Autômatos e Linguagens Formais

S. C. Coutinho
Universidade Federal do Rio de Janeiro

Agradeço a

- David Boechat
- Gabriel Rosário

pelas correções às notas de aula.

Sumário

CAPíTULO 1

Conjuntos e linguagens

Neste primeiro capítulo, revisamos algumas propriedades básicas dos conjuntos e suas operações e introduzimos o conceito de linguagem formal.

1. Exercícios

1. Sejam A e B conjuntos, prove que:
   (a) $\emptyset \cup A = A$;
   (b) $\emptyset \cap A = \emptyset$;
   (c) se $A \subset B$ então $A \cap B = A$;
   (d) se $A \subset B$ então $A \cup B = B$;
   (e) $A \cap A = A = A \cup A$;
   (f) $A \cup B = B \cup A$ e $A \cap B = B \cap A$;
   (g) $A \cup (B \cup C) = (A \cup B) \cup C$;
   (h) $A \cap (B \cap C) = (A \cap B) \cap C$;
   (i) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$;
   (j) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$;
   (k) $A \setminus B = A \cap \overline{B}$ onde $\overline{B}$ é o complemento de B no conjunto *universo*, isto é o conjunto que contém todos os conjuntos com que estamos trabalhando;
   (l) $\overline{(A \cap B)} = \overline{A} \cup \overline{B}$;
   (m) $\overline{(A \cup B)} = \overline{A} \cap \overline{B}$;
   (n) $\overline{\overline{A}} = A$;
   (o) $A \setminus B = A \setminus (B \cap A)$;
   (p) $B \subset \overline{A}$ se, e somente se, $A \cap B = \emptyset$;
   (q) $(A \setminus B) \setminus C = (A \setminus C) \setminus (B \setminus C) = A \setminus (B \cup C)$;
   (r) $A \cap \overline{B} = \emptyset$ e $\overline{A} \cap B = \emptyset$ se e somente se $A = B$.

2. Considere as afirmações abaixo: prove as verdaeiras e dê um contra-exemplo para as falsas.
   (a) se $A \cup B = A \cup C$ então $B = C$;

(b) se $A \cap B = A \cap C$ então $B = C$.

3. Sejam A, B e C conjuntos, prove que:
   (a) $A \times (B \cap C) = (A \times B) \cap (A \times C)$;
   (b) $A \times (B \cup C) = (A \times B) \cup (A \times C)$;
   (c) $A \times (B \setminus C) = (A \times B) \setminus (A \times C)$;

4. Explique a diferença entre $\emptyset$ e $\{\epsilon\}$. Calcule $L \cdot \emptyset$, onde L é uma linguagem qualquer.

5. Seja w uma palavra em um alfabeto Σ. Definimos o *reflexo* de w recursivamente da seguinte maneira: $\epsilon^R = \epsilon$ e se $w = ax$ então $w^R = x^R a$ onde $a \in \Sigma$.
   (a) Determine $(turing)^R$ e $(anilina)^R$.
   (b) Se x e y são palavras no alfabeto Σ, determine $(xy)^R$ em função de x^R e y^R.
   (c) Determine $(x^R)^R$.

6. Sejam L_1 e L_2 linguagens no alfabeto Σ. Determine as seguintes linguagens em função de L_1^R e L_2^R:
   (a) $(L_1 \cdot L_2)^R$;
   (b) $(L_1 \cup L_2)^R$;
   (c) $(L_1 \cap L_2)^R$;
   (d) $\overline{L_1}^R$;
   (e) $(L_1^*)^R$.

7. Mostre, por indução em n, que se $L_0, \ldots, L_n$ são linguagens no alfabeto Σ então
$$L_0 \cdot (L_1 \cup \cdots \cup L_n) = (L_0 \cdot L_1) \cup \cdots \cup (L_0 \cdot L_n).$$

8. Sejam Σ_1 e Σ_2 dois alfabetos e seja $\phi : \Sigma_1 \to \Sigma_2^*$ uma aplicação. Estenda ϕ a Σ_1^* de acordo com a seguinte definição recursiva:
   - $\phi(\epsilon) = \epsilon$;
   - $\phi(x\sigma) = \phi(a)\phi(\sigma)$, onde $a \in \Sigma_1^*$.

   Se L é uma linguagem no alfabeto Σ_1 defina
$$\phi(L) = \{\phi(w) : w \in \Sigma_1^*\}.$$
   Mostre que se L e L' são linguagens no alfabeto Σ_1 então:
   (a) $\phi(L \cup L') = \phi(L) \cup \phi(L')$;
   (b) $\phi(L \cap L') = \phi(L) \cap \phi(L')$;
   (c) $\phi(L \cdot L') = \phi(L) \cdot \phi(L')$;

CAPíTULO 2

Autômatos finitos determinísticos

Neste capítulo introduzimos a noção de autômato finito determinístico através de um exemplo concreto e estudamos alguns outros exemplos que ocorrerão freqüentemente ao longo do livro.

1. Exercícios

1. Seja $\mathcal{A}$ um autômato finito determinístico. Quando é que $\epsilon \in L(\mathcal{A})$?

2. Desenhe o grafo de estados e determine a linguagem aceita por cada um dos seguintes autômatos finitos. Em cada caso o estado inicial é q_1 e o alfabeto é $\{a, b, c\}$
   (a) $F_1 = \{q_5\}$ e a função de transição é dada por:

δ_1	a	b	c
q_1	q_2	q_3	q_4
q_2	q_2	q_4	q_5
q_3	q_4	q_3	q_5
q_4	q_4	q_4	q_5
q_5	q_4	q_4	q_5

   (b) $F_2 = \{q_4\}$ e $\delta_2 = \delta_1$.
   (c) $F_3 = \{q_2\}$ e a função de transição é dada por:

δ_3	a	b	c
q_1	q_2	q_2	q_1
q_2	q_3	q_2	q_1
q_3	q_1	q_3	q_2

3. Considere o autômato finito determinístico no alfabeto $\{a, b\}$, com estados $\{q_0, q_1\}$, estado inicial q_0, estados finais $F = \{q_1\}$ e cuja função de transição é dada por:

δ	a	b
q_0	q_0	q_1
q_1	q_1	q_0

(a) Esboce o diagrama de estados deste autômato.
(b) Descreva a computação deste autômato que tem início na configuração $(q_0, aabba)$. Esta palavra é aceita pelo autômato?
(c) Descreva a computação deste autômato que tem início na configuração $(q_0, aabbab)$. Esta palavra é aceita pelo autômato?
(d) Descreva em português a linguagem aceita pelo autômato definido acima?

4. Seja Σ um alfabeto com n símbolos. Quantos autômatos finitos determinísticos existem com alfabeto Σ e $m > 0$ estados?
Sugestão: Não esqueça de considerar todas as possibilidades para o conjunto de estados finais.

5. Invente autômatos finitos determinísticos que aceitem as seguintes linguagens sobre o alfabeto $\{0, 1\}$:
(a) o conjunto das palavras que acabam em 00;
(b) o conjunto das palavras com três 0s consecutivos;
(c) o conjunto das palavras em que cada 0 está entre dois 1s;
(d) o conjunto das palavras cujos quatro símbolos finais são 1101;
(e) o conjunto dos palíndromos de comprimento igual a 6.

6. Dê exemplo de uma linguagem que é aceita por um autômato finito determinístico com *mais de um estado final*, mas que *não* é aceita por nenhum autômato finito determinístico com *apenas um estado final*. Justifique cuidadosamente sua resposta.

CAPíTULO 3

Expressões regulares

É hora de abordarmos o primeiro dos problemas propostos ao final do capítulo anterior; isto é, como determinar a linguagem aceita por um autômato finito? De quebra, descobriremos uma maneira de caracterizar estas linguagens.

1. Introdução

Desejamos obter um algoritmo que, dado um autômato finito $\mathcal{M}$, determine a linguagem $L(\mathcal{M})$ que ele aceita. O algoritmo é recursivo, e para poder descrevê-lo começamos por generalizar a noção de linguagem aceita por um autômato.

Seja $\mathcal{M}$ um autômato finito definido pelos ingredientes $(\Sigma, Q, q_1, F, \delta)$. Para cada $q \in Q$ definimos L_q como sendo a linguagem

$$L_q = \{w \in \Sigma^* : (q, w) \vdash^* (f, \epsilon) \text{ onde } \ f \in F\}.$$

Em outras palavras, L_q é formada pelas palavras que levam o autômato do estado q a algum estado final. Quando os estados do autômato forem numerados como $q_1, \ldots, q_n$, escrevermos L_i em vez de L_{q_i} para simplificar a notação. Portanto, se q_1 é o estado inicial de $\mathcal{M}$, então

$$L_{q_1} = L_1 = L(\mathcal{M}).$$

Sejam p e q estados de $\mathcal{M}$, e digamos que $\delta(p, \sigma) = q$. Esta transição estabelece uma relação entre as lingaugens L_p e L_q. De fato, temos que $(p, \sigma) \vdash (q, \epsilon)$ ao passo que, se $w \in L_q$, então $(q, w) \vdash^* (f, \epsilon)$. Combinado estas duas computações obtemos

$$(p, \sigma w) \vdash (q, w) \vdash^* (f, \epsilon).$$

Portanto, $\{\sigma\}L_q \subseteq L_p$. Mais uma vez, com a finalidade de não sobrecarregar a notação, eliminaremos as chaves, escrevendo simplesmente $\sigma L_q \subseteq L_p$.

A não ser que o alfabeto Σ tenha apenas um símbolo, não há a menor chance de que $\sigma L_q \subseteq L_p$ seja uma igualdade. Isto porque teremos uma inclusão como esta para cada $\sigma \in \Sigma$. Portanto, se $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ e se $\delta(p, \sigma_i) = q_i$, então

$$\bigcup_{i=1}^{n} \sigma_i L_{q_i} \subseteq L_p.$$

Desta vez, porém, a inclusão é, de fato, uma igualdade, desde que p não seja um estado final do autômato! Para ver isto suponhamos que $u \in L_p$. Podemos isolar o primeiro símbolo de u, escrevendo $u = \sigma_i w$, para algum $\sigma_i \in \Sigma$. Mas, pela definição de L_p,

$$(p, u) = (p, \sigma_i w) \vdash^* (f, \epsilon),$$

para algum $f \in F$. Por outro lado, como $\delta(p, \sigma_i) = q_i$, temos que $(p, \sigma_i) \vdash (q_i, \epsilon)$. Combinando estas duas computações, temos que

$$(p, u) = (p, \sigma_i w) \vdash (q_i, w) \vdash^* (f, \epsilon),$$

de onde segue que $w \in L_{q_i}$.

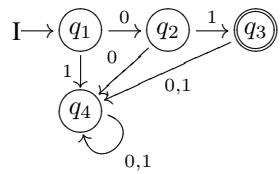
Precisamos ainda analisar o que acontece quando p é um estado final do autômato. Considerando em detalhe o argumento do parágrafo acima, vemos que assumimos implicitamente que $u \neq \epsilon$, já que estamos explicitando o seu primeiro símbolo. Entretanto, se p for um estado final, teremos, além disso, que $\epsilon \in L_p$.

Resumindo, provamos que, se $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$, e se $\delta(p, \sigma_i) = q_i$, então

$$L_p = \begin{cases} \bigcup_{i=1}^{n} \sigma_i L_{q_i} & \text{se } p \notin F \\ \bigcup_{i=1}^{n} \sigma_i L_{q_i} \cup \{\epsilon\} & \text{se } p \in F \end{cases}$$

O algoritmo que desejamos segue diretamente desta equação, como mostra o seguinte exemplo.

Considere o autômato finito determinístico $\mathcal{M}$ com alfabeto é $\{0, 1\}$ e cujo grafo é



Deste grafo extraímos as seguintes equções

$$\begin{aligned} L_1 &= 0L_2 \cup 1L_4 \\ L_2 &= 1L_3 \cup 0L_4 \\ L_3 &= 0L_4 \cup 1L_4 \cup \{\epsilon\} \text{ (já que } q_3 \text{ é estado final)} \\ L_4 &= 0L_4 \cup 1L_4. \end{aligned}$$

Observe que q_4 é um estado morto, de modo que nada escapa de q_4. Em particular, nenhuma palavra leva o autômato de q_4 a um estado final. Portanto, $L_4 = \emptyset$. Isto simplifica drasticamente as equações anteriores, que passam a ser

$$\begin{aligned} L_1 &= 0L_2 \\ L_2 &= 1L_3 \\ L_3 &= \{\epsilon\} \\ L_4 &= \emptyset. \end{aligned}$$

Podemos agora resolver este sistema de equações por mera substituição. Assim, substituindo a terceira equação na segunda, obtemos

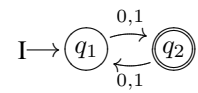
$$L_2 = 1L_3 = 1\{\epsilon\} = \{1\}.$$

Finalmente, substituindo esta última equação na primeira, obtemos

$$L_1 = 0L_2 = 1\{0\} = \{10\}.$$

Como $L_1 = L(\mathcal{M})$, obtivemos uma descrição da linguagem aceita por $\mathcal{M}$.

Como seria de esperar, as coisas nem sempre são tão diretas. Afinal, o autômato deste exemplo tinha um comportamento muito simples. Vejamos o que acontece em um exemplo menos elementar. Por exemplo, seja $\mathcal{N}$ o autômato finito determinístico de alfabeto $\{0, 1\}$ e grafo



As equações correspondentes a L_1 e L_2 são:

$$\begin{aligned} L_1 &= 0L_2 \cup 1L_2 \\ L_2 &= 0L_1 \cup 1L_1 \cup \{\epsilon\} \text{ (já que } q_2 \text{ é estado final)} \end{aligned}$$

Continuando a denotar $\{0\}$ e $\{1\}$ por 0 e 1, podemos reescreer estas equações como

$$\begin{aligned} L_1 &= (0 \cup 1)L_2 \\ L_2 &= (0 \cup 1)L_1 \cup \{\epsilon\}. \end{aligned}$$

À primeira vista, tudo o que temos que fazer para resolver este sistema é substituir a segunda equação na primeira. Contudo, ao fazer isto obtemos

$$L_1 = (0 \cup 1)((0 \cup 1)L_1 \cup \{\epsilon\}),$$

ou seja,

$$L_1 = (0 \cup 1)^2 L_1 \cup (0 \cup 1), \tag{1.1}$$

de modo que L_1 fica escrito em termos do próprio L_1. Parece claro que nenhuma substituição pode nos tirar desta encrenca. O que fazer então?

2. Lema de Arden

Na verdade, ao aplicar o método de substituição para resolver sistemas de equações e achar a linguagem aceita por um autômato vamos nos deparar muitas vezes com equações em que uma linguagem é escrita em termos dela própria. Equações que serão, freqüentemente, ainda mais complicadas que (1.1). Convém, portanto, abordar deste já o problema em um grau de generalidade suficiente para dar cabo de todas as equações que apareçam como resultado do algoritmo de substituição.

Para isso, suponhamos que Σ é um alfabeto e que A e B são linguagens em Σ. Digamos, que X seja uma outra linguagem em Σ que satisfaça

$$X = AX \cup B.$$

O problema que queremos resolver consiste em usar esta equação para determinar X.

Uma coisa que podemos fazer é substituir $X = AX \cup B$ de volta nela própria. Isso dá

$$X = A(AX \cup B) \cup B = A^2X \cup (A \cup \epsilon)B, \tag{2.1}$$

e não parece adiantar de nada porque afinal de contas continuamos com um X do lado direito da equação. Mas não vamos nos deixar abater por tão pouco: façamos a substituição mais uma vez. Desta vez substituíremos $X = AX \cup B$ em (2.1), o que nos dá

$$X = A^2(AX \cup B) \cup (AB \cup B) = A^3X \cup (A^2 \cup A \cup \epsilon)B.$$

Repetindo o mesmo procedimento k vezes, chegamos à equação

$$X = A^{k+1}X \cup (A^k \cup A^{k-1} \cup \cdots \cup A^2 \cup A \cup \epsilon)B.$$

O problema é que o X continua presente. Mas, e se repetíssemos o processo infinitas vezes? Neste caso, "perderíamos de vista o termo que contém X que seria empurrado para o infinito", e sobraria apenas

$$X = (\epsilon \cup A \cup A^2 \cup \cdots)B. \tag{2.2}$$

Para poder fazer isto de maneira formal precisamos introduzir uma nova operação com linguagens, a *estrela de Kleene*. Em geral, se A é uma linguagem no alfabeto Σ, então A^* é definida como a reunião de todas as potências de A; isto é,

$$A^* = \{\epsilon\} \cup A \cup A^2 \cup A^3 \cup \cdots .$$

Por exemplo, se $\Sigma = \{0, 1\}$ e $A = \{01\}$, então

$$A^* = \{(01)^j : j \geq 0\} = \{\epsilon, 01, 0101, 010101, \dots\}.$$

Por outro lado, se $A = \Sigma$, então A^* é o conjunto de todas as palavras no alfabeto Σ–que aliás já vínhamos denotando por Σ^*.

A equação (2.2) sugere que, continuando o processo de substituição indefinidamente, deveríamos obter $X = A^*B$. Este é um conjunto perfeitamente bem definido, resta-nos verificar se realmente é uma solução da equação

$X = AX \cup B$. Para isso substituíremos X por A^*B do lado direito da equação:

$$AX \cup B = A(A^*B) \cup B = AA^*B \cup B.$$

Como $A^* = \{\epsilon\} \cup A \cup A^2 \cup A^3 \cup \cdots$, obtemos

$$AA^*B \cup B = A(\{\epsilon\} \cup A \cup A^2 \cup A^3 \cup \cdots)B \cup B,$$

que dá

$$AA^*B \cup B = (A \cup A^2 \cup A^3 \cup A^4 \cup \cdots)B \cup B.$$

Pondo B em evidência em todo o lado direito, obtemos

$$AA^*B \cup B = (\epsilon \cup A \cup A^2 \cup A^3 \cup A^4 \cup \cdots)B = A^*B.$$

Concluímos que $A(A^*B) \cup B = A^*B$, de modo que A^*B é, de fato, uma solução da equação $X = AX \cup B$.

Infelizmente, isto ainda não é suficiente para completar os cálculos do algoritmo de substituição. O problema é que obtivemos uma solução da equação desejada, mas ainda não sabemos se esta solução corresponde ao maior conjunto $X \subseteq \Sigma^*$ que satisfaz $X = AX \cup B$. Se não for este o caso, quando usarmos A^*B como solução estaremos perdendo algumas palavras do conjunto aceito pelo autômato, o que não queremos que aconteça.

Suponhamos, então, que X é o maior subconjunto de Σ^* que satisfaz $X = AX \cup B$. Como já sabemos que A^*B satisfaz esta equação, podemos escrever $X = A^*B \cup C$, onde C é um conjunto (disjunto de A^*B) que contém as possíveis palavras excedentes. Substituindo $X = A^*B \cup C$ em $X = AX \cup B$, temos

$$A^*B \cup C = A(A^*B \cup C) \cup B = A^*B \cup AC, \tag{2.3}$$

já que, como vimos, $A(A^*B) \cup B = A^*B$. Intersectando ambos os membros de (2.3) com C, e lembrando que, por hipótese, $C \cap A^*B = \emptyset$, chegamos a

$$C = AC \cap C = (A \cap \epsilon)C.$$

Temos, então, duas possibilidades. A primeira é que A não contenha ϵ. Neste caso $A \cap \epsilon = \emptyset$, de modo que $C = \emptyset$; ou seja, A^*B é o maior conjunto solução da equação $X = AX \cup B$. A outra possibilidade é que A contenha ϵ, e neste caso estamos encrencados. Por sorte, esta segunda possibilidade *nunca* ocorre na solução das equações que advêm do algoritmo de substituição! Você pode confirmar isto lendo a demonstração detalhada do algoritmo de substituição na seção 5. Vamos resumir tudo o que fizemos em um lema, provado originalmente por D. N. Arden em 1960.

LEMA DE ARDEN. *Sejam A e B linguagens em um alfabeto Σ. Se $\epsilon \notin A$ então o maior subconjunto de Σ^* que satisfaz $X = AX \cup B$ é $X = A^*B$.*

Vamos aplicar o que aprendemos para resolver a equação (1.1), que resultou da aplicação do método de substituição ao segundo exemplo da seção anterior. A equação é

$$L_1 = (0 \cup 1)^2 L_1 \cup (0 \cup 1).$$

Aplicando o Lema de Arden com $X = L_1$, $A = (0 \cup 1)^2$ e $B = (0 \cup 1)$, teremos que

$$L_1 = X = A^*B = ((0 \cup 1)^2)^*(0 \cup 1).$$

Concluímos, assim, que a linguagem aceita pelo autômato $\mathcal{N}$ é

$$L(\mathcal{N}) = ((0 \cup 1)^2)^*(0 \cup 1).$$

3. O algoritmo de substituição

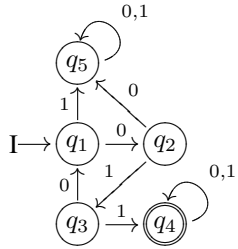
Antes de fazer outro exemplo, vamos descrever em mais detalhes o algoritmo de substituição que usamos para determinar a linguagem aceita pelos autômatos da seção 1. Este algoritmo foi inventado por J. A. Brzozowski em 1964.

Algoritmo de substituição

Entrada: ingredientes $(\Sigma, Q, q_1, F, \delta)$ de um autômato finito determinístico $\mathcal{M}$.
Saída: uma descrição da linguagem aceita por $\mathcal{M}$.

Primeira etapa: Seja $Q = \{q_1, \ldots, q_n\}$. Escreva as equações para as linguagens L_j, para cada $1 \leq j \leq n$.
Segunda etapa: Começando por L_n e acabando em L_1, substitua L_{j+1} na equação para L_j, aplicando o Lema de Arden sempre que uma linguagem for expressa em termos dela própria.
Terceira etapa: A linguagem aceita por $\mathcal{M}$ corresponde à expressão obtida para L_1.

Uma descrição detalhada deste algoritmo, e uma demonstração de que faz o que é pedido, pode ser encontrada na seção 5. Por enquanto, nos contentaremos com a descrição acima, que vamos aplicar ao autômato no alfabeto $\{0, 1\}$ cujo grafo é



As equações correspondents a autômato são

$$\begin{aligned}
L_1 &= 0L_2 \cup 1L_5 \\
L_2 &= 1L_3 \cup 0L_5 \\
L_3 &= 0L_1 \cup 1L_4 \\
L_4 &= 0L_4 \cup 1L_4 \cup \epsilon \\
L_5 &= 0L_5 \cup 1L_5.
\end{aligned}$$

Uma olhada no grafo mostra que q_5 é um estado morto, de modo que $L_5 = \emptyset$. Com isto as equações se simpificam:

$$\begin{aligned} L_1 &= 0L_2 \\ L_2 &= 1L_3 \\ L_3 &= 0L_1 \cup 1L_4 \\ L_4 &= 0L_4 \cup 1L_4 \cup \epsilon \\ L_5 &= \emptyset. \end{aligned}$$

A equação para L_4 nos dá

$$L_4 = (0 \cup 1)L_4 \cup \epsilon,$$

de modo que precisamos aplicar o Lema de Arden. fazendo isto obtemos

$$L_4 = (0 \cup 1)^* \epsilon = (0 \cup 1)^*.$$

Substituindo em L_3,

$$L_3 = 0L_1 \cup 1(0 \cup 1)^*.$$

De modo que, da segunda equação, segue que

$$L_2 = 1(0L_1 \cup 1(0 \cup 1)^*) = 10L_1 \cup 11(0 \cup 1)^*.$$

Com isso, resulta da primeira equação que

$$L_1 = 010L_1 \cup 011(0 \cup 1)^*.$$

Usando o Lema de Arden mais uma vez,

$$L_1 = 010^*(011(0 \cup 1)^*),$$

que é a linguagem aceita pelo autômato.

Antes de encerrar a seção precisamos fazer algumas considerações sobre a aplicação do Lema de Arden.

Em primeiro lugar, o que aconteceria se não tivéssemos notado que q_5 é um estado morto? Neste caso teríamos de confrontar a equação $L_5 = 0L_5 \cup 1L_5$, ou seja $L_5 = (0 \cup 1)L_5$. Como L_5 aparece dos dois lados da equação, será necessário aplicar o Lema de Arden. Note que, neste caso $X = L_5$, $A = (0 \cup 1)$ e $B = \emptyset$, de modo que

$$L_5 = X = (0 \cup 1)^* \emptyset = \emptyset,$$

que é o resultado esperado.

O segundo comentário diz respeito à aplicação do Lema de Arden à equação

$$L_4 = 0L_4 \cup 1L_4 \cup \epsilon.$$

Na aplicação que fizemos anteriormente, tomamos $A = (0 \cup 1)$ e $B = \epsilon$. Mas o que aconteceria se escolhéssemos $A = 0$ e $B = 1L_4 \cup \epsilon$? Neste caso,

$$L_4 = 0^*(1L_4 \cup \epsilon) = 0^*1L_4 \cup 0^*,$$

e continuamos com L_4 dos dois lados da equação. Mas, ao invés de nos deixar intimidar, aplicaremos o Lema de Arden a esta última equação, o que nos dá

$$L_4 = (0^*1)^*0^*. \tag{3.1}$$

Note que se isto estiver correto (e está!) então devemos ter que os conjuntos $(0^*1)^*0^*$ e $(0 \cup 1)^*$ são iguais–quer dizer, têm os mesmos elementos. Portanto, deve ser possível mostrar que toda palavra no alfabeto $\{0, 1\}$ pertence ao conjunto $(0^*1)^*0^*$. Fica por sua conta se convencer disto. Finalmente, se adotarmos esta última maneira de expressar L_4, a descrição da linguagem aceita pelo autômato que resulta do algoritmo de substituição é

$$010^*(011(0^*1)^*0^*).$$

Em particular, o algoritmo de substituição pode retornar linguagens diferentes, todas corretas, dependendo da maneira como for aplicado.

4. Expressões regulares

Utilizando o algoritmo de substituição sempre obtemos uma descrição bastante precisa da linguagem aceita por um autômato finito determinístico. Além disso, por causa da maneira como o algoritmo opera, a linguagem é quase sempre expressa como resultado da aplicação das operações de união, concatenação e estrela, aos conjuntos unitários formados pelos símbolos do alfabeto do autômato. As únicas excessões ocorrem quando a linguagem é vazia ou é ϵ. Formalizaremos isto em uma definição, como segue.

Seja Σ um alfabeto e seja

$$\widetilde{\Sigma} = \Sigma \cup \{\cup, \cdot, *, (,), \emptyset, \epsilon\}.$$

Consideraremos o conjunto $\widetilde{\Sigma}$ como um outro alfabeto, uma extensão de Σ. Além disso, $\cup, \cdot, *, (,), \emptyset, \epsilon$ serão considerados apenas como símbolos (isto é, seu dignificado será ignorado) quando estiverem posando de elementos de $\widetilde{\Sigma}$.

Uma *expressão regular* é uma palavra no alfabeto $\widetilde{\Sigma}$, construída recursivamente pela aplicação sucessiva das seguintes regras:

(1) se $\sigma \in \Sigma$ então σ é uma expressão regular;
(2) $\emptyset$ e ϵ são expressões regulares;
(3) se r_1 e r_2 são expressões regulares, então $(r_1 \cup r_2)$ e $(r_1 \cdot r_2)$ também são;
(4) se r é uma expressão regular, então r^* também é.

Não há nada de misterioso sobre (3) e (4), elas apenas refletem a maneira correta de se usar os símbolos $\cup$, $\cdot$ e $*$, quando são interpretados como operadores de conjuntos. Portanto, se identificarmos 0, 1, ϵ e $\emptyset$ com os conjuntos $\{0\}$, $\{1\}$, $\{\epsilon\}$ e $\emptyset$, uma expressão regular $r \in \widetilde{\Sigma}^*$ corresponderá a um subconjunto $L(r)$ em Σ, a linguagem *denotada* pela expressão regular r.

Suponhamos, por exemplo, que $\Sigma = \{0, 1\}$. Neste caso

$$\widetilde{\Sigma} = \{0, 1, \cup, \cdot, *, (,), \emptyset, \epsilon\}.$$

Como 0 e 1 são expressões regulares por (1), então 0^* também é uma expressão regular por (4). Mas isto implica, por (3), que $(0^* \cdot 1)$ é regular. Usando (4) novamente, obtemos $(0^* \cdot 1)^*$, e por (3) concluímos que $((0^* \cdot 1)^* \cdot 0^*)$ é uma expressão regular. Esta é a expressão regular que denota uma das maneiras de representar a linguagem L_4 do final da seção 3. Entretanto, para obter uma expressão regular corretamente construída, precisamos acrescentar parêntesis à representação de L_4 dada pela equação (3.1). O papel dos parêntesis é apenas o de eliminar qualquer ambigüidade na interpretação das expressões.

A razão para introduzir expressões regulares como palavras em um alfabeto, em vez de pensá-las simplesmente como a descrição de uma linguagem, é que expressões regulares distintas podem denotar o mesmo conjunto. Este é o caso, por exemplo, das expressões $(0 \cup 1)^*$ e $((0^* \cdot 1)^* \cdot 0^*)$, como vimos ao final da seção anterior.

É claro que qualquer conjunto que possa ser representado a partir dos conjuntos unitários ϵ e $\sigma \in \Sigma$ e das operações de união, concatenação e estrela pode ser denotado por uma expressão regular. Em particular, se cuidarmos de pôr os parêntesis no lugar certo, o algoritmo de substituição aplicado a um autômato finito $\mathcal{M}$ sempre retorna uma expressão regular que denota a linguagem $L(\mathcal{M})$.

Resta-nos praticar um pouco a construção de uma expressão regular que denote um conjunto dado, a partir da descrição deste conjunto. Suponhamos que $\Sigma = \{a, b, c\}$. Para obter *todas as palavras* em um certo subconjunto de Σ devemos usar a estrela de Kleene. Assim,

Linguagem formada por todas as palavras	**Expressão regular**
(vazias ou não) que só contêm a	a^*
nos símbolos a, b e c	$((a \cup b) \cup c)^*$
que não contêm a	$(b \cup c)^*$
em a e cujo comprimento é par	$(a \cdot a)^*$

A última expressão merece um comentário. As palavras de comprimento par têm a forma a^{2k}, para algum $k \geq 0$ inteiro. Mas,

$$a^{2k} = (aa)^k \in \{aa\}^k.$$

Portanto, o conjunto das palavras de comprimento par será

$$\bigcup_{k \geq 0} \{aa\}^k = aa^*.$$

Já as palavras no símbolo a cujo comprimento é ímpar são da forma $a^{2k+1} = a \cdot (aa)^k$. Assim, a linguagem formada por essas palavras é denotada por $(a \cdot (a \cdot a)^*)$.

Outro exemplo interessante consiste na linguagem formada pelas palavras que contêm exatamente um a. Isto significa que os outros símbolos da palavra têm que ser bs ou cs. Como estes símbolos tanto podem aparecer antes como

depois do a, uma tal palavra será da forma uav, onde u e v são palavras que contêm apenas b e c. Isto nos remete à expressão regular $(((b \cup c)^* \cdot a) \cdot (b \cup c)^*)$.

Como as expressões regulares já estão ficando bastante complicadas, vamos suprimir os parêntesis e o ponto que denota concatenação, quando não forem absolutamente necessários à interpretação correta da expressão. Evidentemente, ao fazer isto não obtemos uma expressão regular no sentido da definição formal. Fica como exercício acrescentar os símbolos necessários para que cada uma das expressões dadas abaixo se torne uma expressão regular correta.

Duas variações, dignas de nota, do último exemplo acima são a linguagem formada por todas as palavras que contêm exatamente dois as, que corresponde a

$$(b \cup c)^* a (b \cup c)^* a (b \cup c)^*$$

e a linguagem formada por todas as palavras que contêm um número par de as, que é denotada por

$$((b \cup c)^* a (b \cup c)^* a (b \cup c)^*)^*$$

Um exemplo um pouco mais difícil é a linguagem formada pelas palavras que contêm um número ímpar de as. Precisamos adicionar um a extra à expressão acima. O problema é que este a pode aparecer em qualquer lugar da palavra, de modo que não podemos simplesmente concatená-lo no início ou no fim da expressão acima. A saída é tomar

$$((b \cup c)^* a (b \cup c)^* a (b \cup c)^*)^* a ((b \cup c)^* a (b \cup c)^* a (b \cup c)^*)^*.$$

Encerramos com três exemplos de natureza mais prática.

Um dos primeiros passos do processo de compilação de uma linguagem de programação é conhecido como *análise léxica*. Nesta etapa o compilador identifica, por exemplo, quais foram os números inteiros e as variáveis utilizadas no programa. É claro que esta é uma etapa necessária para que seja possível interpretar corretamente o programa. Na prática, isto pode ser feito com um autômato finito. Assim, para identificar as variáveis, construímos um autômato finito que aceita a linguagem que descreve as variáveis de uma linguagem. Isto é feito em duas etapas. Primeiro obtemos uma expressão regular que denote as variáveis da linguagem de programação. Em seguida, construímos um autômato finito que aceite esta linguagem.

Contudo, pôr esta estratégia em prática depende de sermos capazes de resolver algoritmicamente o seguinte problema.

PROBLEMA 3.1. *Dada uma expressão regular r, construir um autômato finito que aceite a linguagem denotada por r.*

Abordaremos este problema detalhadamente a partir do próximo capítulo. Entretanto, já estamos em condições de obter expressões regulares para as linguagens que descrevem inteiros e variáveis de um programa, como veremos a seguir.

Para começar determinaremos uma expressão regular no alfabeto $\{0, 1, \ldots, 9\}$ que denote os inteiros positivos no sistema decimal. À primeira vista pode parecer que a expressão seja simplesmente

$$(0 \cup 1 \cup \cdots \cup 9)^*.$$

O problema é que isto inclui palavras como 00000, que não correspondem a um número formado de maneira correta. A solução é não permitir que as palavras comecem com o símbolo 0, tomando

$$(1 \cup \cdots \cup 9)(0 \cup 1 \cup \cdots \cup 9)^*.$$

Já a expressão que denota os inteiros não negativos é

$$0 \cup (1 \cup \cdots \cup 9)(0 \cup 1 \cup \cdots \cup 9)^*.$$

Finalmente, suponhamos que uma certa linguagem de programação tem por variáveis todas as palavras nos símbolos $0, 1, \ldots, 9, A, B, C, \ldots, Z$ que não começam por um número inteiro. A expressão regular que denota as variáveis nesta linguagem de programação é

$$(A \cup B \cup \cdots \cup Z)(A \cup B \cup \cdots \cup Z \cup 0 \cup 1 \cup \cdots \cup 9)^*.$$

5. Análise formal do algoritmo de substituição

Nesta seção damos uma descrição detalhada do algoritmo de substituição; isto é, uma descrição cuidadosa o suficiente para servir, tanto para programar o algoritmo, quanto para provar que funciona como esperado. Na verdade, encerramos a seção justamente com uma demonstração de que o algoritmo está correto. Contudo, se você não pretende programar o algoritmo, nem sente necessidade de uma demonstração formal, talvez seja melhor pular esta seção e passar ao próximo capítulo.

Começamos com algumas definições um tanto técnicas. Para $t = 0, \ldots, m$ seja Σ_t o alfabeto definido por

$$\Sigma_t = \begin{cases} \Sigma & \text{se } \ t = 0 \\ \Sigma \cup \{\alpha_1, \ldots, \alpha_t\} & \text{se } \ t \geq 1. \end{cases}$$

Seja agora Θ_t o conjunto formado pelas expressões regulares em Σ_t da forma

$$\bigcup_{i \leq t} \eta_i \cdot \alpha_i,$$

onde $\eta_i \neq \epsilon$ é uma expressão regular em $\Sigma_0 = \Sigma$. Note que Θ_0 é o conjunto das expressões regulares em Σ.

Dado $\theta \in \Theta_t$, seja $L(\theta)$ a linguagem obtida de acordo com as seguintes regras:

(1) $L(\epsilon) = \{\epsilon\}$;
(2) $L(\emptyset) = \emptyset$;
(3) $L(\alpha_i) = L_i$;

com L_i como definido na seção 1, e se α e β são expressões regulares em Σ_t então

(4) $L(\alpha \cdot \beta) = L(\alpha) \cdot L(\beta)$;
(5) $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$;
(6) $L(\alpha^*) = L(\alpha)^*$.

Estamos, agora, prontos para dar uma descrição minuciosa do funcionamento do algoritmo.

Algoritmo de Substituição

Entrada: um autômato finito determinístico $\mathcal{A}$ cujos ingredientes são $(\Sigma, Q, q_1, F, \delta)$, onde $Q = \{q_1, \ldots, q_n\}$.

Saída: uma expressão regular que denota a linguagem aceita por $\mathcal{A}$.

Etapa 1: Para cada estado q_i escreva uma expressão regular em Σ_m da forma

$$E_i = \bigcup \{(\sigma_j \cdot \alpha_{j(i)}) : \delta(q_i, \sigma_j) = q_{j(i)}\}$$

se q_m não é estado final, ou

$$E_i = \bigcup \{(\sigma_j \cdot \alpha_{j(i)}) : \delta(q_i, \sigma_j) = q_{j(i)}\} \cup \{\epsilon\}$$

se q_m é estado final; e inicialize $k = m$.

Etapa 2: Se E_k já está escrito como uma expressão regular em Θ_{k-1}, vá para (3), senão E_k é da forma

$$E_k = \eta \cdot \alpha_k \cup B$$

onde $B \in \Theta_{k-1}$. Neste caso escreva $E_k = \eta^* \cdot B$ e vá para para a Etapa 3. Observe que pode acontecer que $B = \emptyset$; se isto ocorrer então $E_k = \emptyset$.

Etapa 3: Subtraia 1 de k. Se $k = 0$, então $L(\mathcal{A})$ é denotada pela expressão regular E_1 no alfabeto Σ_0 e podemos parar; senão, substitua α_k por E_k na expressão regular E_i para $i \neq k$ e volte à Etapa 2.

Resta-nos apenas dar uma demonstração de que este algoritmo funciona. Faremos isso usando indução finita.

DEMONSTRAÇÃO. Digamos que, para um certo inteiro $0 \leq k \leq m$ estamos para executar o $(m-k)$-ésimo laço deste algoritmo. Queremos mostrar que, ao final deste laço

(1) $E_i \in \Theta_{k-1}$ e
(2) $L(E_i) = L_i$,

para $i = 1, \ldots, m$. Para fazer isto, podemos supor que, chegados a este laço, temos $E_i \in \Theta_k$ e $L(E_i) = L_i$ para $i = 1, \ldots, m$. Observe que estas duas últimas afirmações são claramente verdadeiras quando $k = m$.

Começamos considerando a execução da Etapa 2 no $(m-k)$-ésimo laço. Se já temos que $E_k \in \Theta_{k-1}$ então nada há a fazer nesta etapa. Por outro lado, se $E_k \notin \Theta_{k-1}$, então como $E_k \in \Theta_k$ podemos escrever

$$E_k = \eta_k \cdot \alpha_k \cup B, \tag{5.1}$$

onde $B \in \Theta_{k-1}$. Seja $F = \eta_k^* \cdot B$. Temos de (5.1) e (2) que

$$A_k = L(E_k) = L(\eta_k) \cdot A_k \cup L(B).$$

Logo, pelo lema de Arden,

$$A_k = L(\eta_k)^* \cdot L(B),$$

mas isto é igual a $L(F)$. Como o algoritmo manda fazer $E_k = F$, obtemos, ao final desta etapa, que $E_k \in \Theta_{k-1}$ e $L(E_k) = A_k$.

Finalmente, na etapa 3, basta substituir α_k por F na expressão regular de E_j sempre que $j \neq k$. Note que segue imediatamente disto que $E_j \in \Theta_{k-1}$ e que $L(E_j) = A_j$.

6. Exercícios

1. Seja $\Sigma = \{0, 1\}$. Se $L_1 = \{0\}$ e $L_2 = \{1\}^*$, determine:
   (1) $L_1 \cup L_2$ e $L_1 \cap L_2$;
   (2) L_1^* e L_2^*;
   (3) $(L_1 \cup L_2)^*$;
   (4) $L_1^* \cap L_2$;
   (5) $\overline{L_1}$;
   (6) $\overline{L_2} \cap L_1^*$.
2. Se L é uma linguagem em um alfabeto Σ então $L^R = \{w^R : w \in L\}$. Se $L = \{0\} \cdot \{1\}^*$ calcule L^R.
3. Seja L uma linguagem no alfabeto Σ. O que podemos concluir a respeito de L se $L^+ = L^* \setminus \{\epsilon\}$?
4. Sejam Σ_1 e Σ_2 dois alfabetos e seja $\phi : \Sigma_1 \to \Sigma_2^*$ uma aplicação. Se L é uma linguagem no alfabeto Σ_1, mostre que $\phi(L^*) = \phi(L)^*$.
5. Para cada um dos autômatos finitos determinísticos, no alfabeto $\{0, 1\}$, dados abaixo:
   - esboce o diagrama de estados;
   - encontre os sorvedouros e os estados mortos;
   - determine a expressão regular da linguagem aceita pelo autômato usando o algoritmo de substituição.

   (a) Os estado são $\{q_1, \ldots, q_4\}$, o estado inicial é q_1, o conjunto de estados finais é $\{q_2\}$ e a função de transição é dada por:

δ	0	1
q_1	q_2	q_4
q_2	q_3	q_1
q_3	q_4	q_4
q_4	q_4	q_4

   (b) Os estado são $\{q_1, \ldots, q_5\}$, o estado inicial é q_1, o conjunto de estados finais é $\{q_3, q_4\}$ e a função de transição é dada por:

δ	0	1
q_1	q_2	q_4
q_2	q_2	q_3
q_3	q_5	q_5
q_4	q_5	q_5
q_5	q_5	q_5

(c) Os estado são $\{q_1, \ldots, q_4\}$, o estado inicial é q_1, o conjunto de estados finais é $\{q_1\}$ e a função de transição é dada por:

δ	0	1
q_1	q_2	q_4
q_2	q_3	q_1
q_3	q_4	q_2
q_4	q_4	q_4

(d) Os estado são $\{q_1, q_2, q_3\}$, o estado inicial é q_1, o conjunto de estados finais é $\{q_1\}$ e a função de transição é dada por:

δ	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_1	q_2

(e) Os estado são $\{q_1, \ldots, q_6\}$, o estado inicial é q_1, o conjunto de estados finais é $\{q_4\}$ e a função de transição é dada por:

δ	0	1
q_1	q_5	q_2
q_2	q_5	q_3
q_3	q_4	q_3
q_4	q_4	q_4
q_5	q_6	q_2
q_6	q_6	q_4

6. Descreva em português o conjunto denotado por cada uma das expressões regulares abaixo:
   (a) 1^*0;
   (b) $1^*0(0)^*$
   (c) $111 \cup 001$;
   (d) $(1 \cup 00)^*$;
   (e) $(0(0)^*1)^*$;
   (f) $(0 \cup 1)(0 \cup 1)^*00$;

7. Expresse cada uma das seguintes linguagens no alfabeto $\{0, 1\}$ usando uma expressão regular:
   (a) o conjunto das palavras de um ou mais zeros seguidos de um 1;

(b) o conjunto das palavras de dois ou mais símbolos seguidos por três ou mais zeros;
(c) o conjunto das palavras que contêm uma seqüência de 1s, de modo que o número de 1s na seqüência é congruente a 2 módulo 3, seguido de um número par de zeros.

