Autômatos, Linguagens Formais e Computabilidade

S. C. Coutinho
&
Luis Menasché Schechter

Versão 0.9

Universidade Federal do Rio de Janeiro

Agradecimentos

Agradecemos ao professor João Carlos Pereira da Silva e aos ex-alunos abaixo pelas sugestões, revisões e correções a esta apostila.

- André Reis de Brito
- Carlos Eduardo da Silva Martins
- Christian da Silva Cabral Cardozo
- Daniel Artine
- Daniel Atkinson Oliveira
- David Boechat
- Fabiano de Paula Martins
- Gabriel de Sá Rodrigues
- Gabriel Pires da Silva
- Gabriel Rosário
- Giancarllo Alves Rojas
- Hugo Siqueira Gomes
- Igor Carpanese Figueiredo
- Igor da Fonseca Ramos
- Ingrid Pacheco
- Lenise Maria de Vasconcelos Rodrigues
- Luiz Guilherme Ribeiro
- Pedro Carvalho da Silva
- Rodrigo Ming Zhou
- Ygor Luis Mesquita Pereira da Hora
- Yuri de Jesus Lopes Abreu

Sumário

CAPíTULO 0

Introdução

Neste capítulo, apresentamos os principais tópicos de estudo que serão tratados ao longo destas notas e as principais motivações por trás deste estudo. Apresentamos também os conceitos mais básicos relativos à linguagens formais que serão utilizados no restante das notas.

1. Tópicos Centrais de Estudo

Os principais tópicos de estudo que serão cobertos ao longo destas notas são descritos de forma bastante resumida abaixo.

I) **Teoria da Computabilidade ou Teoria da Computação:** Busca descrever e estudar modelos matemáticos formais de computação, com o objetivo de analisar as seguintes questões:
   (1) Quais problemas podem ser *algoritmicamente resolvidos* por um computador?
   (2) Quais são os *limites* do que um computador consegue resolver?

II) **Teoria da Complexidade:**
   (1) Busca formas de avaliar formalmente o consumo de tempo e memória dos algoritmos.
   (2) Busca formas de determinar formalmente se existe um algoritmo eficiente para resolver um dado problema.

III) **Teoria de Autômatos:** Busca o estudo de diferentes modelos formais de computação, a partir de um modelo mais simples até o modelo mais poderoso, que possui o mesmo poder de um computador real. Estes modelos computacionais são conhecidos como *autômatos*. Os modelos computacionais que serão estudados nestas notas são, em ordem crescente de poder computacional, os seguintes:

(1) Autômatos Finitos

(2) Autômatos de Pilha

(3) Máquinas de Turing

2. Conceitos Básicos

DEFINIÇÃO 0.1. *Um* alfabeto *é um conjunto finito (e não-vazio) de símbolos* $\Sigma = \{a_1, \ldots, a_n\}$.

DEFINIÇÃO 0.2. *Uma* palavra *ou uma* cadeia *ou uma* string *em um alfabeto* Σ *é uma sequência finita* $w = w_1 \ldots w_n$, $n \in \mathbb{N}$, *de símbolos de* Σ *(*$w_i \in \Sigma$*, para todo* $1 \leq i \leq n$*). A* palavra vazia, *denotada por* ε*, é uma palavra que não contém nenhum símbolo.*

DEFINIÇÃO 0.3. *O* comprimento *de uma palavra* w*, denotado por* $|w|$*, é definido como a quantidade de símbolos (não necessariamente distintos) que ocorrem na palavra* w*. Então, se* $w = w_1 \ldots w_n$, $|w| = n$.

EXEMPLO 0.4. Seja $\Sigma = \{a, b\}$, $w_1 = baabb$ e $w_2 = aaa$. Então, $|w_1| = 5$ e $|w_2| = 3$.

OBSERVAÇÃO. Em qualquer alfabeto, é possível construir uma única palavra de comprimento zero, também conhecida como *palavra vazia*, denotada por ε. Isto é, $|\varepsilon| = 0$.

DEFINIÇÃO 0.5. *Denotamos por* Σ^* *o conjunto de todas as palavras formadas por símbolos no alfabeto* Σ.

EXEMPLO 0.6. Seja $\Sigma = \{a, b\}$. Então, $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$.

DEFINIÇÃO 0.7. *Definimos uma* linguagem formal, *ou simplesmente* linguagem, L *sobre um alfabeto* Σ *como sendo um subconjunto qualquer de* Σ^**, isto é,* $L \subseteq \Sigma^*$.

EXEMPLO 0.8. Seja $\Sigma = \{a, b\}$. Então, os conjuntos $L_1 = \emptyset$, $L_2 = \Sigma^*$, $L_3 = \{\varepsilon\}$, $L_4 = \{a, bbb, aba\}$ e $L_5 = \{\varepsilon, a, aa, aaa, \ldots\}$ são todos exemplos de linguagens sobre o alfabeto Σ, uma vez que todos estes conjuntos são subconjuntos de Σ^*.

OBSERVAÇÃO. É importante observar que L_1 e L_3 são linguagens diferentes. L_1 é uma linguagem que não contém *nenhuma* palavra, isto é, é uma *linguagem vazia*. Já L_3 é uma linguagem que contém uma única palavra, sendo esta palavra a *palavra vazia*.

EXEMPLO 0.9. Seja $\Sigma = \{A, B, \ldots, Z, a, b, \ldots, z\}$. Vamos denotar por L_{port} o conjunto de todas as palavras da Língua Portuguesa. Vemos então que $L_{port} \subseteq \Sigma^*$, logo L_{port} é efetivamente uma linguagem sobre o alfabeto Σ. Temos que $\varepsilon \in \Sigma^*$, mas $\varepsilon \notin L_{port}$; $aula \in \Sigma^*$ e $aula \in L_{port}$; $class \in \Sigma^*$, mas $class \notin L_{port}$ e $AuLa \in \Sigma^*$, mas $AuLa \notin L_{port}$.

DEFINIÇÃO 0.10. *Uma operação importante sobre palavras é a* concatenação. *Sejam $w_1, w_2 \in \Sigma^*$. Definimos a palavra $w_1.w_2$, ou simplesmente w_1w_2 como a palavra obtida escrevendo-se a palavra w_1 e, imediatamente após o último símbolo de w_1, os símbolos de w_2. Isto é, se $w_1 = \alpha_1\alpha_2 \ldots \alpha_n \in \Sigma^*$ ($|w_1| = n$) e $w_2 = \beta_1\beta_2 \ldots \beta_t \in \Sigma^*$ ($|w_2| = t$), então $w_1w_2 \in \Sigma^*$ e $w_1w_2 = \alpha_1\alpha_2 \ldots \alpha_n\beta_1\beta_2 \ldots \beta_t$, com $|w_1w_2| = n + t$.*

OBSERVAÇÃO. A concatenação de palavras é associativa, mas não comutativa. Além disso, para qualquer palavra w, temos que $w.\varepsilon = \varepsilon.w = w$.

EXEMPLO 0.11. Seja $\Sigma = \{0, 1\}$, $w_1 = 0101$ e $w_2 = 11100$. Então, $w_1.w_2 = 010111100$ e $w_2.w_1 = 111000101$.

3. Operações sobre Linguagens

Como linguagens são conjuntos de palavras em um alfabeto, várias operações sobre linguagens originam-se na teoria de conjuntos. Da mesma forma, o conhecimento dos resultados elementares da teoria de conjuntos é muito útil para o estudo das linguagens formais.

(1) União:

$$L_1 \cup L_2 = \{w : w \in L_1 \text{ ou } w \in L_2\}.$$

EXEMPLO 0.12. Seja $\Sigma = \{0, 1\}$, $L_1 = \{00, 11, 000\}$ e $L_2 = \{00, 101, 100\}$. Então $L_1 \cup L_2 = \{00, 11, 000, 101, 100\}$.

(2) Interseção:

$$L_1 \cap L_2 = \{w : w \in L_1 \text{ e } w \in L_2\}.$$

EXEMPLO 0.13. Sejam Σ, L_1 e L_2 como no exemplo anterior. Então $L_1 \cap L_2 = \{00\}$.

(3) Diferença:

$$L_1 - L_2 = \{w : w \in L_1 \text{ e } w \notin L_2\}.$$

OBSERVAÇÃO. Uma outra notação para a diferença de conjuntos é $L_1 \backslash L_2$.

EXEMPLO 0.14. Sejam Σ, L_1 e L_2 como nos exemplos anteriores. Então $L_1 - L_2 = \{11, 000\}$.

OBSERVAÇÃO. Ao contrário das operações de união e interseção, a operação de diferença não é comutativa. Isto é, $L_1 - L_2 \neq L_2 - L_1$.

(4) Complemento:

$$\overline{L} = \{w : w \notin L\}.$$

OBSERVAÇÃO. Se L é uma linguagem sobre o alfabeto Σ, isto é, se $L \subseteq \Sigma^*$, então $L \cup \overline{L} = \Sigma^*$ e $\overline{L} = \Sigma^* - L$.

OBSERVAÇÃO. A seguinte igualdade é sempre verdadeira: $L_1 - L_2 = L_1 \cap \overline{L_2}$.

(5) Concatenação:

A concatenação de linguagens é definida a partir da operação de concatenação de palavras.

$$L_1.L_2 = \{w_1.w_2 : w_1 \in L_1 \text{ e } w_2 \in L_2\}.$$

EXEMPLO 0.15. Sejam Σ, L_1 e L_2 como nos exemplos anteriores. Então $L_1.L_2 = \{0000, 00101, 00100, 1100, 11101, 11100, 00000, 000101, 000100\}$.

OBSERVAÇÃO. Se L_1 e L_2 são conjuntos finitos, então $|L_1.L_2| \leq |L_1| \times |L_2|$.

OBSERVAÇÃO. Assim como a concatenação de palavras, a concatenação de linguagens não é comutativa, isto é, $L_1.L_2 \neq L_2.L_1$.

OBSERVAÇÃO. Temos que $L.\{\varepsilon\} = \{\varepsilon\}.L = L$ e $L.\emptyset = \emptyset.L = \emptyset$.

(6) Estrela de Kleene:

$$L^* = \{\varepsilon\} \cup L \cup L.L \cup L.L.L \cup \ldots$$

OBSERVAÇÃO. Temos que $\emptyset^* = \{\varepsilon\}$ e $\{\varepsilon\}^* = \{\varepsilon\}$.

4. 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) $A \cap (A \cup B) = A$;

l) $A \cup (A \cap B) = A$;

m) $A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$;

n) $A \setminus B = A \cap \overline{B}$ onde $\overline{B}$ é o complemento de B no conjunto *universo*, isto é o conjunto que contém todos os elementos com que estamos trabalhando;

o) $\overline{(A \cap B)} = \overline{A} \cup \overline{B}$;

p) $\overline{(A \cup B)} = \overline{A} \cap \overline{B}$;

q) $\overline{\overline{A}} = A$;

r) $A \setminus B = A \setminus (B \cap A)$;

s) $B \subset \overline{A}$ se, e somente se, $A \cap B = \emptyset$;

t) $(A \setminus B) \setminus C = (A \setminus C) \setminus (B \setminus C) = A \setminus (B \cup C)$;

u) $A \cap \overline{B} = \emptyset$ e $\overline{A} \cap B = \emptyset$ se e somente se $A = B$.

(2) Considere as afirmações abaixo: prove as verdadeiras 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 $\{\varepsilon\}$. Mostre que $\emptyset.L = L.\emptyset = \emptyset$ e $\{\varepsilon\}.L = L.\{\varepsilon\} = L$ para qualquer linguagem L.

(5) Mostre que $(L^*)^* = L^*$ para qualquer linguagem L.

(6) Seja $\Sigma = \{0, 1\}$. Se $L_1 = \{0\}$ e $L_2 = \{1\}^*$, determine:

a) $L_1 \cup L_2$ e $L_1 \cap L_2$;

b) L_1^* e L_2^*;

c) $(L_1 \cup L_2)^*$;

d) $L_1^* \cap L_2$;

e) $\overline{L_1}$;

f) $\overline{L_2} \cap L_1^*$.

(7) Seja w uma palavra em um alfabeto Σ. Definimos o *reflexo* de w recursivamente da seguinte maneira: $\varepsilon^R = \varepsilon$ e se $w = \sigma x$ então $w^R = x^R\sigma$ onde $\sigma \in \Sigma$.

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 $(x^R)^R$.

(8) 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.

(9) 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) $(L_1^*)^R$.

