Computer Science is a science of abstraction-creating the right model for a problem and devising the appropriate mechanizable techniques to solve it…

Alfred Aho

2

LINGUAGENS FORMAIS E AUTÓMATOS

universidade de aveiro
theoria poiesis praxis

Atenção!

Todo o conteúdo deste documento pode conter alguns erros de sintaxe, científicos, entre outros… **Não estudes apenas a partir desta fonte**. Este documento apenas serve de apoio à leitura de outros livros, tendo nele contido todo o programa da disciplina de Linguagens Formais e Autómatos, tal como foi lecionada, no ano letivo de 2015/2016, na Universidade de Aveiro. Este documento foi realizado por Rui Lopes.

mais informações em ruieduardofalopes.wix.com/apontamentos

Até à data estudámos paradigmas de programação, mas agora, em Linguagens Formais e Autómatos (a2s2) iremos descer de nível e estudar o comportamento e a apresentação de tudo o que é manipulado pelos compiladores.

1. Introdução ao processamento de linguagens

compilador

Os **compiladores** são responsáveis por processar linguagens. Um compilador é um processador que converte um programa-base numa linguagem de programação para uma linguagem de programação alvo, quer esta seja Assembly ou outra. Este processamento está representado na Figura 1.1.

figura 1.1
compilador

interpretador
híbridos

Por outro lado, um **interpretador** é também um processador que interpreta texto e, ao terminar de ler cada instrução executa-a. Também é possível encontrarmos **híbridos**, isto é, processadores que usem sequências de compilação e interpretação, como é o caso do efetuado na linguagem Java, onde o código é primeiramente compilado e depois interpretado pela JVM (Java Virtual Machine).

Aproximação ao conceito de linguagem

linguagem
símbolos
gramática

Durante todo o nosso estudo nesta e noutras disciplinas, temos vindo a referir linguagens. Mas afinal o que é que são linguagens? Uma **linguagem** é um conjunto de símbolos que se usa para transmitir informação. Esta informação, transmitida numa linguagem, representa-se por uma sequência de **símbolos**. Mas será que qualquer sequência de símbolos faz parte de uma linguagem? Vejamos a linguagem portuguesa (o idioma português). Como referimos, uma linguagem é um conjunto de símbolos - no nosso caso, palavras. Consideremos então o seguinte conjunto de palavras como uma linguagem: nós, universidade, informática, na, estudamos, José, melancólico, país, casa, casota. Se dissermos "Nós estudamos informática na universidade" está correto, mas se dissermos "Nós estudamos informática na nós" há qualquer coisa que nos faz pensar que não faz sentido - a seguir ao "na" estávamos à espera de um nome, não de um pronome pessoal. Estas regras são ditadas por uma **gramática** que basicamente se pode traduzir numa forma de descrever as sequências de símbolos válidas.

Em mais rigor, uma linguagem possível será o conjunto dos vocábulos em português, onde os símbolos são as letras do alfabeto com as suas variações acentuadas e cedilhadas e cujas sequências são os vocábulos do português (as palavras) - porque "arroz" faz sentido, mas "aarroz" já não.

Conceitos básicos sobre linguagens

alfabeto, palavra
string
comprimento
palavra vazia

Como já sabemos uma linguagem possui símbolos (também designados por letras), cujo conjunto é um **alfabeto**, sendo este um conjunto finito e não-vazio. Uma **palavra**, também vulgarmente designada de **string**, é uma sequência de símbolos sobre um dado alfabeto.

Outros conceitos importantes são o de **comprimento** de uma palavra u, representado por $|u|$, o que representa o seu número de símbolos. Por exemplo, se $u =$ "Aveiro", então $|u| = 6$. Uma palavra de comprimento 0 é considerada uma **palavra vazia** e denota-se por ε ou λ. Note-se ainda que λ não pertence ao alfabeto. O conjunto de todas as palavras de um alfabeto A inclui a palavra vazia e representa-se por $\boldsymbol{A^*}$, pelo que

se A = {0,1}, então A^* = {λ, 0, 1, 00, 01, 10, 11, 000, 001, ...}. Uma linguagem A é um subconjunto de A^*.

Operações sobre palavras

Tal como as operações que conhecemos à escala aritmética ou lógica, especialmente baseada nesta última, temos as operações sobre palavras. Uma primeira operação é a **concatenação** (ou **produto**). Assim, a concatenação das palavras u e v denota-se por $u.v$ ou simplesmente uv e representa a justaposição de u e v, sendo que $|u.v| = |u| + |v|$. Por exemplo, se u = "micro" e v = "ondas", então $u.v$ = "microondas". Esta operação também tem as suas propriedades, pelo que goza da propriedade associativa $u.(v.w) = (u.v).w = u.v.w$ e da propriedade da existência do elemento neutro: $u.\lambda = \lambda.u = u$.

concatenação, produto

A **potência** de ordem n, por conseguinte da concatenação, é, com $n \geq 0$, a concatenação de n réplicas de u, ou seja, $uuu...u$ n vezes, para uma representação u^n. Note-se que $u^0 = \lambda$, qualquer que seja o u.

potência

A própria palavra pode ser manipulada entre si. Assim denominamos de **prefixo** de uma palavra u uma sequência com parte dos símbolos iniciais de u e **sufixo**, por outro lado, uma sequência de parte dos símbolos finais de u. Tanto o prefixo como o sufixo são **sub-palavras**, isto é, sequências com parte dos símbolos de u. Se pretendermos a palavra com a ordem inversa dos símbolos então procuramos o **reverso** representado por u^R. Assim, se $u_1u_2...u_n$, então $u^R = u_n...u_2u_1$.

prefixo
sufixo
sub-palavras
reverso

Não sendo propriamente uma operação, se pretendermos obter o **número de ocorrências do símbolo x** na palavra u, usamos a função denotada por $\#(x, u)$.

número de ocorrências do símbolo x

Operações sobre linguagens

Da mesma forma que temos operações lógicas e num âmbito mais amplo temos operações sobre conjuntos, tendo operações sobre palavras podemos ter operações sobre linguagens. Assim, o conjunto das linguagens sobre um alfabeto A é **fechado** sobre as seguintes operações: reunião, interseção, diferença, complementação, concatenação, potenciação e fecho de Kleene.

fechado

⦿ Steven Kleene

A **reunião** de duas linguagens L_1 e L_2 denota-se por $L_1 \cup L_2$ e define-se por (1.1).

reunião

$$L_1 \cup L_2 = \{u : u \in L_1 \vee u \in L_2\} \quad (1.1)$$

Num Exercício 1.1 e considerando L_a = {u : u começa com "a"} = {aw : $w \in A^*$} e L_b = {u : u termina com "a"} = {wa : $w \in A^*$}, com $A = \{a, b, c\}$, pretende-se $L_a \cup L_b$. Assim, a reunião de L_a com L_b será $L_a \cup L_b$ = {u : u começa com "a" $\vee$ u termina com "a"} = {xwy : $w \in A^* \wedge (x = \text{a} \vee y = \text{a})$} $\cup$ {a}.

exercício 1.1

A **interseção** de duas linguagens L_1 e L_2 denota-se $L_1 \cap L_2$ e define-se por (1.2).

interseção

$$L_1 \cap L_2 = \{u : u \in L_1 \wedge u \in L_2\} \quad (1.2)$$

Num Exercício 1.2 e tomando as considerações do Exercício 1.1, pretende-se saber $L_a \cap L_b$. Assim, a interseção será $L_a \cap L_b$ = {u : u começa com "a" $\wedge$ u termina com "a"} = {awa : $w \in A^*$} $\cup$ {a}.

exercício 1.2

A **diferença** entre duas linguagens L_1 e L_2 denota-se L_1 - L_2 e define-se por (1.3).

diferença

$$L_1 - L_2 = \{u : u \in L_1 \wedge u \notin L_2\} \quad (1.3)$$

Num Exercício 1.3 e tomando as considerações do Exercício 1.1, pretende-se saber L_a - L_b. Assim, a diferença será L_a - L_b = {u : u começa por "a" ∧ u não termina com "a"}= = {awx : $w \in A^* \wedge x \in A \wedge x \neq$ a}.

exercício 1.3

A **complementação** da linguagem L_1 denota-se por L'_1 (ou com um traço superior) e define-se por (1.4).

complementação

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

(1.4)

Num Exercício 1.4 e tomando as considerações do Exercício 1.1 pretende-se a complementação de $L'_a = A^* - L_a = \{xw : w \in A^* \wedge x \notin L_a\}$ - {a}. Note-se que todas as operações de reunião, interseção, diferença e complementação gozam das **leis de DeMorgan**, como podemos ver em (1.5).

exercício 1.4

leis de DeMorgan

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

(1.5)

A **concatenação** de duas linguagens L_1 e L_2 denota-se $L_1.L_2$ e define-se por (1.6).

concatenação

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

(1.6)

Num Exercício 1.5 e tomando as considerações do Exercício 1.1 pretende-se $L_a.L_b$. Assim, $L_a.L_b$ = {uv : u começa com "a" ∧ v termina com "a"} = {awa : $w \in A^*$}.

exercício 1.5

Note-se que a concatenação goza das seguintes propriedades: associativa, existência do elemento neutro, existência do elemento absorvente, distributiva em relação à reunião e distributiva em relação à interseção.

A **potência** de ordem n da linguagem L denota-se por L^n e é definida indutivamente da forma representada em (1.7).

potência

$$L^0 = \{\lambda\} \quad ; \quad L^{n+1} = L^n.L$$

(1.7)

O **fecho de Kleene** da linguagem L denota-se por L^* e define-se da forma em (1.8).

fecho de Kleene

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

(1.8)

Para demonstrar o fecho de Kleene temos o Exercício 1.6 onde queremos o fecho de L_a.

exercício 1.6

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

(1.9)

Introdução ao conceito de gramática

Como já foi anteriormente referido, as linguagens têm um conjunto de palavras que quando são aplicadas estabelecem relações entre elas que são subjugadas a regras. A estas regras damos o nome de **gramática**. Consideremos então um alfabeto T = {a, b} e o conjunto de regras de (1.10).

gramática

$$S \rightarrow b$$
$$S \rightarrow aS \quad (1.10)$$

Tendo como Exercício 1.7, e tomando (1.10) em consideração, que palavras apenas constituídas por símbolos do alfabeto T se podem gerar a partir de S? Ora temos inicialmente que a regra S gera o símbolo "b". Como segunda regra temos que S gera, para um símbolo "a", uma nova aplicação de S. Em suma teríamos que $S \rightarrow \mathrm{a}^n\mathrm{b}$, com $n \geq 0$.

exercício 1.7

Podemos, depois, escrever a linguagem como $L = \{\mathrm{a}^n\mathrm{b} : n \geq 0\}$ ou como expressão regular da forma a*b, que significa "todos os símbolos anteriores a b, sendo a com b". Iremos verificar isto mais adiante no nosso estudo.

Em termos mais formais, uma gramática é um quádruplo da forma $G = (T, N, P, S)$, onde T é um conjunto finito não-vazio de símbolos terminais, N (sendo $N \cap T \neq \emptyset$) é um conjunto finito não-vazio de símbolos não terminais, P é um conjunto de produções (ou regras de escrita, cada uma da forma $\alpha \rightarrow \beta$) e S é o símbolo inicial, que pertence a N. Temos assim que $\alpha \in (N \cup T)^* N (N \cup T)$ e que $\beta \in (N \cup T)^*$. Por aqui, α e β são **restrições** e definem as gramáticas, por uma **hierarquia de Chomsky**.

restrições, hierarquia de Chomsky

As gramáticas podem ter diversos tipos, em suma, visíveis na Figura 1.2.

tipo	restrição
tipo 0	sem restrições
tipo 1 (dependente do contexto)	$\|\alpha\| \geq \|\beta\|$
tipo 2 (independente do contexto)	$\alpha \in N \wedge \beta \in (T \cup N)^*$
tipo 3 (regulares)	$\alpha \in N \wedge \beta \in T^* \cup T^*N$
tipo 4 (regulares)	$\alpha \in N \wedge \beta \in T^* \cup N^*T$

figura 1.2
tipos de gramáticas

Num Exercício 1.8 tentemos, sobre o alfabeto $T = \{\mathrm{a}, \mathrm{b}\}$ obter a gramática que representa a linguagem $L = \{\mathrm{a}^n\mathrm{b}^m : n \geq m\}$. Esta linguagem tem como lógica ter o lado esquerdo com n símbolos "a" e m símbolos "b" à direita. Primeiramente, podemos verificar que sempre que temos um "a", temos um "b" à direita: como primeira regra podemos colocar $X \rightarrow \mathrm{a}\ X\ \mathrm{b}$. Mas como $n \geq m$, então significa que o símbolo inicial terá um "a", sendo $S \rightarrow \mathrm{a}S$. Assim, podemos representar a gramática em (1.11).

exercício 1.8

$$S \rightarrow aS$$
$$S \rightarrow X$$
$$X \rightarrow aXb$$
$$X \rightarrow \varepsilon \quad (1.11)$$

Num exercício seguinte (Exercício 1.9) podemos tentar fazer o contrário, obtendo uma gramática para $L = \{\mathrm{a}^n\mathrm{b}^m : n \leq m\}$. Sendo a diferença $n \leq m$, temos que agora iremos ter o símbolos "b" em excesso, em comparação com os "a". Assim, podemos escrever a gramática como em (1.12).

exercício 1.9

$$S \rightarrow aSb$$
$$S \rightarrow X$$
$$X \rightarrow bX$$
$$X \rightarrow \varepsilon \quad (1.12)$$

2. Linguagens Regulares e Expressões Regulares

No processo de criação de um compilador a primeira fase de processamento corresponde à **análise lexical** onde se pretende traduzir ou identificar propriedades e outros constituintes de uma determinada linguagem. Em termos mais formais, uma análise lexical é uma fase de processamento onde se convertem sequências de carateres em **lexemas**

análise lexical

lexemas

(também denominados de **tokens**). Um lexema é um tuplo que contém um nome e um valor atribuído. Este nome é um símbolo abstrato que representa um determinado tipo de input e o valor atribuído é o valor passado e levado ao elemento em causa. Por exemplo, o seguinte código em C pos = pos + vel * 5 pode ser reescrito com <id, pos> <=,> <id, pos> <+,> <id, pos> <*,> <int,5>.

tokens

Linguagens regulares

Tendo já estudado o âmbito geral das linguagens e suas operações, uma **linguagem regular** é, apenas: um conjunto vazio sobre o alfabeto A; qualquer que seja o a, $\{a\}$; a união de duas linguagens regulares; a concatenação de duas linguagens regulares; o fecho de uma linguagem regular. Note-se que $\{\varepsilon\}$ é uma linguagem regular (LR), pelo que $\{\varepsilon\} = = \emptyset$.

linguagem regular

Uma possível representação de linguagens regulares acontece através das **expressões regulares**. De forma indutora, uma expressão regular sobre o alfabeto A é, apenas: () que representa a linguagem {}; qualquer que seja o $a \in A^*$, "a" que representa $\{a\}$; se e_1 e e_2 representam as linguagens regulares L_1 e L_2, então $(e_1|e_2)$ representa $L_1 \cup L_2$; se e_1 e e_2 representam as linguagens regulares L_1 e L_2, então $(e_1.e_2)$ representa $L_1.L_2$; se e_1 é uma expressão regular representando L_1, então $e_1{*}$ é representação $(L_1){*}$.

expressões regulares

Num Exercício 2.1 tentemos então obter uma expressão regular que represente o conjunto dos números binários que começam com 1 e terminam com 0. Para tal escrevemos `1(0|1)*0`, podendo tal ser interpretado como números que comecem com 1, tenham combinações (*) de 1 ou 0 (1|0) e um 0 no fim.

exercício 2.1

Num Exercício 2.2 tentemos, por outro lado, criar uma expressão regular que represente palavras cujos "b" tenham sempre um "a" à sua esquerda e um "c" à sua direita, considerando o alfabeto A = {a, b, c}. Para tal temos `(abc|a|c)*`, interpretando como todas as combinações de "abc" (onde o "a" está sempre à esquerda do "b" e o "c" à direita), "a" e "c".