8. Se r e s são expressões regulares, vamos escrever $r \equiv s$ se e somente se $L(r) = L(s)$. Supondo que r, s e t são expressões regulares, mostre que:
   (a) $(r \cup r) \equiv r$;
   (b) $((r \cdot s) \cup (r \cdot t)) \equiv (r \cdot (s \cup t))$;
   (c) $((s \cdot r) \cup (t \cdot r)) \equiv ((s \cup t) \cdot r)$;
   (d) $(r^* \cdot r^*) \equiv r^*$;
   (e) $(r \cdot r^*) \equiv (r^* \cdot r)$;
   (f) $r^{**} \equiv r^*$;
   (g) $(\epsilon \cup (r \cdot r^*)) \equiv r^*$;
   (h) $((r \cdot s)^* \cdot r) \equiv (r \cdot (s \cdot r)^*)$;
   (i) $(r \cup s)^* \equiv (r^* \cdot s^*)^* \equiv (r^* \cup s^*)^*$.

9. Usando as identidades do exercício 4 prove que

$$((abb)^*(ba)^*(b \cup aa)) \equiv (abb)^*((\epsilon \cup (b(ab)^*a))b \cup (ba)^*(aa)).$$

Observe que alguns parênteses e o símbolo "$\cdot$" foram omitidos para facilitar a leitura.

CAPíTULO 4

Linguagens que não são regulares

O objetivo deste capítulo é desenvolver um método que nos permita mostrar que uma dada linguagem não é regular. Nossa estratégia será a seguinte. Em primeiro lugar, mostraremos que toda linguagem regular satisfaz certa propriedade, conhecida como *propriedade do bombeamento*. Assim, para provar que uma dada linguagem não é regular basta constatar que não satisfaz esta propriedade. Isto é, provaremos que uma linguagem não é regular através de uma demonstração por contradição.

1. Propriedade do bombeamento

Começamos por introduzir a terminologia básica e obter uma primeira aproximação para a propriedade de bombeamento das linguagens regulares.

Considere o autômato M da figura abaixo.

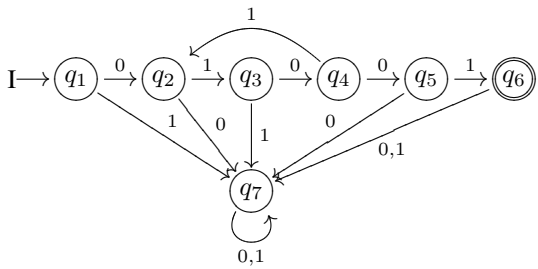


FIGURA 1

Entre as palavras aceitas por este autômato temos:

$$w = 01011001.$$

Se considerarmos o caminho indexado pela palavra $w = 01011001$, vemos que inclui um ciclo que começa em q_2 e acaba em q_4. Este ciclo corresponde à

subpalavra $y = 101$ de 01011001. Mais precisamente, podemos decompor w na forma

$$w = \underbrace{0}_{x}\underbrace{101}_{y}\underbrace{1001}_{z} = xyz.$$

Observe que podemos percorrer o ciclo indexado por y várias vezes e ainda assim obter uma palavra que é aceita por M. Por exemplo, percorrendo o ciclo 3 vezes obtemos

$$xy^3z = \underbrace{0}_{x}\underbrace{101}_{y}\underbrace{101}_{y}\underbrace{101}_{y}\underbrace{1001}_{z},$$

que é aceita por M. De fato, podemos até remover a subpalavra y de w e ainda assim continuaremos com uma palavra aceita pelo autômato, neste caso $xz = 01001$. Resumindo, verificamos que a palavra $w = xyz$ é aceita por M e que admite uma subpalavra $y \neq \epsilon$ que pode ser removida ou repetida várias vezes sem que a palavra resultante fique fora de $L(M)$. Sempre que isto acontecer diremos que y é uma subpalavra de w que é *bombeável* em $L(M)$.

Naturalmente, o ponto chave é o fato da subpalavra y indexar um ciclo no grafo de M. Na verdade, esta não é a única subpalavra bombeável de w uma vez que podemos considerar o ciclo como começando em qualquer um de seus vértices. Assim, se tomarmos o início do ciclo como sendo q_4, concluímos que a subpalavra 110 também é bombeável; isto é,

$$010(110)^k01 \in L(M) \text{ para todo } \ k \geq 0.$$

É conveniente estabelecer a noção de bombeabilidade em um contexto mais geral que o das linguagens regulares. Seja L uma linguagem em um alfabeto Σ e seja $w \in L$. Dizemos que $y \in \Sigma^*$ é uma *subpalavra de w bombeável em L* se

(1) $y \neq \epsilon$;
(2) existem $x, z \in \Sigma^*$ tais que $w = xyz$;
(3) $xy^kz \in L$ para todo $k \geq 0$.

A única função da condição (1) é excluir o caso trivial $y = \epsilon$ da definição de palavra bombeável; já (2) significa que y é subpalavra de w. Quanto a (3), o ponto crucial a observar é que, para que seja y seja bombeável é preciso que seja possível omiti-la ou repeti-la no interior de w tantas vezes quanto desejarmos *sem que a palavra resultante deixe de pertencer a L*. Para entender melhor este ponto, considere o autômato M' da figura 2.

É claro que 0 é uma subpalavra de $10 \in L(M')$. Além disso, podemos repetir a subpalavra 0 várias vezes e ainda assim obter uma palavra em $L(M')$; de fato, 1, 10, 10^2, 10^3 e 10^4 pertencem a $L(M')$. Apesar disto, 0 não é uma subpalavra de 10 bombeável em $L(M')$, porque se $k \geq 5$ então $10^k \notin L(M')$.

Permanecendo com o autômato M' da figura 2, vemos que

$$L(M') = \{1, 10, 10^2, 10^3, 10^4\}.$$

Em particular, não há nenhuma palavra de $L(M')$ que admita uma subpalavra bombeável. Isto não é surpreendente. De fato, se uma linguagem admite

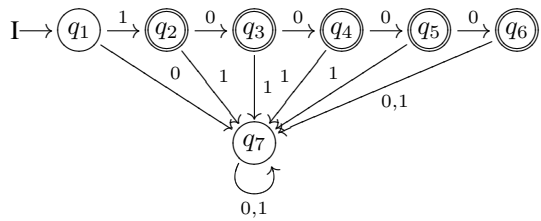


FIGURA 2

uma palavra que tem uma subpalavra bombeável, então é claro que a linguagem é infinita. Há dois pontos importantes nesta discussão que você não deve esquecer:

- nenhuma palavra de uma linguagem finita L admite subpalavra bombeável em L;
- para que uma subpalavra seja bombeável é preciso que possa ser repetida *qualquer número de vezes* sem que a palavra resultante saia de L.

2. Lema do bombeamento

Diante do que acabamos de ver, uma pergunta se impõe de maneira natural:

> dada uma linguagem L, como achar uma palavra de L que admita uma subpalavra bombeável?

Em primeiro lugar, esta pergunta só faz sentido se L for infinita. Além disso, vamos nos limitar, de agora em diante, às linguagens regulares. Sendo assim, vamos supor que $L \subset \Sigma^*$ é uma linguagem infinita que é aceita por um autômato finito determinístico M no alfabeto Σ.

Se conhecemos M o problema é fácil de resolver: basta achar um ciclo em M. Mas suponha que, apesar de conhecer L, sabemos de M apenas que tem n estados. Será que esta informação é suficiente para achar uma palavra de L que tenha uma subpalavra bombeável em L? A resposta é sim, e mais uma vez trata-se apenas de achar um ciclo em M. Só que, como não conhecemos M, não temos uma maneira de identificar qual é este ciclo. Mesmo assim somos capazes de saber que um tal ciclo *tem que existir*. Fazemos isto recorrendo a um princípio que aprendemos a respeitar ainda criança, quando brincamos de dança das cadeiras.

PRINCÍPIO DA CASA DO POMBO. *Se, em um pombal, há mais pombos que casas, então dois pombos vão ter que ocupar a mesmo casa.*

Nossa aplicação deste princípio depende de termos, de um lado uma linguagem **infinita**, de outro um autômato **finito** determinístico. De fato, como L é infinita, terá palavras de comprimento arbitrariamente grande. Em particular, podemos escolher uma palavra w cujo comprimento é muito maior que o

número n de estados de M. Considere o caminho indexado por w no grafo de M. Como M tem n estados e w tem muito mais do que n símbolos, este caminho tem que passar duas vezes por um mesmo estado. Mas um caminho no grafo de M no qual há estados repetidos tem que conter um ciclo. Entretanto, já sabemos que um ciclo no caminho indexado por w nos permite determinar uma subpalavra bombeável de w. Com isto provamos a seguinte *propriedade do bombeamento* das linguagens regulares:

> Seja M um autômato finito determinístico. Se w é uma palavra de $L(M)$ de comprimento maior que o número de estados do autômato, então w admite uma subpalavra bombeável em $L(M)$.

O lema do bombeamento, que é o principal resultado deste capítulo, não passa de uma versão refinada da propriedade do bombeamento enunciada acima.

LEMA DO BOMBEAMENTO. *Seja M um autômato finito determinístico com n estados e seja L a linguagem aceita por M. Se w é uma palavra de L com comprimento maior ou igual a n então existe uma decomposição de w na forma $w = xyz$, onde*

(1) $y \neq \epsilon$;
(2) $|xy| \leq n$;
(3) $xy^k z \in L$ *para todo* $k \geq 0$.

Antes de passar à demonstração, observe que (1) e (3) nos dizem apenas que y é subpalavra de w bombeável em L. A única novidade é a condição (2). Esta condição técnica permite simplificar várias demonstrações de não regularidade, reduzindo o número de casos que precisam ser considerados.

DEMONSTRAÇÃO. A estratégia adotada no início da seção consistiu em considerar o caminho no grafo de M indexado por w. Como observamos no capítulo 2, isto é formalizado através da computação de M determinada por w.

Seja Σ o alfabeto de M. Então podemos escrever $w = \sigma_1 \cdots \sigma_n$, onde $\sigma_1, \ldots, \sigma_n$ são elementos de Σ não necessariamente distintos. Seja q_1 o estado inicial de M. Temos, então, uma computação

$$(q_1, w) = (q_1, \sigma_1 \cdots \sigma_n) \vdash (q_2, \sigma_2 \cdots \sigma_n) \vdash \cdots \vdash (q_n, \sigma_n) \vdash (q_{n+1}, \epsilon).$$

Observe que também não estamos supondo que os estados $q_1, \ldots, q_{n+1}$ são todos distintos. De fato, dois destes estados têm que coincidir, porque M só tem n estados. Digamos que $q_i = q_j$, onde $1 \leq i < j \leq n+1$. Qualquer escolha de i e j que satisfaça as condições acima é suficiente para provar (1) e (3); mas não (2). Para garantir (2) precisamos escolher q_j como sendo o primeiro estado que coincide com algum estado anterior. Assumindo desde já que i e j são inteiros entre 1 e $n+1$, precisamos fazer a seguinte hipótese sobre j:

Hipótese: j é o menor inteiro para o qual existe $i < j$ tal que $q_i = q_j$.

Levando tudo isto em conta, podemos reescrever a computação na forma

$$(q_1, w) = (q_1, \sigma_1 \cdots \sigma_n) \vdash^* (q_i, \sigma_i \cdots \sigma_n) \vdash^* (q_j, \sigma_j \cdots \sigma_n) \vdash^* (q_n, \sigma_n) \vdash (q_{n+1}, \epsilon).$$

O ciclo que procuramos está identificado pelo trecho da computação que vai de q_i a $q_j = q_i$. Isto sugere que devemos tomar

$$x = \sigma_1 \cdots \sigma_{i-1}, \quad y = \sigma_i \cdots \sigma_{j-1} \quad \text{e} \quad z = \sigma_j \cdots \sigma_{n+1}.$$

Além disso, como $i < j$ temos que

$$y = \sigma_i \cdots \sigma_{j-1} \neq \epsilon,$$

de forma que a condição (1) é satisfeita. Usando esta notação, a computação fica

$$(q_1, w) = (q_1, xyz) \vdash^* (q_i, yz) \vdash^* (q_j, z) \vdash^* (q_{n+1}, \epsilon).$$

Note que, como $q_i = q_j$, a palavra y leva a computação do estado q_i ao estado q_i. Desta forma, repetindo ou omitindo y, podemos fazer este trecho repetir-se várias vezes no interior da computação sem alterar o estado em que computação termina, que continuará a ser q_{n+1}. Por exemplo, repetindo y uma vez temos a palavra xy^2z, que dá lugar à computação

$$(q_1, xy^2z) \vdash^* (q_i, y^2z) \vdash^* (q_j, yz) = (q_i, yz) \vdash^* (q_j, z) \vdash^* (q_{n+1}, \epsilon).$$

Como $xyz \in L(M)$ por hipótese, então q_{n+1} é um estado final de M. Portanto, $xy^2z \in L(M)$. De maneira semelhante $xy^kz \in L(M)$ para todo $k \geq 0$, o que prova (3).

Falta-nos apenas explicar porque (2) vale. Mas, $|xy| = j - 1$. Entretanto, q_j é o primeiro estado que coincide com algum estado anterior. Isto é, $q_1, \ldots, q_{j-1}$ são todos estados distintos. Como M tem n estados, isto significa que $j - 1 \leq n$. Portanto, $|xy| \leq n$, o que completa a demonstração.

Antes de passar às aplicações é preciso chamar a atenção para o fato de que a recíproca do lema do bombeamento é falsa. Isto é, o fato de uma linguagem L conter palavras que admitem subpalavras bombeáveis *não* garante que L seja regular. Portanto, não é possível provar regularidade usando o lema do bombeamento. Voltaremos a discutir este ponto no exemplo 5.

3. Aplicações do lema do bombeamento

O maior obstáculo à aplicação do lema do bombeamento está na interpretação correta do seu enunciado. Seja M um autômato finito determinístico com n estados. Segundo o lema do bombeamento, dada **qualquer** palavra $w \in L(M)$ de comprimento maior que n **existe** uma subpalavra $y \neq \epsilon$ que é bombeável em $L(M)$. Note que o lema não diz que qualquer subpalavra de w é bombeável, mas apenas que **existe** uma subpalavra de w que é bombeável.

Por exemplo, considere a linguagem L no alfabeto $\{0\}$ formada pelas palavras de comprimento par. É fácil construir um autômato finito com 2 estados que aceita L, portanto esta é uma linguagem regular e $n = 2$. Vamos escolher uma palavra de L de comprimento maior que 2; digamos, 0^6. Não é verdade

que qualquer subpalavra de 0^6 é bombeável em L. Por exemplo, 0 é uma subpalavra de 0^6, já que temos uma decomposição $0^6 = 0^2 \cdot 0 \cdot 0^3$; mas bombeando 0 obtemos

$$0^2 \cdot 0^k \cdot 0^3 = 0^{5+k},$$

que não pertence a L se k por par. De fato, para que a subpalavra seja bombeável em L é preciso que tenha comprimento par. Assim, neste exemplo, poderíamos escolher as subpalavras 0^2, 0^4 ou 0^6 para bombear.

Tudo isto pode parecer óbvio. O problema é que um nível adicional de dificuldade surge nas aplicações, porque desejamos usar o lema para provar que uma linguagem **não** é regular. Imagine que temos uma linguagem L e que, por alguma razão, desconfiamos que L não é regular. Para provar que L *de fato* não é regular podemos proceder por contradição.

Suponha, então, por contradição, que L seja aceita por algum autômato finito determinístico com n estados. De acordo com o lema do bombeamento qualquer palavra $w \in L$ de comprimento maior que n terá que admitir uma subpalavra bombeável. Assim, para obter uma contradição, basta achar **uma** palavra em L (o que é uma boa notícia!) que não tenha **nenhuma** subpalavra bombeável (o que é uma má notícia!).

Um último comentário antes de passar aos exemplos. Neste esboço de demonstração por contradição supusemos que L é aceita por um autômato finito determinístico com n estados. Entretanto, ao fazer esta hipótese não podemos especificar um valor numérico para n. De fato, se escolhermos $n = 100$, tudo o que teremos provado é que a linguagem não pode ser aceita por um autômato com 100 estados. Mas nada impediria, em princípio, que fosse aceita por um autômato com 101 estados. Resta-nos aplicar estas considerações gerais em alguns exemplos concretos.

Exemplo 1. Considere a linguagem no alfabeto $\{0\}$ definida por

$$L_{primos} = \{0^p : p \text{ é um primo positivo}\}.$$

A primeira coisa a observar é que esta linguagem é infinita. Isto é uma conseqüência de teorema provado pelo matemático grego Euclides por volta de 300 a. C., segundo o qual existem infinitos números primos.

Em seguida devemos considerar se seria possível construir um autômato finito que aceitasse esta linguagem. Para isto, seria necessário que o autômato pudesse determinar se um dado número p é primo ou não. Em outras palavras, o autômato teria que se certificar que p não é divisível pelos inteiros positivos menores que p. Como a quantidade de inteiros menores que p aumenta com p, isto requer uma memória infinita; que é exatamente o que um autômato finito não tem. Esta é uma boa indicação de que L_{primos} não é regular. Vamos comprovar nosso palpite usando o lema do bombeamento.

Suponha, então, por contradição, que L_{primos} é aceita por um autômato finito determinístico com n estados. Precisamos escolher uma palavra com comprimento maior que n em L_{primos}. Para fazer isto, basta escolher um primo $q > n$. A existência de um tal primo é conseqüência imediata do teorema

de Euclides mencionado acima. Portanto, 0^q é uma palavra de L_{primos} de comprimento maior que n.

Nestas circunstâncias, o lema do bombeamento garante que existe uma decomposição $0^q = xyz$ de modo que $y \neq \epsilon$ é bombeável em L_{primos}. Como o que desejamos é contradizer esta afirmação, temos que mostrar que 0^q não admite nenhuma subpalavra bombeável. Neste exemplo é fácil executar esta estratégia neste grau de generalidade. De fato, uma subpalavra não vazia qualquer de 0^q tem que ser da forma 0^j para algum $0 < j \leq q$. Mas x e z também são subpalavras de 0^q; de modo que também são cadeias de zeros. Tomando, $x = 0^i$, teremos que $z = 0^{q-i-j}$.

Bombeando y, concluímos que

$$xy^k z = 0^i(0^j)^k 0^{q-i-j} = 0^{i+jk+(q-i-j)} = 0^{q+(k-1)j}$$

deve pertencer a L_{primos} para todo $k \geq 0$. Mas isto só pode ocorrer se $q+(k-1)j$ for um número primo para todo $k \geq 0$. Entretanto, tomando $k = q + 1$, obtemos

$$q + (k-1)j = q + qj = q(1 + j)$$

que não pode ser primo porque tanto q quanto $j+1$ são números maiores que 1. Temos assim uma contradição, o que confirma nossas supeitas de que L_{primos} não é regular.

Note que a condição (2) do lema do bombeamento não foi usada em nenhum lugar nesta demonstração. Como frisamos anteriormente, esta é uma condição técnica que serve para simplificar o tratamento de exemplos mais complicados, como veremos a seguir.

Exemplo 2. Nosso próximo exemplo é a linguagem

$$L = \{a^m b^m : m \geq 0\}$$

no alfabeto $\{a, b\}$. Também neste caso é fácil dar um argumento heurístico que nos leva a desconfiar que L não pode ser regular. Lembre-se que o autômato lê a entrada da esquerda para a direita. Assim, ele lerá toda a seqüência de as antes de chegar aos bs. Portanto, o autômato tem que lembrar quantos as viu para poder comparar com o número de bs. Mas a memória do autômato é finita, e não há restrições sobre a quantidade de as em uma palavra de L.

Para provar que L não é regular vamos recorrer ao lema do bombeamento. Suponha, por contradição, que L é aceita por um autômato finito determinístico com n estados. Em seguida temos que escolher uma palavra w de L com comprimento maior que n; digamos que $w = a^n b^n$. Como $|w| = 2n > n$, tem que existir uma decomposição

$$a^n b^n = xyz$$

de forma que as condições (1), (2) e (3) do lema do bombeamento sejam satisfeitas.

Mas que decomposições de $a^n b^n$ satisfazem estas condições? Dessa vez começaremos analisando (2), segundo a qual $|xy| \leq n$. Isto é, xy é um prefixo

de $a^n b^n$ de comprimento menor ou igual a n. Como $a^n b^n$ começa com n letras a, concluímos que a é o único símbolo que x e y podem conter. Portanto,

$$x = a^i \quad \text{e} \quad y = a^j.$$

Além disso, $j \neq 0$ pela condição (1). Já z reúne o que sobrou da palavra w, de modo que

$$z = a^{n-i-j} b^n.$$

Observe que não há razão pela qual xy tenha que ser igual a a^n, de modo que podem sobrar alguns as em z.

Resta-nos bombear y. Fazendo isto temos que

$$xy^k z = a^i \cdot (a^j)^k \cdot a^{n-i-j} b^n = a^{n+(k-1)j} b^n,$$

é um elemento de L para todo $k \geq 0$. Contudo, $a^{n+(k-1)j} b^n$ só pode pertencer a L se os expoentes de a e b coincidirem. Porém

$$n + (k-1)j = n \quad \text{para todo} \quad k \geq 0$$

implica que $j = 0$, contradizendo a condição (1) do lema do bombeamento.

Antes de passar ao próximo exemplo convém considerar a escolha que fizemos para a palavra de comprimento maior que n. Não parece haver nada de extraordinário nesta escolha, mas a verdade é que nem toda escolha de w seria satisfatória. Por exemplo, assumindo que $n \geq 2$, teríamos que $|a^{n-1} b^{n-1}| = 2n - 2 \geq n$. Entretanto, esta não é uma boa escolha para w. A razão é que

$$a^{n-1} b^{n-1} = xyz \quad \text{e} \quad |xy| \leq n$$

não excluem a possibilidade de y conter um b. Isto nos obrigaria a considerar dois casos separadamente, a saber, $y = a^j$ e $y = a^j b$, o que complicaria um pouco a demonstração. Diante disto, podemos descrever o papel da condição (2) como sendo o de restringir os possíveis y. O problema é que isto não se dá automaticamente mas, como no exemplo acima, depende de uma escolha adequada para w.

Por sorte, na maioria dos casos, muitas escolhas para w são possíveis. Neste exemplo, bastaria tomar $w = a^r b^r$ com $r \geq n$. Entretanto, para algumas linguagens a escolha da palavra requer bastante cuidado, como mostra o próximo exemplo.

Exemplo 3. Um argumento heurístico semelhante ao usado para a linguagem anterior sugere que

$$L = \{a^m b^r : m \geq r\}$$

não deve ser regular. Vamos provar isto usando o lema do bombeamento.

Suponhamos, por contradição, que L seja aceita por um autômato finito determinístico com n estados. Neste exemplo, como no anterior, uma escolha possível para uma palavra de comprimento maior que n em L é $a^n b^n$. Da condição (2) do lema do bombeamento concluímos que, se $a^n b^n = xyz$, então

$$x = a^i \quad \text{e} \quad y = a^j.$$

Já condição (1) nos garante que $j \neq 0$. Como $z = a^{n-i-j}b^n$, obteremos, ao bombear y, que

$$xy^kz = a^i \cdot (a^j)^k \cdot a^{n-i-j}b^n = a^{n+(k-1)j}b^n.$$

Mas, para que esta palavra esteja em L é preciso que

$$n + (k-1)j \geq n,$$

donde segue que $(k-1)j \geq 0$. Por sua vez, $j \neq 0$ força que $k - 1 \geq 0$, ou seja, que $k \geq 1$. Mas, para que y seja bombeável é preciso que $xy^kz \in L$ para todo $k \geq 0$, e não apenas $k \geq 1$. Portanto, temos uma contradição com o lema do bombeamento, o que prova que L não é regular.

Desta vez estivemos perto de não chegar a lugar nenhum! De fato, uma contradição só é obtida porque tomando $k = 0$,

$$a^{n+(k-1)j}b^n = a^{n-j}b^n$$

não pertence a L. Entretanto, neste exemplo, muitas escolhas aparentemente adequadas de w não levariam a nenhuma contradição. Por exemplo, é fácil se deixar sugestionar pelo sinal $\geq$ e escolher $w = a^{n+1}b^n$. Esta palavra tem comprimento maior que n e qualquer decomposição da forma $a^{n+1}b^n = xyz$ requer que x e y só tenham as. Entretanto, tomando

$$x = a^i, \quad y = a^j \quad \text{e} \quad z = a^{n+1-i-j}b^n,$$

e bombeando y, obtemos

$$xy^kz = a^{n+1+(k-1)j}b^n$$

que pertence a L desde que $1 \geq (1-k)j$. Infelizmente, neste caso isto não leva a contradição nenhuma, a não ser que $j > 1$, e não temos como descartar a possibilidade de j ser exatamente 1.

A próxima linguagem requer uma escolha ainda mais sutil da palavra w.

Exemplo 4. Considere agora a linguagem

$$L_{uu} = \{uu : u \in \{0,1\}^*\}.$$

Como nos exemplos anteriores, é fácil descrever um argumento heurístico para justificar porque seria de esperar que L_{uu} não fosse regular, e deixaremos isto como exercício. Para provar a não regularidade de L_{uu} pelo lema do bombeamento, suporemos que esta linguagem é aceita por um autômato finito determinístico com n estados.

O principal problema neste caso é escolher uma palavra de comprimento maior que n que nos permita chegar facilmente a uma contradição. A escolha mais óbvia é $u = 0^n$, que, infelizmente, não leva a nenhuma contradição, como mostra o exercício 5. Felizmente uma variação simples desta palavra se mostra adequada, a saber $u = 0^n1$. Neste caso, $w = 0^n10^n1$ tem comprimento $2n + 2 > n$, e qualquer decomposição

$$0^n10^n1 = xyz$$

satisfaz

$$x = 0^i,\ y = 0^j \text{ e } z = 0^{n-i-j}10^n1$$

para algum $i \geq 0$ e $j \geq 1$. Bombeando y obtemos

$$xy^kz = 0^{n+(k-1)j}10^n1$$

Para saber se esta palavra pertence ou não a L_{uu} precisamos descobrir se pode ser escrita na forma vv para algum $v \in \{0,1\}^*$. Igualando

$$0^{n+(k-1)j}10^n1 = vv$$

concluímos que v tem que terminar em 1. Como só há um outro 1 na palavra,

$$v = 0^{n+(k-1)j}1 = 0^n1.$$

Isto é, $n + (k-1)j = n$. Como $j \neq 0$, esta igualdade só é verdadeira se $k = 1$. Mas isto contradiz o lema do bombeamento, segundo o qual xy^kz deveria pertencer a L_{uu} para todo $k \geq 0$.

Os exemplos anteriores mostram que a demonstração pelo lema do bombeamento de que uma certa linguagem L não é regular obedece a um padrão, que esboçamos abaixo:

> Suponhamos, por contradição, que L seja aceita por um autômato finito determinístico com n estados.
>
> - Escolha uma palavra $w \in L$, *de comprimento maior que* n, de modo que as possibilidades para uma decomposição da forma $w = xyz$ sejam bastante limitadas.
> - Bombeie y e mostre que se $xy^kz \in L$ então uma contradição é obtida.

As principais dificuldades em fazer funcionar esta estratégia são as seguintes:

- a escolha de uma palavra w adequada;
- a identificação correta da condição que a pertinência $xy^kz \in L$ impõe sobre os dados do problema.

No exercício 7 temos um exemplo de demonstração em que vários erros foram cometidos na aplicação desta estratégia. Resolver este exercício pode ajudá-lo a evitar os erros mais comuns que surgem na aplicação do lema do bombeamento.

Infelizmente o lema do bombeamento está longe de ser uma panacéia infalível. Para ilustrar isto, vamos considerar mais um exemplo.

Exemplo 5. Seja L a linguagem no alfabeto $\{a, b, c\}$ formada pelas palavras da forma $a^ib^jc^r$ para as quais $i, j, r \geq 0$ e, se $i = 1$ então $j = r$. Mostraremos que a única palavra de L que não admite uma subpalavra bombeável é ϵ.

Há dois casos a considerar. No primeiro, $i \neq 1$ e j e r não são ambos nulos. Neste caso a subpalavra bombeável é $y = b$ se $j \neq 0$ ou $y = c$ se $r \neq 0$. O segundo caso consiste em supor que $i = 1$, ou que $i \neq 1$ mas $j = r = 0$. Desta vez podemos tomar $y = a$ como sendo a palavra bombeável.

Como cada palavra de L se encaixa em um destes dois casos, provamos que toda palavra de L admite uma subpalavra bombeável. Entretanto, esta linguagem não é regular. Assim, constatamos neste exemplo que:

- a recíproca do lema do bombeamento é falsa; isto é, não basta que o resultado do lema do bombeamento seja verdadeiro para que a linguagem seja regular;
- nem sempre o lema do bombeamento basta para mostrar que uma linguagem não é regular.

Provaremos que a linguagem acima de fato não é regular no capítulo ???. Para isto, além do lema do bombeamento, vamos precisar usar um resultado sobre a estabilidade das linguagens regulares por intersecção.

4. Exercícios

1. Considere o autômato finito determinístico $\mathcal{M}$ no alfabeto $\{0, 1\}$, com estado inicial q_1, conjunto de estados finais $\{q_5, q_6, q_8\}$ e função de transição δ dada pela seguinte tabela:

δ	0	1
q_1	q_2	q_6
q_2	q_6	q_3
q_3	q_4	q_2
q_4	q_2	q_5
q_5	q_5	q_5
q_6	q_7	q_4
q_7	q_8	q_4
q_8	q_4	q_5

   (a) Esboce o diagrama de estados do autômato M.
   (b) Ache uma subpalavra de 010011101000 que possa ser bombeada na linguagem $L(M)$.
   (c) Seja $w = 00$. Verifique que $w, w^2 \in L(M)$. w é bombeável em $L(M)$?

2. Considere a linguagem L no alfabeto $\{0, 1\}$ dada por

$$L = \{10, 101^20, 101^201^30, 101^201^301^40, \dots\}.$$

   (a) Mostre que L é infinita mas não admite nenhuma palavra que tenha uma subpalavra bombeável.
   (b) Mostre que L não é regular.

3. Seja M um autômato finito determinístico e L a linguagem aceita por M. Vimos que para encontrar uma palavra de L que contém uma subpalavra bombeável basta encontrar um caminho no grafo de M que contenha um ciclo. Suponha agora que não conhecemos M, mas que conhecemos uma

expressão regular r que denota L. De que forma podemos usar r para achar uma palavra de L que tem uma subpalavra bombeável?

4. Ache uma palavra que contenha uma subpalavra bombeável na linguagem denotada pela expressão regular

$$(1 \cdot 1 \cdot 0) \cdot (((1 \cdot 0)^* \cdot 0) \cup 0).$$

5. Considere a linguagem

$$L_{uu} = \{uu : u \in \{0,1\}^*\}.$$

Mostre que, tomando $u = 0^n$, a palavra uu admite uma subpalavra bombeável em L_{uu}.
SUGESTÃO: Tome uma subpalavra de comprimento par.

6. Mostre que se L é uma linguagem regular infinita, então L admite pelos menos uma palavra que tem uma subpalavra bombeável.

7. Considere a linguagem

$$L = \{0^{2^n} : n \geq 0\}.$$

Determine os erros cometidos na demonstração abaixo de que L não é regular. Corrija estes erros e dê uma demonstração correta da não regularidade de L.

> Suponha que L é aceita por um autômato finito determinístico. Seja $w = 0^{2^n}$. Pelo lema do bombeamento podemos decompor w na forma $w = xyz$, onde
>
> $$x = 0^r, y = 0^s \quad \text{e} \quad z = 0^{2^n - r - s}.$$
>
> Bombeando y obtemos
>
> $$xy^k z = 0^{2^n + (k-1)s}.$$
>
> Mas para que esta palavra pertença a L é preciso que $2^n + (k-1)s = 2^n$, o que só é possível se $(k-1)s = 0$. Como $s \neq 0$, concluímos que k só pode ser igual a 1, o que contradiz o lema do bombeamento.

8. Verifique quais das linguagens dadas abaixo são regulares e quais não são. Em cada caso justifique cuidadosamente sua resposta.
   (a) $\{0^i 1^{2i} : i \geq 1\}$;
   (b) $\{(01)^i : i \geq 1\}$;
   (c) $\{1^{2n} : n \geq 1\}$;
   (d) $\{0^n 1^m 0^{n+m} : n, m \geq 1\}$;
   (e) $\{1^{2^n} : n \geq 0\}$;
   (f) $\{w : w = w^r \text{ onde } w \in \{0,1\}\}$;
   (g) $\{wxw^r : w, x \in \{0,1\}^* \setminus \{\epsilon\}\}$.

Se w é uma palavra em um alfabeto Σ então w^r é a palavra obtida invertendo-se a ordem das letras em w. Portanto se uma palavra satisfaz $w = w^r$ então é um palíndromo.

9. Uma palavra w no alfabeto $\{(,)\}$ é *balanceada* se:
   (a) em cada prefixo de w o número de (s não é menor que o número de)s e
   (b) o número de (s em w é igual ao número de)s.

   Isto é, w é balanceada se pode ser obtida a partir de uma expressão aritmética corretamente escrita pela omissão das variáveis, números e símbolos das operações. Mostre que a linguagem L que consiste nas palavras balanceadas no alfabeto $\{(,)\}$ não é regular.

10. Use o lema do bombeamento para mostrar que, se uma linguagem L contém uma palavra de comprimento maior ou igual a n e é aceita por um autômato finito determinístico com n estados, então L é infinita. Use isto para descrever um algoritmo que permite decidir se a linguagem aceita por um autômato finito determinístico dado é ou não infinita.

11. Seja M um autômato finito determinístico com n estados e seja L a linguagem aceita por M.
    (a) Use o lema do bombeamento para mostrar que se L contém uma palavra de comprimento maior ou igual que $2n$, então ela contém uma palavra de comprimento menor que $2n$.
    (b) Mostre que L é infinita se e somente se admite uma palavra de comprimento maior ou igual a n e menor que $2n$.
    (c) Descreva um algoritmo baseado em (3) que, tendo como entrada um autômato finito determinístico M, determina se $L(M)$ é finita ou infinita.

    SUGESTÃO: Para provar (a) use o lema do bombeamento com $k = 0$.

12. Seja $\mathcal{M}$ um autômato finito determinístico com n estados e um alfabeto de m símbolos.
    (a) Use (a) para mostrar que $L(\mathcal{M})$ é não vazia se e somente se contém uma palavra de comprimento menor ou igual a n.
    (b) Explique como isto pode ser usado para criar um algoritmo que verifica se a linguagem de um autômato finito determinístico é ou não vazia.
    (c) Suponha que a linguagem aceita por $\mathcal{M}$ é vazia. Quantas são as palavras que terão que ser testadas antes que o algoritmo de (b) possa chegar a esta conclusão? O que isto nos diz sobre a eficiência deste algoritmo?

CAPíTULO 5

Autômatos finitos não determinísticos

1. Desenhe o diagrama de estados e descreva a linguagem aceita por cada um dos seguintes autômatos finitos não determinísticos. Em cada caso o estado inicial é q_1.
   (a) $F_1 = \{q_4\}$ e a função de transição é dada por:

δ_1	a	b	c
q_1	$\{q_1, q_2, q_3\}$	$\emptyset$	$\emptyset$
q_2	$\emptyset$	$\{q_4\}$	$\emptyset$
q_3	$\emptyset$	$\emptyset$	$\{q_4\}$
q_4	$\emptyset$	$\emptyset$	$\emptyset$

   (b) $\delta_2 = \delta_1$ e $F_2 = \{q_1, q_2, q_3\}$.
   (c) $F_3 = \{q_2\}$ e a função de transição é dada por:

δ_3	a	b
q_1	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\{q_1, q_3\}$
q_3	$\{q_1, q_3\}$	$\emptyset$

2. Determine, usando o algoritmo descrito em aula, autômatos finitos não determinísticos que aceitem as linguagens cujas expressões regulares são dadas abaixo:
   (a) $(10 \cup 001 \cup 010)^*$;
   (b) $(1 \cup 0)^*00101$;
   (c) $((0 \cdot 0) \cup (0 \cdot 0 \cdot 0))^*$.

3. Converta cada um dos autômatos finitos não determinísticos obtidos no exercício anterior em um autômato finito determinístico.

4. Seja A um autômato finito determinístico com um único estado final. Considere o autômato finito não determinístico A' obtido invertendo os papéis

dos estados inicial e final e invertendo também a direção de cada seta no diagrama de estado. Descreva $L(A')$ em termos de $L(A)$.

5. Seja $\Sigma = \{0, 1\}$. Seja $L_1 \subset \Sigma^*$ a linguagem que consiste das palavras onde há pelo menos duas ocorrências de 0, e $L_2 \subset \Sigma^*$ a linguagem que consiste das palavras onde há pelo menos uma ocorrência de 1. Para cada uma das linguagens L abaixo dê uma expressão regular para L e use os algoritmos descritos em aula para criar um autômato finito não determinístico que aceita L.
   (a) $L = L_1 \cup L_2$;
   (b) $L = \Sigma^* \setminus L_1$;
   (c) $L = L_1 \cap L_2$.

6. Sejam L e L' linguagens regulares. Mostre que $L \setminus L'$ é regular

7. Sejam L e L' linguagens tais que L é regular, $L \cup L'$ é regular e $L \cap L' = \emptyset$. Mostre que L' é regular.

8. Sejam M e M' autômatos finitos não determinísticos em um alfabeto Σ. Neste exercício discutimos uma maneira de definir um autômato finito não determinístico N que aceita a concatenação $L(M) \cdot L(M')$ diferente da que foi vista em aula. Seja δ a função de transição de M. Para construir o grafo de N a partir dos grafos de M e M' procedemos da seguinte maneira. Toda vez que um estado q de M satisfaz

   > para algum $\sigma \in \Sigma$, o conjunto $\delta(q, \sigma)$ contém um estado final,

   acrescentamos uma transição de q para o estado inicial de M', indexada por σ.
   (a) Descreva detalhadamente todos os ingredientes de N. Quem são os estados finais de N?
   (b) Mostre que se $\epsilon \notin L(M)$ então N aceita $L(M) \cdot L(M')$.
   (c) Se $\epsilon \in L(M)$ então pode ser necessário acrescentar mais uma transição para que o autômato aceite $L(M) \cdot L(M')$. Que transição é esta?

9. Seja M um autômato finito determinístico no alfabeto Σ, com estado inicial q_1, conjunto de estados Q e função de transição δ. Dizemos que M tem *reinício* se existem $q \in Q$ e $\sigma \in \Sigma$ tais que

$$\delta(q, \sigma) = q_1.$$

   Em outras palavras, no grafo de M há uma seta que aponta para q_1. Mostre como é possível construir, a partir de um autômato finito determinístico M qualquer, um autômato finito determinístico M' *sem* reinício tal que $L(M) = L(M')$.

10. Sejam A e A' autômatos finitos determinísticos com alfabeto Σ, conjunto de estados Q e Q', estados iniciais q_1 e q_1', conjuntos de estados finais F e F' e funções de transição δ e δ'. Seja M o autômato finito determinístico com alfabeto Σ, conjunto de estados $Q_1 \times Q_2$, estado inicial (q_1, q_1'), conjunto

de estados finais $F_1 \times F_2$ e função de transição dada por

$$\delta((q, q'), \sigma) = (\delta_1(q, \sigma), \delta_2(q', \sigma)),$$

onde $q \in Q$, $q' \in Q'$ e $\sigma \in \Sigma$.

(a) Mostre que $L(M) = L(A) \cap L(A')$.

(b) Use (a) para dar uma outra demonstração de que a interseção de linguagens regulares é regular.

(c) Use esta construção para obter um autômato finito determinístico que aceita a linguagem L do exercício 6(c).

11. Seja L uma linguagem que é regular. Definimos o *posto* de L como sendo o *menor* inteiro positivo k para o qual existe um autômato finito determinístico A com k estados e tal que $L = L(A)$.

(a) Se L_1 e L_2 são linguagens regulares cujos postos são k_1 e k_2 respectivamente, determine m (em termos de k_1 e k_2) de modo que o posto de $L_1 \cdot L_2$ seja menor ou igual a m.

(b) Considere a afirmação: se $L_1 \subseteq L_2$ são linguagens regulares então o posto de L_1 tem que ser menor ou igual que o posto de L_2. Esta afirmação é verdadeira ou falsa? Justifique sua resposta com cuidado!

CAPíTULO 6

Operações com autômatos finitos

No capítulo 3 vimos como obter a expressão regular da linguagem aceita por um autômato finito. Agora, começamos a preparar o terreno para poder resolver o problema inverso; isto é, dada uma expressão regular r em um alfabeto Σ, como construir um autômato finito $\mathcal{M}$ que aceita $L(r)$? Nossa estratégia consistirá em usar r como uma receita recursiva para construir $\mathcal{M}$. Contudo, r é construída, a partir dos símbolos de Σ, por aplicações repetidas das operações de união, concatenação e estrela. Assim, para poder usar r como uma receita para construir $\mathcal{M}$ precisamos antes solucionar o seguinte problema:

> dados autômatos finitos $\mathcal{M}$ e $\mathcal{M}'$ em um alfabeto Σ constuir autômatos finitos que aceitem $L(\mathcal{M}) \cup L(\mathcal{M}')$, $L(\mathcal{M}) \cdot L(\mathcal{M}')$ e $L(\mathcal{M})^*$.

