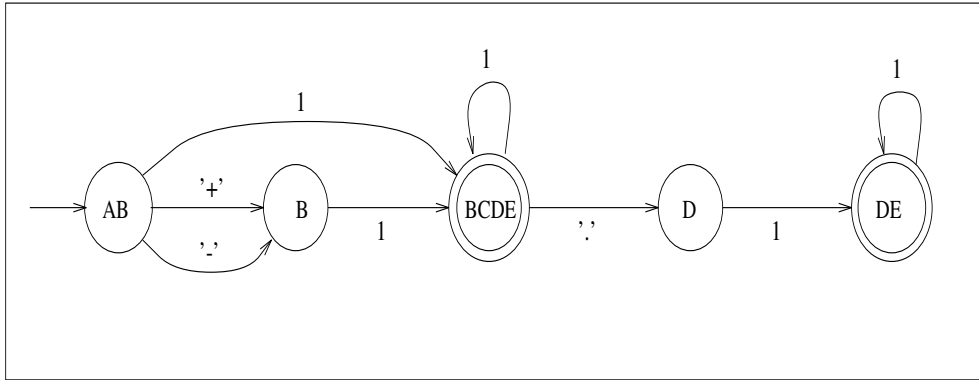
Aplicando então este processo ao caso em estudo, têm-se:

$$\begin{aligned}\mathcal{T}' &= \mathcal{T} \\ \mathcal{S}' &= \epsilon\text{-fecho}(\{A\}) = \{AB\}\end{aligned}$$

e a tabela com as transições e respectivos estados do AFD é:

$\mathcal{Q}_\infty \setminus \mathcal{T}$	'+'	'-'	1	'.'
$\mathcal{S}' = \{AB\}$	$\epsilon\text{-fecho}(\{B\})$ $= \{B\}$	$\epsilon\text{-fecho}(\{B\})$ $= \{B\}$	$\epsilon\text{-fecho}(\{C\})$ $= \{BCDE\}$	
$\{B\}$			$\epsilon\text{-fecho}(\{C\})$ $= \{BCDE\}$	
$\{BCDE\}$			$\epsilon\text{-fecho}(\{CE\})$ $= \{BCDE\}$	$\epsilon\text{-fecho}(\{D\})$ $= \{D\}$
$\{D\}$			$\epsilon\text{-fecho}(\{E\})$ $= \{DE\}$	
$\{DE\}$			$\epsilon\text{-fecho}(\{E\})$ $= \{DE\}$	

Finalmente, representado graficamente o AFD  $\mathcal{A}_2$ :



**Exercício 3.3** Converta o AFND construído no exercício 3.2 num autômato finito determinístico.

**Exercício 3.4** Considere a gramática do exercício 2.15. Determine o seu autômato não determinístico e o determinístico.

**Exercício 3.5** Considere a seguinte gramática  $G = (T, N, S, P)$ , em que

$$\begin{aligned} T &= \{a, b\} & P &= \{S \rightarrow a \mid bB \mid bA \\ N &= \{S, A, B\} & A &\rightarrow b \mid aS \\ S &= S & B &\rightarrow abaA \mid A \\ & & & \} \end{aligned}$$

Calcule uma expressão regular equivalente, converta-a num autómato não determinístico equivalente e, posteriormente, num autómato determinístico.

### 3.2.5 Conversão de Gramáticas e Expressões Regulares em AFD

A implementação de reconhecedores de linguagens regulares é geralmente feito simulando um autómato finito determinístico num programa de computador, tal como foi referido na secção 3.2.4. Sendo assim, é importante definir técnicas que a partir dos modelos estudados para definir a sintaxe de uma linguagem regular, permitam obter um AFD que define essa mesma linguagem.

Um método geralmente usada para o fazer consiste em utilizar as técnicas já apresentadas nesta secção: primeiro as expressões regulares ou as gramáticas regulares são convertidas num autómato finito não determinístico, utilizando as técnicas apresentadas nas secções 3.2.3 e 3.2.2 respectivamente. Posteriormente, este AFND é convertido num AFD equivalente, utilizando a técnica referida na secção 3.2.4.

Deste modo, a construção do AFND é um passo intermédio na obtenção do AFD. No entanto, existem métodos [ASU86] que permitem obter directamente um AFD a partir de uma expressão regular, sem a construção explícita do AFND. Porém estes métodos não vão ser estudados neste texto, e considera-se que a construção de um AFD é sempre feita a partir de um AFND.

### 3.2.6 Reconhecedores baseados em AFD

A tabela de transições de estados está directamente ligada à implementação dos autómatos finitos num programa de computador. Embora existam várias alternativas para definir o conjunto de transições de estado num

programa<sup>2</sup>, o modo mais simples de o fazer consiste em definir esta tabela num *array bidimensional*. Assim, pode ser utilizada a seguinte estrutura de dados:

$$TT = \underline{\text{array}}[Q \times T] \text{ de } Q \cup \{erro\}$$

A utilização desta tabela permite um rápido acesso às transições de um estado dado um símbolo. No entanto, para autómatos que tenham poucas transições por símbolos do vocabulário (e muitas transições- $\epsilon$ ) existe um grande desaproveitamento de espaço, pois uma grande parte das posições da tabela está desocupada.

Este problema poderá ser resolvido utilizando qualquer uma das alternativas referidas, nomeadamente as listas de adjacência e os *arrays D'hashing*. Porém estas alternativas têm a desvantagem de os acessos às transições de um estado serem mais lentas.

Neste texto vai-se apenas considerar a estrutura de dados definida anteriormente. De seguida apresenta-se o algoritmo da função que reconhece uma linguagem definida através de um autómato finito determinístico.

**função**  $\mathcal{R}_{\mathcal{L}}(\delta : TT, \gamma : T^*) : \{aceita, erro\}$

$$\left\{ \begin{array}{l} \alpha \leftarrow \mathcal{S}; \\ \textbf{enquanto} \ (\gamma \neq \epsilon) \wedge (\alpha \neq erro) \\ \quad \longrightarrow \left\{ \begin{array}{l} \alpha \leftarrow \delta(\alpha, head(\gamma)); \\ \gamma \leftarrow tail(\gamma) \end{array} \right. \\ \textbf{fenquanto}; \\ \quad \textbf{se} \quad (\alpha \in \mathcal{Z}) \wedge (\gamma = \epsilon) \\ \quad \quad \longrightarrow \{r \leftarrow aceita \\ \quad \textbf{senão} \longrightarrow \{r \leftarrow erro \\ \quad \textbf{fse} \\ \mathcal{R}_{\mathcal{L}} \leftarrow r \end{array} \right.$$

Como se verifica pelo algoritmo apresentado, os reconhecedores baseados em AFD podem ser implementados num programa de computador de uma forma bastante simples e eficiente.

---

<sup>2</sup>Listas de adjacência, *arrays D'hashing*.



### 3.2.7 Análise de um Problema

Nesta secção vai ser analisado um problema prático, que requer a utilização de um reconhecedor de uma linguagem regular. Na sua resolução serão utilizadas as técnicas apresentadas nas secções anteriores. Assim, primeiro vai-se começar por definir quais as frases válidas da linguagem, escrevendo uma gramática independente do contexto que define a sua sintaxe (*cf.* secção 2.3.2). Posteriormente, esta gramática será convertida numa expressão regular (utilizando as técnicas apresentadas na secção 2.3.5) que por sua vez será convertida num autómato finito não determinístico (*cf.* secção 3.2.3). O AFND obtido é de seguida convertido num autómato finito determinístico equivalente (*cf.* secção 3.2.4), de modo a ser possível utilizar o algoritmo apresentado na secção 3.2.6 para construir o reconhecedor da linguagem. No apêndice B apresenta-se a implementação em linguagem C do respectivo reconhecedor.

**Problema 1** *A configuração de uma placa gráfica de um PC obedece ao seguinte protocolo: a comunicação estabelece-se enviando um código inicial, constituído pelo padrão de bits 000. Posteriormente, são enviados valores em binário, de comprimento arbitrário, para configurar vários parâmetros da placa. Esses valores são separados por uma sequência especial de bits: 110. Para indicar o fim da comunicação envia-se a sequência de bits 111.*

*Construa um programa para efectuar o reconhecimento das frases que obedecem a este protocolo.*

#### Resolução

As frases da linguagem, que define este protocolo, são constituídas por 000 seguido por uma sequência de valores e terminadas por 111. Os valores podem ser um único valor (assume-se que tem sempre de se enviar pelo menos um valor numa comunicação) ou um valor seguido do separador e de mais valores. O separador por sua vez é a sequência 110. Escrevendo a sintaxe destas frases numa gramática obtém-se:

- $G = (T, N, S, P)$ , com:

$$\begin{array}{ll}
 T = \{0, 1\} & P = \{ \begin{array}{ll} S & \rightarrow 000 \text{ Vals } 111 \\
 N = \{S, \text{Vals}, \text{Val}, \text{Sep}\} & \text{Vals} \rightarrow \text{Val} \mid \text{Val Sep Vals} \\
 & \text{Val} \rightarrow 0 \text{ Val} \mid 1 \text{ Val} \mid 0 \mid 1 \\
 & \text{Sep} \rightarrow 110 \end{array} \\
 & \}
 \end{array}$$

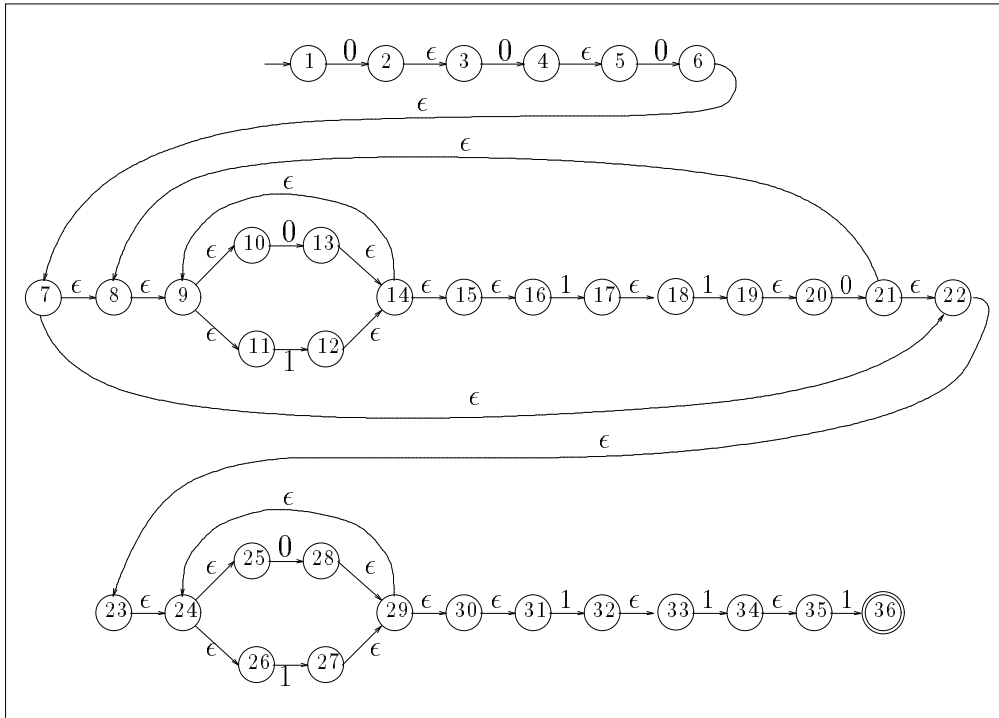
- Conversão da Gramática numa Expressão Regular:

$$\left\{ \begin{array}{l} S = 000 Vals 111 \\ Vals = Val + ValSepVals \\ Val = 0 Val + 1 Val + 0 + 1 \\ Sep = 110 \end{array} \right. \Rightarrow \left\{ \begin{array}{l} S = 000 Vals 111 \\ Vals = Val + Val 110 Vals \\ Val = (0 + 1) + (0 + 1)Val \end{array} \right. \Rightarrow$$

$$\left\{ \begin{array}{l} S = 000 Vals 111 \\ Vals = Val + Val 110 Vals \\ Val = (0 + 1)^*(0 + 1) = (0 + 1)^+ \end{array} \right. \Rightarrow \left\{ \begin{array}{l} S = 000 Vals 111 \\ Vals = (0 + 1)^+ + (0 + 1)^+ 110 Vals \end{array} \right.$$

$$\left\{ \begin{array}{l} S = 000 Vals 111 \\ Vals = ((0 + 1)^+ 110)^*(0 + 1)^+ \end{array} \right. \Rightarrow \left\{ S = 000 ((0 + 1)^+ 110)^*(0 + 1)^+ 111 \right.$$

- Conversão da Expressão Regular num AFND:



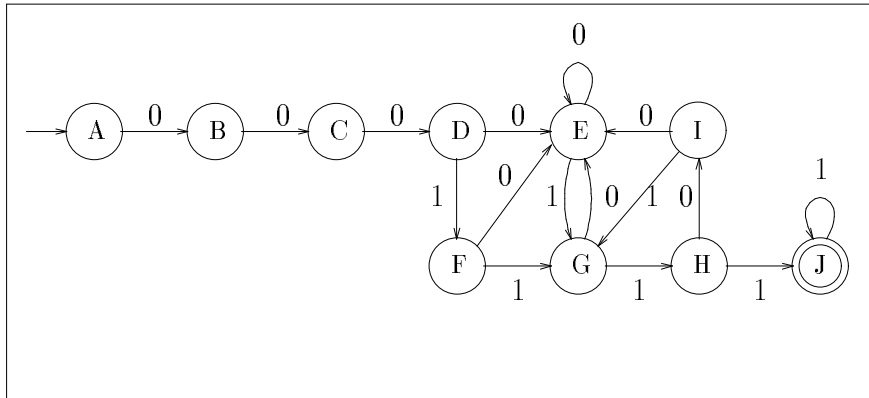
- Conversão do AFND num AFD equivalente:

$\mathcal{Q} \setminus \mathcal{T}$	0	1
$A = \{1\}$	$\epsilon\text{-fecho}(\{2\}) = \{2, 3\}$	
$B = \{2, 3\}$	$\epsilon\text{-fecho}(\{4\}) = \{4, 5\}$	
$C = \{4, 5\}$	$\epsilon\text{-fecho}(\{6\}) = \{6, 7, 8, 9, 10, 11, 22, 23, 24, 25, 26\}$	
$D = \{6, 7, 8, 9, 10, 11, 22, 23, 24, 25, 26\}$	$\epsilon\text{-fecho}(\{13, 28\}) = \{13, 14, 15, 16, 9, 10, 11, 28, 29, 30, 31, 24, 25, 26\}$	$\epsilon\text{-fecho}(\{12, 27\}) = \{12, 14, 15, 16, 9, 10, 11, 27, 29, 30, 31, 24, 25, 26\}$
$E = \{13, 14, 15, 16, 9, 10, 11, 28, 29, 30, 31, 24, 25, 26\}$	$\epsilon\text{-fecho}(\{13, 28\}) = \{13, 14, 15, 16, 9, 10, 11, 28, 29, 30, 31, 24, 25, 26\}$	$\epsilon\text{-fecho}(\{12, 17, 27, 32\}) = \{12, 14, 15, 16, 9, 10, 11, 17, 18, 27, 29, 30, 31, 32, 24, 25, 26, 33\}$
$F = \{12, 14, 15, 16, 9, 10, 11, 27, 29, 30, 31, 24, 25, 26\}$	$\epsilon\text{-fecho}(\{13, 28\}) = \{13, 14, 15, 16, 9, 10, 11, 28, 29, 30, 31, 24, 25, 26\}$	$\epsilon\text{-fecho}(\{12, 17, 27, 32\}) = \{12, 14, 15, 16, 9, 10, 11, 17, 18, 27, 29, 30, 31, 32, 24, 25, 26, 33\}$
$G = \{12, 14, 15, 16, 9, 10, 11, 17, 18, 27, 29, 30, 31, 32, 24, 25, 26, 33\}$	$\epsilon\text{-fecho}(\{13, 28\}) = \{13, 14, 15, 16, 9, 10, 11, 28, 29, 30, 31, 24, 25, 26\}$	$\epsilon\text{-fecho}(\{12, 17, 19, 27, 32, 34\}) = \{12, 14, 15, 16, 9, 10, 11, 17, 18, 19, 20, 27, 29, 30, 31, 32, 24, 25, 26, 33, 34, 35\}$
$H = \{12, 14, 15, 16, 9, 10, 11, 17, 18, 19, 20, 27, 29, 30, 31, 32, 24, 25, 26, 33, 34, 35\}$	$\epsilon\text{-fecho}(\{13, 21, 28\}) = \{13, 14, 15, 16, 9, 10, 11, 21, 8, 22, 23, 28, 29, 30, 31, 24, 25, 26\}$	$\epsilon\text{-fecho}(\{12, 17, 19, 27, 32, 34, 36\}) = \{12, 14, 15, 16, 9, 10, 11, 17, 18, 19, 20, 27, 29, 30, 31, 32, 33, 24, 25, 26, 34, 35, 36\}$
$I = \{13, 14, 15, 16, 9, 10, 11, 21, 8, 22, 23, 28, 29, 30, 31, 24, 25, 26\}$	$\epsilon\text{-fecho}(\{13, 28\}) = \{13, 14, 15, 16, 9, 10, 11, 28, 29, 30, 31, 24, 25, 26\}$	$\epsilon\text{-fecho}(\{12, 17, 27, 32\}) = \{12, 14, 15, 16, 9, 10, 11, 17, 18, 27, 29, 30, 31, 32, 24, 25, 26, 33\}$
$J = \{12, 14, 15, 16, 9, 10, 11, 17, 18, 19, 20, 27, 29, 30, 31, 32, 33, 24, 25, 26, 34, 35, 36\}$	$\epsilon\text{-fecho}(\{13, 21, 28\}) = \{13, 14, 15, 16, 9, 10, 11, 21, 8, 22, 23, 28, 29, 30, 31, 24, 25, 26\}$	$\epsilon\text{-fecho}(\{12, 17, 19, 27, 32, 34, 36\}) = \{12, 14, 15, 16, 9, 10, 11, 17, 18, 19, 20, 27, 29, 30, 31, 32, 33, 24, 25, 26, 34, 35, 36\}$

O autómato finito determinístico que define a linguagem do protocolo é  $\mathcal{A} = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$  com  $\mathcal{T} = \{0, 1\}$ ,  $\mathcal{Q} = \{A, B, C, D, E, F, G, H, I\}$ ,  $\mathcal{S} = A$ ,  $\mathcal{Z} = \{I\}$  e

$\mathcal{Q} \setminus \mathcal{T}$	0	1
A	B	
B	C	
C	D	
D	E	F
E	E	G
F	E	G
G	E	H
H	I	J
I	E	G
J	I	J

- Representando  $\mathcal{A}$  graficamente temos:



No apêndice B apresenta-se a implementação, em linguagem C, deste reconhecedor, utilizando o algoritmo da secção 3.2.6.

□

**Exercício 3.6** Considere a seguinte expressão regular *sb* que define uma linguagem cujas frases são símbolos básicos de uma linguagem de programação:

$$sb = \text{'if'} + \text{'then'} + \text{'while'} + \text{If} + ('+' + '-' + \epsilon)(0 + \dots + 9)^+$$

Implemente na linguagem C um reconhecedor para esta linguagem.

### 3.3 Reconhecimento de Linguagens Não-Regulares Top-Down

Um *reconhecedor Top-Down* (*reconhecedor descendente*) efectua o *parsing* de uma frase construindo a árvore de derivação partindo da raiz e terminando nas folhas, criando os nodos da árvore segundo uma travessia *preorder*<sup>3</sup>.

$$S \Rightarrow^* f$$

Este processo pode também ser visto como a derivação pela esquerda da frase a reconhecer (*c.f.* secção 2.3.3).

<sup>3</sup>Segundo uma travessia descendente, da esquerda para a direita.

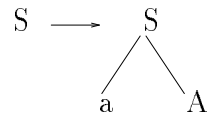
### 3.3.1 Funcionamento Geral

Considere-se a seguinte gramática  $G_1 = (T, N, S, P)$  com  $T = \{a, b, c\}$ ,  $N = \{S, A, B\}$  e

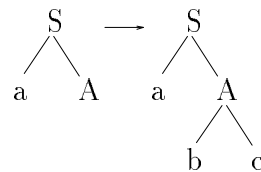
$$P = \left\{ \begin{array}{ll} S \rightarrow aA & (i) \\ S \rightarrow B & (ii) \\ A \rightarrow bc & (iii) \\ A \rightarrow b & (iv) \\ B \rightarrow bc & (v) \end{array} \right\}$$

e a frase  $\mu = abc$ .

O reconhecimento *top-down* desta frase é feito do seguinte modo: cria-se uma árvore com um único nodo associado ao axioma  $S$  e inicia-se o reconhecimento considerando o primeiro símbolo de  $\mu$  que é  $a$ . De seguida, "analisam-se" as produções cujo lado esquerdo é  $S$ , para determinar qual a produção a usar para reconhecer  $a$ . Escolhendo a primeira produção – (i) – a árvore expande-se do seguinte modo:



A folha mais à esquerda da árvore resultante está associada ao símbolo terminal  $a$ , logo corresponde ao símbolo que se pretende reconhecer. Portanto, pode-se avançar um símbolo na frase  $\mu$  (passando para o seu segundo símbolo –  $b$ ) e considerar a folha seguinte, associada ao não terminal  $A$ . Neste momento, existem duas produções pelas quais podemos expandir o símbolo  $A$ , mais concretamente as produções (iii) e (iv). Optando pela primeira hipótese teremos a seguinte árvore:



A folha mais à esquerda está de novo de acordo com o símbolo a reconhecer. Sendo assim, podemos avançar para o próximo símbolo de  $\mu$ , o símbolo

$c$ , e para a folha seguinte. De novo, a folha está associada a um símbolo terminal que corresponde ao símbolo a reconhecer. Portanto, o reconhecimento termina com sucesso pois foi construída uma árvore cuja fronteira é  $\mu$ .

No entanto, esta situação não aconteceria se no reconhecimento do segundo símbolo de  $\mu$  tivéssemos optado pela segunda produção possível, a produção (iv). Neste caso, o segundo símbolo seria *aceite*, mas o mesmo não aconteceria para o símbolo seguinte, o símbolo  $c$ , uma vez que não existia mais nenhuma folha onde isso pudesse ser feito. No reconhecimento do primeiro símbolo também não se conseguiria reconhecer a frase  $\mu$  se se tivesse optado pela produção (ii), pois a partir do símbolo não terminal  $B$  não é possível derivar frases começadas por  $a$ . Nestas duas situações seria necessário recuperar dessas decisões erradas, efectuando *backtracking* e tentando as outras alternativas.

Embora seja possível implementar reconhecedores *top-down* com *backtracking*, de momento este tipo de reconhecedores não serão considerados. Sendo assim, é necessário definir uma classe de gramáticas independentes do contexto mais restritiva, para as quais não existe qualquer ambiguidade na escolha da produção a utilizar durante a derivação de uma frase.

### 3.3.2 Condição LL(1)

Um modo de garantir que nunca surge ambiguidade na escolha da produção a utilizar para derivar uma frase consiste em considerar apenas gramáticas em que todas as produções da forma  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n \in P$  derivam frases cujos símbolos iniciais são todos diferentes. Deste modo, dado um símbolo  $s \in T$  que se pretende reconhecer e o símbolo não terminal  $A$  a expandir, garante-se que existe uma única produção que deriva frases iniciadas por  $s$ . Isto é, a produção correcta é determinada "olhando" unicamente para o *primeiro símbolo*<sup>4</sup> pelo qual as várias alternativas podem derivar, de modo a saber-se antecipadamente por qual das produções seguir.

Porém, a condição de se conhecer o primeiro símbolo pelo qual as várias produções derivam não é suficiente para garantir a inexistência de ambiguidade na escolha da produção. No caso da gramática ter símbolos anuláveis então existem derivações que geram a frase nula, *i.e.*,  $A \Rightarrow^* \epsilon$  (pela própria definição de símbolo anulável, *c.f.* definição 2.10). Nesta situação é necessário

---

<sup>4</sup>Na terminologia inglesa designa-se por *First*.

determinar também quais *os símbolos que se seguem*<sup>5</sup> a derivar por  $A$ , pois um dos símbolos seguintes pode, no caso de se utilizar a derivação que gera a frase nula, ser o início de uma derivação a partir de  $A$ .

As gramáticas independentes do contexto que satisfazem estas duas condições dizem-se  $LL(1)$ , cuja sigla significa:

- $L$  *left-Scan* -> leitura da frase da esquerda para a direita;
- $L$  *leftmost derivation* -> derivação pela esquerda (*cf.* 2.3.3);
- (1) -> um símbolo antecipável.

Formalmente, a condição necessária para uma gramática pertencer à classe  $LL(1)$ , designada *condição  $LL(1)$* , define-se do seguinte modo:

**Definição 3.8** *Uma gramática  $G = (T, N, S, P)$  satisfaz a condição  $LL(1)$  se para toda a produção  $A \rightarrow \alpha_1 | \alpha_2 \dots | \alpha_n \in P$  se verificarem as seguintes condições:*

- Condição 1:

$$\forall_{1 \leq i, j \leq n : i \neq j} \quad \Rightarrow First(\alpha_i) \cap First(\alpha_j) = \emptyset$$

*i.e., o conjunto de símbolos que iniciam frases derivadas de dois lados direitos de produções com o mesmo lado esquerdo devem ser disjuntos.*

- Condição 2:

$$A \Rightarrow^* \epsilon \quad \Rightarrow First_N(A) \cap Follow(A) = \emptyset$$

*i.e., no caso de símbolos anuláveis o conjunto de símbolos que iniciam derivações a partir de  $A$  deve ser disjunto do conjunto de símbolos que se possam seguir a qualquer sequência gerada por  $A$ .*

---

<sup>5</sup>Na terminologia inglesa designa-se por *Follow*.