exercício 2.2

Num último Exercício 2.3 criemos uma expressão regular que represente as sequências binárias com um número par de zeros. Assim, podemos ter `(1|01*01)*` ou `1*(01*01*)*`.

exercício 2.3

As expressões regulares usam duas grandes operações que gozam de propriedades algébricas. A operação de **escolha** (|) goza das propriedades: comutativa; associativa; de existência de elemento neutro; e de idempotência. Já a operação de **concatenação** (.) goza das propriedades: associativa; da existência de elemento neutro; e da existência de elemento absorvente.

escolha
concatenação

Quando conjugamos as operações de escolha com a concatenação também se gozam propriedades de distributividade e a operação de fecho goza das seguintes propriedades:

- $(e^*)^* = e^*$;
- $(e_1{}^*|e_2{}^*)^* = (e_1|e_2)^*$
- $(e_1{}^*.e_2{}^*)^* = (e_1.e_2)^*$

Note-se que o fecho da escolha não é a escolha dos fechos, tal como o fecho da concatenação não é a concatenação dos fechos!

nota

As operações têm **precedências** designadas. O fecho (*) tem precedência à concatenação (.) que, por si, tem precedência à escolha (|). Para alterar as precedências podemos usar parênteses.

precedências

Num Exercício 2.4 pretende-se que, sobre um alfabeto $A = \{0,1\}$ se construa uma expressão que reconheça a linguagem $L = \{w \in A^* : \#(0, w) = 2\}$. Para tal, `1*01*01*`.

exercício 2.4

Num Exercício 2.5 construa-se a expressão regular sobre o alfabeto A = {a, b, c, ..., z} tal que reconheça $L = \{w \in A^* : \#(\text{a}, w) = 3\}$. Para tal `(b|c|...|z)*a(b|c|...|z)*a(b|c|...|z)*`.

exercício 2.5

Para simplificar tais expressões temos **extensões notacionais**. Veja-se a tabela da Figura 2.1.

extensões notacionais

figura 2.1
extensões notacionais

extensão regular	significado
$e^+ = e.e*$	uma ou mais ocorrências
$e? = (e \mid \varepsilon)$	uma ou nenhuma ocorrências
$[a_1a_2a_3...a_n] = (a_1 \mid a_2 \mid ... \mid a_n)$	um símbolo do sub-alfabeto dado
$[a_1\text{-}a_n] = (a_1 \mid ... \mid a_n)$	um símbolo do sub-alfabeto em sequência
$[\hat{}a_1a_2a_3...a_n]$	um símbolo do alfabeto fora do dado
$[\hat{}a_1\text{-}a_n]$	um símbolo do alfabeto fora do dado
`e{n} = e.e...e n vezes`	n ocorrências de e
$e\{n_1, n_2\} = e.e...e$ n_1 a n_2 vezes	de n_1 a n_2 ocorrências de e
`e{n,} = e.e...e n, vezes`	n ou mais ocorrências

Note-se que não existem negações de expressões regulares!

Outras expressões regulares especiais frequentes são algumas da tabela da Figura 2.2.

figura 2.2

expressão	significado
`.`	qualquer símbolo diferente de \n
`.*`	linha completa
`^`	palavra vazia no início da linha
`$`	palavra vazia no fim da linha (antes de \n)
`^.*$`	linha completa sem \n
`\<`	palavra vazia no início de palavra
`\>`	palavra vazia no fim de palavra

Gramáticas regulares

Tendo já estudado o conceito de gramática, uma gramática G diz-se **regular** se, para qualquer produção $(\alpha \to \beta) \in P$, as duas condições de (2.1) forem satisfeitas.

regular

$$\alpha \in N$$
$$\beta \in T^* \cup T^* N \tag{2.1}$$

A linguagem que se gera de uma gramática regular é uma linguagem regular, pelo que é possível converter uma gramática numa expressão regular que represente a mesma linguagem e vice-versa. As gramáticas regulares são fechadas sob as operações de reunião, concatenação, fecho, intersecção e complementação. Assim sendo vejamos como aplicar cada uma destas propriedades. Sejam então $G_1 = (T_1, N_1, P_1, S_1)$ e $G_2 = (T_2, N_2, P_2, S_2)$ duas gramáticas regulares quaisquer, com $N_1 \cap N_2 = \varnothing$. A gramática $G = (T, N, P, S)$ onde (2.2) se verifica é regular e gera a linguagem $L = L(G_1) \cup L(G_2)$.

$$\begin{aligned} &T = T_1 \cup T_2 \\ &N = N_1 \cup N_2 \cup \{S\} \qquad \text{com } S \notin (N_1 \cup N_2) \\ &P = \{S \to S_1,\ S \to S_2\} \cup P_1 \cup P_2 \end{aligned} \tag{2.2}$$

A gramática $G = (T, N, P, S)$ onde (2.3) se verifica é regular e gera a linguagem $L = L(G_1) \cdot L(G_2)$.

A gramática $G = (T, N, P, S)$ onde (2.4) se verifica é regular e gera a linguagem $L = (L(G_1))^*$.

$$
\begin{aligned}
T &= T_1 \cup T_2 \\
N &= N_1 \cup N_2 \\
P &= \{A \to \omega S_2 : (A \to \omega) \in P_1 \wedge \omega \in T_1{}^*\} \\
&\quad \cup \{A \to \omega : (A \to \omega) \in P_1 \wedge \omega \in T_1{}^* N_1\} \\
&\quad \cup P_2 \\
S &= S_1
\end{aligned} \tag{2.3}
$$

$$
\begin{aligned}
T &= T_1 \\
N &= N_1 \cup \{S\} \qquad \text{com } S \notin N_1 \\
P &= \{S \to \varepsilon,\, S \to S_1\} \\
&\quad \cup \{A \to \omega S : (A \to \omega) \in P_1 \wedge \omega \in T_1{}^*\} \\
&\quad \cup \{A \to \omega : (A \to \omega) \in P_1 \wedge \omega \in T_1{}^* N_1\}
\end{aligned} \tag{2.4}
$$

Note-se que em (2.3), as produções da primeira gramática do tipo $\beta \in T^*$ ganham o símbolo inicial da segunda gramática no fim, tal como as produções da primeira gramática do tipo $\beta \in T^*N$ mantêm-se inalteradas, ficando também as produções da segunda inalteradas. Por outro lado, em (2.4), as produções que terminam num não-terminal mantêm-se inalteradas, ganhando o símbolo inicial no fim apenas aquelas que só têm terminais. As novas produções $S \to \varepsilon$ e $S \to S_1$ garante que $(L(G_1))^n \subseteq L(G)$, $\forall n \geq 0$.

Consideremos assim o Exercício 2.6 onde sobre o conjunto de terminais $T = \{a, b, c\}$ se pretende determinar uma gramática regular que represente a linguagem $L = L_1 \cup L_2$, sabendo que $L_1 = \{a\omega : \omega \in T^*\}$ e $L_2 = \{\omega a : \omega \in T^*\}$. Para resolvermos este tipo de exercício devemos começar por obter as gramáticas regulares que representam ambas linguagens. Assim, obtemos primeiramente (2.5).

exercício 2.6

$$
\begin{aligned}
&S_1 \to a\ X_1 \\
&X_1 \to a\ X_1 \\
&X_1 \to b\ X_1 \\
&X_1 \to c\ X_1 \\
&X_1 \to \varepsilon \\
\\
&S_2 \to a\ S_2 \\
&S_2 \to b\ S_2 \\
&S_2 \to c\ S_2 \\
&S_2 \to a
\end{aligned} \tag{2.5}
$$

Tendo as construções de (2.5) podemos agora resumi-las para (2.6).

$$
\begin{aligned}
&S \to S_1 \mid S_2 \\
&S_1 \to a\ X_1 \\
&X_1 \to a\ X_1 \mid b\ X_1 \mid c\ X_1 \\
&X_1 \to \varepsilon \\
&S_2 \to a\ S_2 \mid b\ S_2 \mid c\ S_2 \\
&S_2 \to a
\end{aligned} \tag{2.6}
$$

Num Exercício 2.7 podemos, sobre o mesmo conjunto de terminais que no Exercício 2.1, determinar uma gramática regular que represente a linguagem $L = L_1 \cdot L_2$, sabendo que $L_1 = \{a\omega : \omega \in T^*\}$ e $L_2 = \{\omega a : \omega \in T^*\}$. Mais uma vez, começemos por gerar as combinações necessárias para que as construções possam ser feitas de acordo com os detalhes da linguagem. Veja-se assim (2.7).

exercício 2.7

$$
\begin{aligned}
&S_1 \rightarrow \mathrm{a}\ X_1\\
&X_1 \rightarrow \mathrm{a}\ X_1\\
&X_1 \rightarrow \mathrm{b}\ X_1\\
&X_1 \rightarrow \mathrm{c}\ X_1\\
&X_1 \rightarrow \varepsilon\\
\\
&S_2 \rightarrow \mathrm{a}\ S_2\\
&S_2 \rightarrow \mathrm{b}\ S_2\\
&S_2 \rightarrow \mathrm{c}\ S_2\\
&S_2 \rightarrow \mathrm{a}
\end{aligned}
\tag{2.7}
$$

Tendo as construções de (2.7) podemos agora resumi-las para (2.8).

$$
\begin{aligned}
&S_1 \rightarrow \mathrm{a}\ X_1\\
&X_1 \rightarrow \mathrm{a}\ X_1 \mid \mathrm{b}\ X_1 \mid \mathrm{c}\ X_1\\
&X_1 \rightarrow S_2\\
&S_2 \rightarrow \mathrm{a}\ S_2 \mid \mathrm{b}\ S_2 \mid \mathrm{c}\ S_2\\
&S_2 \rightarrow \mathrm{a}
\end{aligned}
\tag{2.8}
$$

exercício 2.8

Finalmente, vejamos o Exercício 2.8 onde, analogamente aos exercícios 2.1 e 2.2 pretendemos determinar uma gramática regular que represente a linguagem $L = L_1{}^*$, sobre o conjunto de terminais $T = \{\mathrm{a, b, c}\}$, sabendo que $L_1 = \{\mathrm{a}\omega : \omega \in T^*\}$. Começamos de novo por obter a gramática regular que representa L_1, em (2.9).

$$
\begin{aligned}
&S_1 \rightarrow \mathrm{a}\ X_1\\
&X_1 \rightarrow \mathrm{a}\ X_1\\
&X_1 \rightarrow \mathrm{b}\ X_1\\
&X_1 \rightarrow \mathrm{c}\ X_1\\
&X_1 \rightarrow \varepsilon
\end{aligned}
\tag{2.9}
$$

Agora, tendo as construções de (2.9) podemos resumi-las para (2.10).

$$
\begin{aligned}
&S \rightarrow \varepsilon \mid S_1\\
&S_1 \rightarrow \mathrm{a}\ X_1\\
&X_1 \rightarrow \mathrm{a}\ X_1 \mid \mathrm{b}\ X_1 \mid \mathrm{c}\ X_1\\
&X_1 \rightarrow S
\end{aligned}
\tag{2.10}
$$

Conversão de expressões regulares em gramáticas regulares

representações
converter

Tanto as expressões regulares como as gramáticas regulares são duas formas de exibirmos as capacidades e propriedades de uma linguagem: são **representações** de uma linguagem. Sendo que ambas são equivalentes, como fazer para **converter** uma representação na outra? Para converter uma expressão regular numa gramática regular basta obter gramáticas regulares para as expressões regulares primitivas e aplicar as operações regulares sobre gramáticas. A gramática regular para a expressão ε é dada claramente por $S \rightarrow \varepsilon$. Já para um terminal a, qualquer que seja a, é dada por $S \rightarrow a$.

Por outro lado, para converter uma gramática regular numa expressão regular, seja $G = (T, N, P, S)$ uma gramática regular qualquer. Uma expressão regular que represente a mesma linguagem que a gramática G pode ser obtida por um processo de transformação de equivalência. Primeiro converte-se a gramática G no conjunto de triplos de (2.11).

$$\varepsilon = \{(E, \varepsilon, S)\} \cup \{(A, \omega, B) : (A \to \omega B) \in P \wedge B \in N\} \cup \{(A, \omega, \varepsilon) : (A \to \omega) \in P \wedge \omega \in T^*\} \qquad \text{com } E \notin N \tag{2.11}$$

Depois de completar os triplos de (2.11) removem-se, por transformações de equivalência, um a um, todos os símbolos de N, até se obter um único triplo da forma (E, e, ε). Assim, para cada símbolo $B \in N$:

- substituir todos os triplos da forma (A, β_i, B) por um único (A, ω_1, B), onde $\omega_1 = \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$;
- substituir todos os triplos da forma (B, α_i, B) por um único (B, ω_2, B), onde $\omega_2 = \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m$;
- substituir todos os triplos da forma (B, γ_i, C) por um único (B, ω_3, C), onde $\omega_3 = \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$;
- substituir o triplo de triplos $((A, \omega_1, B), (B, \omega_2, B), (B, \omega_3, C))$ pelo triplo $(A, \omega_1\omega_2{}^*\omega_3, C)$.

Num Exercício 2.9 pretendemos obter uma expressão regular equivalente à gramática regular de (2.12). **exercício 2.9**

$$S \to \text{a } S \mid \text{b } S \mid \text{c } S \mid \text{aba } X \qquad X \to \text{a } X \mid \text{b } X \mid \text{c } X \mid \varepsilon \tag{2.12}$$

Para resolvermos este tipo de exercício vejamos (2.13), onde aplicamos as regras de substituição referidas anteriormente.

$$\begin{aligned} \varepsilon &= \{(E,\varepsilon,S), (S,\text{a},S), (S,\text{b},S), (S,\text{c},S), (S,\text{aba},S), (X,\text{a},X), (X,\text{b},X), (X,\text{c},X), (X,\varepsilon,\varepsilon)\} \\ &= \{(E,\varepsilon,S), (S,\text{a}|\text{b}|\text{c},S), (S,\text{aba},S), (X,\text{a},X), (X,\text{b},X), (X,\text{c},X), (X,\varepsilon,\varepsilon)\} \\ &= \{(E,(\text{a}|\text{b}|\text{c})^*\text{aba},X), (X,\text{a},X), (X,\text{b},X), (X,\text{c},X), (X,\varepsilon,\varepsilon)\} \\ &= \{(E,(\text{a}|\text{b}|\text{c})^*\text{aba},X), (X,\text{a}|\text{b}|\text{c},X), (X,\varepsilon,\varepsilon)\} \\ &= \{(E,(\text{a}|\text{b}|\text{c})^*\text{aba}(\text{a}|\text{b}|\text{c})^*, \varepsilon)\} \end{aligned} \tag{2.13}$$

3. Autómatos Finitos

Como já vimos existem várias formas de representar uma linguagem. Tendo estudado as expressões regulares e as gramáticas agora podemos abordar uma terceira, mais visual e fácil de compreender: o **autómato**. Um autómato, primeiramente, **finito**, é um mecanismo reconhecedor das palavras de uma linguagem. Dada uma linguagem L, definida sobre um alfabeto A, um autómato finito reconhecedor de L é um mecanismo que reconhece as palavras de A^* que pertencem a L. Genericamente um autómato finito tem a configuração representada na Figura 3.1. **autómato, finito**

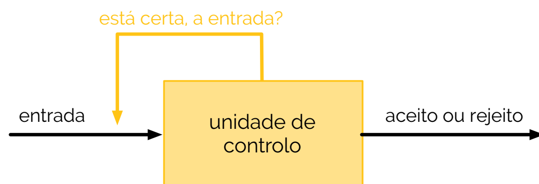


figura 3.1

Um autómato não passa assim de uma máquina de estados, finita, se o primeiro também o for. O canal de entrada é uma unidade só de leitura com acesso sequencial ou aleatório aos símbolos da palavra. No acesso sequencial um símbolo da palavra só pode ser

lido e processado à unidade, não podendo ser lido novamente. Se assumirmos que este canal possui uma cabeça de leitura, esta apenas poderá avançar uma posição de cada vez. Em geral a resposta dos autómatos é do tipo *sim/não* ou *aceito/rejeito*. Quer isto dizer que um autómato não possui propriamente um canal de saída, mas antes uma produção de **aceitação** se a palavra u pertencer à linguagem L e uma **rejeição** caso contrário. Neste sentido funcionam efetivamente como reconhecedores das palavras de uma linguagem.

aceitação, rejeição

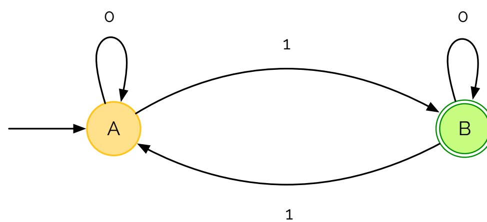
Autómatos finitos deterministas (AFD)

Um **autómato finito determinista** (vulgarmente abreviado para AFD) é um quíntuplo $M = (A,\ Q,\ q_0,\ \delta,\ F)$, em que A é o alfabeto de entrada, Q é um conjunto finito não-vazio de estados, q_0 é o estado inicial (que pertence ao conjunto Q), δ é uma função que determina a transição entre estados (com o detalhe de que $\delta : Q \times A \rightarrow Q$) e F é o conjunto dos estados de aceitação (subconjunto de Q).

autómato finito determinista

Graficamente um autómato finito determinista é representado por um grafo. Os vértices correspondentes aos estados de aceitação estarão representados por duplo contorno e por uma cor verde, enquanto que os de rejeição absoluta estarão representados a vermelho. O estado inicial será sempre marcado por uma seta sem origem e os arcos estão etiquetados com elementos de A. Consideremos assim o autómato M_1 representado na Figura 3.2.

figura 3.2

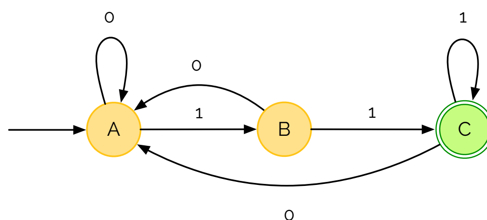


Se experimentarmos executar o nosso autómato M_1 com algumas palavras podemos testar se essas pertencem à linguagem que representa, verificando se o estado final é de aceitação ou não. Por exemplo, se executarmos a palavra 1011 esta é aceite pelo autómato dado que com a leitura sequencial com '1' avança para B, com '0' fica em B, avança para A com '1' e volta a B com '1', sendo aceite. Por outro lado, colocar 1100 na entrada leva a uma rejeição, tal como a palavra vazia λ, ficando por A.