É exatamente isto que faremos neste capítulo.

1. União

Suponhamos que $\mathcal{M}$ e $\mathcal{M}'$ são autômatos finitos não determinísticos em um mesmo alfabeto Σ. Mais precisamente, digamos que

$$\mathcal{M} = (\Sigma, q_1, Q, F, \delta) \text{ e } \mathcal{M}' = (\Sigma, q_1', Q', F', \delta'). \tag{1.1}$$

Assumiremos, também, que $Q \cap Q' = \emptyset$. Observe que esta hipótese não representa nenhuma restrição expressiva, já que para alcançá-la basta renomear os estados de $\mathcal{M}'$, no caso de serem denotados pelo mesmo nome que os estados de $\mathcal{M}$.

Um outro detalhe que vale a pena mencionar é que, tanto nesta construção, como na da concatenação, estamos supondo que *ambos os autômatos estão definidos para um mesmo alfabeto*. Novamente, esta restrição é apenas aparente já que os autômatos não precisam ser determinísticos. De fato, podemos aumentar o alfabeto de entrada de um autômato não determinístico o quanto quisermos, sem alterar o seu comportamento. Para isso basta decretar que as

transições pelos novos estados são todas vazias. No caso da união, isto significa que, se $\mathcal{M}$ e $\mathcal{M}'$ tiverem alfabetos de entrada Σ e Σ' diferentes, então podemos considerar ambos como autômatos no alfabeto $\Sigma \cup \Sigma'$. Para mais detalhes veja o exercício 1.

Vejamos como deve ser o comportamento de um autômato finito $\mathcal{M}_u$ para que aceite $L(\mathcal{M}) \cup L(\mathcal{M}')$. O autômato $\mathcal{M}_u$ aceitará uma palavra $w \in \Sigma^*$ somente quando w for aceita por $\mathcal{M}$ *ou* por $\mathcal{M}'$. Mas, para descobrir isto, $\mathcal{M}_u$ deve ser capaz de simular estes dois autômatos. Como estamos partindo do princípio de que $\mathcal{M}_u$ é não determinístico, podemos deixá-lo escolher qual dos dois autômatos vai querer simular em uma dada computação. Portanto, ao receber uma palavra w, o autômato $\mathcal{M}_u$:

- escolhe se vai simular $\mathcal{M}$ ou $\mathcal{M}'$;
- executa a simulação escolhida e aceita w apenas se for aceita pelo autômato cuja simulação está sendo executada.

Uma maneira de realizar isto em um autômato finito é criar um novo estado inicial q_0 cuja única função é realizar a escolha de qual dos dois autômatos será simulado. Mais uma vez, como $\mathcal{M}_u$ não é determinístico, podemos deixá-lo decidir, por si próprio, qual o autômato que será simulado.

Ainda assim, resta um problema. De fato, todos os símbolos de w serão necessários para que as simulações de $\mathcal{M}$ e $\mathcal{M}'$ funcionem corretamente. Contudo, nossos autômatos precisam consumir símbolos para efetuar suas transições. Digamos, por exemplo, que ao receber w no estado q_0 o autômato $\mathcal{M}_u$ escolhe simular $\mathcal{M}$ com entrada w. Entretanto, para poder simular $\mathcal{M}$ em w, $\mathcal{M}_u$ precisa deixar q_0 e chegar ao estado inicial q_1 de $\mathcal{M}$ ainda tendo w por entrada, o que não é possível. Resolvemos este problema fazendo com que q_0 comporte-se, simultaneamente, como q_1 ou como q_1', dependendo de qual autômato $\mathcal{M}_u$ escolheu simular. Mais precisamente, se $\mathcal{M}_u$ escolheu simular $\mathcal{M}$, então q_0 realiza a transição a partir do primeiro símbolo de w *como se fosse uma transição de $\mathcal{M}$ a partir de q_1 por este mesmo símbolo*.

Para poder formalizar isto, digamos que $w = \sigma u$, onde $\sigma \in \Sigma$ e $u \in \Sigma^*$. Neste caso, se $\mathcal{M}_u$ escolheu simular $\mathcal{M}$, devemos ter que

$$(q_0, w) \vdash (p, u) \text{ se, e somente se, } p \in \delta(q_1, \sigma).$$

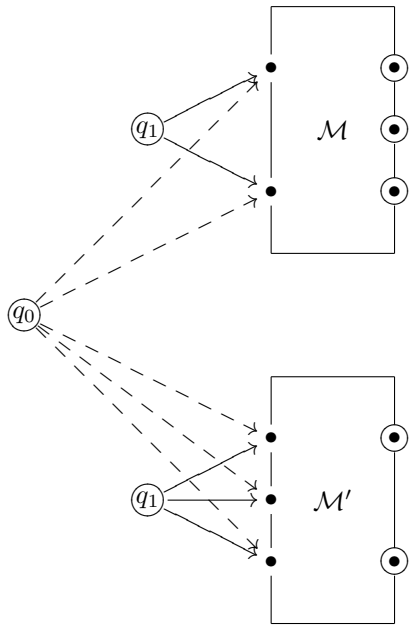
Podemos dissecar o comportamento de $\mathcal{M}_u$ a partir de q_0 como duas escolhas sucessivas:

(1) primeiro $\mathcal{M}_u$ decide se vai simular $\mathcal{M}$ ou $\mathcal{M}'$;
(2) a seguir, $\mathcal{M}_u$ escolhe qual a transição que vai ser executada por σ, a partir de q_1 ou q_1'–dependendo da escolha feita em (1).

Denotando por δ_u a função de transição de $\mathcal{M}_u$, podemos resumir isto escrevendo

$$\delta_u(q_0, \sigma) = \delta(q_1, \sigma) \cup \delta'(q_1', \sigma).$$

O comportamento geral de $\mathcal{M}_u$ pode ser ilustrado em uma figura, como segue. As transições que foram acrescentadas como parte da construção de $\mathcal{M}_u$ aparecem tracejadas.



Como a figura sugere, os estados de $\mathcal{M}_u$ são os mesmos de $\mathcal{M}$ e $\mathcal{M}'$, exceto por q_0. Quanto aos estados finais, precisamos ser mais cuidadosos. Em primeiro lugar, uma vez tendo passado por q_0, o autômato $\mathcal{M}_u$ meramente simula $\mathcal{M}$ ou $\mathcal{M}'$. Portanto, a descrição dos estados finais de $\mathcal{M}_u$ só apresenta algum problema se $q_1 \in F$ ou $q_1' \in F'$. Por exemplo, se q_1 for final, então $\mathcal{M}$ e, portanto, $\mathcal{M}_u$, deverá aceitar ϵ. Mas isto só pode ocorrer se q_0 for final. Portanto, q_0 deverá ser declarado como final sempre que q_1 ou q_1' forem finais.

Vamos formalizar a construção, listando os vários elementos de $\mathcal{M}_u$ um a um:

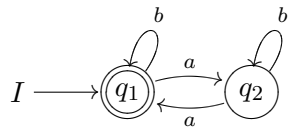
Alfabeto: Σ;
Estados: $\{q_0\} \cup Q \cup Q'$;
Estado inicial: q_0;
Estados finais:

$$\{q_0\} \cup F \cup F' \text{ se } q_1 \in F \text{ ou } q_1' \in F'$$
$$F \cup F' \text{ se } q_1 \notin F \text{ e } q_1' \notin F'$$

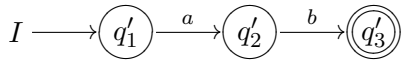
Função de transição:

$$\delta_u(q,\sigma) = \begin{cases} \delta(q_1,\sigma) \cup \delta'(q_1',\sigma) & \text{se } q = q_0 \\ \delta(q,\sigma) & \text{se } q \in Q \\ \delta'(q,\sigma) & \text{se } q \in Q' \end{cases}$$

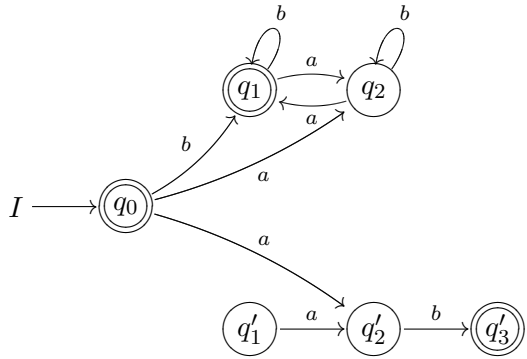
Vejamos como a construção se comporta em um exemplo. Considere o autômato finito determinístico $\mathcal{M}_{\rm par}$, com grafo



e o autômato finito não determinístico $\mathcal{M}_{ab}$ cujo grafo é



Já sabemos que $\mathcal{M}$ aceita a linguagem formada pelas palavras no alfabeto $\{a, b\}$ que têm uma quantidade par de as, e que $L(\mathcal{M}') = \{ab\}$. Aplicando a construção discutida acima para construir um autômato $\mathcal{M}_u$ que aceita $L(\mathcal{M}) \cup L(\mathcal{M}')$, obtemos



Há dois detalhes importantes a serem observados neste exemplo. O primeiro é que, como há uma seta de q_1 nele próprio, indexada por a, você poderia esperar encontrar em $\mathcal{M}_u$ uma seta de q_0 nele próprio com o mesmo índice. No entanto, não é isto que acontece. De fato, denotando por δ a função de transição de $\mathcal{M}$ temos, pela definição de $\mathcal{M}_u$, que

$$\delta(q_1, b) = q_1 \text{ implica } \ q_1 \in \delta_u(q_0, b).$$

Em outras palavras, a transição por b em $\mathcal{M}$, que leva q_1 nele próprio, dá lugar a uma transição por b de q_0 em q_1.

O segundo detalhe diz respeito à eliminação de estados redundantes. À primeira vista, como q_0 passa a desempenhar o papel de estado inicial, em substituição a q_1 e q_1', então estes últimos estados deveriam ter-se tornado redundantes. Entretanto, embora q_1' seja, de fato, redundante, o mesmo não ocorre com q_1. Por exemplo, o autômato $\mathcal{M}$ retorna a q_1

durante a computação

$$(q_1, a^2) \vdash (q_2, a) \vdash (q_1, \epsilon).$$

Simulando a mesma computação em $\mathcal{M}_u$, teremos

$$(q_0, a^2) \vdash (q_2, a) \vdash (q_1, \epsilon).$$

Note que embora $\mathcal{M}_u$ parta de q_0, na terceira etapa da computação ele alcança q_1, exatamente como $\mathcal{M}$. Assim, q_0 substitui q_1 *apenas quando este último está desempenhando o papel de estado inicial*.

Para esclarecer, de forma definitiva, todos os detalhes da construção de $\mathcal{M}_u$, provaremos formalmente que

$$L(\mathcal{M}_u) = L(\mathcal{M}) \cup L(\mathcal{M}').$$

Faremos isto no caso geral, admitindo que $\mathcal{M}$ e $\mathcal{M}'$ são os autômatos não determinísticos definidos em (1.1). Nossa demonstração consiste em provar que $w \in \Sigma^*$ é aceita por $\mathcal{M}_u$ se, e somente se, for aceita por $\mathcal{M}$ ou $\mathcal{M}'$. Se $w = \epsilon$ não há muito o que dizer, por isso deixamos este caso aos seus cuidados. De agora em diante estaremos assumindo que $w \neq \epsilon$. Com isso, podemos escrever w na forma

$$w = \sigma u, \text{ onde } \sigma \in \Sigma \text{ e } u \in \Sigma^*.$$

Antes de prosseguir, convém enunciar claramente as propriedades de $\mathcal{M}_u$ que serão utilizadas na demonstração. Em primeiro lugar, os estados e transições de $\mathcal{M}$ e de $\mathcal{M}'$ correspondem um a um, aos estados e transições de $\mathcal{M}_u$, exceto por q_0 e suas transições. Assim,

- toda computação em $\mathcal{M}$ pode ser reproduzida literalmente (quer dizer, com os mesmos estados e transições) como uma computação de $\mathcal{M}_u$;
- reciprocamente, qualquer computação de $\mathcal{M}_u$ a partir de um estado de $\mathcal{M}$ pode ser reproduzida literalmente em $\mathcal{M}$.

Naturalmente, as mesmas afirmações valem se substituímos $\mathcal{M}$ por $\mathcal{M}'$. Em segundo lugar,

$$\text{se } p \in \delta(q_1, \sigma) \text{ então } p \in \delta_u(q_0, \sigma). \tag{1.2}$$

Digamos, primeiramente, que $w \in L(\mathcal{M})$. Temos, então, uma computação

$$(q_1, \sigma u) \vdash (p, u) \vdash^* (f, \epsilon),$$

onde $p \in Q$ e $f \in F$. Mas, por (1.2),

$$\text{se } (q_1, \sigma u) \vdash (p, u) \text{ em } \mathcal{M}, \text{ então } (q_1, \sigma u) \vdash (p, u) \text{ em } \mathcal{M}_u.$$

Por outro lado, da primeira propriedade acima, a computação $(p, u) \vdash^* (f, \epsilon)$ em $\mathcal{M}$ pode ser reproduzida, passo a passo, e com os mesmos

estados e transições, como uma computação de $\mathcal{M}_u$. Obtemos, assim, a computação

$$(q_0, \sigma u) \vdash (p, u) \vdash^* (f, \epsilon),$$

em $\mathcal{M}_u$. Como f é final em $\mathcal{M}$, ele também será final em $\mathcal{M}_u$, de modo que toda palavra aceita por $\mathcal{M}$ será aceita por $\mathcal{M}_u$; isto é $L(\mathcal{M}) \subseteq L(\mathcal{M}_u)$. Um argumento análogo mostra que $L(\mathcal{M}') \subseteq L(\mathcal{M}_u)$.

Esta primeira parte do argumento pressupõe que todos os estados não redundantes de $\mathcal{M}$ e $\mathcal{M}'$ sejam também estados de $\mathcal{M}_u$. Mas há uma exceção a esta afirmação. Digamos, por exemplo, que q_1 seja inacessível a partir de qualquer estado de $\mathcal{M}$, *inclusive dele próprio*. Neste caso, q_1 vai se tornar redundante em $\mathcal{M}_u$. Isto acontece porque uma computação só pode passar por q_1 uma vez e, mesmo assim, somente se começar deste estado, ao passo que as computações de $\mathcal{M}_u$ começam de q_0. Este fenômeno ocorre, por exemplo, com o estado q_1' no exemplo descrito acima.

Suponha, agora, que

$$w = \sigma u \in L(\mathcal{M}_u).$$

Teremos, então, uma computação em $\mathcal{M}_u$ da forma

$$(q_0, \sigma u) \vdash (p, u) \vdash^* (f, \epsilon), \tag{1.3}$$

onde f é um estado final de $\mathcal{M}_u$. Pela definição das transições em $\mathcal{M}_u$ devemos ter que $p \in Q$ ou $p \in Q'$. Suporemos, sem perda de generalidade, que $p \in Q$. Contudo, qualquer transição em $\mathcal{M}_u$ a partir de um estado de $\mathcal{M}$ também é uma transição de $\mathcal{M}$. Como $p \in Q$, isto significa que

$$(p, u) \vdash^* (f, \epsilon)$$

terá que corresponder, passo a passo, a uma computação em $\mathcal{M}$. Portanto, usando (1.2), obtemos uma computação

$$(q_1, \sigma u) \vdash (p, u) \vdash^* (f, \epsilon),$$

em $\mathcal{M}$. Como fé um estado final de $\mathcal{M}_u$ que pertence a Q, concluímos que $f \in F$. Logo, w é aceita por $\mathcal{M}$.

Observe que a segunda parte da demonstração está ancorada no fato de nenhuma transição de $\mathcal{M}_u$ poder chegar em q_0. Se isto pudesse acontecer, $\mathcal{M}_u$ poderia executar parte da computação em $\mathcal{M}$, retornar a q_0, e continuar com uma computação de $\mathcal{M}'$. Entretanto, o argumento acima só funciona porque, uma vez que $\mathcal{M}_u$ tenha alcançado um estado de $\mathcal{M}$, ele fica *preso* neste último autômato, e assim é obrigado a se comportar como $\mathcal{M}$.

2. Concatenação

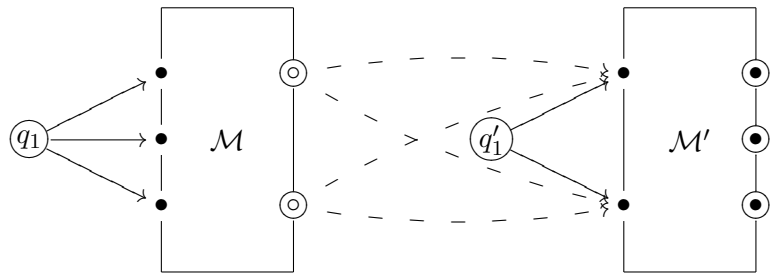
A segunda operação de linguagens que precisamos transcrever para os autômatos é a concatenação. Suponhamos, mais uma vez, que temos dois autômatos $\mathcal{M}$ e $\mathcal{M}'$ no mesmo alfabeto Σ, cujos elementos são

$$\mathcal{M} = (\Sigma, q_1, Q, F, \delta) \text{ e } \mathcal{M}' = (\Sigma, q_1', Q', F', \delta').$$

Continuaremos assumindo que $Q \cap Q' = \emptyset$.

Nosso objetivo é construir um autômato $\mathcal{M}_c$ que aceite a concatenação $L(\mathcal{M}) \cdot L(\mathcal{M}')$. Assim, dada uma palavra $w \in \Sigma^*$, o autômato $\mathcal{M}_c$ precisa verificar se existe um prefixo de w que é aceito por $\mathcal{M}$ e que é seguido de um sufixo aceito por $\mathcal{M}'$. Naturalmente isto sugere conectar os autômatos $\mathcal{M}$ e $\mathcal{M}'$ 'em série'; quer dizer, um depois do outro. Além disso, para que o prefixo esteja em $L(\mathcal{M})$ o último estado de $\mathcal{M}$ alcançado em uma computação por w deve ser final. Finalmente, é mais fácil construir um modelo não determinístico de $\mathcal{M}_c$, já que não temos como saber exatamente onde acaba o prefixo de w que pertence a $L(\mathcal{M})$.

Nossa experiência com a união deixa claro que não adianta conectar os autômatos através de transições de um estado final de $\mathcal{M}$ para o estado inicial de $\mathcal{M}'$. Isto só seria possível se nossos autômatos pudessem efetuar transições sem consumir nenhum símbolo da entrada. Contornamos o problema, como fizemos no caso da união, construindo as transições diretamente dos estados finais de $\mathcal{M}$ para os sucessores do estado inicial de $\mathcal{M}'$. Podemos ilustrar a construção em uma figura, como segue.



As setas correspondentes às transições entre os autômatos $\mathcal{M}$ e $\mathcal{M}'$ foram tracejadas para dar maior clareza à figura. Observe também que os estados que seriam finais em $\mathcal{M}$ aparecem com a bolinha central vazada ($\circ$ em vez de $\bullet$). Fizemos isto porque nem sempre um estado final de $\mathcal{M}$ continuará final em $\mathcal{M}_c$. Na verdade, se todo estado em F for final em $\mathcal{M}_c$ então toda computação em $\mathcal{M}$ que acabar em F será

aceita por $\mathcal{M}_c$. Quer dizer, toda palavra em $L(\mathcal{M})$ também pertence a $L(\mathcal{M}_c) = L(\mathcal{M}) \cdot L(\mathcal{M}')$. Entretanto, isto só pode acontecer se $\epsilon \in L(\mathcal{M}')$; ou, o que é equivalente, se q_1' é estado final de $\mathcal{M}'$. Caso contrário, apenas os estados finais de $\mathcal{M}'$ serão finais em $\mathcal{M}$.

Formalizaremos a construção, listando os vários elementos de $\mathcal{M}_c$ um a um:

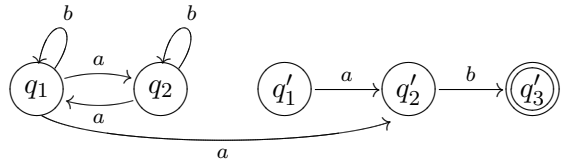
Alfabeto: Σ;
Estados: $Q \cup Q'$;
Estado inicial: q_1;
Estados finais:

$$\begin{aligned} F \cup F' &\text{ se } q_1' \in F' \\ F' &\text{ se } q_1' \notin F' \end{aligned}$$

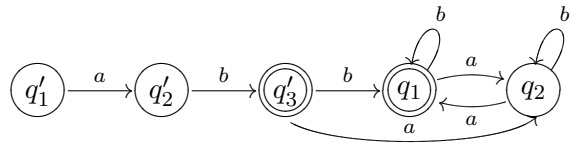
Função de transição:

$$\delta_c(q,\sigma) = \begin{cases} \delta(q,\sigma) & \text{se } q \in Q \setminus F \\ \delta(q,\sigma) \cup \delta'(q_1',\sigma) & \text{se } q \in F \\ \delta'(q,\sigma) & \text{se } q \in Q' \end{cases}$$

A demonstração de que esta construção funciona corretamente é bastante simples e segue a linha da demonstração para a união dada na seção anterior, por isso vamos omiti-la. Vejamos como a construção se comporta quando é aplicada aos autômatos utilizados no exemplo da seção 1. Começaremos concatenando $\mathcal{M}_{\text{par}}$ com $\mathcal{M}_{ab}$:



Observe que o estado q_1 deixou de ser final nesta concatenação, porque q_1' não é estado final de $\mathcal{M}_{ab}$. Por outro lado, se concatenarmos $\mathcal{M}_{ab}$ com $\mathcal{M}_{\text{par}}$, então q_3' continuará sendo final, já que q_1 é final em $\mathcal{M}_{\text{par}}$. O grafo do autômato resultante desta concatenação é o seguinte:

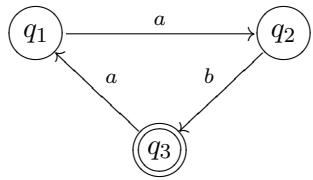


Neste caso temos duas transições a partir do estado final de $\mathcal{M}_{ab}$, porque q_1 tem como sucessores q_2 e o próprio q_1.

3. Estrela

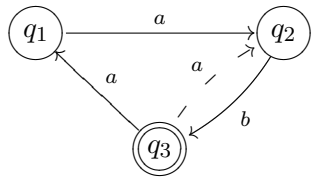
Tendo mostrado como construir um autômato para a concatenação, não parece tão difícil lidar com a estrela. Afinal, se L é uma linguagem, então L^* é obtida concatenando L com ela própria mais e mais vezes; isto é, calculando a união das L^j, para todo $j \geq 0$. Portanto, uma maneira de realizar um autômato $\mathcal{M}$, que aceite L, consistiria em conectar a saída de $\mathcal{M}$ com sua entrada. Com isso a concatenação passaria infinitas vezes pelo próprio $\mathcal{M}$, e teríamos um novo autômato $\mathcal{M}^*$ que aceita L^*. Vamos testar esta idéia em um exemplo e ver o que acontece.

Considere o autômato finito $\mathcal{N}$ no alfabeto $\{a, b\}$, cujo grafo é dado por



A linguagem aceita por $\mathcal{N}$ é denotada pela expressão regular $(aba)^*ab$, como é fácil de ver.

Para concatenar $\mathcal{N}$ com ele próprio precisamos criar transições entre seus estados finais e os sucessores do estado inicial. Neste exemplo, há apenas o estado final q_3, e q_1 tem apenas um sucessor; a saber, o estado q_2 que é acessível a partir de q_1 por a. Por isso precisamos criar uma nova transição de q_3 para q_2 por a, obtendo o seguinte autômato

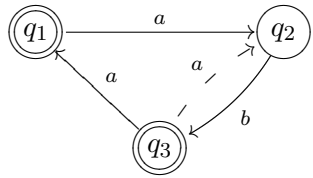


A transição acrescentada foi desenhada tracejada para que possa ser mais facilmente identificada.

Infelizmente, não pode ser verdade que este autômato aceita $L(\mathcal{N})^*$. Afinal, ϵ pertence a $L(\mathcal{N})^*$ e não é aceito pelo autômato que acabamos de construir. A essa altura, sua reação pode ser:

> "Não seja por isso! Para resolver este problema basta marcar q_1 como sendo estado final do novo autômato."

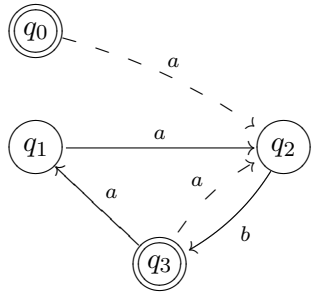
Vamos fazer isto e ver o que acontece. O grafo do autômato passaria a ser o seguinte:



Contudo, este autômato aceita a palavra aba que não pertence a

$$L(\mathcal{N})^* = ((aba)^*ab)^*.$$

O problema não está em nossa idéia original de concatenar um autômato com ele próprio, mas sim na emenda que adotamos para resolver o problema de fazer o novo autômato aceitar ϵ. Ao declarar q_1 como sendo estado final, não levamos em conta que o autômato admite transições que retornam ao estado q_1. Isto fez com que outras palavras, além de ϵ, passassem a ser aceitas pelo novo autômato. Mas não é isso que queremos. A construção original funcionava perfeitamente, exceto porque ϵ não era aceita. Uma maneira simples de contornar o problema é inspirar-se na união, e criar um novo estado q_0 para fazer o papel de estado inicial. Este estado vai comportar-se como q_1, e também será final. Entretanto, como no caso da união, as transições por q_0 são exatamente as mesmas que as transições por q_1. Em particular, nenhuma transição do novo autômato aponta para q_0. Isto garante que a introdução de q_0 leva à aceitação de apenas uma nova palavra, que é ϵ. Esboçamos abaixo o grafo do autômato resultante desta construção, no exemplo que vimos considerando.

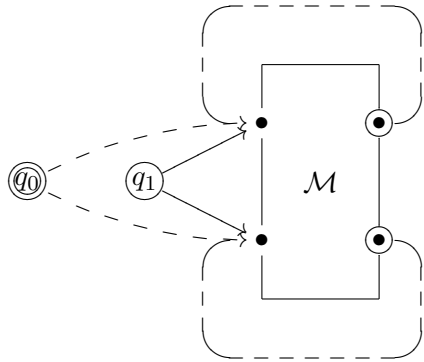


Em geral, quando é dado um autômato

$$\mathcal{M} = (\Sigma, q_1, Q, F, \delta)$$

podemos construir um autômato $\mathcal{M}^*$ que aceita $L(\mathcal{M}^*)$ segundo o modelo estabelecido no exemplo anterior. Como no caso da união e da concatenação, podemos ilustrar o autômato $\mathcal{M}^*$ em uma figura nas quais as

novas transições aparecem tracejadas.



Formalizaremos a construção, listando os vários elementos de $\mathcal{M}^*$, como segue:

Alfabeto: Σ;
Estados: $\{q_0\} \cup Q$;
Estado inicial: q_0;
Estados finais: $F \cup \{q_0\}$
Função de transição:

$$\delta^*(q,\sigma) = \begin{cases} \delta(q_1,\sigma) & \text{se } \ q = q_0 \\ \delta(q,\sigma) & \text{se } \ q \in Q \setminus F \\ \delta(q,\sigma) \cup \delta(q_1,\sigma) & \text{se } \ q \in F \end{cases}$$

Encerramos a seção com a demonstração de que esta construção funciona corretamente. Em primeiro lugar, como ϵ é aceita por construção, podemos limitar nossa discussão a uma palavra $w \neq \epsilon$ que pertença a Σ^*.

Como em nossa discussão do autômato que aceita a união de linguagens, isolaremos duas propriedades da definição de $\mathcal{M}^*$ que serão utilizadas na demonstração. Em primeiro lugar, se $(q_1,\sigma) \vdash (p,\epsilon)$ em $\mathcal{M}$, então existirão transições em $\mathcal{M}^*$ da forma

$$(q_0,\sigma) \vdash (p,\epsilon),$$

e, também, da forma

$$(f,\sigma) \vdash (p,\epsilon) \text{ para cada estado final } \ f \text{ de } \ \mathcal{M}.$$

Que q_0 comporta-se como se fosse q_1 é óbvio da maneira como suas transições foram definidas. Já a segunda parte da afirmação segue do fato de que também fizemos cada estado final de $\mathcal{M}^*$ comportar-se como se fosse q_1 ao concatenar $\mathcal{M}$ com ele próprio. Outro ponto importante é que toda computação em $\mathcal{M}$ pode ser simulada por $\mathcal{M}^*$ com os mesmos

estados e transições, já que $\mathcal{M}$ foi inteiramente preservado "dentro" de $\mathcal{M}^*$.

Seja $L = L(\mathcal{M})$. Como sempre, temos que provar que toda palavra que pertence a L^* é aceita por $\mathcal{M}^*$, e vice-versa. Começaremos mostrando $L^* \subseteq L(\mathcal{M}^*)$. Digamos que $w \in L^*$. Neste caso, podemos escrever w na forma

$$w = w_1 w_2 \cdots w_k \text{ onde } w_1, w_2, \ldots, w_k \in L.$$

Digamos que $w_j = \sigma_j u_j$, com $\sigma_j \in \Sigma$ e $u_j \in \Sigma^*$. Como cada w_j pertence a L, tem que existir uma computação

$$(q_1, w_j) \vdash (p_j, u_j) \vdash^* (f_j, \epsilon),$$

onde $p_j \in Q$ e $f_j \in F$. Mas o estado inicial q_0 de $\mathcal{M}^*$ simula o comportamento de q_1, e $\mathcal{M}^*$ é capaz de simular qualquer computação em $\mathcal{M}$. Portanto, podemos construir uma computação em $\mathcal{M}^*$ da forma:

$$(q_0, \sigma_1 u_1 w_2 \cdots w_k) \vdash (p_1, u_1 w_2 \cdots w_k) \vdash^* (f_1, w_2 \cdots w_k).$$

Por outro lado, também podemos construir computações da forma

$$\begin{aligned}(f_j, w_{j+1} \cdots w_k) = (f_j, \sigma_{j+1} u_{j+1} \cdots w_k) &\vdash (p_{j+1}, u_{j+1} \cdots w_k) \\ &\vdash^* (f_{j+1}, w_{j+2} \cdots w_k),\end{aligned}$$

para todo $1 \leq j < k$. Emendando o início de cada uma dessas computações ao final da outra, obtemos

$$\begin{aligned}(q_0, w_1 w_2 \cdots w_k) \vdash^* (f_1, w_2 \cdots w_k) \vdash^* (f_2, w_3 \cdots w_k) \cdots \vdash^* (f_{k-1}, w_k) \\ \vdash^* (f_k, \epsilon),\end{aligned}$$

provando, assim, que $\mathcal{M}^*$ aceita w.

Passamos, agora, a mostrar que $L(\mathcal{M}^*) \subseteq L^*$. Para facilitar a discussão, distinguiremos dois tipos de transição em $\mathcal{M}^*$:

Primeiro tipo: as transições a partir de q_0, além de todas aquelas que já eram transições de $\mathcal{M}$;

Segundo tipo: transições que $\mathcal{M}^*$ faz a partir dos estados finais de $\mathcal{M}$, como se fossem q_1.

Suponha, agora, que w é aceita por $\mathcal{M}^*$. Neste caso existe uma computação

$$(q_0, w) \vdash *(f, \epsilon),$$

em $\mathcal{M}^*$. Percorrendo, agora, esta computação, vamos dividi-la em partes de modo que a primeira transição de cada segmento, a partir do segundo, seja uma transição do segundo tipo. Se w_j for a subpalavra de

w consumida ao longo do j-ésimo segmento, podemos escrever os segmentos na forma

$$\begin{aligned}(q_0, w) &\vdash^* (f_1, w_2 \cdots w_k)\\ (f_1, w_2 \cdots w_k) &\vdash^* (f_2, w_3 \cdots w_k)\\ &\vdots\\ (f_{k-1}, w_k) &\vdash^* (f_k, \epsilon),\end{aligned}$$

onde $f_1, \ldots, f_k \in F$. Além disso, se $w_j = \sigma_j u_j$, com $\sigma_j \in \Sigma$ e $u_j \in \Sigma^*$, então a transição do segundo tipo pode ser isolada em cada segmento

$$\begin{aligned}(f_j, w_j w_{j+1} \cdots w_k) = (f_j, \sigma_j u_j w_{j+1} \cdots w_k) &\vdash (p_j, u_j w_{j+1} \cdots w_k)\\ &\vdash^* (f_{j+1}, w_{j+1} \cdots w_k),\end{aligned}$$

para todo $1 \leq j < k$, onde $p_j \in Q$. Temos, assim, computações da forma

$$(f_j, w_j) = (f_j, \sigma_j u_j) \vdash (p_j, u_j) \vdash^* (f_{j+1}, \epsilon),$$

em $\mathcal{M}^*$, que por sua vez dão lugar a computações da forma

$$(q_1, w_j) = (f_j, \sigma_j u_j) \vdash (p_j, u_j) \vdash^* (f_{j+1}, \epsilon),$$

em $\mathcal{M}$. Portanto, $w_2, \ldots, w_k \in L$. Um argumento semelhante mostra que $w_1 \in L$, e nos permite concluir que

$$w = w_1 w_2 \cdots w_k \in L^*,$$

como nos faltava mostrar.

4. Exercícios

1. Sejam $\Sigma \subset \Sigma'$ dois alfabetos. Suponha que $\mathcal{M}$ é um autômato finito não determinístico no alfabeto Σ. Construa formalmente um autômato finito não determinístico $\mathcal{M}'$ no alfabeto Σ', de modo que $L(\mathcal{M}) = L(\mathcal{M}')$. Prove que sua construção funciona corretamente.

   SUGESTÃO: Defina todas as transições de $\mathcal{M}'$ por símbolos de $\Sigma' \setminus \Sigma$ como sendo vazias.

2. Seja $\Sigma = \{0, 1\}$. Suponha que $L_1 \subset \Sigma^*$ é a linguagem que consiste das palavras onde há pelo menos duas ocorrências de 0, e que $L_2 \subset \Sigma^*$ é a linguagem que consiste das palavras onde há pelo menos uma ocorrência de 1.
   (a) Construa autômatos finitos não determinísticos que aceitem L_1 e L_2.
   (b) Use os algortimos desta seção para criar autômatos finitos não determinísticos que aceitem $L_1 \cup L_2$, $L_1 \cdot L_2$, L_1^* e L_2^*.

3. Sejam M e M' autômatos finitos não determinísticos em um alfabeto Σ. Neste exercício discutimos uma maneira de definir um autômato finito não determinístico N, que aceita a concatenação $L(M)\cdot L(M')$, e que é diferente do que foi vista na seção 2. Seja δ a função de transição de M. Para construir o grafo de N a partir dos grafos de M e M' procedemos da seguinte maneira. Toda vez que um estado q de M satisfaz

   > para algum $\sigma \in \Sigma$, o conjunto $\delta(q,\sigma)$ contém um estado final,

   acrescentamos uma transição de q para o estado inicial de M', indexada por σ.
   (a) Descreva detalhadamente todos os elementos de N. Quem são os estados finais de N?
   (b) Mostre que se $\epsilon \notin L(M)$ então N aceita $L(M)\cdot L(M')$.
   (c) Se $\epsilon \in L(M)$ então pode ser necessário acrescentar mais uma transição para que o autômato aceite $L(M)\cdot L(M')$. Que transição é esta?

4. Seja M um autômato finito determinístico no alfabeto Σ, com estado inicial q_1, conjunto de estados Q e função de transição δ. Dizemos que M tem *reinício* se existem $q \in Q$ e $\sigma \in \Sigma$ tais que
$$\delta(q,\sigma) = q_1.$$
   Em outras palavras, no grafo de M há uma seta que aponta para q_1. Mostre como é possível construir, a partir de um autômato finito determinístico M qualquer, um autômato finito determinístico M' *sem* reinício tal que $L(M) = L(M')$.

5. Sejam M e M' autômatos finitos *determinísticos* no alfabeto Σ cujos elementos são $(\Sigma, Q, q_1, F, \delta)$ e $(\Sigma, Q', q_1', F', \delta')$, respectivamente. Defina um novo autômato finito *determinístico* N construído a partir de M e M' como segue:
   Alfabeto: Σ;
   Estados: pares (q,q') onde $q \in Q$ e $q' \in Q'$;
   Estado inicial: (q_1, q_1');
   Estados finais: pares (q,q') onde $q \in F$ ou $q' \in F'$;
   Transição: a função de transição é definida em um estado (q,q') de N e símbolo $\sigma \in \Sigma$ por
$$\hat{\delta}((q,q'),\sigma) = (p,p'),$$
   onde $p = \delta(q,\sigma)$ e $p' = \delta'(q',\sigma)$.
   Calcule $L(N)$ em função de $L(M)$ e $L(M')$ e prove que a sua resposta está correta.

CAPíTULO 7

Autômatos de expressões regulares

No capítulo ?? vimos como obter a expressão regular da linguagem aceita por um autômato finito. Neste capítulo, resolvemos o problema inverso; isto é, dada uma expressão regular r em um alfabeto Σ, construímos um autômato finito que aceita $L(r)$. Na verdade, o autômato que resultará de nossa construção não será determinístico. Mas isto não é um problema, já que sabemos como convertê-lo em um autômato determinístico usando a construção de subconjuntos.

1. Considerações gerais

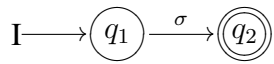
Seja r uma expressão regular r no alfabeto Σ. Nossa estratégia consistirá em utilizar a expressão regular r como uma receita para a construção de um autômato finito não determinístico $\mathcal{M}(r)$ que aceita $L(r)$. Entretanto, como vimos no capítulo ??, r é definida, de maneira recursiva, a partir dos símbolos de Σ, ϵ e $\emptyset$, por aplicação sucessiva das operações de união, concatenação e estrela. Por isso, efetuaremos a construção de $\mathcal{M}(r)$ passo à passo, a partir dos autômatos que aceitam os símbolos de Σ, ϵ e $\emptyset$.

Contudo, para que isto seja possível, precisamos antes de resolver alguns problemas. O primeiro, e mais simples, consiste em construir autômatos finitos que aceitem um símbolos de Σ, ou ϵ ou $\emptyset$. Estes autômatos funcionarão como os átomos da construção. Os problemas mais interessantes dizem respeito à construção de novos autômatos a partir de autômatos já existentes. Suponhamos, então, que $\mathcal{M}_1$ e $\mathcal{M}_2$ são autômatos finitos não determinísticos. Precisamos saber construir

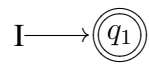
(1) um autômato $\mathcal{M}_u$ que aceita $L(\mathcal{M}_1) \cup L(\mathcal{M}_2)$;
(2) um autômato $\mathcal{M}_c$ que aceita $L(\mathcal{M}_1) \cdot L(\mathcal{M}_2)$; e
(3) um autômato $\mathcal{M}_1*$ que aceita $L(\mathcal{M}_1)^*$.

Se formos capazes de inventar maneiras de construir todos estes autômatos, então seremos capazes de reproduzir a montagem de r passo à passo no domínio dos autômatos finitos, o que nos levará a um autômato finito que aceita $L(r)$, como desejamos.

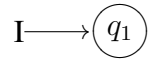
Começaremos descrevendo os átomos desta construção; isto é, os autômatos que aceitam símbolos de Σ, ϵ e $\emptyset$. Seja $\sigma \in \Sigma$. Um autômato simples que aceita σ é dado pelo grafo



O autômato que aceita ϵ é ainda mais simples



Para obter o que aceita $\emptyset$ basta não declarar nenhum estado como sendo final. A maneira mais simples de fazer isto é alterar o autômato acima, como segue.



Construídos os átomos, falta-nos determinar como podem ser conectados para obter estruturas maiores e mais complicadas. Passamos, assim, à solução dos problemas enunciados acima.

2. União

Suponhamos que $\mathcal{M}$ e $\mathcal{M}'$ são autômatos finitos não determinísticos em um mesmo alfabeto Σ. Mais precisamente, digamos que

$$\mathcal{M} = (\Sigma, q_1, Q, F, \delta) \text{ e } \mathcal{M}' = (\Sigma, q_1', Q', F', \delta').$$

Suporemos, também, que $Q \cap Q' = \emptyset$. Isto não representa nenhuma restrição expressiva. Significa, no máximo, que pode ser necessário renomear os estados de $\mathcal{M}'$ caso sejam denotados pelo mesmo nome que os estados de $\mathcal{M}$.

Vejamos como deve ser o comportamento de um autômato finito $\mathcal{M}_u$ para que aceite $L(\mathcal{M}) \cup L(\mathcal{M}')$. Dada uma palavra $w \in \Sigma*$ a $\mathcal{M}_u$ ele deve aceitá-la apenas se w for aceita por $\mathcal{M}$ ou por $\mathcal{M}'$. Mas, para descobrir isto, $\mathcal{M}_u$ deve ser capaz de simular estes dois autômatos. Como estamos partindo do princípio que $\mathcal{M}_u$ é não determinístico, podemos deixá-lo escolher qual dos dois autômatos ele vai querer simular

em uma dada computação. Portanto, dada uma palavra w a $\mathcal{M}_u$, ele executa a seguinte estratégia:

- escolhe se vai simular $\mathcal{M}$ ou $\mathcal{M}'$;
- executa a simulaç ao escolhida e aceita w apenas se for aceita pelo autômato cuja simulação está sendo executada.

Uma maneira de realizar isto em um autômato finito é criar um novo estado inicial q_0 cuja única função é realizar a escolha de qual dos dois autômatos será simulado. Mais uma vez, como $\mathcal{M}_u$ não é determinístico, podemos deixá-lo decidir, por si próprio, qual o autômato que será simulado.

Ainda assim, resta um problema. De fato, todos os símbolos de w serão necessários para que as simulações de $\mathcal{M}$ e $\mathcal{M}'$ funcionem corretamente. Contudo, nossos autômato precisam consumir símbolos para se mover. Digamos, por exemplo, que ao receber w no estado q_0 o autômato $\mathcal{M}_u$ escolhe simular $\mathcal{M}$ com entrada w. Entretanto, para poder simular $\mathcal{M}$ em w, $\mathcal{M}_u$ precisa deixar q_0 e chegar ao estado inicial q_1 de $\mathcal{M}$ ainda tendo w por entrada, o que não é possível. Resolvemos este problema fazendo com que q_0 comporte-se, simultaneamente, como q_1 ou como q_1', dependendo de qual autômato $\mathcal{M}_u$ escolheu simular. Mais precisamente, se $\mathcal{M}_u$ escolheu simular $\mathcal{M}$, então q_0 realiza a transição a partir do primeiro símbolo de w *como se fosse uma transição de* $\mathcal{M}$ *a partir de* q_1 *por este mesmo símbolo*.