A função *First* dá como resultado o conjunto de símbolos que iniciam derivações a partir de uma sequência de símbolos terminais e não terminais. A função *First<sub>N</sub>* determina o conjunto de símbolos que iniciam derivações a partir de um símbolo não terminal. Por último, a função *Follow* calcula o conjunto de símbolos que se podem seguir a derivar por um dado símbolo não terminal. Estas funções apresentam-se de seguida.

**Definição 3.9** *A função First de uma sequência de símbolos define-se do seguinte modo:*

$$First : (N \cup T)^* \rightarrow 2^T$$

com *First*( $\alpha$ ) definido por, para  $a \in T$  e  $X \in N$ :

1. Se  $\alpha = a\alpha'$  então  $First(\alpha) = \{a\}$ ;
2. Se  $\alpha = A\alpha' \wedge A \rightarrow \beta_1 \mid \cdots \mid \beta_n \wedge \forall_i \beta_i \not\Rightarrow^* \epsilon$  então

$$First(\alpha) = \bigcup_{1 \leq i \leq n} First(\beta_i)$$

3. Se  $\alpha = A\alpha' \wedge A \rightarrow \beta_1 \mid \cdots \mid \beta_n \wedge \exists_i \beta_i \Rightarrow^* \epsilon$  então

$$First(\alpha) = First(\alpha') \cup \bigcup_{1 \leq i \leq n} First(\beta_i)$$

Exemplo: Considere a seguinte gramática  $G = (T, N, S, P)$  com  $T = \{a, b, c\}$ ,  $N = \{S, A, B\}$  e

$$P = \left\{ \begin{array}{l} S \rightarrow aBa \mid BAc \mid ABc \\ A \rightarrow aA \mid \epsilon \\ B \rightarrow ba \mid c \end{array} \right\}$$

O cálculo do *First* dos lados direitos das produções do axioma de  $G$  será:

- $First(aBa) = \{a\}$  pela primeira regra da definição de *First*;
- $First(BAc) = First(ba) \cup First(c)$ , uma vez que  $B \not\Rightarrow^* \epsilon$ . Logo,  $First(BAc) = \{b, c\}$ ;



- $First(ABc) = First(aA) \cup First(\epsilon) \cup First(ba) \cup First(c)$ , uma vez que  $A \Rightarrow \epsilon$ . Sendo assim,  $First(ABc) = \{a, b, c\}$ .

□

**Exercício 3.7** Calcule os restantes *First* dos lados direitos das produções da gramática  $G$ .

**Definição 3.10** A função  $First_N$  de um símbolo não terminal define-se como:

$$\begin{aligned} First_N : & \quad N \rightarrow 2^T \\ First_N(X) : & \quad \bigcup_{(X \rightarrow \alpha_i) \in P} First(\alpha_i) \end{aligned}$$

Esta função aplicada a um símbolo não terminal dá como resultado o conjunto de símbolos terminais que são inícios válidos de derivações a partir desse não terminal.

Exemplo: Considere a gramática  $G$  do exemplo anterior, então o cálculo dos  $First_N$  dos seus símbolos não terminais será:

- $First_N(S) = First(aBa) \cup First(BAc) \cup First(ABc) = \{a, b, c\}$ ;
- $First_N(A) = First(aA) \cup First(\epsilon) = \{a\}$ ;
- $First_N(B) = First(ba) \cup First(c) = \{b, c\}$ .

□

**Definição 3.11** A função *Follow* de um símbolo não terminal define-se do seguinte modo:

$$\begin{aligned} Follow : & \quad N \rightarrow 2^T \\ Follow(X) : & \quad \bigcup_{(Y \rightarrow \alpha X \beta) \in P} (First(\beta) \cup \begin{cases} \emptyset & \text{se } \beta \not\Rightarrow^* \epsilon \\ Follow(Y) & \text{se } \beta \Rightarrow^* \epsilon \end{cases}) \end{aligned}$$

A função *Follow* de um símbolo não terminal calcula o conjunto de símbolos terminais que se podem seguir a uma derivação a partir do símbolo não terminal. Considere-se a seguinte derivação:

$$S \Rightarrow AB\underline{c} \Rightarrow A\underline{b}cc$$

Sendo assim,  $c \subseteq \text{Follow}(B)$ , pois aparece após  $B$  na primeira derivação. Logo, será o início da derivação após se derivar por  $B$ . De igual, modo  $b \subseteq \text{Follow}(A)$ , uma vez que aparece a seguir a  $A$  (segunda derivação).

Exemplo: Considerando de novo a gramática  $G$ , então o cálculo dos *Follow* dos seus símbolos não terminais será:

- $\text{Follow}(S) = \emptyset$ ;
- $\text{Follow}(A) = \text{First}(c) \cup \text{First}(Bc) \cup (\text{First}(\epsilon) \cup \text{Follow}(A)) = \{c\} \cup \{b, c\} = \{b, c\}$ ;
- $\text{Follow}(B) = \text{First}(a) \cup \text{First}(Ac) \cup \text{First}(c) = \{a\} \cup \{a, c\} \cup \{c\} = \{a, c\}$ .

□

**Exercício 3.8** Considere a seguinte gramática  $G = (T, N, S, P)$ , em que

$$\begin{array}{ll} T = \{a, b\} & P = \{ \quad X \rightarrow Yy \mid xZ \\ N = \{X, Y, Z, W\} & \quad Y \rightarrow W \mid Yy \\ S = X & \quad Z \rightarrow YW \mid xy \\ & \quad W \rightarrow w \mid \epsilon \\ & \quad \} \end{array}$$

Determine os conjuntos  $\text{First}_N$  e  $\text{Follow}$  dos seus símbolos não terminais.

De seguida analisa-se um exemplo em que se verifica se uma gramática satisfaz ou não a condição LL(1).

Exemplo: Considere a seguinte gramática  $G_2 = (T, N, S, P)$  com  $T = \{a, b, c, d, e\}$ ,  $N = \{S, A, B, C\}$  e

$$P = \left\{ \begin{array}{ll} S \rightarrow bAb & (i) \\ S \rightarrow aBb & (ii) \\ A \rightarrow aA & (iii) \\ A \rightarrow \epsilon & (iv) \\ B \rightarrow cC & (v) \\ B \rightarrow d & (vi) \\ C \rightarrow e & (vii) \end{array} \right\}$$

Pretende-se verificar se  $G_2$  satisfaz a condição LL(1).

Isto verifica-se se todas as produções de  $P$  satisfizerem as duas condições referidas na definição 3.8. Começando pelas produções cujo lado esquerdo é o símbolo  $S$ , têm-se de provar as duas condições. A *condição 1* verifica-se se:

$$First(bAb) \cap First(aBb) = \emptyset$$

Pela definição 3.9 têm-se  $First(bAb) = \{b\}$  e  $First(aBb) = \{a\}$  (utilizando, em ambos os casos, a regra 1 da definição). Como  $\{b\} \cap \{a\} = \emptyset$  a primeira condição é satisfeita. A *condição 2* também o é, uma vez que  $S$  não é um símbolo anulável, pois  $S \not\Rightarrow \epsilon$ .

Para as produções cujo lado esquerdo é o símbolo  $A$  tem de se verificar se:

$$First(aA) \cap First(\epsilon) = \emptyset$$

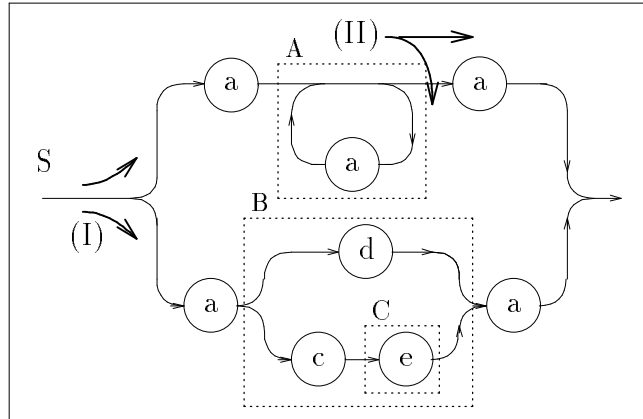
Como  $First(aA) = \{a\}$  e  $First(\epsilon) = \emptyset$  esta condição verifica-se. Neste caso vai ter também de se verificar a *condição 2*, uma vez que  $A$  é um símbolo anulável. Assim, é necessário calcular  $First_N(A)$  e  $Follow(A)$ . Utilizando a definição 3.10 têm-se  $First_N(A) = First(aA) \cup First(\epsilon) = \{a\}$ . Pela definição 3.11 vem  $Follow(A) = First(b) \cup (First(\epsilon) \cup Follow(A)) = \{b\}$ . Como  $\{a\} \cap \{b\} = \emptyset$  então a *condição 2* também se verifica. Para as produções com o símbolo  $B$  no lado esquerdo as duas condições verificam-se, pois  $(First(cC) = \{c\}) \cap (First(d) = \{d\}) = \emptyset$  e  $B$  não é anulável. Para as produções com o símbolo  $C$  no lado esquerdo as duas condições também se verificam, pois só existe uma produção com esse lado esquerdo (logo a *condição 1* verifica-se) e o símbolo  $C$  também não é anulável. Portanto, pode-se

concluir que  $G_2$  satisfaz a condição LL(1).

□

Porém, nem todas as gramáticas satisfazem a condição LL(1), uma vez que as gramáticas que obedecem à condição LL(1) são um subconjunto das GIC. Considere agora a gramática  $G_3$ , que se obtém a partir de  $G_2$  substituindo apenas o símbolo terminal  $b$  pelo símbolo  $a$ . As únicas produções alteradas são a (i) e a (ii) que passam a ser  $S \rightarrow aAa$  e  $S \rightarrow aBa$ , respectivamente. É fácil verificar que  $G_3$  não satisfaz a condição LL(1), pois a *condição 1* é logo violada pelas produções com  $S$  como lado esquerdo. Isto acontece porque  $First(aAa) = \{a\}$  e  $First(aBa) = \{a\}$ , e como  $\{a\} \cap \{a\} = \{a\} \neq \emptyset$ , logo a *condição 1* não é satisfeita. No entanto, esta não é a única situação em que a condição LL(1) é violada. Repare-se que para as produções com o símbolo  $A$  no lado esquerdo têm-se:  $First_N(A) = \{a\}$  e  $Follow(A) = First(a) \cup (First(\epsilon) \cup Follow(A)) = \{a\}$ , como  $\{a\} \cap \{a\} = \{a\} \neq \emptyset$ , logo a *condição 2* também não se verifica.

Estas duas situações constata-se facilmente analisando o grafo de sintaxe da gramática  $G_3$  que se apresenta a seguir.



Neste grafo estão sinalizadas as duas situações – (I) e (II) –, referidas anteriormente, em que a condição LL(1) não é satisfeita. Como se verifica, essas situações surgem quando existe ambiguidade na escolha do caminho/produção a seguir/usar, mais concretamente, quando numa bifurcação não é possível decidir qual o caminho a seguir, olhando para o início das várias alternativas (*i.e.*, para o símbolo que as iniciam). Na situação (I) vê-se

perfeitamente o conflito resultante da violação da *condição 1* (da condição LL(1)). Por sua vez, na situação (II) constata-se a violação da *condição 2*.

Como foi referido anteriormente, no reconhecimento *top-down* não pode existir ambiguidade na escolha da produção a utilizar em cada passo da derivação de uma frase. A *condição LL(1)* garante que as GIC que a satisfazem não têm essa ambiguidade, e portanto, admitem um reconhecimento *top-down*. Um outro modo de garantir que uma gramática não possui ambiguidade na escolha da produção a utilizar, consiste no seguinte: sempre que se vai expandir um símbolo não terminal  $A$ , tenta-se *olhar para a frente*<sup>6</sup>, de modo a determinar quais os símbolos terminais que iniciam as várias alternativas. Deste modo, para não existir ambiguidade na escolha do lado direito, pelo qual  $A$  vai ser expandido, é necessário que esses conjuntos sejam disjuntos.

Formalmente, a função que permite "olhar para a frente", designada *Lookahead*, define-se do seguinte modo:

**Definição 3.12** A função *Lookahead* de uma produção define-se como:

$$\begin{aligned} \text{Lookahead} : & \quad P \rightarrow 2^T \\ \text{Lookahead}(A \rightarrow \alpha) : & \quad \text{First}(\alpha) \cup \begin{cases} \emptyset & \text{se } \alpha \not\Rightarrow^* \epsilon \\ \text{Follow}(A) & \text{se } \alpha \Rightarrow^* \epsilon \end{cases} \end{aligned}$$

Exemplo: Considerando de novo a gramática  $G$  (página 72) o cálculo do *Lookahead* das produções com o símbolo  $A$  do lado esquerdo será:

- $\text{Lookahead}(A \rightarrow aA) = \text{First}(aA) \cup \emptyset = \{a\};$
- $\text{Lookahead}(A \rightarrow \epsilon) = \text{First}(\epsilon) \cup \text{Follow}(A) = \{a, b, c\}.$

□

**Exercício 3.9** Determine o conjunto *Lookahead* das produções da gramática do exercício 3.8.

A condição LL(1) pode ser redefinida utilizando para tal a função *Lookahead*. Esta definição apresenta-se de seguida.

---

<sup>6</sup>Na terminologia inglesa designa-se por *Lookahead*.

**Definição 3.13** Uma gramática  $G = (T, N, S, P)$  satisfaz a condição LL(1) se

$$\forall_{A \rightarrow \alpha_1, A \rightarrow \alpha_2 \in P} : Lookahead(A \rightarrow \alpha_1) \cap Lookahead(A \rightarrow \alpha_2) = \emptyset$$

Considerando de novo a gramática  $G_2$  é fácil provar, utilizando a definição 3.13, que esta gramática satisfaz a condição LL(1) (tal como foi feito anteriormente usando a definição 3.8). Calculando os *Lookaheads* das várias produções têm-se:

$$\begin{aligned} Lookahead(S \rightarrow bAb) &= First(bAb) \cup \emptyset = \{b\} \\ Lookahead(S \rightarrow aBb) &= First(aBb) \cup \emptyset = \{a\} \\ Lookahead(A \rightarrow aA) &= First(aA) \cup \emptyset = \{a\} \\ Lookahead(A \rightarrow \epsilon) &= First(\epsilon) \cup Follow(A) = \{b\} \\ Lookahead(B \rightarrow cC) &= First(cC) \cup \emptyset = \{c\} \\ Lookahead(B \rightarrow d) &= First(d) \cup \emptyset = \{d\} \\ Lookahead(C \rightarrow e) &= First(e) \cup \emptyset = \{e\} \end{aligned}$$

Como  $\{b\} \cap \{a\} = \emptyset$ ,  $\{a\} \cap \{b\} = \emptyset$  e  $\{c\} \cap \{d\} = \emptyset$ , então  $G_2$  satisfaz a condição LL(1).

**Exercício 3.10** Considere a seguinte gramática  $G = (T, N, S, P)$  com  $T = \{x, y, z\}$ ,  $N = \{S, X, Y\}$  e

$$P = \left\{ \begin{array}{l|l} S \rightarrow zX & zY \\ X \rightarrow Xx & x \\ Y \rightarrow yY & \epsilon \end{array} \right\}$$

Prove que  $G$  não satisfaz a condição LL(1) e diga quais os seus conflitos LL(1).

**Exercício 3.11** Considere a seguinte gramática  $G = (T, N, S, P)$  com  $T = \{1, 2, 3\}$ ,  $N = \{S, A, B\}$  e

$$P = \left\{ \begin{array}{l|l} S \rightarrow A1 & B \\ A \rightarrow A1 & \epsilon \\ B \rightarrow 23 & 1 \end{array} \right\}$$

Mostre que  $G$  é uma gramática ambígua e que não satisfaz a condição LL(1).

**Exercício 3.12** Considere duas gramáticas  $G = (T, N, S, P)$  e  $G' = (T', N', S, P')$  que satisfazem a condição  $LL(1)$ . Em que condições é que a gramática

$$G'' = (T \cup T', N \cup N', S, P \cup P')$$

satisfaz também a condição  $LL(1)$ ? Justifique convenientemente.

Como foi referido anteriormente, nem todas as gramáticas satisfazem a condição  $LL(1)$  (veja-se o caso da gramática  $G_3$ ). Para essas gramáticas não é possível construir um reconhecedor *top-down*. Porém, é sempre possível alterar as suas produções, de modo a que a gramática resultante não possua conflitos  $LL(1)$  e defina a mesma linguagem. Na secção seguinte apresentam-se as transformações habituais para gramáticas que não satisfazem a condição  $LL(1)$ .

### 3.3.3 Transformações Essenciais à Condição $LL(1)$

Existem várias situações típicas onde se verificam conflitos  $LL(1)$ , nomeadamente em:

1. Gramáticas ambíguas;
2. Gramáticas recursivas à esquerda;
3. Produções com o mesmo lado esquerdo e cujos lados direitos têm o mesmo prefixo.

Para cada uma destas situações existem transformações que podem ser feitas nas produções da gramática de modo a que a condição  $LL(1)$  seja satisfeita. Basicamente, o que se pretende é a partir de uma gramática  $G$ , que possui conflitos  $LL(1)$ , obter-se, por transformação das suas produções, uma gramática  $G'$ , tal que  $G'$  satisfaz a condição a  $LL(1)$  e  $\mathcal{L}_{G'} = \mathcal{L}_G$ .

De seguida analisa-se detalhadamente cada uma das situações referidas anteriormente.

- *Gramáticas ambíguas*: Estas gramáticas possuem conflitos  $LL(1)$ , uma vez que nessas gramáticas existe mais que uma sequência de derivação possível para uma mesma frase (*c.f.* definição 2.14). Sendo assim, existe

então pelo menos um símbolo não terminal para o qual existem duas produções alternativas para prosseguir o reconhecimento da frase.

Um modo de resolver este conflito consiste em reescrever a gramática de modo a que a gramática resultante não seja ambígua. Isto pode ser feito tendo em conta as prioridades e a associatividade dos operadores (ver secção 2.3.3).

- *Gramáticas recursivas à esquerda:* As gramáticas recursivas à esquerda possuem conflitos LL(1). Considere-se a seguinte produção recursiva à esquerda:

$$A \rightarrow A\alpha \mid \beta$$

Calculando o  $First_N(A)$  têm-se:

$$First_N(A) = First(A\alpha) \cup First(\beta)$$

Então existe um conflito LL(1), uma vez que

$$Lookahead(A \rightarrow A\alpha) \cap Lookahead(A \rightarrow \beta) \supseteq First(\beta)$$

A solução para este conflito consiste em eliminar a recursividade à esquerda, reescrevendo a gramática com recursividade à direita. Assim, a produção anterior é transformada em

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Esta transformação prova-se facilmente recorrendo à álgebra de expressões regulares (*c.f.* secção 2.2.1).

$$\begin{aligned} A \rightarrow A\alpha \mid \beta &\Leftrightarrow A = A\alpha + \beta \\ &\Leftrightarrow A = \beta\alpha^* && \text{regra 13} \\ &\Leftrightarrow A = (\beta\alpha^*)\epsilon && \text{regra 6} \\ &\Leftrightarrow A = \beta(\underbrace{\alpha^*\epsilon}_{A'}) && \text{regra 5} \\ &\Leftrightarrow \begin{cases} A = \beta A' \\ A' = \epsilon + \alpha A' \end{cases} && \text{regra 12} \\ &\Leftrightarrow \begin{cases} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \epsilon \end{cases} \end{aligned}$$



- *Produções com o mesmo lado esquerdo e cujos lados direitos têm o mesmo prefixo:*

Considere-se a seguinte produção:

$$A \rightarrow \alpha\beta \mid \alpha\gamma$$

Existe um conflito LL(1), pois

$$Lookahead(A \rightarrow \alpha\beta) \cap Lookahead(A \rightarrow \alpha\gamma) \supseteq First(\alpha)$$

A solução consiste em efectuar uma factorização à esquerda, *i.e.*, colocar o símbolo que causa o conflito em "evidência". A produção é transformada em:

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta \mid \gamma \end{aligned}$$

Esta transformação prova-se também através da álgebra de expressões regulares.

$$\begin{aligned} A \rightarrow \alpha\beta \mid \alpha\gamma &\Leftrightarrow A = \alpha\beta + \alpha\gamma \\ &\Leftrightarrow A = \alpha(\underbrace{\beta + \gamma}_{A'}) \quad \text{regra 7} \\ &\Leftrightarrow \begin{cases} A = \alpha A' \\ A' = \beta + \gamma \end{cases} \\ &\Leftrightarrow \begin{cases} A \rightarrow \alpha A' \\ A' \rightarrow \beta + \gamma \end{cases} \end{aligned}$$

Considere a gramática  $G$  do exercício 3.10. Nesse exercício provou-se que  $G$  não satisfazia a condição LL(1). Sendo assim, é necessário transformar as suas produções de modo a eliminar os conflitos LL(1). Facilmente se constata que os conflitos resultam de dois factores: as produções com o símbolo  $S$  como lado esquerdo têm o mesmo prefixo (o símbolo  $z$ ) e existe uma produção recursiva à esquerda. Efectuando-se as respectivas transformações (descritas anteriormente) obtém-se a seguinte gramática  $G' = (T, N, S, P)$  com  $T = \{x, y, z\}$ ,  $N = \{S, S', X, X', Y\}$  e

$$P = \left\{ \begin{array}{l|l} S \rightarrow zS' & Y \\ S' \rightarrow X & \\ X \rightarrow xX' & \\ X' \rightarrow xX' & \epsilon \\ Y \rightarrow yY & \epsilon \end{array} \right\}$$

A prova de que  $G'$  satisfaz a condição LL(1) é deixada como exercício.

A transformação das gramáticas de modo a satisfazerem a condição LL(1), embora seja uma técnica necessária para implementar um reconhecedor *top-down*, tem várias desvantagens, como sejam:

- A gramática "aumenta", uma vez que se introduzem novos símbolos não terminais;
- O reconhecimento de uma frase é feito através de uma sequência de derivação com um maior numero de passos;
- A árvore de derivação que se obtém, durante o reconhecimento de uma frase, não representa fielmente a sua estrutura sintática.

No entanto, os reconhecedores *top-down* têm a grande vantagem de a sua implementação ser bastante simples, sendo por isso muito utilizados na prática para construir reconhecedores "manualmente", *i.e.*, sem recurso a um gerador automático de *parsers*. Um exemplo de um reconhecedor desenvolvido segundo esta estratégia é o reconhecedor da linguagem C, existente no compilador *lcc*<sup>7</sup> [FH95].

A implementação de um reconhecedor *top-down* pode ser feita utilizando duas estratégias distintas:

- *Reconhecedores recursivos descendentes*, e
- *Reconhecedores dirigidos por uma tabela*.

Estas duas estratégias vão ser analisadas em detalhe nas duas secções seguintes.

---

<sup>7</sup>O *lcc* é um compilador de ANSI C, de domínio público, desenvolvido na Universidade de Princeton a nos Laboratórios AT&T. O *lcc* gera código para os processadores SPARC, MIPS R3000 e INTEL 386 e seus sucessores.

### 3.3.4 Reconhecedor Recursivo Descendente

Um *Reconhecedor Recursivo Descendente* — RRD — de uma linguagem  $\mathcal{L}$ , definida pela gramática  $G = (T, N, S, P)$ , é construído definindo-se um procedimento para cada símbolo  $X \in N$ . Assim, definem-se vários "sub-reconhecedores", um para cada uma das classes sintáticas de  $G$ . O reconhecedor de  $G$  obtém-se combinando os vários sub-reconhecedores definidos.

A definição dos sub-reconhecedores pode ser feita usando duas estratégias diferentes:

1. Escrever os procedimentos de reconhecimento com base nos grafos de sintaxe da linguagem [Wir76, Oli91, WC93];
2. Escrever os procedimentos de reconhecimento com base nas produções da gramática [AB90].

A primeira estratégia utiliza os grafos de sintaxe para "guiar" a construção do RRD, enquanto a segunda utiliza as produções da gramática. De seguida analisa-se cada uma destas estratégias.

#### Construção baseada no grafo de sintaxe

A transformação de um grafo de sintaxe num reconhecedor de uma linguagem é feita segundo um conjunto de regras que irão ser apresentadas a seguir. Porém, esta transformação só é possível para grafos de sintaxe que não contêm ambiguidade na escolha do caminho a seguir durante o reconhecimento de uma frase (*i.e.*, cuja gramática equivalente satisfaz a condição LL(1)).