Num Exercício 3.1 vejamos então que palavras são aceites pelo autómato da Figura 3.3.

exercício 3.1

figura 3.3



Se verificarmos bem, para que uma palavra entre no estado de aceitação C, do autómato da Figura 3.3, é necessário que esta acabe com um '1' ou mais, pelo que, as palavras binárias reconhecidas são apenas aquelas que terminam em 11 (dois '1' porque precisa de um '1', pelo menos, para transitar entre o estado A e B e outro para B e C.

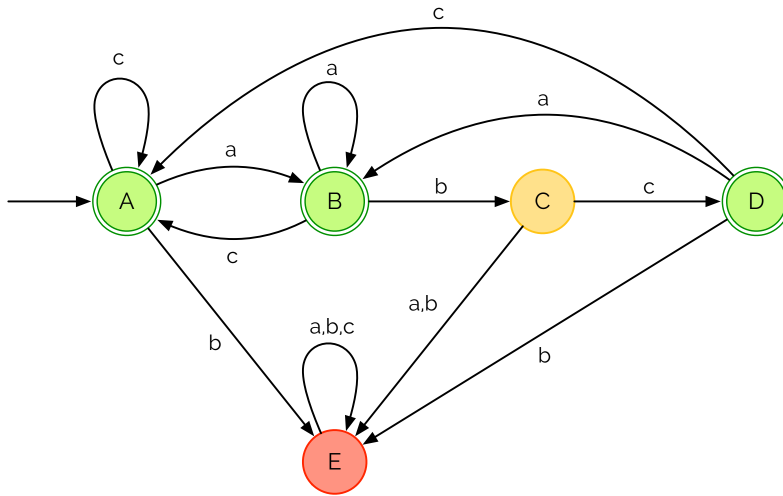
Num Exercício 3.2 podemos tentar representar o autómato da Figura 3.3 de forma textual, isto é, descrevendo cada um dos componentes do quíntuplo. Para fazer isto podemos começar por designar o alfabeto A como sendo $\{0,1\}$ e os seus estados $\{A,\ B,\ C\}$ (tal como o estado inicial q_0 e F como sendo $\{C\}$). Já δ será o conjunto de transições possíveis do autómato $\{(A,0,A),\ (A,1,B),\ (B,0,A),\ (B,1,C),\ (C,0,A),\ (C,1,C)\}$.

exercício 3.2

Até agora os exercícios propostos vão muito ao encontro de interpretar o conteúdo de um autómato. Se fizermos a questão ao contrário, como no Exercício 3.3, podemos ter de criar um autómato finito determinista que reconheça as sequências definidas sobre o alfabeto $A = \{a, b, c\}$ que satisfaçam o requisito de qualquer 'b' ter um 'a' imediatamente à sua esquerda e um 'c' imediatamente à sua direita. Para começar tentemos avaliar a palavra vazia λ: esta palavra, não tendo qualquer elemento satisfaz o requisito. Isto significa à partida que o estado inicial é estado de aceitação. Consideremos este estado como A. Se inserirmos um 'a' a uma palavra vazia continuamos a ter uma aceitação por parte do autómato, mas se inserirmos um 'b' automaticamente ficamos rejeitados, porque para ter um 'b' é necessário ter um 'a' à esquerda. Criamos assim um estado de rejeição absoluta (também denominado de estado de morte), a que vamos dar o nome de E. Tendo um 'a' ou um 'c' podemos inserir um 'b', mas se de seguida inserirmos outra letra que não um 'c' entramos novamente no estado E. Veja-se assim o autómato de Figura 3.4.

exercício 3.3

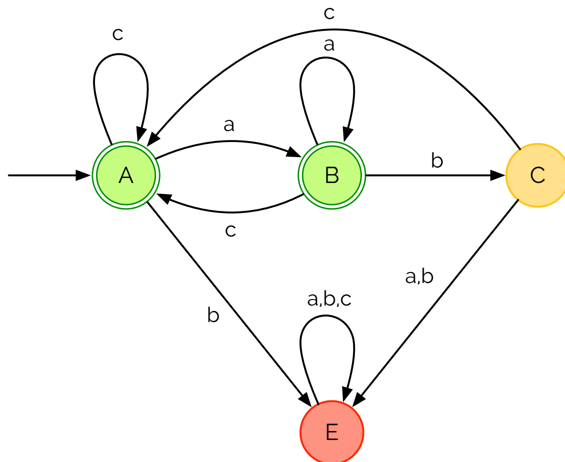
figura 3.4



Quando olhamos para a Figura 3.4 muito provavelmente estranhamos a quantidade de estados de aceitação em comparação com o número total de estados, tal como algumas transições que parecem estar a ser feitas "a mais". Por exemplo, se compararmos os estados A e D podemos ver que ambos são equivalentes, isto é, permanecem em aceitação quando recebem um 'c', passam para B quando recebem um 'a' e entram em rejeição total quando recebem um 'b'. Assim sendo podemos **reduzir** o nosso autómato ao da Figura 3.5, que resulta na fusão das transições para D com as de A.

reduzir

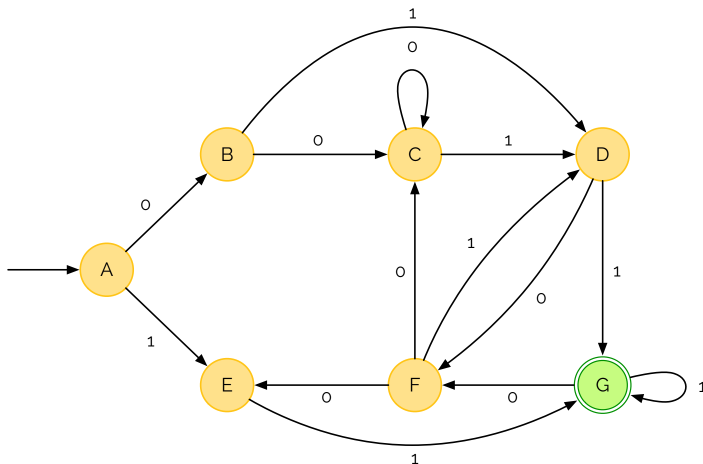
figura 3.5



Tentemos então, identificar estados redundantes e equivalentes e reduzir um autómato no Exercício 3.4, com o autómato da Figura 3.6.

exercício 3.4

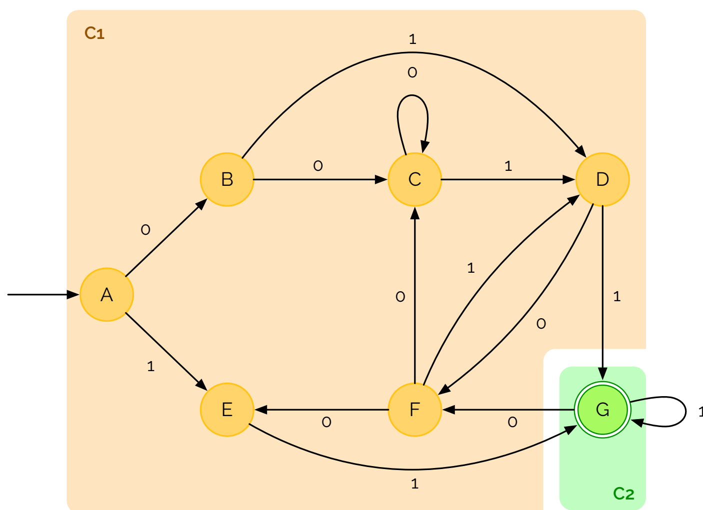
figura 3.6



Para respondermos sucintamente a um exercício deste tipo há que saber como **classificar** cada vértice do grafo. Assim sendo, primeiro devemos dividir os estados em aceitação e não-aceitação, obtendo duas classes, como podemos ver na Figura 3.7 - a classe $C_1 = \{A,\ B,\ C,\ D,\ E,\ F\}$ e a $C_2 = \{G\}$.

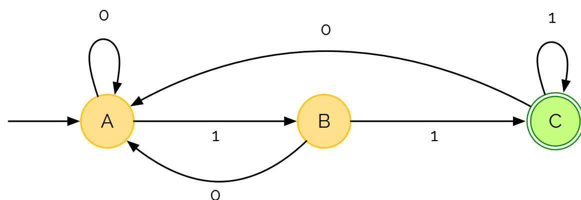
classificar

figura 3.7



Dividindo agora a classe C_1 em dois grupos $\{A,\ B,\ C,\ F\}$ e $\{D,\ E\}$ podemos considerar as três classes como três vértices de um grafo menor, representado na Figura 3.8.

figura 3.8



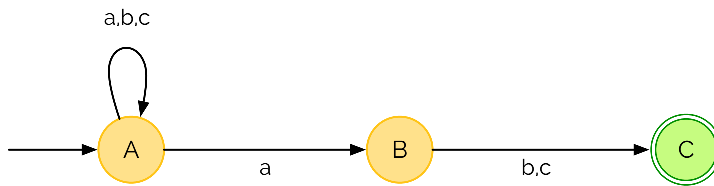
Autómatos finitos não-deterministas (AFND)

Consideremos o seguinte autómato da Figura 3.9 sobre o alfabeto {a, b, c}. Define-se tal autómato como **autómato finito não-determinista** (AFND) se possui as seguintes caraterísticas:

autómato finito não-determinista

- as transições estão associadas a símbolos individuais do alfabeto ou a ε;
- de cada estado saem zero ou mais transições por cada símbolo do alfabeto ou ε;
- há um estado inicial;
- há zero ou mais estados de aceitação, que determinam as palavras aceites;
- uma dada palavra sobre o alfabeto faz o sistema avançar do estado inicial a zero ou mais estados finais, determinando estes a aceitação ou a rejeição da palavra.

figura 3.9



No caso do grafo da Figura 3.9 não há nenhum arco etiquetado com 'a' a sair de B e basta isso para garantir que este autómato é não-determinista.

Como consequência de ser não-determinista, um autómato destes gera várias **alternativas** para caminhos de processamento da entrada. Consideremos a palavra 'abab' - pertence à linguagem L, representada pelo autómato da Figura 3.9? Se analisarmos bem a nossa figura podemos ver que existem 3 caminhos possíveis, dos quais um não leva a uma aceitação:

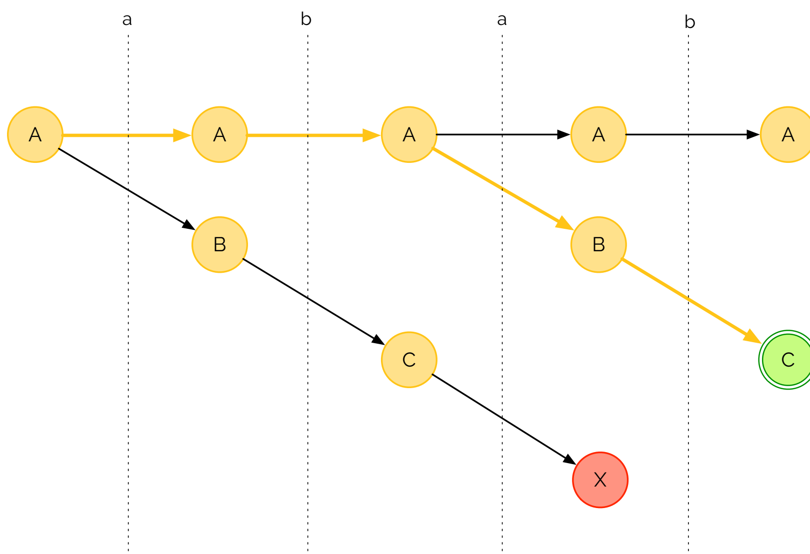
alternativas

- A —a→ B —b→ C —a→ X
- A —a→ A —b→ A —a→ B —b→ C
- A —a→ A —b→ A —a→ A —b→ A

Estes resultados podem ser refletidos numa **estrutura arbórea**, como podemos ver na Figura 3.10.

estrutura arbórea

figura 3.10



Vejamos agora um exemplo novo, mas cujo autómato contém transições provocadas por ε, na Figura 3.11.

Num Exercício 3.5 experimentemos averiguar se a palavra '1011' faz parte da linguagem descrita pelo autómato finito não-determinista da Figura 3.11. Para resolver vejamos se existe algum caminho para o estado de aceitação que é D. Basta haver um caminho para podermos responder que '1011' pertence à linguagem L, representada pelo autómato.

exercício 3.5

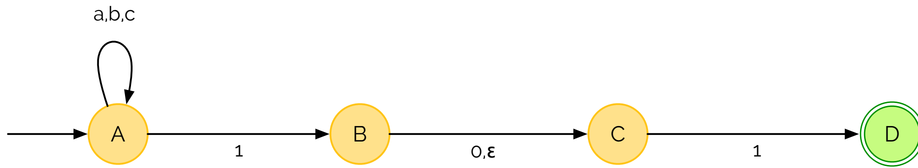


figura 3.11

Vejamos assim os vários caminhos para '1011':

- A —1→ A —0→ A —1→ A —1→ A
- A —1→ A —0→ A —1→ A —1→ B
- A —1→ A —0→ A —1→ A —1→ B —ε→ C
- A —1→ A —0→ A —1→ B —ε→ C —1→ D

Como o último caminho de todos chega a D (estado de aceitação), então temos que a palavras '1011' pertence à linguagem L. Vendo estes caminho representados em forma arbórea temos a Figura 3.12.