Para poder formular isto formalmente, digamos que $w = \sigma u$, onde $\sigma \in \Sigma$ e $u \in \Sigma^*$. Neste caso, se $\mathcal{M}_u$ escolheu simular $\mathcal{M}$, devemos ter que

$$(q_0, w) \vdash (p, u) \text{ se, e somente se, } p \in \delta(q_1, \sigma).$$

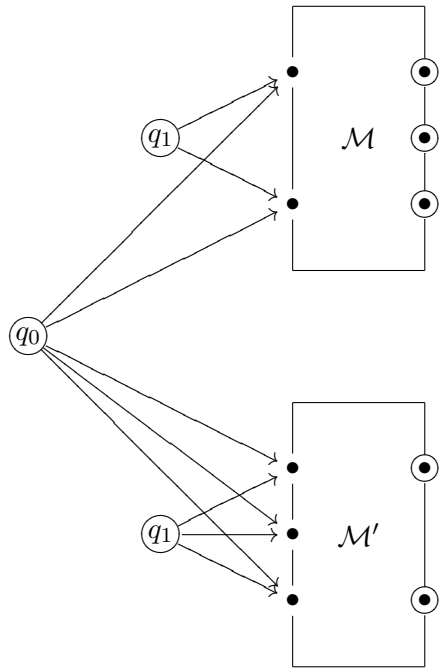
Podemos dissecar o comportamento de $\mathcal{M}_u$ a partir de q_0 como duas escolhas sucessivas:

(1) primeiro $\mathcal{M}_u$ decide se vai simular $\mathcal{M}$ ou $\mathcal{M}'$;
(2) a seguir, $\mathcal{M}_u$ escolhe qual a transição que vai ser executada por σ, a partir de q_1 ou q_1'–dependendo da escolha feita em (1).

Denotando por δ_u a função de transição de $\mathcal{M}_u$, podemos resumir isto escrevendo

$$\delta_u(q_0, \sigma) = \delta(q_1, \sigma) \cup \delta'(q_1', \sigma).$$

O comportamento geral de $\mathcal{M}_u$ pode ser ilustrado em uma figura, como segue.



Como a figura sugere, a parte q_0, os estados de $\mathcal{M}_u$ são os mesmos de $\mathcal{M}$ e $\mathcal{M}'$. Quanto aos estados finais, precisamos ser mais cuidadosos. Como, uma vez tendo passado por q_0, o autômato $\mathcal{M}_u$ meramente simula $\mathcal{M}$ ou $\mathcal{M}'$, só pode haver algum problema se um dos estados q_1 ou q_1' for final. Por exemplo, se q_1 for final, então $\mathcal{M}$ e, portanto, $\mathcal{M}_u$, deverá aceitar ϵ. Mas isto só pode ocorrer se q_0 for final. Portanto, q_0 deverá ser declarado como final sempre que q_1 ou q_1' forem finais.

Vamos formalizar a construção de $\mathcal{M}_u$ antes de fazer um exemplo. Listando os vários elementos de $\mathcal{M}_u$ um a um temos

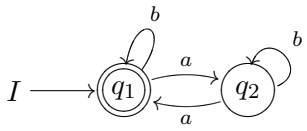
Alfabeto: Σ;
Estados: $\{q_0\} \cup Q \cup Q'$;
Estado inicial: q_0;
Estados finais:

$$\begin{aligned} \{q_0\} \cup F \cup F' &\text{ se } \ q_1 \in F \text{ ou } \ q_1' \in F' \\ F \cup F' &\text{ se } \ q_1 \notin F \text{ e } \ q_1' \notin F' \end{aligned}$$

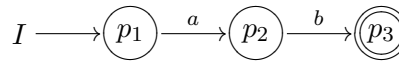
Função de transição:

$$\delta_u(q,\sigma) = \begin{cases} \delta(q_1,\sigma) \cup \delta'(q_1',\sigma) & \text{se } \ q = q_0 \\ \delta(q,\sigma) & \text{se } \ q \in Q \\ \delta'(q,\sigma) & \text{se } \ q \in Q' \end{cases}$$

Vejamos como a construção se comporta em um exemplo. Considere os autômatos finitos $\mathcal{M}$ e $\mathcal{M}'$ cujos grafos são, respectivamente



e



CAPíTULO 8

Gramáticas lineares à direita

Neste capítulo começamos o estudo das gramáticas formais, que serão introduzidas a partir da noção de autômato finito.

1. Exercícios

1. Determine gramáticas lineares à direita que gerem as linguagens denotadas pelas seguintes expressões regulares:
   (a) $(0^* \cdot 1) \cup 0$;
   (b) $(0^* \cdot 1) \cup (1^* \cdot 0)$;
   (c) $((0^* \cdot 1) \cup (1^* \cdot 0))^*$.
2. Ache a expressão regular que denota a linguagem regular gerada pela gramática com variáveis S, A, B, terminais a, b, símbolo inicial S e regras:
$$\begin{aligned} S &\to bA \quad | \quad aB \quad | \quad \epsilon \\ A &\to abaS \\ B &\to babS. \end{aligned}$$
3. Ache uma gramática linear à direita que gere a seguinte linguagem
$$\{w \in \{0,1\}^* : w \text{ não contém a seqüência } aa\}.$$
4. Ache gramáticas lineares à direita que gerem cada uma das linguagens regulares do exercício 2 da 2ª lista de exercícios.
5. Ache autômatos finitos que aceitem as linguagens geradas pelas gramáticas lineares à direita no alfabeto $\{0,1\}$ e símbolo inicial S, com as seguintes regras:
   (a) $S \to 011X$, $S \to 11S$, $X \to 101Y$, $Y \to 111$.
   (b) $S \to 0X$, $X \to 1101Y$, $X \to 11X$, $Y \to 11Y$, $X \to 1$.

6. Seja L uma linguagem regular no alfabeto Σ. Mostre que $L \setminus \{\epsilon\}$ pode ser gerada por uma gramática linear à direita cujas regras são dos tipos
   - $X \to \sigma Y$ ou
   - $X \to \sigma$,

   onde X, Y são variáveis e σ é um símbolo de Σ.

CAPíTULO 9

Linguagens livres de contexto

Neste capítulo começamos a estudar uma classe de gramáticas fundamental na descrição das linguagens de programação: as gramáticas livres de contexto. [1]

1. Gramáticas e linguagens livres de contexto

Já vimos no capítulo anterior que uma gramática é descrita pelos seguintes ingredientes: terminais, variáveis, símbolo inicial e regras. É claro que, em última análise, quando pensamos em uma gramática o que nos vem a cabeça são suas regras; os outros ingredientes são, de certo modo, circunstanciais. Assim, o que diferencia uma classe de gramáticas da outra é o tipo de regras que nos é permitido escrever. No caso de gramáticas lineares à direita, as regras são extremamente rígidas: uma variável só pode ser levada na concatenação de uma palavra nos terminais com alguma variável, além disso a variável tem que estar à direita dos terminais.

Já as linguagens livres de contexto, que estaremos estudando neste capítulo, admitem regras muito mais flexíveis. De fato a única restrição é que à esquerda da seta só pode aparecer uma variável. Talvez você esteja se perguntando: mas o que mais poderia aparecer à esquerda de

[1] Agradeço ao David Boechat pelas correções a uma versão anterior deste capítulo

uma seta? Voltaremos a esta questão na seção 2, depois de considerar vários exemplos de linguagens que *são* livres de contexto. Mas antes dos exemplos precisamos dar uma definição formal do que é uma gramática livre de contexto.

Seja $\mathcal{G}$ uma gramática com conjunto de terminais $\mathcal{T}$, conjunto de variáveis $\mathcal{V}$ e símbolo inicial $S \in \mathcal{V}$. Dizemos que $\mathcal{G}$ é *livre de contexto* se todas as suas regras são do tipo $X \to w$, onde $X \in \mathcal{V}$ e $w \in (\mathcal{T} \cup V)^*$.

Observe que as regras de uma gramática linear à direita são deste tipo. Portanto, toda gramática linear à direita é livre de contexto. Por outro lado, é evidente que nem toda gramática livre de contexto é linear à direita. Um exemplo simples é dado pela gramática $\mathcal{G}_1$ com terminais $T = \{0, 1\}$, variáveis $V = \{S\}$, símbolo inicial S e conjunto de regras

$$\{S \to 0S1,\ S \to \epsilon\}.$$

Como do lado esquerdo de cada seta só há uma variável, $\mathcal{G}_1$ é livre de contexto. Contudo, $\mathcal{G}_1$ não é linear à direita porque $S \to 0S1$ tem um terminal à direita de uma variável.

Outro exemplo simples é a gramática $\mathcal{G}_2$ com terminais $T = \{0, 1\}$, variáveis $V = \{S, X\}$, símbolo inicial S e conjunto de regras

$$\{S \to X1X, X \to 0X,\ X \to \epsilon\}.$$

Mais uma vez, apesar de ser claramente livre de contexto, esta gramática não é linear à direita, por causa da regra $S \to X1X$.

Os dois exemplos construídos acima estão relacionados a linguagens que são velhas conhecidas nossas. Entretanto, para constatar isto precisamos entender como é possível construir uma linguagem a partir de uma gramática livre de contexto. Isto se faz de maneira análoga ao que já ocorria com linguagens lineares à direita.

Seja $\mathcal{G}$ uma gramática livre de contexto com terminais $\mathcal{T}$, variáveis $\mathcal{V}$, símbolo inicial S e conjunto $\mathcal{R}$ de regras, e sejam $w, w' \in (\mathcal{T} \cup \mathcal{V})^*$. Dizemos que w' *pode ser derivada em um passo* a partir de w em $\mathcal{G}$, e escrevemos $w \Rightarrow_{\mathcal{G}} w'$, se w' pode ser obtida a partir de w substituindo-se uma variável que aparece em w pelo lado direito de alguma regra da gramática $\mathcal{G}$. Em outras palavras, para que $w \Rightarrow_{\mathcal{G}} w'$ é preciso que:

- exista uma decomposição da forma $w = uXv$, onde $u, v \in (\mathcal{T} \cup \mathcal{V})$ e $X \in \mathcal{V}$, e
- exista uma regra $X \to \alpha$ em $\mathcal{R}$ tal que $w' = u\alpha v$.

Como já estamos acostumados a fazer, dispensaremos o $\mathcal{G}$ subscrito na notação acima quando não houver dúvidas quanto à gramática que está sendo considerada (isto é, quase sempre!).

Como já ocorria com as gramáticas lineares à direita, estaremos gerando palavras a partir de uma gramática livre de contexto pelo encadeamento de várias derivações de um passo. Assim, dizer que w' pode ser *derivada a partir de* w *em* $\mathcal{G}$ *em* n *passos* significa que existem palavras $w_1, \ldots, w_{n-1} \in (\mathcal{T} \cup \mathcal{V})^*$ tais que

$$w = w_0 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_{n-1} \Rightarrow w_n = w'$$

Chamamos a isto uma *derivação* de w' a partir de w em $\mathcal{G}$ e escrevemos $w \Rightarrow^n w'$. Caso não seja conveniente indicar o número de passos da derivação substituímos o n por uma $*$ como já estamos acostumados a fazer. É conveniente também adotar a convenção de que w pode ser derivada dela própria em zero passos, de modo que faz sentido escrever $w \Rightarrow^* w$.

O conjunto de todas as palavras de $\mathcal{T}^*$ que podem ser derivadas a partir do símbolo inicial S na gramática $\mathcal{G}$ é a *linguagem gerada por* $\mathcal{G}$. Denotando $L(\mathcal{G})$ a linguagem gerada por $\mathcal{G}$, e usando a notação definida acima, temos que

$$L(\mathcal{G}) = \{w \in \mathcal{T}^* : \text{ existe uma derivação } S \Rightarrow^* w \text{ em } \mathcal{G}\}.$$

Vejamos alguns exemplos. Seja $\mathcal{G}_1$ a gramática definida acima. Temos que

$$S \Rightarrow 0S1 \Rightarrow 0^2S1^2 \Rightarrow 0^21^2.$$

Note que $S \Rightarrow 0S1$ e $0S1 \Rightarrow 0^2S1^2$ são derivações de um passo em $\mathcal{G}_1$ porque, em ambos os casos a palavra à direita de $\Rightarrow$ foi obtida da que está à esquerda trocando-se S por $0S1$. Isto é permitido porque $S \to 0S1$ é uma regra de $\mathcal{G}_1$. Já $0^2S1^2 \Rightarrow 0^21^2$ foi obtida trocando-se S por ϵ. Podemos fazer isto por que $S \to \epsilon$ também é uma regra de $\mathcal{G}_1$. Toda esta cadeia de derivações pode ser abreviada como

$$S \Rightarrow^3 0^21^1 \text{ ou } S \Rightarrow^* 0^21^1.$$

Por outro lado, $0^2S1^2 \Rightarrow 0^21^3$ *não* é uma derivação legítima em $\mathcal{G}_1$ porque neste caso a regra usada foi $S \to 1$, que não pertence a $\mathcal{G}_1$.

Do que fizemos acima segue que $0^21^2 \in L(\mathcal{G}_1)$. Mas podemos ser muito mais ambiciosos e tentar determinar todas as palavras de $L(\mathcal{G}_1)$. Para começar, note que se $S \Rightarrow^n w$ em $\mathcal{G}_1$ e $w \notin \mathcal{T}^*$, então $w = 0^nS1^n$. Para provar isto basta usar indução em n. Por outro lado, como a única regra de $\mathcal{G}_1$ que permite eliminar a variável S é $S \to \epsilon$, concluímos que toda palavra derivada a partir de S em $\mathcal{G}_1$ é da forma 0^n1^n, para algum inteiro $n \geq 0$. Assim,

$$L(\mathcal{G}_1) = \{0^n1^n : n \geq 0\}.$$

Já vimos no capítulo 4, que esta linguagem não é regular. Em particular, não existe nenhuma gramática linear à direita que gere $L(\mathcal{G}_1)$.

Passando à gramática $\mathcal{G}_2$, temos a derivação

$$S \Rightarrow \underline{X}1X \Rightarrow 0\underline{X}1X \Rightarrow 0^2X1\underline{X} \Rightarrow 0^2X1 \Rightarrow 0^21.$$

Note que, na derivação acima, algumas ocorrências da variável X foram sublinhadas. Fizemos isto para indicar à qual instância da variável X foi aplicada a regra da gramática que leva ao passo seguinte da derivação. Assim, no segundo passo da derivação, aplicamos a regra $X \to 0X$ ao X mais à esquerda $X1X$. Com isto este X foi trocado por $0X$, mas nada foi feito ao segundo X. Coisa semelhante ocorreu no passo seguinte. Além disso, a regra $X \to 0X$ nunca foi aplicada ao segundo X. Fica claro deste exemplo que:

> A cada passo de uma derivação apenas uma ocorrência de uma variável pode ser substituída pelo lado direito de uma seta. Além disso, cada ocorrência de uma variável é tratada independentemente da outra.

Sempre que for conveniente, sublinharemos a instância da variável à qual a regra está sendo aplicada em um determinado passo da derivação.

Vamos tentar, também neste caso, determinar todas as palavras que podem ser derivadas a partir de S em $\mathcal{G}_2$. Suponhamos que $S \Rightarrow^* w$ em $\mathcal{G}_2$. É fácil ver que em w

- há um único 1;
- pode haver, no máximo, duas ocorrências de X, uma de cada lado do 1.

Assim, w pode ser de uma das seguintes formas

$$0^nX10^mX \text{ ou } 0^n10^mX \text{ ou } 0^nX10^m \text{ ou } 0^n10^m$$

Portanto, se $w \in \mathcal{T}^*$ e $S \Rightarrow^* w$, então $w = 0^n10^m$, para inteiros $m, n \geq 0$. Concluímos que

$$L(\mathcal{G}_2) = \{0^n10^m : n, m \geq 0\}.$$

Mas esta é, na verdade, uma linguagem regular, que pode ser descrita na forma $L(\mathcal{G}_2) = 0^*10^*$. Temos assim um exemplo de gramática livre de contexto que não é linear à direita mas que gera uma linguagem regular.

Voltando ao caso geral, seja L uma linguagem no alfabeto Σ. Dizemos que L é *livre de contexto* se existe pelo menos uma gramática livre de contexto $\mathcal{G}$, com conjunto de terminais Σ, tal que $L(\mathcal{G}) = L$.

Portanto, toda linguagem regular é livre de contexto. Isto ocorre porque, pelo teorema de Kleene, uma linguagem regular sempre pode ser gerada por uma gramática linear à direita. Mas, como já vimos, toda gramática linear à direita é livre de contexto.

Por outro lado, nem toda linguagem livre de contexto é regular. Por exemplo, vimos na seção 3 do capítulo 4 que a linguagem $\{0^n 1^n : n \geq 0\}$ não é regular. No entanto, ela é gerada pela gramática $\mathcal{G}_1$ e, portanto, é livre de contexto. Na seção 3 veremos mais exemplos de linguagens livres de contexto, incluindo várias que não são regulares.

Antes que você comece a alimentar falsas esperanças, é bom esclarecer que existem linguagens que não são livres de contexto. Para mostrar, *diretamente da definição*, que uma linguagem L não é livre de contexto seria necessário provar que não existe nenhuma gramática livre de contexto que gere L, o que não parece possível. Como no caso de linguagens regulares, a estratégia mais prática consiste em mostrar que há uma propriedade de toda linguagem livre de contexto que não é satisfeita por L. Veremos como fazer isto no capítulo 11.

2. Linguagens sensíveis ao contexto

É hora de voltar à questão de como seria uma gramática que não é livre de contexto.

Lembre-se que o que caracteriza as gramáticas livres de contexto é o fato de que todas as suas regras têm apenas uma variável (e nenhum terminal) do lado esquerdo da seta. Na prática isto significa que sempre que X aparece em uma palavra, ele pode ser trocado por w, desde que $X \to w$ seja uma regra da gramática.

Para uma gramática não ser livre de contexto basta que, do lado esquerdo da seta de alguma de suas regras, apareça algo mais complicado que uma variável isolada. Um exemplo simples, mas importante, é a gramática com terminais $\{a, b, c\}$, variáveis $\{S, X, Y\}$, símbolo inicial S e regras

$$\begin{aligned} S &\to abc \\ S &\to aXbc \\ Xb &\to bX \\ Xc &\to Ybc^2 \\ bY &\to Yb \\ aY &\to a^2 \\ aY &\to a^2X. \end{aligned}$$

Considere uma derivação nesta gramática que comece com $S \Rightarrow aXbc$. Queremos, no próximo passo, substituir o X por alguma coisa. Mas, apesar de haver duas regras com X à esquerda da seta na gramática, só uma delas pode ser aplicada. Isto ocorre porque na palavra $aXbc$ o X

vem seguido de b, e a única regra em que X aparece neste 'contexto' é $Xb \to bX$. Aplicando esta regra, obtemos

$$S \Rightarrow aXbc \Rightarrow abXc \Rightarrow abYbc^2.$$

Continuando desta forma, podemos derivar a palavra $a^2b^2c^2$ nesta gramática. A derivação completa é a seguinte:

$$S \Rightarrow aXbc \Rightarrow abXc \Rightarrow abYbc^2 \Rightarrow aYb^2c^2 \Rightarrow a^2b^2c^2.$$

De maneira semelhante, podemos derivar qualquer palavra da forma $a^nb^nc^n$, com $n \geq 1$. De fato, pode-se mostrar que o conjunto das palavras desta forma constitui toda a linguagem gerada pela gramática dada acima.

Este exemplo se encaixa em uma classe de gramáticas chamadas de sensíveis ao contexto. A definição formal é a seguinte. Seja $\mathcal{G}$ uma gramática com conjunto de terminais T, conjunto de variáveis V e símbolo inicial S. Dizemos que $\mathcal{G}$ é *sensível ao contexto* se todas as regras de $\mathcal{G}$ são da forma $u \to v$, onde $u, v \in (\mathcal{T} \cup V)^*$ e $|u| \leq |v|$. Esta última condição é claramente satisfeita pelas regras de uma gramática livre de contexto, de modo que toda gramática livre de contexto é sensível ao contexto

Diremos que uma linguagem L no alfabeto Σ é *sensível ao contexto* se existe uma gramática sensível ao contexto, com conjunto de terminais Σ, que gera L. Portanto, toda linguagem livre de contexto é sensível ao contexto

A discussão acima mostra que a linguagem

$$\{a^nb^nc^n : n \geq 1\}$$

é sensível ao contexto. Entretanto, como provaremos no capítulo 11 não existe nenhuma gramática livre de contexto que gere esta linguagem. Temos, assim, que a classe das linguagens livres de contexto está propriamente contida na classe das linguagens sensíveis ao contexto. A relação entre os diversos tipos de linguagens é detalhada na hierarquia de Chomsky, que discutiremos em um capítulo posterior.

3. Mais exemplos

Nesta seção veremos mais exemplos de linguagens livres de contexto. Começamos com uma gramática que gera fórmulas que sejam expressões aritméticas envolvendo apenas os operadores soma e multiplicação, além dos parênteses. Isto é, expressões da forma

$$((x + y) * x) * (z + w) * y), \tag{3.1}$$

onde x, y, z e w são variáveis (no sentido em que o termo é usado em álgebra).

Note que, para saber se uma dada expressão é legítima, não precisamos conhecer as variáveis que nela aparecem, mas apenas sua localização em relação aos operadores. Assim, podemos construir uma gramática mais simples, em que as posições ocupadas por variáveis em uma expressão são todas marcadas com um mesmo símbolo. Este símbolo é chamado de *identificador*, e abreviado id. Portanto, o identificador id será um terminal da gramática que vamos construir, juntamente com os operadores $+$ e $*$ e os parênteses (e). Substituindo as variáveis da expressão (3.1) pelo identificador, obtemos

$$((\mathrm{id}+\mathrm{id})*\mathrm{id})*(\mathrm{id}+\mathrm{id})*\mathrm{id}). \tag{3.2}$$

Em resumo, queremos construir uma gramática livre de contexto $\mathcal{G}_{exp}$, com conjunto terminais $\{+,*,(,),\mathrm{id}\}$, que gere todas as expressões aritméticas legítimas nos operadores $+$ e $*$. Note que $\mathcal{G}_{exp}$ deve gerar todas as expressões legítimas, e apenas estas. Isto é, queremos gerar expressões como (3.2), mas não como

$$(+\mathrm{id})\mathrm{id}*\,.$$

Na prática estas expressões aritméticas são produzidas recursivamente, somando ou multiplicando expressões mais simples, sem esquecer de pôr os parênteses no devido lugar. Para obter a gramática, podemos criar uma variável E (de *expressão*), e introduzir as seguintes regras para combinação de expressões:

$$\begin{aligned} E &\to E+E \\ E &\to E*E \\ E &\to (E). \end{aligned}$$

Finalmente, para eliminar E e fazer surgir o identificador, adicionamos a regra $E \to \mathrm{id}$.

Vejamos como derivar a expressão $\mathrm{id}+\mathrm{id}*\mathrm{id}$ nesta gramática. Para deixar claro o que está sendo feito em cada passo, vamos utilizar a convenção, estabelecida na seção em 1, de sublinhar a instância da variável à qual uma regra está sendo aplicada:

(3.3)

$$\underline{E} \Rightarrow E+\underline{E} \Rightarrow \underline{E}+E*E \Rightarrow id+\underline{E}*E \Rightarrow \mathrm{id}+\mathrm{id}*\underline{E} \Rightarrow \mathrm{id}+\mathrm{id}*\mathrm{id}.$$

Construímos esta derivação aplicando em cada passo uma regra da gramática a uma variável escolhida sem nenhum critério especial. Entretanto podemos ser mais sistemáticos. Por exemplo, poderíamos, em

cada passo da derivação, aplicar uma regra sempre à variável que está mais à esquerda da palavra. Usando esta estratégia chegamos a uma derivação de $\mathrm{id} + \mathrm{id} * \mathrm{id}$ diferente da que foi obtida acima:

$$E \Rightarrow E + E \Rightarrow \mathrm{id} + E \Rightarrow \mathrm{id} + E * E \Rightarrow \mathrm{id} + \mathrm{id} * E \Rightarrow \mathrm{id} + \mathrm{id} * \mathrm{id}.$$

Neste caso não há necessidade de sublinhar a variável à qual a regra está sendo aplicada já que, em cada passo, escolhemos sempre a que está mais à esquerda da palavra.

As derivações deste tipo são muito importantes no desenvolvimento da teoria e em suas aplicações. Por isso é conveniente ter um nome especial para elas. Seja $\mathcal{G}$ uma gramática livre de contexto e $w \in L(\mathcal{G})$. Uma *derivação mais à esquerda* de w em $\mathcal{G}$ é aquela na qual, em cada passo, a variável à qual a regra é aplicada é a que está mais à esquerda da palavra. Analogamente, podemos definir *derivação mais à direita* em uma gramática livre de contexto.

Uma gramática como $\mathcal{G}_{exp}$ é usada em uma linguagem de programação com duas finalidades diferentes. Em primeiro lugar, para verificar se as expressões aritméticas de um programa estão bem construídas; em segundo, para informar ao computador qual é a interpretação correta destas expressões. Por exemplo, o computador tem que ser informado sobre a precedência correta entre os operadores soma e multiplicação. Isto é necessário para que, na ausência de parêntesis, o computador saiba que deve efetuar primeiro as multiplicações e só depois as somas. Do contrário, confrontado com $\mathrm{id} + \mathrm{id} * \mathrm{id}$ ele não saberia como efetuar o cálculo da forma correta. Voltaremos a esta questão em mais detalhes no próximo capítulo.

Analisando $\mathcal{G}_{exp}$ com cuidado, verificamos que permite derivar (id). Apesar do uso dos parênteses ser desnecessário neste caso, não se trata de uma expressão aritmética ilegítima. Entretanto, não é difícil resolver este problema introduzindo uma nova variável, como é sugerido no exercício 5.

O segundo exemplo que desejamos considerar é o de uma gramática que gere a linguagem formada pelos palíndromos no alfabeto $\{0, 1\}$. Lembre-se que um palíndromo é uma palavra cujo reflexo é igual a ela própria. Isto é, w é um palíndromo se e somente se $w^R = w$. Precisamos de dois fatos sobre palíndromos que são conseqüência imediata da definição. Digamos que $w \in (0 \cup 1)^*$ e que σ é 0 ou 1; então:

(1) w é palíndromo se e somente se w começa e termina com o mesmo símbolo;
(2) $\sigma w \sigma$ é palíndromo se e somente se w também é palíndromo.

Isto sugere que os palíndromos podem ser construídos recursivamente onde, a cada passo, ladeamos um palíndromo já construído por duas letras iguais.

Com isto podemos passar à construção da gramática. Vamos chamá-la de $\mathcal{G}_{pal}$: terá terminais $\{0, 1\}$ e apenas uma variável S, que fará o papel de símbolo inicial. As observações acima sugerem as seguintes regras

$$S \to 0S0$$
$$S \to 1S1$$

Estas regras ainda não bastam, porque apenas com elas não é possível eliminar S da palavra. Para fazer isto precisamos de, pelo menos, mais uma regra. A primeira idéia seria introduzir $S \to \epsilon$. Entretanto, se fizermos isto só estaremos gerando palíndromos que não têm um símbolo no meio; isto é, os que têm comprimento par. Para gerar os de comprimento ímpar é necessário também introduzir regras que permitam substituir S por um terminal; isto é,

$$S \to 0 \text{ e } S \to 1.$$

Podemos resumir o que fizemos usando a seguinte notação. Suponha que $\mathcal{G}$ é uma gramática livre de contexto que tem várias regras

$$X \to w_1, \ldots, X \to w_k$$

todas com uma mesma variável do lado esquerdo. Neste caso escrevemos abreviadamente

$$X \to w_1 \,|w_2 \,\ldots\, |w_k,$$

onde a barra vertical tem o valor de 'ou'. Como G_{pal} tem uma única variável, podemos escrever todas as suas regras na forma

$$S \to 0S0 \,|\, 1S1 \,|\, \epsilon \,|\, 0 \,|\, 1.$$

De quebra, obtivemos a gramática $\mathcal{G}^+_{pal}$ que gera os palíndromos de comprimento par. A única diferença entre as duas gramáticas é que o conjunto de regras de $\mathcal{G}^+_{pal}$ é ainda mais simples:

$$S \to 0S0 \,|\, 1S1 \,|\, \epsilon.$$

4. Combinando gramáticas

Muitas vezes é possível decompor uma linguagem livre de contexto L como união, concatenação ou estrela de outras linguagens livres de contexto. Nesta seção introduzimos técnicas que facilitam a construção

de uma gramática livre de contexto para L a partir das gramáticas das suas componentes.

Começaremos com um exemplo cuja verdadeira natureza só será revelada no próximo capítulo. Considere a linguagem

$$L_{in} = \{a^i b^j c^k : i, j, k \geq 0 \text{ e } \ i = j \text{ ou } \ j = k\}.$$

Podemos decompô-la, facilmente, como união de outras duas, a saber

$$L_1 = \{a^i b^j c^k : i, j, k \geq 0 \text{ e } \ i = j\}, \quad \text{e}$$
$$L_2 = \{a^i b^j c^k : i, j, k \geq 0 \text{ e } \ j = k\}.$$

Além disso, $L_1 = L_3 \cdot c^*$ e $L_2 = a^* \cdot L_4$, onde

$$L_3 = \{a^i b^j : i, j \geq 0 \text{ e } \ i = j\}, \text{ e } \ L_4 = \{b^j c^k : j, k \geq 0 \text{ e } \ j = k\}.$$

Chegados a este ponto, já conseguimos obter uma decomposição de L_{in} em linguagens para as quais conhecemos gramáticas. De fato, a^* e c^* são linguagens regulares para as quais existem gramáticas lineares à direita bastante simples. Por outro lado, a gramática $\mathcal{G}_1$ definida na seção 1 gera uma linguagem análoga a L_3 e L_4. Resta-nos descobrir como esta decomposição pode ser usada para construir uma gramática para L_{in}. Contudo, é preferível discutir o problema de combinar gramáticas em geral, e só depois aplicá-lo no exemplo acima.

Suponhamos, então, que $\mathcal{G}_1$ e $\mathcal{G}_2$ são gramáticas livres de contexto que geram linguagens L_1 e L_2, respectivamente. O que queremos são receitas que nos permitam obter gramáticas livres de contexto que gerem $L_1 \cup L_2$ e $L_1 \cdot L_2$ a partir de $\mathcal{G}_1$ e $\mathcal{G}_2$.

Digamos que $\mathcal{G}_1$ tem ingredientes $(\mathcal{T}, \mathcal{V}_1, \mathcal{S}_1, \mathcal{R}_1)$ e $\mathcal{G}_2$ ingredientes $(\mathcal{T}, \mathcal{V}_2, \mathcal{S}_2, \mathcal{R}_2)$. Observe que estamos supondo que as duas gramáticas têm os mesmos terminais, e que os conjuntos de variáveis são disjuntos; isto é, $\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset$.

Vamos começar com $L_1 \cup L_2$. Neste caso a nova gramática, que chamaremos de $\mathcal{G}_\cup$, deve gerar uma palavra que pertença a L_1 ou a L_2. Assim, o conjunto de regras de $\mathcal{G}_\cup$ precisa conter $\mathcal{R}_1$ e $\mathcal{R}_2$. Mas queremos também que, depois do primeiro passo, a derivação só possa proceder usando regras de uma das duas gramáticas. Fazemos isto criando um novo símbolo inicial S e duas novas regras

$$S \to S_1 \text{ e } \ S \to S_2.$$

Assim, o primeiro passo da derivação força uma escolha entre S_1 e S_2. Esta escolha determina de maneira inequívoca se a palavra a ser gerada estará em L_1 ou L_2. Em outras palavras, depois do primeiro passo a derivação fica obrigatoriamente restrita às regras de uma das duas gramáticas. Mais precisamente, $\mathcal{G}_\cup$ fica definida pelos seguintes ingredientes:

Terminais: $\mathcal{T}_1 \cup \mathcal{T}_2$;
Variáveis: $\mathcal{V}_1 \cup \mathcal{V}_2 \cup \{S\}$;
Símbolo inicial: S;
Regras: $\mathcal{R}_1 \cup \mathcal{R}_2 \cup \{S \to S_1, S \to S_2\}$.

Podemos proceder de maneira semelhante para criar uma gramática $\mathcal{G}_\bullet$ que gere a concatenação $L_1 \cdot L_2$. Neste caso, as palavras que queremos derivar são construídas escrevendo uma palavra de L_1 seguida de uma palavra de L_2. Como as palavras de L_1 são geradas a partir de S_1 e as de L_2 a partir de S_2, basta acrescentar $S \to S_1 S_2$ às regras de $\mathcal{G}_1$ e $\mathcal{G}_2$. Os ingredientes da gramática $\mathcal{G}_\bullet$ são:

Terminais: $\mathcal{T}_1 \cup \mathcal{T}_2$;
Variáveis: $\mathcal{V}_1 \cup \mathcal{V}_2 \cup \{S\}$;
Símbolo inicial: S;
Regras: $\mathcal{R}_1 \cup \mathcal{R}_2 \cup \{S \to S_1 S_2\}$.

Finalmente, há uma receita semelhante às que foram dadas acima para criar uma gramática livre de contexto para L^* a partir de uma gramática livre de contexto que gere L. Deixamos os detalhes desta construção para o exercício 7.

Podemos, agora, voltar à linguagem L_{in}. Como $L_{in} = L_3 \cdot c^* \cup a^* \cdot L_4$, precisamos começar criando gramáticas que gerem as linguagens L_3, L_4, c^* e a^*. Para L_3 e L_4 podemos usar adaptações da gramática $\mathcal{G}_1$ da seção 1. Já c^* e a^* são regulares, geradas por gramáticas lineares à direita extremamente simples. Resumimos os ingredientes destas várias gramáticas na tabela abaixo:

Linguagem	L_3	L_4	a^*	c^*
Terminais	$\{0,1\}$	$\{0,1\}$	$\{0,1\}$	$\{0,1\}$
Variáveis	S_1	S_2	S_3	S_4
Símbolo inicial	S_1	S_2	S_3	S_4
Regras	$S_1 \to aS_1b$ $S_1 \to \epsilon$	$S_2 \to bS_2c$ $S_2 \to \epsilon$	$S_3 \to aS_3$ $S_4 \to \epsilon$	$S_4 \to cS_4$ $S_3 \to \epsilon$

Seguindo a receita dada acima para a gramática de uma concatenação, obtemos:

Linguagem	$L_3 \cdot c^*$	$a^* \cdot L_4$
Terminais	$\{0,1\}$	$\{0,1\}$
Variáveis	$\{S', S_1, S_4\}$	$\{S'', S_2, S_3\}$
Símbolo inicial	S'	S''
Regras	$S' \to S_1 S_4$	$S'' \to S_2 S_3$
	$S_1 \to aS_1 b$	$S_2 \to bS_2 c$
	$S_1 \to \epsilon$	$S_2 \to \epsilon$
	$S_4 \to cS_4$	$S_3 \to aS_3$
	$S_3 \to \epsilon$	$S_4 \to \epsilon$

Resta-nos, apenas, proceder à união destas gramáticas, o que nos dá uma gramática livre de contexto para L_{in}, com os seguintes ingredientes:

Terminais: $\{0,1\}$;
Variáveis: $S, S', S'', S_1, S_2, S_3, S_4$;
Símbolo inicial: S;
Regras: $\{S \to S', S \to S'', S' \to S_1 S_4, S'' \to S_2 S_3, S_1 \to aS_1 b, S_2 \to bS_2 c, S_1 \to \epsilon, S_2 \to \epsilon, S_4 \to cS_4, S_3 \to aS_3, S_3 \to \epsilon, S_4 \to \epsilon\}$.

5. Exercícios

1. Considere a gramática $\mathcal{G}$ com variáveis S, A, terminais a, b, símbolo inicial S e regras

$$S \to AA$$
$$A \to AAA \,|\, a \,|\, bA \,|\, Ab$$

   (a) Quais palavras de $L(\mathcal{G})$ podem ser produzidas com derivações de até 4 passos?
   (b) Dê pelo menos 4 derivações distintas da palavra $babbab$.
   (c) Para $m, n, p > 0$ quaisquer, descreva uma derivação em $\mathcal{G}$ de $b^m a b^n a b^p$.

2. Determine gramáticas livres de contexto que gerem as seguintes linguagens:
   (a) $\{(01)^i : i \geq 1\}$;
   (b) $\{1^{2n} : n \geq 1\}$;
   (c) $\{0^i 1^{2i} : i \geq 1\}$;
   (d) $\{w \in \{0,1\}^* : w$ em que o número de 0s e 1s é o mesmo$\}$;
   (e) $\{wcw^r : w \in \{0,1\}^*\}$;

(f) $\{w : w = w^r \text{ onde } w \in \{0,1\}\}$.

3. Considere o alfabeto $\Sigma = \{0, 1, (,), \cup, *, \emptyset\}$. Construa uma gramática livre de contexto que gere todas as palavras de Σ^* que são expressões regulares em $\{0, 1\}$.

4. A gramática livre de contexto $\mathcal{G}$ cujas regras são
$$S \to 0S1 \,|\, 0S0 \,|\, 1S0 \,|\, 1S1 \,|\, \epsilon$$
não é linear à direita. Entretanto, $L(\mathcal{G})$ é uma linguagem regular. Ache uma gramática linear à direita $\mathcal{G}'$ que gere $L(\mathcal{G})$.

5. Modifique a gramática $\mathcal{G}_{exp}$ (introduzindo novas variáveis) de modo que não seja mais possível derivar a expressão (id) nesta gramática.

6. Seja $\mathcal{G}$ uma gramática livre de contexto com conjunto de terminais T, conjunto de variáveis V e símbolo inicial S. Dizemos que $\mathcal{G}$ é *linear* se todas as suas regras são da forma
$$X \to \alpha Y \beta \text{ ou } X \to \alpha,$$
onde X e Y são variáveis e $\alpha, \beta \in T^*$. Isto é pode haver no máximo uma variável do lado direito de cada regra de $\mathcal{G}$. Uma linguagem é *linear* se pode ser gerada por alguma gramática linear. Mostre que se L é linear então L pode ser gerada por uma gramática cujas regras são de um dos 3 tipos seguintes:
$$X \to \sigma Y \text{ ou } X \to Y\sigma \text{ ou } X \to \sigma$$
onde X é uma variável e σ é um terminal ou $\sigma = \epsilon$.

7. Seja $\mathcal{G}$ uma gramática livre de contexto que gere uma linguagem L. Mostre como construir, a partir de $\mathcal{G}$, uma gramática livre de contexto que gera L^*.
SUGESTÃO: $S \to SS$.

8. Seja L uma linguagem livre de contexto. Mostre que $L^r = \{w^r : w \in L\}$ também é livre de contexto.

CAPíTULO 10

Árvores Gramaticais

No capítulo anterior vimos como gerar palavras a partir de uma gramática livre de contexto, por derivação. Neste capítulo consideramos outra maneira pela qual uma gramática livre de contexto gera palavras: as árvores gramaticais. Esta noção tem origem na necessidade de diagramar a análise sintática de uma sentença em uma língua natural, como o português ou o javanês.

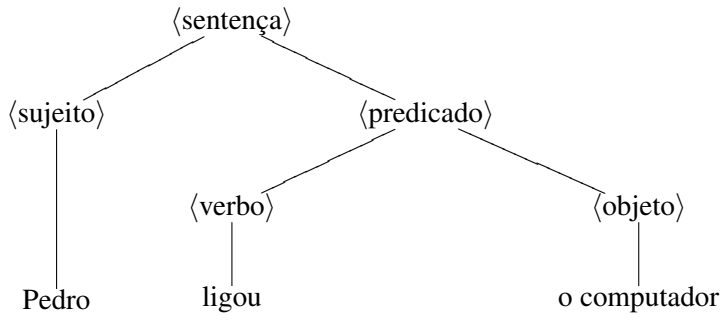
1. Análise Sintática e línguas naturais

Até agora as gramáticas formais que introduzimos têm servido basicamente para gerar palavras que pertencem a uma dada linguagem. Já a gramática da língua portuguesa não serve apenas para gerar frases com sintaxe correta, mas também para analisar a estrutura de uma frase e determinar o seu significado. A isso se chama análise sintática. Descobrir quem fez o quê, e a quem, é uma questão de identificar sujeito, verbo e objetos de uma frase. Entretanto, a noção de derivação não é uma ferramenta adequada para realizar a análise sintática em uma gramática; para isto precisamos das árvores gramaticais.

Vejamos como usar uma árvore para diagramar a análise sintática da frase

Pedro ligou o computador.

O sujeito da sentença é *Pedro* e o predicado é *ligou o computador*. Por sua vez o predicado pode ser decomposto no verbo *ligou* seguido do objeto direto, *o computador*. Esta análise da frase pode ser diagramada em uma árvore como a da figura abaixo.



Note que as palavras sentença, sujeito, predicado, verbo e objeto direto aparecem entre ⟨ ⟩. Fizemos isto para diferenciar o conceito, da palavra da língua portuguesa que o denota. Em outras palavras, ⟨ sujeito ⟩ indica que na gramática da língua portuguesa há uma variável chamada de sujeito.

Existe uma estreita relação entre a estrutura da árvore acima e as regras da gramática da língua portuguesa. Lendo esta árvore de cima para baixo, cada bifurcação corresponde a uma regra da gramática portuguesa. As regras que aparecem na árvore da figura acima são as seguintes:

⟨sentença⟩ → ⟨sujeito⟩⟨predicado⟩
⟨predicado⟩ → ⟨verbo⟩⟨objeto direto⟩
⟨sujeito⟩ → Pedro
⟨verbo⟩ → ligou
⟨objeto direto⟩ → o computador.