(10) Mostre, por indução em n, que se $L_0, \ldots, L_n$ são linguagens no alfabeto Σ então

$$L_0 \cdot (L_1 \cup \cdots \cup L_n) = (L_0 \cdot L_1) \cup \cdots \cup (L_0 \cdot L_n).$$

(11) Seja L uma linguagem no alfabeto Σ. Considerando que

$$L^* = \{\varepsilon\} \cup L \cup L.L \cup L.L.L \cup \ldots \text{ e}$$

$$L^+ = L \cup L.L \cup L.L.L \cup \ldots,$$

o que podemos concluir a respeito de L se $L^+ = L^* \setminus \{\varepsilon\}$?

(12) 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:

- $\phi(\varepsilon) = \varepsilon$;
- $\phi(\sigma x) = \phi(\sigma)\phi(x)$, onde $\sigma \in \Sigma_1$.

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) $\phi(L \cdot L') = \phi(L) \cdot \phi(L')$;

d) $\phi(L^*) = \phi(L)^*$.

Parte 1

Linguagens Regulares

CAPíTULO 1

Linguagens Regulares e Autômatos Finitos Determinísticos

Neste capítulo, apresentamos o primeiro modelo formal de computação que iremos estudar, os *autômatos finitos determinísticos*. Eles são o modelo de computação menos poderoso dentre os que estudaremos. Apresentamos neste capítulo também uma classe de linguagens formais associadas a estes autômatos, a classe das *linguagens regulares*.

1. Autômatos Finitos Determinísticos (AFD's)

DEFINIÇÃO 1.1. *Um autômato finito determinístico (abreviado como AFD)* A *é uma 5-upla* $A = (\Sigma, Q, q_0, F, \delta)$*, onde:*

- Σ *é um* alfabeto *(lembrando que, pela definição do capítulo anterior, alfabetos são sempre* finitos*);*
- Q *é um* conjunto finito *(e não-vazio) de* estados*;*
- $q_0 \in Q$ *é o* estado inicial*;*
- $F \subseteq Q$ *é o conjunto de* estados finais *e*
- δ *é a* função de transição*. Esta função tem o formato*

$$\delta : Q \times \Sigma \to Q,$$

isto é, para cada par formado por um estado do conjunto Q *e um símbolo do alfabeto* Σ*, a função de transição fornece como resposta um estado de* Q*.*

EXEMPLO 1.2. Seja A o AFD definido pelos componentes $(\Sigma, Q, q_0, F, \delta)$, onde:

- $\Sigma = \{0, 1\}$;

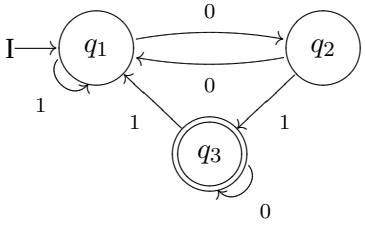
- $Q = \{q_1, q_2, q_3\}$;
- $q_0 = q_1$;
- $F = \{q_3\}$;
- A função δ é dada pela tabela abaixo.

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

Esta tabela representa que $\delta(q_1, 0) = q_2$, $\delta(q_1, 1) = q_1$, $\delta(q_2, 0) = q_1$ e assim por diante.

OBSERVAÇÃO. Um AFD também admite uma representação através de um grafo direcionado com arestas rotuladas. Neste grafo, conseguimos representar os 5 componentes da tupla $A = (\Sigma, Q, q_0, F, \delta)$. Os vértices do grafo representam os elementos do conjunto Q. Estes vértices são representados por círculos simples, com o nome do respectivo elemento de Q escrito dentro do círculo, com uma diferenciação para os elementos do conjunto F, que tem seus vértices representados por círculos duplos. O estado inicial q_0 é demarcado por uma seta com um I maiúsculo apontando para o vértice correspondente a este estado. Os símbolos do alfabeto Σ aparecem como rótulos das arestas do grafo, que são rotuladas de acordo com a função de transição δ. Se, para um determinado $q \in Q$ e para um determinado $\sigma \in \Sigma$, temos $\delta(q, \sigma) = q'$, então colocamos no grafo uma aresta direcionada do vértice de q para o vértice de q' e rotulamos esta aresta com o símbolo σ.

EXEMPLO 1.3. O grafo que representa o AFD do exemplo anterior é mostrado abaixo.



OBSERVAÇÃO. O autômato se chama *determinístico* porque, para cada par de estado de Q e símbolo de Σ, a sua função de transição δ oferece *exatamente um* estado como resposta. Ou seja, a resposta da função de transição é *completamente determinada*. Ao representarmos a função de transição em forma de tabela, como fizemos no exemplo anterior, esta característica da função se traduz no fato de que a tabela é sempre completamente preenchida e cada campo da tabela possui exatamente um estado. Já ao representarmos o AFD como um grafo, esta característica da função se traduz no fato de que cada vértice possui exatamente uma aresta rotulada por cada símbolo do alfabeto saindo dele.

A grosso modo, um AFD é um dispositivo que processa uma palavra que recebe como entrada (lembrando que, por nossa definição no capítulo anterior, palavras são sempre *finitas*) e fornece como saída uma resposta do tipo Sim/Não. O AFD $A = (\Sigma, Q, q_0, F, \delta)$ inicia seu processamento no estado q_0 e recebe como entrada uma palavra $w \in \Sigma^*$. Ele lê um símbolo de w de cada vez, começando pelo seu primeiro símbolo mais à esquerda. Se o autômato se encontra em um estado q e lê o símbolo σ, então, após a leitura, ele irá para o estado $q' = \delta(q, \sigma)$.

A partir destas transições de estado, podemos definir a noção de uma *computação* do autômato com uma palavra w.

DEFINIÇÃO 1.4. *Uma* configuração *do AFD A é um par formado por um estado de A e uma palavra de Σ^* (um elemento de $Q \times \Sigma^*$). O primeiro componente de uma configuração representa o* estado atual *do autômato, enquanto o segundo componente representa o trecho da palavra de entrada que ainda não foi lida pelo autômato.*

DEFINIÇÃO 1.5. *A relação de* configuração seguinte *(notação $\vdash$) é definida como $(q, w) \vdash (q', w')$ se $w = \sigma w'$ e $\delta(q, \sigma) = q'$.*

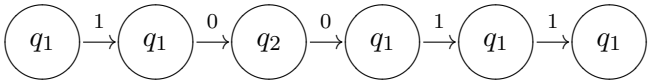
DEFINIÇÃO 1.6. *Se $C_0, \ldots, C_n$ são configurações de A e se $C_0 \vdash \ldots \vdash C_n$, então temos uma* computação *em A, notação $C_0 \vdash^* C_n$ (em particular, assumimos que $C_0 \vdash^* C_0$).*

DEFINIÇÃO 1.7. *Dizemos que uma palavra $w \in \Sigma^*$ é* aceita *ou* reconhecida *pelo AFD A se $(q_0, w) \vdash^* (f, \varepsilon)$ e $f \in F$, isto é, se após a leitura de todos os símbolos da palavra w, o autômato termina em algum de seus estados finais. Caso contrário, dizemos que a palavra é* rejeitada *ou* não aceita *ou* não reconhecida *pelo AFD A.*

EXEMPLO 1.8. Consideramos o autômato descrito no exemplo anterior. Vamos descrever qual é a sua computação ao receber como entrada a palavra $w_1 = 10011$.

$$(q_1, 10011) \vdash (q_1, 0011) \vdash (q_2, 011) \vdash (q_1, 11) \vdash (q_1, 1) \vdash (q_1, \varepsilon)$$

Graficamente:

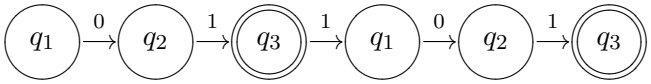


Como o autômato termina esta computação no estado q_1, que não é um dos seus estados finais, a palavra w_1 *não é aceita* pelo autômato.

EXEMPLO 1.9. Consideramos novamente o autômato descrito no exemplo anterior e agora vamos descrever qual é a sua computação ao receber como entrada a palavra $w_2 = 01101$.

$$(q_1, 01101) \vdash (q_2, 1101) \vdash (q_3, 101) \vdash (q_1, 01) \vdash (q_2, 1) \vdash (q_3, \varepsilon)$$

Graficamente:



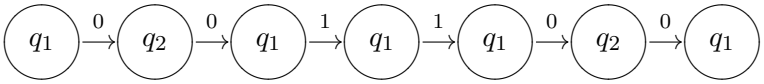
Como o autômato termina esta computação no estado q_3, que é um dos seus estados finais (o único estado final neste caso), a palavra w_2 *é aceita* pelo autômato.

DEFINIÇÃO 1.10. *Definimos a* linguagem aceita *por um AFD A com alfabeto Σ, denotada por $L(A)$, como sendo o conjunto de todas as palavras $w \in \Sigma^*$ que são* aceitas *por A. Observe que temos $L(A) \subseteq \Sigma^*$.*

EXEMPLO 1.11. Consideramos novamente o autômato descrito no exemplo anterior e agora vamos descrever qual é a sua computação ao receber como entrada a palavra $w_3 = 001100$.

$$(q_1, 001100) \vdash (q_2, 01100) \vdash (q_1, 1100) \vdash (q_1, 100) \vdash (q_1, 00) \vdash (q_2, 0) \vdash (q_1, \varepsilon)$$

Graficamente:

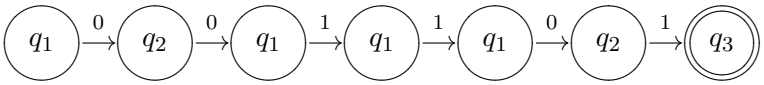


Como o autômato termina esta computação no estado q_1, que não é um dos seus estados finais, a palavra w_3 *não é aceita* pelo autômato. Isto significa que $w_3 \notin L(A)$.

EXEMPLO 1.12. Consideramos novamente o autômato descrito no exemplo anterior e agora vamos descrever qual é a sua computação ao receber como entrada a palavra $w_4 = 001101$.

$$(q_1, 001101) \vdash (q_2, 01101) \vdash (q_1, 1101) \vdash (q_1, 101) \vdash (q_1, 01) \vdash (q_2, 1) \vdash (q_3, \varepsilon)$$

Graficamente:



Como o autômato termina esta computação no estado q_3, que é um dos seus estados finais (o único estado final neste caso), a palavra w_4 *é aceita* pelo autômato. Isto significa que $w_4 \in L(A)$.

DEFINIÇÃO 1.13. *Seja L uma linguagem. Dizemos que L é uma* linguagem regular *se existe um AFD A tal que $L = L(A)$, isto é, se é possível construir um AFD tal que as palavras aceitas por ele sejam exatamente as palavras que pertencem à linguagem L.*

2. Exercícios

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

(2) Desenhe o grafo de estados de 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 $n > 0$ símbolos. Dado o valor de n e a quantidade $m > 0$ de estados, quantos autômatos finitos determinísticos existem satisfazendo estes valores? (A resposta será em função de m e n.)

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 2

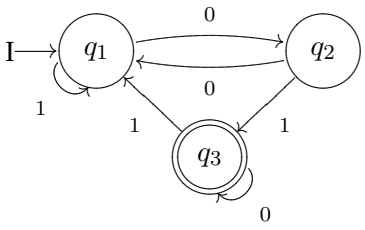
Expressões Regulares

Neste capítulo, apresentamos uma maneira compacta e uniforme de descrever as *linguagens regulares*, isto é, as linguagens que são aceitas por autômatos finitos determinísticos. Esta notação compacta é dada pelas *expressões regulares*.

1. Breve Motivação

Vamos observar novamente o exemplo do AFD A mostrado no capítulo anterior. Reproduzimos novamente abaixo o seu grafo.

EXEMPLO 2.1.



Olhando para o grafo deste AFD, podemos tentar determinar as palavras que pertencem a $L(A)$. Serão palavras que descrevam um caminho de q_1 até q_3 no grafo. Não é complicado descrever algumas possíveis palavras que satisfaçam essa propriedade. O grande problema é que, para descrever $L(A)$, precisamos ser capazes de descrever *todas* estas palavras. Mostramos abaixo a descrição, em português, de algumas palavras que pertencem a $L(A)$.

(1) A palavra começa com um número ímpar de 0's, seguido de um único 1, terminando com uma quantidade arbitrária de 0's.

(2) A palavra começa com uma quantidade arbitrária de 1's, depois segue o padrão do item anterior.

(3) A palavra começa com uma quantidade arbitrária de repetições da sequência 011 e continua como um dos padrões anteriores.