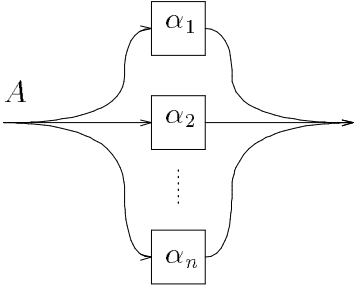
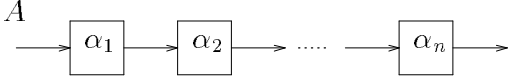
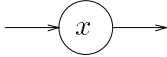
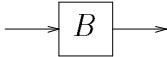
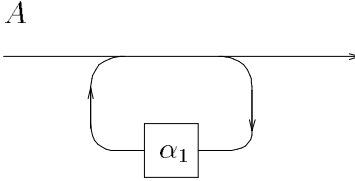
A transformação de um grafo num programa é feita considerando os vários subgrafos básicos<sup>8</sup> que ele contém e associando "pedaços de código" a cada um desses subgrafos, de acordo com um conjunto de regras. Posteriormente, o programa obtém-se compondo esses "pedaços de código".

Antes de se definirem as regras de transformação, vai-se assumir (por uma questão de simplicidade) o seguinte: os símbolos que constituem a frase a reconhecer são caracteres individuais. Assume-se a existência de uma variável global  $simb : T$  que conterà o próximo símbolo a reconhecer. Avançar para o símbolo seguinte da frase é feito através da função  $ler()$ . Por último, o "pedaço de código" que se obtém a partir do grafo  $g$  denota-se por  $T(g)$ .

---

<sup>8</sup>Estes subgrafos básicos são os subgrafos que foram definidos na secção 2.3.2.

Regras para a transformação de grafos de sintaxe em programas:

Grafo de Sintaxe	Código
	<pre> <b>se</b>   <math>simb \in First(\alpha_1) \longrightarrow \{T(\alpha_1) \square</math>        <math>simb \in First(\alpha_2) \longrightarrow \{T(\alpha_2) \square</math>        ...        <math>simb \in First(\alpha_n) \longrightarrow \{T(\alpha_n)</math>        <b>senao</b> <math>\longrightarrow \{erro</math> <b>fse</b> </pre>
	$\left\{ \begin{array}{l} T(\alpha_1); \\ T(\alpha_2); \\ \dots \\ T(\alpha_n); \end{array} \right.$
	<pre> <b>se</b>   <math>simb = x \longrightarrow \{simb \leftarrow ler(); \square</math>        <b>senao</b> <math>\longrightarrow \{erro</math> <b>fse</b> </pre>
	$B;$
	<pre> <b>enq</b>   <math>simb \in First(\alpha_1)</math>         <math>\longrightarrow \{T(\alpha_1)</math> <b>fenq</b> </pre>

De notar que se  $First(\alpha_i)$  consiste apenas num único símbolo, então  $simb \in First(\alpha_i)$  pode ser expresso por  $simb = First(\alpha_i)$ .

Exemplo: Considere o grafo de sintaxe  $g_1$ , que define a linguagem  $\mathcal{L}_1$ , apresentado na secção 2.3.2 (página 23). Aplicando as regras de transformação

definidas anteriormente, obtém-se o seguinte procedimento para o reconhecimento de  $\mathcal{L}_1$ .

**reconhece**  $\overline{S} \equiv$

$$\left\{ \begin{array}{l} \text{se } simb = a \\ \rightarrow \left\{ \begin{array}{l} \text{se } simb = a \rightarrow \{ simb \leftarrow ler(); \square \\ \text{senao} \rightarrow \{ erro \\ \text{fse} \\ S; \end{array} \right. \quad \square \\ \text{se } simb = b \rightarrow \{ simb \leftarrow ler(); \square \\ \text{senao} \rightarrow \{ erro \\ \text{fse} \\ simb \in \{b, c\} \\ \rightarrow \left\{ \begin{array}{l} \text{enq } simb \in \{b\} \\ \rightarrow \left\{ \begin{array}{l} \text{se } simb = b \rightarrow \{ simb \leftarrow ler(); \square \\ \text{senao} \rightarrow \{ erro \\ \text{fse} \end{array} \right. \quad \square \\ \text{fenq} \\ \text{se } simb = c \rightarrow \{ simb \leftarrow ler(); \square \\ \text{senao} \rightarrow \{ erro \\ \text{fse} \\ \text{senão} \rightarrow \{ erro \\ \text{fse} \end{array} \right. \end{array} \right.$$

O programa que efectua o reconhecimento de  $\mathcal{L}_1$  será ainda constituído por uma função para o tratamento de erros – a função *erro()* –, e da função principal que neste caso apenas lê o primeiro símbolo da frase e invoca o procedimento **reconhece**  $\overline{S}$ .

□

Os procedimentos que se obtêm utilizando as regras de transformação anteriores podem ser simplificados utilizando regras de programação óbvias. Uma estrutura que aparece frequentemente no código obtido é:

$$\begin{array}{l} \text{se } simb = x \\ \rightarrow \left\{ \begin{array}{l} \text{se } simb = x \rightarrow \{ simb \leftarrow ler(); \square \\ \text{senao} \rightarrow \{ erro \\ \text{fse} \end{array} \right. \\ \dots \\ \text{fse} \end{array}$$

Esta estrutura pode ser simplificada do seguinte modo:

```

se   simb = x
       $\longrightarrow \{ \textit{simb} \leftarrow \textit{ler}(); \square$ 
      ...
fse

```

### Construção baseada nas produções da gramática

A construção de um reconhecedor recursivo descendente com base nas produções da gramática  $G = (T, N, S, P)$ , que define a sintaxe da linguagem a reconhecer, é feito do seguinte modo:

Para cada símbolo  $A \in N$  com produções da forma

$$A \rightarrow \alpha_1\alpha_2 \cdots \alpha_n \mid \beta_1\beta_2 \cdots \beta_n \mid \cdots \mid \gamma_1\gamma_2 \cdots \gamma_n \in P$$

associa-se o seguinte procedimento de reconhecimento:

```

reconhece A  $\equiv$ 
{
  se   simb  $\in \textit{Lookahead}(A \rightarrow \alpha_1\alpha_2 \cdots \alpha_n)$ 
       $\longrightarrow \left\{ \begin{array}{l} \textit{reconhece\_}\alpha_1; \\ \textit{reconhece\_}\alpha_2; \\ \dots \\ \textit{reconhece\_}\alpha_n; \end{array} \right. \square$ 
  simb  $\in \textit{Lookahead}(A \rightarrow \beta_1\beta_2 \cdots \beta_n)$ 
       $\longrightarrow \{ \dots \square$ 
  simb  $\in \textit{Lookahead}(A \rightarrow \gamma_1\gamma_2 \cdots \gamma_n)$ 
       $\longrightarrow \left\{ \begin{array}{l} \textit{reconhece\_}\gamma_1; \\ \textit{reconhece\_}\gamma_2; \\ \dots \\ \textit{reconhece\_}\gamma_n; \end{array} \right. \square$ 
  senão  $\longrightarrow \{ \textit{erro}$ 
fse

```

O procedimento de reconhecimento associado aos símbolos terminais consiste unicamente em verificar se o símbolo da frase a reconhecer corresponde ao símbolo pelo qual se pretende derivar. Nesse caso, avança-se a entrada

para o símbolo seguinte da frase. Assim, pode-se considerar um único procedimento de reconhecimento para os símbolos terminais, que recebe como argumento o símbolo pelo qual se pretende derivar. Esse procedimento será:

$$\text{reconhece\_Term } (t: T) \equiv \left\{ \begin{array}{l} \text{se } t = \text{simb} \\ \quad \longrightarrow \{ \text{simb} \leftarrow \text{Ler}(); \square \\ \text{senão} \longrightarrow \{ \text{erro} \\ \text{fse} \end{array} \right.$$

Note-se que tal como na construção de um RRD a partir dos grafos de sintaxe, também nesta estratégia se assume a existência da variável global  $\text{simb} : T$  e da função  $\text{ler}()$ , que têm exactamente as mesmas funções.

De seguida apresenta-se um exemplo da utilização desta estratégia para a construção de um RRD.

Exemplo: Considere-se de novo a linguagem  $\mathcal{L}_1$  (definida pela gramática  $G_1$  na secção 2.3.2), para a qual se desenvolveu um RRD na secção anterior. Efectuando-se agora a construção do RRD com base nas produções de  $G_1$  obtém-se os dois procedimentos de reconhecimento (um para cada símbolo não terminal) que se seguem.

$$\text{reconhece\_S} \equiv \left\{ \begin{array}{l} \text{se } \text{simb} \in \{a\} \\ \quad \longrightarrow \left\{ \begin{array}{l} \text{reconhece\_Term}('a'); \\ \text{reconhece\_S}; \end{array} \right. \square \\ \text{simb} \in \{b, c\} \\ \quad \longrightarrow \left\{ \begin{array}{l} \text{reconhece\_B}; \\ \text{reconhece\_Term}('c'); \end{array} \right. \square \\ \text{senão} \longrightarrow \{ \text{erro} \\ \text{fse} \end{array} \right.$$

$$\text{reconhece\_B} \equiv \left\{ \begin{array}{l} \text{se } \text{simb} \in \{b\} \\ \quad \rightarrow \left\{ \begin{array}{l} \text{reconhece\_Term('b')}; \quad \square \\ \text{reconhece\_B}; \end{array} \right. \\ \text{simb} \in \{c\} \\ \quad \rightarrow \left\{ \begin{array}{l} \square \\ \text{senão} \rightarrow \{erro\} \end{array} \right. \\ \text{fse} \end{array} \right.$$

□

Este último procedimento tem uma forma particular de recursividade, que se designa na literatura inglesa por *tail recursion* (recursividade na cauda). Este tipo de recursividade traduz o comportamento dos ciclos nas linguagens imperativas, podendo desse modo ser implementada de uma forma mais eficiente, i.e., através de um ciclo (ver [Val93]).

Efectuando-se esta alteração e mais algumas substituições o programa resultante seria semelhante ao obtido na secção anterior. Estas alterações são deixadas como exercício.

Os reconhecedores recursivos descendentes são bastante simples de desenvolver. No entanto, o código destes reconhecedores está bastante dependente da gramática da linguagem. Isto origina que o tamanho de código estático seja geralmente grande. Uma outra característica óbvia destes reconhecedores é que são muito recursivos, podendo eventualmente esta recursividade traduzir-se em ineficiência.

A solução para estes problemas consiste em armazenar a informação gramatical numa estrutura de dados exterior aos procedimentos de reconhecimento. Na secção seguinte apresentam-se os reconhecedores dirigidos por tabela, que utilizam esta técnica.

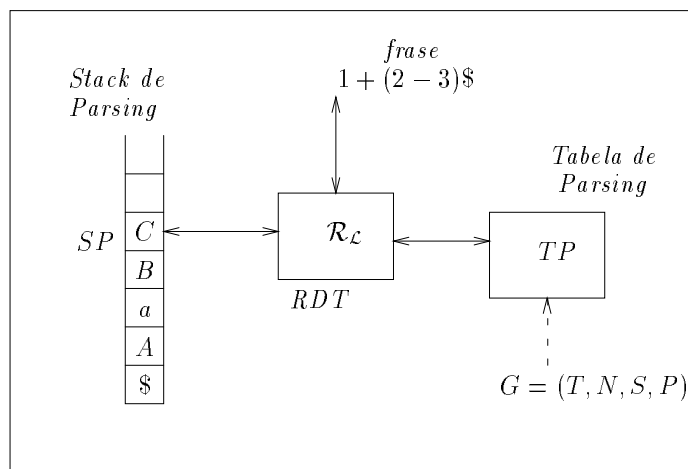
### 3.3.5 Reconhecedor Dirigido por Tabela

Um *Reconhecedor Dirigido por Tabela* — RDT — é uma versão não recursiva de um reconhecedor *top-down*. Nestes reconhecedores o estado do reconhecimento é mantido numa *stack*, chamada *stack de parsing*, explícita ao programa, contrariamente aos RRD's cuja *stack* está implícita nas chamadas recursivas das funções.



A informação gramatical da linguagem é codificada numa estrutura de dados, designada *tabela de parsing*, que vai indicar ao reconhecedor qual a acção de reconhecimento a tomar. Em particular, vai indicar qual a produção a ser aplicada na derivação ou, caso contrário, se ocorreu uma situação de erro.

Um reconhecedor dirigido por tabela vai então ser constituído por: a frase a reconhecer, uma *stack de parsing*, uma tabela de *parsing* e o procedimento reconhecedor. A estrutura conceptual de um RDT apresenta-se na figura seguinte.



Nestes reconhecedores a frase a processar vai ser terminada por um carácter especial – '\$' – que indica que não existem mais símbolos a processar. Este carácter é necessário para o procedimento de reconhecimento "saber" parar, como se verá mais à frente.

A *stack de parsing* contém uma sequência de símbolos da gramática que descrevem o estado do reconhecimento. Esta estrutura de dados será do tipo:

$$SP = (N \cup T)^*$$

Uma vez que os reconhecedores *top-down* iniciam o reconhecimento partindo da raiz da árvore de derivação, então a *stack* deve ser inicializada com o axioma da gramática. Porém, para o algoritmo parar é necessário também colocar o carácter '\$' no fundo da *stack*, ficando, deste modo, o axioma por cima.

A tabela de *parsing* indica ao RDT qual a produção a usar no reconhecimento do proximo símbolo da frase. Esta tabela indica, dado um símbolo da frase a reconhecer e um símbolo não terminal a reescrever, qual a produção a usar. Esta estrutura de dados será então um *array* bidimensional, em que as linhas estão associadas aos símbolos não terminais da gramática e as colunas aos símbolos terminais. A tabela de *parsing* é do tipo:

$$TP = \underline{array}[N \times T]_{de} P \cup \{erro\}$$

O algoritmo para a construção da tabela é o seguinte:

**função** *Inicializa* $TP(\alpha : TP) : TP$

$$\left\{ \begin{array}{l} \alpha' \leftarrow \alpha; \\ \textbf{para} \quad n \in N, t \in T \\ \quad \longrightarrow \{\alpha'[n, t] \leftarrow erro; \\ \textbf{fpara} \\ \textbf{para} \quad (A \rightarrow \beta) \in P \\ \quad \textbf{para} \quad t \in Lookahead(A \rightarrow \beta) \\ \quad \quad \longrightarrow \{\alpha'[A, t] \leftarrow (A \rightarrow \beta); \\ \quad \textbf{fpara} \\ \textbf{fpara} \\ \quad Inicializa TP \leftarrow \alpha'; \end{array} \right.$$

Como em qualquer reconhecedor *top-down* a gramática tem de satisfazer a condição LL(1).

Nestes reconhecedores, a gramática da linguagem apenas influencia a construção da tabela de *parsing* e não o algoritmo de reconhecimento como nos RRD's. O procedimento de reconhecimento é um procedimento genérico que baseado na informação dessa tabela efectua o reconhecimento da linguagem.

O algoritmo do reconhecedor dirigido por tabela vai consistir num ciclo que terminará quando se atingir uma situação de reconhecimento ou de erro. A primeira situação surgirá quando na *stack* de *parsing* existir apenas o símbolo '\$' (que indica que não falta reconhecer mais nada) e não existirem mais símbolos da frase de entrada a reconhecer.

Exemplo: Considere a gramática  $G_2 = (T, N, S, P)$ , apresentada na página 74 e que se pretende efectuar o reconhecimento da frase  $\mu = baab$  utilizando um RDT.

O primeiro passo consiste na construção da tabela de *parsing*. Utilizando o algoritmo anterior obtém-se a seguinte tabela  $\alpha : SP$ :

	$a$	$b$	$c$	$d$	$e$
$S$	$S \leftarrow aBb$	$S \leftarrow bAb$			
$A$	$A \leftarrow aA$	$A \leftarrow \epsilon$			
$B$			$B \leftarrow c$	$B \leftarrow d$	
$C$					$C \leftarrow e$

Note-se que os *Lookaheads* das várias produções foram calculados anteriormente (ver página 3.3.2).

Inicializando a *stack* de *parsing* com o axioma da gramática e o caracter '\$' têm-se a seguinte configuração inicial:

<i>stack de parsing</i>	entrada
$\$S$	$baab\$$

Consultado a tabela de *parsing* com o símbolo que se encontra no topo da *stack* e o símbolo da frase a reconhecer, têm-se  $\alpha[S, b] = S \leftarrow bAb$ . Portanto, a acção a executar corresponde a derivar por essa produção, deste modo o símbolo  $S$  vai ser substituído pelo lado direito da produção.

<i>stack de parsing</i>	entrada
$\$bAb$	$baab\$$

Nesta situação o símbolo da frase corresponde ao símbolo que se encontra no topo da *stack*, então esse símbolo é reconhecido. Nesta situação tira-se o símbolo do topo da *stack* e passa-se para o reconhecimento do símbolo seguinte da frase.

<i>stack de parsing</i>	entrada
$\$bA$	$aab\$$

Nesta situação têm-se  $\alpha[A, a] = A \leftarrow aA$ , e obtém-se o seguinte estado de reconhecimento.

<i>stack de parsing</i>	entrada
$\$bAa$	$aab\$$

Prosseguindo,

<i>stack de parsing</i>	entrada
\$bA	ab\$
\$bAa	ab\$
\$bA	b\$

Neste ponto têm-se  $\alpha[A, b] = A \leftarrow \epsilon$ , e obtém-se

<i>stack de parsing</i>	entrada
\$b	b\$
\$	\$

O reconhecimento termina neste ponto com sucesso, pois não existem mais símbolos da frase a reconhecer e a *stack* contém apenas o símbolo '\$'.

□

O algoritmo de um reconhecedor *top-down* dirigido por tabela apresenta-se de seguida.

**função**  $RDT(\alpha : TP, \beta : SP) : bool$

$$\left\{ \begin{array}{l}
 Erro \leftarrow Falso; \\
 simb \leftarrow ler(); \\
 \beta \leftarrow Push(S, <>); \\
 \mathbf{rep} \\
 \left\{ \begin{array}{l}
 t \leftarrow Top(\beta); \\
 \mathbf{se} \quad t \in N \\
 \rightarrow \left\{ \begin{array}{l}
 p = \alpha(t, simb); \\
 \mathbf{se} \quad p = erro \rightarrow \{ Erro \leftarrow Verdade; \\
 \mathbf{senão} \rightarrow \left\{ \begin{array}{l} \beta \leftarrow Pop(\beta) \\ \beta \leftarrow Aplica(Ldp(p), \beta); \end{array} \right. \\
 \mathbf{fse}
 \end{array} \right. \\
 t \in T \vee t = \$ \\
 \rightarrow \left\{ \begin{array}{l}
 \mathbf{se} \quad t = simb \\
 \rightarrow \left\{ \begin{array}{l} \beta \leftarrow Pop(\beta); \\ simb \leftarrow ler(); \end{array} \right. \\
 \mathbf{senão} \rightarrow \{ Erro \leftarrow Verdade; \\
 \mathbf{fse}
 \end{array} \right. \\
 \mathbf{fse}
 \end{array} \right. \\
 \mathbf{até} \quad t = \$ \vee Erro = Verdade \\
 RDT \leftarrow (t = \$ \wedge Erro = Falso)
 \end{array} \right.$$

com  $t : (N \cup T)$ ,  $simb : T$  e  $p : P \cup \{erro\}$ . A função  $ler()$  devolve o próximo símbolo a reconhecer.

Considera-se ainda a existência das seguintes funções:

- $Ldp : (N \cup T)^+ \times (N \cup T)^* \rightarrow (N \cup T)^*$   
 $Ldp(A \rightarrow \alpha) = \alpha$ , devolve o lado direito  $\alpha$  da produção;
- $Push : (N \cup T) \times SP \rightarrow SP$   
 $Push(x, s) = \langle x.s \rangle$ , i.e., coloca  $x$  à cabeça de  $s$ ;
- $Pop : SP \rightarrow SP$   
 $Pop(s) = tail(s)$ , i.e., retira o primeiro elemento (a cabeça) de  $s$ ;
- $Top : SP \rightarrow (N \cup T)$   
 $Top(s) = head(s)$ , i.e., devolve o primeiro elemento (a cabeça) de  $s$ ;
- $Aplica : (N \cup T)^* \times SP \rightarrow SP$   
 $Aplica(p, s) = ps$ , i.e., coloca à cabeça de  $s$  os elementos de  $p$ .

**Exercício 3.13** Considere a seguinte gramática  $G = (T, N, S, P)$ , que define expressões numéricas LISP muito simples, em que

$$\begin{aligned} T &= \{., (, ), +, *, int\} & P &= \{ \begin{array}{l} S \rightarrow Expr'.' \\ Exp \rightarrow int \mid '(' Funcao ')' \\ Funcao \rightarrow '+' Lista \mid '*' Lista \\ Lista \rightarrow Exp Lista \mid \epsilon \end{array} \} \\ N &= \{S, Expr, Funcao, Lista\} \end{aligned}$$

1. Calcule o conjunto dos Firsts, Follows e Lookaheads;
2. Prove que  $G$  satisfaz a condição  $LL(1)$ ;
3. Represente  $G$  utilizando Grafos de Sintaxe e escreva os procedimentos de reconhecimento que lhes estão associados.
4. Construa a tabela de Parsing para a gramática  $G$ ;
5. Verifique, utilizando o algoritmo dirigido por tabela, se  $\mu = (+ 3 (* 1 3)). \in \mathcal{L}_G$

**Exercício 3.14** Verifique se as seguintes gramáticas (definidas unicamente pelas suas produções) satisfazem a condição LL(1):

1.  $P = \{ \begin{array}{l} S \rightarrow aA \mid bcB \\ A \rightarrow bA \mid C \mid ac \\ B \rightarrow cC \mid d \\ C \rightarrow c \end{array} \}$
2.  $P = \{ \begin{array}{l} S \rightarrow aA \\ A \rightarrow bBc \mid cBd \mid ac \\ B \rightarrow a \mid \epsilon \end{array} \}$
3.  $P = \{ \begin{array}{ll} Frase & \rightarrow Palavra \text{ sep } Frase \mid Palavra \\ & \mid Int \text{ sep } Frase \mid Int \\ Palavra & \rightarrow \text{car } Palavra \mid \text{car} \\ Int & \rightarrow Int \text{ dig } \mid \text{dig} \end{array} \}$

Caso alguma das gramáticas não satisfaça a condição LL(1) retire os conflitos e prove que a gramática resultante satisfaz a condição referida.

**Exercício 3.15** Dada a seguinte gramática (definida unicamente pelas suas produções):

$$\begin{array}{l} S \rightarrow a \mid bA \mid cA \\ A \rightarrow aS \mid bS \mid cB \\ B \rightarrow aB \mid bS \end{array}$$

1. Verifique se a gramática satisfaz a condição LL(1);
2. Construa o Grafo de Sintaxe e escreva os procedimentos de reconhecimento associados ao Grafo de Sintaxe;
3. Construa a Tabela de Parsing;
4. Verifique, utilizando o algoritmo dirigido por tabela, se a frase  $\mu = \text{babcaaba}$  pertence à linguagem definida pela gramática.

### 3.4 Reconhedores de Linguagens Não-Regulares Bottom-Up

Como foi referido anteriormente, os reconhecedores *top-down* aplicam-se unicamente a gramáticas independentes do contexto que satisfazem a condição LL(1). Esta limitação dos reconhecedores *top-down* condiciona a sua

aplicação, uma vez que vários contrutores das linguagem de programação não podem ser definidos utilizando esta estratégia.

Esta limitação é ultrapassada pelos *reconhecedores bottom-up* (*reconhecedores ascendentes*) que se aplicam virtualmente a qualquer gramática independente do contexto. Estes reconhecedores são bastante poderosos e podem ser aplicados a qualquer construtor de uma linguagem de programação. No entanto, estes reconhecedores são muito trabalhosos, tornando-se pouco prático serem implementados manualmente. Sendo assim, na prática utilizam-se ferramentas geradoras de analisadores sintáticos, que produzem automaticamente estes reconhecedores.

Nesta secção descreve-se o funcionamento dos reconhecedores *bottom-up*, analisam-se técnicas para a sua construção e apresenta-se o seu algoritmo de reconhecimento.

### 3.4.1 Reconhecedor Desloca/Reduz

Fundamentalmente, este método difere do anterior no modo como a árvore de derivação é construída. Nos reconhecedores *bottom-up* a árvore de derivação, de uma frase  $f$ , é construída começando pelas folhas (*bottom*) e subindo até à raiz  $S$  (*top*).

$$f \Rightarrow^* S$$

Este processo pode ser visto como a *redução* da frase  $f$  até ao axioma da gramática  $\rightarrow S$ . Esta redução é feita por reduções sucessivas, uma vez que em cada passo uma dada subfrase de  $f$ , que coincida com o lado direito de uma produção, é substituída pelo não terminal do lado esquerdo da produção.

Considere-se a seguinte gramática  $G = (T, N, S, P)$  com

$$\begin{array}{lcl} T = \{a, b, c\} & P = & \{ \quad S \rightarrow ABc \mid B \\ N = \{S, A\} & & \quad A \rightarrow Aa \mid a \\ & & \quad B \rightarrow b \\ & & \} \end{array}$$

Então a frase  $f = aabc$  pode ser *reduzida* a  $S$  através dos seguintes passos de redução:

- $aabc$  (1)
- $Aabc$  (2)
- $Abc$  (3)
- $ABc$  (4)
- $S$  (5)

Isto é feito percorrendo a frase  $f$  até se encontrar uma subfrase que coincide com o lado direito de alguma produção. No primeiro passo isso verifica-se com a subfrase  $a$  e  $b$ , considerando o símbolo mais à esquerda  $a$  ele é substituído por  $A$ , o lado esquerdo da produção  $A \rightarrow a$ . Após esta substituição obtém-se a frase  $Aabc$ . Esta frase possui três subfrases —  $Aa, a$  e  $b$  — que estão de acordo com o lado direito das produções. Considerando a substituição de  $Aa$  obtém-se a frase  $Abc$  (passo 3) que, por sua vez (passo 4), pode ser reduzida a  $ABc$  pela produção  $B \rightarrow b$ . Finalmente, esta frase pode ser reduzida a  $S$  pela produção  $S \rightarrow ABc$ .