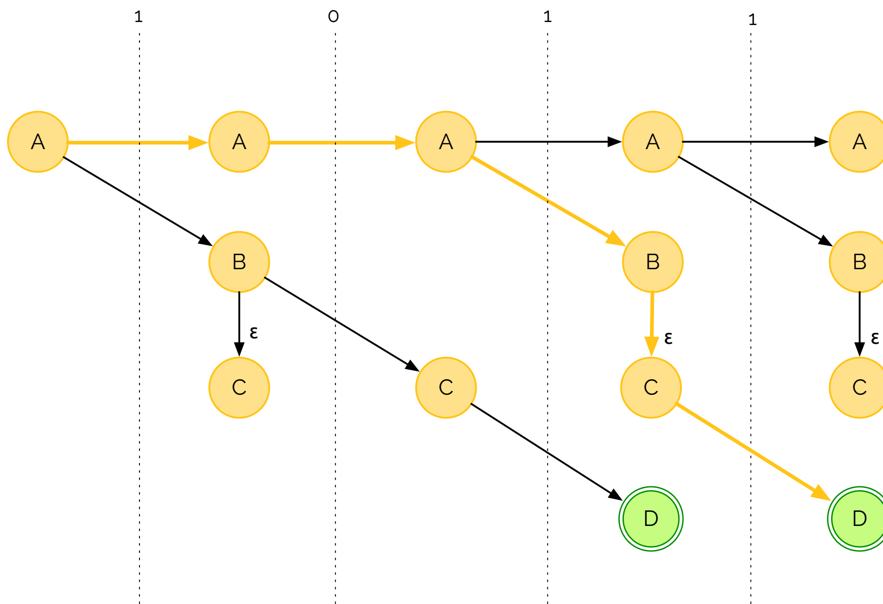


figura 3.12

Consideremos agora os seguintes exercícios:

- Exercício 3.6 - Que palavras são reconhecidas pelo autómato da Figura 3.9? Para resolver este exercício podemos partir do estado de aceitação. Para cumprir uma aceitação as palavras têm de terminar com um 'b' ou um 'c'. Mas do estado incial para o de aceitação, embora só haja um caminho, este não é direto, pelo que é necessário ter 1 e só 1 'a', precedido de qualquer carater do alfabeto {a, b, c} 0 ou mais vezes. Assim, a nossa linguagem será $L = \{\omega a X : \omega \in A^* \wedge X \in \in \{b,c\}\}$.

exercício 3.6

- Exercício 3.7 - Que palavras são reconhecidas pelo autómato da Figura 3.11? Para resolver este aplicamos um método análogo. Olhando incialmente para o fim (estado de aceitação) temos que qualquer palavra que chegue a este estado tem de terminar em '1'. Mas para chegar ao estado de aceitação precisa de passar pelo estado C onde terá de conter um '0' ou nada antes do '1', por sua vez, passando por B, onde terá de ter um '1'. Assim a nossa linguagem L é tal que $L = \{\omega \in (0,1)^* : \omega$ termina com 11 ou 101$\}$.

exercício 3.7

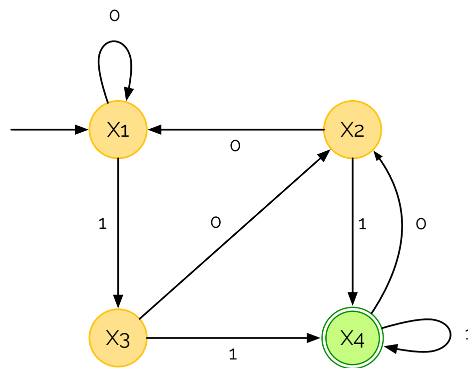
Equivalência entre autómatos finitos deterministas e não-deterministas

A classe de linguagens cobertas por autómatos finitos deterministas é a mesma que a classe das linguagens cobertas por não-deterministas. Mas como é que podemos determinar um não-determinista equivalente a um determinista dado e vice-versa? Pelas defincões algébricas de AFD e de AFND, um autómato finito determinista é um autómato finito não-determinista, pelo que Q, q_0 e F têm a mesma definição, sendo que nos deterministas $\delta : Q \times A \to Q$ e nos não-deterministas δ está contido em $Q \times A_\varepsilon \times Q$. Mas, se $\delta : Q \times A \to Q$ então temos que $\delta \subset Q \times A \times Q \subset Q \times A_\varepsilon \times Q$. Logo provamos que um autómato finito determinístico é um autómato finito não-determinístico.

Vejamos assim o Exercício 3.8 onde se pretende obter um autómato finito determinista equivalente ao não-determinista da Figura 3.11. Resumindo alguns conjuntos de estados a estados novos podemos obter o seguinte autómato da Figura 3.13.

exercício 3.8

figura 3.13



Os autómatos finitos são fechados sobre as operações vulgares de reunião, concatenação, fecho, interceção e complementação.

Conversão de uma expressão regular num AFND

Dada uma expressão regular qualquer, ela é: um elemento primitivo; uma expressão do tipo e^*, sendo e uma expressão regular qualquer; uma expressão do tipo e_1e_2, sendo e_1 e e_2 duas expressões regulares quaisquer; uma expressão do tipo $e_1|e_2$, sendo e_1 e e_2 duas expressões regulares quaisquer. Se se identificar os autómatos equivalentes das expressões primitivas, tem-se o problema da conversão de uma expressão regular para um autómato finito resolvido, visto que se sabe como fazer a reunião, a concatenação e o fecho dos autómatos.

Para converter elementos primitivos em autómatos podemos seguir o conteúdo da Figura 3.14, sendo que, na realidade, o automáto referente a ε pode ser obtido aplicando o fecho ao autómato de ().

figura 3.14

expressão regular	autómato finito
( )	
ε	
a	a

Basicamente, para converter uma expressão regular num autómato finito não-determinístico temos primeiro que, se a expressão regular é do tipo primitivo, o autómato correspondente pode ser obtido da tabela anterior. Continuando, se a expressão for do tipo $e*$ aplica-se este mesmo algoritmo na obtenção de um autómato equivalente à expressão regular e e, de seguida, aplica-se o fecho de autómatos. Caso não se verifique este caso então estamos perante o caso de uma expressão do tipo e_1e_2 ou do tipo $e_1|e_2$, aos quais se aplica este mesmo algoritmo para obter autómatos para as expressões e_1 e e_2 e, de seguida, aplica-se a concatenação ou a reunião, respetivamente, de autómatos.

Na realidade o algoritmo corresponde a um processo de decomposição arbórea a partir da raiz seguido de um processo de construção arbórea a partir das folhas.

Consideremos assim o Exercício 3.9 onde se pretende que se construa um autómato equivalente à expressão regular e = a|a(a|b|c)*a. Como foi dito o algoritmo de conversão inicia com uma decomposição arbórea. Se decompusermos a expressão numa árvore podemos ter o resultado da Figura 3.15.

exercício 3.9

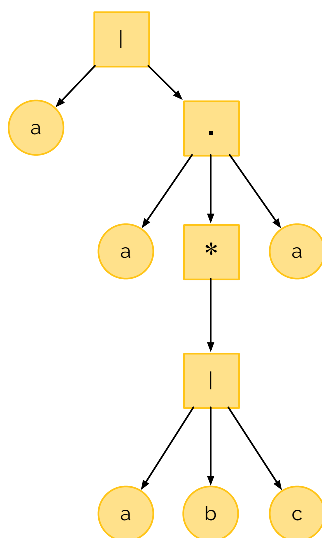


figura 3.15

Por cada expressão primitiva temos um autómato primitivo como o representado na Figura 3.16, onde x = {a, b, c}.

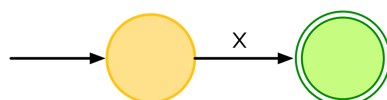


figura 3.16

Na decisão entre 'a', 'b' ou 'c' temos que implementar a reunião de autómatos como em Figura 3.17 (simplificação à direita).

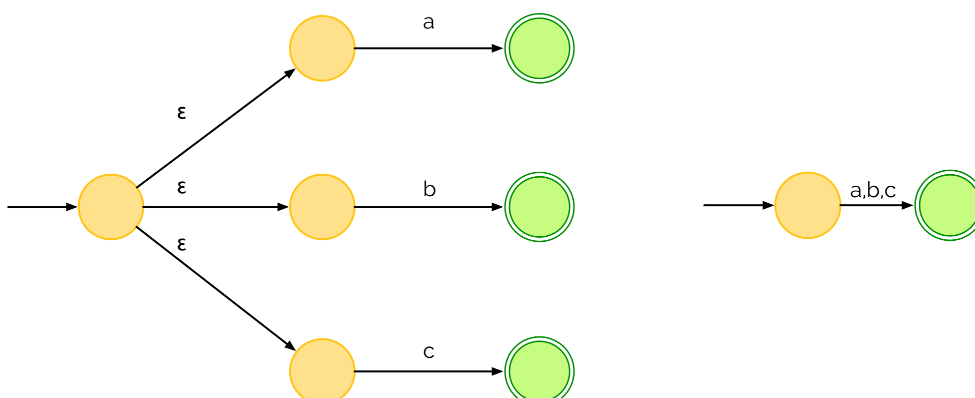
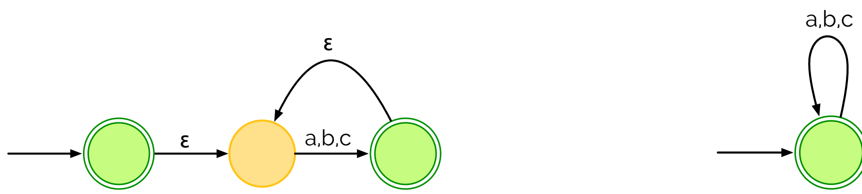


figura 3.17

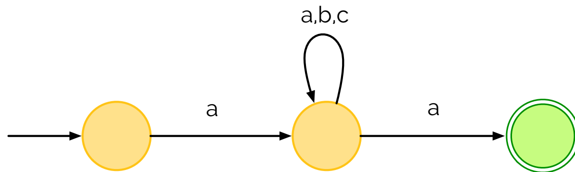
Dada a operação de fecho temos que fazer o fecho dos autómatos como vemos na Figura 3.18 (simplificação à direita).

figura 3.18



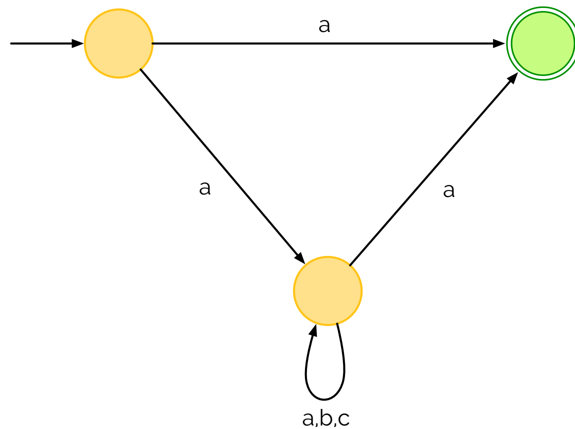
Terminando, para uma concatenação, efetuamos a concatenação dos autómatos (já com simplificação), na Figura 3.19.

figura 3.19



Para finalizar esta nossa conversão agora precisamos de anexar a decisão com 'a', que de forma simplificada fica como a Figura 3.20.

figura 3.20



Conversão de um autómato finito numa expressão regular

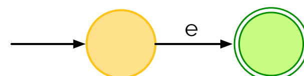
Então mas como é que podemos converter uma representação de autómato finito numa expressão regular? A conversão de autómatos finitos em expressões regulares baseia-se num novo tipo de autómatos denominados de **autómatos finitos generalizados** (AFG). Este tipo de autónomos é semelhante aos autómatos finitos não-deterministas mas em que as etiquetas dos arcos são expressões regulares.

autómatos finitos generalizados

Para converter temos então o seguinte **algoritmo** que se resume a uma transformação de um autómato num autómato finito generalizado reduzido (com a forma da Figura 3.21, onde e é uma expressão regular).

algoritmo

figura 3.21



A redução de um autómato finito generalizado pode ser feita aplicando o seguinte algoritmo: primeiro transformamos um AFG num outro autómato cujo estado inicial não tenha arcos de chegada; de seguida transformamos um AFG num outro autómato que só tenha um estado de aceitação, este, que não tenha arcos de saída; entretanto eliminamos estados um a um.

Consideremos assim novamente a Figura 3.2 para o Exercício 3.10, onde queremos aplicar o algoritmo anterior para o converter para uma expressão regular.

exercício 3.10

Para começarmos a resolução primeiro temos de transformar o nosso autómato num cujo estado inicial não tenha arcos de chegada (a não ser o inicial sem origem). Veja-se assim a Figura 3.22.

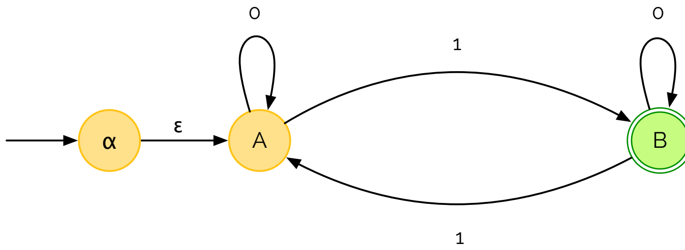


figura 3.22

De seguida devemos transformar o autómato num cujo estado de aceitação seja único e não tenha arcos de saída. Vejamos assim a Figura 3.23.

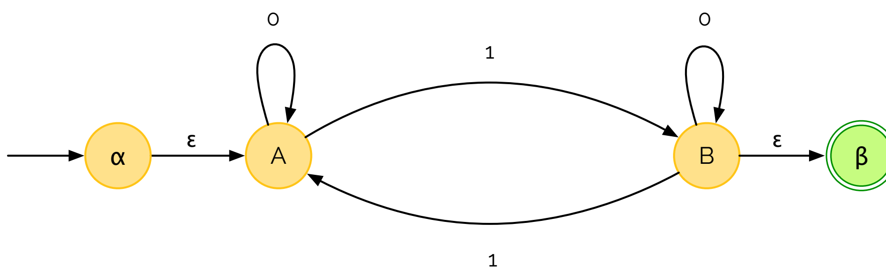


figura 3.23

Agora temos de eliminar os estados intermédios um a um, construindo a expressão regular final. Se removermos o vértice A temos que de α para B ficamos com 0*1 e de B para B 0|10*1. Cortando o estado B ficamos com a expressão final 0*1(0|10*1)*.

4. Gramáticas Independentes do Contexto (GIC)

Como já tivémos oportunidade de ver anteriormente, uma gramática é um quádruplo $G = (T, N, P, S)$ onde T é um conjunto finito não-vazio de símbolos terminais, N, sendo que $N \cap T = \emptyset$, é um conjunto não-vazio de símbolos não-terminais, P é um conjunto de produções (ou regras de rescrita), cada uma com a forma $\alpha \to \beta$ e $S \in N$ é o símbolo inicial. Por α e β pretendemos referir **cabeça de produção** e **corpo da produção**, respetivamente. No caso geral, veja-se (4.1).

cabeça de produção, corpo da produção

$$\alpha \in (T \cup N)^+$$
$$\beta \in (T \cup N)^* \tag{4.1}$$

Introdução às gramáticas independentes do contexto

Uma gramática diz-se **independente do contexto** se, para qualquer produção do conjunto de produções as duas condições de (4.2) se verificam.

independente do contexto

$$\alpha \in N$$
$$\beta \in (T \cup N)^* \tag{4.2}$$

A linguagem gerada por uma gramática independente do contexto diz-se independente do contexto, igualmente. Um exemplo de gramática desta génese são as gramáticas regulares. Estas são fechadas sob as operações de reunião, concatenação e fecho, mas não sob as operações de interseção ou complementação.