Assim podemos usar esta árvore para derivar a frase "Pedro ligou o computador" a partir das regras acima:

$$\langle\text{sentença}\rangle \Rightarrow \langle\text{sujeito}\rangle\langle\text{predicado}\rangle \Rightarrow \langle\text{sujeito}\rangle\langle\text{verbo}\rangle\langle\text{objeto direto}\rangle \Rightarrow$$
$$\text{Pedro}\langle\text{verbo}\rangle\langle\text{objeto direto}\rangle \Rightarrow \text{Pedro ligou } \langle\text{objeto direto}\rangle$$
$$\Rightarrow \text{Pedro ligou o computador}$$

O principal resultado deste capítulo mostra que existe uma estreita relação entre árvores gramaticais e derivações. Antes, porém, precisaremos definir formalmente a noção de árvore gramatical de uma linguagem livre de contexto.

2. Árvores Gramaticais

Nesta seção, além de introduzir formalmente o conceito de árvore gramatical de uma linguagem livre de contexto, estabelecemos a relação entre árvores e derivações.

Lembre-se que uma árvore é um grafo conexo que não tem ciclos. As árvores que usaremos são na verdade grafos orientados. Entretanto, não desenharemos as arestas destas árvores como setas. Em vez disso, indicaremos que o vértice v precede v' desenhando v acima de v'. Além disso suporemos sempre que estamos lidando com árvores que satisfazem as seguintes propriedades:

- há um único vértice, que será chamado de *raiz*, e que não é precedido por nenhum outro vértice;
- os sucessores de um vértice estão totalmente ordenados.

Se v' e v'' são sucessores de v, e se v' precede v'' na ordenação dos vértices, então desenharemos v' à esquerda de v''. De agora em diante a palavra árvore será usada para designar um grafo com todas as propriedades relacionadas acima.

————Falta um grafo aqui————

Um exemplo de árvore com estas propriedades, é a árvore genealógica que descreve os descendentes de uma dada pessoa. Note que, em uma árvore genealógica, os irmãos podem ser naturalmente ordenados pela ordem do nascimento. A terminologia que passamos a descrever é inspirada neste exemplo. Suponhamos que $\mathcal{T}$ é uma árvore e que v' é um vértice de $\mathcal{T}$. Então há um único caminho ligando a raiz a v'. Digamos que este caminho contém um vértice v. Neste caso,

- v é *ascendente* de v' e v' é *descendente* de v;
- se v e v' estão separados por apenas uma aresta então v é *pai* de v' e v' é *filho* de v;
- se v' e v'' são filhos de um mesmo pai, então são *irmãos*;
- um vértice sem filhos é chamado de *folha*;
- um vértice que não é uma folha é chamado de *vértice interior*.

A ordenação entre irmãos, que faz parte da definição de árvore, pode ser estendida a uma ordenação de todas as folhas. Para ver como isto é feito, digamos que f' e f'' são duas folhas de uma árvore. Começamos procurando o seu primeiro ascendente comum, que chamaremos de v. Observe que f e f'' têm que ser descendentes de filhos diferentes de v, a não ser que sejam eles próprios filhos de v. Digamos que f é descendente de v' e f'' de v''. Como v' e v'' são irmãos, sabemos que estão ordenados; digamos que v' precede v''. Neste caso dizemos que

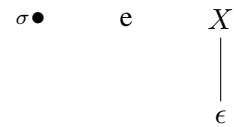
f' precede f''. Esta é uma ordem total; isto é, todas as folhas de uma tal árvore podem ser escritas em fila, de modo que a seguinte sucede à anterior.

Nenhuma preocupação adicional com esta ordenação é necessáriaão na hora de desenhar uma árvore; basta obedecer à convenção de sempre ordenar os vértices irmãos da esquerda para a direita. Se fizermos isto, as folhas ficarão automaticamente ordenadas da esquerda para a direita.

Seja $\mathcal{G}$ uma gramática livre de contexto com terminais T e variáveis V. De agora em diante estaremos considerando árvores, no sentido acima, cujos vértices estão rotulados por elementos de $T \cup V$. Contudo, não estaremos interessados em todas estas árvores, mas apenas naquelas resultantes de um procedimento recursivo bastante simples, as árvores gramaticais. Como se trata de uma definição recursiva, precisamos estabelecer quem são os átomos da construção, que chamaremos de árvores básicas, e de que maneira podemos combiná-las.

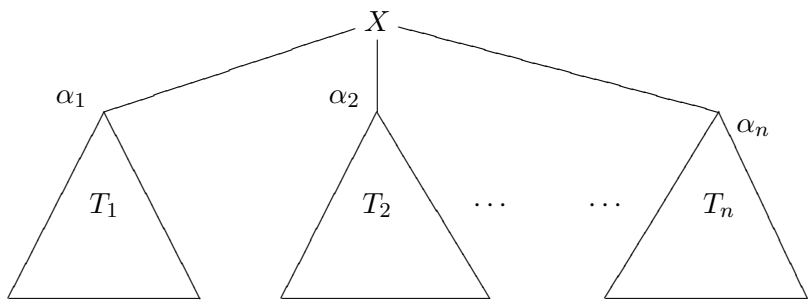
As *árvores gramaticais* são definidas recursivamente da seguinte maneira:

Árvores básicas: se $\sigma \in T$, $X \in V$ e $X \to \epsilon$ é uma regra de $\mathcal{G}$, então



são árvores gramaticais e suas colheitas são, respectivamente, σ e ϵ.

Regras de combinação: sejam $T_1, \ldots, T_n$ árvores gramaticais, e suponhamos que o rótulo da raiz de T_j é α_j. Se $X \to \alpha_1 \cdots \alpha_n$ é uma regra de $\mathcal{G}$, então a árvore T definida por



também é uma árvore gramatical.

Um exemplo simples é a árvore na gramática $\mathcal{G}_{exp}$ (definida na seção 3 do capítulo 7) esboçada abaixo.

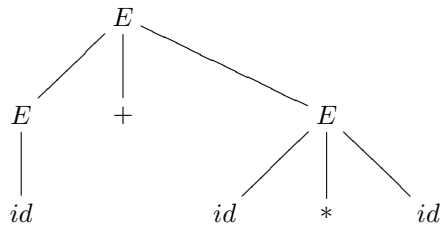


FIGURA 1. Árvore gramatical

Uma conseqüência muito importante desta definição recursiva, e uma que usaremos em várias oportunidades, é a seguinte. Suponhamos que uma árvore gramatical $\mathcal{T}$ tem um vértice v rotulado por uma variável X. Então podemos trocar toda a parte de $\mathcal{T}$ que descende de v por qualquer outra árvore gramatical cuja raiz seja rotulada por X.

Segue também da definição que os únicos vértices de uma árvore gramatical que podem ser rotulados por elementos de $T \cup \{\epsilon\}$ são as folhas. Portanto, um vértice interior de uma árvore gramatical só pode ser rotulado por uma variável.

Por outro lado, se o vértice v de uma árvore gramatical $\mathcal{T}$ está rotulado por uma variável X, e seus filhos por $\alpha_1, \ldots, \alpha_n \in T \cup V$ então $X \to \alpha_1 \cdots \alpha_n$ tem que ser uma regra da gramática $\mathcal{G}$. Diremos que esta é a regra *associada* ao vértice v. No caso de v ser a raiz, $\mathcal{T}$ é uma X-*árvore*. Caso a árvore gramatical consista apenas de uma folha rotulada por um terminal σ, diremos que se trata de uma σ-árvore.

Uma vez tendo introduzido árvores gramaticais temos uma outra maneira de gerar palavras a partir de uma gramática livre de contexto. Para isso, definimos a *colheita* $c(\mathcal{T})$ de uma árvore gramatical $\mathcal{T}$. Como a definição de árvore é recursiva, assim será a definição de colheita. As colheitas das árvores básicas são

$$c(\bullet\,\sigma) = \sigma \quad \text{e} \quad c\begin{pmatrix} X \\ | \\ \epsilon \end{pmatrix} = \epsilon$$

Por outro lado sejam $T_1, \ldots, T_n$ árvores gramaticais, e suponhamos que o rótulo da raiz de T_j é α_j. Se $X \to \alpha_1 \cdots \alpha_n$ é uma regra de $\mathcal{G}$, então a árvore T construída de acordo com a *regra de combinação* satisfaz

$$c(T) = c(T_1) \cdot c(T_2) \cdots c(T_n).$$

Como conseqüência da definição recursiva, temos que a colheita de uma árvore gramatical T é a palavra obtida concatenando-se os rótulos

de todas as folhas de T, da esquerda para a direita. Como as folhas de uma árvore gramatical estão totalmente ordenadas, não há nehuma ambigüidade nesta maneira de expressar a colheita. Para uma demonstração formal deste fato veja o exercício 1. Portanto, a árvore da figura 1 tem colheita $\mathrm{id} + \mathrm{id} * \mathrm{id}$.

Digamos que w é uma palavra que pode ser derivada em uma gramática livre de contexto $\mathcal{G}$ com símbolo inicial S. Uma *árvore de derivação* para w é uma S-árvore de $\mathcal{G}$ cuja colheita é w. Note que reservamos o nome de árvore de derivação para o caso especial das S-árvores.

3. Colhendo e derivando

No capítulo 7 vimos como gerar uma palavra em terminais a partir do símbolo inicial de uma linguagem livre de contexto por derivação. A noção de colheita de uma árvore gramatical nos dá uma segunda maneira de produzir uma tal palavra. Como seria de esperar, há uma estreita relação entre estes dois métodos, que será discutida em detalhes nesta seção.

Para explicitar a relação entre árvores gramaticais e derivações vamos descrever um algoritmo que constrói uma derivação mais à esquerda de uma palavra a partir de sua árvore gramatical. Heuristicamente falando, o algoritmo desmonta a árvore gramatical da raiz às folhas. Contudo, ao remover a raiz de uma árvore gramatical, não obtemos uma nova árvore gramatical, mas sim uma seqüência ordenada de árvores. Isto sugere a seguinte definição. Seja $\mathcal{G}$ uma gramática livre de contexto com terminais T e variáveis V. Se

$$w = \alpha_1 \cdots \alpha_n \in (T \cup V)^*$$

então uma uma *w-floresta* $\mathcal{F}$ é uma seqüência ordenada $T_1, \ldots, T_n$ de árvores gramaticais, onde T_j é uma α_j-árvore. A *colheita da floresta* $\mathcal{F}$ é a concatenação das colheitas de suas árvores; isto é

$$c(\mathcal{F}) = c(T_1) \cdots c(T_n).$$

Podemos agora descrever o algoritmo. Lembre-se que quando dizemos 'remova a raiz da árvore $\mathcal{T}$' estamos implicitamente assumindo que as arestas incidentes à raiz também estão sendo removidas. Se v é a raiz de uma árvore $\mathcal{T}$ da floresta $\mathcal{F}$, denotaremos por $\mathcal{F} \setminus v$ a floresta obtida quando v é removido de $\mathcal{T}$.

Precisamos considerar, separadamente, o efeito desta construção quando $\mathcal{T}$ é a árvore básica que tem a raiz rotulada pela variável X e uma única folha rotulada por ϵ. Neste caso, quando removemos a raiz sobra apenas um vértice rotulado por ϵ, que não constitui uma árvore

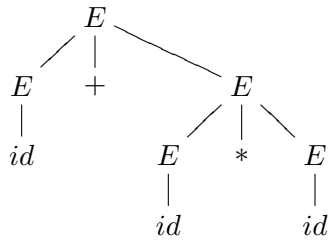
gramatical. Suporemos, então, que remover a raiz de uma tal árvore tem o efeito de apagar toda a árvore.

ALGORITMO 10.1. *Constrói uma derivação a partir de uma árvore gramatical.*

> **Entrada:** *uma X-árvore $\mathcal{T}$, onde X é uma variável.*
> **Saída:** *uma derivação mais à esquerda de $c(\mathcal{T})$.*
>
> **Etapa 1:** *Inicialize $\mathcal{F}$ com $\mathcal{T}$.*
> **Etapa 2:** *Se $\mathcal{F}$ é uma w-floresta, então imprima w. Páre se todas as árvores de $\mathcal{F}$ são rotuladas por terminais.*
> **Etapa 3:** *Seja v a raiz da árvore mais à esquerda de $\mathcal{F}$ que é rotulada por uma variável. Faça $\mathcal{F} = \mathcal{F} \setminus v$ e volte à etapa 2.*

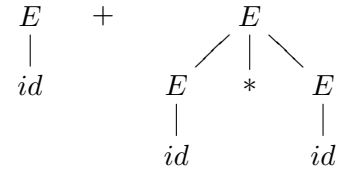
À seguir, você encontrará um exemplo da aplicação passo a passo deste algoritmo a uma árvore na gramática $\mathcal{G}_{exp}$.

Inicialização



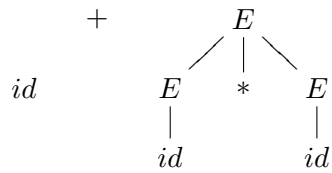
Imprime: E

Passo 1



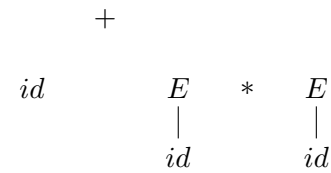
Imprime: $E + E$

Passo 2

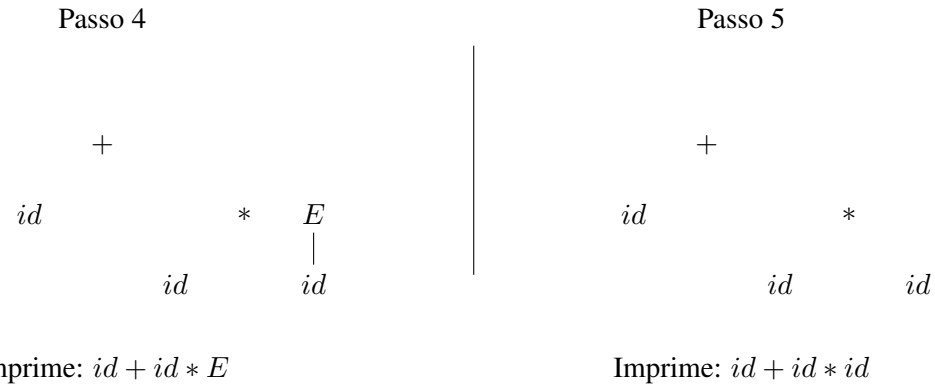


Imprime: $id + E$

Passo 3



Imprime: $id + E * E$



Resta-nos demonstrar que este algoritmo funciona. Começamos por analisar o que o algoritmo faz quando um de seus passos é executado. Suponhamos que, ao final do k-ésimo passo, temos uma $\alpha_1 \cdots \alpha_n$-floresta $\mathcal{F}$ formada pelas árvores $\mathcal{T}_1, \ldots, \mathcal{T}_n$. Portanto, a palavra impressa pelo algoritmo no passo k é

$$\alpha_1 \cdots \alpha_n.$$

Digamos que α_j é uma variável, mas que $\alpha_1, \ldots, \alpha_{j-1}$ são terminais. Portanto, a árvore mais à esquerda de $\mathcal{F}$, cuja raiz é rotulada por uma variável, é $\mathcal{T}_j$. Assim, ao executar o $(k+1)$-ésimo passo do algoritmo deveremos remover a raiz de $\mathcal{T}_j$. Mas ao fazer isto estamos substituindo $\mathcal{T}_j$ em $\mathcal{F}$ por uma $\beta_1 \cdots \beta_r$-floresta, onde $\alpha_j \to \beta_1 \cdots \beta_r$ é a regra associada à raiz de $\mathcal{T}_j$. Ao final deste passo, o algoritmo terá impresso

$$\alpha_1 \cdots \alpha_{j-1}\beta_1 \cdots \beta_r\alpha_{j+1} \cdots \alpha_n.$$

Entretanto, α_j era a variável mais à esquerda de $\alpha_1 \cdots \alpha_n$, e

$$\alpha_1 \cdots \alpha_{j-1}\alpha_j\alpha_{j+1} \cdots \alpha_n \Rightarrow \alpha_1 \cdots \alpha_{j-1}\beta_1 \cdots \beta_r\alpha_{j+1} \cdots \alpha_n.$$

Como o algoritmo começa imprimindo X, podemos concluir que produz uma derivação mais à esquerda em $\mathcal{G}$, a partir de X. Falta apenas mostrar que o que é derivado é mesmo $c(\mathcal{T})$. Contudo, a colheita das florestas a cada passo da aplicação do algoritmo é sempre a mesma. Além disso, as árvores gramaticais que constituem a floresta no momento que o algoritmo termina têm suas raízes indexadas por terminais. Como o algoritmo não apaga nenhum vértice indexado por terminal diferente de ϵ, concluímos que a concatenação das raízes das árvores no momento em que o algoritmo pára é $c(T)$, como desejávamos.

4. Equivalência entre árvores e derivações

Passemos à recíproca da questão considerada na seção 3. Mais precisamente, queremos mostrar que se $\mathcal{G}$ é uma gramática livre de contexto então toda palavra que tem uma derivação em $\mathcal{G}$ é colheita de alguma árvore de derivação de $\mathcal{G}$.

Para resolver este problema usando um algoritmo precisaríamos inventar uma receita recursiva para construir uma árvore de derivação a partir de uma derivação qualquer em $\mathcal{G}$. Isto é possível, mas o algoritmo resultante não é tão enxuto quanto o anterior. Por isso vamos optar por dar uma demonstração indireta, por indução.

PROPOSIÇÃO 10.2. *Seja X uma variável da gramática livre de contexto $\mathcal{G}$. Se existe uma derivação $X \Rightarrow^* w$, então w é colheita de uma X-árvore em $\mathcal{G}$.*

DEMONSTRAÇÃO. Suponhamos que a gramática livre de contexto $\mathcal{G}$ tem terminais T e variáveis V. A proposição será provada por indução no número p de passos de uma derivação $X \Rightarrow^p w$.

A base da indução consiste em supor que existe uma derivação $X \Rightarrow w$ de apenas um passo. Mas isto só pode ocorrer se existem terminais $t_1, \ldots, t_n$ e uma regra da forma $X \to t_1 \cdots t_n$. Assim, w é a colheita de uma X-árvore que tem a raiz rotulada por X e as folhas por $t_1, \ldots, t_n$, o que prova a base da indução.

A hipótese de indução afirma que, se $Y \in \mathcal{V}$ e se existe uma derivação

$$Y \Rightarrow^p u \in \mathcal{T}^*,$$

então u é colheita de uma Y-árvore de $\mathcal{G}$. Digamos, agora, que $w \in T^*$ pode ser derivado a partir de $X \in V$ em $p+1$ passos. O primeiro passo desta derivação será da forma $X \Rightarrow v_1 v_2 \cdots v_n$, onde $v_1, \ldots, v_n \in T \cup V$ e $X \to v_1 v_2 \cdots v_n$ é uma regra de $\mathcal{G}$. A derivação continua com cada v_i deflagrando uma derivação da forma $v_i \Rightarrow^* u_i$ onde $u_1 \cdots u_n = w$. Como cada uma destas derivações tem comprimento menor ou igual a p, segue da hipótese de indução que existem v_i-árvores $\mathcal{T}_i$ com colheita u_i, para cada $i = 1, \ldots, n$. Mas $\mathcal{T}_1, \ldots, \mathcal{T}_n$ é uma $v_1 \cdots v_n$-floresta, e a primeira regra utilizada na derivação de w foi $X \to v_1 \cdots v_n$. Portanto, pela definição de árvore gramatical, podemos colar as raízes desta floresta de modo a obter uma X-árvore cuja colheita é

$$c(\mathcal{T}_1) \cdots c(\mathcal{T}_n) = u_1 \cdots u_n = w,$$

o que prova o passo de indução. O resultado desejado segue pelo princípio de indução finita.

Só nos resta reunir, para referência futura, tudo o que aprendemos sobre a relação entre derivações e árvores gramaticais em um único teorema. Antes de enunciar o teorema, porém, observe que tudo o que fizemos usando derivações mais à esquerda vale igualmente para derivações mais à direita.

TEOREMA 10.3. *Seja $\mathcal{G}$ uma gramática livre de contexto e $w \in L(\mathcal{G})$. Então:*

(1) *existe uma árvore de derivação cuja colheita é w;*
(2) *a cada árvore de derivação cuja colheita é w corresponde uma única derivação mais à esquerda de w;*
(3) *a cada árvore de derivação cuja colheita é w corresponde uma única derivação mais à direita de w.*

Temos que (1) segue da proposição acima e que (2) é conseqüência do algoritmo da seção 3. Já (3) segue de uma modificação óbvia deste mesmo algoritmo. A importância deste teorema ficará clara nos próximos capítulos.

5. Ambigüidade

Como vimos na seção 1, as árvores gramaticais são usadas na gramática da língua portuguesa para representar a análise sintática de uma frase em um diagrama. Portanto, têm como finalidade ajudar-nos a interpretar corretamente uma frase.

Contudo, é preciso não esquecer que é possível escrever sentenças gramaticalmente corretas em português que, apesar disso, admitem duas interpretações distintas. Por exemplo, a frase

> a seguir veio uma mãe com uma criança empurrando
> um carrinho,

pode significar que a mãe empurrava um carrinho com a criança dentro; ou que a mãe vinha com uma criança que brincava com um carrinho. Ambos os sentidos são admissíveis, mas a função sintática das palavras num, e noutro caso, é diferente. No primeiro caso o sujeito que corresponde ao verbo *empurrar* é *mãe*, no segundo caso é *criança*. Assim, teríamos que escolher entre duas árvores gramaticais diferentes para representar esta frase, cada uma correspondendo a um dos sentidos acima.

No caso de uma gramática livre de contexto $\mathcal{G}$, formalizamos esta noção de dupla interpretação na seguinte definição. A gramática $\mathcal{G}$ é *ambígüa* se existe uma palavra $w \in L(\mathcal{G})$ que admite duas árvores de derivação distintas em $\mathcal{G}$.

É bom chamar a atenção para o fato de que nem toda frase com duplo sentido em português se encaixa na definição acima. Por exemplo,

"a galinha está pronta para comer" tem dois significados diferentes, mas em ambos as várias palavras da frase têm a mesma função sintática. Assim, não importa o significado que você dê à frase, o sujeito é sempre 'a galinha'.

Frases que admitem significados diferentes são uma fonte inesgotável de humor; mas para o compilador de uma linguagem de programação uma instrução que admite duas interpretações distintas pode ser um desastre. Voltemos por um momento à gramática $\mathcal{G}_{exp}$. Na figura abaixo, à direita, reproduzimos a árvore de derivação de $\mathrm{id} + \mathrm{id} * \mathrm{id}$ que já havíamos esboçado na seção 2. Uma árvore de derivação diferente para a mesma expressão pode ser encontrada à esquerda, na mesma figura.

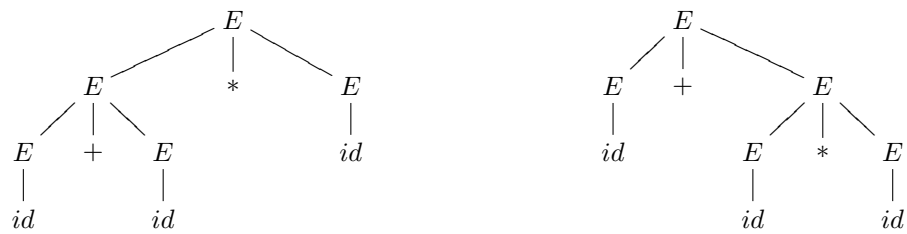


FIGURA 2. Duas árvores e uma mesma colheita

A existência desta segunda árvore de derivação significa que, se um computador estiver usando a gramática $\mathcal{G}_{exp}$, ele não saberá como distinguir a precedência correta entre os operadores $+$ e $*$. Dito de outra maneira, esta gramática não permite determinar que, quando nos deparamos com $\mathrm{id} + \mathrm{id} * \mathrm{id}$ devemos primeiro efetuar a multiplicação e só depois somar o produto obtido com a outra parcela da soma.

Uma saída possível é tentar inventar uma outra gramática que gere a mesma linguagem, mas que não seja ambígüa. A idéia básica consiste em introduzir novas variáveis e novas regras que forcem a precedência correta. Para fazer isto, compare novamente as duas árvores gramaticais com colheita $\mathrm{id} + \mathrm{id} * \mathrm{id}$ da figura 2.

Observe que, na árvore da direita, $\mathrm{id} * \mathrm{id}$ é obtida a partir da expressão $E * E$, e os vértices que correspondem a estes três rótulos são todos irmãos. Por outro lado, o único vértice do qual descendem todos os símbolos da expressão $\mathrm{id} + \mathrm{id} * \mathrm{id}$ é a própria raiz. Portanto, interpretando a expressão $\mathrm{id} + \mathrm{id} * \mathrm{id}$ conforme a árvore da direita, teremos primeiro que calcular o produto, e só depois a soma. Interpretando a mesma expressão de acordo com a árvore da esquerda, vemos que neste caso a adição $\mathrm{id} + \mathrm{id}$ é efetuada antes, e o resultado, então, multiplicado

por id. Isto é, a árvore que dá a interpretação correta da precedência dos operadores é a que está à direita da figura 2.

Assim, precisamos construir uma nova gramática na qual uma árvore como a que está à direita da figura possa ser construída, mas não uma como a da esquerda.

A estratégia consiste em introduzir novas variáveis de modo a forçar a precedência correta dos operadores. Faremos isto deixando a variável E controlar as adições, e criando uma nova variável F (de *fator*) para controlar a multiplicação. O conjunto de regras resultante é o seguinte:

$$\begin{aligned} E &\to E + F \\ E &\to F \\ F &\to F * F \\ F &\to (E) \\ F &\to \mathrm{id} \end{aligned}$$

Não há nada de mais a comentar sobre a primeira e a terceira regras; e a segunda apenas permite passar de somas a multiplicações. A regra que realmente faz a diferença é a quarta. Ela nos diz que, após efetuar uma multiplicação, só podemos voltar à variável E (que controla as somas) colocando os parêntesis.

Embora esta nova gramática não permita a construção de uma árvore como a da esquerda na figura 2, ainda assim ela é ambígüa. Por exemplo, as árvores da figura 3 estão de acordo com a nova gramática mas, apesar de diferentes, têm ambas colheita $\mathrm{id} * \mathrm{id} * \mathrm{id}$.

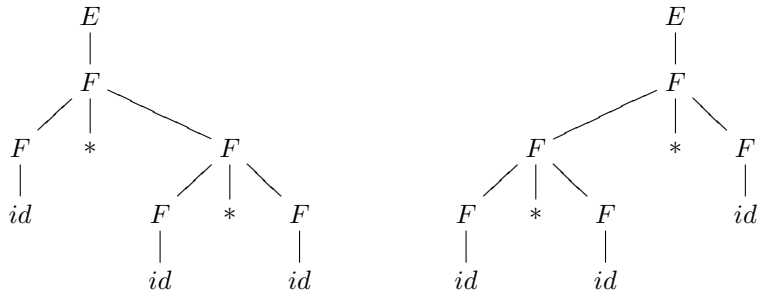


FIGURA 3. Outras duas árvores com mesma colheita

A saída é introduzir mais uma variável, que vamos chamar de T (para *termo*). A nova gramática, que chamaremos de $\mathcal{G}'_{exp}$, tem variáveis

E, T e F, símbolo inicial E, e as seguintes regras:

$$\begin{aligned} E &\to E + T \quad | \quad T \\ T &\to T * F \quad | \quad F \\ F &\to (E) | \quad \text{id} \end{aligned}$$

A gramática $\mathcal{G}'_{exp}$ não é ambígüa, mas provar isto não é fácil, e não faremos os detalhes aqui.

Ainda há um ponto que precisa ser esclarecido. A discussão anterior pode ter deixado a impressão de que, para estabelecer a precedência correta entre os operadores aritméticos, basta eliminar a ambigüidade da gramática. Mas isto não é verdade. Por exemplo, a gramática $\mathcal{G}"_{exp}$ cujas regras são:

$$\begin{aligned} S &\to E \\ E &\to (E)R \quad | \quad VR \\ R &\to +E \quad | \quad *E \quad | \quad A \\ V &\to \text{id} \\ A &\to \epsilon \end{aligned}$$

gera L_{exp}. Além disso, não é difícil provar que esta gramática não pode ser ambígüa. Isto decorre dos seguintes fatos:

- não há mais de duas regras em $\mathcal{G}"_{exp}$ cujo lado esquerdo seja ocupado por uma mesma variável;
- se há duas regras a partir de uma mesma variável, o lado direito de uma começa com uma variável, e o da outra com um terminal;
- cada terminal só aparece uma vez como prefixo do lado direito de alguma regra de $\mathcal{G}"_{exp}$.

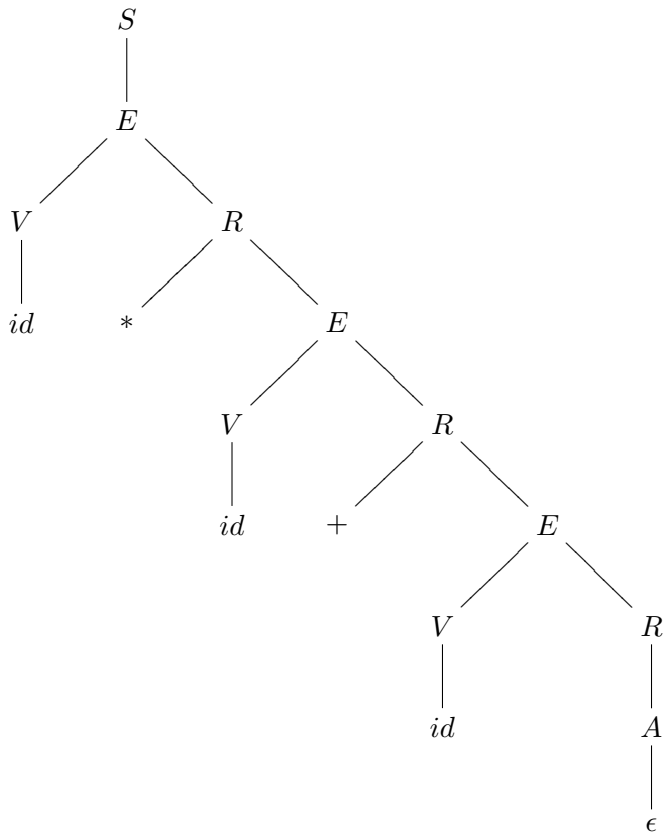
Imagine, então, que estamos tentando calcular uma derivação à esquerda em $\mathcal{G}"_{exp}$ para uma dada palavra $w \in L_{exp}$. Digamos que, isolando o n-ésimo passo da derivação, obtivemos o seguinte

$$S \Rightarrow^* uEv \Rightarrow w,$$

onde u só contém terminais, mas v pode conter terminais e variáveis. Precisamos decidir qual a regra a ser aplicada a seguir. Neste exemplo, a variável mais à esquerda no n-ésimo passo da derivação é E. Assim, há duas regras que podemos aplicar. Para saber qual das duas será escolhida voltamos nossa atenção para a palavra w que está sendo derivada. É claro que w tem u como prefixo; digamos que o terminal seguinte a u em w seja σ. Temos então duas possibilidades. Se $\sigma =$ (então a regra

a ser aplicada tem que ser $E \to (E)R$. Por outro lado, se $\sigma \neq ($ então só nos resta a possibilidade de aplicar $E \to VR$. Assim, a regra a ser aplicada neste passo está completamente determinada pela variável mais à esquerda presente, e pela seqüência de terminais da palavra que está sendo derivada.

Entretanto, apesar de $\mathcal{G}"_{exp}$ não ser ambígüa, a árvore de derivação de $\mathrm{id} + \mathrm{id} * \mathrm{id}$ em $\mathcal{G}"_{exp}$ não apresenta a precedência desejada entre os operadores, como mostra a figura abaixo.



Voltaremos a discutir $\mathcal{G}"_{exp}$ quando tratarmos de autômatos de pilha determinísticos.

6. Removendo ambigüidade

O que fizemos na seção anterior parece indicar que, ao se deparar com uma gramática ambígüa, tudo o que temos que fazer é adicionar algumas variáveis e alterar um pouco as regras de maneira a remover a

ambigüidade. É verdade que criar novas variáveis e regras para remover ambigüidade pode não ser muito fácil, e é bom não esquecer que não chegamos a provar que a gramática $\mathcal{G}'_{exp}$ não é ambígüa.

Por outro lado, isto parece bem o tipo de problema que poderia ser deixado para um computador fazer, bastaria que descobríssemos o algoritmo. É aí justamente que está o problema. Um computador não consegue sequer detectar que uma linguagem é ambígüa. De fato:

> não pode existir um algoritmo para determinar se uma dada linguagem livre de contexto é ou não ambígüa.

Voltaremos a esta questão em um capítulo posterior.

Infelizmente, as más notícias não acabam aí. Há linguagens livres de contexto que não podem ser geradas por nenhuma gramática livre de contexto que não seja ambígüa. Tais linguagens são chamadas de *inerentemente ambígüas*. Note que ambigüidade é uma propriedade da gramática, mas ambigüidade inerente é uma propriedade da própria linguagem.

Um exemplo bastante simples de linguagem inerentemente ambígüa é

$$L_{in} = \{a^i b^j c^k : i, j, k \geq 0 \text{ e } \ i = j \text{ ou } \ j = k\}.$$

Com isso você descobre porque demos este nome a L_{in}. Não é fácil mostrar que esta linguagem é inerentemente ambígüa, e por isso não vamos fazer a demonstração aqui. Os detalhes podem ser encontrados em [**1**, theorem 7.2.2, p. 236] ou [**2**, theorem 4.7, p. 100].

Para não encerrar o capítulo num clima pessimista, vamos analisar o problema da ambigüidade para o caso particular das linguagens regulares. Sabemos que estas linguagens podem ser geradas por gramáticas livres de contexto de um tipo bastante especial: as gramáticas lineares à direita. Assim a primeira pergunta é: uma gramática linear à direita pode ser ambígüa? A resposta é sim. Por exemplo, considere a gramática com terminal $\{0\}$, que tem S como única variável, e cujas regras são

$$S \to 0 \quad | \quad 0S \quad | \quad 0^2.$$

A palavra 0^2 tem duas árvores gramaticais distintas que estão esboçadas na abaixo.

Felizmente, no caso de gramáticas lineares à direita é sempre possível remover a ambigüidade. Em outras palavras, não existem linguagens regulares inerentemente ambígüas. Além disso, existe um algoritmo simples que, tendo como entrada uma gramática linear à direita, constrói uma outra que gera a mesma linguagem regular mas não é ambígüa. O algoritmo é conseqüência do seguinte fato:

> a gramática linear à direita construída a partir de um autômato finito determinístico pelo algoritmo do capítulo 8 *não* é ambígüa.

Para entender porque isto é verdade, suponha que M seja um autômato finito determinístico e $\mathcal{G}$ seja a gramática construída a partir de M pelo algoritmo do capítulo ??. Vimos que as derivações em $\mathcal{G}$ simulam computações em M e vice-versa. Como M é determinístico, só há uma computação possível para cada palavra de $L(M)$. Logo cada palavra de $L(\mathcal{G}) = L(M)$ só tem uma derivação possível. Como $\mathcal{G}$ é linear à direita, toda derivação em $\mathcal{G}$ é mais à esquerda. Logo $\mathcal{G}$ não é ambígüa pelo teorema da seção 4.

Diante deste resultado, é claro que, ao receber uma gramática linear à direita $\mathcal{G}$ como entrada, o algoritmo procede da seguinte maneira:

Etapa 1: determina um autômato finito não determinístico M tal que $L(M) = L(\mathcal{G})$;

Etapa 2: determina um autômato finito determinístico M' tal que $L(M) = L(M')$;

Etapa 3: determina uma gramática $\mathcal{G}'$, obtida a partir de M', e que gera $L(M')$.

Como já vimos, $\mathcal{G}'$ não pode ser uma gramática ambígüa.

7. Exercícios

1. Prove, por indução no número de vértices internos, que a colheita de uma árvore gramatical T é igual à palavra obtida concatenando-se os rótulos das folhas de T da esquerda para a direita.

2. Seja $\mathcal{G}$ uma gramática livre de contexto e seja X uma variável de $\mathcal{G}$. Seja w uma palavra somente em terminais e que pode ser derivada em $\mathcal{G}$ a partir de X. Prove, por indução no número de passos de uma derivação de w a partir de X que existe uma X-árvore em $\mathcal{G}$ cuja colheita é w.

3. Considere a gramática não ambígua G'_{exp} que gera as expressões aritméticas.

(a) Esboce as árvores de derivação de $\mathrm{id} + (\mathrm{id} + \mathrm{id}) * \mathrm{id}$ e de $(\mathrm{id} * \mathrm{id} + \mathrm{id} * \mathrm{id})$
(b) Dê uma derivação à esquerda e uma derivação à direita da expressão $(\mathrm{id} * \mathrm{id} + \mathrm{id} * \mathrm{id})$.

4. Repita o exercício anterior para a gramática $\mathcal{G}"_{exp}$.

5. Descreva detalhadamente um algoritmo que, tendo como entrada a derivação de uma palavra w em uma gramática livre de contexto, constrói uma árvore gramatical cuja colheita é w. O principal problema consiste em, tendo dois passos consecutivos da derivação, determinar qual a regra que foi aplicada.

6. Prove que a gramática $\mathcal{G}"_{exp}$ não é ambígüa.
SUGESTÃO: Use indução no número de passos de uma derivação à esquerda.

7. Mostre que a gramática cujas regras são

$$S \to 1A \,|\, 0B$$
$$A \to 0 \,|\, 0S \,|\, 1AA$$
$$B \to 1 \,|\, 1S \,|\, 0BB$$

é ambígua.

8. Seja G a gramática linear à direita com terminais $\{0, 1\}$, variáveis $\{\}$ e símbolo inicial S, cujas regras são dadas por

$$S \to 0X$$
$$X \to 10Y \,|\, 1Z$$
$$Z \to 01W \,|\, 1$$
$$Y \to 1 \,|0$$
$$W \to \epsilon$$

(a) Mostre que G é ambígüa.
(b) Use o algoritmo descrito em 6 para construir uma gramática linear à direita G' não ambígüa que gere $L(G)$.

CAPíTULO 11

Linguagens que não são livres de contexto

Neste capítulo, finalmente, confrontamos a inevitável pergunta: como provar que uma dada linguagem não é livre de contexto? A estratégia é muito semelhante à adotada para linguagens regulares, apesar do lema do bombeamento resultante ser um pouco mais difícil de aplicar na prática.

1. Introdução

Já conhecemos muitas linguagens livres de contexto, mas ainda não temos nenhum exemplo de uma linguagem que não seja deste tipo. Quer dizer, já dissemos no capítulo 9 que a linguagem

$$L_{abc} = \{a^n b^n c^n : n \geq 0\}$$

não é livre de contexto, mais ainda não provamos isto. É claro que isto não é satisfatório. O problema é que nada podemos concluir do simples fato de não termos sido capazes de inventar uma gramática livre de contexto que gere esta linguagem. Talvez a gramática seja muito complicada, ou quem sabe foi só falta de inspiração.

Mas como será possível provar que não existe nenhuma gramática livre de contexto que gere uma dada linguagem? A estratégia é a mesma adotada para o caso de linguagens regulares. Isto é, provaremos que toda linguagem livre de contexto satisfaz uma propriedade de bombeamento. Portanto, uma linguagem que *não* satisfizer esta propriedade *não* pode ser livre de contexto. Começamos com um resultado relativo a árvores que será necessário na demonstração do lema do bombeamento.

Como no capítulo anterior, consideraremos apenas árvores enraizadas. Dizemos que uma árvore é m*-ária* se cada vértice tem, no máximo,

m filhos. Desejamos relacionar o número de folhas de uma árvore m-ária com a altura desta árvore. Lembre-se que a *altura* de uma árvore é o mais longo caminho entre a raiz e alguma de suas folhas.

Quando todos os vértices internos da árvore têm exatamente m-filhos, a árvore m-ária é *completa*. Seja $f(h)$ o número de folhas de uma árvore m-ária completa de altura h. Como a árvore m-ária completa é aquela que tem o maior número possível de folhas, o problema estará resolvido se formos capazes de encontrar uma fórmula para $f(h)$ em função de h e m. Faremos isto determinando uma relação de recorrência para $f(h)$ e resolvendo-a.

Para começar, se a árvore tem altura zero, então consiste apenas de um vértice. Neste caso há apenas uma folha, de modo que $f(0) = 1$. Para estabelecer a relação de recorrência podemos imaginar que $\mathcal{T}$ é uma árvore m-ária completa de altura h. É claro que, removendo todas as folhas de $\mathcal{T}$, obtemos uma árvore m-ária completa $\mathcal{T}'$ de altura $h-1$. Para reconstruir $\mathcal{T}$ a partir de $\mathcal{T}'$ precisamos repor as folhas. Fazemos isso dando m-filhos a cada folha de $\mathcal{T}'$. Como $\mathcal{T}'$ tem $f(h-1)$ folhas, obtemos

$$f(h) = mf(h-1).$$

Assim,

$$f(h) = mf(h-1) = m^2 f(h-2) = \cdots = m^h f(0) = m^h.$$

Portanto, $f(h) = m^h$ é a fórmula desejada.

Para estabelecer a relação entre este resultado e o lema do bombeamento precisamos de uma definição. Seja $\mathcal{G}$ uma linguagem livre de contexto. A *amplitude* $\alpha(\mathcal{G})$ de uma gramática livre de contexto $\mathcal{G}$ é o comprimento máximo das palavras que aparecem à direita de uma seta em uma regra de $\mathcal{G}$. Por exemplo, para as gramáticas $\mathcal{G}_{exp}$ e $\mathcal{G}''_{exp}$ definidas no capítulo anterior, temos $\alpha(\mathcal{G}_{exp}) = 3$ e $\alpha(\mathcal{G}''_{exp}) = 4$.

Se uma gramática livre de contexto tem amplitude α, então todas as suas árvores gramaticais são α-árias. A fórmula para o número de folhas de uma árvore α-ária completa nos dá então seguinte lema.

LEMA 11.1. *Seja $\mathcal{G}$ uma linguagem livre de contexto. Se X é uma variável de $\mathcal{G}$ e se w é colheita de uma X-árvore de $\mathcal{G}$ de altura h então*

$$|w| \leq \alpha(\mathcal{G})^h.$$

2. Lema do bombeamento

Antes de enunciar e provar o lema do bombeamento de maneira formal, vamos considerar o seu funcionamento de maneira informal.