Repare-se que a derivação correspondente aos vários passos de redução, *i.e.*,

$$S \Rightarrow ABc \Rightarrow Abc \Rightarrow Aabc \Rightarrow aabc$$

é uma derivação pela direita (*c.f.* definição 2.9), pois em cada passo é o não terminal mais à direita que é substituído (veja-se a derivação  $\dots ABc \Rightarrow Abc \dots$ ). No entanto, a derivação da frase  $f$  não foi feita como está aqui representado, mas sim em sentido contrário. Deste modo, o reconhecimento *bottom-up* corresponde à inversão de uma derivação pela direita. Daqui resulta que estes reconhecedores se designem *reconhecedores LR* — RLR —, cuja sigla significa:

- L *left-Scan* -> leitura da frase da esquerda para a direita;
- R *rightmost derivation* -> derivação pela direita;

No entanto, existem reduções que produzem frases que posteriormente não podem ser reduzidas a  $S$ . Por exemplo, esta situação surgiria se no passo 2 (da redução de  $f = aabc$  a  $S$ ) se opta-se por reduzir a sub-frase  $a$  a  $A$  (pela produção  $A \rightarrow a$ ). Nesse caso obtinha-se a frase  $AAbc$  que não se consegue reduzir a  $S$ .



É óbvio que as reduções que interessa considerar são as que conduzem a uma redução ao símbolo  $S$ . Assim, designa-se por *redex* de uma frase a uma sua subfrase que está de acordo com o lado direito de uma produção, e cuja redução, ao não terminal do lado esquerdo dessa produção, é possível encadear numa derivação pela direita.

Utilizando-se *redexes* nas sucessivas reduções garante-se que se efectua sempre uma redução pela produção correcta (não surgindo nunca a situação descrita anteriormente). Deste modo, o reconhecedor irá percorrer  $f$  até encontrar um *redex*, altura em que se irá efectuar uma redução.

Um modo eficiente e adequado de implementar um reconhecedor *bottom-up* consiste em utilizar uma *stack* —*stack de parsing*— que terá informação sobre o estado do reconhecimento (tal como nos reconhecedores *top-down* dirigidos por tabela). Nesta *stack* são armazenados os símbolos da gramática.

Utilizando a *stack de parsing* o reconhecedor procederá do seguinte modo: irá deslocando símbolos de  $f$  para a *stack* até encontrar um *redex* no seu topo<sup>9</sup>. Nessa situação efectua-se uma redução pela produção respectiva. O reconhecedor repete estas acções até detectar um erro ou até já ter deslocado todos os símbolos de  $f$  para a *stack* e esta conter unicamente o símbolo inicial. Neste último caso a frase é aceite.

Considerando de novo a redução da frase  $f = aabc$  a  $S$ , a sequência de acções que o reconhecedor executa, para reconhecer  $f$  é dada pela seguinte tabela:

<i>stack</i>	$f$	acção
	$aabc$	<i>desloca</i>
$a$	$abc$	<i>reduz por <math>A \rightarrow a</math></i>
$A$	$abc$	<i>desloca</i>
$Aa$	$bc$	<i>reduz por <math>A \rightarrow Aa</math></i>
$A$	$bc$	<i>desloca</i>
$Ab$	$c$	<i>reduz por <math>B \rightarrow b</math></i>
$AB$	$c$	<i>desloca</i>
$ABc$		<i>reduz por <math>S \rightarrow ABc</math></i>
$S$		<i>aceita</i>

<sup>9</sup>Este é um aspecto que justifica a utilização de uma *stack*, pois assim garante-se que se existir um *redex* ele está no topo da *stack* e não no meio.

Como se pode verificar nesta tabela as acções principais deste reconhecedor são as acções *desloca* e *reduz*, sendo por isso designado *reconhecedor bottom-up tipo desloca-reduz*. No entanto, existem mais duas acções: a de *aceitar* uma frase e a acção de *erro*.

Embora intuitivamente seja fácil determinar quais as acções a executar pelo reconhecedor, de modo a reduzir-se uma frase ao símbolo inicial da gramática, no caso da construção de um reconhecedor é necessário ter uma estrutura de dados adicional para "orientar" as decisões *desloca/reduz*, com a garantia de se efectuar uma derivação pela direita.

Uma possibilidade será representar nessa estrutura de dados um *autómato finito determinístico* (c.f. definição 3.7) cujos estados contêm a informação necessária para o reconhecedor decidir entre as acções *desloca* e *reduz*. Os estados do autómato serão colocados na *stack* de *parsing* de modo a orientar as decisões do reconhecedor, com base em *tabelas de transição de estados*.

O autómato finito determinístico será representado numa tabela, designada *tabela de parsing*, composta por duas partes: uma com as transições de estados e uma outra com as acções a tomar. Estas tabelas podem ser definidas do seguinte modo:

- Tabela de *Transição de estados*

$$TT = \underline{\text{array}}[Q \times (T \cup N)] \underline{\text{de}} Q \cup \{\text{erro}\}$$

- Tabela de *Acções*

$$TA = \underline{\text{array}}[Q \times T] \underline{\text{de}} P \cup \{\text{desloca}, \text{aceita}, \text{erro}\}$$

A construção das tabelas de *parsing* pode ser feita utilizando três metodologias distintas:

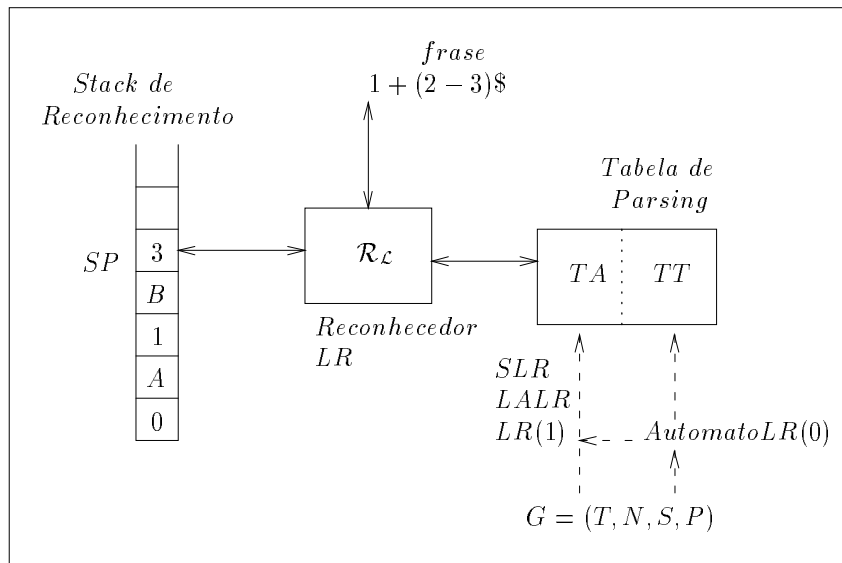
1. *Método SLR (Simple LR)*: Este método é o mais fácil de implementar. No entanto, é o menos poderoso, uma vez que nem sempre é possível construir a tabela de *parsing* para algumas gramáticas (como se verá a seguir).
2. *Método LR "canónico"*: É o método mais poderoso. Porém, é o mais complicado de implementar e é o que necessita de mais memória para armazenar a tabela.

3. *Método LALR (LookAhead LR)*: É um método intermédio entre os outros dois.

Estas metodologias diferem apenas no modo de construção da tabela de acções. Na secção seguinte será analisada a construção das tabelas de *parsing* utilizando o método SLR. Neste texto apenas se analisa este método, devido à sua "simplicidade". Os métodos LALR e LR "canónico" não serão estudados. Porém, podem ser encontrados no livro dragão [ASU86].

**Definição 3.14** *Uma gramática  $G$  diz-se uma gramática SLR (respectivamente LALR, LR) sse fôr possível construir para ela um reconhecedor SLR (respectivamente LALR, LR).*

A estrutura conceptual de um reconhecedor LR apresenta-se na figura seguinte:



### 3.4.2 Construção das Tabelas de Parsing

A ideia central do método LR consiste na construção, a partir da gramática independente do contexto, de um autómato finito determinístico (*c.f.* secção 3.2.1), designado *autómato LR(0)*, cujos estados contêm a informação

necessária para o reconhecedor decidir entre as acções *desloca* e *reduz*. Este autómato é posteriormente representado numa tabela, que define a *tabela de transições* de estados do algoritmo LR. Com base no autómato LR(0) é ainda construída a *tabela de acções*.

Para melhor explicar a construção das tabelas de *parsing*, vai ser analisado um caso concreto. Considere a seguinte gramática  $G_1 = (T, N, S, P)$ , em que  $T = \{a, b\}$ ,  $N = \{S, A\}$  e  $P = \{S \rightarrow Aa, S \rightarrow b, A \rightarrow Aa, A \rightarrow \epsilon\}$ .

**Exercício 3.16** *Prove que a gramática  $G_1$  não satisfaz a condição  $LL(1)$ .*

Tal como na construção de um reconhecedor de uma linguagem regular, a construção do autómato finito determinístico é feita construindo previamente um autómato finito não determinístico, que é posteriormente convertido em determinístico (ver secção 3.2.5).

Antes de se começar a construção do autómato é necessário estender a gramática original com uma nova produção — $p$ , um novo símbolo inicial — $S'$ , e um novo símbolo terminal — $\$$ . Essa produção será da forma:  $S' \rightarrow S\$$  em que  $S$  é o axioma da gramática original. Esta extensão é necessária para o reconhecedor saber quando deve terminar o reconhecimento (tal como nos reconhecedores *top-down* dirigidos por tabela).

Na gramática que estamos a analisar o novo conjunto de produções — $P'$ — será  $P' = P \cup \{S' \rightarrow S\ \$\}$

$$\begin{aligned} S' &\rightarrow S\$ & (I) \\ S &\rightarrow Aa & (II) \\ S &\rightarrow b & (III) \\ A &\rightarrow Aa & (IV) \\ A &\rightarrow \epsilon & (V) \end{aligned}$$

Relembrando a definição de autómato finito, apresentada na secção 3.2.1, têm-se  $\mathcal{A} = (\mathcal{V}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$ , com:

- $\mathcal{V} = N \cup T$ ;
- $\mathcal{Q}$  é um conjunto finito de estados;

- $S \in Q$  é o estado inicial;
- $Z \in Q$  é o estado final;
- $\delta$  é a função de transição de estados.

em que os estados — $Q$ — do autómato representam *situações de reconhecimento*, *i.e.*, indicam o que já foi "visto" de uma produção num dado ponto do processo de reconhecimento.

Nestes autómatos os elementos de  $Q$ , designados por *items*, são *pares*, constituídos por uma produção da gramática e por uma posição. Os elementos de  $Q$  definem-se do seguinte modo:

$$Q = \{(p, n) \in P \times \mathbb{N} \mid p = A \rightarrow \beta \in P \wedge n \leq |\beta| + 1\}$$

Geralmente utiliza-se uma notação mais sugestiva considerando um ponto (indicador) numa dada posição do lado direito da produção em vez do par  $(p, n)$ .

Considerando a produção  $S \rightarrow Aa$  (do exemplo anterior), teremos os seguintes *items*:

$$S \rightarrow .Aa$$

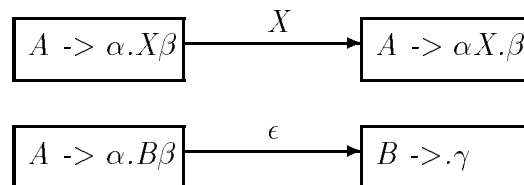
$$S \rightarrow A.a$$

$$S \rightarrow Aa.$$

Informalmente o primeiro *item* indica que se espera encontrar na entrada uma subfrase derivável de  $Aa$ . O segundo *item* indica que se encontrou na entrada uma subfrase derivável de  $A$  e que se espera encontrar de seguida uma subfrase derivável de  $a$ . No último *item* o ponto está no fim da produção e significa que se pode efectuar uma redução.

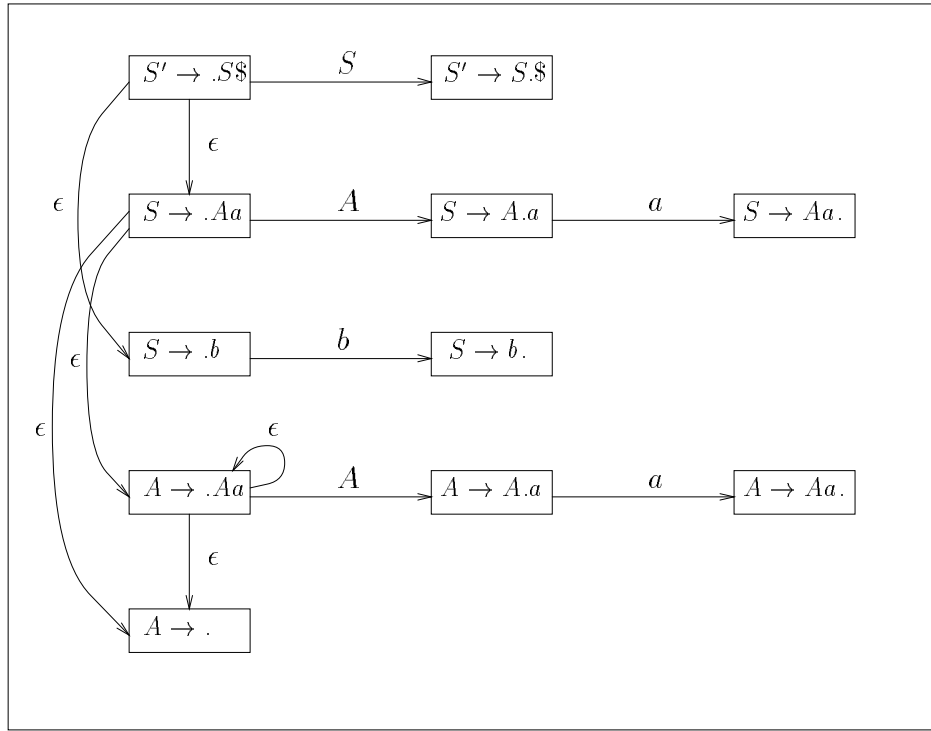
Assim, o estado inicial será constituído pelo *item*  $S = S' \rightarrow .S\$, i.e.$ , é a produção adicionada com o ponto no início do seu lado direito.

A função de transição de estados é  $\delta : (Q \times \mathcal{V}) \rightarrow Q$  em que:



Informalmente estas duas transições entre estados podem ser descritas do seguinte modo: no primeiro caso, num determinado ponto do processo de reconhecimento espera-se reconhecer uma subfrase derivável de  $X\beta$ , se nessa situação aparecer uma subfrase derivável  $X$  então transita-se para um novo estado em que já se "viu"  $X$  e espera-se ver a seguir  $\beta$ . No segundo caso, num dado ponto do *parsing* espera-se encontrar uma sub-frase derivável de  $B\beta$ , então se  $B \rightarrow \gamma$  é uma produção da gramática, pode também esperar ver-se uma sub-frase derivável de  $\gamma$ . Sendo assim, transita-se entre os estados por  $\epsilon$ .

Voltando ao exemplo que se está a considerar, então o autómato finito não determinístico que se obtém aplicando as regras anteriores é o seguinte:



Como se pode verificar na figura cada estado representa uma possível situação do processo de reconhecimento. O estado inicial  $[S' \rightarrow .S\$]$  corresponde á produção adicionada com o ponto no início do lado direito. Como existe um símbolo não terminal a seguir ao ponto —símbolo  $S$ —, então pela

função de transição descrita anteriormente, existem transições  $\epsilon$  para os estados que têm produções em que esse símbolo não terminal pode derivar e o ponto no início do seu lado direito. Esta situação volta a acontecer com o estado  $[S \rightarrow .Aa]$ , e incluem-se novamente no autómato transições  $\epsilon$  para os estados com produções cujo lado esquerdo é o símbolo  $A$  e em que o ponto está no início do lado direito. No estado inicial existe ainda uma transição pelo símbolo  $S$  para um outro estado que contém a mesma produção mas cujo ponto se deslocou para a direita de  $S$ . Esta transição corresponde a aplicar a primeira regra da função de transição definida.

A conversão do autómato não determinístico em determinístico é feita eliminando as transições  $\epsilon$ . Isto é feito calculando o  $\epsilon$ -*fecho* dos estados (ver secção 3.2.4). A função  $\epsilon$ -*fecho* foi definida na secção 3.2.4, no entanto apresenta-se de novo de seguida.

$$\begin{aligned}\epsilon\text{-fecho}: 2^Q &\rightarrow 2^Q \\ \epsilon\text{-fecho}(X) &= X \cup \bigcup_{x \in X} \epsilon\text{-fecho}(\delta(x, \epsilon))\end{aligned}$$

*i.e.*, o fecho de um conjunto de estados  $S$  é o conjunto de estados que se atingem partindo de  $S$  e transitando por  $\epsilon$ .

Considerando de novo o exemplo que está a ser analisado e supondo  $I_0$  o novo estado inicial, então:

$$\begin{aligned}I_0 &= \epsilon\text{-fecho}([S' \rightarrow .S\$]) = \\ &= [S' \rightarrow .S\$, S \rightarrow .Aa, S \rightarrow .b, A \rightarrow .Aa, A \rightarrow \epsilon]\end{aligned}$$

o estado  $I_1$  será:

$$I_1 = \delta'(I_0, S) = \epsilon\text{-fecho}(\bigcup_{q \in I_0} \delta(q, S)) = [S' \rightarrow S.]$$

o estado  $I_2$  é:

$$I_2 = \delta'(I_0, b) = \epsilon\text{-fecho}(\bigcup_{q \in I_0} \delta(q, b)) = [S \rightarrow b.]$$

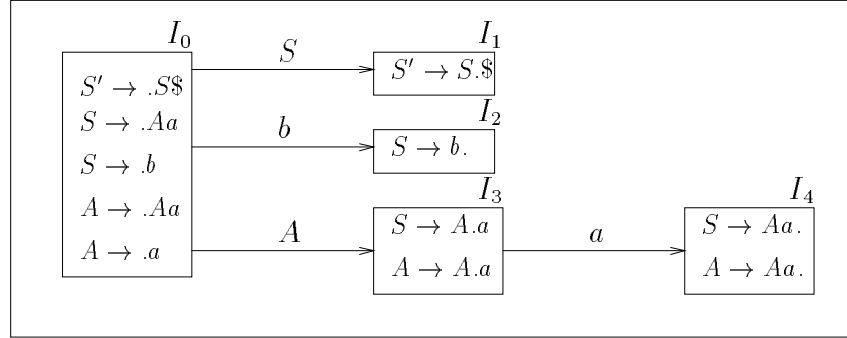
o estado  $I_3$  é:

$$I_3 = \delta'(I_0, A) = \epsilon\text{-fecho}(\bigcup_{q \in I_0} \delta(q, A)) = [S \rightarrow A.a, A \rightarrow A.a]$$

Finalmente, o estado  $I_4$  é:

$$I_4 = \delta'(I_3, a) = \epsilon\text{-fecho}(\bigcup_{q \in I_3} \delta(q, a)) = [S \rightarrow Aa., A \rightarrow Aa.]$$

Obtendo-se o autómato finito determinístico representado na figura seguinte:



Nota: A conversão do autómato não determinístico em determinístico pode ser feita, tal como foi referido na secção 3.2.4, construindo uma tabela em que a primeira coluna associa-se aos estados do autómato determinístico e as seguintes colunas aos vários símbolos da gramática (e não apenas aos símbolos terminais como nas expressões regulares). A tabela é preenchida pondo o  $\epsilon$ -*fecho*( $S$ ) na primeira coluna da primeira linha e nas colunas associadas aos símbolos da gramática colocam-se todos os estados que se alcançam a partir de *fecho* ( $S$ ) transitando por ramos com o peso desse símbolo. Para cada novo estado encontrado procede-se de igual modo. O processo termina quando não se alcançar qualquer estado novo.

□

Com a construção deste autómato determinístico obtém-se a *tabela de transições* de estados, necessário ao algoritmo LR. No entanto, tal como foi referido, neste algoritmo é ainda necessária uma *tabela de acções* que define qual a acção a executar durante o reconhecimento.

### Construção das Tabelas SLR

No método SLR a *tabela de acções* é construída com base nas seguintes regras, supondo  $ta \in TA$ :

- $A \rightarrow \alpha.a\beta \in I_i \Rightarrow ta[i, a] = desloca$



- $A \rightarrow \alpha. \in I_i \Rightarrow \forall_{a \in \text{Follow}(A)}. ta[i, a] = A \rightarrow \alpha$
- $S' \rightarrow S.\$ \in I_i \Rightarrow ta[i, \$] = Ok$
- Todas as outras entradas da tabela correspondem a terminação sem sucesso, *i.e.*, situações de erro.

A primeira destas regras corresponde à acção *desloca* e a segunda a uma acção de *redução*. No método SLR só são efectuadas reduções quando o símbolo terminal que se segue pertencer às continuções válidas do *item* que se está a reduzir (*i.e.*, pertencer ao *Follow* do símbolo do lado esquerdo do *item*). A terceira regra corresponde à situação de terminação com sucesso e a última a situações de erro.

Considerando de novo o exemplo que está a ser analisado e tendo em conta o autómato determinístico construído e as regras para construção da tabela de acções, a tabela SLR é:

Q	TA			TT					
	T			T $\cup$ N					
	a	b	\$	a	b	\$	S'	S	A
0	<i>v</i>	d			2			1	3
1			Ok						
2			<i>iii</i>						
3	d			4					
4	<i>iv</i>		<i>ii</i>						

Tabela SLR

Onde a entrada *d* significa *desloca* e os números romanos significam reduções pelas produções respectivas (ver conjunto de produções de *G*).

As entradas correspondentes a acções de redução foram determinadas tendo em conta que  $\text{Follow}(S) = \{\$ \}$  e  $\text{Follow}(A) = \{a\}$ . No autómato determinístico existem três estados — $I_0, I_2, I_4$ <sup>10</sup>— com situações de possíveis reduções. No estado  $I_0$  existe a possibilidade de redução pela produção  $A \rightarrow \epsilon$ . Sendo assim, e utilizando a segunda regra para construção da tabela de acções, na linha associada a esse estado e nas colunas dos símbolos que

<sup>10</sup>Na tabela os estados estão representados unicamente pelo seu número.

pertencem ao *Follow* do símbolo do lado esquerdo (neste caso unicamente o símbolo  $a$ ) efectua-se uma redução por essa produção (que é a número  $v$ ). No estado  $I_2$  pode-se efectuar uma redução pela produção  $S \rightarrow b$ , deste modo temos que  $ta[2, \$] = iii$  uma vez que  $\$ \in Follow(S)$ . No estado  $I_4$  existem duas situações de redução: uma pela produção  $S \rightarrow Aa$  e outra por  $A \rightarrow Aa$ , sendo assim na coluna associada ao símbolo  $a$  e na linha deste estado, existe uma entrada de redução por essa produção uma vez que  $a \in Follow(A)$ . No caso da outra produção acontece o mesmo mas para a coluna associada ao símbolo  $\$$ , pois  $\$ \in Follow(S)$ .

A entrada *desloca* surge nos estados  $I_0$  e  $I_3$ , pois é quando o ponto está imediatamente antes de um símbolo terminal. Nessa situação a entrada referente ao estado e ao símbolo terminal respectivo fica com o valor *desloca*. Neste caso, é  $ta[0, b] = d$  e  $ta[3, a] = d$ .

A entrada de terminação com sucesso —*Ok*— surge na linha referente ao estado  $I_1$ , pois é nesse estado que existe o *item*  $S' \rightarrow S.\$$ . Todas as entradas da tabela sem qualquer valor são situações de erro.

### 3.4.3 Algoritmo de Reconhecimento LR

Um reconhecedor LR necessita de várias componentes de modo a efectuar o reconhecimento *bottom-up*. Como foi referido anteriormente, necessita de uma *stack de parsing* que orienta as decisões do reconhecedor. Necessita ainda de uma *tabela de parsing* de modo a decidir entre as acções *desloca/reduz*. Esta tabela, tal como foi dito, é constituída pela tabela de transição de estados e pela tabela de acções. O reconhecedor é ainda constituido por uma entrada, onde se armazena a frase a reconhecer; a saída; e por último o próprio reconhecedor.

Este reconhecedor lê os vários lexemas da entrada, um de cada vez. A entrada é da forma:

$$t_i \ t_{i+1} \ \cdots \ T_n \ \$$$

em que os  $ts$  são os vários lexemas e  $t_i$  é o primeiro lexema que está na entrada para ser reconhecido.

Na *stack de parsing* são armazenadas frases da forma:

$$q_0 \ X_1 \ q_1 \ X_2 \ \cdots \ X_m \ q_m,$$

em que  $X_i$  são símbolos da gramática e  $q_i$  são estados do autómato determinístico. Considera-se ainda que  $q_m$  está no topo da *stack*.