Derivação por regras

Dada uma palavra $\alpha A\beta$ e uma produção $A \to v$ damos o nome de **derivação direta** à rescrita de $\alpha A\beta$ em $\alpha v\beta$, denotando-se como se pode ver em (4.3).

derivação direta

$$\alpha A\beta \Rightarrow \alpha v\beta \quad (4.3)$$

Dada uma palavra $\alpha A\beta$ com $\beta \in T^*$ e uma produção $A \to v$ damos o nome de **derivação direta à direita** à rescrita de $\alpha A\beta$ em $\alpha v\beta$, denotando-se como se pode ver em (4.4).

derivação direta à direita

$$\alpha A\beta \overset{D}{\Rightarrow} \alpha v\beta \quad (4.4)$$

Dada uma palavra $\alpha A\beta$ com $\alpha \in T^*$ e uma produção $A \to v$ damos o nome de **derivação direta à esquerda** à rescrita de $\alpha A\beta$ em $\alpha v\beta$, denotando-se como se pode ver em (4.5).

derivação direta à esquerda

$$\alpha A\beta \overset{E}{\Rightarrow} \alpha v\beta \quad (4.5)$$

Chamamos assim de **derivação à direita** a uma sucessão de zero ou mais derivações diretas à direita, denotando-se (4.6)(a) e **derivação à esquerda** a uma sucessão de zero ou mais derivações diretas à esquerda, denotando-se (4.6)(b).

derivação à direita
derivação à esquerda

$$\alpha \overset{D}{\Rightarrow}{}^* \beta \qquad \text{(a)} \quad (4.6)$$
$$\alpha \overset{E}{\Rightarrow}{}^* \beta \qquad \text{(b)}$$

Uma forma de esquematizarmos todo o processamento de derivação é através de **árvores de derivação** (em inglês *parse trees*). Estas árvores são representações de uma derivação onde os nós-ramos são elementos de N e os nós-folhas são elementos de T. Consideremos assim a seguinte gramática de (4.7) sobre o alfabeto $T = \{a, b, c\}$ e, num Exercício 4.1 tentemos determinar as derivações à direita e à esquerda da palavra 'aabcbc' e desenhar a árvore de derivação da mesma palavra (Figura 4.1).

árvores de derivação

exercício 4.1

$$S \to \varepsilon \mid a\ B \mid b\ A \mid c\ S \quad (4.7)$$
$$A \to a\ S \mid b\ A\ A \mid c\ A$$
$$B \to a\ B\ B \mid b\ S \mid c\ B$$

Em termos de derivações à direita temos que a partir de S podemos começar por construir aB, seguido de aaBB e, como B pode ser bS, aa$B$bS, seguido de aa$B$bc$S$. Como S pode ser ε, então podemos terminar de derivar B para bS, seguido de ε, dando a palavra querida 'aabcbc' - veja-se melhor em (4.8).

$$S \Rightarrow \mathrm{a}B \Rightarrow \mathrm{aa}BB \Rightarrow \mathrm{aa}B\mathrm{b}S \Rightarrow \mathrm{aa}B\mathrm{bc}S \Rightarrow \mathrm{aa}B\mathrm{bc} \Rightarrow \mathrm{aab}S\mathrm{bc} \Rightarrow \mathrm{aabc}S\mathrm{bc} \Rightarrow \mathrm{aabcbc} \quad (4.8)$$

Em termos de derivações à esquerda temos que a partir de S podemos construir (4.9).

$$S \Rightarrow \mathrm{a}B \Rightarrow \mathrm{aa}BB \Rightarrow \mathrm{aab}SB \Rightarrow \mathrm{aabc}SB \Rightarrow \mathrm{aabc}B \Rightarrow \mathrm{aabcb}S \Rightarrow \mathrm{aabcbc}S \Rightarrow \mathrm{aabcbc} \quad (4.9)$$

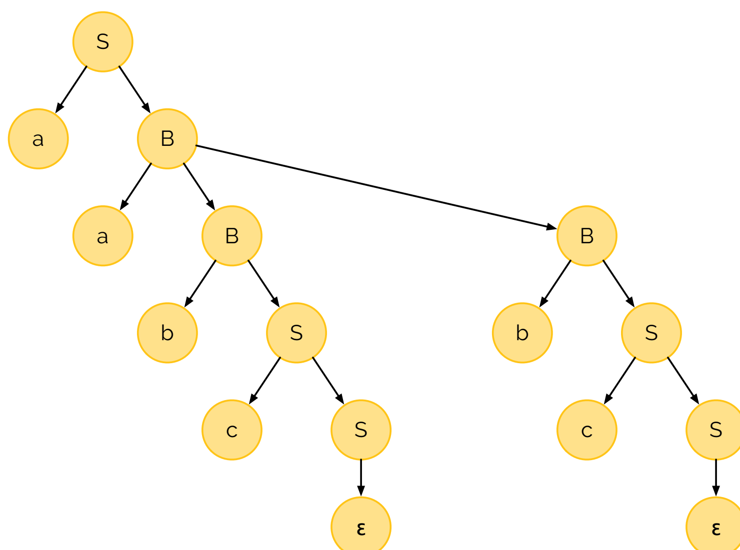


figura 4.1

Ambiguidade gramatical

Algumas das construções geradas, dadas uma gramática, podem gerar situações de indecisão na criação de árvores de derivação. Quando uma palavra possui duas ou mais árvores de derivação distintas diz-se que essa é derivada **ambiguamente**, sendo, por conseguinte, considerada, uma gramática, **ambígua**, se possuir pelo menos uma palavra gerada ambiguamente. Para resolver este problema poder-se-á, na grande parte das situações, criar uma gramática alternativa, não-ambígua, que gere a mesma linguagem que a ambígua.

ambiguamente
ambígua

Consideremos assim a gramática em (4.10) e comparemos as derivações à esquerda, diferentes, em (4.11)(a) e (4.11)(b) para a palavra '1+1.0'.

$$S \rightarrow S + S \mid S\ .\ S \mid \neg S \mid (\ S\ ) \mid 0 \mid 1 \tag{4.10}$$

$$S \Rightarrow S + S \Rightarrow 1 + S \Rightarrow 1 + S\ .\ S \Rightarrow 1 + 1\ .\ S \Rightarrow 1 + 1\ .\ 0 \qquad \text{(a)}$$
$$S \Rightarrow S\ .\ S \Rightarrow S + S\ .\ S \Rightarrow 1 + S\ .\ S \Rightarrow 1 + 1\ .\ S \Rightarrow 1 + 1\ .\ 0 \qquad \text{(b)} \tag{4.11}$$

Olhando para as derivações de (4.11) será que estas são equivalentes? Criemos assim as árvores de derivação para cada uma destas derivações. Veja-se assim a Figura 4.2.

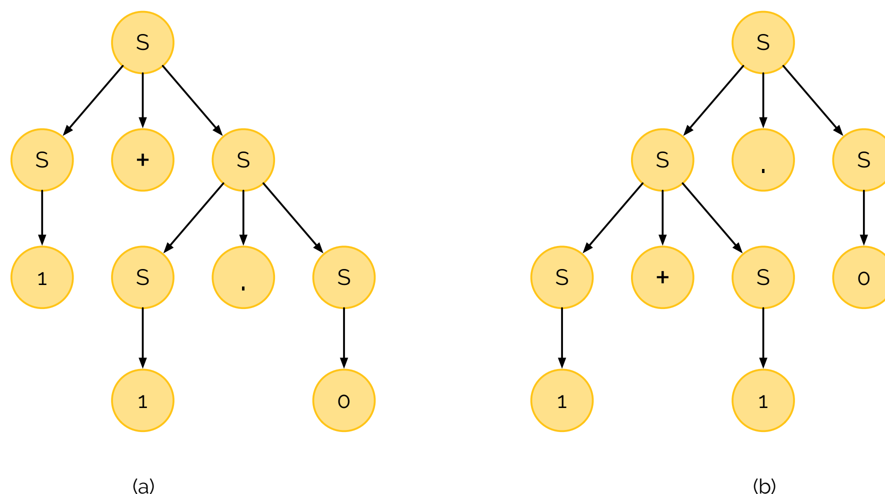


figura 4.2

Como podemos ver a gramática de (4.10), pela Figura 4.2, é ambígua. Mas como é que podemos criar agora uma alternativa não-ambígua, desta gramática? Vejamos a gramática em (4.12).

$$\begin{aligned} &S \to T \mid S + T \\ &T \to F \mid T \,.\, F \\ &F \to K \mid \neg K \\ &K \to 0 \mid 1 \mid (\ S\ ) \end{aligned} \tag{4.12}$$

Note-se que há linguagens que são **inerentemente ambíguas**, isto é, em que é impossível definir-se uma alternativa não-ambígua, como a linguagem L descrita em (4.13).

inerentemente ambíguas

$$L = \{\mathrm{a}^i\,\mathrm{b}^j\,\mathrm{c}^k : i = j \vee j = k\} \tag{4.13}$$

Num Exercício 4.2 tentemos averiguar se a gramática em (4.14) é ambígua, testando, por exemplo, a palavra 'icicoeo'.

exercício 4.2

$$S \to \mathrm{o} \mid \mathrm{i\ c}\ S \mid \mathrm{i\ c}\ S\ \mathrm{e}\ S \tag{4.14}$$

Ora, para testarmos a ambiguidade podemos criar a árvore de derivação ou apenas efetuar a derivação direta e verificar se há mais do que uma forma de criar a palavra 'icicoeo'. Vejamos (4.15).

$$\begin{aligned} &S \Rightarrow \mathrm{i\ c}\ S \Rightarrow \mathrm{i\ c\ i\ c}\ S\ \mathrm{e}\ S \Rightarrow \mathrm{i\ c\ i\ c\ o\ e}\ S \Rightarrow \mathrm{i\ c\ i\ c\ o\ e\ o} \\ &S \Rightarrow \mathrm{i\ c}\ S\ \mathrm{e}\ S \Rightarrow \mathrm{i\ c\ i\ c}\ S\ \mathrm{e}\ S \Rightarrow \mathrm{i\ c\ i\ c\ o\ e}\ S \Rightarrow \mathrm{i\ c\ i\ c\ o\ e\ o} \end{aligned} \tag{4.15}$$

A aplicação deste exemplo prático pode ser verificada no controlo de fluxo de uma linguagem de programação - o *if-then-else*. Mas como podemos ver esta solução em particular mostra pelo menos dois casos para uma determinada palavra, como vimos em (4.15). Para resolver este problema temos que criar uma nova solução, como em (4.16).

$$\begin{aligned} &S \to \mathrm{o} \mid \mathrm{i\ c}\ S \mid \mathrm{i\ c}\ X\ \mathrm{e}\ S \\ &X \to \mathrm{o} \mid \mathrm{i\ c}\ X\ \mathrm{e}\ X \end{aligned} \tag{4.16}$$

5. Gramáticas de Atributos

Como vimos na primeira secção de §Introdução ao processamento de linguagens, numa primeira fase aplicamos uma análise lexical, na qual não nos preocupamos com a forma como as palavras estão ordenadas ou qual o sentido delas, mas antes em saber se o que temos num todo cumpre regras de pertença a um alfabeto, por exemplo. Numa fase seguinte, aí sim, há que saber avaliar o contexto de cada um dos elementos do nosso texto, verificando o seu **significado**. Mas ao avaliar um texto, como vimos, de acordo com uma gramática, atingimos um ponto "terminal" no qual de seguida temos de recuar para continuar a avaliação do texto restante - os **atributos**. Estes atributos têm de ter algum sentido quando chega a fase de análise semântica, de forma a que não sejam mal-julgados. Assim, na fase de análise sintática há uma atribuição de significado às produções de uma gramática, criando-se atributos para designar valores a símbolos terminais e não-terminais. Cada símbolo terminal ou não-terminal, assim, poderá ter um conjunto de zero ou mais atributos, sendo estes palavras, números, tipos, posições de memória, entre outros.

significado

atributos

Às produções estão sempre associadas regras semânticas que determinam os valores dos atributos e que podem ter efeitos laterais, como a alteração de uma estrutura de dados, ... Cria-se assim uma **definição semântica** que é composta por uma gramática independente do contexto, um conjunto de atributos associados aos seus símbolos e um conjunto de regras semânticas associadas às suas produções.

definição semântica

Análise de expressões aritméticas e declarações

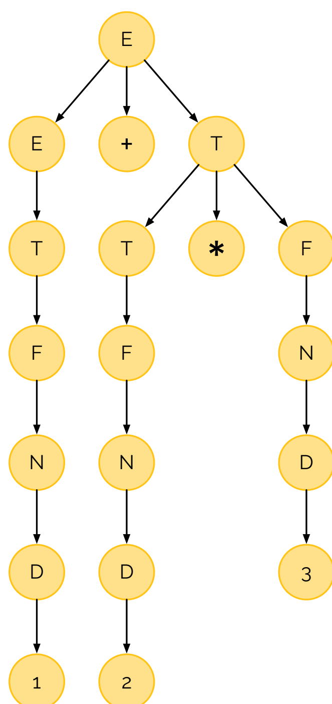
Consideremos a gramática em (5.1), onde se pretende desenhar o esquema de construção de expressões aritméticas como 1 * 2 + 3 * 0.

$$
\begin{aligned}
&E \rightarrow T + E \mid T \\
&T \rightarrow F * T \mid F \\
&F \rightarrow N \mid (\ E\ ) \\
&N \rightarrow D \mid N\ D \\
&D \rightarrow 0 \mid 1 \mid 2 \mid 3
\end{aligned}
\tag{5.1}
$$

Num Exercício 5.1 pretende-se criar a árvore de derivação para a palavra '1+2*3' sobre a gramática em (5.1). Veja-se assim a Figura 5.1.

exercício 5.1

figura 5.1



Para aplicar uma gramática de atributos temos de saber designar uma estrutura de dados por cada nó, de modo a que cada símbolo não-terminal seja associado um atributo que guarde o valor que a sub-árvore representa. Veja-se assim a Figura 5.2, onde se pretende demonstrar essa ação.

Um segundo exemplo de aplicação da gramática de atributos é na declaração de variáveis, como conhecemos de linguagens de alto nível. Consideremos para tal a gramática em (5.2).

$$
\begin{aligned}
&D \rightarrow T\ L \\
&T \rightarrow \mathrm{i} \mid \mathrm{f} \\
&L \rightarrow V \mid L\ ,\ \ V \\
&V \rightarrow \mathrm{a} \mid \mathrm{b} \mid \mathrm{c} \mid \mathrm{d}
\end{aligned}
\tag{5.2}
$$

Num Exercício 5.2 pretende-se criar a árvore de derivação para a palavra 'i a,b' sobre a gramática em (5.2). Veja-se assim a Figura 5.3.

exercício 5.2

Para aplicar uma gramática de atributos, mais uma vez, temos de saber designar uma estrutura de dados por cada nó, de modo a que cada símbolo não-terminal seja associado um atributo que guarde o valor ou o tipo que a sub-árvore representa. Veja-se assim a Figura 5.4, onde se pretende demonstrar essa ação.

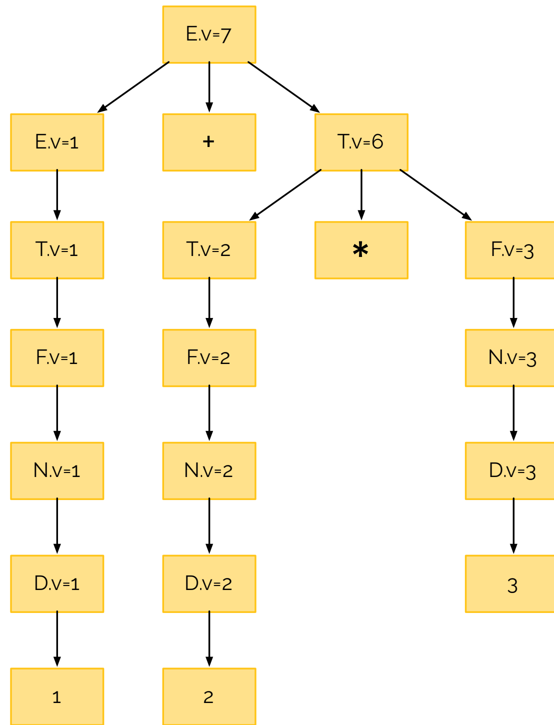


figura 5.2

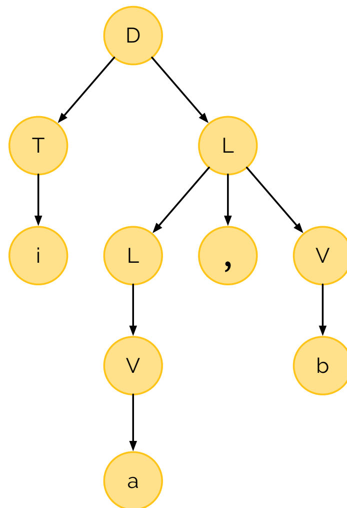


figura 5.3

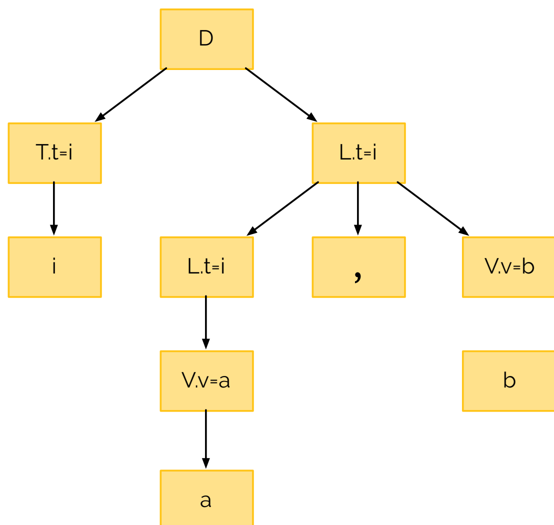


figura 5.4

Tipos de atributos e herança

Seja $G = (T, N, S, P)$ uma gramática independente do contexto. A cada produção $X \rightarrow u \in P$, com $u \in (T \cup N)^*$, podem associar-se regras semânticas para o cálculo dos valores dos atributos $b = f(c_1, c_2, \ldots, c_n)$ onde $c_1, c_2, \ldots, c_n$ são atributos dos símbolos que ocorrem na produção e b é um atributo do símbolo X ou de um dos símbolos não-terminais presentes em u. Podem ainda associar-se regras semânticas com efeitos colaterais $g(c_1, \ldots, c_n)$ sendo que aqui o atribuito considera-se fictício.