Esta listagem de padrões ainda não descreve todas as palavras de $L(A)$ e, mesmo assim, já começa a ficar bastante extensa. É importante então buscarmos uma notação uniforme e compacta que nos permita descrever exatamente a linguagem aceita por um AFD.

2. Expressões Regulares

Seja Σ um alfabeto e seja

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

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

DEFINIÇÃO 2.2. *Uma* expressão regular *(abreviada como ER) sobre o alfabeto Σ 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;

(5) Nada mais é considerado expressão regular.

EXEMPLO 2.3. Seja $\Sigma = \{0, 1\}$. Então, $r_1 = (((((0)^*.0).1).(1)^*) \cup (0.0))$ é uma expressão regular sobre Σ. Por outro lado, $r_2 = (1.(1)^* \cup)$ não é uma expressão regular, pois o operador $\cup$ deve conter expressões regulares de ambos os lados, de acordo com a regra (3) acima.

Para diminuir o uso de parênteses que não sejam necessários, adotamos uma convenção de precedência entre os operadores que podem aparecer nas expressões regulares (*,. e $\cup$). O operador * é o de maior precedência, seguido do operador ., sendo o operador $\cup$ o de menor precedência. Outra convenção que adotamos

é a omissão do operador . nas expressões regulares, assim como o operador . de multiplicação é geralmente omitido em expressões algébricas.

EXEMPLO 2.4. A expressão regular r_1 do exemplo acima pode ser re-escrita, de acordo com nossas convenções, como $r_1' = 0^*011^* \cup 00$.

Uma expressão regular sobre Σ representa uma linguagem $L \subseteq \Sigma^*$. Se r é uma ER, denotamos por $L(r)$ a linguagem *representada* ou *denotada* ou *gerada* por r.

DEFINIÇÃO 2.5. *Dada uma expressão regular r sobre Σ, podemos definir a linguagem $L(r)$ recursivamente a partir das seguintes regras.*

(1) *Se $\sigma \in \Sigma$, então $L(\sigma) = \{\sigma\}$;*

(2) $L(\emptyset) = \emptyset$*;*

(3) $L(\varepsilon) = \{\varepsilon\}$*;*

(4) $L(r_1 \cup r_2) = L(r_1) \cup L(r_2)$*;*

(5) $L(r_1.r_2) = L(r_1).L(r_2)$*;*

(6) $L(r^*) = L(r)^*$

EXEMPLO 2.6. Voltando ao exemplo que utilizamos como motivação no início deste capítulo, vamos descrever cada um daqueles três padrões de palavras que foram escritos em português naquela seção utilizando agora a notação das expressões regulares.

(1) Para descrever a ocorrência de um número arbitrário de 0's, podemos utilizar a expressão regular 0^*, uma vez que

$$L(0^*) = L(0)^* = \{0\}^* = \{\varepsilon, 0, 00, 000, \ldots\}.$$

Analogamente, para descrever uma quantidade par qualquer de 0's, podemos utilizar a expressão regular $(00)^*$. Logo, para descrever uma quantidade ímpar qualquer de 0's, podemos utilizar a expressão regular $(00)^*0$. Desta forma, o primeiro padrão descrito no início do capítulo pode ser representado pela expressão regular $r_1 = (00)^*010^*$.

(2) O segundo padrão pode ser representado pela expressão regular $r_2 = 1^*(00)^*010^*$.

(3) O terceiro padrão pode ser representado pela expressão regular $r_3 = (011)^*1^*(00)^*010^*$.

EXEMPLO 2.7. Seja $\Sigma = \{0, 1\}$ e $r_4 = (01 \cup 100)^*$. Então

$$L(r_4) = L((01 \cup 100)^*) = L(01 \cup 100)^* = (L(01) \cup L(100))^* =$$

$$= (\{01\} \cup \{100\})^* = \{01, 100\}^* =$$

$$= \{\varepsilon, 01, 100, 0101, 01100, 10001, 100100, 010101, 0101100, \ldots\}.$$

OBSERVAÇÃO. Repare que duas expressões regulares distintas podem denotar a mesma linguagem, isto é, podemos ter duas expressões regulares r_1 e r_2 tais que $r_1 \neq r_2$, mas $L(r_1) = L(r_2)$. Este é o caso, por exemplo, das expressões $(0 \cup 1)^*$ e $((0^* \cdot 1)^* \cdot 0^*)$.

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. Resta-nos praticar um pouco a construção de uma expressão regular que denote um conjunto dado, a partir da descrição deste conjunto.

EXEMPLO 2.8. Suponhamos que $\Sigma = \{a, b, c\}$. Para obter *todas as palavras* em um certo subconjunto de Σ devemos usar o operador estrela (*). 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	$(a \cdot a)^*$

EXEMPLO 2.9. 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 $(b \cup c)^* \cdot a \cdot (b \cup c)^*$.

EXEMPLO 2.10. 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

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

EXEMPLO 2.11. 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

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

Agora que já conhecemos a notação das *expressões regulares*, que nos permite descrever de forma uniforme e compacta algumas linguagens, resta-nos mostrar que dado qualquer AFD A, podemos descrever $L(A)$ através de uma expressão regular. Isto é, precisamos resolver o seguinte problema.

PROBLEMA 2.12. Dado um AFD $A = (\Sigma, Q, q_0, F, \delta)$, quero construir uma expressão regular r tal que $L(r) = L(A)$, isto é, tal que a linguagem gerada por r seja exatamente a linguagem aceita por A.

Estudaremos um algoritmo que resolve este problema no próximo capítulo.

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, que é, de certa forma, o inverso do problema que já descrevemos acima.

PROBLEMA 2.13. Dada uma expressão regular r, construir um autômato finito que aceite a linguagem gerada por r.

Abordaremos este problema detalhadamente mais adiante nestas notas. Entretanto, já estamos em condições de obter expressões regulares para as linguagens que descrevem inteiros e variáveis de um programa, como veremos a seguir.

EXEMPLO 2.14. 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

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

EXEMPLO 2.15. Já a expressão que denota os inteiros não negativos é

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

EXEMPLO 2.16. 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

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

3. Exercícios

(1) 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) $(0 \cup 1)(0 \cup 1)^*00$;

(2) 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 sequência de 1s, de modo que o número de 1s na sequência é congruente a 2 módulo 3, seguido de um número par de zeros.

(3) 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) $(\varepsilon \cup (r \cdot r^*)) \equiv r^*$;

h) $((r \cdot s)^* \cdot r) \equiv (r \cdot (s \cdot r)^*)$;

i) $(r \cup s)^* \equiv (r^* \cdot s^*)^* \equiv (r^* \cup s^*)^*$.

(4) Usando as identidades do exercício anterior, prove que

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

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

CAPíTULO 3

Relação entre AFD's e Expressões Regulares

Conforme já vimos, uma *linguagem regular* é uma linguagem que é aceita por algum AFD. No capítulo anterior, estudamos as *expressões regulares*, uma forma uniforme e compacta de descrever algumas linguagens. Neste capítulo, vamos mostrar que toda linguagem regular pode ser denotada por uma expressão regular (de forma que o nome "expressão *regular*" é realmente apropriado). Para isso, vamos mostrar que, para todo AFD A, podemos construir uma expressão regular r tal que $L(A) = L(r)$ e que podemos fazer isto através de um método algorítmico. Na verdade, existe mais de um algoritmo capaz de calcular r, mas um que é bastante simples e intuitivo é o algoritmo de substituição desenvolvido por John Brzozowski.

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 $(\Sigma, Q, q_0, F, \delta)$, onde $Q = \{q_1, q_2, \ldots, q_n\}$ e $q_0 = q_1$. Para cada $q \in Q$ definimos L_q como sendo a linguagem

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

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 linguagens L_p e L_q. De fato, temos que $(p, \sigma) \vdash (q, \varepsilon)$ ao passo que, se $w \in L_q$, então $(q, w) \vdash^* (f, \varepsilon)$. Combinado estas duas computações obtemos

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

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,

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

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

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

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

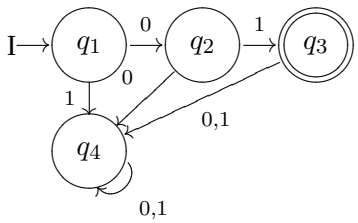
Precisamos ainda analisar o que acontece quando p é um estado final do autômato. Considerando em detalhe o argumento do parágrafo acima, vemos que assumimos implicitamente que $u \neq \varepsilon$, já que estamos explicitando o seu primeiro símbolo. Entretanto, se p for um estado final, teremos, além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

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

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

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

EXEMPLO 3.1. Considere o autômato finito determinístico $\mathcal{M}$ com alfabeto é $\{0, 1\}$ e cujo grafo é



Deste grafo extraímos as seguintes equações

$$L_1 = 0L_2 \cup 1L_4$$

$$L_2 = 1L_3 \cup 0L_4$$

$$L_3 = 0L_4 \cup 1L_4 \cup \{\varepsilon\} \text{ (já que } q_3 \text{ é estado final)}$$

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

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

$$L_1 = 0L_2$$

$$L_2 = 1L_3$$

$$L_3 = \{\varepsilon\}$$

$$L_4 = \emptyset.$$

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

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

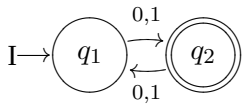
Finalmente, substituindo esta última equação na primeira, obtemos

$$L_1 = 0L_2 = 0\{1\} = \{01\}.$$

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.

EXEMPLO 3.2. Seja $\mathcal{N}$ o autômato finito determinístico de alfabeto $\{0, 1\}$ e grafo



As equações correspondentes a L_1 e L_2 são:

$$L_1 = 0L_2 \cup 1L_2$$

$$L_2 = 0L_1 \cup 1L_1 \cup \{\varepsilon\} \text{ (já que } q_2 \text{ é estado final)}$$

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

$$L_1 = (0 \cup 1)L_2$$

$$L_2 = (0 \cup 1)L_1 \cup \{\varepsilon\}.$$

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

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

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, frequentemente, ainda mais complicadas que (1.1). Convém, portanto, abordar desde 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

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

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, substituiremos $X = AX \cup B$ em (2.1), o que nos dá

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

Repetindo o mesmo procedimento k vezes, chegamos à equação

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

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

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

Para poder fazer isto de maneira formal, vamos recordar a operação *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,

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

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

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

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, substituiremos X por A^*B do lado direito da equação:

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

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

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

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

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

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

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

Temos, então, duas possibilidades. A primeira é que A não contenha ε. Neste caso $A \cap \varepsilon = \emptyset$, 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 4. 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 $\varepsilon \notin A$ então o maior subconjunto de Σ^* que satisfaz $X = AX \cup B$ é $X = A^*B$.*

EXEMPLO 3.3. 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 $(\Sigma, Q, q_0, F, \delta)$, onde $Q = \{q_1, \ldots, q_n\}$ e $q_0 = q_1$, de um autômato finito determinístico $\mathcal{M}$.

Saída: Uma expressão regular r tal que $L(r) = L(\mathcal{M})$.

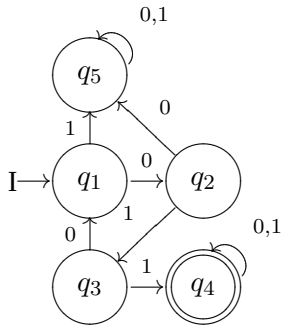
Primeira etapa: Escreva as equações para as linguagens L_j, para cada $1 \leq j \leq n$.

Segunda etapa: Começando por L_n e acabando em L_1, substitua L_j em todas as equações anteriores, aplicando primeiro o Lema de Arden se L_j 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 4. Por enquanto, nos contentaremos com a descrição acima.

EXEMPLO 3.4. Vamos aplicar ao autômato no alfabeto $\{0, 1\}$ cujo grafo é



As equações correspondentes a autômato são

$$L_1 = 0L_2 \cup 1L_5$$

$$L_2 = 1L_3 \cup 0L_5$$

$$L_3 = 0L_1 \cup 1L_4$$

$$L_4 = 0L_4 \cup 1L_4 \cup \varepsilon$$

$$L_5 = 0L_5 \cup 1L_5.$$

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

$$L_1 = 0L_2$$

$$L_2 = 1L_3$$

$$L_3 = 0L_1 \cup 1L_4$$

$$L_4 = 0L_4 \cup 1L_4 \cup \varepsilon$$

$$L_5 = \emptyset.$$

A equação para L_4 nos dá

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

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

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

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

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

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

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

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. 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.