O reconhecedor usa o estado que está no topo da *stack* — $q_m$ — e o lexema a ser reconhecido para indexar a tabela de *parsing*, de modo a decidir qual a acção a tomar. Como foi referido, as acções podem ser: *deslocar*, *reduzir* por uma produção, *aceitar* e *erro*.

Para cada um destes casos o reconhecedor terá de efectuar um determinado número de passos. Considerando  $\alpha \in TA$  a tabela de acções e  $\delta \in TT$  a tabela de transição de estados, teremos:

- Se  $\alpha[q_m, t_i] = \textit{desloca}$  então o reconhecedor efectua um deslocamento, do lexema da entrada  $t_i$  e do estado seguinte, para a *stack*. Este estado é dado indexando a tabela de transições com o estado que estava no topo da *stack* e com o lexema da entrada. Supondo  $\delta[q_m, t_i] = q$ , então a *stack* passará a ser:

$$q_0 \ X_1 \ q_1 \ X_2 \ \cdots \ X_m \ q_m \ t_i \ q$$

e a entrada será:

$$t_{i+1} \ \cdots \ T_n \ \$$$

em que  $t_{i+1}$  é o novo lexema a reconhecer;

- Se  $\alpha[q_m, t_i] = X \rightarrow \beta \in P$  então o reconhecedor efectua uma redução pela produção respectiva. Sendo  $r = |\beta|$  (*i.e.*, o número de símbolos do lado direito da produção), então retiram-se  $2r$  símbolos da *stack* ( $r$  estados e  $r$  símbolos da gramática). Nesta situação fica-se com o estado  $q_{m-r}$  no topo da *stack*. De seguida, põe-se na *stack* o símbolo do lado esquerdo da produção — $X$ — (ao qual foi reduzido o seu lado direito) e o novo estado. Este estado é dado indexando a tabela de transições com o estado que ficou no topo da *stack* — $q_{m-r}$ — e o símbolo inserido no *stack*. Supondo  $\delta[q_{m-r}, X] = q$ , então a *stack* passará a ser:

$$q_0 \ X_1 \ q_1 \ X_2 \ \cdots \ X_{m-r} \ q_{m-r} \ X \ q$$

Numa redução a entrada não é alterada;

- Se  $\alpha[q_m, t_i] = aceita$  então o reconhecedor termina com sucesso;
- Se  $\alpha[q_m, t_i] = erro$  então o reconhecedor termina em erro.

Inicialmente a *stack* tem unicamente o estado inicial do autômato determinístico e a entrada terá a frase a reconhecer terminada com o caracter \$.

Considerando de novo o exemplo que se tem analisado, pretende-se verificar se a frase  $f = aaa \in L_{G_1}$ . Utilizando o algoritmo LR descrito anteriormente, teremos no início:

<i>stack de parsing</i>	entrada
0	aaa\$

Consultado a tabela de acção teremos  $\alpha[0, a] = v$ , *i.e.* uma redução pela produção  $A \rightarrow \epsilon$ . Como  $|\epsilon| = 0$  não se retiram nenhuns elementos da *stack*. O elemento a inserir na *stack* será o  $A$  (lado esquerdo da produção) e o estado é dado por  $\delta[0, A] = 3$ . Assim teremos:

<i>stack de parsing</i>	entrada
0 A 3	aaa\$

Nesta situação teremos um deslocamento, uma vez que  $\alpha[3, a] = desloca$ . O estado a colocar na *stack* é dado por  $\delta[3, a] = 4$ . Assim teremos:

<i>stack de parsing</i>	entrada
0 A 3 a 4	aa\$

Nesta situação teremos uma redução de novo, pois  $\alpha[4, a] = iv$  (produção  $A \rightarrow Aa$ ). Como  $|Aa| = 2$  vão-se retirar  $2 \times 2$  símbolos da *stack*. Assim, a *stack* fica com o estado 0 no seu topo. O símbolo a por na *stack* é o  $A$ . Indexando a tabela de transições, vem  $\delta[0, A] = 3$  e obtém-se:

<i>stack de parsing</i>	entrada
0 A 3	aa\$

Neste ponto efectua-se uma redução pela produção  $S \rightarrow Aa$  e obtém-se

<i>stack de parsing</i>	entrada
0 A 3	aa\$
0 A 3 a 4	a\$
0 A 3	a\$
0 A 3 a 4	\$

O algoritmo de reconhecimento LR apresenta-se de seguida.

```

{
   $simb \leftarrow ler()$ ;
   $\beta \leftarrow \text{Push}(0, < >)$ ;
  rep
  {
     $ac \leftarrow \alpha[\text{Top}(\beta), simb]$ ;
    se  $ac = d$  /* desloca */
     $\rightarrow \left\{ \begin{array}{l} q \leftarrow \delta[\text{Top}(\beta), simb]; \\ simb \leftarrow ler(); \\ \beta \leftarrow \text{Push}(q, \beta); \end{array} \right. \quad \square$ 
     $ac = A \rightarrow \gamma$  /* reduz */
     $\rightarrow \left\{ \begin{array}{l} \text{para } i \in \{1, 2, \dots, 2 \times |\gamma|\} \\ \quad \rightarrow \{\beta \leftarrow \text{Pop}(\beta)\} \\ \text{fpara} \\ q \leftarrow \delta[\text{Top}(\beta), A]; \\ \beta \leftarrow \text{Push}(A, \beta); \\ \beta \leftarrow \text{Push}(q, \beta); \end{array} \right. \quad \square$ 
  }
  fse
  até  $ac \in \{ok, erro\}$ 
   $RLR \leftarrow (ac = ok)$ 
}

```

com  $\beta : SP$ ,  $q : Q$ ,  $simb : T$  e  $ac : P \cup \{d, ok, erro\}$ . A função  $ler()$  devolve o próximo símbolo a reconhecer.

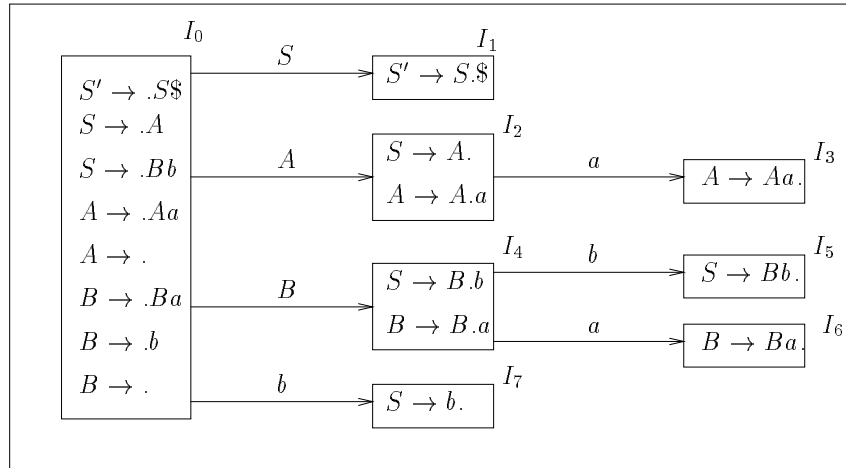
### 3.4.4 Conflitos no Reconhecimento LR

Os reconhecedores LR são bastante poderosos, no entanto, existem gramáticas para as quais não é possível decidir qual *acção* executar. Um reconhecedor do tipo desloca/reduz para essas gramáticas pode atingir um estado em que, baseado na informação existente na *stack* de *parsing* e no símbolo da frase, não consegue decidir se há-de deslocar ou reduzir, ou ainda decidir por qual das várias hipóteses de redução há-de reduzir. O primeiro destes conflitos designa-se *conflito transição/redução* e o segundo *conflito redução/redução*.

Considere-se a seguinte gramática  $G_2 = (T, N, S, P)$ , com

$$\begin{aligned} T &= \{a, b\} & P &= \{ S \rightarrow A \mid Bb \\ N &= \{S, A, B\} & & A \rightarrow Aa \mid \epsilon \\ & & & B \rightarrow Ba \mid b \mid \epsilon \end{aligned}$$

De seguida apresenta-se o autómato LR(0) definido a partir da gramática  $G_2$ . A sua construção é deixada como exercício.



Pretende-se agora construir a tabela de *parsing* para  $G_2$ , utilizando o método SLR. Como foi referido anteriormente, a tabela de transições é obtida

representando o autômato LR(0) numa tabela. Na construção da tabela de acções, segundo o método SLR, verificam-se dois conflitos, logo no primeiro estado do autômato.

Repare-se que  $Follow(S) = \{\$, \}$ ,  $Follow(A) = \{a, \$\}$  e  $Follow(B) = \{a, b\}$ . No estado  $I_0$  do autômato LR(0) existem dois *items* de redução, mais concretamente  $A \rightarrow .$  e  $B \rightarrow ..$  Considerando as regras para a construção das tabelas SLR, então no estado 0 e na coluna associada ao símbolo  $a$  vão existir duas entradas que correspondem a acções de redução. Isto acontece, porque o símbolo  $a$  pertence ao conjunto de *Follows* dos dois símbolos que são o lado esquerdo dos *items* de redução. Neste caso, o reconhecedor não saberá por qual das produções reduzir e, portanto, existe um conflito *redução/redução*.

No entanto, esta não é a única situação de conflito. No estado  $I_0$  do autômato existe o *item*  $B \rightarrow .b$ , então este *item* dá origem a uma acção de transição. Concretamente, na coluna associada ao símbolo  $b$  corresponde uma acção *desloca*. Porém, uma vez que  $b \in Follow(B)$  e existe um *item* de redução com o símbolo  $B$  do lado esquerdo em  $I_0$ , então na coluna associada a  $b$  corresponde também uma acção de redução pela produção  $B \rightarrow ..$  Deste modo, verifica-se um conflito *transição/redução*.

Na tabela seguinte constata-se mais facilmente estes conflitos, que originam que entradas da tabela tenham mais que uma acção.

Q \ TP	TA			TT						
	T			TUN						
	a	b	\$	a	b	\$	S'	S	A	B
0	$A \rightarrow$ $B \rightarrow$	$B \rightarrow$ $d$	$A \rightarrow$		7			1	2	4
1			ok							

Estes conflitos podem ser resolvidos utilizando métodos mais poderosos, nomeadamente o método LALR ou o LR(1).

**Exercício 3.17** Considere a seguinte gramática  $G = (T, N, S, P)$ , em que  $T = \{a, b, c\}$ ,  $N = \{S, A\}$  e  $P = \{S \rightarrow Aa, S \rightarrow b, A \rightarrow aA, A \rightarrow c\}$ . Determine:

1. O Autômato finito não determinístico equivalente e converta-o posteriormente em determinístico;
2. A Tabela de Transição de Estados e a Tabela de Acções.
3. Utilizando um reconhecedor bottom-up prove que a frase  $\mu = aca \in L_G$ .

**Exercício 3.18** Considere a gramática  $G$  do exercício 3.13.

1. Prove que a gramática é SLR.
2. Verifique, utilizando o algoritmo LR, se  $\mu = (+ 3 (* 1 3)). \in L_G$

**Exercício 3.19** Considere a seguinte gramática  $G = (T, N, S, P)$ , em que

$$\begin{array}{ll} T = \{a, b, c\} & P = \{ \quad S \rightarrow Sb \mid bAa \\ N = \{S, A\} & \quad A \rightarrow aSc \mid aSb \mid a \end{array}$$

Verifique se  $G$  tem conflitos SLR.

**Exercício 3.20** Considere a seguinte gramática  $G = (T, N, S, P)$ , em que

$$\begin{array}{ll} T = \{a, b\} & P = \{ \quad S \rightarrow AaAb \mid BbBa \\ N = \{S, A, B\} & \quad A \rightarrow \epsilon \\ & \quad B \rightarrow \epsilon \end{array}$$

1. Prove que  $G$  satisfaz a condição  $LL(1)$ ;
2. Prove que  $G$  não é uma gramática SLR.



## Capítulo 4

# Processamento de Linguagens

### 4.1 Autómatos Finitos com Acções Semânticas

O simples reconhecimento de linguagens regulares não é suficiente para resolver muitos problemas de processamento de linguagens, tal como foi referido anteriormente. Sendo assim, torna-se necessário a execução de acções semânticas durante o reconhecimento de uma linguagem. Considerando a metodologia estudada para construir reconhecedores de linguagens regulares — reconhecedores baseados em AFD's (ver secção 3.2.6) — então, uma forma de introduzir acções semânticas durante o reconhecimento de uma linguagem é através da associação dessas acções às transições do AFD.

Os autómatos finitos extendidos com estas acções semânticas designam-se *autómatos finitos com acções semânticas* ou *autómatos finitos reactivos* — AFR.

Estes autómatos finitos reactivos executam as acções semânticas, associadas às transições do autómato, durante o reconhecimento de uma frase. Deste modo, as transições dos autómatos finitos têm de ser extendidas com a função a executar sempre que se transita por cada transição.

Uma vez que neste texto foram estudados unicamente reconhecedores baseados em autómatos finitos determinísticos, vão-se apenas considerar processadores baseados nesses autómatos.

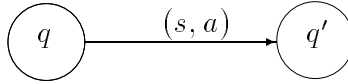
Formalmente os *autómatos finitos determinísticos reactivos* — AFDR —

definem-se do seguinte modo:

**Definição 4.1** *Um autómato finito determinístico reactivo é um autómato finito determinístico  $\mathcal{A} = (\mathcal{T}, \mathcal{Q}, \mathcal{S}, \mathcal{Z}, \delta)$  em que:*

- $\mathcal{T} = \mathcal{I} \cup \mathcal{R}$  é constituído por um conjunto finito não vazio de símbolos do vocabulário  $I$  e por um conjunto finito de acções semânticas  $R$  (ou reacções);
- A tabela de transições  $\delta$  é composta por duas partes: a tabela de transição de estados  $\gamma \subseteq (\mathcal{Q} \times \mathcal{I}) \times \mathcal{Q}$  e a tabela de acções semânticas  $\alpha \subseteq (\mathcal{Q} \times \mathcal{I}) \times \mathcal{R}$ .

Estes autómatos podem também ser representados por um grafo pesado, tal como os autómatos finitos em geral (ver secção 3.2.1). A única diferença para a notação apresentada anteriormente é que nos AFDR os ramos são etiquetados com pares  $(s, a)$ , com  $s \in \mathcal{I}$  e  $a \in \mathcal{R}$ , em vez de o serem por um único símbolo do vocabulário. Sendo assim, a transição  $((q, (s, a)), q') \in \delta$ , com  $q, q' \in \mathcal{Q}$  é representada graficamente do seguinte modo:



Esta transição deve lê-se: "O autómato transita do estado  $q$  para o estado  $q'$  através do reconhecimento do símbolo  $s$  e executando a acção semântica  $a$ ".

De notar que os autómatos finitos apresentados na secção 3.2.1 são casos particulares de autómatos finitos reactivos. Nesses autómatos a função  $f$  executada em cada transição do autómato é uma função vazia, não havendo por isso nenhuma reacção.

Sendo assim, o algoritmo de um processador baseado num autómato finito reactivo é bastante semelhante ao algoritmo de um reconhecedor baseado num AFD (ver secção 3.2.6). A única diferença entre estes algoritmos reside no facto de os primeiros executarem as funções associadas às transições do autómato sempre que se transita de estado, e os outros não.

Uma aplicação bastante frequente das expressões regulares, e consequentemente dos autómatos finitos determinísticos, é o reconhecimento de padrões (ver secção 2.2.2). De seguida analisa-se um problema concreto, em que se utilizam as expressões regulares para definir a linguagem/padrões, e onde se usam AFDR para processar os padrões.

**Problema 2** *Considere-se de novo o problema 1 (página 65) no qual se pretendia desenvolver um reconhecedor de uma linguagem que defina o protocolo de configuração da placa gráfica de um PC. Suponha agora que a placa gráfica necessita, para a sua configuração, de determinar quais os valores, no sistema de numeração decimal, que lhe são enviados pelo PC.*

### Resolução

Para ser possível determinar quais os valores que são enviados à placa gráfica é necessário ir armazenando os vários valores que são transmitidos em binário. Sempre que se recebe o *separador*, isso é sinal que foi enviado um valor e que ele já está armazenado. Portanto é unicamente necessário convertê-lo para a sua representação decimal.

Sendo assim, é óbvio que será necessário definir um autómato finito reactivo para se executarem as acções semânticas necessárias ao longo do reconhecimento de uma frase, a fim de se determinar os valores que ela contém.

Para efectuar este processamento será necessário utilizar um *buffer* onde são colocados os símbolos que definem os valores. Sempre que se transita para um novo estado, guarda-se o símbolo usado na transição no *buffer*. Como foi referido anteriormente, quando se reconhece um separador é porque foi transmitido um valor, e os símbolos que o constituem estão armazenados nesse *buffer*. Sendo assim, considere-se a existência de uma função `processa_valores`, abreviadamente `p_v`, que realiza as tarefas descritas. Vai-se ainda considerar que o *buffer* está implementado utilizando um *array* e é uma estrutura de dados global a todas as funções do processador.

O seu algoritmo é o seguinte:

João A. Saraiva (1995)

No apêndice C apresenta-se a implementação, em linguagem C, deste processador.

□

## 4.2 Analisadores Léxicos

Na secção 2.2 foi referido que as expressões regulares podem ser usadas para definir a sintaxe dos símbolos básico das linguagens de programação. Sendo assim, é possível utilizar expressões regulares para especificar quais as classes de símbolos básicos de uma linguagem e desenvolver um reconhecedor, baseado num AFD. Este reconhecedor deverá identificar sequências de caracteres (num texto) que são instâncias de uma dada classe de símbolos básicos.

Porém, o simples reconhecimento/identificação dos símbolos básicos da linguagem pode não ser suficiente. Considere-se o problema de desenvolver um programa que efectua a *análise léxica* de uma linguagem, *i.e.*, identifica os seus símbolos básicos num texto<sup>1</sup> e *converte-os numa outra representação*. Por *converter os símbolos básicos* entende-se a extracção de informação e a sua representação num formato mais apropriado. Por exemplo, este programa terá o seguinte comportamento: reconhece a sequência de caracteres '125', e então essa sequência é identificada como uma instância de um símbolo básico pertencente à classe **inteiro**. Esta sequência de dígitos identificada deve ser *convertida*, de modo a obter-se a representação do inteiro correspondente, *i.e.*, o número 125. Deste modo, sabe-se que se identificou um **inteiro**, e ainda, qual o valor desse inteiro. No entanto, nem todas as instâncias de classes de símbolos básicos necessitam de ser convertidas numa outra representação. As instâncias de símbolos pertencentes a classes que contêm um único elemento não precisam de ser processadas. Por exemplo, a sequência de caracteres 'while' é identificada como uma instância da classe **while** e esta informação é por si só suficiente.

Em resumo, os *analisadores léxicos* são constituídos por duas tarefas principais:

- *Reconhecimento*: identificação de sequências de caracteres (num texto) que são instâncias de classes de símbolos básicos;

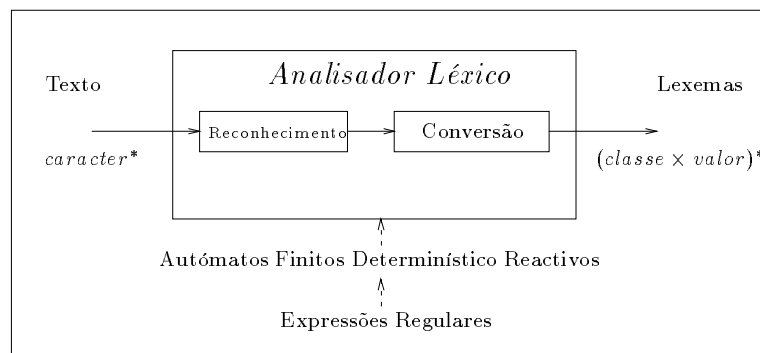
---

<sup>1</sup>Repare-se que um texto é uma sequência de caracteres.

- *Conversão*: obtenção da representação mais conveniente do símbolo reconhecido.

Os símbolos básicos, e a informação que lhes é associada durante a tarefa de *conversão*, designam-se *lexemas* ou *tokens* da linguagem. Os símbolos básicos, em que a identificação da classe é suficiente para os identificar, designam-se *símbolos literais*. As palavras reservadas e os sinais de pontuação de uma linguagem de programação são exemplos de símbolos literais. Os símbolos que necessitam de informação adicional designam-se *símbolos não literais*. Por exemplo, os identificadores e os números de uma linguagem de programação.

A estrutura conceptual de um analisador léxico apresenta-se na figura seguinte:



Para além das tarefas descritas, um analisador léxico tem ainda a função de "ignorar" os *comentários* de uma linguagem de programação e os *espaços* (caracteres **espaço**, **tab** e **newline**) que separam os vários símbolos da linguagem. Isto acontece porque tanto os comentários como os espaços não são relevantes na estrutura sintática de um texto.

Exemplo: Considere o seguinte texto em linguagem C.

```

int inc (i)    /* funcao que incrementa uma variavel */
int i;
{ return i + 1;
}
  
```

Um analisador léxico desta linguagem produzirá a seguinte sequência de lexemas:

classe	valor
INT	tipo_int
ID	"inc"
'('	
ID	"i"
'),'	
INT	tipo_int
ID	"i"
','	
'{'	
RETURN	
ID	"i"
'+'	
CONS_INT	1
','	
','	
'}'	
FIM_DO_TEXTO	

□

Um analisador léxico pode ser facilmente implementado utilizando um autómato finito determinístico reactivo. Neste autómato a acção semântica a executar nas transições do AFDR consistirá no armazenamento dos vários caracteres num *buffer*. Quando se atingir um estado final, então o *buffer* contém a sequência de caracteres que constituem o símbolo básico da linguagem, e podem, caso necessário, ser convertidos para uma outra representação.

De seguida analisa-se o problema de implementar um analisador léxico para uma linguagem concreta.

**Problema 3** *Considere o exercício 3.6. Nesse exercício era proposta a implementação de um reconhecedor para a linguagem definida pela seguinte expressão regular:*

$$sb = 'if' + 'then' + 'while' + If + ('+' + '-' + \epsilon)(0 + \dots + 9)^+$$

*Considere agora que se pretende implementar um analisador léxico para esta linguagem.*

## Resolução

As acções semânticas a executar durante a tarefa de *reconhecimento* de uma frase consistirão no armazenamento num *buffer* dos símbolos reconhecidos. Sendo assim, as funções a executar nas transições do AFDR apenas guardam os símbolos da frase numa estrutura de dados.

Nas transições para estados terminais é possível efectuar (caso necessário) as acções semânticas para realizar a tarefa de *conversão*, uma vez que se garante que uma instância de uma classe de um símbolo básico foi identificada. A sequência de caracteres que constitui essa instância está, como é óbvio, no *buffer* considerado.

Supondo, tal como na resolução do problema 2, a existência do *array buffer*, global a todas as funções do processador da linguagem, então o algoritmo da função executada nas transições para estados não terminais do AFDR é:

$$\begin{array}{l} \textbf{função } f(\gamma : T) : \emptyset \\ \left\{ \begin{array}{l} \textit{buffer}[\textit{ind\_b}] \leftarrow \gamma; \\ \textit{ind\_b} \leftarrow \textit{ind\_b} + 1; \end{array} \right. \end{array}$$

em que *ind\_b* indica a primeira posição livre do *buffer*.

Nas transições para estados terminais a função a executar vai depender da classe do símbolo básico. Considerando a transição para o estado final em que termina o reconhecimento do símbolo *if*, então a função a associar a essa transição será:

$$\begin{array}{l} \textbf{função } f_1(\gamma : T) : \textit{classe} \\ \left\{ \begin{array}{l} \textit{buffer} \leftarrow \langle \rangle; \\ f_1 \leftarrow \textit{if}; \end{array} \right. \end{array}$$

Esta função retira os caracteres que definem o símbolo do *buffer* e devolve como resultado a classe a que essa sequência de caracteres pertence.

Todas as funções —  $f_2$ ,  $f_3$  e  $f_4$  — a associar às transições para estados terminais que completam o reconhecimento dos restantes símbolos literais são semelhantes a esta (ver AFDR apresentado na figura seguinte).

Note-se que nesta linguagem existem dois símbolos básicos que designam a frase *if*, um escrito em letras minúsculas e o outro iniciado por uma letra maiúscula. Se se considerar que a linguagem não é sensível ao tipo de letra (minúscula ou maiúscula) com que se escreve a frase *if*, então a classe destes dois símbolos será a mesma. Assim, a função  $f_4 = f_1$ .