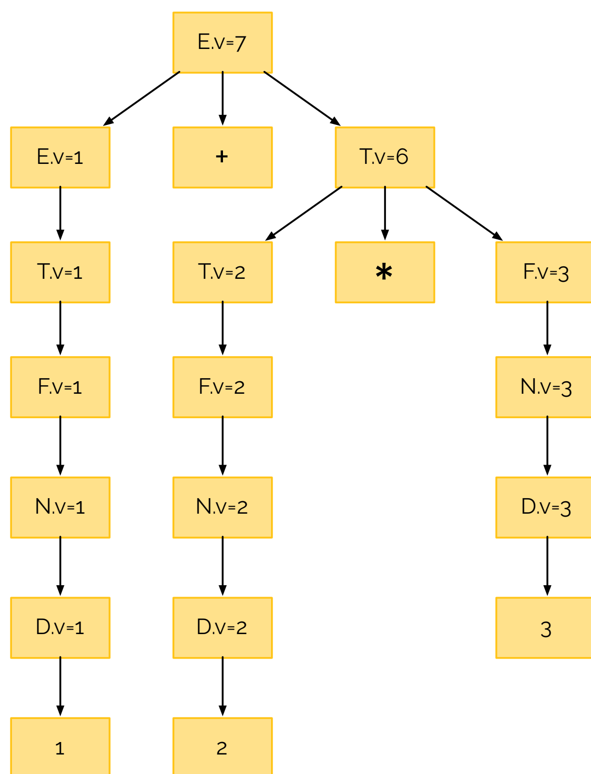
Considerando ainda as mesmas premissas que no início desta secção, um atributo b diz-se **sintetizado** se b está associado a X e todos os c_i com $i = 1, 2, \ldots, n$ estão associados a símbolos de u, ou **herdado** se b está associado a um dos símbolos não-terminais de u.

sintetizado

herdado

Vejamos de novo a Figura 5.2, replicada na Figura 5.5.

figura 5.5



Tal como se pode ver, na Figura 5.5 foi associado a cada símbolo não-terminal um atributo que guarda o valor que a sub-árvore representa. Sintetizando estes atributos podemos obter a Figura 5.6 onde o valor do atributo do símbolo da cabeça de produção é calculado com base nos valores dos atributos dos símbolos do corpo da produção.

Olhando para os atributos herdados podemos ter em conta a Figura 5.7 (rep. 5.5).

figura 5.7

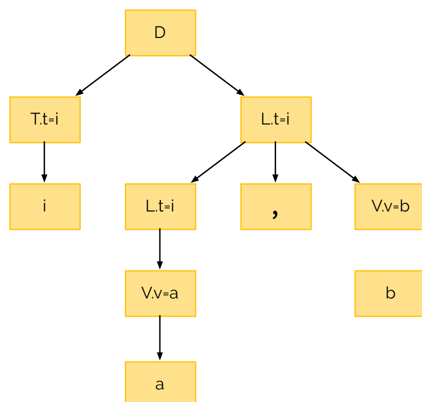


figura 5.6

produção	regras semânticas
D → 0	D.v = 0;
D → 1	D.v = 1;
D → 2	D.v = 2;
D → 3	D.v = 3;
N → D	N.v = D.v
N_1 → N_2 D	N_1.v = N_2.v * 4 + D.v
F → N	F.v = N.v
F → (E)	F.v = E.v
T → F	T.v = F.v
T_1 → T_2 * F	T_1.v = T_2.v * F.v
E → T	E.v = T.v
E_1 → E_2 + T	E_1.v = E_2.v * T.v

No caso da herança de atributos também podemos formar uma tabela ao género da da Figura 5.6, na Figura 5.8, onde há atributos de símbolos do corpo das produções que dependem do símbolo na cabeça dessas produções, havendo ainda ações laterais.

figura 5.8

produção	regras semânticas
T → i	T.t = int
T → f	T.t = float
D → T L	L.t = T.t
L_1 → L_2 , V	L_2.t = L_1.t; addsym(V.v, L_1.t)
L → V	addsym(V.v, L.t)
V → a	V.v = a
V → b	V.v = b
V → c	V.v = c
V → d	V.v = d

Avaliação dirigida pela sintaxe e grafo de dependências

Numa avaliação **dirigida pela sintaxe** o cálculo dos atributos é feito à medida que é feita a análise sintática. Num analisador sintático ascendente (como é o caso da ferramenta *bison*) todos os atributos têm de ser sintetizados. Pelo contrário, num analisador sintático descendente além de sintetizadores os atributos podem ser herdados, desde que estes sejam símbolos à esquerda ou do símbolo pai.

dirigida pela sintaxe

Em termos gerais é possível definir aquilo que chamamos de **grafo de dependências**, que define a ordem de cálculo dos atributos.

grafo de dependências

6. Autómatos de Pilha

Neste capítulo iremos tratar da análise sintática (*parsers*) baseados em **autómatos de pilha**. Um autómato de pilha é uma estrutura semelhante à da Figura 3.1 onde os conteúdos lidos são transportados para uma estrutura de dados de pilha, primeiramente. Por outras palavras, expressões como $a + b$ levam a e b para a pilha e quando lê $+$ retira b e a da pilha. Isto permite que expressões como '(E)' sejam avaliadas analogamente a uma pessoa que compara dois objetos em cada uma das mãos, guardando '(' e retirando da pilha se e só se houver ')'.

autómatos de pilha

Transformações em gramáticas independentes do contexto

Em muitas situações práticas é necessário saber transformar uma gramática numa outra que lhe seja equivalente e possua determinadas propriedades. Uma das ações possíveis de serem executadas é a **eliminação de produções-ε**. Uma produção-ε é uma tal do tipo $A \rightarrow \varepsilon$, para um qualquer símbolo não-terminal A. Se L for uma linguagem independente do contexto tal que $\varepsilon \notin L$, é possível descrever a linguagem L por uma gramática independente do contexto e sem produções-ε. Se assim for, então tem de ser possível transformar uma gramática que descreva uma linguagem L e possua produções-ε numa outra equivalente que as não possua. Consideremos assim a gramática em (6.1).

eliminação de produções-ε

$$I \rightarrow 0\ I \mid 1\ P$$
$$P \rightarrow \varepsilon \mid 0\ P \mid 1\ I \tag{6.1}$$

A gramática em (6.1) descreve a linguagem L formada pelas palavras definidas sobre o alfabeto {0,1}, com número ímpar de átomos '1'. Claramente, a palavra vazia (ε) não pertence a L porque não tem número ímpar de '1'. Mas a gramática contém a regra que de $P \rightarrow \varepsilon$. De acordo com o que já foi referido é então possível criar uma gramática que não tem produções-ε. Isto significa que as produções $I \rightarrow 1\ P$ e $P \rightarrow 0\ P$ podem produzir as derivações $I \Rightarrow 1$ e $P \Rightarrow 0$, respetivamente. Mas estas derivações podem ser contempladas acrescentando as produções $I \rightarrow 1$ e $P \rightarrow 0$ à gramática, tornando desnecessária a produção-ε. Criamos assim a gramática em (6.2).

$$I \rightarrow 0\ I \mid 1\ P \mid 1$$
$$P \rightarrow 0\ P \mid 0 \mid 1\ I \tag{6.2}$$

Em geral, o papel da produção $A \rightarrow \varepsilon$ sobre uma produção $B \rightarrow \alpha B \beta$ pode ser representado pela inclusão da produção $B \rightarrow \alpha\beta$. Assim a eliminação das produções-ε de uma gramática pode ser obtido por aplicação do algoritmo seguinte, do Código 6.1.

código 6.1

para cada $A \rightarrow \varepsilon$:
  para cada $B \rightarrow \alpha A \beta$:
    adicionar $B \rightarrow \alpha\beta$ a P ;
  remover $A \rightarrow \varepsilon$ de P ;

O algoritmo do Código 6.1 pode introduzir novas produções-ε na gramática, mas se $B \rightarrow A$ for uma produção em P a eliminação de $A \rightarrow \varepsilon$ introduz a produção $B \rightarrow \varepsilon$. Nestes casos podemos reproduzir o algoritmo em Código 6.1 e experimentá-lo para um A sendo B ($A = B$).

Uma segunda ação que pode ser conduzida como transformação de gramáticas é a eliminação da **recursividade à esquerda**. Diz-se que uma determinada gramática é **recursiva à esquerda** se possuir um símbolo não-terminal A que admita uma derivação do tipo $A \Rightarrow^{+} A\gamma$, ou seja, que seja possível num ou mais passos de derivação transformar A numa expressão que tem A no início. A gramática em (6.3) é um exemplo de gramática recursiva à esquerda.

recursividade à esquerda
recursiva à esquerda

$$E \rightarrow X\ T$$
$$X \rightarrow \varepsilon \mid E +$$
$$T \rightarrow \mathrm{a} \mid \mathrm{b} \mid (\ E\ ) \tag{6.3}$$

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

Determina-se que se a obtenção de uma expressão começada por A se faz em apenas um passo de derivação a recursividade é **imediata**. Mas este tipo de situação só ocorre se a gramática possuir uma ou mais produções do tipo $A \rightarrow A\ \alpha$. Veja-se (6.4) para um exemplo semelhante ao anterior de (6.3).

imediata

$$E \rightarrow T \mid E + T$$
$$T \rightarrow \mathrm{a} \mid \mathrm{b} \mid (\ E\ ) \tag{6.4}$$

A eliminação de recursividade imediata à esquerda faz-se com um algoritmo bastante simples. Consideremos primeiro que $A \rightarrow \beta$ e $A \rightarrow A\alpha$, onde A é um símbolo não-terminal e α e β são sequências de zero ou mais símbolos terminais ou não-terminais, são duas produções de uma gramática qualquer. Será possível substituir as duas prroduções por outras que não possuam recursividade à esquerda e produzam uma gramática equivalente? Para responder a esta pergunta observem-se as palavras que se podem obter a partir de A. Numa derivação de um passo obtém-se que $A \Rightarrow \beta$. Numa de dois passos obtemos que $A \Rightarrow A\alpha \Rightarrow \beta\alpha$. Numa de n passos, com $n > 0$, obtemos que $A \Rightarrow$ $\Rightarrow \beta\alpha^{n\text{-}1}$. Mas estas palavras também podem ser obtidas com as produções de (6.5).

$$A \rightarrow \beta\ X$$
$$X \rightarrow \varepsilon \mid \alpha\ X \tag{6.5}$$

O que podemos ver em (6.5) é que esta nova gramática continua a ser recursiva, mas agora à direita.

Fatorização à esquerda

A **fatorização à esquerda** é uma transformação da gramática (um pouco mais especial que as outras, daí ter uma secção própria) que é mais útil para produzir uma gramática prática para os métodos de análise sintática que iremos abordar mais à frente. Quando a escolha entre duas alternativas de produções de A não é clara, então talvez a rescrita das produções que inferem essa decisão seja o melhor a fazer, até que a entrada produza a decisão correta. Por exemplo, considerando a gramática em (6.6) ao ver a entrada *if* não podemos automaticamente inferir qual das produções é que havemos de escolher para extender *statement*.

fatorização à esquerda

$$\begin{aligned} statement \rightarrow & \text{ if } expression \text{ then } statement \text{ else } statement \\ \mid & \text{ if } expression \text{ then } statement \end{aligned} \tag{6.6}$$

Em termos gerais, se $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ são duas produções para A e a entrada começa com uma string não vazia derivada de α, nós deixamos de conseguir designar se havemos de seguir $\alpha\beta_1$ ou $\alpha\beta_2$. No entanto podemos modificar a decisão expandindo A para $\alpha A'$. Então, depois de ver a entrada derivada de α, expandimos A' para β_1 ou para β_2. Por outras palavras, as produções, por fatorização à esquerda, ficam como em (6.7).

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2 \tag{6.7}$$

Para fazer a fatorização, tendo uma gramática G, para cada símbolo não-terminal A, temos de encontrar o maior prefixo de α comum a duas ou mais alternativas. Se $\alpha \neq$ $\neq \varepsilon$, isto é, se existe um prefixo comum não-trivial, substituímos todas as produções de A $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$, onde γ representa todas as alternativas que não começam com α, pela gramática de (6.8). Nesta última gramática A' é um novo símbolo não-terminal. Fazendo isto de forma repetida, até que não haja duas alternativas para o mesmo não-

terminal produz um algoritmo para a fatorização à esquerda, tendo como saída uma gramática equivalente fatorizada à esquerda.

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \quad (6.8)$$

Os conjuntos first, follow e predict

A construção de reconhecedores sintáticos (em inglês vulgarmente denominados de **parsers**) é auxiliada por duas funções associadas às gramáticas. Estas funções manipulam os conjuntos denominados de *first*, *follow* e *predict*.

parsers

O **conjunto first**, sendo $G = (T, N, P, S)$ uma gramática qualquer e α uma sequência de símbolos terminais e não-terminais, contém todos os símbolos terminais que podem aparecer no início de sequências obtidas a partir de α por aplicação de produções da gramática. Por outras palavras, um símbolo terminal x pertence a *first*(α) se e só se $\alpha \Rightarrow^* x\beta$, com β qualquer. Adicionalmente, considera-se que ε pertence ao conjunto *first*(α) se $\alpha \Rightarrow^* \varepsilon$.

conjunto first

Vejamos, no Código 6.2, o algoritmo do cálculo do conjunto *first*.

código 6.2

```
first(α):
   se α == ε :
      devolve {ε};
   caso contrário, se (α == a e a ∈ T) :
      devolve {a};
   caso contrário, se (α == B e B ∈ N) :
      M = {};
      para cada (B → γ) ∈ P :
         M = M ∪ first(γ)
      devolve M ;
   caso contrário :
      x = head(α);      // primeiro símbolo
      β = tail(α);      // todos os itens exceto primeiro
      M = first(x);
      se ε ∉ M :
         devolve M ;
      caso contrário :
         devolve (M - {ε}) ∪ first(β);
```

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

conjunto follow

O cálculo dos conjuntos *follow* dos símbolos não-terminais da gramática G baseia-se na aplicação das quatro regras seguintes:

- \$ pertence a *follow*(S);
- se $(A \rightarrow \alpha B) \in P$, então *follow*($B$) contém *follow*($A$);
- se $(A \rightarrow \alpha B\beta) \in P$ e $\varepsilon \notin first(\beta)$, então *follow*($B$)contém *first*(β);
- se $(A \rightarrow \alpha B\beta) \in P$ e $\varepsilon \in first(\beta)$, então *follow*($B$) contém $(first(\beta)\text{-}\{\varepsilon\}) \cup$ $\cup follow(A)$.