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 $\eta_i \neq \varepsilon$ 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) $L(\varepsilon) = \{\varepsilon\}$;
(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 $(\Sigma, Q, q_0, F, \delta)$, onde $Q = \{q_1, \ldots, q_n\}$ e $q_0 = q_1$.

Saída: uma expressão regular r tal que $L(r) = L(\mathcal{A})$.

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

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

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

$$E_k = \eta_k \cdot \alpha_k \cup B, \tag{4.1}$$

onde $B \in \Theta_{k-1}$. Seja $F = \eta_k^* \cdot B$. Temos de (4.1) e (2) que

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

Logo, pelo lema de Arden,

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

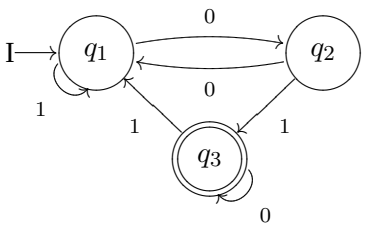
mas isto é igual a $L(F)$. Como o algoritmo manda fazer $E_k = F$, obtemos, ao final desta etapa, que $E_k \in \Theta_{k-1}$ e $L(E_k) = A_k$.

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 $L(E_j) = A_j$.

5. Último Exemplo

Nesta seção, retornamos ao exemplo da seção inicial do capítulo anterior e mostramos como obter, utilizando o algoritmo de substituição, uma notação compacta na forma de expressão regular para a linguagem aceita por aquele autômato.

EXEMPLO 3.5. Reproduzimos novamente abaixo o grafo daquele autômato.



As equações para este autômato são as seguintes:

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

Aplicando o Lema de Arden a L_3, com $A = 0$ e $B = 1.L_1 \cup \varepsilon$, obtemos

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

Substituindo L_3 nas equações de L_1 e L_2, obtemos as seguintes equações:

$$\begin{aligned} L_1 &= 0L_2 \cup 1L_1 \\ L_2 &= 0L_1 \cup 1(0^*1L_1 \cup 0^*) = (0 \cup 10^*1)L_1 \cup 10^* \end{aligned}$$

Substituindo L_2 na equação de L_1, obtemos

$$L_1 = 0((0 \cup 10^*1)L_1 \cup 10^*) \cup 1L_1 = (0(0 \cup 10^*1) \cup 1)L_1 \cup 010^*.$$

Aplicando o Lema de Arden a L_1, com $A = 0(0 \cup 10^*1) \cup 1$ e $B = 010^*$, obtemos

$$L_1 = (0(0 \cup 10^*1) \cup 1)^*010^*.$$

Esta é a expressão regular que denota a linguagem aceita pelo autômato.

6. Exercícios

(1) 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

(2) 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

CAPíTULO 4

Autômatos Finitos Não-Determinísticos

Neste capítulo, apresentamos e estudamos uma generalização simples dos autômatos finitos determinísticos. Retiramos da função de transição do autômato a propriedade do determinismo, obtendo os *autômatos finitos não-determinísticos*.

1. Autômatos Finitos Não-Determinísticos (AFND's)

DEFINIÇÃO 4.1. *Um autômato finito não-determinístico (abreviado como AFND)* A *é uma 5-upla* $A = (\Sigma, Q, q_0, F, \Delta)$*, onde:*

- Σ*,* Q*,* q_0 *e* F *são definidos da mesma forma que em um AFD e*
- Δ *é a* função de transição *não-determinística. Esta função tem o formato*

$$\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q),$$

isto é, para cada par formado por um estado do conjunto Q *e um símbolo do alfabeto* Σ *ou o símbolo* ε*, a função de transição fornece como resposta um conjunto de elementos de* Q*.*

OBSERVAÇÃO. $\mathcal{P}(Q)$ denota o conjunto das partes de Q, ou seja, o conjunto dos subconjuntos de Q. Como Q é um conjunto finito, se Q possui n elementos, $\mathcal{P}(Q)$ possui 2^n elementos.

EXEMPLO 4.2. Se $S = \{1, 2, 3\}$, então

$$\mathcal{P}(S) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

OBSERVAÇÃO. Nestas notas, sempre iremos diferenciar funções de transição determinísticas e não-determinísticas através da notação. Utilizamos uma letra delta minúscula (δ) para transições determinísticas e uma letra delta maiúscula (Δ) para transições não-determinísticas.

Vamos analisar as diferenças entre a função de transição $\delta : Q \times \Sigma \to Q$ de um AFD e a função de transição $\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ de um AFND.

A resposta da função δ é um elemento do conjunto Q, ou seja, é um estado do AFD para onde o autômato irá ao executar a transição. Já a resposta da função Δ é um elemento do conjunto $\mathcal{P}(Q)$, ou seja, é um conjunto de estados do AFND. Este conjunto de estados representa as *possibilidades* de estados para onde o autômato poderá ir ao executar a transição. Isto significa que, quando um AFND executa uma transição, o estado para onde ele vai é escolhido *aleatoriamente* dentre o conjunto de possibilidades oferecido pela resposta da função de transição. Esta é uma das fontes de *não-determinismo* do autômato.

Outra diferença entre a função de transição determinística e a função de transição não-determinística está nas entradas destas funções. A entrada da função determinística δ é sempre um par formado por um estado do autômato e um símbolo do alfabeto. Já a entrada da função não-determinística Δ admite duas possibilidades: a primeira é um par no mesmo formato anterior e a segunda é um par formado por um estado do autômato e o símbolo ε. As transições com entrada neste segundo formato oferecem ao AFND a possibilidade de mudar de estado sem consumir nenhum símbolo da palavra sendo processada pelo autômato, uma possibilidade que não existia nos AFD's. Esta é uma segunda fonte de não-determinismo do autômato.

Vamos analisar um exemplo concreto de AFND.

EXEMPLO 4.3. Seja A o AFND $A = (\Sigma, Q, q_0, F, \Delta)$, onde:

- $\Sigma = \{0, 1\}$;
- $Q = \{q_1, q_2, q_3, q_4\}$;
- $q_0 = q_1$;
- $F = \{q_4\}$;
- A função Δ é dada pela tabela abaixo.

Δ	0	1	ε
q_1	$\{q_2, q_3\}$	$\{q_3\}$	$\emptyset$
q_2	$\{q_4\}$	$\emptyset$	$\emptyset$
q_3	$\{q_3\}$	$\{q_2\}$	$\emptyset$
q_4	$\{q_1, q_2\}$	$\{q_4\}$	$\{q_3\}$

A partir desta tabela, temos que $\Delta(q_1, 0) = \{q_2, q_3\}$, $\Delta(q_1, 1) = \{q_3\}$, $\Delta(q_1, \varepsilon) = \emptyset$ e assim por diante. Isto significa que, se o autômato está atualmente no estado q_1 e o próximo símbolo da palavra a ser processado é 0, a função de transição Δ oferece duas possibilidades para o próximo estado do autômato: q_2 ou q_3. O autômato irá então escolher *aleatoriamente* entre estas duas possibilidades. Já se o autômato está atualmente no estado q_1 e o próximo símbolo da palavra a ser processado é 1, a função de transição Δ oferece apenas uma possibilidade para o próximo estado do autômato: q_3. Desta forma, este será certamente o estado do autômato após executar a transição. Finalmente, como $\Delta(q_1, \varepsilon) = \emptyset$, isto significa que a função de transição Δ não oferece nenhuma possibilidade para que o autômato, estando em q_1, faça uma mudança de estado sem consumir nenhum símbolo da palavra sendo processada.

Analisando um pouco mais a tabela, temos que $\Delta(q_2, 1) = \Delta(q_2, \varepsilon) = \emptyset$. Desta forma, se o autômato está atualmente no estado q_2 e o próximo símbolo da palavra a ser processado é 1, então a função Δ não oferece nenhuma possibilidade de transição para o autômato. O autômato não pode consumir o símbolo 1 e mudar de estado porque $\Delta(q_2, 1) = \emptyset$ e também não pode mudar de estado sem consumir nenhum símbolo da palavra porque $\Delta(q_2, \varepsilon) = \emptyset$. Assim, caso o autômato esteja no estado q_2 e o próximo símbolo da palavra seja 1, ele irá travar sem completar o processamento da palavra. Um AFD sempre processa a palavra que recebe como entrada completamente. Vemos que isto pode não acontecer em todos os casos com um AFND.

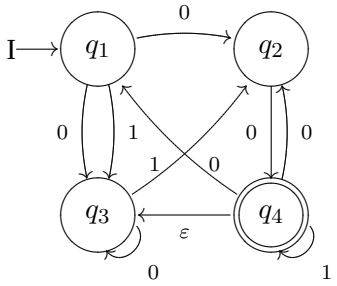
Para finalizar a análise da tabela, temos que $\Delta(q_4, 0) = \{q_1, q_2\}$ e $\Delta(q_4, \varepsilon) = \{q_3\}$. Desta forma, se o autômato está atualmente no estado q_4 e o próximo símbolo da palavra a ser processado é o 0, ele tem a possibilidade de duas escolhas

aleatórias para realizar com relação ao seu próximo estado. A primeira escolha é se ele irá realizar a transição consumindo o símbolo 0 ou se irá realizar a transição sem consumir nenhum símbolo da palavra que está processando. Caso ele opte pela segunda alternativa, como apenas q_3 está no conjunto $\Delta(q_4, \varepsilon)$, o autômato irá para o estado q_3. Caso ele opte pela primeira alternativa, ele irá então realizar uma escolha aleatória entre os dois estados que a transição $\Delta(q_4, 0)$ oferece como possibilidades: q_1 e q_2.

DEFINIÇÃO 4.4. *As transições da forma* $\Delta(q, \varepsilon)$*, para algum* $q \in Q$*, são chamadas de* ε-transições.

OBSERVAÇÃO. Um AFND também admite uma representação através de um grafo direcionado com arestas rotuladas, de maneira análoga à representação dos AFD's. Os símbolos do conjunto $\Sigma \cup \{\varepsilon\}$ aparecem como rótulos das arestas do grafo, que são rotuladas de acordo com a função de transição Δ. Se, para um determinado $q \in Q$ e para um determinado $\gamma \in \Sigma \cup \{\varepsilon\}$, temos $q' \in \Delta(q, \gamma)$ (lembrando que $\Delta(q, \gamma)$ é um *conjunto*), então colocamos no grafo uma aresta direcionada do vértice de q para o vértice de q' e rotulamos esta aresta com o símbolo γ.

EXEMPLO 4.5. O grafo que representa o AFND do exemplo anterior é mostrado abaixo.



OBSERVAÇÃO. Ao representarmos a função de transição Δ na forma de tabela, o não-determinismo da transição se traduz no fato de que a tabela é agora preenchida por subconjuntos de Q e não mais por elementos de Q como nos AFD's. Um caso particular importante de se salientar é que alguns campos da tabela podem ser preenchidos com o conjunto vazio $(\emptyset)$ que denota a ausência de possibilidades

de transição. Já ao representarmos o AFND como um grafo, o não-determinismo da transição se traduz na liberdade para que um vértice tenha *várias* arestas rotuladas por um mesmo símbolo saindo dele ou mesmo para que não tenha *nenhuma* aresta rotulada por um dado símbolo saindo dele.

Um AFND também processa uma palavra que recebe como entrada e fornece como saída uma resposta do tipo Sim/Não. O AFND $A = (\Sigma, Q, q_0, F, \Delta)$ inicia seu processamento no estado q_0 e recebe como entrada uma palavra $w \in \Sigma^*$. Ele lê um símbolo de w de cada vez, começando pelo seu primeiro símbolo mais à esquerda. Se o autômato se encontra em um estado q, ele pode ir para algum estado $q' \in \Delta(q, \varepsilon)$ sem ler nenhum símbolo da entrada (desde que $\Delta(q, \varepsilon) \neq \emptyset$) ou ele pode ler o próximo símbolo da entrada (denotado por σ) e, após a leitura, ele irá para algum estado $q'' \in \Delta(q, \sigma)$.

A partir destas transições de estado, podemos definir a noção de uma *computação* do autômato com uma palavra w.

DEFINIÇÃO 4.6. *Uma* configuração *do AFND A é um par formado por um estado de A e uma palavra de Σ^* (um elemento de $Q \times \Sigma^*$). O primeiro componente de uma configuração representa o* estado atual *do autômato, enquanto o segundo componente representa o trecho da palavra de entrada que ainda não foi lida pelo autômato.*

DEFINIÇÃO 4.7. *A relação de* configuração seguinte *(notação $\vdash$) é definida como $(q, w) \vdash (q', w')$ se*

(1) $w = \sigma w'$ *e* $q' \in \Delta(q, \sigma)$ *OU*

(2) $w = w'$ *e* $q' \in \Delta(q, \varepsilon)$.