No caso do processamento do símbolo não literal pertencente à classe **inteiro**, os dígitos que constituem este símbolo têm de ser processados. O algoritmo da função de conversão será:

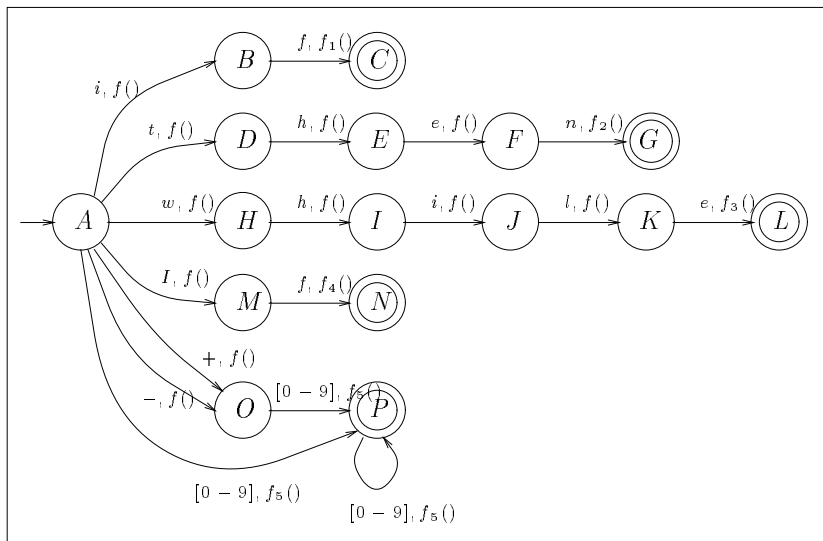
```

função  $f_5(\gamma : T) : \emptyset \cup \text{lexema}$ 
{
  se  $\gamma \neq \epsilon$ 
   $\rightarrow \begin{cases} \text{buffer}[\text{ind\_b}] \leftarrow \gamma; \\ \text{ind\_b} \leftarrow \text{ind\_b} + 1; \\ \text{res} \leftarrow \emptyset \end{cases}$ 
  senão
   $\rightarrow \begin{cases} \text{res.classe} \leftarrow \text{inteiro} \\ \text{res.valor} \leftarrow \text{conv\_alfa\_int}(\text{buffer}) \\ \text{buffer} \leftarrow \langle \rangle; \end{cases}$ 
  fse
   $f_5 \leftarrow \text{res}$ 
}

```

em que a função **conv\_alfa\_int** converte uma sequência de dígitos no seu valor decimal.

Considerando o AFD construído na resolução do exercício 3.6, e as funções apresentadas, então o autómato finito determinístico reactivo para processar a linguagem será:



Repare-se que não está a ser considerado o processamento de **espaços** que obviamente existem a separar os vários símbolos de um hipotético texto desta

linguagem. Como foi referido anteriormente, o analisador deve "ignorar" estes espaços. Isto poderia ser feito criando um laçete no estado inicial do autómato com transição pelo símbolo **espaço**. A função semântica a executar nesta nova transição seria vazia, ignorando assim o espaço reconhecido.

Uma outra característica deste analisador léxico é que após o reconhecimento, ou não, de um símbolo básico, termina o processamento. Sendo assim, para processar os vários símbolos de um texto da linguagem, o processador teria de ser invocado consecutivamente até não existirem mais caracteres na frase a processar. Em cada invocação, o processador devolve o lexema encontrado.

□

**Exercício 4.1** *Implemente em linguagem C o analisador léxico proposto no problema 3.*

#### 4.2.1 Construção de Geradores de Analisadores Léxicos

Utilizando as técnicas estudadas é possível especificar um analisador léxico de uma linguagem e construir o programa respectivo de um modo sistemático. É ainda possível desenvolver ferramentas que, utilizando estas técnicas, produzem automaticamente um analisador léxico a partir de uma especificação. Estas ferramentas designam-se *geradores de analisadores léxicos*.

Nesta secção analisa-se sucintamente como é possível construir uma tal ferramenta. Posteriormente, na secção 4.2.2 apresenta-se uma destas ferramentas.

As classes de símbolos básicos de uma linguagem podem ser especificadas através de um conjunto de expressões regulares. Posteriormente, as expressões regulares são convertidas em autómatos finitos determinísticos de acordo com as técnicas apresentadas na secção 3.2.5. Porém, o simples reconhecimento dos símbolos básicos não é suficiente, sendo ainda necessário especificar as acções semânticas a executar durante esse reconhecimento. As acções semânticas necessárias para implementar um analisador léxico são as seguintes:

- Acções associadas à tarefa de reconhecimento que consistem no armazenamento dos símbolos processados num *buffer*.
- Acções associadas à tarefa de conversão que são específicas de cada classe de símbolos básicos descrita por uma expressão regular. Estas acções manipulam a informação que foi armazenada no *buffer* durante o reconhecimento.

Associando funções que implementam estas acções ao AFD obtém-se um AFDR que pode ser facilmente convertido num programa que efectua a análise léxica.

Repare-se que as acções semânticas necessárias ao reconhecimento dos símbolos são muito simples, e são genéricas para qualquer classe de símbolos. Como se pode verificar no problema 3 estas acções consistem unicamente no armazenamento dos símbolos processados numa estrutura de dados (ver função  $f()$ ). Sendo assim, não é necessário descrever estas acções na linguagem de especificação do analisador léxico e considera-se que elas são executadas durante o reconhecimento. Nas acções de conversão, isto já não se verifica, uma vez que estas acções são específicas de cada classe. Assim, é necessário especificar para cada classe de símbolos básicos qual a acção de conversão.

Um analisador léxico pode então ser especificado por um conjunto de expressões regulares, uma para cada classe de símbolos básicos. A cada expressão regular associa-se a acção semântica necessária à tarefa de conversão. Portanto, a especificação de um analisador léxico será da forma:

$$\begin{array}{ll} e_1 & \{\text{acção}_1\} \\ e_2 & \{\text{acção}_2\} \\ \dots & \dots \\ e_n & \{\text{acção}_n\} \end{array}$$

em que  $e_i$  é uma expressão regular que denota uma classe de símbolos básicos e  $\text{acção}_i$  é a acção semântica que será executada sempre que uma sequência de caracteres (do texto a processar) concordar com  $e_i$ .

Note-se que a acção  $\text{acção}_i$  implementa a tarefa de conversão do analisador léxico. Assume-se que a sequência de caracteres que concordam com a expressão regular  $e_i$  está armazenada numa estrutura de dados global ao processador.

*Exemplo:* Considere de novo o problema 3, no qual se pretendia construir um analisador léxico. Esse analisador léxico pode ser especificado do seguinte modo:

<i>Expressão Regular</i>	<i>Acção semântica de conversão</i>
<i>if</i>	{ res.classe = if }
<i>then</i>	{ res.classe = then }
<i>while</i>	{ res.classe = while }
<i>If</i>	{ res.classe = if }
$( '+' + '-' + \epsilon )( 0 + \dots + 9 )^+$	{ res.classe = inteiro res.valor = conv_alfa_int(buffer) }
$' ' + \backslash t + \backslash n$	{ }

nesta especificação foi já considerada a possibilidade do texto fonte conter **espacos**. Como se verifica a acção semântica associada à expressão regular que denota esses caracteres é nula. Repare-se ainda na diferença de tratamento dos símbolo literais e não literais. Como foi referido, nos primeiro basta unicamente determinar a sua classe (símbolos *if*, *then*, *while* e *If*), enquanto nos símbolos não literais é necessário obter o "valor" do símbolo (símbolo *inteiro*).

□

Usando as técnicas estudadas é possível a partir desta especificação construir automaticamente um programa para efectuar a análise léxica da linguagem. As expressões regulares são transformadas num AFD usando os métodos apresentados na secção 3.2. As transições do AFD são extendidas com as acções semânticas necessárias para armazenar os símbolos reconhecidos no *buffer*. As acções semânticas descritas na especificação (que implementam a tarefa de conversão) associam-se às transições para estados finais do AFD. O AFDR obtido é transformado num programa utilizando o algoritmo da secção 3.2.6 com as (pequenas) alterações descritas na secção 4.1.

Na secção seguinte analisa-se uma ferramenta que com base numa especificação semelhante a esta produz automaticamente um analisador léxico.

## 4.2.2 Gerador de Analisadores Léxicos: *lex*

O *lex* [LMB92] é um gerador de analisadores léxicos que faz parte do sistema operativo UNIX. O *lex* usa expressões regulares para descrever os

símbolos básicos ou lexemas da linguagem. Com base nesta especificação é produzida uma função, em linguagem C, que realiza a análise léxica.

A notação usada pelo `lex` para descrever expressões regulares é a notação usual em UNIX (ver secção 2.2.2). Associado a cada expressão regular é ainda especificado um "pedaço de código" (em linguagem C) que é executado sempre que uma sequência de caracteres concorda com a respectiva expressão regular. Esta linguagem usada para descrever analisadores léxicos é designada uma *especificação do lex*.

O `lex` por si só não produz um programa executável. Porém, transforma uma especificação do `lex` num ficheiro que contém uma rotina C chamada `yylex()`. Um programa (qualquer) pode invocar esta função para efectuar a análise léxica de um texto (de acordo com a especificação que deu origem a essa função). Os ficheiros C produzidos pelo `lex` podem ser compilados tal como qualquer ficheiro C<sup>2</sup>.

Uma especificação do `lex` consiste em três partes distintas separadas pelos caracteres `'%%'`:

```
declarações
%%
regras de conversão
%%
funções auxiliares
```

A secção de *declarações* destina-se à declaração de variáveis, de constantes e de definições regulares. As duas primeiras destinam-se a configurar o ambiente em que o analisador léxico vai operar. As definições regulares são um mecanismo que permite simplificar a definição de expressões regulares longas (ver secção 2.2). Estas definições são usadas posteriormente na secção seguinte. Nesta secção é ainda possível escrever código C puro, que é copiado textualmente pelo `lex` para o analisador léxico produzido. Este código tem de ser delimitado por `%{` e `%}`.

---

<sup>2</sup>É possível transportar esses ficheiros para outros computadores, mesmo que eles não possuam o `lex`, e compilar aí os ficheiros de modo a gerar o programa executável que efectua a análise léxica.

A secção das *regras de conversão* contém um conjunto de descrição de padrões e das respectivas acções de conversão, que definem o analisador léxico. Cada regra é da forma:

$$p \quad \{acção\}$$

em que  $p$  é uma expressão regular e *acção* é um "pedaço de código"  $C$  que descreve a acção a realizar pelo analisador léxico, sempre que uma sequência de caracteres concorda com  $p$ . As expressões regulares são definidas utilizando a notação do UNIX (*c.f.* secção 2.2.2). As expressões regulares podem usar as definições regulares (definidas na secção anterior) envolvendo o seu identificador com chavetas. O **lex** substitui o identificador pela expressão regular definida anteriormente.

O **lex** tenta concordar a maior sequência de caracteres possível com as expressões regulares definidas. Porém, pode surgir a situação de uma sequência de caracteres concordar com mais que uma expressão regular. Neste caso, o **lex** efectua a concordância com a expressão regular definida mais cedo (*i.e.*, numa linha anterior da especificação do **lex**).

Repare que esta linguagem é semelhante à linguagem descrita na secção anterior. Sendo assim, com base numa especificação do **lex** é possível construir um analisador léxico manualmente, utilizando as técnicas descritas nessa mesma secção.

A terceira e última secção permite definir funções  $C$  necessárias para as acções semânticas. Estas funções são copiadas textualmente para o ficheiro produzido pelo **lex**.

Exemplo: Considere que se pretende desenvolver um programa usando o **lex** para contar quantos *números* existem num texto, qual o somatório do comprimento das suas *palavras* e, por último, qual o número de linhas do texto. Suponha que um número é uma sequência de um ou mais dígitos e que uma palavra é uma sequência de uma ou mais letras.

Uma possível especificação do **lex** para implementar este programa apresenta-se a seguir:

```
%{  
  unsigned int contanum = 0 , somacomp , contalin = 0;  
}%
```

```

digito      [0-9]
fdl         \n      /* Fim De Linha */
%%
{digito}+   { contanum++; }
[a-zA-A]+   somacomp += strlen(yytex);
{fdl}       contalin++;
%%
main()
{
    somacomp = 0;
    yylex();
    printf("%d %d %d\n",contanum,somacomp,contalin);
}

```

□

**Exercício 4.2** *Escreva uma especificação do LEX que construa um histograma com o comprimento das palavras de um dado ficheiro de texto. Uma palavra é definida pela seguinte expressão regular:*

$$\text{digito}^* \text{letra} (\text{letra} + \text{digito} + ' - ')^*$$

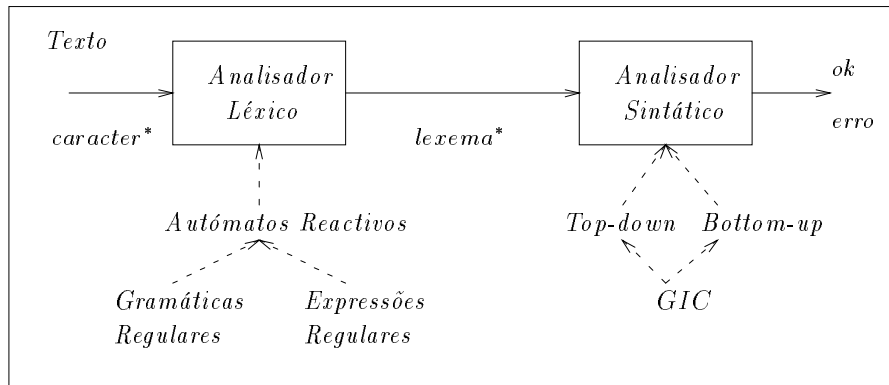
*em que digito representa todos os dígitos e letra representa todas as letras, quer maiúsculas quer minúsculas. O programa a desenvolver não deve considerar as palavras que estão entre parêntesis.*

## 4.3 Tradução Dirigida pela Sintaxe

Os reconhecedores baseados em autómatos reactivos são geralmente usados para o reconhecimento léxico de uma linguagem, sendo a análise sintática feita utilizando métodos mais poderosos, nomeadamente os reconhecedores *top-down* ou *bottom-up*, baseados em gramáticas independentes do contexto (*c.f.* secções 3.3 e 3.4).

Sendo assim, usualmente um reconhecedor é dividido em dois módulos: O primeiro módulo efectua a análise léxica de uma frase, *i.e.*, a partir de uma sequência de caracteres produz os lexemas da linguagem (eliminando espaços e comentários). O segundo módulo recebe esses lexemas e efectua o reconhecimento sintático propriamente dito.

A estrutura conceptual destes reconhecedores apresenta-se na figura seguinte.



Note-se que o analisador léxico e o analisador sintático necessitam passar valores entre eles (os valores dos vários lexemas). Assim, o reconhecedor terá de possuir uma estrutura de dados, partilhada pelos dois módulos, que permita a passagem desses valores. Esta estrutura de dados será do tipo:

$$classe \times valor$$

em que *classe* é o identificador da classe a que o lexema pertence. Usualmente este identificador é um número inteiro. E *valor* é uma estrutura de dados que permite armazenar os diferentes valores de todas as classes de símbolos não literais. Repare-se que um lexema apenas necessita ter associado um único valor, mais concretamente o valor que lhe é associado na tarefa de conversão da analisador léxico.

Exemplo: Considere um analisador léxico que possui vários símbolos literais e duas classes de símbolos não literais. Uma classe que define os números reais e uma outra que define os identificadores. A estrutura de dados (definida em linguagem C) necessária para passar os dados para o analisador sintático será:



```
struct lexemas
{
    int classe
    union valor
    {
        float numero;
        char *ident;
    }
}
```

□

Segundo a arquitectura que está a ser considerada, a análise sintática da linguagem é feita por um reconhecedor *top-down* ou *bottom-up*. É este módulo que controla a tarefa de processamento de uma linguagem. Durante o reconhecimento da linguagem o analisador sintático para aceder aos diferentes lexemas invocará a função que implementa a análise léxica. Esta função devolverá o próximo lexema que deverá ser reconhecido pelo analisador sintático<sup>3</sup>.

### 4.3.1 Gramática Tradutora

No reconhecimento de linguagens não regulares, tal como no reconhecimento de linguagens regulares, é necessário executar acções semânticas ao longo do reconhecimento sintático. Estas acções semânticas podem consistir na construção da árvore sintática (que define a estrutura sintática da frase reconhecida), na execução de acções que verificam se a frase está semanticamente correcta ou ainda em acções de tradução da frase reconhecida numa nova frase de uma outra linguagem (geralmente designada por *linguagem destino*). Esta tarefa que à medida que se efectua o reconhecimento sintático executa as acções semânticas designa-se por *tradução dirigida pela sintaxe* — TDS.

Sendo assim, é necessário estender as gramáticas independentes do contexto com acções semânticas, obtendo-se o que se designa por *gramática tradutora*, abreviadamente GT.

**Definição 4.2** *Uma gramática tradutora é uma gramática independente do contexto  $G = (T, N, S, P)$  em que  $T = I \cup A$  é constituído por um con-*

---

<sup>3</sup>Repare na função construída na resolução do problema 3. Essa função sempre que é invocada devolve um novo lexema encontrado.

junto não vazio de símbolos (do vocabulário)  $I$  e por um conjunto de acções semânticas  $A$ . Estes dois conjuntos são disjuntos, i.e.,  $I \cap A = \emptyset$ .

Repare que se  $A = \emptyset$  então  $G$  não possui nenhuma acção semântica e, portanto, um processador da linguagem definida por  $G$  efectua apenas o seu reconhecimento sintático.

As gramáticas tradutoras podem ser utilizadas em tarefas complexas como, por exemplo, no desenvolvimento de um tradutor entre duas linguagens. Porém, é possível utilizar GT's em tarefas bastante mais simples. De seguida apresenta-se um exemplo bastante simples de uma GT.

Exemplo: Considere a seguinte gramática tradutora  $G_t = (T, N, S, P)$ , em que

$$\begin{aligned} T &= I \cup A \\ I &= \{', ', \text{numero}, \text{palavra}\} \\ A &= \{\underline{\text{inicio}}, \underline{\text{fim}}, \underline{\text{soma}}\} \\ N &= \{S, \text{Lista}, \text{Elem}\} \\ P &= \left\{ \begin{array}{l} S \rightarrow \underline{\text{inicio}} \text{ Lista } \underline{\text{fim}} \\ \text{Lista} \rightarrow \text{Elem} \mid \text{Elem } ', ' \text{ Lista} \\ \text{Elem} \rightarrow \text{numero } \underline{\text{soma}} \mid \text{palavra} \end{array} \right\} \end{aligned}$$

o código associado às acções semânticas é:

```
inicio  somatorio ← 0;
fim    escrever (somatorio);
soma   somatorio ← somatorio + valor(numero);
```

O processador da linguagem definida por  $G_t$  calcula o somatório dos números de uma lista constituída por palavras e números separados por virgulas (além de efectuar o reconhecimento sintático de uma frase da linguagem). Para tal foi necessário introduzir as seguintes acções semânticas: a acção inicio que inicializa a variável *somatorio*, a acção fim que escreve o seu valor e, por último, a acção soma que adiciona à variável somatório o valor do número reconhecido. Repare-se que este *número* é um símbolo não literal passado pelo analisador léxico. Supondo que a estrutura de dados utilizada para a passagem dos lexemas é a definida anteriormente, então a função

$valor(numero)$  não faz mais do que devolver o valor do campo `numero` da estrutura `lexemas`.

Para facilitar a leitura da GT, geralmente as acções semânticas são escritas nas produções da GIC não lhes associando qualquer identificador. Delimitando as acções semânticas por chavetas as produções da GT que se obtêm são:

$$\begin{aligned} S &\rightarrow \{ \text{somatorio} \leftarrow 0; \} Lista \{ \text{escrever}(\text{somatorio}) \} \\ Lista &\rightarrow Elem \mid Elem \text{ '}' Lista \\ Elem &\rightarrow numero \{ \text{somatorio} \leftarrow \text{somatorio} + \text{valor}(\text{numero}) \} \mid palavra \end{aligned}$$

□

Para construir um processador de uma linguagem definida por uma gramática tradutora é necessário alterar os algoritmos de reconhecimento *top-down* e *bottom-up*, de modo a manusearem as acções semânticas e executarem-nas durante o reconhecimento sintático. Na secção seguinte apresentam-se os algoritmos *top-down*, posteriormente apresentam-se os algoritmos *bottom-up*.

### 4.3.2 Tradução Dirigida pela Sintaxe Top-Down

Na secção 3.3 foram apresentados duas estratégias distintas para construção de reconhecedores *top-down*: os reconhecedores recursivos descendentes e os reconhecedores dirigidos por uma tabela. De seguida analisa-se o problema da execução de acções semânticas, durante o reconhecimento sintático, em ambas as técnicas.

#### • Reconhecedor Recursivo Descendente com Acções Semânticas

A construção de um reconhecedor recursivo descendente é feito definindo um procedimento para cada símbolo da gramática (*c.f.* secção 3.3.4). No caso de uma gramática tradutora irá também ser associado um procedimento a cada símbolo de acção semântica. Este procedimento consiste na execução do código da respectiva acção semântica.

Supondo o símbolo não terminal  $A \in N$  com produções da forma

$$A \rightarrow \alpha a_1 B \mid a_2 C$$

com  $A, B, C \in N$ ;  $\alpha \in I$  e  $a_1, a_2 \in A$ . Então o procedimento tradutor associado a este símbolo será:

$$\text{reconhece\_A} \equiv \left\{ \begin{array}{l} \text{se } \text{simb} \in \text{Lookahead}(A \rightarrow \alpha a_1 B) \\ \quad \rightarrow \left\{ \begin{array}{l} \text{reconhece\_Term}(\alpha); \\ a_1; \\ \text{reconhece\_B}(); \end{array} \right. \quad \square \\ \text{simb} \in \text{Lookahead}(A \rightarrow a_2 C) \\ \quad \rightarrow \left\{ \begin{array}{l} a_2; \\ \text{reconhece\_C}(); \end{array} \right. \quad \square \\ \text{senão} \rightarrow \{\text{erro}\} \\ \text{fse} \end{array} \right.$$

Note-se que as acções semânticas não influenciam o cálculo dos *Lookaheads*. Assim, têm-se  $\text{Lookahead}(A \rightarrow \alpha a_1 B) = \{\alpha\}$  e  $\text{Lookahead}(A \rightarrow a_2 C) = \text{Lookahead}(A \rightarrow C)$

#### • Reconhecedor Dirigido por Tabela com Acções Semânticas

Num reconhecedor dirigido por tabela as acções semânticas têm um tratamento semelhante ao dos símbolos terminais. Neste reconhecedor sempre que se encontra um símbolo terminal  $t$  no topo da *stack de parsing* procede-se do seguinte modo: se  $t$  é igual ao símbolo da frase que se pretende reconhecer, então remove-se  $t$  da *stack* e avança-se para o próximo símbolo a reconhecer. No caso de se encontrar um identificador de uma acção semântica  $a$  no topo da *stack*, então executa-se o código associado a  $a$  e remove-se  $a$  da *stack*. Como é óbvio nesta situação não se avança para o símbolo seguinte da frase a reconhecer.

O algoritmo completo do reconhecedor dirigido por tabela com acções semânticas — *RDT\_AS* — apresenta-se de seguida.

**função**  $RDT\_AS(\alpha : TP, \beta : SP) : bool$

$$\left\{ \begin{array}{l}
 Erro \leftarrow Falso; \\
 simb \leftarrow ler(); \\
 \beta \leftarrow Push(S, <>); \\
 \mathbf{rep} \\
 \left\{ \begin{array}{l}
 t \leftarrow Top(\beta); \\
 \mathbf{se} \quad t \in N \\
 \longrightarrow \left\{ \begin{array}{l}
 p = \alpha(t, simb); \\
 \mathbf{se} \quad p = erro \longrightarrow \{ Erro \leftarrow Verdade; \\
 \mathbf{senão} \longrightarrow \left\{ \begin{array}{l} \beta \leftarrow Pop(\beta) \\ \beta \leftarrow Aplica(Ldp(p), \beta); \end{array} \right. \quad \square \\
 \mathbf{fse} \\
 t \in I \vee t = \$
 \end{array} \right. \\
 \longrightarrow \left\{ \begin{array}{l}
 \mathbf{se} \quad t = simb \\
 \longrightarrow \left\{ \begin{array}{l} \beta \leftarrow Pop(\beta); \\ simb \leftarrow ler(); \end{array} \right. \quad \square \\
 \mathbf{senão} \longrightarrow \{ Erro \leftarrow Verdade; \\
 \mathbf{fse} \\
 t \in A \\
 \longrightarrow \left\{ \begin{array}{l} executar(t); \\ \beta \leftarrow Pop(\beta); \end{array} \right. \\
 \mathbf{fse}
 \end{array} \right. \\
 \mathbf{até} \quad t = \$ \vee Erro = Verdade \\
 RDT\_AS \leftarrow (t = \$ \wedge Erro = Falso)
 \end{array} \right.$$

em que a função  $executar(t)$  não é mais que a execução do código associado à acção semântica  $t$ .

### 4.3.3 Tradução Dirigida pela Sintaxe Bottom-Up

Na tradução dirigida pela sintaxe *bottom-up* as acções semânticas associadas a uma produção  $p \in P$  são invocadas sempre que se efectua uma redução por  $p$ .

O algoritmo de reconhecimento LR  $RLR()$  apresentado na página 109 tem portanto de ser alterado de modo a manipular acções semânticas. Por

uma questão de simplicidade na construção do algoritmo vai-se considerar que cada produção tem apenas uma acção semântica e o identificador dessa acção é o símbolo mais à direita do lado direito da produção. Se uma produção não satisfizer esta condição então tem de ser transformada do modo à condição se verificar. Isto pode ser feito criando uma produção auxiliar que deriva na frase nula e que tem a acção semântica no seu lado direito. Assim, uma produção do tipo:

$$A \rightarrow \underline{ac_1} B \underline{ac_2}$$

(com  $A, B \in N$  e  $\underline{ac_1}, \underline{ac_2} \in I$ ) é transformada em

$$\begin{aligned} A &\rightarrow A_1 B \underline{ac_2} \\ A_1 &\rightarrow \epsilon \underline{ac_1} \end{aligned}$$

Note-se que estas transformações podem originar a ocorrência de conflitos redução/redução.