Suponhamos que $\mathcal{G}$ é uma gramática livre de contexto que gera uma linguagem infinita. Segundo o lema da seção 1, quanto maior o comprimento da colheita de uma árvore gramatical maior tem que ser a sua altura. Portanto, se o comprimento da colheita de uma árvore de derivação $\mathcal{T}$ é suficientemente grande, então o caminho mais longo entre a raiz e alguma folha conterá mais vértices interiores do que há variáveis na gramática. Em particular, haverá dois vértices diferentes em $\mathcal{T}$ cujos rótulos são iguais. Vamos chamar de ν_1 e ν_2 estes vértices, e de A a variável que os rotula, como na figura 1.

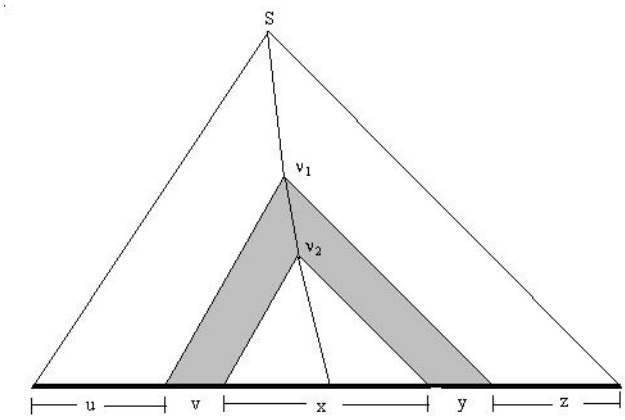


FIGURA 1. Decompondo a palavra para bombear

Observe que as regras associadas a ν_1 e ν_2 têm ambas A do seu lado esquerdo. Mas isto significa que podemos construir a partir de $\mathcal{T}$ uma nova árvore $\mathcal{T}'$ da seguinte maneira. Comece construindo $\mathcal{T}'$ exatamente como $\mathcal{T}$ até chegar a ν_2. Como ν_2 é rotulado pela mesma variável que ν_1, podemos construir a partir dele a mesma A-árvore que estava associada a ν_1 em $\mathcal{T}$, como mostra a figura 2. Note que os trechos da colheita da árvore $\mathcal{T}$ da figura 1 marcados como v e y aparecem repetidos em $\mathcal{T}'$. Como $\mathcal{T}'$ é uma árvore de derivação em $\mathcal{G}$, sua colheita é um elemento de $L(\mathcal{G})$ no qual os trechos x e y de $c(\mathcal{T})$ aparecem bombeados. Naturalmente o processo acima pode ser repetido quantas vezes quisermos, de modo que podemos bombear estes trechos qualquer número de vezes.

Para que esta propriedade do bombeamento possa ser usada para provar que uma linguagem não é livre de contexto precisamos formulá-la de maneira mais precisa. Além disso, como no caso do lema correspondente para linguagens regulares, incluiremos algumas condições

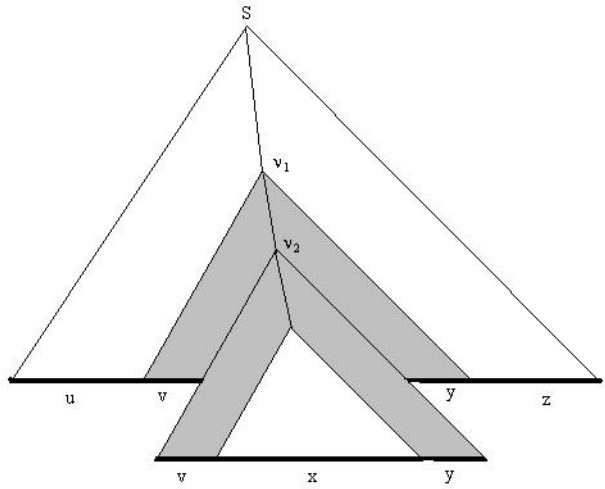


FIGURA 2. Bombeando uma vez

técnicas extras que reduzem o número de possíveis decomposições da palavra a ser bombeada que precisamos considerar. De fato, há várias versões diferentes do lema do bombeamento para linguagens livres de contexto. A que apresentamos aqui não é a mais forte, mas é suficiente para cobrir muitos dos exemplos mais simples. Versões mais sofisticadas são discutidas em [**2**, Chapter 6, p. 125].

LEMA DO BOMBEAMENTO. *Seja $\mathcal{G}$ uma gramática livre de contexto. Existe um número inteiro ρ, que depende de $\mathcal{G}$, tal que, se $w \in L(\mathcal{G})$ e $|w| \geq \rho$, então existe uma decomposição de w na forma $w = uvxyz$, onde*

(1) $vy \neq \epsilon$;
(2) $|vxy| \leq \rho$;
(3) $uv^n xy^n z \in L(\mathcal{G})$, *para todo* $n \geq 0$.

DEMONSTRAÇÃO. Suponhamos que $\mathcal{G}$ tem k variáveis; neste caso escolheremos $\rho = \alpha(\mathcal{G})^{k+1}$. Seja $w \in L(\mathcal{G})$ uma palavra com comprimento maior que ρ. Já sabemos, pelo teorema da seção 4 do capítulo 10, que têm que existir árvores de derivação com colheita w. Entre todas estas árvores escolha uma, que chamaremos de $\mathcal{T}$, que satisfaça a seguinte propriedade:

Hipótese 1: $\mathcal{T}$ tem o menor número possível de folhas entre todas as árvores de derivação de colheita w em $\mathcal{G}$.

Como a colheita de $\mathcal{T}$ tem comprimento maior que $\rho = \alpha(\mathcal{G})^{k+1}$, segue do lema da seção 1 que $\mathcal{T}$ tem altura pelo menos $k+1$. Chamando de C o mais longo caminho em $\mathcal{T}$ que vai de sua raiz a uma folha, concluímos que C tem, pelo menos, $k+1$ arestas. Logo C tem, no mínimo, $k+2$ vértices. Como só pode haver uma folha num tal caminho, então C tem $k+1$ vértices interiores. Contudo k é o número de variáveis da gramática, e cada vértice de C está rotulado por uma variável. Portanto, pelo princípio da casa do pombo, há dois vértices *diferentes* em C rotulados pela mesma variável. Entre todos os pares de vértices de C rotulados pela mesma variável, escolha aquele que satisfaz a seguinte propriedade:

> **Hipótese 2:** o vértice ν_1 precede o vértice ν_2 e todos os vértices de C entre ν_1 e a folha são rotulados por variáveis distintas.

Lembre-se que as árvores que estamos considerando são grafos orientados. Portanto, C é um caminho orientado; logo faz sentido dizer, de dois vértices de C, que um precede o outro.

Seja A a variável que rotula ν_1 e ν_2. Temos então duas A-árvores: $\mathcal{T}_1$, com raiz em ν_1, e $\mathcal{T}_2$ com raiz em ν_2. Denotaremos por x a colheita de $\mathcal{T}_2$. Observe que, como ν_1 precede ν_2 ao longo de C, então x é uma subpalavra da colheita de $\mathcal{T}_1$. Assim, podemos decompor a colheita de $\mathcal{T}_1$ na forma vxy. A relação entre estas árvores e suas colheitas é ilustrada na figura 1.

Entretanto, pela hipótese 2, o caminho (ao longo de C) que vai de ν_1 à folha tem, no máximo, $k+2$ vértices (contando com a folha, que é rotulada por um terminal!). Além disso, como C é o mais longo caminho que vai da raiz de $\mathcal{T}$ a uma folha, o trecho de C que começa em ν_1 é o mais longo caminho em $\mathcal{T}_1$ entre sua raiz (que é ν_1) e uma folha. Portanto, $\mathcal{T}_1$ tem altura no máximo $k+1$. Concluímos, utilizando novamente o lema da seção 1 que, como vxy é a colheita de $\mathcal{T}_1$, então

$$|vxy| \leq \alpha(\mathcal{G})^{k+1} = \rho.$$

Isto prova (2) do enunciado do lema.

Considere agora o que acontece na construção de $\mathcal{T}$ quando chegamos a ν_2. Este vértice é rotulado pela variável A e a ele está associada uma regra que tem A do lado esquerdo da seta. Mas suponha que, chegados a ν_2, decidimos aplicar a mesma regra que aplicamos quando chegamos a ν_1. Podemos fazer isto porque esta também é uma regra que tem A do lado esquerdo. Se continuarmos, vértice a vértice, copiando o trecho da árvore $\mathcal{T}$ hachurado na figura, teremos uma nova árvore gramatical em $\mathcal{G}$, cuja colheita é uv^2xy^2z. Se repetirmos este procedimento

n vezes, obteremos uma árvore cuja colheita é uv^nxy^nz. Isto prova (3) quando $n > 0$.

Por outro lado, ao chegar ao vértice ν_1, também podemos usar a regra associada a ν_2 e continuar a construir a árvore como se fosse $\mathcal{T}_2$. Neste caso obteremos uma árvore $\mathcal{T}_0$ com menos vértices que $\mathcal{T}$ e com colheita uxz, que corresponde a tomar $n = 0$ em (3).

Só nos resta mostrar que $vy \neq \epsilon$. Mas se vy fosse igual a ϵ então a árvore $\mathcal{T}_0$, construída no parágrafo anterior, teria colheita igual a w, e menos folhas que $\mathcal{T}$, o que contradiz a hipótese 1. Portanto, $vy \neq \epsilon$ e provamos (1), concluindo assim a demonstração do lema do bombeamento.

3. Exemplos

Veremos a seguir que várias das linguagens que já encontramos anteriormente, e outras que ainda vamos encontrar à frente, não são livres de contexto. Antes, porém, precisamos discutir como o lema do bombeamento é utilizado para provar que uma linguagem não é livre de contexto.

Digamos que L é uma linguagem que você suspeita não ser livre de contexto. Para aplicar o lema do bombeamento a L usamos a mesma estratégia já utilizada no caso de linguagens regulares. Assim,

(1) suporemos, por contradição, que L é gerada por uma gramática livre de contexto;
(2) escolheremos uma palavra $w \in L$ de comprimento maior que ρ;
(3) mostraremos que não há *nenhuma maneira possível* de decompor w na forma do lema do bombeamento, de modo que w tenha subpalavras bombeáveis.

Com isto podemos concluir que L não é livre de contexto.

É claro que, se o seu palpite estiver errado e L for livre de contexto, então você não chegará a nenhuma contradição. Por outro lado, o fato de uma contradição não ter sido obtida *não* significa que L é livre de contexto.

Como no caso das linguagens regulares, a escolha da palavra em (2) depende de ρ, uma variável inteira positiva. Além disso, escolher w de maneira a obter facilmente uma contradição envolve uma certa dose de tentativa e erro. Finalmente, por causa da maneira mais complicada de decompor w, podemos ter vários casos a analisar antes de esgotar todas as possibilidades. Vejamos alguns exemplos.

Exemplo 1. Já havíamos dito no capítulo 7 que a linguagem

$$L_{abc} = \{a^n b^n c^n : n \geq 0\}$$

não é livre de contexto. Temos, agora, as ferramentas necessárias para provar que isto é verdade.

Suponha, por contradição, que L_{abc} é livre de contexto. Pelo lema do bombeamento existe um inteiro positivo ρ tal que, se $n > \rho$, então é possível decompor $w = a^n b^n c^n$ na forma $w = uvxyz$, onde:

(1) $vy \neq \epsilon$;
(2) $|vxy| \leq \rho$;
(3) $uv^n xy^n z \in L_{abc}$ para todo $n \geq 0$.

Como $n > \rho$ mas $|vxy| \leq \rho$, temos que vxy não pode conter, ao mesmo tempo, as, bs e cs. Digamos que vxy só contenha as e bs. Neste caso, nem v, nem y, podem conter c, mas vy tem que conter pelo menos um a ou um b. Assim, quando $n > 1$ o número de as ou bs em $uv^n xy^n z$ tem que ser maior que n, ao passo que o número de cs não foi alterado, e continua sendo n. O caso em que vxy só contém bs e cs pode ser tratado de maneira análoga. Portanto, $uv^n xy^n z \notin L_{abc}$ o que contradiz o lema do bombeamento. Concluímos que, de fato, L não é uma linguagem livre de contexto.

Exemplo 2. No capítulo 3 vimos que a linguagem

$$L_{primos} = \{0^p : p \text{ é um primo positivo}\},$$

não é regular. Como já sabemos que há linguagens livres de contexto que não são regulares, faz sentido perguntar se esta linguagem é livre de contexto. A resposta é não.

Suponhamos, por contradição, que L_{primos} seja livre de contexto. Então, de acordo com o lema do bombeamento, deve existir um inteiro positivo ρ tal que se p é um primo maior que ρ, então podemos decompor 0^p na forma $0^p = uvxyz$, onde

(1) $vy \neq \epsilon$;
(2) $|vxy| \leq \rho$;
(3) $uv^n xy^n z \in L_{primos}$ para todo $n \geq 0$.

Mas u, v, x, y e z são todas palavras em 0^*. Logo existem inteiros $m, r, s, t \geq 0$ tais que

$$u = 0^m,\ v = 0^r,\ x = 0^s,\ y = 0^t \text{ e } z = 0^{p-r-s-t}.$$

Além disso, segue de (1) que $r + t > 0$, e de (3) que

$$uv^n xy^n z = 0^m (0^r)^n 0^s (0^t)^n 0^{p-r-s-t} = 0^{p+(r+t)(n-1)}$$

pertence a L_{primos} para todo $n > 0$. Mas, para que esta última afirmação seja verdadeira, os números $p + (r + t)(n - 1)$ têm que ser primos para todo $n \geq 0$. Tomando $n = p + 1$ temos que

$$p + (r + t)(n - 1) = p(1 + r + t),$$

é composto, pois $r + t > 0$; uma contradição. Portanto, L_{primos} não é uma linguagem livre de contexto.

Exemplo 3. Considere, agora, a linguagem

$$L_{rr} = \{rr : s \in (0 \cup 1)^*\}.$$

Já vimos no capítulo 3 que esta linguagem não é regular, queremos provar que também não é livre de contexto.

Suponhamos, por contradição, que L_{rr} é livre de contexto. Aproveitando uma idéia já usada no capítulo 3, escolhemos $w = 0^m 10^m 1$ como nossa primeira tentativa. Entretanto, escolhendo $\rho = 3$ e decompondo $0^m 10^m 1$ na forma

$$u = 0^{m-1}, v = 0, x = 1, y = 0 \quad \text{e} \quad z = 0^{m-1}1,$$

verificamos que todas as condições do lema do bombeamento são satisfeitas. Portanto, não é possível chegar a uma contradição escolhendo $r = 0^m 1$.

A saída é escolher uma palavra mais complexa. Observe que, no exemplo acima, a decomposição proposta não funcionaria se o número de 1s também dependesse de m. Isto sugere que devemos escolher $r = 0^m 1^m$.

Supondo, por contradição, que L_{rr} é livre de contexto e escolhendo $w = 0^m 1^m 0^m 1^m$ temos, pelo lema do bombeamento, que se $m \geq \rho$ então w pode ser decomposta na forma

$$0^m 1^m 0^m 1^m = uvxyz$$

de modo que as condições (1), (2) e (3) do lema do bombeamento sejam satisfeitas. Há dois casos a considerar.

O primeiro caso consiste em supor que vxy é uma subpalavra do primeiro $0^m 1^m$. Escrevendo $0^m 1^m = uvxyz'$, onde $z = z'0^m 1^m$, temos que

$$2m < |uv^2xy^2z'| \leq 2m + \rho.$$

Isto significa que yz' foi deslocado $|vxy|$ casas para à direita. Contudo,

$$|vxy| \leq \rho \leq m$$

e vxy^2z' acaba em m uns. Como o segundo $0^m 1^m$ não foi bombeado, temos que o símbolo seguinte ao meio da palavra é 1. Em particular, se $uv^2xy^2z = rr$ então o segundo r começa com 1 e o primeiro com 0, o

que é uma contradição. O caso em que vxy é subspalavra do segundo 0^m1^m pode ser tratado de maneira análoga.

Finalmente, resta-nos supor que vxy inclui o meio da palavra. Neste caso, vxy tem que ser subpalavra de 1^m0^m, já que $|vxy| \leq \rho \leq m$. Assim, ou v contém algum 1 ou y algum 0. Removendo-os, concluímos que $uxz = 0^m1^s0^t1^m$, onde s ou t (ou ambos) são menores que m. Entretanto, se $0^m1^s0^t1^m = rr$ então r começa em 0 e acaba em 1, de forma que precisamos ter $s = m = t$, o que é uma contradição. Concluímos que L_{rr} não é livre de contexto.

Exemplo 4. Vimos na seção 4 do capítulo 7 que a união, concatenação e estrela de linguagens livres de contexto também é livre de contexto. Resta-nos analisar o que acontece com a intersecção e o complemento de linguagens livres de contexto.

Considere, por exemplo as linguagens

$$L_1 = \{a^ib^jc^k : i, j, k \geq 0 \text{ e } i = j\}, e$$
$$L_2 = \{a^ib^jc^k : i, j, k \geq 0 \text{ e } j = k\}.$$

Já vimos no capítulo 7 que estas duas linguagens são livres de contexto. Entretanto,

$$L_1 \cap L_2 = \{a^ib^jc^k : i = j \text{ e } j = k\}.$$

Em outras palavras, $L_1 \cap L_2 = L_{abc}$. Porém, mostramos no exemplo 1 acima que esta linguagem não é livre de contexto. Assim, a intersecção de duas linguagens livres de contexto pode não ser livre de contexto.

E quanto ao complemento? Já sabemos do capítulo 5, que existe uma maneira de relacionar o complemento e a intersecção de duas linguagens através da união. Como a intersecção de linguagens livres de contexto não é necessariamente livre de contexto, seria de esperar que o mesmo valesse para o complemento. Provaremos isto argumentando por contradição.

Suponhamos, por contradição, que o complemento de linguagens livres de contexto seja livre de contexto. Como L_1 e L_2 são livres de contexto, então $\overline{L_1}$ e $\overline{L_2}$ também serão. Mas a união de linguagens livres de contexto é livre de contexto. Portanto, $\overline{L_1} \cup \overline{L_2}$ é livre de contexto. Usando mais uma vez a hipótese sobre o complemento, concluímos que

$$\overline{(\overline{L_1} \cup \overline{L_2})} = L_1 \cap L_2,$$

é livre de contexto. Entretanto já vimos que isto é falso neste caso. Portanto, o complemento de linguagens livres de contexto pode não ser livre de contexto.

Note que nossa conclusão diz apenas que a intersecção e o complemento de linguagens livres de contexto *pode não ser* livre de contexto. Isto porque, em muitos casos, estas operações produzem linguagens livres de contexto. Por exemplo, toda linguagem regular é livre de contexto; e já vimos que a intersecção e o complemento de linguagens regulares é regular. Portanto, neste caso estamos intersectando ou tomando o complementar de linguagens livres de contexto e obtendo como resultado uma linguagem do mesmo tipo. É possível aperfeiçoar este resultado, e mostrar que a intersecção de uma linguagem livre de contexto com uma linguagem *regular* é sempre livre de contexto. Mas, para fazer isto, precisamos usar os autômatos de pilha que serão introduzidos no próximo capítulo.

4. Exercícios

1. Mostre que nenhuma das linguagens abaixo é livre de contexto usando o lema do bombeamento.
   (a) $\{a^{2^n} : n \text{ é primo}\}$;
   (b) $\{a^{n^2} : n \geq 0\}$;
   (c) $\{a^n b^n c^r : r \geq n\}$;
   (d) o conjunto das palavras em $\{a, b, c\}^*$ que têm o mesmo número de as e bs, e cujo número de cs é maior ou igual que o de as;
   (e) $\{0^n 1^n 0^n 1^n : n \geq 0\}$;
   (f) $\{rrr : s \in (0 \cup 1)^*\}$;
   (g) $\{wcwcw : w \in \{0, 1\}^*\}$;
   (h) $\{0^{n!} : n \geq 1\}$;
   (i) $\{0^k 1^k 0^k : k \geq 0\}$;

CAPíTULO 12

Autômatos de Pilha

Neste capítulo começamos a estudar a classe de autômatos que aceita as linguagens livres de contexto: os autômatos de pilha não determinísticos. Ao contrário dos autômatos finitos, os autômatos de pilha têm uma memória infinita. Contudo o acesso a esta memória é feito de maneira extremamente restrita, uma vez que o último item que foi posto na memória é obrigatoriamente o primeiro a ser consultado.

1. Heurística

Suponhamos que L é uma linguagem livre de contexto em um alfabeto Σ. Nesta seção consideramos como construir um procedimento que, tendo como entrada uma palavra w no alfabeto Σ, determina se w pertence ou não a L. Como sempre, assumiremos que w é lida, um símbolo de cada vez, da esquerda para a direita.

Nossos procedimentos utilizarão uma memória infinita, em forma de pilha. Para tornar o problema mais concreto, podemos imaginar que esta memória é constituída por discos perfurados ao meio, que são empilhados em uma haste. Os discos vêm em várias cores e temos um estoque infinito deles. Além disso, a haste pode ser feita tão longa quanto for necessário.

Para 'lembrar' alguma coisa, enfiamos discos coloridos na haste. Como os discos estão trespassados pela haste, só é possível removê-los um a um, começando sempre pelo que está mais acima.

Nosso primeiro exemplo é a linguagem L_1 formada pelas palavras no alfabeto $\{a, b, c\}$ que são da forma vcv^R, onde v é uma palavra qualquer nos as e bs. Por exemplo, as palavras $abaacaaba$ e $bbacabb$ pertencem a L_1, mas $abcab$ e $abba$ não pertencem. É fácil mostrar que esta

linguagem não é regular usando o lema do bombeamento; portanto, não existe nenhum autômato finito que a aceite.

Para poder construir um procedimento que verifica se uma dada palavra de $\{a, b, c\}$ pertence ou não a L_1 precisamos ter uma maneira de 'lembrar' exatamente qual é a seqüência de as e bs que apareceu antes do c. Comparamos, então, esta seqüência com a que vem depois do c.

Faremos isto usando uma pilha e duas cores diferentes de discos: preto e branco. Procedemos da seguinte forma:

Etapa 1: Se achar a empilhe um disco preto na haste, e se achar b, um disco branco.

Etapa 2: Se achar c, mude de atitude e prepare-se para desempilhar discos.

Etapa 3: Compare o símbolo que está sendo lido com o que está no topo da pilha: se lê a e o topo da pilha é ocupado por um disco preto, ou se lê b e o topo é ocupado por um disco branco, desempilhe o disco.

Este procedimento obedece ainda a uma instrução que não foi explicitada acima, e que diz: se em alguma situação nenhuma das etapas acima puder ser aplicada, então páre de executar o procedimento.

Por exemplo, se a palavra dada for $w = abcba$, o procedimento se comporta da seguinte maneira:

Acha	Faz	Pilha	Resta na entrada
a	empilha disco preto	•	$bcba$
b	empilha disco branco	○ •	cba
c	muda de atitude	○ •	ba
b	compara e desempilha	•	a
a	compara e desempilha		

Observe que a pilha registra os símbolos de w que antecedem o c *de baixo para cima*. Os discos da pilha, por sua vez, são comparados, *de cima para baixo*, com a parte da palavra que sucede o c. Assim, o reflexo da parte da palavra que antecede o c é comparado com a parte da palavra que sucede o c. Portanto, se $w \in L_1$, então todos os símbolos de w devem ter sido consumidos e a pilha deve estar vazia quando o procedimento parar.

Por outro lado, se a palavra não está em L_1 então podem acontecer duas coisas. A primeira é que a entrada não possa ser totalmente

consumida por falta de instruções adequadas. Isto ocorre, por exemplo, quando a entrada é cab. A segunda possibilidade é que a entrada seja totalmente consumida, mas a pilha não se esvazie. Este é o caso, por exemplo, da entrada a^2bcba. Concluímos que a entrada deverá ser aceita se, e somente se, ao final da execução do procedimento ela foi totalmente consumida e a pilha está vazia.

A instrução de parar nos casos omissos faz com que o procedimento descrito acima seja completamente determinístico. Entretanto, nem sempre é possível criar um procedimento determinístico deste tipo para testar se uma palavra pertence a uma dada linguagem livre de contexto. Por exemplo, a linguagem

$$L_2 = \{vv^R : v \in \{a, b\}^*\},$$

no alfabeto $\{a, b\}$, é livre de contexto.

A única diferença desta linguagem para L_1 é que não há um símbolo especial marcando o meio da palavra. Portanto, se descobrirmos como identificar o meio da palavra, o resto do procedimento pode ser igual ao anterior. A saída é deixar por conta de quem está executando o procedimento o ônus de advinhar onde está o meio da palavra. Como somente um símbolo da palavra é visto de cada vez, a decisão acaba tendo que ser aleatória. Portanto, tudo o que precisamos fazer é alterar a etapa 2, que passará a ser:

> **Nova etapa 2:** decida (aleatoriamente) se quer mudar de atitude e passar a desempilhar os discos.

Observe que, se o procedimento determinar que a palavra dada pertence a L_2, então podemos estar seguros de que isto é verdade. Entretanto, como o procedimento não é determinístico, uma saída negativa não garante que a palavra não pertence a L_2. Podemos apenas ter sido infelizes na nossa escolha de onde estaria o meio da palavra.

2. Definição e exemplos

Vamos analisar os elementos utilizados na construção destes procedimentos e, a partir deles, sistematizar nossa definição de autômato de pilha.

Os elementos mais óbvios são: o alfabeto de entrada e a pilha. Entretanto, em ambos os exemplos temos mudanças de atitude de *empilha* para *compara e desempilha*. Como no caso de autômatos finitos, a atitude dos autômatos de pilha serão codificadas nos estados. Com isto precisamos também de um estado inicial que indica qual é a primeira

etapa do procedimento. Finalmente, precisamos de uma função de transição que nos diz o que fazer com a entrada (e a pilha!), e que seja flexível o suficiente para codificar procedimentos não determinísticos.

Comparando a análise acima com a definição de autômato finito, verificamos que não foi necessário mencionar estados finais. Afinal, a aceitação de uma entrada pelos nossos procedimentos foi determinada pelo fato da entrada ter sido totalmente consumida e pelo esvaziamento da pilha. Apesar disso, introduziremos a noção de estado final em nossa definição formal, porque isto nos dá maior flexibilidade à construção dos autômatos. Sistematizando estas considerações, obtemos a seguinte definição.

Um *autômato de pilha não determinístico* M fica completamente definido pelos seguintes elementos:

ALFABETO DE ENTRADA: Σ;

ALFABETO DA PILHA: Γ;

CONJUNTO DE ESTADOS: $Q = \{q_1, \ldots, q_n\}$;

ESTADO INICIAL: q_1;

CONJUNTO DE ESTADOS FINAIS: $F \subseteq Q$;

FUNÇÃO DE TRANSIÇÃO: $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \to \mathcal{P}_f(Q \times \Gamma^*)$.

Como no caso dos autômatos finitos, abreviaremos os elementos de um autômato finito listando-os em um vetor, sempre na ordem acima. Portanto, o autômato de pilha M que acabamos de definir tem como elementos $(\Sigma, \Gamma, Q, q_1, F, \delta)$.

Há algumas considerações que precisamos fazer sobre a função de transição definida acima. A primeira diz respeito ao seu conjunto de chegada. Se, por analogia com autômatos finitos não determinísticos, escolhêssemos este conjunto como sendo $\mathcal{P}(Q \times \Gamma^*)$, abriríamos a possibilidade de uma quantidade infinita de escolhas em uma transição. Isto porquê, ao contrário do que ocorria com autômatos finitos, o conjunto $Q \times \Gamma^*$ é infinito. Para evitar este problema, restringimos o conjunto de chegada a $\mathcal{P}_f(Q \times \Gamma^*)$, que é o conjunto formado pelos subconjuntos finitos de $Q \times \Gamma^*$.

Voltando nossa atenção agora para o conjunto de partida da função de transição, note que δ toma valores em triplas formadas por um estado, um símbolo do alfabeto de entrada e um símbolo do alfabeto da pilha. Não há nada de muito surpreendente até aí, porque estamos tentando modelar os procedimentos da seção 1, que consultam tanto a entrada quanto a pilha. Entretanto, diante destas considerações, esperaríamos

que o domínio de δ fosse $Q \times \Sigma \times \Gamma$, contudo, o que de fato obtemos é

$$Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}).$$

Isto significa que o autômato pode efetuar uma transição sem consultar a entrada ou sem consultar a pilha, ou ambos! Portanto podemos esperar destes autômatos um comportamento ainda 'mais indeterminístico' que o dos autômatos finitos.

Para poder enquadrar os procedimentos da seção 1 na estrutura da definição acima precisamos saber como interpretar o comportamento da função de transição em termos mais concretos. Digamos que M é um autômato de pilha não determinístico cujos elementos obedecem à notação adotada na definição de autômato de pilha não determinístico. Suponhamos que

$$q \in Q, \quad \sigma \in \Sigma \cup \{\epsilon\} \quad \text{e} \quad \gamma \in \Gamma \cup \{\epsilon\}.$$

Dados $p \in Q$ e $u \in \Gamma^*$, a condição

$$(p, u) \in \delta(q, \sigma, \gamma),$$

significa que se o autômato M está no estado q, lendo o símbolo σ na entrada, e tendo γ no topo da pilha, então ao consumir σ:

- M muda para o estado p;
- M troca γ por u no topo da pilha.

Além disso, se $\sigma = \epsilon$ temos que a entrada não é consultada. Já quando $\gamma = \epsilon$, a transição é efetuada sem que o topo da pilha seja consultado.

Note que M pode trocar o símbolo do topo da pilha por uma palavra inteira. Em termos da pilha de discos perfurados da seção anterior isto significa que o disco do alto da pilha pode ser trocado por uma pilha de vários discos em uma única transição.

O fato de u poder ser uma palavra qualquer de Γ^* inclui a possibilidade de u ser ϵ. Mas o que significa trocar γ por ϵ no topo da pilha? Como ϵ não tem símbolos, retiramos γ e não pusemos nada no seu lugar. Portanto γ foi apenas removido da pilha.

Vale a pena enumerar os vários casos em que ϵ aparece para não deixar dúvidas quanto à interpretação correta de cada um:

Caso			Interpretação
$\sigma = \epsilon$			a entrada não é consultada
$\gamma = \epsilon$			o topo da pilha não é consultado
$\sigma = \gamma = \epsilon$			nem a entrada, nem o topo da pilha são consultados
$\gamma \neq \epsilon$	e	$u = \epsilon$	remove γ da pilha
$\gamma = \epsilon$	e	$u \neq \epsilon$	empilha u
$\gamma = \epsilon$	e	$u = \epsilon$	não altera a pilha

O que vimos já é suficiente para que possamos adaptar os procedimentos da seção 1 ao modelo de autômato de pilha não determinístico.

Analisando o procedimento que aceita a linguagem L_1, vemos que há apenas uma mudança de atitude, que corresponde à passagem de *empilha* para *desempilha*. Isto significa que o autômato de pilha $\mathcal{M}_1$ que desejamos construir deve ter dois estados, digamos q_1 e q_2.

No primeiro estado, $\mathcal{M}_1$ põe um símbolo diferente na pilha para cada a ou b encontrado na entrada. Para facilitar, podemos imaginar que o alfabeto da pilha é $\{a, b\}$. Assim, para cada a achado na entrada, o autômato acrescenta um a no topo da pilha; e para cada b da entrada, um b é acrescentado ao topo da pilha. Temos assim as duas transições

$$\delta(q_1, a, \epsilon) = \{(q_1, a)\},$$
$$\delta(q_2, b, \epsilon) = \{(q_2, b)\}.$$

Note o ϵ indicando que um símbolo é empilhado, não importando o que esteja no topo da pilha.

Por outro lado, ao achar um c na entrada $\mathcal{M}_1$ muda de atitude, preparando-se para passar a desempilhar. Como esta mudança de atitude ocorre sem que nada seja feito à pilha, temos a transição

$$\delta(q_1, c, \epsilon) = \{(q_2, \epsilon)\}.$$

Já no estado q_2, o autômato passa a comparar o símbolo da entrada com o que está no topo da pilha. Se os símbolos na entrada e na pilha coincidem, o autômato remove o símbolo que está no topo da pilha. Portanto,

$$\delta(q_2, a, a) = \{(q_2, \epsilon)\},$$
$$\delta(q_2, b, b) = \{(q_2, \epsilon)\}.$$

Para completar a descrição de $\mathcal{M}_1$ precisamos ainda decidir sobre seu estado inicial e seus estados finais. Não há problema quanto ao estado inicial que, claramente, deve ser q_1. E quanto aos estados finais? De acordo com a definição do procedimento em 1, uma palavra é aceita

quando for inteiramente consumida e a pilha estiver vazia. Mas isso só pode acontecer quando o estado final for q_2. Como é razoável esperar que, se o autômato aceita a entrada, então parou em um estado final, vamos declarar q_2 como sendo final.

A descrição do autômato acima seria, sem dúvida, mais fácil de interpretar se tivéssemos uma maneira mais compacta de descrever as transições. Como no caso dos autômatos finitos, faremos isto usando uma tabela que, além da palavra de entrada e do estado, registrará o que o autômato faz à pilha.

Suponhamos que temos um autômato de pilha e que $q, p_1, \ldots, p_s$ são estados, σ é um símbolo do alfabeto de entrada, γ um símbolo do alfabeto da pilha e $u_1, \ldots, u_s$ palavras no alfabeto da pilha. Uma transição

$$\delta(q, \sigma, \gamma) = \{(p_1, u_1), \ldots, (p_s, u_s)\},$$

corresponderá a uma entrada da seguinte forma na tabela:

estado	entrada	topo da pilha	transições
q	σ	γ	(p_1, u_1)
			$\vdots$
			(p_s, u_s)

Observe que os diferentes pares $(\text{estado}, \text{símbolo da pilha})$ de uma mesma transição são listados um sobre o outro sem uma linha divisória. As linhas horizontais da tabela separarão transições distintas uma da outra. Em muitos casos é conveniente acrescentar a este modelo básico de tabela uma quinta coluna com comentários sobre o que o autômato está fazendo naquela transição. É claro que estes comentários não fazem parte da descrição formal do autômato de pilha, eles apenas nos ajudam a entender como ele se comporta.

Utilizando esta notação, a tabela do autômato $\mathcal{M}_1$ é seguinte:

estado	entrada	topo da pilha	transições	comentários
q_1	a	ϵ	(q_1, a)	acha a e empilha a
q_1	b	ϵ	(q_1, b)	acha b e empilha b
q_1	c	ϵ	(q_2, ϵ)	acha c e muda de estado
q_2	a	a	(q_2, ϵ)	acha a na entrada e pilha e o desempilha
q_2	b	b	(q_2, ϵ)	acha b na entrada e pilha e o desempilha

No caso da linguagem L_2 da seção 1, o procedimento que criamos era não determinístico. Por isso, para modelá-lo como um autômato de pilha precisamos dar ao autômato a capacidade de, a qualquer momento, alterar o seu comportamento, deixando de empilhar e passando a desempilhar. Assim, enquanto está no estado q_1 o autômato pode ter duas atitudes:

Primeira: verificar qual é o símbolo de entrada e acrescentar o símbolo correspondente ao topo da pilha; ou

Segunda: ignorar a entrada e a pilha, mudar de estado e passar a desempilhar.

Levando em conta estas considerações, obtemos um autômato de pilha $\mathcal{M}_2$ que tem $\{a, b\}$ como alfabeto de entrada e da pilha; estados q_1 e q_2; estado inicial q_1, estado final q_2 e função de transição dada por:

estado	entrada	topo da pilha	transições	comentários
q_1	a	ϵ	(q_1, a)	acha a e empilha a
q_1	b	ϵ	(q_1, b)	acha b e empilha b
q_1	ϵ	ϵ	(q_2, ϵ)	muda de estado
q_2	a	a	(q_2, ϵ)	acha a na entrada e pilha e o desempilha
q_2	b	b	(q_2, ϵ)	acha b na entrada e pilha e o desempilha

Observe que vale para a tabela a ressalva feita para os procedimentos na seção 1. Isto é, se surgir uma situação que leve a uma transição que não esteja especificada na tabela, então o autômato pára de se mover. Aliás, o mesmo valia para os autômatos finitos não determinísticos.

3. Computando e aceitando

Apesar de já termos uma descrição formal dos autômatos de pilha, ainda precisamos definir de maneira precisa o que significa um autômato de pilha aceitar uma linguagem. Faremos isto adaptando as noções correspondentes da teoria de autômatos finitos.

Seja M um autômato de pilha cujos elementos são dados pelo vetor $(\Sigma, \Gamma, Q, q_1, F, \delta)$. Uma *configuração* de M é um elemento de $Q \times \Sigma^* \times \Gamma^*$; ou seja, é uma tripla (q, w, u) onde

- q é um estado de M;
- w é uma palavra no alfabeto de entrada;
- u é uma palavra no alfabeto da pilha.

Por uma questão de coerência com a maneira como a palavra w é lida, listamos os elementos da pilha em u de modo que o topo da pilha corresponda ao símbolo mais à esquerda de u.

Como ocorreu com os autômatos finitos, a finalidade desta definição é permitir que possamos acompanhar de maneira simples o comportamento de M com uma dada entrada. Portanto, esta definição só faz sentido quando vem conjugada à relação *configuração seguinte*. Suponhamos que $C = (q, \sigma w, \gamma u)$ é uma configuração de M, onde $\sigma \in \Sigma \cup \{\epsilon\}$, $\gamma \in \Gamma \cup \{\epsilon\}$. Dizemos que $C' = (q', w, vu)$ é uma das configurações *seguintes* a C se

$$(q', v) \in \delta(q, \sigma, \gamma).$$

Neste caso, escrevemos $C \vdash_M C'$.

Uma *computação* de M é uma seqüência de configurações $C_0, \cdots, C_k$ tais que C_{i+1} é uma das configurações seguintes a C_i. Freqüentemente abreviaremos a computação acima na forma $C_0 \vdash_M^k C_k$, já que, na maioria dos casos, estaremos interessados apenas nas configurações com as quais a computação começa e termina. Se o número de etapas da computação não for conhecido, ou não for relevante para as nossas considerações, escreveremos apenas $C_0 \vdash_M^* C_k$. Usaremos $\vdash$ em lugar de $\vdash_M$ a não ser que haja risco de confusão sobre qual o autômato que está sendo considerado.

Como no caso de autômatos finitos, convencionaremos que se C é uma configuração, então $C \vdash^0 C$, e também $C \vdash^* C$. Note, contudo, que $C \vdash^0 C$ não é equivalente a uma transição com entrada vazia. Por exemplo, o autômato $\mathcal{M}_2$ satisfaz

$$(q_1, \epsilon, \epsilon) \vdash (q_2, \epsilon, \epsilon).$$

Mas, apesar da entrada e da pilha não terem sido alteradas, as configurações inicial e final não coincidem.

Vejamos estas definições em ação em alguns exemplos. Partindo da configuração $(q_1, abcba^2, a)$ temos a seguinte computação no autômato $\mathcal{M}_1$ da seção 2:

$$(q_1, abcba^2, a) \vdash (q_1, bcba^2, a^2) \vdash (q_1, cba^2, ba^2) \vdash (q_2, ba^2, ba^2) \vdash (q_2, a^2, a^2) \vdash (q_2, a, a) \vdash (q_2, \epsilon, \epsilon).$$

Esta é, essencialmente, a única computação possível partindo de $(q_1, abcba^2, a)$. A única coisa que podemos fazer para obter uma computação diferente é interromper a computação acima antes de chegar a $(q_2, \epsilon, \epsilon)$. Entretanto, isto está longe de ser sempre verdade, como mostra o exemplo seguinte.

Desta vez queremos considerar o autômato de pilha $\mathcal{M}_2$ definido ao final da seção 2. Digamos que a configuração inicial seja (q_1, a^2b, ϵ). Uma computação possível é

$$(q_1, a^2b, \epsilon) \vdash (q_1, ab, a) \vdash (q_1, b, a^2) \vdash (q_2, b, a^2).$$

Mas há muitas outras possibilidades, como:

$$(q_1, a^2b, \epsilon) \vdash (q_2, a^2b, \epsilon)$$

ou ainda

$$(q_1, a^2b, \epsilon) \vdash (q_1, ab, a) \vdash (q_2, ab, a).$$

A diferença é que o primeiro autômato era basicamente determinístico, ao passo que este último é claramente não determinístico.

Supondo, como antes, que M é um autômato de pilha cujos elementos são dados pelo vetor $(\Sigma, \Gamma, Q, q_1, F, \delta)$, diremos que uma palavra $w \in \Sigma^*$ é *aceita* por M se existe uma computação

$$(q_1, w, \epsilon) \vdash^* (p, \epsilon, \epsilon),$$

onde p é um estado final de M. Como já ocorria no caso de autômatos finitos não determinísticos, basta que exista uma computação como acima para que a palavra seja aceita. A *linguagem $L(M)$ aceita por M* é o conjunto de todas as palavras que M aceita.

Em geral não é fácil determinar por simples inspecção qual a linguagem aceita por um autômato de pilha não determinístico dado. Contudo, veremos no próximo capítulo que é possível construir uma gramática livre de contexto que gere $L(M)$ diretamente da descrição de M.

Observe que três condições precisam ser *simultaneamente* satisfeitas para que possamos afirmar, ao final de uma computação, que M aceita w:

(1) a palavra w tem que ter sido completamente consumida;
(2) a pilha tem que estar vazia;
(3) o autômato tem que ter atingido um estado final.

A condição referente ao estado final não aparece na descrição dos procedimentos na seção 1. De fato, ela surgiu na definição formal de autômato de pilha sob a pífia justificativa de que seria razoável exigir que o autômato parasse aceitando num estado final!

Como sempre acontece, a exigência de que o autômato tenha que atingir um estado final para que a entrada seja aceita simplifica a construção de alguns autômatos e complica a de outros. A verdade é que, como o exercício 7 mostra, teria sido possível suprimir toda menção a estados finais, embora isto não seja desejável do ponto de vista do desenvolvimento da teoria.

4. Variações em um tema

Nesta seção discutimos a construção de autômatos de pilha para três linguagens definidas de maneira muito semelhante. Com isto teremos a oportunidade de chamar a atenção para algumas dificuldades comuns; além de desenvolver técnicas simples, mas úteis, na construção de autômatos mais sofisticados.

Exemplo 1. Consideremos, em primeiro lugar, a linguagem livre de contexto

$$L = \{a^i b^i : i \geq 0\},$$

É fácil descrever um procedimento extremamente simples que usa uma pilha com apenas um tipo de disco para aceitar as palavras de L:

Etapa 1: Se achar um a na entrada, ponha um disco na pilha.