OBSERVAÇÃO. Em um AFD, cada configuração (q, w) possui exatamente uma configuração seguinte, se $w \neq \varepsilon$, ou nenhuma configuração seguinte no caso em que $w = \varepsilon$. Já em um AFND, uma configuração (q, w) pode possuir uma, várias ou mesmo nenhuma configuração seguinte.

DEFINIÇÃO 4.8. *Se $C_0, \ldots, C_n$ são configurações de A e se $C_0 \vdash \ldots \vdash C_n$, então temos uma* computação *em A, notação $C_0 \vdash^* C_n$ (em particular, assumimos que $C_0 \vdash^* C_0$).*

OBSERVAÇÃO. Como uma configuração pode possuir várias configurações seguintes, o autômato pode ter a possibilidade de realizar diversas computações distintas com uma mesma palavra w. Devido às ε-transições, é possível também a existência de computações infinitas em um AFND, o que nunca pode acontecer em um AFD. Finalmente, devido à possibilidade de que a resposta para algumas transições seja o conjunto vazio, é possível a existência de computações em um AFND que travam sem conseguir ler a palavra da entrada completamente, algo que nunca acontece em um AFD.

Como uma mesma palavra w pode produzir diversas computações distintas em um AFND, a questão de quando uma palavra é ou não aceita pelo autômato se torna mais sutil. Algumas computações com w podem ser infinitas, outras podem travar sem ler w completamente, outras ainda podem ler w completamente e terminarem em um estado que não é final e, finalmente, outras podem ler w completamente e terminarem em um estado que é final. Como classificamos w neste caso? Esta palavra é aceita ou rejeitada?

A definição abaixo estabelece os critérios que utilizaremos para determinar se uma palavra é ou não aceita por um AFND.

DEFINIÇÃO 4.9. *Dizemos que uma palavra $w \in \Sigma^*$ é* aceita *ou* reconhecida *pelo AFND A se* existe pelo menos uma computação finita $(q_0, w) \vdash^* (f, \varepsilon)$ *onde $f \in F$. Isto é, uma palavra w é aceita se* existe pelo menos uma computação finita *tal que*

(1) w é completamente lida E

(2) O AFND termina a computação em algum de seus estados finais.

Caso contrário, dizemos que a palavra é rejeitada *ou* não aceita *ou* não reconhecida *pelo AFND A.*

OBSERVAÇÃO. É importante ressaltar que uma computação em que o autômato trava sem conseguir ler a palavra w completamente nunca é uma computação que satisfaz os critérios acima, mesmo que o travamento ocorra em um dos estados finais do autômato.

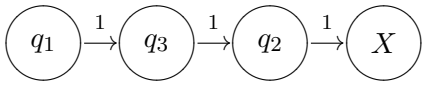
OBSERVAÇÃO. Vemos que as definições de aceitação e rejeição são de certa forma assimétricas em um AFND. Para uma palavra w ser aceita, basta que *uma* computação satisfaça os critérios acima. Já para uma palavra w ser rejeitada, é necessário que *todas as possíveis computações* com w não satisfaçam os critérios acima.

DEFINIÇÃO 4.10. *Definimos a* linguagem aceita *por um AFND A com alfabeto Σ, denotada por $L(A)$, como sendo o conjunto de todas as palavras $w \in \Sigma^*$ que são* aceitas *por A. Observe que temos $L(A) \subseteq \Sigma^*$.*

EXEMPLO 4.11. Consideramos o autômato descrito no exemplo anterior. Vamos descrever o comportamento do autômato ao receber como entrada a palavra $w_1 = 11101$.

$$(q_1, 11101) \vdash (q_3, 1101) \vdash (q2, 101) \vdash X$$

Graficamente:



Não existe nenhuma outra computação possível neste autômato para a palavra w_1 e esta computação trava sem processar a parte 101 da palavra. Logo, esta palavra não é aceita pelo autômato.

EXEMPLO 4.12. Consideramos o autômato descrito no exemplo anterior. Vamos descrever o comportamento do autômato ao receber como entrada a palavra $w_2 = 0011$. Neste caso, temos diversas possibilidades de computações neste autômato para a palavra w_2. Vamos descrevê-las.

(1) $(q_1, 0011) \vdash (q_2, 011) \vdash (q_4, 11) \vdash (q_4, 1) \vdash (q_4, \varepsilon)$

(2) $(q_1, 0011) \vdash (q_2, 011) \vdash (q_4, 11) \vdash (q_4, 1) \vdash (q_4, \varepsilon) \vdash (q_3, \varepsilon)$

(3) $(q_1, 0011) \vdash (q_2, 011) \vdash (q_4, 11) \vdash (q_4, 1) \vdash (q_3, 1) \vdash (q_2, \varepsilon)$

(4) $(q_1, 0011) \vdash (q_2, 011) \vdash (q_4, 11) \vdash (q_3, 11) \vdash (q_2, 1) \vdash X$

(5) $(q_1, 0011) \vdash (q_3, 011) \vdash (q_3, 11) \vdash (q_2, 1) \vdash X$

Vemos que destas cinco computações possíveis, duas travam sem ler a palavra w_2 completamente, duas leem toda a palavra e terminam em estados que não são finais (q_3 e q_2) e uma lê toda a palavra e termina em um estado final (q_4). Desta forma, como existe pelo menos uma computação finita que lê toda a palavra e termina em um estado final, a palavra w_2 é aceita pelo autômato. De fato, uma vez que a computação que aceita w_2 foi logo a primeira descrita e precisamos de apenas uma computação desta forma para determinar que a palavra é aceita, temos que para efeitos práticos era desnecessário ter calculado as demais computações possíveis.

OBSERVAÇÃO. Um AFD é um caso particular de um AFND. Considere um AFND em que a função de transição Δ satisfaz as seguintes propriedades:

(1) $\Delta(q, \varepsilon) = \emptyset$, para todo $q \in Q$ E

(2) $\Delta(q, \sigma)$ é um conjunto unitário para todo par (q, σ), onde $q \in Q$ e $\sigma \in \Sigma$.

Neste caso, o que temos na prática é um AFD escrito no "formato" de um AFND, uma vez que retiramos todas as possibilidades de não-determinismo. Logo, um AFD é um caso particular de um AFND.

2. Algoritmo de Construção de Subconjuntos

Vimos, na seção anterior, que um AFD é um caso particular de um AFND. Desta forma, as linguagens que são aceitas por algum AFD (as *linguagens regulares*) são aceitas também por algum AFND. Entretanto, será que a recíproca também é verdadeira? Isto é, será que qualquer linguagem L que seja aceita por algum AFND também será aceita por algum AFD ou os AFND's são capazes de aceitar algumas linguagens que estão fora da classe das linguagens regulares?

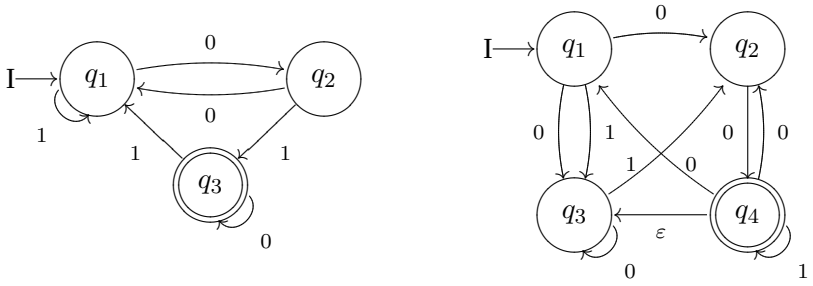
É indiscutível que a transição não-determinística possui muito menos amarras do que a transição determinística. Isto parece indicar que AFND's podem ser capazes de aceitar algumas linguagens que não sejam aceitas por AFD's ao se utilizarem dessa flexibilidade maior de suas funções de transição. Entretanto, provavelmente no sentido inverso da intuição inicial da maioria das pessoas, isto não é verdade.

Mesmo com esta flexibilização nas funções de transição, os AFND's não possuem maior poder computacional do que os AFD's, isto é, se existir um AFND que aceite uma dada linguagem L, também existe um AFD que aceita L.

Vamos mostrar este resultado de forma algorítmica, isto é, dado um AFND $A = (\Sigma, Q, q_0, F, \Delta)$, vamos descrever um algoritmo que permite construir, a partir dos componentes de A, um AFD $A^d = (\Sigma, Q^d, q_0^d, F^d, \delta)$ tal que $L(A^d) = L(A)$. Isto significa que o AFD A^d construído pelo algoritmo aceita exatamente a mesma linguagem que o AFND A original, porém sem fazer uso de qualquer tipo de não-determinismo. Este algoritmo que permite calcular o AFD A^d a partir do AFND A é conhecido como *Algoritmo de Construção de Subconjuntos*.

Para compreendermos como podemos representar o comportamento de um AFND utilizando apenas transições determinísticas, vamos analisar em paralelo a computação de dois autômatos: um determinístico, que foi descrito em um exemplo no capítulo 2, e um não-determinístico, descrito no exemplo acima.

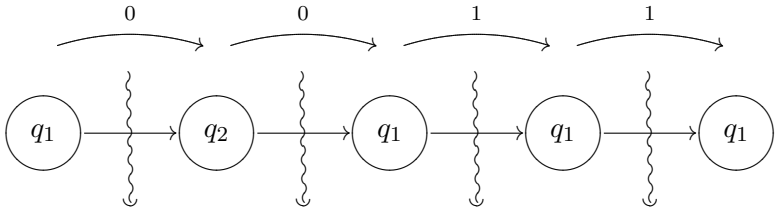
EXEMPLO 4.13. Colocamos abaixo os grafos destes dois autômatos.



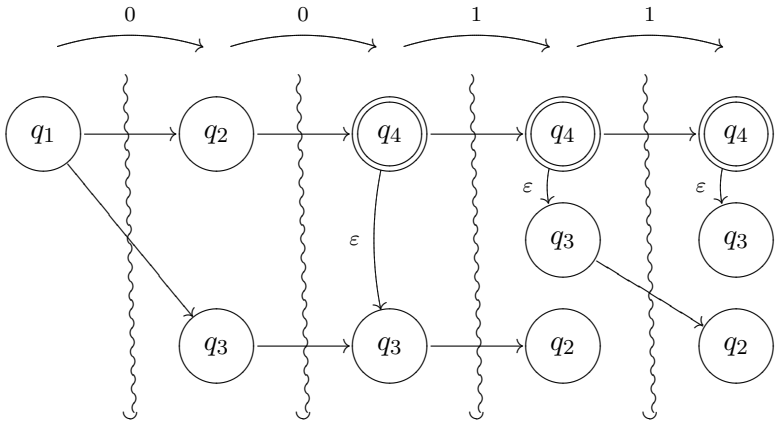
Vamos imaginar agora, apenas para efeito de melhor compreensão, que as computações nestes autômatos são controladas por um relógio que começa a contagem de tempo no instante $t = 0$ e a cada novo "instante" de tempo, aumenta sua contagem em uma unidade, isto é, $t = 1$, $t = 2$ e assim por diante. A cada incremento na contagem do relógio, os autômatos consomem um símbolo da palavra na entrada e realizam uma transição com este símbolo. No caso de um AFND, ele também tem a opção de realizar ε-transições entre os incrementos do relógio. Vamos então, com esta ideia em mente, mostrar abaixo os possíveis estados em que cada um dos autômatos pode estar durante uma computação com a palavra $w = 0011$ em cada

um dos instantes de tempo $t = 0$ até $t = 4$. Como um símbolo da palavra é consumido em cada incremento de tempo, o processamento da palavra é concluído em $t = 4$.

Vamos mostrar primeiro os possíveis estados em que o AFD descrito acima pode estar em cada um dos instantes de tempo.



Vemos que com uma transição determinística, existe sempre exatamente uma única possibilidade de estado em que o autômato pode estar em cada instante de tempo. Vamos agora mostrar os possíveis estados em que o AFND descrito acima pode estar em cada instante de tempo.



Vemos que, no caso de uma transição não-determinística, em alguns instantes de tempo, existem mais do que uma possibilidade de estado onde o autômato pode estar, dependendo das escolhas aleatórias realizadas por eles. Entretanto, analisando o diagrama acima, sabemos que, se o AFND está no estado q_1, então, após

consumir um símbolo 0, ele estará em um dos estados do conjunto $\{q_2, q_3\}$. Analogamente, se o AFND está em um dos estados do conjunto $\{q_2, q_3, q_4\}$, então, após consumir um símbolo 1, ele continuará em um dos estados do conjunto $\{q_2, q_3, q_4\}$.