O algoritmo LR tem unicamente de ser alterado na parte referente à redução por uma produção  $p$ . Nessa situação, após se reduzir o lado direito de  $p$  ao seu lado esquerdo é necessário executar a respectiva acção semântica (caso possua). Isto pode ser feito invocando uma função, designada **executa**( $p: P$ ), que dada uma produção executa a respectiva acção semântica. De seguida apresenta-se um fragmento do algoritmo  $RLR()$  que sofre alterações para manipular acções semânticas.

```

se  ac = d                                     /* desloca */
   → { ...
   ac = A → γ                                 /* reduz */
   → {
       para i ∈ {1, 2, ..., 2 × |γ|}
         → {β ← Pop(β)}
       fpara
         q ← δ[Top(β), A];
         β ← Push(A, β);
         β ← Push(q, β);
         executa(ac);
   }

```

Exemplo: Considere a gramática tradutora  $G_t$  que calcula o somatório dos números numa lista de palavras e números (exemplo da página 130). Considere-

rando as transformações referidas anteriormente obtém-se o seguinte conjunto de produções:

$$p = \{ \begin{array}{ll} S' \rightarrow S\$ & (i) \\ S \rightarrow A_1 \text{ Lista } \underline{fim} & (ii) \\ A_1 \rightarrow \epsilon \underline{inicio} & (iii) \\ Lista \rightarrow Elem & (iv) \\ Lista \rightarrow Elem', ' Lista & (v) \\ Elem \rightarrow \underline{numerosoma} & (vi) \\ Elem \rightarrow palavra & (vii) \end{array} \}$$

Para desenvolver um tradutor dirigido pela sintaxe *bottom-up* é necessário construir as tabelas de *parsing* SLR (*c.f.* secção 3.4.2).

**Exercício 4.3** Construa as tabelas de *parsing* SLR para a gramática  $G_t$ .

Resolvendo o exercício anterior obtém-se a seguinte tabela:

Q	TA				TT								
	T				T $\cup$ N								
	num	pal	' , '	\$	num	pal	' , '	\$	S'	S	A <sub>1</sub>	Lista	Elem
0	<i>iii</i>	<i>iii</i>								1	2		
1				<i>Ok</i>									
2	d	d			6	7						3	4
3				<i>ii</i>									
4			d	<i>iv</i>			5						
5	d	d			6	7						8	4
6			<i>vi</i>	<i>vi</i>									
7			<i>vii</i>	<i>vii</i>									
8				<i>v</i>									

Utilizando o algoritmo apresentado anteriormente, o processamento da frase  $f = \text{são}, 12, 20 \in \mathcal{L}_{G_t}$  é feito do seguinte modo:

<i>stack de parsing</i>	entrada	acção semântica
0	são,12,20\$	<u>início</u>
0 $A_1$ 2	são,12,20\$	
0 $A_1$ 2 <i>pal</i> 7	,12,20\$	
0 $A_1$ 2 <i>Elem</i> 4	,12,20\$	
0 $A_1$ 2 <i>Elem</i> 4 , 5	12,20\$	
0 $A_1$ 2 <i>Elem</i> 4 , 5 <i>num</i> 6	,20\$	
0 $A_1$ 2 <i>Elem</i> 4 , 5 <i>Elem</i> 4	,20\$	<u>soma</u>
0 $A_1$ 2 <i>Elem</i> 4 , 5 <i>Elem</i> 4 , 5	20\$	
0 $A_1$ 2 <i>Elem</i> 4 , 5 <i>Elem</i> 4 , 5 <i>num</i> 6	\$	
0 $A_1$ 2 <i>Elem</i> 4 , 5 <i>Elem</i> 4 , 5 <i>Elem</i> 4	\$	<u>soma</u>
0 $A_1$ 2 <i>Elem</i> 4 , 5 <i>Elem</i> 4 , 5 <i>Lista</i> 8	\$	
0 $A_1$ 2 <i>Elem</i> 4 , 5 <i>Lista</i> 8	\$	
0 $A_1$ 2 <i>Lista</i> 3	\$	
0 $S$ 1	\$	<u>fim</u>
<i>ok!</i>		

Como se verifica no início do processamento é executada a acção que inicializa a variável *somatorio* (como pretendido) e no fim escreve-se o seu valor. Sempre que se processa um número então o seu valor é adicionado à variável *somatorio*.

□

#### 4.3.4 Estudo de um Caso

Nesta secção vai ser analisado um problema que requer a utilização de um tradutor dirigido pela sintaxe. Para desenvolver este tradutor vão ser consideradas as técnicas apresentadas ao longo deste texto. O tradutor vai ser implementado usando um reconhecedor recursivo descendente. Assim, em primeiro lugar irá definir-se a sintaxe da linguagem usando uma GIC. Posteriormente, esta gramática irá ser alterada de modo a satisfazer a condição LL(1) (ver secção 3.3.2). De seguida, define-se uma gramática tradutora (*c.f.* 4.3.1) considerando as acções semânticas necessárias para a análise semântica da linguagem e para o processamento pretendido. Considerando a arquitectura apresentada na secção 4.3 o analisador léxico será implementado utilizando a ferramenta LEX (ver secção 4.2.2). O *parser* irá ser implementado usando um reconhecedor recursivo descendente com acções semânticas. No apêndice D apresenta-se a implementação em linguagem C do respectivo tradutor dirigido pela sintaxe.



**Problema 4** Considere que se pretende desenvolver um processador de expressões aritméticas simples. As expressões são escritas da forma usual, i.e., sequência de operandos separados pelo operador que lhes será aplicado (usa-se uma notação infixa). Os operandos poderão ser números reais (sem sinal), outras expressões delimitadas pelos caracteres '(' e ')' e podem, ainda, ser identificadores. Os operadores são o '+', '-', '\*', e '/'; com a associatividade e precedência usual. Considera-se ainda a possibilidade de atribuir expressões a identificadores, podendo estes posteriormente ser utilizados em novas expressões. O processador deve detectar o uso, numa expressão, de um identificador ao qual ainda não foi atribuído nenhuma expressão. Neste caso deverá assinalar a existência de um erro. No caso de não existirem erros o processador deve enviar para a saída o resultado da expressão. Existe ainda a possibilidade de se escreverem comentários usando a sintaxe do C, i.é, entre '/\*' e '\*/'.

Um exemplo de uma frase válida da linguagem<sup>4</sup> será:

```

μ =
    <var> = 3 + 4; /* atrib a variavel */
    2 * (var - 5) ;
quit

```

## Resolução

A sintaxe da linguagem pode ser definida pela gramática independente do contexto  $G = (T, N, S, P)$ , em que

$T = \{+, -, *, /, (, ), <, >, =, ;, \text{num}, \text{ident}, \text{quit}\}$ ,

$N = \{S, \text{Insts}, \text{Inst}, \text{Expr}, \text{Atrib}, \text{Termo}, \text{Factor}\}$  e

$P = \{$

- $S \rightarrow \text{Insts quit}$
- $\text{Insts} \rightarrow \text{Inst} ; \text{Insts} \mid \epsilon$
- $\text{Inst} \rightarrow \text{Expr} \mid \text{Atrib}$
- $\text{Expr} \rightarrow \text{Expr} + \text{Termo} \mid \text{Expr} - \text{Termo} \mid \text{Termo}$
- $\text{Termo} \rightarrow \text{Termo} * \text{Factor} \mid \text{Termo} / \text{Factor} \mid \text{Factor}$
- $\text{Factor} \rightarrow \text{num} \mid \text{ident} \mid ( \text{Expr} )$
- $\text{Atrib} \rightarrow < \text{ident} > = \text{Expr}$

$\}$

---

<sup>4</sup>Esta linguagem é um subconjunto da linguagem aritmética utilizada pelo BC (iniciais de *Binary Calculator*), um comando do sistema operativo UNIX.

Em que *ident* representa a classe de todos os identificadores constituídos unicamente por letras minúsculas e *num* representa a classe de todos os números reais sem sinal.

De notar que esta gramática resolve os problemas da ambiguidade habitualmente existentes em gramáticas que especificam a sintaxe das expressões aritméticas. Isto acontece porque se consideram dois símbolos não terminais —*Expr*, *Termo*—, um para cada nível de precedência dos operadores. Assim, esta gramática considera as expressões como sendo uma lista de termos separados pelos operadores de menor precedência (os operadores '+' e '-') e, por sua vez, um termo é uma lista de factores separados pelos operadores de maior precedência (os operadores '\*' e '/'). Um factor define as unidades básicas das expressões, neste caso são os números, os identificadores e expressões entre parêntesis.

No entanto, esta gramática não satisfaz a *condição LL(1)*, pois, por exemplo, nas produções com o símbolo não terminal *Expr* do lado esquerdo, verifica-se o seguinte:

$$First(Expr) = First(Expr + Termo) \cup First(Expr - Termo) \cup First(Termo)$$

e

$$\begin{aligned} Lookahead(Expr \rightarrow Expr + Termo) &= First(Expr) \\ Lookahead(Expr \rightarrow Expr - Termo) &= First(Expr) \\ Lookahead(Expr \rightarrow Termo) &= First(Termo) \end{aligned}$$

sendo assim,

$$\begin{aligned} Lookahead(Expr \rightarrow Expr + Termo) \cap Lookahead(Expr \rightarrow Expr - Termo) \cap \\ Lookahead(Expr \rightarrow Termo) \neq \emptyset \end{aligned}$$

Como se pode constatar esta situação também acontece nas produções com o símbolo não terminal *Termo* do lado esquerdo.

Os conflitos LL(1) verificam-se em qualquer gramática recursiva à esquerda e a solução para o problema consiste em reescrever a gramática utilizando recursividade à direita (ver secção 3.3.3).

Efectuando a transformação respectiva em *G* obtém-se o seguinte conjunto de produções:

$$\begin{aligned}
P = \{ & S \rightarrow Insts \textit{ quit} \\
& Insts \rightarrow Inst ; Insts \mid \epsilon \\
& Inst \rightarrow Expr \mid Atrib \\
& Expr \rightarrow Termo \textit{ MaisTermos} \\
& MaisTermos \rightarrow + Termo \textit{ MaisTermos} \mid - Termo \textit{ MaisTermos} \mid \epsilon \\
& Termo \rightarrow Factor \textit{ MaisFactores} \\
& MaisFactores \rightarrow * Factor \textit{ MaisFactores} \mid / Factor \textit{ MaisFactores} \mid \epsilon \\
& Factor \rightarrow num \mid ident \mid ( Expr ) \\
& Atrib \rightarrow < ident > = Expr \\
& \}
\end{aligned}$$

em que o conjunto de símbolos não terminais de  $G$  passa a ser

$$N \cup \{MaisTermos, MaisFactores\}$$

Para efectuar o reconhecimento de uma frase é necessário saber qual a produção a utilizar ao derivar-se por um símbolo não terminal. Nos reconhecedores *top-down* isto é feito calculando, para cada produção da gramática, o conjunto de símbolos terminais que são inícios válidos de derivações a partir de um símbolo não terminal.

No caso da gramática que estamos a estudar o cálculo dos *Lookahead* é feito do seguinte modo:

$$\begin{aligned}
Lookahead(S \rightarrow Insts \textit{ quit}) &= First(Insts \textit{ quit})\{ident, num, (, <, quit\} \\
Lookahead(Insts \rightarrow \epsilon) &= Follow(Insts) = \{quit\} \\
Lookahead(Insts \rightarrow Inst ; Insts) &= First(Inst) = \\
&= First(Expr) \cup First(Atrib) = \\
&= \{ident, num, (, <\} \\
Lookahead(Inst \rightarrow Expr) &= First(Expr) = \{ident, num, (\} \\
Lookahead(Inst \rightarrow Atrib) &= First(Atrib) = \{<\} \\
Lookahead(Expr \rightarrow Termo \textit{ MaisTermos}) &= First(Termo) = \{num, ident, (\} \\
Lookahead(MaisTermos \rightarrow + Termo \textit{ MaisTermos}) &= \{+\} \\
Lookahead(MaisTermos \rightarrow - Termo \textit{ MaisTermos}) &= \{-\} \\
Lookahead(MaisTermos \rightarrow \epsilon) &= Follow(MaisTermos) = Follow(Expr) = \\
&= Follow(Inst) \cup \{\} \cup Follow(Atrib) = \{;, )\} \\
Lookahead(Termo \rightarrow Factor \textit{ MaisFactores}) &= \{ident, num, (\} \\
Lookahead(MaisFactores \rightarrow * Factor \textit{ MaisFactores}) &= \{*\} \\
Lookahead(MaisFactores \rightarrow / Factor \textit{ MaisFactores}) &= \{/ \}
\end{aligned}$$

$$\begin{aligned}
Lookahead(MaisFactores \rightarrow \epsilon) &= Follow(MaisFactores) = Follow(Termo) = \\
&= First(MaisTermos) \cup Follow(Expr) = \\
&= \{+, -, ), ;\} \\
Lookahead(Factor \rightarrow num) &= \{num\} \\
Lookahead(Factor \rightarrow ident) &= \{ident\} \\
Lookahead(Factor \rightarrow ( Expr ) &= \{(\} \\
Lookahead(Atrib \rightarrow < ident >= Expr) &= \{<\}
\end{aligned}$$

Como se pode verificar pelos conjuntos de *Lookahead*'s calculados, a gramática  $G$  satisfaz a condição LL(1), pois

$$\forall_{A \rightarrow \alpha_1, A \rightarrow \alpha_2 \in P} : Lookahead(A \rightarrow \alpha_1) \cap Lookahead(A \rightarrow \alpha_2) = \emptyset$$

Sendo assim, não existe ambiguidade na escolha da produção a utilizar no reconhecimento, sendo portanto possível construir um reconhecedor recursivo descendente.

### Processamento das Expressões

Neste exemplo, de reconhecimento de expressões aritméticas pretende-se ainda que o reconhecedor, além de efectuar a análise sintática das frases, processe essas mesmas expressões, de modo a apresentar o seu resultado. Um outro aspecto que também se pretende ter em conta é a validação semântica das frases. Mais concretamente, o reconhecedor/processador de expressões deverá assinalar o uso de um identificador ao qual não tenha sido previamente atribuído o valor de uma expressão.

Um modo de satisfazer estes aspectos consiste em estender a gramática  $G$  com acções semânticas que serão posteriormente executadas durante o reconhecimento sintático, obtendo-se assim o que se designa por *gramática tradutora*.

Assim, vão ter de definir várias acções semânticas de modo a implementar o pretendido. Para efectuar o cálculo das expressões vamos utilizar uma *stack*, que será uma estrutura de dados global, i.e., pode ser acedida por todos os procedimentos do reconhecedor. Existe a necessidade de utilizar uma *stack*, uma vez que uma expressão pode, por sua vez, conter uma outra expressão (pois, como se pode verificar nas produções de  $G$ , são permitidos aninhamentos), sendo necessário guardar o contexto da primeira expressão,

numa estrutura de dados quando se começa a processar a nova expressão, de modo a não se perderem o valor dos cálculos efectuados na expressão inicial. Assim, estas acções semânticas são:

<p><u>soma</u> :    <math>operando2 \leftarrow \text{pop}()</math>                      <math>operando1 \leftarrow \text{pop}()</math>                      <math>\text{push}(operando1 + operando2)</math></p>	<p><u>sub</u> :    <math>operando2 \leftarrow \text{pop}()</math>                      <math>operando1 \leftarrow \text{pop}()</math>                      <math>\text{push}(operando1 - operando2)</math></p>
<p><u>mul</u> :    <math>operando2 \leftarrow \text{pop}()</math>                      <math>operando1 \leftarrow \text{pop}()</math>                      <math>\text{push}(operando1 * operando2)</math></p>	<p><u>div</u> :    <math>operando2 \leftarrow \text{pop}()</math>                      <math>operando1 \leftarrow \text{pop}()</math>                      <math>\text{push}(operando1 / operando2)</math></p>
<p><u>guarda</u> : <math>\text{push}(num.valor)</math></p>	<p><u>esc</u> :    <math>result \leftarrow \text{pop}()</math>                      <math>\text{escreve}(result)</math></p>

As produções da gramática são extendidas com estas acções, do seguinte modo:

$$Factor \rightarrow num \underline{guarda}$$

Nesta produção insere-se a acção semântica que guarda na *stack* o valor associado ao número reconhecido. O valor associado ao símbolo terminal é passado pelo analisador léxico.

$MaisTermos \rightarrow + Termo \underline{soma} MaisTermos \mid - Termo \underline{sub} MaisTermos \mid \epsilon$   
 $MaisFactores \rightarrow * Factor \underline{mul} MaisFactores \mid / Factor \underline{div} MaisFactores \mid \epsilon$

Nas produções que contém os símbolos terminais que são os operadores, e após se reconhecer o termo que se segue ao operador, executa-se a acção semântica que retira da *stack* os dois operandos e efectua a respectiva operação. O resultado da operação é colocado de novo na *stack*.

$$Inst \rightarrow Expr \underline{esc}$$

Após se reconhecer uma instrução que é uma expressão, invoca-se a acção semântica que retira da *stack* o valor da expressão e escreve esse valor.

Para efectuar as validações semânticas referidas é também necessário incluir acções semânticas. Uma acção semântica que deverá existir é a acção

que armazena os vários identificadores, que aparecem nas atribuições, numa estrutura de dados global (designada **tabela de símbolos**). Durante o cálculo de expressões se aparecer um identificador terá de se verificar se ele já existe na tabela de símbolos. Caso exista será considerado, para o cálculo da expressão, o valor que foi armazenado na tabela de símbolos, caso contrário existirá um erro semântico. Deste modo, as acções semânticas são:

```

ins :   result ← pop()
         tabSimb ← insere_simbolo(tabSimb, ident.valor, result)

cons :   se valor ← procura_simbolo(tabSimb, ident.valor)
         → {push(valor)
            senão → {erro semântico

```

Estas acções são colocadas nas seguintes produções:

$$Factor \rightarrow ident \text{ cons}$$

Sempre que um *Factor* derivar num identificador põe-se na *stack* o valor armazenado na tabela de símbolos. Caso o identificador não exista, dá erro.

$$Atrib \rightarrow < ident > = Expr \text{ ins}$$

Quando se reconhece uma atribuição insere-se o identificador e o valor da expressão na tabela de símbolos.

Após se terem feito todas estas extensões obtém-se o seguinte conjunto de produções:

$$\begin{aligned}
P = \{ & S \rightarrow Insts \textit{quit} \\
& Insts \rightarrow Inst ; Insts \mid \epsilon \\
& Inst \rightarrow Expr \textit{esc} \mid Atrib \\
& Expr \rightarrow Termo \textit{MaisTermos} \\
& MaisTermos \rightarrow + Termo \textit{soma} MaisTermos \\
& \quad \mid - Termo \textit{sub} MaisTermos \mid \epsilon \\
& Termo \rightarrow Factor \textit{MaisFactores} \\
& MaisFactores \rightarrow * Factor \textit{mul} MaisFactores \\
& \quad \mid / Factor \textit{div} MaisFactores \mid \epsilon \\
& Factor \rightarrow num \textit{guarda} \mid ident \textit{cons} \mid ( Expr ) \\
& Atrib \rightarrow < ident > = Expr \textit{ins} \\
& \}
\end{aligned}$$

### Implementação do Processador

Como se verificou anteriormente  $G$  satisfaz a condição LL(1) admitindo, portanto, um tradutor dirigido pela sintaxe *top-down*. De seguida analisa-se o problema de desenvolver um tradutor baseado num algoritmo recursivo descendente. Nestes reconhecedores associa-se um procedimento a cada símbolo da gramática.

Antes de se escrever cada um destes procedimentos, recorda-se a arquitectura dos tradutores dirigidos pela sintaxe que está a ser considerada (*c.f.* secção 4.3). Nesta arquitectura, a análise léxica é feita por um módulo diferente da análise sintática. Esse analisador léxico recebe os vários caracteres do texto de entrada e agrupa-os em símbolos do vocabulário da linguagem —**lexemas**— passando estes lexemas ao analisador sintático. Um modo bastante eficiente para implementar um analisador léxico consiste em utilizar autómatos determinísticos finitos.

Deste modo, e voltando ao processador de expressões, vai-se construir o analisador léxico utilizando estes autómatos. Para tal, utiliza-se o LEX que a partir de uma especificação baseada em expressões regulares gera automaticamente um autómato determinístico finito eficiente.

O programa LEX que implementa o analisador léxico é constituído por várias expressões regulares, uma para cada símbolo do vocabulário (ver conjunto de símbolos terminais), cujas acções semânticas respectivas devolvem

os símbolos reconhecidos ao analisador sintático. Este programa apresenta-se de seguida.

```

partedec  "."[0-9]*
real      [0-9]+({partedec})?
%%
"+"       {return('+');}
"_"       {return('-');}
"*"       {return('*');}
"/"       {return('/');}
"("       {return('(');}
")"       {return(')');}
"<"       {return('<');}
">"       {return('>');}
"="       {return('=');}
";"       {return(';');}
quit      {return(QUIT);}
[a-z]+    {strcpy(vocab.ident,yytext);
           return(IDENT);}
{real}    {vocab.num = atof(yytext);
           return(NUMERO);}
"/*" [^"*/"] "*" /" ;
[ \n\t]   ;
.         {printf (" Erro lexico \n");
           exit (1);}

```

Nas acções semânticas do analisador léxico utiliza-se uma estrutura — **vocab**—, partilhada pelo analisador sintático, que permite a passagem de valores dos lexemas entre os dois analisadores.

Para escrever os procedimentos do reconhecedor recursivo descendente considera-se a existência de:

- Uma variável global —**Simbolo\_Frase**— que tem o símbolo da frase (lexema) que se pretende reconhecer;
- Uma função **Ler\_Lexema()** que invoca o analisador léxico (gerado pelo LEX) de modo a obter um novo lexema da frase a processar.



Sendo assim, os procedimentos são:

Para os símbolos terminais o procedimento consiste em verificar se o símbolo da frase a reconhecer coincide com o símbolo terminal pelo qual se pretende derivar. Caso isso se verifique será devolvido o símbolo da frase reconhecido e actualiza-se a variável global `Simbolo_Frase` com o novo símbolo a reconhecer. Caso não se verifique surge uma situação de erro local. Assim, a função associada aos símbolos terminais é:

$$\text{função } reconhece\_Term(t : T) : TIPO\_LEXEMAS$$

$$\left\{ \begin{array}{l} \text{se } t = Simbolo\_Frase \\ \quad \longrightarrow \left\{ \begin{array}{l} valor \leftarrow Simbolo\_Frase; \\ Simbolo\_Frase \leftarrow Ler\_Lexema(); \end{array} \right. \quad \square \\ \text{senão } \longrightarrow \{erro\ local; \\ \text{fse} \\ reconhece\_Term \leftarrow valor; \end{array} \right.$$

A função principal do analisador sintático, irá inicializar as variáveis necessárias (neste caso a tabela de símbolos), invocar a função `Ler_Lexema()` de modo a actualizar o símbolo a reconhecer e posteriormente invoca o procedimento associado ao axioma da gramática.

$$\text{função } parser()$$

$$\left\{ \begin{array}{l} inic(tabSimb) \\ Simbolo\_Frase \leftarrow Ler\_Lexema() \\ reconhece\_S() \end{array} \right.$$

Nos procedimentos associados aos símbolos não terminais terá de se decidir qual a produção a utilizar na derivação. Isto é feito a partir do conjunto de *Lookahead*. Assim, se o símbolo da frase a reconhecer pertencer ao conjunto de *Lookahead* de uma produção, então será essa a produção a usar e, obviamente, terão de se reconhecer os vários símbolos do lado direito da produção. O reconhecimento destes símbolos do lado direito da produção é feito invocando os procedimentos que reconhecem cada um deles.