Etapa 2: Se achar um b na entrada mude de atitude e passe a comparar a entrada com a pilha, removendo um disco da pilha para cada b que achar na entrada (incluindo o primeiro!).

Para precisar o comportamento do autômato de maneira a não deixar dúvida sobre o que ele realmente faz basta construir sua tabela de transição.

estado	entrada	topo da pilha	transições	comentários
q_1	a	ϵ	(q_1, a)	empilha a
q_1	b	a	(q_2, ϵ)	desempilha a e muda de estado
q_2	b	a	(q_2, ϵ)	desempilha a

É claro que queremos que q_1 seja o estado inicial, mas precisamos tomar cuidado com a escolha dos estados finais. A primeira impressão talvez seja que q_2 é o único estado final. Entretanto, o autômato só pode

alcançar q_2 se houver algum símbolo na entrada, o que faria com que o autômato não aceitasse ϵ.

Há duas soluções possíveis. A primeira é declarar que q_1 também é um estado final. Isto com certeza faz com que ϵ seja aceita. Porém, precisamos nos certificar de que a inclusão de q_1 entre os estados finais não cria novas palavras aceitas que não pertencem a L. Fazemos isto analisando em detalhes o comportamento do autômato. Suponhamos, então, que $\epsilon \neq w \in \{a, b\}^*$ e que o autômato recebe w como entrada. Temos que:

- se w começa por b então nenhum de seus símbolos é consumido;
- se w começa por a então os as são empilhados e para desempilhá-los é preciso chegar ao estado q_2.

No primeiro caso a palavra não será aceita; no segundo, só será aceita se os as são seguidos pelo mesmo número de bs e nada mais. Ou seja, se declaramos que os estados finais são $\{q_1, q_2\}$ então o autômato resultante aceita L.

A segunda possibilidade é alterar as transições e dar ao autômato a alternativa de, sem consultar a entrada ou a pilha, mudar de estado de q_1 para q_2. Neste caso, o único estado final é $\{q_2\}$. Além disso, a transição que vamos acrescentar torna completamente redundante a terceira linha da tabela acima. A tabela que descreve a função de transição do autômato passa, então, a ser:

estado	entrada	topo da pilha	transições	comentários
q_1	ϵ	ϵ	(q_2, ϵ)	muda de estado
q_1	a	ϵ	(q_1, a)	empilha a
q_2	b	a	(q_2, ϵ)	desempilha a

O problema com esta última estratégia é que a transição que acrescentamos pode ser executada enquanto o autômato está no estado q_1, não importando o que está sendo lido, nem o que há na pilha. Para ter certeza que tudo ocorre como esperado, devemos analisar o que o autômato faria se usasse esta transição "no momento errado". Em primeiro lugar, se o autômato está no estado q_1 então ele não pode se mover ao ler b, a não ser que ignore a entrada e mude para o estado q_2. Portanto, o autômato se comportará de maneira anómala apenas se ainda estiver lendo a e executar a transição que permite mudar de estado sem afetar a entrada nem a pilha. Neste caso ainda haveria as a serem lidos. Como o autômato não se move se está no estado q_2 e lê a, então a computação

simplesmente pára. A conclusão é que o autômato continua aceitando o que devia apesar da alteração na tabela de transição.

Exemplo 2. A linguagem deste segundo exemplo é uma generalização da que aparece no primeiro. Seja L a linguagem formada pelas palavras $w \in \{a, b\}$ para as quais o número de as e bs é o mesmo. Assim, $abaaabbabb \in L$, mas $aaabaa \notin L$. Observe que a linguagem do exemplo anterior está contida em L.

À primeira vista, podemos construir um autômato de pilha que aceite L usando a mesma estratégia do Exemplo 1. Isto é, quando achamos a empilhamos a e quando achamos um b desempilhamos um a. O problema é o que fazer quando nos deparamos com uma palavra como $abba$. Neste caso começaríamos empilhando um a, mas em seguida teríamos de desempilhar dois as. Só que há apenas um a na pilha: como seria possível desempilhar mais as do que há na pilha?

A estratégia que vamos adotar consiste em construir um contador que:

- ao achar a soma 1 ao contador;
- ao achar b soma -1 ao contador.

Poderíamos fazer isto usando $\{-1, 1\}$ como alfabeto da pilha, entretanto, palavras como $-1 - 1$ estão demasiadamente sujeitas a causar confusão. Por isso, vamos simplesmente empilhar a para cada a encontrado na entrada, e b para cada b encontrado na entrada. Só precisamos lembrar que 'empilhar um b sobre um a' tem o efeito de desempilhar o b da pilha, e vice-versa.

O que ocorre quando aplicamos esta estratégia à palavra $abba$? O primeiro a contribui um a para a pilha, que por sua vez é removido pelo b seguinte. Portanto, depois de ler o prefixo ab o autômato está com a pilha vazia. Em seguida aparece um b e o autômato põe um b na pilha. Este b é seguido por um a. Portanto teríamos que acrescentar um a no topo da pilha. Mas isto tem o efeito de desempilhar o b anterior, e a pilha acaba vazia, como desejado.

Parece fácil transcrever esta estratégia em uma tabela de transição, mas ainda há um obstáculo a vencer antes de podermos fazer isto. Afinal, para que esta estratégia funcione o autômato precisa ser capaz de detectar que a pilha está vazia. Entretanto, escrever ϵ para o topo da pilha em uma tabela de transição *não significa que a pilha está vazia*; significa apenas que o autômato não precisa saber o que está no topo da pilha ao realizar esta transição.

A saída é inventar um novo símbolo β para o alfabeto da pilha. Este símbolo é acrescentado à pilha ainda vazia, no início da computação.

Daí em diante, o autômato opera apenas com as e bs na pilha. Assim, ao avistar β no topo da pilha o autômato sabe que a pilha não contém mais nenhum a ou b. Para garantir que β só vai ser usado uma vez, convém reservar ao estado inicial apenas a ação de marcar o fundo da pilha. Portanto, se q_1 for o estado inicial teremos

$$\delta(q_1, \epsilon, \epsilon) = \{(q_2, \beta)\}.$$

Ao final desta transição, a entrada não foi alterada, o fundo da pilha foi marcado e o autômato saiu do estado q_1 para onde não vai poder mais voltar. O que o autômato deve fazer no estado q_2 está descrito resumidamente na seguinte tabela:

Acha na entrada	Acha na pilha	Faz
a	β ou a	empilha a
b	β ou b	empilha b
b	a	desempilha a
a	b	desempilha b

Com isto se, ao final da computação, a pilha contém apenas o marcador β então a palavra está em L e é aceita; do contrário a palavra é rejeitada. Entretanto, pela definição formal, dada na seção 3, o autômato só pode aceitar a palavra se a pilha estiver completamente vazia. Isto sugere que precisamos de mais uma transição para remover o β do fundo da pilha.

Com isto surge um novo problema. Se a palavra está em L, então será totalmente consumida deixando na pilha apenas β. Portanto, a transição que remove o β ao final desta computação, e esvazia completamente a pilha, não tem nenhuma entrada para consultar. O problema é que se permitimos ao autômato remover β sem consultar a entrada, ele pode realizar este movimento no momento errado, antes que a entrada tenha sido consumida. Como os nossos autômatos são não determinísticos, isto não apresenta nenhum problema conceitual.

Infelizmente o fato de ainda haver símbolos na entrada abre a possibilidade do autômato continuar a computação depois de ter retirado o marcador do fundo. Para evitar isto, o autômato deve, ao remover β, passar a um novo estado q_3 a partir do qual não há transições. Assim, se o autômato decidir remover o marcador antes da entrada ser totalmente consumida ele será obrigado a parar, e não poderá aceitar a entrada nesta computação. Como a pilha só vai poder se esvaziar em q_3, é claro que este será o único estado final deste autômato.

Resumindo, temos um autômato que tem $\{a, b\}$ como alfabeto de entrada e da pilha; estados q_1, q_2 e q_3; estado inicial q_1; estado final q_3; e cuja função de transição é definida na tabela 1.

estado	entrada	topo da pilha	transições	comentários
q_1	ϵ	ϵ	(q_2, β)	marca o fundo da pilha
q_2	a	β	$(q_2, a\beta)$	acha a e empilha um a
q_2	b	β	$(q_2, b\beta)$	acha b e empilha um b
q_2	a	a	(q_2, aa)	acha a e empilha um a
q_2	b	b	(q_2, bb)	acha b e empilha um b
q_2	b	a	(q_2, ϵ)	desempilha um a
q_2	a	b	(q_2, ϵ)	desempilha um b
q_2	ϵ	β	(q_3, ϵ)	esvazia a pilha

TABELA 1

Exemplo 3. A terceira linguagem que desejamos considerar é o conjunto das palavras $w \in \{a, b\}$ que têm mais as que bs. Vamos chamá-la de L_3. À primeira vista, pode parecer que autômato é essencialmente igual ao anterior, bastando alterar a condição sob a qual a palavra é aceita. Afinal, se ao final da computação do autômato do Exemplo 2 sobram as na pilha, então a palavra de entrada tem mais as que bs. Mas surgem alguns problemas técnicos quando tentamos implementar esta estratégia.

A primeira dificuldade é que não temos como codificar nas transições o fato de não haver mais símbolos na entrada. A saída é permitir que o autômato tente advinhar isto por conta própria. Assim, ao achar um a na pilha, o autômato deve poder decidir que a computação chegou ao fim e, sem consultar a entrada, passar a um estado final. Naturalmente uma palavra não será aceita se a decisão de aplicar esta transição for tomada antes que a entrada tenha sido completamente consumida.

A segunda dificuldade é que estamos identificando que uma palavra está em L_3 porque sobram as na pilha. Entretanto, um dos requisitos para que uma palavra seja aceita por um autômato de pilha é que não sobrem símbolos na pilha ao final da computação! Resolvemos este conflito fazendo com que, ao chegar ao estado final, o autômato possa esvaziar a pilha sem se preocupar com a entrada.

Tomando por base o autômato do Exemplo 2, teremos que substituir a transição codificada na última linha da tabela, e acrescentar as três

transições que permitem ao autômato esvaziar a pilha. O autômato resultante terá alfabetos de entrada e da pilha iguais a $\{a, b\}$; estados q_1, q_2 e q_3; estado inicial q_1; estado final q_3; e sua função de transição será dada na tabela 2.

estado	entrada	topo da pilha	transições	comentários
q_1	ϵ	ϵ	(q_2, β)	marca o fundo da pilha
q_2	a	β	$(q_2, a\beta)$	empilha um a
q_2	b	β	$(q_2, b\beta)$	empilha um b
q_2	a	a	(q_2, aa)	empilha um a
q_2	b	b	(q_2, bb)	empilha um b
q_2	b	a	(q_2, ϵ)	desempilha um a
q_2	a	b	(q_2, ϵ)	desempilha um b
q_2	ϵ	a	(q_3, a)	decide que a computação acabou com as sobrando na pilha
q_3	ϵ	a	(q_3, ϵ)	retira a da pilha
q_3	ϵ	b	(q_3, ϵ)	retira b da pilha
q_3	ϵ	β	(q_3, ϵ)	retira β da pilha

TABELA 2

5. Exercícios

1. Considere o autômato de pilha não determinístico $\mathcal{M}$ com alfabetos $\Sigma = \{a, b\}$ e $\Gamma = \{a\}$, estados q_1 e q_2, estado inicial q_1 e final q_2 e transições dadas pela tabela:

estado	entrada	topo da pilha	transições
q_1	a	ϵ	(q_1, a) (q_2, ϵ)
q_1	b	ϵ	(q_1, a)
q_2	a	a	(q_2, ϵ)
q_2	b	a	(q_2, ϵ)

   (a) Descreva todas as possíveis seqüências de transições de $\mathcal{M}$ na entrada aba.
   (b) Mostre que aba, aa e abb não pertencem a $L(\mathcal{M})$ e que baa, bab e $baaaa$ pertencem a $L(\mathcal{M})$.

(c) Descreva a linguagem aceita por $\mathcal{M}$ em português.

2. Ache um autômato de pilha não determinístico cuja linguagem aceita é L onde:
   (a) $L = \{a^n b^{n+1} : n \geq 0\}$;
   (b) $L = \{a^n b^{2n} : n \geq 0\}$;
   (c) $L = \{w \in \{0,1\}^* : w$ cujo número de as é diferente do de bs$\}$;
   (d) $L = \{a^n b^m : m, n \geq 0$ e $m \neq n\}$;
   (e) $L = \{w_1 c w_2 : w_1, w_2 \in \{a,b\}^*$ e $w_1 \neq w_2^r\}$;

3. Um autômato finito não determinístico que aceita a linguagem denotada por $0 \cdot 0^* \cdot 1 \cdot 0$ não pode ter menos de 4 estados. Construa um autômato de pilha não determinístico com apenas 2 estados que aceita esta linguagem.

4. Considere a linguagem dos parênteses balanceados descrita no exercício 2 da lista 4.
   (a) Dê exemplo de uma gramática livre de contexto que gere esta linguagem.
   (b) Dê exemplo de um autômato de pilha não determinístico que aceita esta linguagem.

5. Esta questão trata da existência ou inexistência de computações infinitas.
   (a) Explique porque um autômato finito (determinístico ou não) não admite uma computação com um número infinito de etapas.
   (b) Dê exemplo de um autômato de pilha não determinístico que admite uma computação com um número infinito de etapas.

6. Seja M um autômato de pilha. Mostre como definir a partir de M um novo autômato de pilha M_s que aceita a mesma linguagem que M e que, além disso, satisfaz às seguintes condições:
   (a) a única transição de M_s a partir do seu estado inicial i é $\delta(i, \epsilon, \epsilon) = \{(q, \beta)\}$, onde q é um estado, β um símbolo do alfabeto da pilha e δ a função de transição de $\mathcal{M}_s$;
   (b) o único estado de M_s a partir do qual há transições sem consultar a pilha é i;
   (c) M_s tem um único estado final f;
   (d) as transições que desempilham β levam o autômato ao estado f;
   (e) não há transições a partir de f.

7. Construa uma autômato de pilha não determinístico que aceite o *complementar* da linguagem

$$L = \{ww^R : w \in (0 \cup 1)^*\}.$$

8. Seja M um autômato de pilha não determinístico com alfabeto de entrada Σ, estado inicial q_1 e conjunto de estados Q. A *linguagem que M aceita por pilha vazia* é definida como:

$$N(M) = \{w \in \Sigma^* : \text{ existe } (q_1, w, \epsilon) \vdash^* (q, \epsilon, \epsilon) \text{ onde } q \in Q\}.$$

Note que a diferença entre $L(M)$ e $N(M)$ é que, para a palavra ser aceita em $L(M)$ tem que ser possível chegar a uma configuração (q, ϵ, ϵ) em que q é um estado final, ao passo que não há restrições sobre o estado no caso de $N(M)$.

(a) Dê exemplo de um autômato de pilha não determinístico M para o qual $N(M) \neq L(M)$.
(b) Mostre que dado um autômato de pilha não determinístico M existe um autômato de pilha não determinístico M' tal que $L(M) = N(M')$.
(c) Mostre que dado um autômato de pilha não determinístico M existe um autômato de pilha não determinístico M' tal que $N(M) = L(M')$.

SUGESTÃO: use o autômato M_s construído no exercício anterior.

Resolução do Exercício 7

O autômato tem alfabeto de entrada $\{0, 1\}$, alfabeto da pilha $\{\beta, 0, 1\}$, conjunto de estados $\{q_1, \ldots, q_4\}$, estado inicial q_1, estado final q_4 e a seguinte tabela de transição:

estado	entrada	topo da pilha	transições	comentários
q_1	ϵ	ϵ	(q_2, β)	marca o fundo da pilha
q_2	0	ϵ	$(q_2, 0)$	empilha 0
q_2	1	ϵ	$(q_2, 1)$	empilha 1
q_2	ϵ	ϵ	$(q_3, 0)$	tenta advinhar o meio da entrada
q_3	0	0	(q_3, ϵ)	desempilha 0s casados
q_3	1	1	(q_3, ϵ)	desempilha 1s casados
q_3	0	1	(q_4, ϵ)	acha símbolo descasado
q_3	1	0	(q_4, ϵ)	acha símbolo descasado
q_4	0	0	(q_4, ϵ)	continua a desempilhar
q_4	0	1	(q_4, ϵ)	continua a desempilhar
q_4	1	0	(q_4, ϵ)	continua a desempilhar
q_4	1	1	(q_4, ϵ)	continua a desempilhar
q_4	ϵ	β	(q_4, ϵ)	esvazia a pilha

Depois de marcar o fundo da pilha o autômato empilha 0s e 1s até decidir, de maneira não determinística, que o meio da palavra de entrada foi encontrado. Então, muda de estado e passa a desempilhar em busca de um símbolo da pilha que não corresponda ao que está vendo na entrada. Ao achar um tal símbolo o autômato muda para o estado final e esvazia a pilha, comparando-a símbolo a símbolo com a entrada. Precisamos fazer isto para ter certeza de que o autômato advinhou o meio da palavra corretamente. Se não sobrarem nem 0s, nem 1s na pilha, o autômato remove o marcador do fundo e esvazia a pilha.

Note que, se o autômato advinhar incorretamente o meio da palavra, então sobrarão símbolos na entrada (se advinhar cedo demais) ou na pilha (se advinhar tarde demais). Em nenhum destes casos a palavra será aceita. Um destes casos ocorrerá obrigatoriamente se a palavra tiver comprimento ímpar. Finalmente, se a palavra da entrada estiver em L e se o autômato advinhar corretamente onde está o meio da palavra então a entrada será consumida e todos os 0s e 1s serão removidos da pilha. Entretanto, como o autômato continua no estado q_3 o marcador do fundo da pilha não é removido, de modo que a pilha não será esvaziada e a palvra será não será aceita.

CAPíTULO 13

Gramáticas e autômatos de pilha

Nosso objetivo neste capítulo é dar uma demonstração do seguinte resultado fundamental.

TEOREMA 13.1. *Uma linguagem é livre de contexto se, e somente se, é aceita por algum autômato de pilha não determinístico.*

Como uma linguagem é livre de contexto se for gerada por uma gramática livre de contexto, o que precisamos fazer é estabelecer um elo entre gramáticas livres de contexto e autômatos de pilha.

Já temos alguma experiência com este tipo de questão. De fato, provamos que uma linguagem gerada por uma gramática linear à direita é regular construindo um autômato finito cujas computações simulam derivações na gramática dada. No caso de linguagens livres de contexto a correspondência será entre computações no autômato de pilha e derivações mais à esquerda na gramática.

Como no caso das linguagens regulares o problema pode ser dividido em dois. Assim, dada uma gramática livre de contexto $\mathcal{G}$ precisamos de uma receita para construir um autômato de pilha não determinístico que aceite $L(\mathcal{G})$. Reciprocamente, dado um autômato de pilha $\mathcal{M}$ queremos construir uma gramática livre de contexto que gere a linguagem aceita por $\mathcal{M}$.

1. O autômato de pilha de uma gramática

Seja $\mathcal{G}$ uma gramática livre de contexto. Nosso objetivo nesta seção consiste em construir um autômato de pilha cujas computações simulem as derivações à esquerda em $\mathcal{G}$. É claro que a pilha tem que desempenhar um papel fundamental nesta simulação. O que de fato acontece é que o

papel dos estados é secundário, e a simulação se dá na pilha. Dizendo de outra maneira, construiremos um autômato de pilha que simula *na pilha* as derivações mais à esquerda em $\mathcal{G}$.

Digamos que a gramática livre de contexto $\mathcal{G}$ é definida pela quádrupla de elementos (T, V, S, R), e seja $\mathcal{M}$ o autômato de pilha a ser construído a partir de $\mathcal{G}$. Como $\mathcal{M}$ deve aceitar $L(\mathcal{G})$, é claro que seu alfabeto de entrada tem que ser T. Chamaremos a função de transição de $\mathcal{M}$ de δ e seu estado inicial de i.

A maneira mais concreta de pôr em prática a idéia delineada acima consiste em escolher o alfabeto da pilha como sendo $T \cup V$, e fazer corresponder a cada derivação de $\mathcal{G}$ uma transição do autômato. Entretanto, como a derivação de palavras de $L(\mathcal{G})$ é feita a partir do símbolo inicial S, o autômato deve começar pondo este símbolo no fundo da pilha. Observe que o autômato não pode consumir entrada ao marcar o fundo da pilha, já que precisamos da entrada para guiá-lo na busca da derivação correta. Nem adianta consultar a pilha, já que ainda está totalmente vazia. Entretanto, uma transição que não consulta a entrada nem a pilha pode ser executada em qualquer momento da computação. Por isso forçamos o autômato a mudar de estado depois desta transição, impedindo assim que volte a ser executada. Temos, então, que

$$\delta(i, \epsilon, \epsilon) = \{(f, S)\},$$

onde f é um estado do autômato diferente de i.

Daí em diante toda a ação vai se processar na pilha, e podemos simular a derivação sem nenhuma mudança de estado adicional. Portanto, f será o estado final de $\mathcal{M}$. Assim, à regra $X \to \alpha$ de $\mathcal{G}$ fazemos corresponder a transição

$$(f, \epsilon, \alpha) \in \delta(f, X) \tag{1.1}$$

de $\mathcal{M}$. Contudo, a construção do autômato ainda não está completa. O problema é que $\mathcal{M}$ só pode aplicar uma transição como (1.1) se a variável X estiver *no topo da pilha*. Infelizmente isto nem sempre acontece, como ilustra o exemplo a seguir.

Suponhamos que $\mathcal{G}_1$ é a gramática com terminais $\{a, b, c\}$, variável $\{S\}$, símbolo inicial S, e regras

$$S \to Sc \quad |aSb \quad | \quad \epsilon.$$

De acordo com a discussão acima o autômato $\mathcal{M}_1$ correspondente a $\mathcal{G}_1$ deve ter $\{a, b, c\}$ como alfabeto de entrada, $\{a, b, c, S\}$ como alfabeto da pilha e conjunto de estados $\{i, f\}$, onde i é o estado inicial e f o estado final. Lembrando que a cada regra de $\mathcal{G}_1$ deve corresponder uma transição de $\mathcal{M}_1$, concluímos que a tabela resultante deve ser

Estado	Entrada	Topo da pilha	Transição
i	ϵ	ϵ	(f, S)
f	ϵ	S	(f, Sc) (f, aSb) (f, ϵ)

A palavra abc^2 tem derivação mais à esquerda

$$S \Rightarrow Sc \Rightarrow Sc^2 \Rightarrow aSbc^2 \Rightarrow abc^2$$

em $\mathcal{G}_1$. Resta-nos verificar se somos capazes, ao menos neste exemplo, de construir uma computação de $\mathcal{M}_1$ que copie na pilha esta derivação de abc^2. A computação deve começar marcando o fundo da pilha com S e deve prosseguir, a partir daí, executando, uma a uma, as transições de $\mathcal{M}_1$ que correspondem às regras aplicadas na derivação acima. Isto nos dá

(1.2)

$$(i, abc^2, \epsilon) \vdash (f, abc^2, S) \vdash (f, abc^2, Sc) \vdash (f, abc^2, Sc^2) \vdash (f, abc^2, aSbc^2).$$

Até aqui tudo bem, mas a transição seguinte deveria substituir S por ϵ. Só que para isto ser possível a variável S tem que estar no topo da pilha, o que não acontece neste caso. Observe, contudo, que o a que apareceu na pilha acima do S na última configuração de (1.2) corresponde ao primeiro símbolo da palavra de entrada. Além disso, como se trata de uma derivação mais à esquerda, este símbolo não será mais alterado. Concluímos que, como o primeiro símbolo da palavra já foi corretamente derivado, podemos 'esquecê-lo' e partir para o símbolo seguinte. Para implementar isto na prática basta apagar da pilha os terminais que precedem a variável mais à esquerda da pilha *e que já foram corretamente construídos*. Isto significa acrescentar à tabela acima transições que permitam apagar terminais que apareçam simultaneamente na entrada e na pilha:

Estado	Entrada	Topo da pilha	Transição
f	a	a	(f, ϵ)
f	b	b	(f, ϵ)
f	c	c	(f, ϵ)

Levando isto em conta a computação acima continua, a partir da última configuração de (1.2) da seguinte maneira

(1.3)

$$(f, abc^2, aSbc^2) \vdash (f, bc^2, Sbc^2) \vdash (f, bc^2, bc^2) \vdash (f, c^2, c^2) \vdash (f, c, c) \vdash (f, \epsilon, \epsilon).$$

Observe que a passagem da segunda para a terceira configuração em (1.3) corresponde à regra $S \to \epsilon$. Todas as outras etapas são aplicações das transições da segunda tabela.

2. A receita e mais um exemplo

Podemos agora descrever de maneira sistematica a receita usada para construir um autômato de pilha não determinístico $\mathcal{M}$ cuja linguagem aceita é $L(\mathcal{G})$. Se a gramática livre de contexto $\mathcal{G}$ tem por elementos (T, V, S, R), então o autômato ficará completamente determinado pelos seguintes os elementos:

- o alfabeto de entrada T;
- o alfabeto da pilha $T \cup V$;
- o conjunto de estados $\{i, f\}$;
- o estado inicial i;
- o conjunto de estados finais $\{f\}$;
- a função de transição

$$\delta : \{i, f\} \times (T \cup \{\epsilon\}) \times (T \cup V \cup \{\epsilon\}) \to \{i, f\} \times (T \cup V)^*$$

que é definida por

$$\delta(q, \sigma, \gamma) = \begin{cases} (f, S) & \text{se } q = i \text{ e } \sigma = \gamma = \epsilon \\ \{(f, u) : \text{onde } X \to u \text{ é regra de } \mathcal{G}\} & \text{se } q = f \text{ e } \gamma = X \in V \\ (f, \epsilon) & \text{se } q = f \text{ e } \sigma = \gamma \in T \end{cases}$$

Este autômato executa dois tipos diferentes de transição a partir do estado f. As primeiras permitem substituir uma variável X no topo da pilha por $u \in (T \cup V)^*$ quando $X \to u$ é uma regra se $\mathcal{G}$, e serão chamadas *substituições*. As segundas permitem remover terminais que aparecem casados na pilha e na entrada, e vamos chamá-las *remoções*. Observe que, em geral, este autômato terá um comportamento muito pouco determinístico porque cada uma de suas transição corresponde ao conjunto de todas as regras que têm uma mesma variável do lado direito.

Vejamos mais um exemplo de autômato de pilha construído a partir de uma gramática livre de contexto pela receita acima. Considere a

gramática $\mathcal{G}'_{exp}$, definida no capítulo 9, cujas regras são

$$\begin{aligned} E &\to E + T \quad | \quad T \\ T &\to T * F \quad | \quad F \\ F &\to (E) \quad | \quad \text{id}. \end{aligned}$$

O alfabeto de entrada do autômato de pilha $\mathcal{M}'_{exp}$ correspondente a esta gramática é $\{id, +, *, (,)\}$, o alfabeto da pilha é $\{id, +, *, (,), E, T, F\}$, os estados são $\{i, f\}$, o estado inicial é i, o estado final é f e a função de transição é definida pela tabela 1.

Estado	Entrada	Topo da pilha	Transição
i	ϵ	ϵ	(f, S)
f	ϵ	E	$(f, E + T)$ (f, T)
f	ϵ	T	$(f, T * F)$ (f, F)
f	ϵ	F	$(f, (E))$ (f, id)
f	(	(	(f, ϵ)
f	)	)	(f, ϵ)
f	$+$	$+$	(f, ϵ)
f	$*$	$*$	(f, ϵ)
f	id	id	(f, ϵ)

TABELA 1. Tabela de transição

Quando criamos a receita acima, nosso objetivo era inventar um autômato que aceitasse a linguagem gerada por uma gramática livre de contexto dada. Mas para ter certeza de que a receita funciona precisamos provar o seguinte resultado.

> Se $\mathcal{G}$ é uma gramática livre de contexto e $\mathcal{M}$ é o autômato de pilha não determinístico construído de acordo com a receita acima, então $L(\mathcal{G}) = L(\mathcal{M})$.

A demonstração deste resultado consiste em formalizar a relação entre derivações mais à esquerda em $\mathcal{G}$ e computações em $\mathcal{M}$, que foi o nosso ponto de partida para a criação da receita. Antes de fazer isto, porém, vale a pena tentar equiparar cada etapa de uma derivação mais à

esquerda em $\mathcal{G}'_{exp}$ com uma computação no autômato que acabamos de descrever. Por exemplo, vamos construir a computação correspondente à derivação mais à esquerda

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow \mathrm{id} + T \Rightarrow \mathrm{id} + F \Rightarrow \mathrm{id} + \mathrm{id}$$

e compará-las passo-a-passo. A derivação começa marcando o fundo da pilha com o símbolo inicial de $\mathcal{G}'_{exp}$

$$(i, \mathrm{id} + \mathrm{id}, \epsilon) \vdash (f, \mathrm{id} + \mathrm{id}, E),$$

mas isto não corresponde a nenhuma etapa da derivação. Em seguida, utilizando transições por substituição, reproduzimos na pilha as quatro primeiras etapas da derivação

$$(f, \mathrm{id}{+}\mathrm{id}, E) \vdash (f, \mathrm{id}{+}\mathrm{id}, E{+}T) \vdash (f, \mathrm{id}{+}\mathrm{id}, T{+}T) \vdash (f, \mathrm{id}{+}\mathrm{id}, F{+}T) \vdash (f, \mathrm{id} + \mathrm{id}, \mathrm{id} + T).$$

Neste ponto nos deparamos com a necessidade de usar remoções para eliminar os terminais que obstruem o topo da pilha:

$$(f, \mathrm{id} + \mathrm{id}, \mathrm{id} + T) \vdash (f, +\mathrm{id}, +T) \vdash (f, \mathrm{id}, T).$$

A computação prossegue com a aplicação de mais duas substituições e a remoção do último terminal na pilha e na entrada:

$$(f, \mathrm{id}, T) \vdash (f, \mathrm{id}, F) \vdash (f, \mathrm{id}, \mathrm{id}) \vdash (f, \epsilon, \epsilon).$$

Como era esperado, o número de passos na computação excede em muito o número de etapas da derivação por causa das transições de remoção, que não correspondem a nenhuma regra de $\mathcal{G}'_{exp}$. Observe que é muito fácil determinar se um passo da computação corresponde ou não a uma etapa da derivação; isto é, a uma transição por substituição. De fato, estas transições só podem ser aplicadas a configurações nas quais há uma variável no topo da pilha. Esta observação vai desempenhar um papel importante na demonstração de que nossa receita funciona.

3. Provando a receita

Como sempre a demonstração pode ser dividida em duas partes, que correspondem às inclusões $L(\mathcal{G}) \subseteq L(\mathcal{M})$ e $L(\mathcal{G}) \supseteq L(\mathcal{M})$.

Seja $\mathcal{G}$ é uma gramática livre de contexto formada pelos elementos (T, V, S, R) e seja w uma palavra gerada por $\mathcal{G}$. Queremos mostrar que w é aceita por $\mathcal{M}$. Suponhamos que conhecemos uma derivação mais à esquerda $S \Rightarrow^* w$ em $\mathcal{G}$. Devemos ser capazes de usar esta derivação para obter uma computação

$$(i, w, \epsilon) \vdash^* (f, \epsilon, \epsilon).$$

Além disso, sabemos que cada etapa desta computação deve corresponder a uma etapa da derivação, ou à aplicação de uma das transições por remoção, que apagam os prefixos de terminais que já tenham sido corretamente gerados na pilha de $\mathcal{M}$.

Digamos que, depois de k etapas, a derivação seja

$$S \Rightarrow^k \alpha X v,$$

onde estamos assumindo que $\alpha \in T^*$ e que $X \in V$. Em particular, X é a variável mais à esquerda de $\alpha X v$. Isto significa, por um lado, que w deve ser da forma $\alpha\beta$, para alguma palavra $\beta \in T^*$, e por outro que a próxima regra deve ser aplicada à variável X.

Se imaginarmos que já construímos a computação passo-a-passo até este ponto, devemos ter obtido

$$(i, w, \epsilon) \vdash (f, w, S) \vdash^* (f, \beta, Xv),$$

porque os terminais de α foram sendo eliminados na mesma medida em que foram produzidos, para que a computação pudesse avançar.

Suponhamos que a regra aplicada na próxima etapa da derivação seja $X \to u$, onde $u \in (T \cup V)^*$. Avançando mais esta etapa na derivação, obtemos

$$S \Rightarrow^k \alpha X v \Rightarrow \alpha u v.$$

Queremos reproduzir esta etapa na computação de $\mathcal{M}$. Como já chegamos a um ponto em que X está no topo da pilha, basta aplicar a transição $(f, u) \in \delta(f, \epsilon, X)$, que nos dá

$$(i, w, \epsilon) \vdash (f, w, S) \vdash^* (f, \beta, Xv) \vdash (f, \beta, uv).$$

Contudo isto não basta, porque queremos deixar a variável mais à esquerda de uv à descoberto, para poder aplicar a próxima regra sem obstáculos. Para isto precisamos localizar onde está esta variável, que vamos batizar de Y. Digamos que $uv = \gamma Y v'$, onde $\gamma \in T^*$. Como

$$\alpha u v = \alpha\gamma Y v' \Rightarrow w = \alpha\beta.$$

e γ só tem terminais, podemos concluir que γ é prefixo de β. Portanto, $\beta = \gamma\beta'$ e usando as regras de eliminação obtemos a computação

$$(i, w, \epsilon) \vdash^* (f, \beta, uv) = (f, \gamma\beta', \gamma Y v') \vdash^* (f, \beta', Yv'),$$

o que nos deixa prontos para prosseguir com a construção da computação exatamente como fizemos na etapa anterior.

Note que, continuando desta maneira, quando chegarmos ao final da derivação teremos obtido uma computação

$$(i, w, \epsilon) \vdash^* (f, \epsilon, \epsilon),$$

demonstrando, portanto, que $w \in L(\mathcal{M})$.

Provamos, assim, que $L(\mathcal{G}) \subseteq L(\mathcal{M})$, falta mostrar que a inclusão oposta também vale. Para isto, basta verificar que se w é aceita por $\mathcal{M}$ então existe uma derivação $S \Rightarrow^* w$.

A primeira impressão talvez seja de que, como a pilha de $\mathcal{M}$ reproduz uma derivação de w em $\mathcal{G}$, deve ser suficiente olhar para o que há na pilha em cada etapa da computação. Contudo, por causa das regras de remoção, cada vez que uma parte de w for gerada na pilha, ela será apagada antes da computação poder continuar. Portanto, para obter a etapa da derivação que corresponde a um dado passo da computação basta concatenar o prefixo de w que foi removido da entrada com o conteúdo da pilha. Em outras palavras, se a configuração de $\mathcal{M}$ em uma dada etapa da computação for (f, u, Yv) e $w = \alpha u$, então a etapa correspondente da computação será $\alpha Y v$.

O problema desta correspondência é que mais de uma etapa da computação pode corresponder a uma única etapa da derivação. Para evitar isto só vamos construir as etapas da derivação que correspondem a um passo da computação em que há uma variável no topo da pilha. Note que escolhemos estes passos porque é exatamente neles que se aplicam as transições por substituição.

Para concluir a demonstração precisamos nos convencer de que esta seqüência de palavras de $(T \cup V)^*$ é de fato uma derivação de w a partir de S. Há três coisas a verificar:

(1) a seqüência começa com S;
(2) para passar de uma palavra da seqüência para a próxima trocamos sua variável mais à esquerda pelo lado direito de uma regra de $\mathcal{G}$;
(3) a última palavra da seqüência é w.

Mas (1) e (2) são conseqüências imediatas da maneira como $\mathcal{M}$ foi definido, e (3) segue do fato de que a computação acaba na configuração (f, ϵ, ϵ). Provamos, portanto, que $w \in L(\mathcal{G})$.

4. Autômatos de pilha cordatos

Na próxima seção consideramos a recíproca do problema descrito na seção anterior. Isto é, descrevemos um algoritmo que, a partir de um autômato de pilha $\mathcal{M}$ constrói uma gramática livre de contexto $\mathcal{G}$ tal que $L(\mathcal{G}) = L(\mathcal{M})$.

Para tornar a construção da gramática a partir do autômato mais fácil, começaremos transformando o autômato de pilha não determinístico $\mathcal{M}$ dado. Construiremos a partir de $\mathcal{M}$ um autômato de pilha $\mathcal{M}'$ que aceita a mesma linguagem que $\mathcal{M}$, mas cujo comportamento é mais

predizível. Em particular, $\mathcal{M}'$ terá apenas um estado final e a pilha só poderá se esvaziar neste estado.

Para melhor sistematizar os algoritmos é conveniente introduzir a noção de um autômato de pilha *cordato*. Seja $\mathcal{N}$ um autômato de pilha com estado inicial i e função de transição δ. Dizemos que $\mathcal{N}$ é *cordato* se as seguintes condições são satisfeitas:

(1) a única transição a partir de i é $\delta(i, \epsilon, \epsilon) = \{(q, \beta)\}$, onde q é um estado de $\mathcal{N}$ e β é um símbolo do alfabeto da pilha;
(2) $\mathcal{N}$ tem um único estado final f;
(3) a pilha só se esvazia no estado final f;
(4) não há transições a partir de f.

Note que por causa de (1), (3) e (4), há um elemento $\beta \in \Gamma$ que nunca sai do fundo da pilha até que o estado final seja atingido. Além disto, (3) nos diz que se $\delta(q, \sigma, \beta) = (p, \epsilon)$ então $p = f$.

Vejamos como é possível construir a partir de um autômato de pilha $\mathcal{M}$ qualquer um autômato de pilha cordato $\mathcal{N}$ que aceita $L(\mathcal{M})$. Suponhamos que $\mathcal{M}$ é definido pelos ingredientes $(\Sigma, \Gamma, Q, q_1, F, \delta)$. Então $\mathcal{N}$ será o autômato de pilha com os seguintes ingredientes:

- alfabeto de entrada Σ;
- alfabeto da pilha $\Gamma \cup \{\beta\}$, onde $\beta \notin \Gamma$;
- conjunto de estados $Q \cup \{i, f\}$, onde $i, f \notin Q$;
- estado inicial i;
- conjunto de estados finais $\{f\}$;
- função de transição δ' definida de acordo com a tabela abaixo:

estado	entrada	topo da pilha	transição
i	ϵ	ϵ	(q_1, β)
$q \neq i, f$ e $q \notin F$	σ	γ	$\delta(q, \sigma, \gamma)$
$q \neq i, f$ e $q \in F$	σ	$\gamma \neq \epsilon$	$\delta(q, \sigma, \gamma)$
$q \neq i, f$ e $q \in F$	ϵ	β	$\delta(q, \epsilon, \epsilon) \cup \{(f, \epsilon)\}$

onde estamos supondo que $\sigma \in \Sigma \cup \{\epsilon\}$ e que $\gamma \in \Gamma \cup \{\epsilon\}$.

É claro que $\mathcal{N}$ é cordato; falta apenas mostrar que $L(\mathcal{N}) = L(\mathcal{M})$. Observe que o comportamento de $\mathcal{N}$ pode ser descrito suncintamente dizendo que, depois de marcar o fundo da pilha com β, o autômato simula o comportamento de $\mathcal{M}$. De fato, na primeira transição $\mathcal{N}$ apenas põe β no fundo da pilha e passa ao estado q_1 de $\mathcal{M}$. A partir daí $\mathcal{N}$ se comporta como $\mathcal{M}$ até que um estado final p de $\mathcal{M}$ é atingido. Neste caso, se a pilha contém apenas β, $\mathcal{N}$ tem a opção de entrar no estado f

e esvaziar a pilha. Observe que a pilha só é esvaziada no estado f, e isto obriga o autômato a parar porque não existem transições a partir de f.

Suponha agora que $\mathcal{N}$ computa a partir de uma entrada $w \in \Sigma^*$. Se w for aceita por $\mathcal{M}$, então existirá uma computação

$$(q_1, w, \epsilon) \vdash^* (p, \epsilon, \epsilon) \quad \text{em } \mathcal{M}.$$

Esta computação dará lugar a uma computação em $\mathcal{N}$ da forma

$$(i, w, \epsilon) \vdash (q_1, w, \beta) \vdash^* (p, \epsilon, \beta) \vdash (f, \epsilon, \epsilon).$$

Portanto se w for aceita por $\mathcal{M}$ será aceita por $\mathcal{N}$.

Por outro lado, uma computação

$$(i, w, \epsilon) \vdash^* (f, \epsilon, \epsilon), \tag{4.1}$$

implica que a configuração do autômato $\mathcal{N}$ anterior à última tinha que ser (p, ϵ, β), com p um estado final de $\mathcal{M}$. Assim, a computação (4.1) de $\mathcal{N}$ procede ao longo das seguintes etapas

$$(i, w, \epsilon) \vdash (q_1, w, \epsilon) \vdash^* (p, \epsilon, \beta) \vdash (f, \epsilon, \epsilon).$$

Temos assim que $(q_1, w, \epsilon) \vdash^* (p, \epsilon, \epsilon)$ é uma computação válida de $\mathcal{M}$. Como p é estado final de $\mathcal{M}$ concluímos que w é aceita por $\mathcal{M}$. Portanto $\mathcal{M}$ e $\mathcal{N}$ aceitam exatamente as mesmas palavras.

5. A gramática de um autômato de pilha

Suponha agora que $\mathcal{N}$ é um autômato de pilha cordato. Veremos como é possível construir uma gramática livre de contexto G que gera as palavras aceitas por $\mathcal{N}$. Digamos que $\mathcal{N}$ está definido pelos ingredientes $(\Sigma, \Gamma, Q, i, \{f\}, \delta)$. Além disso, suporemos que β é o símbolo que fica no fundo da pilha de $\mathcal{N}$ enquanto ele simula $\mathcal{M}$.

Naturalmente os terminais da gramática G serão os elementos de Σ. A construção das variáveis é mais complicada. Cada variável será indexada por três símbolos: dois estados e um símbolo da pilha. Como isto complica muito a notação é preferível identificar cada variável com a própria tripla que lhe serve de índice. Assim o conjunto de variáveis de G será

$$Q \times \Gamma \times Q.$$

Portanto uma variável de G será uma tripla (q, γ, p), onde q e p são estados de $\mathcal{N}$ e γ é um símbolo do alfabeto da pilha. Note que γ não pode ser ϵ.