Desta forma, se passarmos a pensar em termos de *conjuntos de estados* do AFND ao invés de em termos de *estados individuais*, nos tornamos capazes de descrever as transições de forma completamente determinada, como representado abaixo:

$$\{q_1\} \xrightarrow{0} \{q_2, q_3\} \xrightarrow{0} \{q_3, q_4\} \xrightarrow{1} \{q_2, q_3, q_4\} \xrightarrow{1} \{q_2, q_3, q_4\}.$$

Esta é a ideia que utilizamos para construir um AFD A^d que aceita a mesma linguagem que um dado AFND A. Os estados de A^d serão todos os possíveis conjuntos de estados de A. Isto é, se o conjunto de estados de A é Q, então o conjunto de estados de A^d será $\mathcal{P}(Q)$.

Para que uma palavra seja aceita pelo AFND, deve existir ao menos uma computação finita no autômato que processe a palavra inteira e termine em um estado final. Dito de outra forma, uma palavra será aceita se o conjunto de estados onde o autômato pode estar no instante final de tempo (o instante após o processamento do último símbolo) contiver ao menos um estado final do AFND. Desta forma, como queremos que o AFND A e o AFD A^d aceitem as mesmas palavras, se F é o conjunto de estados finais de A e S é um estado de A^d (S é um conjunto de estados de A), S será estado final de A^d se $S \cap F \neq \emptyset$.

Já descrevemos os conjuntos Q^d e F^d do AFD A^d. Para descrevermos q_0^d e δ, precisamos de uma maneira de descrever as ε-transições que podem ocorrer no AFND entre os incrementos do relógio. Seja S um conjunto de estados de A. Definimos o conjunto $E(S)$ como o conjunto de todos os estados alcançáveis a partir de algum estado do conjunto S por uma sequência de *zero ou mais* ε-transições, isto é, $E(S) = \{p \in Q : (q, \varepsilon) \vdash^* (p, \varepsilon), \text{ onde } q \in S\}$.

EXEMPLO 4.14. No AFND do exemplo anterior, temos $E(\{q_1\}) = \{q_1\}$, uma vez que não há ε-transições saindo de q_1. Analogamente, $E(\{q_2\}) = \{q_2\}$ e $E(\{q_3\}) = \{q_3\}$. Já no caso de q_4, temos uma ε-transição saindo para q_3. Então,

$E(\{q_4\}) = \{q_3, q_4\}$. Com estes conjuntos, podemos calcular $E(S)$ para qualquer conjunto S de estados do AFND ($S \subseteq Q$). Por exemplo, $E(\{q_2, q_4\}) = \{q_2, q_3, q_4\}$.

OBSERVAÇÃO. Repare que, para qualquer conjunto S, sempre temos $S \subseteq E(S)$, uma vez que consideramos a possibilidade de *zero* ou mais ε-transições.

O estado inicial q_0^d de A^d será o conjunto de todos os estados onde o AFND A poderá estar no instante de tempo $t = 0$. Este conjunto é formado pelo estado inicial q_0 de A e por todos os estados que podem ser alcançados a partir de q_0 por ε-transições. Isto é, $q_0^d = E(\{q_0\})$.

Finalmente, vamos descrever as transições de A^d, isto é, a função δ. Se S é um estado de A^d (S é um conjunto de estados de A), e $\sigma \in \Sigma$, o estado $T = \delta(S, \sigma)$ será justamente o conjunto de possíveis estados em que o AFND A poderá estar no próximo instante de tempo, como ilustrado no diagrama acima. Para calcular T, procedemos em duas etapas. Primeiramente, para cada estado $q \in S$ de A, calculamos o conjunto de estados $\Delta(q, \sigma)$ e tomamos a união de todos estes conjuntos. Vamos denotar este novo conjunto por S'. S' contém os possíveis estados em que o AFND A poderá estar após o incremento no relógio. Entretanto, este não é ainda o conjunto de todos os possíveis estados onde A pode estar neste novo instante de tempo, pois nos resta ainda a possibilidade de A realizar ε-transições. Desta forma, o conjunto T será $T = E(S')$.

Resumidamente, temos abaixo a lista dos componentes Q^d, q_0^d, F^d e δ de A^d dados em função dos componentes Q, q_0, F e Δ do AFND original A.

- $Q^d = \mathcal{P}(Q)$;
- $q_0^d = E(\{q_0\})$;
- $F^d = \{S \in Q^d : S \cap F \neq \emptyset\}$ e
- $\delta(S, \sigma) = E(\cup_{q \in S} \Delta(q, \sigma))$.

Vemos pela construção acima que, apesar de todo AFND possuir um AFD equivalente, no sentido de que ambos aceitam a mesma linguagem, o AFD poderá

ter um tamanho exponencialmente maior do que o AFND original (já que se $|Q| = n$, $|Q^d| = |\mathcal{P}(Q)| = 2^n$).

Entretanto, nem sempre todos os estados no conjunto $\mathcal{P}(Q)$ são realmente necessários. Todos os estados de Q^d que não podem ser alcançados a partir do estado inicial q_0^d podem ser removidos de A^d sem nenhum prejuízo. Isto nos sugere que a melhor abordagem para a construção do AFD A^d é uma abordagem incremental. Começamos com o estado inicial q_0^d e calculamos as suas transições. Vemos quais novos estados aparecem como resultado destas transições e repetimos este processo com os novos estados. Continuamos assim até que não surjam mais novos estados como resultado do cálculo das transições.

Vamos ilustrar esta abordagem incremental através de exemplos.

EXEMPLO 4.15. Vamos construir o AFD equivalente ao AFND descrito no exemplo acima. Começamos calculando o estado inicial $q_0^d = E(\{q_0\}) = E(\{q_1\}) = \{q_1\}$, já que não existem ε-transições saindo de q_1. Logo, vamos iniciar nossa tabela de transições da função δ do nosso AFD com a linha

δ	0	1
$\{q_1\}$		

Vamos calcular $\delta(\{q_1\}, 0)$ e $\delta(\{q_1\}, 1)$.

$$\delta(\{q_1\}, 0) = E(\cup_{q\in\{q_1\}}\Delta(q, 0)) = E(\Delta(q_1, 0)) = E(\{q_2, q_3\}) = \{q_2, q_3\}.$$

$$\delta(\{q_1\}, 1) = E(\cup_{q\in\{q_1\}}\Delta(q, 1)) = E(\Delta(q_1, 1)) = E(\{q_3\}) = \{q_3\}.$$

Assim, surgiram dois novos estados após o cálculo destas transições: $\{q_2, q_3\}$ e $\{q_3\}$. Adicionamos estes estados à tabela e continuamos os cálculos para estes novos estados.

δ	0	1
$\{q_1\}$	$\{q_2, q_3\}$	$\{q_3\}$
$\{q_2, q_3\}$		
$\{q_3\}$		

$$\delta(\{q_2, q_3\}, 0) = E(\cup_{q\in\{q_2,q_3\}}\Delta(q, 0)) = E(\Delta(q_2, 0) \cup \Delta(q_3, 0)) =$$

$$= E(\{q_4\} \cup \{q_3\}) = E(\{q_3, q_4\} = \{q_3, q_4\}.$$

$$\delta(\{q_2, q_3\}, 1) = E(\cup_{q \in \{q_2, q_3\}} \Delta(q, 1)) = E(\Delta(q_2, 1) \cup \Delta(q_3, 1)) =$$

$$= E(\emptyset \cup \{q_2\}) = E(\{q_2\}) = \{q_2\}.$$

$$\delta(\{q_3\}, 0) = E(\cup_{q \in \{q_3\}} \Delta(q, 0)) = E(\Delta(q_3, 0)) = E(\{q_3\}) = \{q_3\}.$$

$$\delta(\{q_3\}, 1) = E(\cup_{q \in \{q_3\}} \Delta(q, 1)) = E(\Delta(q_3, 1)) = E(\{q_2\}) = \{q_2\}.$$

Surgiram dois novos estados após o cálculo destas transições: $\{q_3, q_4\}$ e $\{q_2\}$. Adicionamos estes estados à tabela e continuamos os cálculos para estes novos estados.

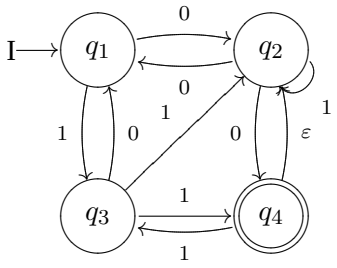
δ	0	1
$\{q_1\}$	$\{q_2, q_3\}$	$\{q_3\}$
$\{q_2, q_3\}$	$\{q_3, q_4\}$	$\{q_2\}$
$\{q_3\}$	$\{q_3\}$	$\{q_2\}$
$\{q_3, q_4\}$		
$\{q_2\}$		

Continuamos os cálculos desta forma até que não surjam novos estados no cálculo das transições. A tabela completa da função δ ficará da seguinte forma:

δ	0	1
$\{q_1\}$	$\{q_2, q_3\}$	$\{q_3\}$
$\{q_2, q_3\}$	$\{q_3, q_4\}$	$\{q_2\}$
$\{q_3\}$	$\{q_3\}$	$\{q_2\}$
$\{q_3, q_4\}$	$\{q_1, q_2, q_3\}$	$\{q_2, q_3, q_4\}$
$\{q_2\}$	$\{q_3, q_4\}$	$\emptyset$
$\{q_1, q_2, q_3\}$	$\{q_2, q_3, q_4\}$	$\{q_2, q_3\}$
$\{q_2, q_3, q_4\}$	$\{q_1, q_2, q_3, q_4\}$	$\{q_2, q_3, q_4\}$
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_1, q_2, q_3, q_4\}$	$\{q_1, q_2, q_3, q_4\}$	$\{q_2, q_3, q_4\}$

Como o único estado final do AFND original era q_4, os estados finais do nosso AFD são $\{q_3, q_4\}$, $\{q_2, q_3, q_4\}$ e $\{q_1, q_2, q_3, q_4\}$.

EXEMPLO 4.16.



Considere o AFND descrito acima. Vamos calcular o AFD equivalente a ele. Começamos calculando $q_0^d = E(\{q_0\}) = E(\{q_1\}) = \{q_1\}$. Agora calculamos incrementalmente a tabela de transição δ.

δ	0	1
$\{q_1\}$	$\{q_2\}$	$\{q_3\}$
$\{q_2\}$	$\{q_1, q_2, q_4\}$	$\{q_2\}$
$\{q_3\}$	$\{q_1\}$	$\{q_2, q_4\}$
$\{q_1, q_2, q_4\}$	$\{q_1, q_2, q_4\}$	$\{q_2, q_3\}$
$\{q_2, q_4\}$	$\{q_1, q_2, q_4\}$	$\{q_2, q_3\}$
$\{q_2, q_3\}$	$\{q_1, q_2, q_4\}$	$\{q_2, q_4\}$

Como o único estado final do AFND original era q_4, os estados finais do nosso AFD são $\{q_1, q_2, q_4\}$ e $\{q_2, q_4\}$. Repare também que, com a abordagem incremental, não precisamos construir os 16 estados de $\mathcal{P}(Q)$. A construção de 6 estados foi suficiente.

3. Exercícios

(1) Desenhe o diagrama de estados de cada um dos seguintes autômatos finitos não determinísticos e construa o autômato finito determinístico equivalente a cada um deles. 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) 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 $L(A)$.

(3) Mostre que todo AFND pode ser convertido em outro equivalente que possui apenas um único estado final.

CAPíTULO 5

Operações com Autômatos Finitos e Linguagens Regulares

No capítulo 3 vimos como obter a expressão regular da linguagem aceita por um autômato finito. Neste capítulo, iremos resolver 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.

Na segunda parte deste capítulo, iremos analisar as chamadas *propriedades de fechamento* das linguagens regulares com relação às seis operações de linguagens que definimos: união, concatenação, estrela de Kleene, interseção, complemento e diferença.

1. Considerações Gerais

Seja r uma expressão regular 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 2, 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 a passo, a partir dos autômatos que aceitam os símbolos de Σ, ε e $\emptyset$.