A primeira regra é clarividente, mas na segunda regra, com mais cuidado, é dito que se $A \rightarrow \alpha B$ é uma produção da gramática e x pertence ao conjunto *follow*(A), então x pertence ao conjunto *follow*(B).

Finalmente, o **conjunto predict** aplica-se às produções de uma gramática e envolve os conjuntos *first* e *follow*. Veja-se assim, como sua definição, (6.9).

conjunto predict

$$predict(A \rightarrow \alpha) = \begin{cases} first(\alpha) & \varepsilon \notin first(\alpha) \\ (first(\alpha) - \{\varepsilon\}) \cup follow(A) & \varepsilon \in first(\alpha) \end{cases} \quad (6.9)$$

Análise sintática descendente

Consideremos uma gramática $G = (T, N, P, S)$ e uma palavra $u \in T^*$, $u \in L(G)$ se e só se existir uma derivação que produza u a partir de S, isto é, se $S \Rightarrow^* u$. Um **reconhecedor sintático** é um mecanismo que responde à pergunta "$u \in L(G)$?", tentando produzir a derivação anterior. Para o fazer, o reconhecedor por partir de S e tentar chegar a u ou a partir de u e tentar chegar a S. No primeiro caso dizemos que o reconhecedor é **descendente**, dado que o seu procedimento corresponde à geração da árvore de derivação da palavra u de cima (da raiz) para baixo (para as suas folhas).

reconhecedor sintático

descendente

O papel da análise sintática é definir procedimentos que permitam então construir reconhecedores sintáticos a partir da gramática. Por exemplo, consideremos a linguagem L descrita pela gramática em (6.10).

$$S \rightarrow \mathrm{a}\ S\ \mathrm{b} \mid \mathrm{c}\ S \mid \varepsilon \quad (6.10)$$

Será que a palavra 'acacbb' pertence à linguagem L? Pertencerá se de algum modo a derivação $S \Rightarrow^*$ acacbb. Como podemos ver pertence, dado (6.11).

$$S \Rightarrow \mathrm{a}S\mathrm{b} \Rightarrow \mathrm{ac}S\mathrm{b} \Rightarrow \mathrm{aca}S\mathrm{bb} \Rightarrow \mathrm{acac}S\mathrm{bb} \Rightarrow \mathrm{acacbb} \quad (6.11)$$

A grande questão em termos de reconhecimento de derivações é saber produzir, perante duas ou mais produções com o mesmo símbolo à cabeça, a derivação anterior, de forma determinística, não estocástica. Por exemplo, consideremos novamente o caso da gramática de (6.10) e vejamos como é que se escolhe a opção correta para expandir S usando $S \rightarrow$ a S b e não $S \rightarrow$ c S ou $S \rightarrow \varepsilon$. A solução baseia-se na observação antecipada dos próximos símbolos à entrada, através de mecanismos de **lookahead**. No exemplo, se o próximo símbolo à entrada for um 'a' expandimos usando a regra $S \rightarrow \mathrm{a}S\mathrm{b}$, por exemplo. Podemos assim definir uma tabela que, para cada símbolo não-terminal da gramática e para cada valor do *lookahead*, indica qual a produção da gramática que deve ser usada na expansão. A profundidade da observação antecipada (número de símbolos de *lookahead*) pode ser qualquer, mas aqui iremos sempre considerar 1 como profundidade máxima. Veja--se assim a tabela da Figura 6.1 onde está representado o *lookahead* para a gramática (6.10).

lookahead

símbolo	a	b	c	$
S	S → a S b	S → ε	S → c S	S → ε

figura 6.1

Na tabela da Figura 6.1 o preenchimento das colunas 'a' e 'c' são óbvias, visto que as produções associadas começam pelo próprio símbolo do *lookahead*. Nas colunas 'b' e '$' tal não acontece. O preenchimento da tabela de reconhecimento preditivo baseia-se nos conjuntos *first*, *follow* e *predict*, usando o algoritmo do Código 6.3.

código 6.3

para cada $(A \to \alpha) \in P$:
  para cada $\alpha \in predict(A \to \alpha)$:
    adicionar $(A \to \alpha)$ **a** $T[A, \alpha]$;

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

LL(1)

Num Exercício 6.1 calculemos a tabela de reconhecimento de um reconhecedor preditivo para a gramática seguinte, em (6.12).

exercício 6.1

$$S \to A\ B$$
$$A \to \varepsilon \mid \mathrm{a}\ A$$
$$B \to \varepsilon \mid \mathrm{b}\ B \qquad (6.12)$$

Para iniciar este exercício precisamos de calcular o conjunto *predict* para cada uma das regras de (6.12). Para nos ser mais fácil tomar partido dos resultados dos conjuntos *first* e *follow* calculemos estes resultados para cada uma das produções de (6.12). Começemos por calcular o conjunto *first* de S, A e B[1]. O *first*(A) é o conjunto dos primeiros terminais de A, isto é, pode ser tanto 'a' como ε. O *first*(B), analogamente, pode ser 'b' ou ε. Assim sendo, o *first*(S) é o *first*(A) e, como este tem ε, o *first*(B) também. Como ambos podem gerar ε, num pior caso, o *first*(S) também deve conter ε, pelo que *first*$(S) = \{\mathrm{a}, \mathrm{b}, \varepsilon\}$. Veja-se a Figura 6.2 onde se expõe os cálculos do *first*.

símbolo	first
S	{a, b, ε}
A	{a, ε}
B	{b, ε}

figura 6.2

De seguida calculemos os conjuntos *follow*, para facilitar o trabalho caso estes sejam necessários. Assim, o *follow*(S) é $\{\$\}$ apenas, dado que não é chamado em nenhuma regra. O *follow*(A) ocorre com o *first*(B) ou caso este não ocorra, com o *follow*(S). Já o *follow*(B) ocorre com o *follow*(S). Veja-se assim a Figura 6.3 onde se resumem as nossas conclusões.

símbolo	first	follow
S	{a, b, ε}	{$}
A	{a, ε}	{b, $}
B	{b, ε}	{$}

figura 6.3

Continuamos então por calcular o *predict*$(S \to A\ B)$. Como ε faz parte do *first*$(A\ B)$, então temos que o conjunto é $\{\mathrm{a}, \mathrm{b}, \$\}$. O *predict*$(A \to \varepsilon)$, como ε é *first*$(\varepsilon)$ temos que resulta na união de não-ε com o *follow*(B), logo $\{\mathrm{b}, \$\}$. O *predict*$(A \to \mathrm{a}\ A)$ é, dado que não contém ε, o *first*$(\mathrm{a}A)$, isto é, $\{\mathrm{a}\}$. O *predict*$(B \to \varepsilon)$, como ε é *first*(ε) temos que resulta na união de não-ε com o *follow*(S), logo $\{\$\}$. O *predict*$(B \to \mathrm{b}\ B)$ é, dado que não contém ε, o *first*$(\mathrm{b}B)$, isto é, $\{\mathrm{b}\}$. Vejamos assim a Figura 6.4.

1 Embora seja cometida uma incoerência, consideremos no caso do first que prentendemos tratar por A, B ou S as regras (produções) de cada A, B ou S.

símbolo	a	b	$
S	S → A B	S → A B	S → A B
A	A → a A	A → ε	A → ε
B		B → b B	B → ε

figura 6.4

O reconhecedor preditivo, sintetizado na tabela de reconhecimento apresentada na Figura 6.4 permite construir programas de reconhecimento. Uma solução para a construção do reconhecedor é baseada numa estrutura na qual cada símbolo não-terminal da gramática dá origem a uma função, possivelmente recursiva.

Uma outra solução para implementar o reconhecedor usa uma pilha para reter o estado no processo de reconhecimento e usa a tabela de reconhecimento preditivo para decidir como evoluir no processo de reconhecimento. Seja assim uma gramática $G = (T, N, P, S)$ independente do contexto, que se assume que seja LL(1). Seja também M a tabela de reconhecimento preditivo de G para um *lookahead* de profundidade 1. Finalmente, consideremos que dispomos de uma pilha onde podemos armazenar elementos do conjunto $Z = T \cup N \cup \{\$\}$ e que pode ser manipulada com as funções de *push*, *pop* e *top*. Se considerarmos a gramática em (6.12) a execução de tal *parser* sobre a palavra 'aab' resultaria na tabela da Figura 6.5. Na coluna da pilha, o símbolo mais à direita é o que está no topo e, na coluna entrada, o símbolo mais à esquerda é o *lookahead*.

pilha	entrada	ação
$ S	**a** a b $	pop(), push(A B)
$ B A	**a** a b $	pop(), push(a A)
$ B A a	**a** a b $	pop(), lookahead=token()
$ B A	**a** b $	pop(), push(a A)
$ B A a	**a** b $	pop(), lookahead=token()
$ B A	**b** $	pop()
$ B	**b** $	pop(), push(b B)
$ B b	**b** $	pop(), lookahead=token()
$ B	**$**	pop()
$	**$**	ACCEPT

figura 6.5

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

Análise sintática ascendente

Consideremos agora a gramática em (6.13) que representa uma declaração de variáveis como conhecemos da linguagem C.

$$D \to T\ L\ ;$$
$$T \to \mathrm{i} \mid \mathrm{r}$$
$$L \to \mathrm{v} \mid L\ ,\ \mathrm{v} \qquad (6.13)$$

Como reconhecer a palavra 'i v , v ;' como pertencente à linguagem gerada pela gramática em (6.13)? Se u pertence à linguagem gerada pela gramática, então $D \Rightarrow^*$

$\Rightarrow^* u$. Tentemos chegar até lá, mas no sentido inverso da derivação, isto é, da palavra u para D. Veja-se assim (6.14).

(6.14)

$$
\begin{array}{ll}
\text{i v , v ;} & \\
\Leftarrow T \text{ v , v ;} & \text{(por aplicação da regra } T \rightarrow \text{i)} \\
\Leftarrow T\ L \text{ , v ;} & \text{(por aplicação da regra } L \rightarrow \text{v)} \\
\Leftarrow T\ L \text{ ;} & \text{(por aplicação da regra } L \rightarrow L \text{ , v)} \\
\Leftarrow D & \text{(por aplicação da regra } D \rightarrow T\ L \text{ ;)}
\end{array}
$$

Aplicando este raciocínio a um autómato de pilha obtemos uma execução como a definida na tabela da Figura 6.6.

figura 6.6

pilha	entrada	ação
$	i v , v ; $	deslocamento
$ i	v , v ; $	redução por T → i
$ T	v , v ; $	deslocamento
$ T v	, v ; $	redução por L → v
$ T L	, v ; $	deslocamento
$ T L ,	v ; $	deslocamento
$ T L , v	; $	redução por L → L , v
$ T L	; $	deslocamento
$ T L ;	$	redução por D → T L ;
$ D	$	ACCEPT

Inicialmente, o topo da pilha apenas possui um símbolo especial, representado por um $, que indica, quando no topo, que a pilha está vazia. A entrada possui a palavra a reconhecer seguida também de um símbolo especial, aqui representado por um $, que indica fim de entrada. Em cada ciclo realiza-se, normalmente, uma operação de deslocamento ou de redução. A operação de **deslocamento** (em inglês *shift*) transfere o símbolo não-terminal da entrada para o topo da pilha. Por outro lado, a operação de **redução** (em inglês *reduce*) substitui os símbolos do topo da pilha que correspondem ao corpo de uma produção da gramática pela cabeça dessa regra. Se se atingir uma situação em que a entrada apenas possui o símbolo $ e a pilha apenas possui o símbolo $ e o símbolo inicial da gramática, tal como acontece na tabela da Figura 6.6, a palavra é reconhecida como pertencendo à linguagem, caso contrário é rejeitada. Veja-se um exemplo de rejeição na Figura 6.7, onde tentamos executar para a palavra 'i v v ;'.

deslocamento

redução

figura 6.7

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

O que acontece na Figura 6.7 é que com 'T v' e 'v' na entrada é impossível chegar-se a uma aceitação, porque se se reduzir 'v' para L ficar-se-ia com um $T\ L$ na pilha e um 'v' na entrada, que não pertence ao conjunto $\mathit{follow}(L)$.

Mas a rejeição e a aceitação não são os únicos estados conclusivos deste algoritmo - podem acontecer outras eventualidades a que denominamos de **conflitos**. Consideremos, para analisar estes casos, a gramática de (6.15) e tentemos executar o procedimento de reconhecimento para a palavra 'icicaea' na Figura 6.8.

conflitos

$S \rightarrow$ i c S | i c S e S | a (6.15)

figura 6.8

pilha	entrada	ação
$	i c i c a e a $	deslocamento
$ i	c i c a e a $	deslocamento
$ i c	i c a e a $	deslocamento
$ i c i	c a e a $	deslocamento
$ i c i c	a e a $	deslocamento
$ i c i c a	e a $	redução por S → a
$ i c i c S	e a $	**conflito** (redução ou desl.)

O que acontece na tabela da Figura 6.8 é que chegamos a um ponto em que "tanto faz" escolher executar uma redução usando a regra $S \rightarrow$ i c S ou um deslocamento para tentar $S \rightarrow$ i c S e S. A este tipo de conflito damos o nome de conflito **deslocamento-redução** (em inglês *shift-reduce conflict*) e não temos como escolher a melhor forma de avançar. No entanto há que ter cuidado porque, embora possa não ser a opção mais ótima, ferramentas como o *bison* efetua deslocamentos em conflitos deste tipo.

deslocamento-redução

Vejamos agora um segundo caso, como o da gramática em (6.16).

$S \rightarrow A \mid B$
$A \rightarrow$ c | A a
$B \rightarrow$ c | B b (6.16)

Tentemos novamente executar o procedimento de reconhecimento agora para a gramática em (6.16) e com a palavra 'c'. Veja-se a Figura 6.9.

figura 6.9

pilha	entrada	ação
$	c $	deslocamento
$ c	$	**conflito** (que redução?)