Se

$$\delta(i, \epsilon, \epsilon) = \{(i', \beta)\},$$

então o símbolo inicial de G será (i', β, f). Precisamos agora construir as regras de G. Suponhamos que

$$(p, u) \in \delta(q, \sigma, \gamma)$$

onde $u \in \Gamma^*$. Há dois casos a considerar:

Primeiro caso: $u = \gamma_1 \cdots \gamma_k$.
Neste caso, para cada k-upla $(r_1, \ldots, r_k) \in Q^k$ contruímos uma regra

$$(q, \gamma, r_k) \to \sigma(p, \gamma_1, r_1)(r_1, \gamma_2, r_2) \cdots (r_{k-1}, \gamma_k, r_k).$$

Observe que isto nos dá, não apenas uma, mas n^k regras distintas, onde n é o número de estados de $\mathcal{N}$.

Segundo caso: $u = \epsilon$
Neste caso construímos apenas a regra

$$(q, \gamma, p) \to \sigma.$$

De nossa experiência anterior sabemos que, de alguma maneira, uma derivação mais à esquerda por G deve simular uma computação por $\mathcal{N}$. Mais precisamente,

(5.1)

$$(i, w, \epsilon) \vdash (i', w, \beta) \vdash^* (f, \epsilon, \epsilon) \text{ se, e somente se } (i', \beta, f) \Rightarrow^* w.$$

Entretanto, neste caso a simulação procede de maneira bem mais sutil. Para melhor identificar o problema, consideremos uma etapa da computação acima

$$(q, \sigma v, \gamma u) \vdash (p, w, \gamma_1 \cdots \gamma_k u),$$

onde $v \in \Sigma^*$ e $u \in \Gamma^*$. Evidentemente para que esta etapa seja legítima é preciso que

$$(p, \gamma_1 \cdots \gamma_k) \in \delta(q, \sigma, \gamma). \tag{5.2}$$

Mas (5.2) dá lugar a n^k regras: qual delas devemos escolher? Para decidir isto precisamos verificar como a computação continua.

Em outras palavras, para construir o i-ésimo passo da derivação não é suficiente considerar apenas o i-ésimo passo da computação, mas vários outros passos; até mesmo todo o restante da computação ! É claro que isto torna a demonstração da correspondência bem mais difícil que no caso de autômatos finitos. Felizmente podemos generalizar a equivalência (5.1) de modo a tornar a demonstração mais transparente. Para isso substituimos em (5.1) os estados i' e f por estados quaisquer p e q, e β por um símbolo qualquer X do alfabeto da pilha. Esta generalização é o conteúdo do seguinte lema.

LEMA 13.2. *Sejam p e q estados do autômato de pilha não determinístico $\mathcal{N}$. Então*

$$(q, w, X) \vdash^* (p, \epsilon, \epsilon) \text{ se, e somente se } (q, X, p) \Rightarrow^* w$$

na gramática $G(\mathcal{N})$.

De acordo com esta correspondência se o autômato está no estado p quando X é desempilhado, então a variável que inicia a derivação é (q, X, p). Assim, para achar a variável da partida é preciso olhar a computação até o último passo!

A demonstração é por indução finita, e para torná-la mais clara vamos dividi-la em duas partes.

Primeira parte: Queremos mostrar a afirmação

$$(\mathbf{A}(m)) \qquad \text{se } (q, w, X) \vdash^m (p, \epsilon, \epsilon), \text{ então } (q, X, p) \Rightarrow^* w.$$

por indução em m.

Se $m = 1$, então

$$w = \sigma \in \Sigma \cup \{\epsilon\} \text{ e } (p, \epsilon) \in \delta(q, \sigma, X).$$

Mas, por construção, isto significa que na gramática $G(\mathcal{N})$ há uma regra do tipo

$$(q, X, p) \to \sigma.$$

Como, neste caso, toda a derivação se reduz a uma aplicação desta regra, a base da indução está provada.

Suponhamos, agora, que $s > 1$ é um número inteiro, e que $\mathbf{A}(m)$ vale para todo $m < s$. Seja

$$(q, w, X) \vdash^s (p, \epsilon, \epsilon) \tag{5.3}$$

uma computação em $\mathcal{N}$ com s etapas.

Isolando o primeiro símbolo de w, podemos escrever $w = \sigma v$, onde $\sigma \in \Sigma$ e $v \in \Sigma^*$. Neste caso o primeiro passo da computação (5.3) será da forma

$$(q, \sigma v, X) \vdash (q^1, v, Y_1 \cdots Y_k),$$

que corresponde à transição

$$(q^1, v, Y_1 \cdots Y_k) \in \delta(q, \sigma, X). \tag{5.4}$$

Note que, ao final da computação (5.3) cada Y foi desempilhado. Lembre-se que dizer, por exemplo, que Y_1 é removido da pilha significa que a pilha passou a ser $Y_2 \cdots Y_k$. Isto não tem que acontecer em apenas uma transição. Assim, Y_1 pode ser trocado por uma palavra no alfabeto da pilha sem que seja necessariamente desempilhado. Por isso, durante os passos seguintes da computação a pilha pode crescer bastante antes

de diminuir ao ponto de ser apenas $Y_2 \cdots Y_k$. Digamos então que v_1 é o prefixo de v que é consumido para que Y_1 seja desempilhado. Se $v = v_1 v$, temos uma computação

$$(q^1, v_1 v', Y_1 \cdots Y_k) \vdash^* (q^2, v', Y_2 \cdots Y_k),$$

onde q^2 é algum estado de $\mathcal{N}$.

Analogamente, sejam $v_2, v_3, \ldots, v_k$ as subpalavras de v que têm que ser consumidas para que $Y_2, \ldots, Y_k$ sejam desempilhados. Assim, $v = v_1 v_2 \ldots v_k$, e existem estados $q^1, \ldots, q^k$ de $\mathcal{N}$ modo que (5.3) pode ser subdividida em $k-1$ etapas da forma

$$(q^i, v_i \cdots v_k, Y_i \cdots Y_k) \vdash^* (q^{i+1}, v_{i+1} \cdots v_k, Y_{i+1} \cdots Y_k),$$

seguidas de uma etapa final da forma

$$(q^k, v_k, Y_k) \vdash^* (p, \epsilon, \epsilon).$$

Como cada uma destas computações tem menos de s passos, segue da hipótese de indução, que

$$(q^i, v_i, Y_i) \vdash^* (q^{i+1}, \epsilon, \epsilon),$$

dá lugar à derivação

$$(q^i, Y_i, q^{i+1}) \Rightarrow^* v_i, \tag{5.5}$$

ao passo que a etapa final da computação corresponde a

$$(q^k, Y_k, p) \Rightarrow^* v_k. \tag{5.6}$$

Falta-nos apenas reunir de modo ordenado tudo o que fizemos até agora. Em primeiro lugar, a transição (5.4) dá lugar a uma regra de $G(\mathcal{N})$ da forma

$$(q, X, p) \to \sigma(q^1, Y_1, q^2) \cdots (q^k, Y_k, p).$$

Substituindo, finalmente, as derivações de (5.5) e (5.6) nos lugares apropriados, obtemos

$$(q, X, p) \Rightarrow \sigma(q^1, Y_1, q^2) \cdots (q^k, Y_k, p) \Rightarrow^* \sigma v_1 \cdots v_k = w,$$

como queríamos.

Segunda parte: Queremos mostrar a afirmação

$$(\mathbf{B}(m)) \qquad \text{se} \quad (q, X, p) \Rightarrow^m w \ \text{ então } \ (q, w, X) \vdash^* (p, \epsilon, \epsilon)$$

por indução em m

Se $m = 1$ então a derivação se resume a à regra

$$(q, X, p) \to w,$$

da gramática $G(\mathcal{N})$. Mas, pela construção da gramática, w tem que ser um símbolo $\sigma \in \Sigma \cup \{\epsilon\}$. Além disto, esta regra só pode ocorrer se houver uma transição da forma

$$(p, \epsilon) \in \delta(q, \sigma, X)$$

em $\mathcal{N}$. Mas esta transição dá lugar à computação

$$(q, \sigma, X) \vdash (p, \epsilon, \epsilon)$$

que esperávamos obter.

Suponha, agora, que $s > 1$ é um número inteiro e que o resultado vale para toda derivação com menos de s passos. Seja

$$(q, X, p) \Rightarrow^s w \tag{5.7}$$

uma derivação de $G(\mathcal{N})$ com s etapas. Como $s > 1$, a primeira regra a ser aplicada nesta derivação deve ser da forma

$$(q, X, p) \to \sigma(s_1, Y_1, s_2)(s_2, Y_2, s_3) \cdots (s_k, Y_k, p), \tag{5.8}$$

onde $(s_1, \ldots, s_k) \in Q^k$ e σ é o símbolo mais à esquerda de w. Digamos que $w = \sigma v$, para alguma palavra $v \in \Sigma^*$. Para que a derivação possa acabar produzindo w deve ser possível decompor v na forma $v = v_1 \cdots v_k$ de modo que, para $i = 1, \ldots, k$, temos

$$(s_i, Y_i, s_{i+1}) \Rightarrow^* v_i$$

onde $s_{k+1} = p$. Como a derivação de v_is tem que ter um número de etapas menor que s, podemos aplicar a hipótese de indução, obtendo assim computações

$$(s_i, v_i, Y_i) \vdash^* (s_{i+1}, \epsilon).$$

para $i = 1, \ldots, k$. Além disso, se pusermos $Y_{i+1}, \ldots, Y_k$ abaixo de Y_i, no fundo da pilha, obteremos

$$(s_i, v_i, Y_i Y_{i+1} \cdots Y_k) \vdash^* (s_{i+1}, \epsilon).$$

Agora, (5.8) provém da transição

$$(p, Y_1 \cdots Y_k) \in \delta(q, \sigma, X),$$

que pode ser reescrita na forma

$$(q, \sigma, X) \vdash (p, Y_1 \cdots Y_k).$$

Lembrando que $w = \sigma v_1 \cdots v_k$, vemos que $\mathcal{N}$ admite uma computação da forma

$$(q, w, X) \vdash (s_1, v_1 \cdots v_k, Y_1 \cdots Y_k) \vdash^* (s_2, v_2 \cdots v_k, Y_2 \cdots Y_k) \vdash^* \cdots \vdash^* (s_k, v_k, Y_k) \vdash (p, \epsilon, \epsilon),$$

como queríamos mostrar.

6. Exercícios

1. Construa a computação no autômato descrito na seção 2 que corresponde à derivação mais à esquerda de $\mathrm{id} * (\mathrm{id} + \mathrm{id})$ na gramática $\mathcal{G}'_{exp}$.

2. Ache um autômato de pilha não determinístico que aceita a linguagem gerada pela gramática cujas regras são:
$$S \to 0AA$$
$$A \to 1S|0S|0$$

3. Considere a linguagem dos parênteses balanceados descrita no exercício 2 do capítulo ??.
   (1) Dê exemplo de uma gramática livre de contexto que gere esta linguagem.
   (2) Dê exemplo de um autômato de pilha não determinístico que aceita esta linguagem.

4. Para cada uma das linguagens L, abaixo, invente uma gramática livre de contexto que gere L e use a receita da seção 2 para construir um autômato de pilha não determinístico que aceite L.
   (a) $L = \{wc^4w^r : w \in \{0,1\}\}$;
   (b) $L = \{a^n b^m c : n \geq m \geq 1\}$;
   (c) $L = \{0^m 1^n : n \leq m \leq 2n\}$;
   (d) $L = \{a^{i+3} b^{2i+1} : i \geq 0\}$;
   (e) $L = \{a^i b^j c^j d^i e^3 : i, j \geq 0\}$.

5. Seja $\mathcal{G}$ uma gramática livre de contexto cujo símbolo inicial é S, e seja $\mathcal{M}$ o autômato de pilha construído a partir de $\mathcal{G}$ pela receita da seção 2. Suponhamos que w é uma palavra de comprimento k em $L(\mathcal{G})$. Determine o número de passos da computação de $\mathcal{M}$ que corresponde à derivação mais à esquerda $S \Rightarrow^n w$.

CAPíTULO 14

Máquinas de Turing

1. Exercícios

1. Considere a máquina de Turing cujo alfabeto é $\{a, b, \sqcup, \triangleright\}$, conjunto de estados $\{q_0, q_1, h\}$, estado inicial q_0 e transições dadas pela tabela:

estado	entrada	transições
q_0	0	$(q_1, 1)$
	1	$(q_1, 0)$
	$\sqcup$	$(h, \sqcup)$
	$\triangleright$	$(q_0, \rightarrow)$
q_1	0	$(q_0, \rightarrow)$
	1	$(q_0, \rightarrow)$
	$\sqcup$	$(q_0, \rightarrow)$
	$\triangleright$	$(q_1, \rightarrow)$

   (a) Descreva a computação de M a partir da configuração $(q_0, \triangleright\underline{0}01110)$.
   (b) Descreva informalmente o que M faz quando iniciada no estado q_0 e em alguma casa de sua fita.

2. Descreva a tabela de transição de uma máquina de Turing, no alfabeto $\{a, b, \sqcup, \triangleright\}$, que se move para a esquerda até encontrar três as na fita e então pára.

3. Explique porque as máquinas DE e ED nem sempre têm a mesma saída.

4. Esboce o esquema de máquinas de Turing que aceitem as seguintes linguagens:

(a) 010^*1;
(b) $\{w \in \{0,1\}^* : |w| \text{ é par}\}$;
(c) $\{a^n b^n c^m : m \geq n\}$;
(d) $\{w \in \{0,1\} : w = w^r\}$;
(e) $\{0^{n^2} : n \geq 1\}$.

5. Construa, a partir do diagrama dado em aula, a tabela de transições da máquina de Turing M_e que move uma palavra $w \in (0 \cup 1)^*$, precedida de uma casa vazia, uma casa para a esquerda; isto é, que transforma $\sqcup w$ em w.

6. Construa máquinas de Turing que calculem as seguintes funções $f : \mathbb{N} \to \mathbb{N}$ definidas por:
   (a) $f(n) = n + 1$;
   (b) $f(n)$ é o resto da divisão de n por 2;
   (c)
   $$f(n,m) = \begin{cases} n - m & \text{se } n - m \geq 0 \\ 0 & \text{se } n < m. \end{cases}$$

7. Descreva uma máquina de Turing que, tendo como entrada uma palavra $w \in \{0,1\}^*$ encontra o símbolo do meio da palavra (se existir!).

8. Descreva uma máquina de Turing que, tendo como entrada uma palavra $w \in \{0,1\}^*$ com comprimento par, substitui os 0s por a ou c e os 1s por bs ou ds, de modo que a palavra fica escrita na forma $w_1 w_2$ onde $w_1 \in \{a,b\}^*$ e $w_2 \in \{c,d\}^*$.

9. Utilizando a máquina de Turing da questão anterior construa uma máquina de Turing que aceita a linguagem $\{ww : w \in \{0,1\}^*\}$.

10. Mostre que a linguagem $\{ww : w \in \{0,1\}^*\}$ é recursiva.

11. Construa a fita de entrada para que a máquina de Turing universal simule a computação da máquina de Turing do exercício 1 a partir da configuração $(q_0, \triangleright \underline{0}01110)$.

12. Sejam L_1 e L_2 linguagens recursivas aceitas por máquinas de Turing M_1 e M_2, respectivamente. Mostre como construir uma máquina de Turing $\mathcal{M}$ que aceite a linguagem $L_1 \cup L_2$.

13. A intersecção de linguagens recursivas é recursiva? Explique sua resposta.

14. Dê a definição formal de uma máquina de Turing cuja fita é duplamente infinita (isto é, vai de $-\infty$ a $+\infty$). Mostre como é possível simular uma máquina destas usando uma máquina de Turing $\mathcal{M}$ cuja

fita é infinita somente à direita. As máquinas definidas originalmente por Alan Turing tinham fitas duplamente infinitas.
Sugestão: Escolha um ponto de referência na fita duplamente infinita e escreva os símbolos das casas à direita do referencial nas casas pares da fita de $\mathcal{M}$, e aqueles que estão à esquerda nas casas ímpares. Explique como deve ser o comportamento de $\mathcal{M}$. Note que $\mathcal{M}$ chega a $\triangleright$ quando a máquina original cruza o ponto de referência. Qual vai ser o comportamento de $\mathcal{M}$ neste caso?

15. Seja Σ_0 um alfabeto e L uma linguagem no alfabeto Σ_0. Mostre que, se L e $\Sigma_0 \setminus L$ são recursivamente enumeráveis, então L é recursiva.

16. Seja Σ_0 um alfabeto e L uma linguagem no alfabeto Σ_0 que é recursivamente enumerável mas não é recursiva. Suponha que M é uma máquina de Turing que aceita L. Mostre que existe uma quantidade infinita de palavras em Σ_0 que não é aceita por M.

CAPíTULO 15

Máquinas de Turing e Linguagens

Neste capítulo discutimos várias propriedades das linguagens recursivas e recursivamente enumeráveis. Entretanto, o principal resultado do capítulo é o fato de que existe uma máquina de Turing que é capaz de simular qualquer outra máquina de Turing. Esta máquina é conhecida como *máquina de Turing Universal*, e foi originalmente construída por Alan Turing em 1937.

O capítulo começa com a descrição de uma construção que permite conectar duas máquinas de Turing *em paralelo*, e se encerra com o *Problema da Parada*.

1. Conectando Máquinas em Paralelo

Suponhamos que $M = (\Sigma, Q, q_1, F, \delta)$ e $M' = (\Sigma, Q', q_1', F', \delta')$ são duas máquinas de Turing no alfabeto Σ. Defina um novo alfabeto por

$$\widetilde{\Sigma} = \Sigma \cup \{\underline{\sigma} : \sigma \in \Sigma\}.$$

Quer dizer, para cada símbolo $\sigma \in \Sigma$, existem dois símbolos em $\widetilde{\Sigma}$: o próprio σ e $\underline{\sigma}$. A nova máquina $\mathcal{M}$ é constituída pelos seguintes elementos:

ALFABETO: $\widetilde{\Sigma}^2 \cup \{\triangleright, \sqcup\}$.

CONJUNTO DE ESTADOS: $\mathcal{Q} \supseteq Q \times Q'$.

ESTADO INICIAL: (q_1, q_1'),

ESTADOS FINAIS: $H \subseteq Q \times Q'$ que depende do que eu quero que a máquina faça.

Antes de discutir as transições, precisamos explicar como é a fita da máquina. Como sempre, a extremidade esquerda da fita está marcada pelo símbolo $\triangleright$. Já a casa imediatamente à direita desta sempre conterá

$$(\triangleright, \triangleright) \text{ ou } (\underline{\triangleright}, \triangleright) \text{ ou } (\triangleright, \underline{\triangleright}) \text{ ou } (\underline{\triangleright}, \underline{\triangleright}).$$

Além disso, a máquina que estamos construindo só poderá imprimir um destes símbolos nesta casa.

TRANSIÇÃO: Vamos descrever o comportamento da máquina na forma de um ciclo completo que ela fica repetindo até parar. No começo do ciclo, $\mathcal{M}$ está no estado $(q, q') \in Q \times Q'$.

A máquina $\mathcal{M}$ precisa lembrar que está simulando M a partir do estado q e M' a partir do estado q', até chegar na etapa 5. Nesta etapa haverá uma mudança nos estados a partir dos quais a simulação está sendo feita. Estes novos estados serão lembrados até chegar à etapa 8. Como isto é feito será discutido ao final da descrição da transição.

ETAPA 1: Se $(q, q') \in H$ a máquina pára.

ETAPA 2: Lembrando que está simulando M a partir de q e M' a partir de q', a máquina $\mathcal{M}$ recua seu cabeçote para a esquerda até encontrar $\triangleright$.

ETAPA 3: Lembrando que está simulando M a partir de q e M' a partir de q', a máquina $\mathcal{M}$ move o cabeçote para a direita até encontrar um símbolo da forma $(\underline{\sigma}, \sigma')$.

ETAPA 4: Lembrando que está simulando M a partir de q e M' a partir de q', a máquina $\mathcal{M}$ atualiza a fita e os estados que está simulando de M e M' conforme a tabela abaixo

$\delta(q, \sigma)$	Novo estado de M	Novo estado de M'	Fita
(p, τ)	p	q'	imprime $(\underline{\tau}, \sigma')$ sem mover cabeçote
$(p, \rightarrow)$	p	q'	imprime (σ, σ') e move para a direita: • se está lendo (α, α'), imprime $(\underline{\alpha}, \alpha')$ • se está lendo $\sqcup$, imprime $(\underline{\sqcup}, \sqcup)$
$(p, \leftarrow)$	p	q'	imprime (σ, σ') e move para a esquerda: • se está lendo (α, α'), imprime $(\underline{\alpha}, \alpha')$ • se está lendo $\sqcup$, imprime $(\underline{\sqcup}, \sqcup)$

Se, na última linha da tabela, $(\sigma, \sigma') = (\triangleright, \triangleright)$ então a máquina será forçada a mover o cabeçote para a direita, efetuando a mudança de estados correspondente em M.

Ao final desta etapa os estados M e M' que estão sendo lembrados por $\mathcal{M}$ passam a ser p e q'.

ETAPA 5: Se $(p, q') \in H$ a máquina pára.

ETAPA 6: Lembrando que está simulando M a partir de p e M' a partir de q', a máquina $\mathcal{M}$ recua seu cabeçote para a esquerda até encontrar $\triangleright$.

ETAPA 7: Lembrando que está simulando M a partir de p e M' a partir de q', a máquina $\mathcal{M}$ move o cabeçote para a direita até encontrar um símbolo da forma $(\sigma, \underline{\sigma'})$.

ETAPA 8: Lembrando que está simulando M a partir de p e M' a partir de q', a máquina $\mathcal{M}$ atualiza a fita e os estados que está simulando de M e M' conforme a tabela abaixo

$\delta(q', \sigma')$	Novo estado de M	Novo estado de M'	Fita
(p', τ')	p	p'	imprime $(\sigma, \underline{\tau'})$ sem mover cabeçote
$(p', \rightarrow)$	p	p'	imprime (σ, σ') e move para a direita: • se está lendo (α, α'), imprime $(\alpha, \underline{\alpha'})$ • se está lendo $\sqcup$, imprime $(\sqcup, \underline{\sqcup})$
$(p', \leftarrow)$	p	p'	imprime (σ, σ') e move para a esquerda • se está lendo (α, α'), imprime $(\alpha, \underline{\alpha'})$ • se está lendo $\sqcup$, imprime $(\sqcup, \underline{\sqcup})$

Naturalmente, se, na última linha da tabela, $(\sigma, \sigma') = (\triangleright, \triangleright)$ então a máquina será forçada a mover o cabeçote para a direita, efetuando a mudança de estados correspondente em M.

Ao final desta transição a máquina entra o estado (p, p') e o ciclo é reinicializado.

Falta somente explicar como é que a máquina lembra que está simulando M a partir de q e M' a partir de q'. Para isto, para cada par (q, q'), construímos um conjunto de estados associado a este par que a máquina deverá entrar ao executar as etapas 1, 2, 3 e 4 do ciclo, e outro conjunto de estados que corresponderá às etapas 5, 6, 7 e 8 do ciclo. Em outras palavras, a máquina lembra quais são os estados q e q' a partir dos quais está simulando M e M' por causa dos estados que está vendo.

2. Fechamento de linguagens

Estamos prontos para usar a máquina construída na seção anterior para discutir como é o comportamento das linguagens recursivas e recursivamente enumeráveis com respeito às operações de união e intersecção.

Para começar, digamos que L é uma linguagem aceita por uma máquina de Turing $M = (\Sigma, Q, q_1, F, \delta)$ e L' por $M' = (\Sigma, Q', q_1', F', \delta')$. Assim, estamos supondo que L e L' são recursivamente enumeráveis.

A partir destas máquinas construiremos uma máquina $\mathcal{M}$, como na seção 1, tomando $H = (F \times Q') \cup (Q \times F')$. O efeito desta escolha de estados finais é fazer com que esta mąquina páre quando M ou M' atinge um estado final.

Seja $w = \sigma_1 \cdots \sigma_n$ uma palavra de Σ^*. Daremos a $\mathcal{M}$ a entrada

$$(\triangleright, \triangleright)(\sqcup, \sqcup)(\sigma_1, \sigma_1) \cdots (\sigma_n, \sigma_n). \tag{2.1}$$

Por causa de nossa escolha de estados finais, a computação de $\mathcal{M}$ a partir desta palavra vai parar se, e somente se, $w \in L$ ou $w \in L'$. Portanto, a linguagem aceita por $\mathcal{M}$, sob esta escolha de estados finais é $L \cup L'$. Concluímos, assim, que se L e L' são linguagens recursivamente enumeráveis, então $L \cup L'$ é recursivamente enumerável.

Também é verdade que se L e L' linguagens recursivamente enumeráveis, então $L \cap L'$ é recursivamente enumerável. Fica como exercício para você, determinar qual a escolha de conjunto de estados finais para a qual $\mathcal{M}$ aceita $L \cap L'$.

Suponhamos, agora, que L e L' são linguagens recursivas. Sejam M e M', definidas como acima, de modo que L seja decidida por M e L' por M'. Lembre-se que, neste caso, $F = F' = \{s, n\}$. Escolheremos

$$H = F \times F'.$$

Dando a $\mathcal{M}$ uma entrada como (2.1), verificamos que esta máquina só pára no estado (n, n) se $w \notin L$ e $w \notin L'$. Por outro lado, se $w \in L$ ou $w \in L'$, então $\mathcal{M}$ pode parar em um dos estados (s, n), (n, s) ou (s, s). Isto não é satisfatório, porque se $w \in L \cup L'$ esperaríamos que $\mathcal{M}$ parasse sempre no mesmo estado. Para curar este problema, basta construir uma máquina $\mathcal{N}$ que executa $\mathcal{M}$ como explicado acima e que, terminada a computação de $\mathcal{M}$, altera o estado para (s, s) se $\mathcal{M}$ alcançou um dos estados de aceitação (s, n), (n, s) ou (s, s). Verificamos, portanto, que se L e L' são linguagens recursivas, então $L \cup L'$ também é recursiva.

Deixamos aos seus cuidados escolher os estados finais de $\mathcal{M}$ e efetuar as mudanças necessárias para provar que se L e L' são linguagens recursivas, então $L \cap L'$ também é recursiva.

De todas as operações básicas com conjuntos só não analisamos ainda o que ocorre com o complementar. Neste ponto as duas classes de linguagens divergem radicalmente. Suponhamos, para começar, que L seja uma linguagem recursiva no alfabeto Σ_0 e que M é uma máquina

de Turing, em um alfabeto que contém Σ_0, e que decide L. Então, o conjunto de estados finais de M é $\{s, n\}$ e, dado $w \in \Sigma_0^*$, temos que

- $w \in L$ se e só se M pára no estado s;
- $w \in \overline{L} = \Sigma_0^* \setminus L$ se e só se M pára no estado n.

Para construir uma mquina $\overline{M}$ que decida $\overline{L}$, basta alterar as transições de M de modo que quando M entra o estado s, então $\overline{M}$ entra o estado n; e vice-versa. Logo se L é recursiva, então $\overline{L}$ também é.

Infelizmente, uma estratégia deste tipo não vai funcionar quando L for apenas recursivamente enumerável porque, neste caso, a máquina que aceita L simplesmente não pára se a palavra estiver fora de L. Veremos, na seção 5, que o complementar de uma linguagem recursivamente enumerável não tem que ser recursivamente enumerável. Por hora, resumiremos tudo o que vimos nesta seção em um teorema.

TEOREMA 15.1. *Sejam L e L' linguagens em um alfabeto Σ_0.*

(1) *Se L e L' são recursivamente enumeráveis, então $L \cup L'$ e $L \cap L'$ também são.*
(2) *Se L e L' são recursivas, então $L \cup L'$ e $L \cap L'$ também são.*
(3) *Se L for recursiva, então $\overline{L} = \Sigma_0^* \setminus L$ também é.*

3. A máquina de Turing universal

Nesta seção descrevemos uma construção para a *máquina de Turing universal* $\mathcal{U}$. Trata-se de uma máquina de Turing capaz de simular qualquer outra máquina de Turing. Naturalmente, para que isto seja possível, a máquina universal precisa receber como entrada o "programa" da máquina que está sendo simulada. Em outras palavras, precisamos ser capazes de descrever o alfabeto, o conjunto de estados e as transições de uma máquina de Turing qualquer a partir do alfabeto de $\mathcal{U}$.

Há muitas maneiras diferentes de construir a máquina $\mathcal{U}$. Na construção que faremos o alfabeto será

$$\Sigma = \{\triangleright, 0, 1, \sigma, q, X, Y, Z, \star, \sqcup, a, b\}.$$

Descreveremos as transições de uma máquina de Turing M na fita de $\mathcal{U}$ enumerando (separadamente) os símbolos e os estados de M no sistema unário. Para distinguir símbolos de estados vamos adicionar aos símbolos o prefixo σ e aos estados o prefixo q. Para os propósitos desta descrição, é conveniente representar as setas $\rightarrow$ e $\leftarrow$ como se fossem símbolos de M.

Digamos que a máquina de Turing M que desejamos simular usando $\mathcal{U}$ tem n estados e um alfabeto com m símbolos. Para que o número de casas da fita de $\mathcal{U}$ ocupadas pela descrição de um símbolo de M seja

sempre o mesmo, reservaremos $m+3$ casas para representar cada símbolo, preenchendo as casas redundantes com 1s. Mas se o alfabeto de M só tem m símbolos, por que precisamos de $m+3$ casas? Em primeiro lugar, estamos considerando as setas como símbolos 'honorários', o que dá conta de duas, das três casas extra. A outra casa é usada para pôr o prefixo σ, que indica que a seqüência de 0s e 1s que vem a seguir é um símbolo, e não um estado de M.

Procederemos de maneira análoga para os estados, reservando neste caso $n+1$ casas para cada estado. Assim, o r-ésimo símbolo do alfabeto de M será denotado por $\sigma 0^r 1^{m+2-r}$, e o r-ésimo estado de M por $q0^r1^{n-r}$. Lembre-se que nesta descrição estão incluídas $\rightarrow$ e $\leftarrow$. Por razões mnemônicas, convencionaremos que $\rightarrow$, $\leftarrow$ e $\triangleright$, serão sempre os três primeiros símbolos a serem enumerados. Assim,

símbolo	código
$\rightarrow$	$\sigma 01^{m+1}$
$\leftarrow$	$\sigma 0^2 1^m$
$\triangleright$	$\sigma 0^3 1^{m-1}$

Sejam M uma máquina de Turing e w uma palavra que deverá servir de entrada para M. Para dar início à simulação de M por $\mathcal{U}$ precisamos preparar a fita de $\mathcal{U}$ com os dados de M. Consideremos, em primeiro lugar, a maneira de codificar uma transição de M da forma $\delta(q_i, \sigma_j) = (q_r, \sigma_s)$. Se $\sigma_s \in \Sigma \cup \{\rightarrow, \leftarrow\}$, então esta transição corresponderá a um segmento da fita da forma

X	q	0^i	1^{n-i}	σ	0^j	1^{m+2-j}	q	0^r	1^{n-r}	σ	0^s	1^{m+2-s}	X

Note que o 0^i representa na verdade um segmento de i casas da fita, e o mesmo vale para 1^i.

A descrição completa de M e w na fita de entrada de $\mathcal{U}$ será feita segundo o modelo abaixo, onde S_i denota um segmento, da forma descrita acima, entre os dois Xs.

$\triangleright$	X	S_1	X	S_2	$\cdots$	X	S_t	Y	Q	Z	σ	u_1	σ	u_2	$\star$	u_3	$\cdots$

Nesta fita temos que:

De	Até	Conteúdo do segmento
$\triangleright$	Y	descrição do comportamento de M
Y	Z	$Q = q0^i1^{n-i}$ é o estado em que M se encontra no estágio atual da computação
Z	final	w onde cada símbolo está escrito em unário e precedido de σ

Note que o número unário que descreve um dos símbolos da entrada da máquina M na fita acima está precedido de $\star$ e não de σ. A $\star$ serve para marcar o símbolo atualmente lido por M na simulação por $\mathcal{U}$. Assim, quando a fita é preparada para a entrada de $\mathcal{U}$, o segmento Q é preenchido com o estado inicial de M e o símbolo $\star$ ocupa a segunda casa à direita do Z (porque a segunda e não a primeira?). A sucessão de símbolos do trecho da fita de $\mathcal{U}$ que contém a descrição da máquina M é uma palavra no alfabeto Σ. Vamos denotar esta palavra por $c(M)$.

Exemplo. Considere a máquina de Turing com alfabeto $\{0, \sqcup, \triangleright\}$, conjunto de estados $\{q_0, q_1, h\}$, estado inicial q_0, conjunto de estados finais $\{h\}$ e tabela de transição

estado	entrada	transições
q_0	0	$(q_1, \sqcup)$
	$\sqcup$	$(h, \sqcup)$
	$\triangleright$	$(q_0, \rightarrow)$
q_1	0	$(q_0, 0)$
	$\sqcup$	$(q_0, \rightarrow)$
	$\triangleright$	$(q_1, \rightarrow)$

Neste caso os símbolos e estados de M são codificados como abaixo. Lembre-se que, neste exemplo, deve haver 4 casas da fita para cada estado e para cada símbolo (já que há 3 estados e 3 símbolos), sendo que as casas redundantes devem ser preenchidas com 1s.

estado	código	símbolo	código
q_0		$\rightarrow$	
q_1		$\leftarrow$	
h		$\triangleright$	
$*$	$*$	$\sqcup$	
$*$	$*$	0	

Exercício. Descreva a fita de $\mathcal{U}$ correspondente a esta máquina com entrada $000\sqcup$.

4. Comportamento de $\mathcal{U}$

O procedimento da máquina de Turing universal é constituído de duas partes: na primeira a máquina verifica se a fita contém uma descrição legítima de alguma máquina de Turing; na segunda $\mathcal{U}$ simula a máquina cuja descrição lhe foi dada.

Na primeira parte do processamento da entrada, $\mathcal{U}$ verifica que a fita que recebeu contém uma descrição legítima de máquina de Turing. Para fazer isto, $\mathcal{U}$ varre a fita a partir de $\triangleleft$ e verifica que os Xs, qs, σs e Zs estão posicionados na ordem correta. À medida que faz isto, $\mathcal{U}$ utiliza a máquina *contagem* para comparar o bloco de 0s e 1s entre q e σ com o bloco correspondente que vem depois do próximo X; e faz o mesmo com o bloco entre σ e X. Finalmente, $\mathcal{U}$ compara o bloco entre os dois Zs com o bloco entre o primeiro q e o primeiro σ para ver se têm o mesmo número de casas. Se alguma destas condições não é verificada $\mathcal{U}$ caminha para a direita pela fita sem parar.

A segunda parte do processamento de $\mathcal{U}$ pode ser descrito como um ciclo com 3 etapas:

Primeira Etapa: No início o cabeçote está parado sobre o Z. Então entra em ação uma máquina do tipo *localiza* que, a partir de $\triangleright$ localiza uma quádrupla que comece com o par (Q, τ), onde τ é o símbolo à direita do Z marcado com $\star$ e Q é o estado entre Y e Z. À medida que vai tentando achar este par, a máquina vai trocando os 0s e 1s (da esquerda para a direita) por as e bs, respectivamente. Isto permite identificar até onde já foi feita a busca. Ao achar o par desejado, a máquina também troca os 0s e 1s deste par por as e bs. Com isto o $q0^i1^{n-i}$ mais à esquerda da fita corresponde ao estado seguinte a Q na transição de M por τ. Note que se $\mathcal{U}$ não conseguir localizar a transição desejada então o estado atual de M é final. Portanto, neste caso, $\mathcal{U}$ deve entrar em um estado final, o que a faz parar também.

Segunda Etapa: Agora *copie* o estado seguinte ao atual na transição de M a partir de Q e τ no campo que corresponde ao estado atual de M. Troque os números por letras também neste estado. Considere agora o σ seguido de algum número que fica mais à esquerda da fita: ele nos diz o que M faria com sua fita de entrada nesta transição. O comportamento de $\mathcal{U}$ está resumido na tabela abaixo:

Acha	Faz
$\sigma 0^1 1^{m+1}$	move o $\star$ para o lugar do σ seguinte
$\sigma 0^2 1^m$	move o $\star$ para o lugar do σ anterior
$\sigma 0^j 1^{m+2-j}$ e $j \geq 3$	substitui o segmento entre $\star$ e σ por $0^j 1^{m+2-j}$

Terceira Etapa: Agora $\mathcal{U}$ volta o cabeçote até $\triangleright$ e move-se para à direita trocando todos os as e bs por 0s e 1s, respectivamente até chegar ao segundo σ depois da $\star$ (por quê?). Finalmente o cabeçote volta a Z. Assim $\mathcal{U}$ está pronta para começar um novo ciclo.

5. Linguagens não recursivas

Seja M uma máquina de Turing no alfabeto Σ. Já vimos que podemos simular M usando a máquina universal $\mathcal{U}$. Para fazer isto, precisamos preparar a fita da máquina $\mathcal{U}$ de modo a codificar o comportamento de M. A descrição de M vem no começo da vida e está inteiramente contida na palavra $c(M) \in \Sigma^*$.

Considere agora a linguagem $\mathcal{L}$ formada pelas palavras de Σ^* que representam corretamente a descrição de uma máquina de Turing no alfabeto Σ. Em outras palavras

$$\mathcal{L} = \{c(M) : M \text{ é uma máquina de Turing no alfabeto } \Sigma\}.$$

A linguagem que desejamos considerar é um subconjunto de $\mathcal{L}$:

$$\mathcal{L}_0 = \{c(M) : \text{a máquina } M \, aceita \quad c(M)\}.$$

Por mais exótica que possa parecer, $\mathcal{L}_0$ é uma linguagem recursivamente enumerável. Para ver isto basta exibir uma máquina de Turing que aceita $\mathcal{L}_0$. Seja $\mathcal{C}$ a máquina de Turing que, tendo como entrada $\triangleleft w \sqcup$, produz a saída $\triangleleft w Z w \sqcup$. Então a linguagem aceita por $\mathcal{C} \cdot \mathcal{U}$ é $\mathcal{L}_0$. Observe que se $w \in \Sigma^*$ não é uma descrição legítima de alguma máquina de Turing, então wZw é automaticamente rejeitada por $\mathcal{U}$ no momento da checagem inicial da entrada.

Diante do que acabamos de ver a seguinte pergunta imediatamente se impõe.

PERGUNTA. *$\mathcal{L}_0$ é uma linguagem recursiva?*

Se $\mathcal{L}_0$ fosse recursiva, seu complemento $\overline{\mathcal{L}_0}$ também seria. Surpreendentemente isto está longe de ser verdade.

TEOREMA 15.2. *$\overline{\mathcal{L}_0}$ não é uma linguagem recursivamente enumerável.*

Suponhamos, por contradição, que $\overline{\mathcal{L}_0}$ seja recursivamente enumerável. Então é aceita por alguma máquina de Turing M no alfabeto Σ. O código desta máquina é um elemento de $\mathcal{L}$. Podemos perguntar:

$$c(M) \text{ pertence a } \overline{\mathcal{L}_0}?$$

Digamos que a resposta desta pergunta seja sim. Neste caso

$$c(M) \in \overline{\mathcal{L}_0} = L(M).$$

Mas se $c(M) \in L(M)$ então $c(M)$ é aceita por M. Assim, por definição, $c(M) \in \mathcal{L}_0$; e obtemos uma contradição.

Por outro lado, se a resposta da pergunta for não, então

$$c(M) \notin \overline{\mathcal{L}_0} = L(M).$$

Em outras palavras, $c(M)$ não é aceita por M. Mas isto significa que $c(M) \notin \mathcal{L}_0$; isto é $c(M) \in \overline{\mathcal{L}_0}$; e mais uma vez obtemos uma contradição.

Mostramos assim que existem linguagens recursivamente enumeráveis que não são recursivas. De quebra, obtemos o seguinte resultado.

COROLÁRIO 15.3. *O complemento de uma linguagem recursivamente enumerável não é necessariamente recursivamente enumerável.*

6. O problema da parada

Podemos enunciar este problema da seguinte maneira:

PROBLEMA 15.1. *Dada uma máquina de Turing M no alfabeto Σ e uma palavra $w \in \Sigma^*$ existe uma máquina de Turing $\mathcal{P}$ que, tendo $c(M) \cdot Z \cdot w$ como entrada decide se M pára com entrada w?*

Se o problema da parada tivesse uma resposta afirmativa, então toda linguagem recursivamente enumerável seria recursiva. Para mostrar isto, suponha que L é uma linguagem recursivamente enumerável qualquer e seja M uma máquina de Turing que aceita L. Se conhecemos M, então conhecemos $c(M)$. Seja $\mathcal{C}$ a máquina que transforma a entrada $\triangleleft w \sqcup$ em $\triangleleft c(M) Z w \sqcup$. Se $w \in L$ então a máquina M pára na entrada w; portanto $C \cdot \mathcal{P}$ pára no estado *sim*. Do contrário, M não pára com entrada M; mas neste caso $C \cdot \mathcal{P}$ também pára, só que no estado *não*.

Diante disto podemos concluir que o problema da parada não pode ter uma resposta afirmativa porque, como vimos na seção anterior, existem linguagens recursivamente enumeráveis que não são recursivas.

A resposta negativa para o problema da parada tem importantes conseqüências, tanto de natureza teórica quanto prática. Por exemplo, seria

certamente desejável ter algoritmo que, recebendo um programa P e uma entrada E de P, decidisse se P pára com entrada E. Um algoritmo deste tipo ajudaria muito na elaboração e teste de programas. Infelizmente a resposta negativa para o problema da parada mostra que um tal algoritmo não pode existir.

Uma conseqüência de natureza teórica muito importante do mesmo problema é que nem toda questão matemática pode ser decidida de maneira algorítmica. Em outras palavras, há limites teóricos claros para o que uma máquina pode fazer.

Além disto o problema da parada está longe de ser o único problema matemático que não admite solução algorítmica. Outros problemas, ainda na área de linguagens formais, que padecem do mesmo mal são os seguintes:

- dadas duas linguagens livres de contexto, determinar se sua intersecção é livre de contexto.
- dada uma linguagem livre de contexto, determinar se é regular.
- dada uma linguagem livre de contexto, determinar se é inerentemente ambígüa.

Referências Bibliográficas

[1] M. A. Harrison, *Introduction to formal language theory*, Addison-Wesley (1978).
[2] J. E. Hopcroft e J. D. Ullman, *Introduction to automata theory, languages and computation*, Addison-Wesley (1979).