Contudo, para que isto seja possível, precisamos antes resolver alguns problemas. O primeiro, e mais simples, consiste em construir autômatos finitos que aceitem um símbolo 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

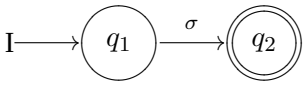
(1) um autômato $\mathcal{M}_u$ que aceita $L(\mathcal{M}_1) \cup L(\mathcal{M}_2)$;

(2) um autômato $\mathcal{M}_c$ que aceita $L(\mathcal{M}_1) \cdot L(\mathcal{M}_2)$; e

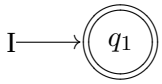
(3) um autômato $\mathcal{M}_1*$ que aceita $L(\mathcal{M}_1)^*$.

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

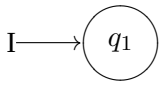
Começaremos descrevendo os átomos desta construção; isto é, os autômatos que aceitam símbolos de Σ, ε e $\emptyset$. Seja $\sigma \in \Sigma$. Um autômato simples que aceita σ é dado pelo grafo



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

$$\mathcal{M} = (\Sigma, Q, q_0, F, \Delta) \text{ e } \mathcal{M}' = (\Sigma, Q', q_0', F', \Delta'). \tag{2.1}$$

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 $\mathcal{M}$.

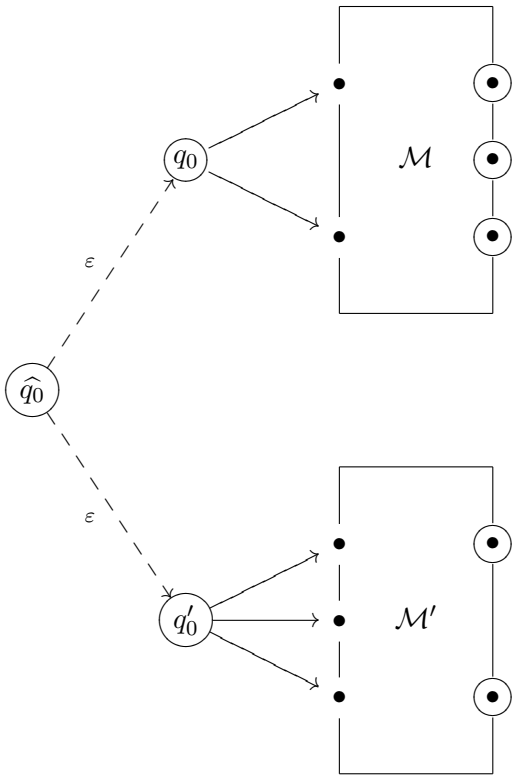
Um outro detalhe que vale a pena mencionar é que, tanto nesta construção, como na da concatenação, estamos supondo que *ambos os autômatos estão definidos para um mesmo alfabeto*. Novamente, esta restrição é apenas aparente já que os autômatos não precisam ser determinísticos. De fato, podemos aumentar 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 símbolos 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 $\widehat{q_0}$ 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. Para isto, basta acrescentarmos uma ε-transição do novo estado inicial para cada um dos estados iniciais q_0 e q'_0.

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 $\widehat{q_0}$. Vamos formalizar a construção, listando os vários elementos de $\mathcal{M}_u$ um a um:

Alfabeto: Σ;

Estados: $\{\widehat{q_0}\} \cup Q \cup Q'$;

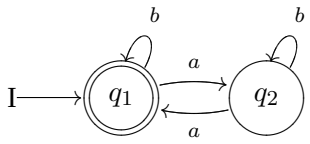
Estado inicial: $\widehat{q_0}$;

Estados finais: $F \cup F'$;

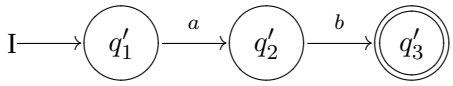
Função de transição:

$$\Delta_u(q,\sigma) = \begin{cases} \{q_0, q_0'\}, & \text{se} \quad q = \widehat{q_0} \quad \text{e} \quad \sigma = \varepsilon \\ \emptyset, & \text{se} \quad q = \widehat{q_0} \quad \text{e} \quad \sigma \neq \varepsilon \\ \Delta(q,\sigma), & \text{se} \quad q \in Q \\ \Delta'(q,\sigma), & \text{se} \quad q \in Q' \end{cases}$$

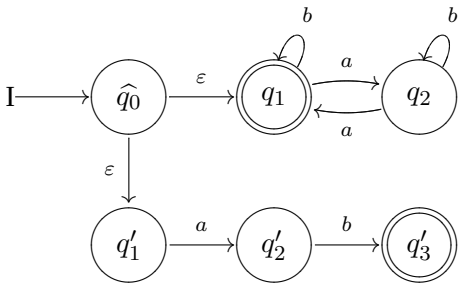
EXEMPLO 5.1. Vejamos como a construção se comporta em um exemplo. Considere o autômato finito determinístico $\mathcal{M}_{\text{par}}$, com grafo



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



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



A chave para entender a razão do funcionamento desta construção está em perceber que inicialmente as únicas transições possíveis no autômato são as ε-transições a partir do estado inicial. Ao realizar uma destas transições, o autômato ficará para sempre "preso" em um dos seus dois setores: o setor que é uma cópia de $\mathcal{M}_{\text{par}}$ ou o setor que é uma cópia de $\mathcal{M}_{ab}$. Assim, ele fica impossibilitado de, no meio do processamento abandonar a simulação de $\mathcal{M}_{\text{par}}$ para continuar de acordo com $\mathcal{M}_{ab}$ ou vice-versa. Ele deverá realizar a simulação completa de $\mathcal{M}_{\text{par}}$ ou de $\mathcal{M}_{ab}$.

3. Concatenação

A segunda operação de linguagens que precisamos transcrever para os autômatos é a concatenação. Suponhamos, mais uma vez, que temos dois autômatos

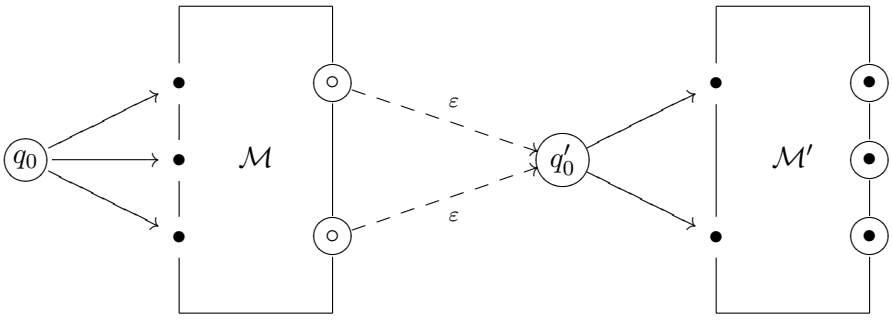
$\mathcal{M}$ e $\mathcal{M}'$ no mesmo alfabeto Σ, cujos elementos são

$$\mathcal{M} = (\Sigma, Q, q_0, F, \Delta) \text{ e } \mathcal{M}' = (\Sigma, Q', q_0', F', \Delta').$$

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 $L(\mathcal{M})$.

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 os estados finais de $\mathcal{M}$ não continuam sendo estados finais em $\mathcal{M}_c$.

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

Alfabeto: Σ;

Estados: $Q \cup Q'$;

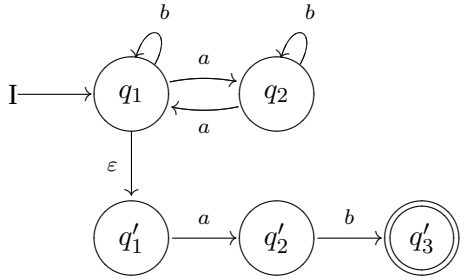
Estado inicial: q_0;

Estados finais: F';

Função de transição:

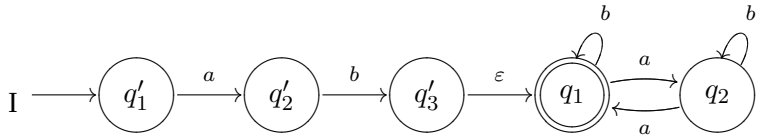
$$\Delta_c(q,\sigma) = \begin{cases} \Delta(q,\sigma) & \text{se} \quad q \in Q \setminus F \\ \Delta(q,\sigma) & \text{se} \quad q \in F \quad \text{e} \quad \sigma \neq \varepsilon \\ \Delta(q,\sigma) \cup \{q_0'\} & \text{se} \quad q \in F \quad \text{e} \quad \sigma = \varepsilon \\ \Delta'(q,\sigma) & \text{se} \quad q \in Q' \end{cases}$$

EXEMPLO 5.2. Vejamos como a construção se comporta quando é aplicada aos autômatos utilizados no exemplo da seção 2. Começaremos concatenando $\mathcal{M}_{\text{par}}$ com $\mathcal{M}_{ab}$:



Observe que o estado q_1 deixou de ser final nesta concatenação.

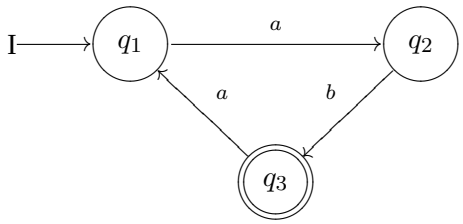
EXEMPLO 5.3. Por outro lado, se concatenarmos $\mathcal{M}_{ab}$ com $\mathcal{M}_{\text{par}}$, então teremos o seguinte grafo resultante:



4. Estrela

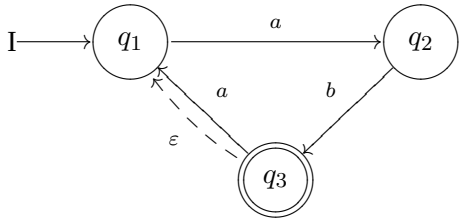
Tendo mostrado como construir um autômato para a concatenação, não parece tão difícil lidar com a estrela. Afinal, se L é uma linguagem, então L^* é obtida concatenando L com ela própria mais e mais vezes; isto é, calculando a união das L^j, para todo $j \geq 0$. Portanto, uma maneira de realizar um 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 ideia em um exemplo e ver o que acontece.

EXEMPLO 5.4. 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 ε-transições entre seus estados finais e o seu estado inicial. Neste exemplo, há apenas o estado final q_3. Por isso precisamos criar uma nova ε-transição de q_3 para q_1, obtendo o seguinte autômato

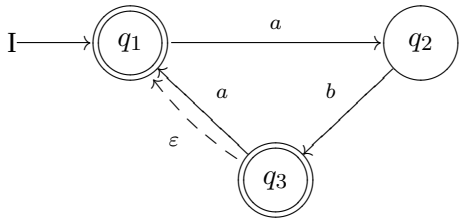


A transição acrescentada foi desenhada tracejada para que possa ser mais facilmente identificada.

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:

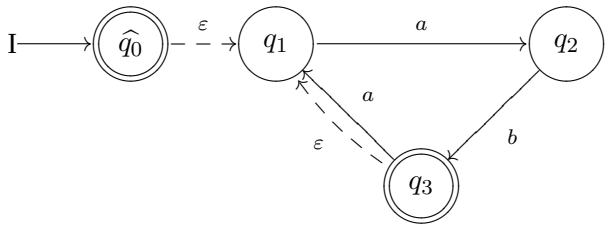


Contudo, este autômato aceita a palavra aba que não pertence a

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

O problema não está em nossa ideia 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 $\widehat{q_0}$ para fazer o papel de estado inicial. Este estado também será um estado final. Além disso, nenhuma transição do novo autômato aponta para $\widehat{q_0}$. Isto garante que a introdução de $\widehat{q_0}$ leva à aceitação de apenas uma nova palavra, que é ε.

EXEMPLO 5.5. Esboçamos abaixo o grafo do autômato resultante desta construção, no exemplo que vimos considerando.

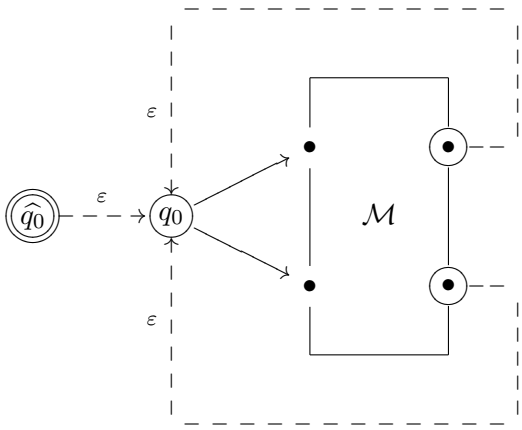


Em geral, quando é dado um autômato

$$\mathcal{M} = (\Sigma, Q, q_0, F, \Delta)$$

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: $\{\widehat{q_0}\} \cup Q$;

Estado inicial: $\widehat{q_0}$;

Estados finais: $F \cup \{\widehat{q_0}\}$

Função de transição:

$$\Delta^*(q,\sigma) = \begin{cases} \{q_0\} & \text{se} \quad q = \widehat{q_0} \quad \text{e} \quad \sigma = \varepsilon \\ \emptyset & \text{se} \quad q = \widehat{q_0} \quad \text{e} \quad \sigma \neq \varepsilon \\ \Delta(q,\sigma) & \text{se} \quad q \in Q \setminus F \\ \Delta(q,\sigma) \cup \{q_0\} & \text{se} \quad q \in F \end{cases}$$

Com a conclusão deste método para obter um AFND equivalente a uma dada expressão regular e com os outros métodos estudados nos capítulos anteriores, obtemos a equivalência entre as três estruturas estudadas até agora: AFD's, AFND's e expressões regulares. Isto pode ser resumido no teorema abaixo.