O que aconteceu na execução retratada na Figura 6.9 é que na última o *parser* não consegue decidir entre duas reduções possíveis: $A \rightarrow$ c ou $B \rightarrow$ c. Perante este tipo de conflitos, ferramentas como o *bison* optam pela produção que aparece primeiro, podendo, como no outro tipo de conflito, não ser a solução mais adequada. A este tipo de conflitos damos o nome de conflitos **redução-redução** (em inglês *reduce-reduce conflict*).

redução-redução

Construção de um reconhecedor ascendente

O procedimento de reconhecimento, como podemos ver, é um processo iterativo. Em cada passo, a operação a realizar depende da configuração nesse momento. Uma **configuração** é assim formada pelo conteúdo da pilha mais a parte da entrada ainda não processada. A pilha é conhecida, mas entrada é desconhecida, conhecendo-se antes apenas o próximo símbolo (*lookahead*). Então a decisão a tomar só pode basear-se no conteúdo da pilha e no *lookahead*. Mas, se se quiser construir um reconhecedor apenas com capacidade de observar o topo da pilha, uma pilha onde se guardam os símbolos terminais e não-terminais tem pouco interesse. Deve-se guardar, então, um símbolo que represente tudo o que está para trás. Para definir tais símbolos precisamos de considerar um conjunto de estados, onde cada um representa um conjunto de itens. Um item de uma gramática é uma produção com um ponto (*dot*) numa posição do seu corpo. Por exemplo, a produção $A \rightarrow B_1\ B_2\ B_3$ produz 4 itens: $A \rightarrow \cdot\ B_1\ B_2\ B_3$, $A \rightarrow B_1 \cdot B_2\ B_3$, $A \rightarrow B_1\ B_2 \cdot B_3$ e

configuração

$A \rightarrow B_1\ B_2\ B_3\ \cdot$. Um **item** representa assim o quanto de uma produção é que já foi realizada e, simultaneamente, o quanto falta obter.

item

Consideremos agora a gramática em (6.17).

$$S \rightarrow E$$
$$E \rightarrow \mathrm{a} \mid (\ E\ ) \tag{6.17}$$

O estado inicial - primeiro elemento da coleção de conjunto de itens - contém o item $Z_0 = \{S \rightarrow \cdot\ E\ \$\}$. Este conjunto tem de ser fechado. O facto do ponto se encontrar imediatamente à esquerda de um símbolo não-terminal, significa que para se avançar no processo de reconhecimento é preciso obter esse símbolo. Isso é considerado juntando ao conjunto Z_0 os itens iniciais das produções cuja cabeça é E. Fazendo-o, Z_0 passa a (6.18).

$$Z_0 = \{S \rightarrow \cdot\ E\ \$\} \cup \{E \rightarrow \cdot\ \mathrm{a},\ E \rightarrow \cdot\ (\ E\ )\ \} \tag{6.18}$$

Se nos novos elementos adicionados ao conjunto voltasse a acontecer de o ponto ficar imediatamente à esquerda de outros símbolos não-terminais o processo deve ser repetido para esses símbolos. O estado Z_0 pode então evoluir por ocorrência de E, um 'a' ou um '('. Correspondem aos símbolos que aparecem imediatamente à direita do ponto e produzem 3 novos estados, como vemos em (6.19).

$$Z_1 = \delta(Z_0, E) = \{S \rightarrow E \cdot \$\}$$
$$Z_2 = \delta(Z_0, \mathrm{a}) = \{E \rightarrow \mathrm{a}\ \cdot\ \}$$
$$Z_3 = \delta(Z_0, (\ ) = \{E \rightarrow (\ \cdot\ E\ )\ \} \cup \{E \rightarrow \cdot\ \mathrm{a},\ E \rightarrow \cdot\ (\ E\ )\ \} \tag{6.19}$$

Note-se que Z_3 foi extendido pela função de fecho, uma vez que o ponto ficou imediatamente à esquerda de um símbolo não-terminal, o E. Z_1 representa uma situação de aceitação se o símbolo à entrada - o *lookahead* - for igual a \$ e de erro caso contrário. Z_2 representa uma possível situação de redução pela regra $E \rightarrow \mathrm{a}$. Esta redução só faz sentido se o símbolo à entrada for um elemento do conjunto $follow(E)$. Caso contrário corresponde a uma situação de erro. Finalmente, Z_3 pode evoluir por ocorrência de um E, um 'a' ou um '(', que correspondem aos símbolos que aparecem imediatamente à direita do ponto. Estas evoluções são indicadas em (6.20).

$$Z_4 = \delta(Z_3, E) = \{E \rightarrow (\ E \cdot\ )\ \}$$
$$\delta(Z_3, \mathrm{a}) = Z_2$$
$$\delta(Z_3, (\ ) = Z_3 \tag{6.20}$$

Como podemos ver apenas um novo estado fora criado, este, que evolui por ocorrência de um ')', como podemos ver em (6.21).

$$Z_5 = \delta(Z_4,)\ ) = \{E \rightarrow (\ E\ ) \cdot\ \} \tag{6.21}$$

Agrupando todos os estados obtemos a **coleção de itens** seguinte (6.22).

coleção de itens

$$Z_0 = \{S \rightarrow \cdot\ E\ \$\} \cup \{E \rightarrow \cdot\ \mathrm{a},\ E \rightarrow \cdot\ (\ E\ )\ \}$$
$$Z_1 = \delta(Z_0, E) = \{S \rightarrow E \cdot \$\}$$
$$Z_2 = \delta(Z_0, \mathrm{a}) = \{E \rightarrow \mathrm{a}\ \cdot\ \}$$
$$Z_3 = \delta(Z_0, (\ ) = \{E \rightarrow (\ \cdot\ E\ )\ \} \cup \{E \rightarrow \cdot\ \mathrm{a},\ E \rightarrow \cdot\ (\ E\ )\ \}$$
$$Z_4 = \delta(Z_3, E) = \{E \rightarrow (\ E \cdot\ )\ \}$$
$$Z_5 = \delta(Z_4,)\ ) = \{E \rightarrow (\ E\ ) \cdot\ \} \tag{6.22}$$

Tabela de reconhecimento

A coleção de conjuntos de itens (conjuntos de estados) fornece a base para a construção de uma tabela usada no algoritmo de reconhecimento. A **tabela de reconhecimento** é uma matriz dupla, em que as linhas são indexadas pelo alfabeto da pilha (coleção de conjunto de itens da pilha) e as colunas são indexadas pelos símbolos terminais e não-terminais da gramática. Representam-se assim, simultaneamente, duas funções, designadas de ACTION e GOTO. A função ACTION tem como argumentos um estado (símbolo da pilha) e um símbolo terminal (incluindo o $) e define a ação a realizar (entre deslocamento, redução, aceitação ou rejeição). Por outro lado, a função GOTO mapeia um estado e um símbolo não-terminal num estado. É usada após uma operação de redução. Veja-se assim um exemplo. Considerando a gramática em (6.17) e a coleção de conjunto de itens em (6.22), podemos obter a seguinte tabela de reconhecimento na Figura 6.10.

tabela de reconhecimento

figura 6.10

Z	a	(	)	$	E
Z_0	desl, Z_2	desl, Z_3			Z_1
Z_1				**aceitação**	
Z_2			red, E → a	red, E → a	
Z_3	desl, Z_2	desl, Z_3			Z_4
Z_4			desl, Z_5		
Z_5			red, E → (E)	red, E → (E)	

Na Figura 6.10, em suma, temos que: Z representa o alfabeto da pilha, este, obtido pelo cálculo da coleção de conjuntos de itens; sempre que temos "desl, Z_n" temos um deslocamento, no qual é consumido o símbolo à entrada e é feito o empilhamento do símbolo Z_n; sempre que temos "reduce, $A \to \alpha$" pretende-se representar uma redução, na qual são retirados da pilha tantos símbolos quantos os símbolos do corpo da regra; os símbolos Z da última coluna, representam os símbolos a empilhar após uma redução; as células vazias representam situações de erro.

Algoritmo de reconhecimento

No Código 6.4 podemos ver o algoritmo para a criação da tabela da Figura 6.10. Nele, sempre que se referem *top*, *pop* e *push* pretende-se apontar para as funções de manipulação das pilhas e, sempre que se referirem *lookahead* e *adv* pretende-se apontar para funções de manipulação de entrada que, respetivamente, devolve o próximo símbolo terminal à entrada e consome um símbolo.

código 6.4

```
push(Z0) ;
para sempre
    se top() == Z1 e lookahead() == $ :
        aceita a entrada como pertencendo à linguagem ;
    acc = tabela[top(), lookahead()] ;
    se acc é deslocamento Zi :
        adv() ;
        push(Zi) ;
    caso contrário, se acc é redução A → α :
        pop() |α| símbolos ;
        push(tabela[top(), A]) ;
    caso contrário :
        rejeita a entrada ;
```

A aplicação deste algoritmo à palavra, por exemplo, '((a))' resulta na tabela da Figura 6.11[2].

figura 6.11

pilha	entrada	ação
Z_0	((a)) $	deslocamento Z_3
$Z_0\ Z_3$	(a)) $	deslocamento Z_3
$Z_0\ Z_3\ Z_3$	a)) $	deslocamento Z_2
$Z_0\ Z_3\ Z_3\ Z_2$	)) $	reduzir E → a
$Z_0\ Z_3\ Z_3$	)) $	goto Z_4
$Z_0\ Z_3\ Z_3\ Z_4$	)) $	deslocamento Z_5
$Z_0\ Z_3\ Z_3\ Z_4\ Z_5$	) $	reduzir E → (E)
$Z_0\ Z_3$	) $	goto Z_4
$Z_0\ Z_3\ Z_4$	) $	deslocamento Z_5
$Z_0\ Z_3\ Z_4\ Z_5$	$	reduzir E → (E)
Z_0	$	goto Z_1
$Z_0\ Z_1$	$	**aceitação**

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

7. Limpeza de Gramáticas

Consideremos uma gramáica $G = (T, N, P, S)$. Nesta gramática designamos um símbolo não-terminal A como **produtivo** se e só se for possível transformá-lo numa expressão contendo apenas símbolos terminais. Isto é, A é produtivo se $A \Rightarrow^* u \wedge u \in T^*$. Caso contrário, diz-se que A é **improdutivo**. Uma gramática, por conseguinte, diz-se improdutiva se o seu símbolo inicial for improdutivo. Sobre o alfabeto $T = \{a, b, c\}$ consideremos para o efeito a gramática em (7.1).

produtivo

improdutivo

$$S \to \mathrm{a\ b} \mid \mathrm{a}\ S\ \mathrm{b} \mid X$$
$$X \to \mathrm{c}\ X \qquad (7.1)$$

Como referido, dizemos então que S é produtivo, porque $S \Rightarrow \mathrm{a\ b} \wedge \mathrm{a\ b} \in T^*$. Em contrapartida, podemos dizer que X é improdutivo porque é impossível transformar X numa sequência de símbolos terminais, como podemos ver em (7.2).

$$X \Rightarrow \mathrm{c}X \Rightarrow \mathrm{cc}X \Rightarrow^* \mathrm{c}\ldots\mathrm{c}X \qquad (7.2)$$

Conjunto dos símbolos produtivos

Seja novamente $G = (T, N, P, S)$ uma gramática qualquer. O conjunto dos **símbolos produtivos**, N_p, pode ser obtido por aplicação das seguintes regras construtivas de (7.3).

símbolos produtivos

$$\textbf{se}\ (A \to \alpha) \in P\ \textbf{e}\ \alpha \in T^*\ \textbf{então}\ A \in N_p\ ; \qquad (7.3)$$
$$\textbf{se}\ (A \to \alpha) \in P\ \textbf{e}\ \alpha \in (T \cup N_p)^*\ \textbf{então}\ A \in N_p\ ;$$

2 No preenchimento da tabela optou-se por separar em duas linhas as operações de pop e de push das ações de redução. Desta forma fica mais claro que o símbolo a empilhar resulta do símbolo no topo da pilha após os pop's.

Começando com um N_p igual ao resultado da aplicação da primeira regra a todas as produções da gramática e extendendo depois esse conjunto por aplicação sucessiva da segunda regra obtem-se o conjunto de todos os símbolos produtivos de G. O algoritmo seguinte em Código 7.1 executa esse procedimento.

código 7.1

seja $N_p = \emptyset$, $P_p = P$;
repetir :
  $nadaAdicionado$ = True ;
  para cada $(A \to \alpha) \in P_p$:
    se $\alpha \in (T \cup N_p)^*$:
      se $A \notin N_p$:
        $N_p = N_p \cup \{A\}$;
        $nadaAdicionado$ = False ;
até que $nadaAdicionado$ **ou** $N_p = N$;

No Código 7.1 N_p representa o conjunto dos símbolos produtivos já identificados e P_p o conjunto das produções contendo símbolos ainda não identificados como produtivos. Se numa iteração nenhum símbolo for marcado como produtivo o algoritmo para, sendo o conjunto dos símbolos produtivos o conjunto N_p tido nesse momento. Obviamente que o algoritmo também para, se no fim de uma iteração, $N_p = N$, isto é, se todos os símbolos foram marcados como produtivos.

Símbolos acessíveis e não-acessíveis

Com as mesmas considerações de gramática da secção anterior, um símbolo terminal x diz-se **acessível** se for possível transformar S (o símbolo inicial) numa expressão que contenha x, isto é, se $S \Rightarrow^* \alpha x \beta$. Caso tal não se consiga comprovar diz-se que x é **inacessível** (ou não-acessível). Consideremos então para o efeito a gramática de (7.4).

acessível

inacessível

$$\begin{aligned} S &\to \varepsilon \mid \text{a}\ S\ \text{b} \mid \text{c}\ C\ \text{c} \\ C &\to \text{c}\ S\ \text{c} \\ D &\to \text{d}\ X\ \text{d} \\ X &\to C\ C \end{aligned} \quad (7.4)$$

Se verificarmos bem, podemos reparar que é impossível transformar S numa expressão que contenha D, d ou X, pelo que estes símbolos são inacessíveis. Já os restantes símbolos são acessíveis.

Podemos também considerar o conjunto dos **símbolos acessíveis**, V_A, este podendo ser obtido através das regras produtivas de (7.5).

símbolos acessíveis

$S \in V_A$;
se $A \to \alpha B \beta \in P$ **e** $A \in V_A$ **então** $B \in V_A$; (7.5)

Começando com $V_A = \{S\}$ e aplicando sucessivamente a segunda regra de (7.5), temos que até que ela não acrescente nada a V_A obtém-se o conjunto dos símbolos acessíveis. O algoritmo seguinte de Código 7.2 executa esse procedimento. Nele, V_A representa o conjunto dos símbolos acessíveis já identificados e N_x o conjunto dos símbolos não-terminais acessíveis já identificados mais ainda não processados. No fim, quando N_x for o conjunto vazio, V_A contém já todos os símbolos acessíveis.

código 7.2

seja $V_A = \{S\}$, $N_x = V_A$;
repetir :
  seja A = um elemento de N_x ;

$N_x = N_x \setminus \{A\}$;
para cada $(A \to \alpha) \in P$:
para cada (x) **em** α :
se $x \notin V_A$:
$V_A = V_A \cup \{A\}$;
se $x \in N$:
$N_x = N_x \cup \{A\}$;
até que $N_x = \emptyset$;

Gramáticas limpas

gramática limpa
inúteis

Uma **gramática limpa** é tal que, possuindo todas as capacidades de uma gramática, não tenha símbolos inacessíveis ou improdutivos, estes, considerados **inúteis**, porque não contribuem para as palavras que a gramática pode gerar. Se tais símbolos forem removidos obtém-se uma gramática equivalente, em termos da linguagem que descreve. Para limpar as gramáticas devemos então limpar primeiro os símbolos improdutivos e só depois os inacessíveis.

E assim termina a disciplina de Linguagens Formais e Autómatos (a2s2) dando continuidade na aplicação destes conhecimentos em Introdução à Inteligência Artificial (a3s1) - não diretamente, mas sendo uma das possíveis aplicações.

1. Introdução ao processamento de linguagens

Aproximação ao conceito de linguagem 2
Conceitos básicos sobre linguagens 2
Operações sobre palavras 3
Operações sobre linguagens 3
Introdução ao conceito de gramática 4

2. Linguagens Regulares e Expressões Regulares

Linguagens regulares 6
Gramáticas regulares 7
Conversão de expressões regulares em gramáticas regulares 9

3. Autómatos Finitos

Autómatos finitos deterministas (AFD) 11
Autómatos finitos não-deterministas (AFND) 13
Equivalência entre autómatos finitos deterministas e não-deterministas 16
Conversão de uma expressão regular num AFND 16
Conversão de um autómato finito numa expressão regular 18

4. Gramáticas Independentes do Contexto (GIC)

Introdução às gramáticas independentes do contexto 19
Derivação por regras 20
Ambiguidade gramatical 21

5. Gramáticas de Atributos

Análise de expressões aritméticas e declarações 23
Tipos de atributos e herança 25
Avaliação dirigida pela sintaxe e grafo de dependências 26

6. Autómatos de Pilha

Transformações em gramáticas independentes do contexto 27
Fatorização à esquerda 28
Os conjuntos first, follow e predict 29
Análise sintática descendente 30
Análise sintática ascendente 32
Construção de um reconhecedor ascendente 34
Tabela de reconhecimento 36
Algoritmo de reconhecimento 36

7. Limpeza de Gramáticas

Conjunto dos símbolos produtivos 37
Símbolos acessíveis e não-acessíveis 38
Gramáticas limpas 39

Apontamentos de Linguagens Formais e Autómatos
1ª edição - junho de 2016

lfa

Autor: Rui Lopes
Fontes bibliográficas: Compilers: Principles, Techniques, & Tools, AHO, Alfred V., LAM, Monica S., SETHI, Ravi, ULLMAN, Jeffrey D., 2nd Edition; Linguagens Formais e Autómatos: apontamentos, PEREIRA, Artur, Fevereiro de 2016; Introduction to Theory of Computation, MAHESHWARI, Anil, SMID, Michiel, School of Computer Science, Ottawa, 2016.
Outros recursos: Slides das aulas teóricas;
Agradecimentos: Professor Artur Pereira.

Todas as ilustrações gráficas são obra de Rui Lopes e as imagens são provenientes das fontes bibliográficas divulgadas.

apontamentos

© Rui Lopes 2016 Copyright: Pela Creative Commons, não é permitida a cópia e a venda deste documento. Qualquer fraude será punida. Respeite os autores e as suas marcas. Original - This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/deed.en_US.