De seguida, apresentam-se os algoritmos de algumas funções do exemplo do processador de expressões.

$$\begin{array}{l}
\text{reconhece\_S}() \equiv \\
\left\{ \begin{array}{l}
\text{se } \text{Simbolo\_Frase} \in \{IDENT, NUMERO, (, <\} \\
\rightarrow \left\{ \begin{array}{l} \text{reconhece\_Insts}() \\ \text{reconhece\_Term}(\text{QUIT}) \end{array} \right. \quad \square \\
\text{senao} \rightarrow \{\text{erro global}\} \\
\text{fse}
\end{array} \right.
\end{array}$$

$$\begin{array}{l}
\text{reconhece\_Insts}() \equiv \\
\left\{ \begin{array}{l}
\text{se } \text{Simbolo\_Frase} \in \{\text{QUIT}\} \\
\rightarrow \{ \square \} \\
\text{Simbolo\_Frase} \in \{IDENT, NUMERO, (, <\} \\
\rightarrow \left\{ \begin{array}{l} \text{reconhece\_Inst}() \\ \text{reconhece\_Term}(;) \end{array} \right. \quad \square \\
\text{senão} \rightarrow \{\text{erro global}\} \\
\text{fse}
\end{array} \right.
\end{array}$$

$$\begin{array}{l}
\text{reconhece\_Inst}() \equiv \\
\left\{ \begin{array}{l}
\text{se } \text{Simbolo\_Frase} \in \{IDENT, NUMERO, (\} \\
\rightarrow \left\{ \begin{array}{l} \text{reconhece\_Expr}() \\ \text{result} \leftarrow \text{pop}() \\ \text{escreve}(\text{result}) \end{array} \right. \quad \square \\
\text{Simbolo\_Frase} \in \{<\} \\
\rightarrow \{ \text{reconhece\_Atrib}() \} \quad \square \\
\text{senão} \rightarrow \{\text{erro global}\} \\
\text{fse}
\end{array} \right.
\end{array}$$

Como se pode verificar neste procedimento, o reconhecimento de um símbolo de uma acção semântica durante a análise sintática, corresponde a executar o código dessa mesma acção.

$$\begin{array}{l}
\text{reconhece\_Atrib}() \equiv \\
\left\{ \begin{array}{l}
\text{se } Simbolo\_Frase \in \{<\} \\
\quad \rightarrow \left\{ \begin{array}{l}
reconhece\_Term(<) \\
aux \leftarrow reconhece\_Term(IDENT) \\
reconhece\_Term(>) \\
reconhece\_Term(=) \\
reconhece\_Expr() \\
result \leftarrow pop() \\
tabSimb \leftarrow insere\_simbolo(tabSimb, aux.valor, result)
\end{array} \right. \quad \square \\
\text{senão} \rightarrow \{erro\ global \\
\text{fse}
\end{array} \right.
\end{array}$$

$$\begin{array}{l}
\text{reconhece\_Factor}() \equiv \\
\left\{ \begin{array}{l}
\text{se } Simbolo\_Frase \in \{NUMERO\} \\
\quad \rightarrow \left\{ \begin{array}{l}
aux \leftarrow reconhece\_Term(IDENT) \quad \square \\
push(aux.valor)
\end{array} \right. \\
Simbolo\_Frase \in \{IDENT\} \\
\quad \rightarrow \left\{ \begin{array}{l}
aux \leftarrow reconhece\_Term(IDENT) \\
\quad \text{se } (valor \leftarrow procura\_simbolo(tabSimb, aux.valor)) \quad \square \\
\quad \quad \rightarrow \{push(valor)\} \\
\quad \text{senão} \rightarrow \{erro\ semântico
\end{array} \right. \\
Simbolo\_Frase \in \{(\} \\
\quad \rightarrow \left\{ \begin{array}{l}
reconhece\_Term(()) \\
reconhece\_Expr() \quad \square \\
reconhece\_Term(())
\end{array} \right. \\
\text{senão} \rightarrow \{erro\ global \\
\text{fse}
\end{array} \right.
\end{array}$$

Os restantes algoritmos são deixados como exercício.

A implementação deste tradutor dirigido pela sintaxe, em linguagem C, apresenta-se no apêndice D.

**Exercício 4.4** Considere que se pretende processar as frases da linguagem definidas pela gramática do exercício 3.13, de modo a calcular o valor das expressões. Indique que acções semânticas seriam necessárias e escreva a respectiva Gramática Tradutora.



# Apêndice A

## Meta-Gramática

Nesta secção apresenta-se uma gramática independente do contexto que define a sintaxe da notação utilizada ao longo deste texto para escrever gramáticas.

$G = (T, N, S, P)$  com

$T = \{ \text{G}, =, (, ), \text{N}, \text{T}, \text{S}, \text{P}, \{, \}, ,, -, >, |, \epsilon, \text{IdTerm}, \text{IdNTerm} \}$

$N = \{ S, \text{Tuplo}, \text{Elems}, \text{NTerm}, \text{Term}, \text{Axioma}, \text{Prod}, \text{DefT}, \text{SeqSimbTerm}, \text{DefN}, \text{SeqSimbNTerm}, \text{DefP}, \text{SeqProducoes}, \text{Producao}, \text{LadoDirProd}, \text{DefS} \}$

$P = \{$   
     $S \rightarrow \text{Tuplo } \text{Elems}$   
     $\text{Elems} \rightarrow \text{DefT } ' , ' \text{DefN } ' , ' \text{DefS } ' , ' \text{DefP} \mid \epsilon$   
     $\text{Tuplo} \rightarrow \text{G} = ( \text{Term } ' , ' \text{NTerm } ' , ' \text{Axioma } ' , ' \text{Prod} )$   
     $\text{Term} \rightarrow \text{T} \mid \text{DefT}$   
     $\text{NTerm} \rightarrow \text{N} \mid \text{DefN}$   
     $\text{Axioma} \rightarrow \text{S} \mid \text{DefS}$   
     $\text{Prod} \rightarrow \text{P} \mid \text{DefP}$   
     $\text{DefT} \rightarrow \text{T} = \{ \text{SeqSimbTerm} \} \mid \epsilon$   
     $\text{SeqSimbTerm} \rightarrow \text{IdTerm} \mid \text{IdTerm } ' , ' \text{SeqSimbTerm}$   
     $\text{DefN} \rightarrow \text{N} = \{ \text{SeqSimbNTerm} \} \mid \epsilon$   
     $\text{SeqSimbNTerm} \rightarrow \text{IdNTerm} \mid \text{IdNTerm } ' , ' \text{SeqSimbNTerm}$   
     $\text{DefS} \rightarrow \text{S} = \text{IdNTerm}$   
     $\text{DefP} \rightarrow \text{P} = \{ \text{SeqProduções} \} \mid \epsilon$   
 $\}$

$$\begin{aligned}
&SeqProduções \rightarrow Produção \mid Produção \text{ ' , ' } SeqProduções \\
&Produção \rightarrow IdNTerm \rightarrow LadoDirProd \\
&LadoDirProd \rightarrow IdNTerm \ LadoDirProd \mid IdTerm \ LadoDirProd \\
&\qquad\qquad\qquad \mid \text{ ' ' } LadoDirProd \mid \epsilon \mid \epsilon \\
&\}
\end{aligned}$$

Em que o símbolo terminal **IdTerm** representa a classe de todas as frases constituídas por caracteres que são letras minúsculas. O símbolo **IdNTerm** representa a classe das frases constituídas por um primeiro caracter que é uma letra maiúscula seguido por zero ou mais caracteres que são letras maiúsculas ou minúsculas.

Utilizando-se expressões regulares estes símbolos terminais podem ser expressos do seguinte modo:

$$\begin{aligned}
IdTerm &= [a - z]^+ \\
IdNTerm &= [A - Z][A - Z a - z]^*
\end{aligned}$$

# Apêndice B

## Implementação de um Reconhecedor de uma Linguagem Regular

```
#include <stdio.h>

#define TRUE 1
#define FALSE 0
#define MOSTRA_TRANSICOES TRUE
#define ACEITA 1
#define ERRO -1
#define EMPTY 0

#define A 0
#define B 1
#define C 2
#define D 3
#define E 4
#define F 5
#define G 6
#define H 7
#define I 8
#define J 9

#define S A
#define Z J

int head(char *frase)
{
    return ((int) (*frase - '0'));
}

char *tail(char *frase)
{
    return (++frase);
}
```

```

int R_L (int TT[][2],char *frase)
{
    int res;
    int alpha;

    alpha = S;
    while ( (*frase != EMPTY) && (alpha != ERRO) )
    {
        if (MOSTRA_TRANSICOES)
            printf(" ((%c,%c),%c)\n",alpha+'A',*frase,
                TT[alpha][*frase-'0']+'A');

        alpha = TT[alpha][head(frase)];
        frase = tail(frase);
    }
    if ( (alpha == Z) && (*frase == EMPTY) )
        res = ACEITA;
    else
        res = ERRO;
    return res;
}

void main()
{
    int TT[10][2] = { B , ERRO ,
                      C , ERRO ,
                      D , ERRO ,
                      E , F ,
                      E , G ,
                      E , G ,
                      E , H ,
                      I , J ,
                      E , G ,
                      I , J    };

    char c,frase[256];
    char erro = FALSE;
    int i,res;

    for ( i = 0 ; ((c = getchar()) != EOF) && (i < 256) ; ++i )
    {
        if ( ((c-'0') != 0) && ((c-'0') != 1) )
        { printf (" Erro %c nao 'e um simbolo valido \n",c);
          erro = TRUE;
        }
        else
            frase[i] = c;
    }
    frase[i] = EMPTY;

    if (!erro)
    {
        res = R_L (TT,frase);

        (res == ACEITA) ? printf(" Frase Aceite\n") : printf(" Erro\n");
    }
}

```



# Apêndice C

## Implementação de um Processador de uma Linguagem Regular

```
#include <stdio.h>

#define TRUE 1
#define FALSE 0
#define MOSTRA_TRANSICOES TRUE

#define LIMITE 256

#define ACEITA 1
#define ERRO -1
#define EMPTY 0

#define A 0
#define B 1
#define C 2
#define D 3
#define E 4
#define F 5
#define G 6
#define H 7
#define I 8
#define J 9

#define S A
#define Z J

typedef struct transicao
{
    int estado_por_0;
    void (*f_0)(char);
    int estado_por_1;
```

```

    void (*f_1)(char);
} TRANSICAO;

char buffer[LIMITE];
int  ind_b;
int  valores[100];
int  ind_v;

char head(char *frase)
{
    return ((char) *frase);
}

char *tail(char *frase)
{
    return (++frase);
}

void erro ()
{
    printf (" Erro!! \n");
}

int e_separador (char buffer[])
{
    if ( (buffer[ind_b - 1] == '0') && (buffer[ind_b - 2] == '1') &&
        (buffer[ind_b - 3] == '1') )
        return TRUE;
    else
        return FALSE;
}

int myexp (int b, int e)
{
    int i,aux;

    aux = 1;
    for ( i = 1 ; i <= e ; ++i )
        aux *= b;
    return aux;
}

int conv_val_decimal (char buffer[])
{
    int res,i,j;

    res = j = 0;
    for ( i = ind_b-4 ; i >= 0 ; --i)
    {
        if ( buffer[i] == '1' )
            res += myexp(2,j);
        buffer[i] = EMPTY;
        ++j;
    }
    ind_b = 0;
    return res;
}

```

---

```

void proc_val (char simbolo)
{
    buffer[ind_b++] = simbolo;
    buffer[ind_b] = EMPTY;

    if ( e_separador(buffer) )
    {
        valores[ind_v++] = conv_val_decimal(buffer);
    }
}

int novo_estado (TRANSICAO t,char simbolo)
{
    switch (simbolo)
    {
        case '0': { if (MOSTRA_TRANSICOES)
                    printf(" %c),( %d,%s)) \n",simbolo,t.estado_por_0,"f_pv()");
                    t.f_0(simbolo);
                    return t.estado_por_0;
                }
        case '1': { if (MOSTRA_TRANSICOES)
                    printf(" %c),( %d,%s)) \n",simbolo,t.estado_por_1,"f_pv()");
                    t.f_1(simbolo);
                    return t.estado_por_1;
                }
    }
}

void f_vazia()
{
}

int R_L (TRANSICAO TT[],char *frase)
{
    int res;
    int alpha;

    alpha = S;
    while ( (*frase != EMPTY) && (alpha != ERRO) )
    {
        if (MOSTRA_TRANSICOES)
            printf(" ((%c,",alpha+'A');

        alpha = novo_estado(TT[alpha],head(frase));
        frase = tail(frase);
    }
    if ( (alpha == Z) && (*frase == EMPTY) )
    {
        valores[ind_v++] = conv_val_decimal (buffer);
        res = ACEITA;
    }
    else
        res = ERRO;
    return res;
}

```

```

void main()
{
    TRANSICAO TT[10] = { { B , f_vazia , ERRO, erro } ,
                          { C , f_vazia , ERRO, erro } ,
                          { D , f_vazia , ERRO, erro } ,
                          { E , proc_val , F, proc_val } ,
                          { E , proc_val , G, proc_val } ,
                          { E , proc_val , G, proc_val } ,
                          { E , proc_val , H, proc_val } ,
                          { I , proc_val , J, proc_val } ,
                          { E , proc_val , G, proc_val } ,
                          { I , proc_val , J, proc_val }
    };

    char c,frase[256];
    char erro = FALSE;
    int i,res;

    for ( i = 0 ; ((c = getchar()) != EOF) && (i < 256) ; ++i )
    {
        if ( ((c-'0') != 0) && ((c-'0') != 1) )
        {
            printf (" Erro %c nao 'e um simbolo valido \n",c);
            erro = TRUE;
        }
        else
            frase[i] = c;
    }
    frase[i] = EMPTY;

    if (!erro)
    {
        ind_v = ind_b = 0;
        res = R_L (TT,frase);

        (res == ACEITA) ? printf(" Frase Aceite\n") : printf(" Erro \n");

        for ( i = 0 ; i < ind_v ; ++i )
            printf (" val_%d -> %d \n",i,valores[i]);
    }
}

```

# Apêndice D

## Implementação de um Processador Top-Down

### Lexico.l

```
partedec  "."[0-9]*
real      [0-9]+({partedec})?
%%
"+"      {return('+' );}
"_"      {return('-' );}
"*"      {return('*' );}
"/"      {return('/') };
"("      {return('(' );}
")"      {return(')' );}
"<"      {return('<' );}
">"      {return('>' );}
"="      {return('=' );}
";"      {return(';' );}
quit     {return(QUIT);}
[a-z]+   {strcpy(vocab.ident,yytext);
          return(IDENT);}
{real}   {vocab.num = atof(yytext);
          return(NUMERO);}
"/"["~*/*"]*"/" ;
[ \n\t]  ;
.        {printf (" Erro lexico \n"); exit(1);}
```

### Lexico.h

```
#ifndef DEF_lexico
#define DEF_lexico

#define QUIT          300
#define IDENT         301
#define NUMERO        302
typedef int TIPO_LEXEMAS;
#endif
```

## TabSimb.c

```
#include <stdio.h>

#include "TabSimb.h"

#include "Debug.h"

WODO insere_simbolo (tabela,ident,valor)
WODO  tabela;
char  *ident;
float  valor;
{
    WODO aux;

    if (DEBUG)
        printf ("IS -> Ident: %s  valor: %f \n",ident,valor);

    aux = (WODO) malloc (sizeof(struct nodo));
    strcpy(aux->ident,ident);
    aux->valor = valor;
    aux->apseg = tabela;

    tabela = aux;
}

int procura_simbolo (tabela,ident,valor)
WODO  tabela;
char  *ident;
float *valor;
{
    WODO aux;

    if (DEBUG)
        printf ("PS -> Ident: %s \n",ident);

    for ( aux = tabela ; (aux != NULL) && ( strcmp(aux->ident,ident)!=0 ) ;
        aux = aux->apseg )
        ;

    if ( ( aux != NULL ) &&
        ( strcmp (aux->ident,ident) == 0 ) )
    {
        *valor = aux->valor;
        return (TRUE);
    }
    else
        return (FALSE) ;
}
```

## TabSimb.h

```
#ifndef DEF_TabSimb
#define DEF_TabSimb
```

---

```

struct nodo
{
    char ident[256];
    float valor;
    struct nodo *apseg;
};

typedef struct nodo * NODO;

extern NODO  insere_simbolo();
extern int   procura_simbolo();

#endif

```

## Stack.c

```

#include <stdio.h>

float  stack[300];
int    sp=0;

push (elemento)
float elemento;
{
    stack[sp++] = elemento;
}

pop (fl)
float *fl;
{
    float fl_aux;

    if ( sp == 0 )
        { fprintf (stderr, " Stack empty!! \n");
          exit (1);
        }
    else
        {
            sp--;
            *fl = stack[sp];
        }
}

```

## Parser.c

```

#include <stdio.h>
#include <math.h>

#include "Lexico.h"

struct    simbolos_vocabulario
{
    float  num;
    char   ident[256];
}

```

```

} vocab;

#include "lex.yy.c"

typedef struct simbolos_vocabulario *SIMB_VOCAB;

TIPO_LEXEMAS Simbolo_Frase;

#include "TabSimb.h"
#include "Debug.h"

#define ERRO_G { printf (" Erro Sintatico !! \n"); exit (1); }
#define ERRO_L { printf (" Erro Sintatico !! "); }

void reconhece_Termo();
void reconhece_Expressao();
void reconhece_Instrucao();
void reconhece_Instrucoes();
void reconhece_S();

NODO          tabela_Simbolos;

/*-----*/
/*-----*/

TIPO_LEXEMAS Ler_Lexemas()
{
    TIPO_LEXEMAS aux;

    aux = yylex();

    if (DEBUG)
        printf (" Lexema: %d \n",aux);

    return (aux);
}

SIMB_VOCAB reconhece_Term (terminal)
TIPO_LEXEMAS terminal;
{
    if ( terminal == Simbolo_Frase )
    {
        Simbolo_Frase = Ler_Lexemas ();
    }
    else
    {
        ERRO_L
        printf (" Aviso -> Simbolo da frase e' %c e esperava-se o terminal %c\n",
                Simbolo_Frase,terminal);

        /* Recuperacao do erro -> Devolver como resultado o terminal que se
        pretendia reconhecer e atribuir a Simbolo_Frase o novo lexema a
        processar */
    }
}

```



---

```

        Simbolo_Frase = Ler_Lexemas ();
    }
    return (&vocab);
}

void reconhece_MaisTermos()
{
    SIMB_VOCAB    aux1;
    float         resultado,operando_1,operando_2,aux2;

    if (DEBUG)
        printf ("Entrei no reconhece_MaisTermos: %d \n",Simbolo_Frase);

    if ( Simbolo_Frase == '+' )
    {
        reconhece_Term ('+');
        reconhece_Termo ();
        pop (&operando_2);
        pop (&operando_1);
        resultado = operando_1 + operando_2;
        push (resultado);
        reconhece_MaisTermos ();
    }
    else
        if ( Simbolo_Frase == '-' )
        {
            aux1 = reconhece_Term ('-');
            reconhece_Termo ();
            pop (&operando_2);
            pop (&operando_1);
            resultado = operando_1 - operando_2;
            push (resultado);
            reconhece_MaisTermos ();
        }
        else
            if ( Simbolo_Frase == ')' ||
                Simbolo_Frase == ';' )
            {
            }
            else
                ERRO_G
    }

void reconhece_Factor()
{
    SIMB_VOCAB    aux1;
    float         aux,valor;

    if (DEBUG)
        printf ("Entrei no reconhece_Factor %d\n",Simbolo_Frase);

    if ( Simbolo_Frase == NUMERO )
    {
        aux1 = reconhece_Term (NUMERO);

```

```

        aux = aux1->num;
        push (aux);
    }
    else
    if ( Simbolo_Frase == IDENT )
    {
        aux1 = reconhece_Term (IDENT);
        if ( procura_simbolo (tabela_Simbolos,aux1->ident,&valor) )
            push (valor);
        else
        {
            printf (" Erro Semantico -> Identificador %s nao definido! \n",aux1->ident);
            exit (0);
        }
    }
    else
    if ( Simbolo_Frase == '(' )
    {
        reconhece_Term '(';
        reconhece_Expressao ();
        reconhece_Term (')');
    }
    else
        ERRO_G
}

void reconhece_MaisFactores()
{
    SIMB_VOCAB    aux1;
    float         resultado,operando_1,operando_2,aux2;

    if (DEBUG)
        printf ("Entrei no reconhece_MaisFactores: %d \n",Simbolo_Frase);

    if ( Simbolo_Frase == '*' )
    {
        reconhece_Term ('*');
        reconhece_Factor ();
        pop (&operando_2);
        pop (&operando_1);
        resultado = operando_1 * operando_2;
        push (resultado);
        reconhece_MaisFactores ();
    }
    else
    if ( Simbolo_Frase == '/' )
    {
        reconhece_Term ('/');
        reconhece_Factor ();
        pop (&operando_2);
        pop (&operando_1);
        resultado = operando_1 / operando_2;
        push (resultado);
        reconhece_MaisFactores ();
    }
    else

```

---

```

        if ( Simbolo_Frase == '-' ||
            Simbolo_Frase == '+' ||
            Simbolo_Frase == ')' ||
            Simbolo_Frase == ';' )
        {

        }
        else
            ERRO_G
    }

void reconhece_Termo()
{
    float aux,aux1;

    if (DEBUG)
        printf ("Entrei no reconhece_Termo: %d \n",Simbolo_Frase);

    if ( Simbolo_Frase == NUMERO ||
        Simbolo_Frase == IDENT ||
        Simbolo_Frase == '(' )
    {
        reconhece_Factor ();
        reconhece_MaisFactores ();
    }
    else
        ERRO_G
}

void reconhece_Expressao()
{
    float aux,aux1,aux2;

    if (DEBUG)
        printf ("Entrei no reconhece_Expressao \n");

    if ( Simbolo_Frase == NUMERO ||
        Simbolo_Frase == IDENT ||
        Simbolo_Frase == '(' )
    {
        reconhece_Termo ();
        reconhece_MaisTermos ();
    }
    else
        ERRO_G
}

void reconhece_Atribuicao()
{
    SIMB_VOCAB aux1;
    float resultado;
    char str[256];

    if (DEBUG)
        printf ("Entrei no reconhece_Atribuicao \n");

```

```

    if ( Simbolo_Frase == '<' )
    {
        reconhece_Term ('<');
        aux1 = reconhece_Term (IDENT);
        strcpy(str,aux1->ident);
        reconhece_Term ('>');
        reconhece_Term ('=');
        reconhece_Expressao();

        pop (&resultado);
        tabela_Simbolos = (NODO) insere_simbolo (tabela_Simbolos,
                                                    str,resultado);
    }
    else
        ERRO_G
}

void reconhece_Instrucoes()
{
    if (DEBUG)
        printf ("Entrei no reconhece_Instrucoes \n");

    if ( Simbolo_Frase == QUIT )
    {
    }

    else
        if ( Simbolo_Frase == IDENT ||
            Simbolo_Frase == NUMERO ||
            Simbolo_Frase == '(' ||
            Simbolo_Frase == '<' )
        {
            reconhece_Instrucao();
            reconhece_Term (';');
            reconhece_Instrucoes();
        }
        else
            ERRO_G
}

void reconhece_Instrucao()
{
    float aux;
    float resultado;

    if (DEBUG)
        printf ("Entrei no reconhece_Instrucao \n");

    if ( Simbolo_Frase == NUMERO ||
        Simbolo_Frase == IDENT ||
        Simbolo_Frase == '(' )
    {
        reconhece_Expressao();
        pop (&resultado);
        printf (" Resultado da expressao: %f \n",resultado);
    }
}

```

---

```
    else
        if ( Simbolo_Frase == '<' )
        {
            reconhece_Atribuicao();
        }
        else
            ERRO_G
    }

void reconhece_S()
{
    if (DEBUG)
        printf ("Entrei no reconhece_S \n");

    if ( Simbolo_Frase == IDENT ||
        Simbolo_Frase == NUMERO ||
        Simbolo_Frase == '(' ||
        Simbolo_Frase == '<' )
    {
        reconhece_Instrucoes();
        reconhece_Term (QUIT);
    }
    else
        ERRO_G
}

main()
{
    tabela_Simbolos = NULL;
    Simbolo_Frase = Ler_Lexemas();

    reconhece_S();

    exit (0);
}

/*-----*/
/*-----*/
```



# Bibliografia

- [AB90] JOSÉ JOÃO ALMEIDA E JOSÉ BERNARDO BARROS. *Linguagens*. Texto Pedagógico - Departamento de Informática, Universidade do Minho, Outubro 1990.
- [ASU86] ALFRED AHO, RAVI SETHI E JEFFREY ULLMAN. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [BS93] JOSÉ CARLOS BACELAR E JOÃO SARAIVA. *Linguagens de Programação*. Texto Pedagógico - Departamento de Informática, Universidade do Minho, Outubro 1993.
- [FB94] ROBERT W. FLOYD E RICHARD BEIGEL. *The Languages of Machines*. W. H. Freeman and Company, 1994.
- [FH95] CHRISTOPHER FRASER E DAVID HANSON. *A Retargetable C Compiler: Design and Implementation*. Benjamin Cummings, 1995.
- [Hen92] PEDRO R. HENRIQUES. *Gramáticas de Atributos*. Texto Pedagógico (mestrado de informática) - Departamento de Informática, Universidade do Minho, Janeiro 1992.
- [LMB92] JOHN LEVINE, TONY MASON E DOUG BROWN. *lex & yacc*. O'Reilly & Associates, Inc., 1992.
- [Mar92] FERNANDO MÁRIO MARTINS. *Algoritmos e Estruturas de Dados*. Texto Pedagógico - Departamento de Informática, Universidade do Minho, Outubro 1992.
- [Oli91] JOSÉ NUNO OLIVEIRA. *Teoria das Linguagens e Compilação*. Texto Pedagógico - Departamento de Informática, Universidade do Minho, Outubro 1991.

- [Val86] JOSÉ MANUEL VALENÇA. *Teoria das Linguagens da Programação*. Texto Pedagógico - Departamento de Informática, Universidade do Minho, Outubro 1986.
- [Val93] JOSÉ MANUEL VALENÇA. *Programação Estruturada de Computadores*. Texto Pedagógico - Departamento de Informática, Universidade do Minho, Outubro 1993.
- [WC93] WILLIAM WAITE E LYNN CARTER. *An Introduction to Compiler Construction*. Harper Collins, 1993.
- [Wir76] NIKLAUS WIRTH. *Algorithms + Data Structures = Programs*. Prentice Hall, 1976.