TEOREMA 5.6. *As seguintes afirmações são equivalentes entre si:*

(1) L é uma linguagem regular.

(2) L é aceita por algum AFD.

(3) L é aceita por algum AFND.

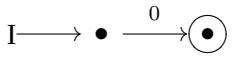
(4) L pode ser gerada por uma expressão regular.

5. Exemplo Mais Extenso

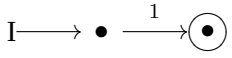
Nesta seção, vamos mostrar, através de um exemplo, como utilizar o método desenvolvido neste capítulo para, a partir de uma expressão regular r, construir um AFND A tal que $L(A) = L(r)$.

EXEMPLO 5.7. Vamos considerar a expressão regular $r = 00^* \cup (10)^*$. Iremos construir o AFND equivalente a esta expressão regular incrementalmente, a partir de AFND's para as expressões regulares atômicas 0 e 1.

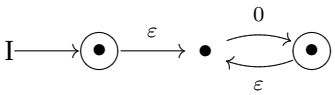
(1) AFND para 0:



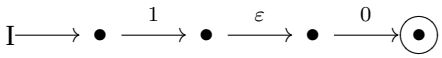
(2) AFND para 1:



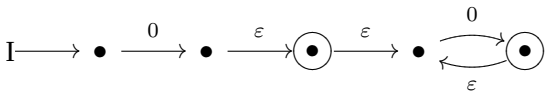
(3) AFND para 0^*:



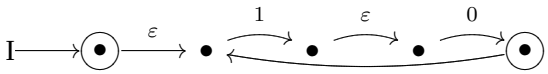
(4) AFND para 10:



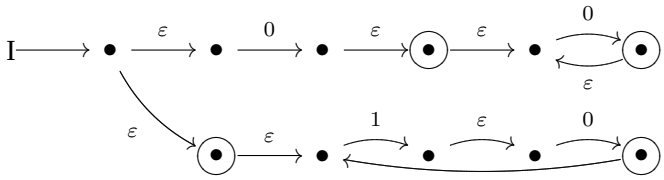
(5) AFND para 00^*:



(6) AFND para $(10)^*$:



(7) AFND para $r = 00^* \cup (10)^*$:



OBSERVAÇÃO. É importante ressaltar que os AFND's obtidos através da aplicação do método descrito neste capítulo não possuem, em geral, o menor número possível de estados. Entretanto, ele é um método geral o suficiente para poder ser aplicado a qualquer expressão regular dada.

6. Propriedades de Fechamento das Linguagens Regulares

Nesta seção, vamos discutir as propriedades de *fechamento* das linguagens regulares com relação às seis operações de linguagens que definimos: união, concatenação, estrela de Kleene, interseção, complemento e diferença.

DEFINIÇÃO 5.8. *Uma classe de linguagens é* fechada *com relação a uma operação de linguagens se, quando tomamos duas linguagens quaisquer (ou apenas uma linguagem no caso das operações de estrela de Kleene e complemento)* definidas sobre o mesmo alfabeto Σ *dentro desta classe e aplicamos a elas esta operação, a linguagem resultante também está dentro desta mesma classe de linguagens.*

OBSERVAÇÃO. É importante ressaltar que estaremos considerando sempre a união, concatenação, interseção e diferença de pares de linguagens definidas sobre o mesmo alfabeto.

(1) **União, concatenação e estrela de Kleene:**

Queremos mostrar que as linguagens regulares são fechadas com relação a estas três operações. Para isso, precisamos mostrar que a união e a concatenação de duas linguagens regulares quaisquer também são linguagens regulares e que a estrela de uma linguagem regular qualquer também é uma linguagem regular.

Já conhecemos três maneiras equivalentes de descrever uma linguagem regular: AFD's, AFND's e expressões regulares. Podemos utilizar qualquer uma destas três estruturas para mostrar o fechamento das linguagens regulares com relação a alguma operação, escolhendo a que for mais conveniente para a operação em questão.

No caso das operações de união, concatenação e estrela, podemos mostrar o resultado de diversas formas. Vamos começar mostrando este resultado de maneira bem simples através de expressões regulares. Se L_1 e L_2 são linguagens regulares, então existem expressões regulares r_1 e r_2 que geram L_1 e L_2, respectivamente. Mas se r_1 e r_2 são expressões regulares, então, pelas regras de formação das expressões regulares, $r_1 \cup r_2$, $r_1.r_2$ e r_1^* também são expressões regulares e estas expressões geram, respectivamente, as linguagens $L_1 \cup L_2$, $L_1.L_2$ e L_1^*. Como existem expressões regulares que geram as linguagens $L_1 \cup L_2$, $L_1.L_2$ e L_1^*, então elas também são linguagens regulares. Logo, a classe das linguagens regulares é *fechada* com relação às operações de união, concatenação e estrela.

Um argumento análogo pode ser feito com a utilização de AFND's. Se L_1 e L_2 são linguagens regulares, então existem AFND's A_1 e A_2 que aceitam L_1 e L_2, respectivamente. Estudamos neste capítulo um método para construir, a partir de AFND's que aceitam L_1 e L_2, AFND's que aceitem $L_1 \cup L_2$, $L_1.L_2$ e L_1^*. Como existem AFND's que aceitam as linguagens $L_1 \cup L_2$, $L_1.L_2$ e L_1^*, então elas também são linguagens regulares.

Finalmente, para o caso específico da operação de união, podemos apresentar um terceiro argumento, que utilizará AFD's. Se L_1 e L_2 são linguagens regulares, então existem AFD's $B_1 = (\Sigma, Q, q_0, F, \delta)$ e $B_2 = (\Sigma, Q', q_0', F', \delta')$ que aceitam L_1 e L_2, respectivamente. Queremos construir um AFD $B_\cup = (\Sigma, Q_\cup, q_{0\cup}, F_\cup, \delta_\cup)$ que aceite $L_1 \cup L_2$.

A ideia para a construção deste autômato é simular em paralelo a computação dos autômatos B_1 e B_2 com uma mesma palavra, executando as transições nos dois autômatos de forma sincronizada. Se pelo menos um dos autômatos aceitar a palavra, ela está na união das linguagens e o nosso novo autômato deverá aceitar. Se os dois autômatos rejeitarem a palavra, então o nosso novo autômato também deverá rejeitá-la. Para implementar esta ideia, os estados do nosso novo autômato serão pares (q, q'), onde q é estado de B_1 e q' é estado de B_2. Temos então:

- $Q_\cup = Q \times Q'$;
- $q_{0_\cup} = (q_0, q'_0)$;
- $F_\cup = F \times Q' \cup Q \times F'$ (B_1 termina em estado final ou B_2 termina em estado final) e
- $\delta_\cup((q, q'), \sigma) = (\delta(q, \sigma), \delta'(q', \sigma))$ (execução sincronizada das transições).

(2) **Interseção:**

Para mostrar que a interseção de duas linguagens regulares também é sempre uma linguagem regular, vamos utilizar um argumento análogo ao último argumento que utilizamos para a operação de união. Se L_1 e L_2 são linguagens regulares, então existem AFD's $B_1 = (\Sigma, Q, q_0, F, \delta)$ e $B_2 = (\Sigma, Q', q'_0, F', \delta')$ que aceitam L_1 e L_2, respectivamente. Queremos construir um AFD $B_\cap = (\Sigma, Q_\cap, q_{0_\cap}, F_\cap, \delta_\cap)$ que aceite $L_1 \cap L_2$. Novamente, a ideia para a construção deste autômato é simular em paralelo a computação dos autômatos B_1 e B_2 com uma mesma palavra, executando as transições nos dois autômatos de forma sincronizada. O que muda em relação à operação de união, é a condição de aceitação do novo autômato. No caso da interseção, se ambos os autômatos aceitarem a palavra, ela está na interseção das linguagens e o nosso novo autômato deverá aceitar. Se pelo menos um dos autômatos rejeitar a palavra, então o nosso novo autômato também deverá rejeitá-la. Temos então:

- $Q_\cap = Q \times Q'$;

- $q_{0\cap} = (q_0, q'_0)$;
- $F_\cap = F \times F'$ (B_1 e B_2 terminam em estado final) e
- $\delta_\cap((q, q'), \sigma) = (\delta(q, \sigma), \delta'(q', \sigma))$ (execução sincronizada das transições).

(3) **Complemento:**

Se L é uma linguagem regular, então existe um AFD $A = (\Sigma, Q, q_0, F, \delta)$ que aceita L_1. Para obtermos um AFD que aceita $\overline{L}$, basta alterarmos o conjunto de estados finais do autômato A, obtendo o autômato $A' = (\Sigma, Q, q_0, F', \delta)$, onde $F' = Q - F$. Isto é, basta invertermos os estados finais e os não-finais.

OBSERVAÇÃO. Este raciocínio da inversão de estados finais e não-finais para a obtenção de um autômato que aceite o complemento da linguagem aceita pelo autômato original só funciona com autômatos *determinísticos*.

(4) **Diferença:**

Sejam L_1 e L_2 linguagens regulares. Queremos mostrar que $L_1 - L_2$ também é sempre uma linguagem regular. Temos que $L_1 - L_2 = L_1 \cap \overline{L_2}$. Mas já mostramos acima que se L_2 é regular, então $\overline{L_2}$ também será regular. Por outro lado, também mostramos acima que se L_1 e $\overline{L_2}$ são regulares, então $L_1 \cap \overline{L_2}$ também será regular. Então $L_1 - L_2$ também será regular.

7. 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) Determine, usando o algoritmo descrito neste capítulo, 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) Sejam L e L' linguagens regulares. Mostre que $L \setminus L'$ é regular.

(5) 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.

(6) 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 mostrada neste capítulo. 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 $\varepsilon \notin L(M)$ então N aceita $L(M) \cdot L(M')$.

c) Se $\varepsilon \in L(M)$ então pode ser necessário acrescentar mais uma transição para que o autômato aceite $L(M) \cdot L(M')$. Que transição é esta?

(7) 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!

(8) 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.

a) Construa expressões regulares para L_1 e L_2.

b) A partir destas expressões regulares, construa autômatos finitos não determinísticos que aceitem L_1 e L_2.

c) Use os algoritmos deste capítulo para criar autômatos finitos não determinísticos que aceitem $L_1 \cup L_2$, $L_1 \cdot L_2$, L_1^* e L_2^*.

(9) Explique por que o raciocínio da inversão de estados finais e não-finais para a obtenção de um autômato que aceite o complemento da linguagem aceita pelo autômato original pode não funcionar quando estamos utilizando autômatos *não-determinísticos*.

(10) 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.

(a) Mostre como é possível construir, a partir de um autômato finito determinístico M qualquer, um autômato finito determinístico M' *sem* reinício tal que $L(M) = L(M')$.

(b) Mostre que, para um autômato finito sem reinício, a ideia original apresentada neste capítulo para a construção do autômato que aceita

a estrela de uma linguagem irá funcionar, ao contrário do que mostramos no caso geral.

(11) Cada uma das linguagens a seguir é a interseção de duas linguagens mais simples. Em cada caso, construa AFD's para as linguagens mais simples e depois combine-as utilizando a construção que estudamos neste capítulo para obter um AFD para a linguagem dada. Em todos os casos, $\Sigma = \{a, b\}$.

(a) Palavras que possuem pelo menos dois a's e pelo menos três b's.

(b) Palavras que possuem um número par de a's e no máximo três b's.

(c) Palavras que possuem comprimento ímpar e uma quantidade par de b's.

(12) Cada uma das linguagens a seguir é o complemento de uma linguagem mais simples. Em cada caso, construa um AFD para a linguagem mais simples e depois transforme-o utilizando a construção que estudamos neste capítulo para obter um AFD para a linguagem dada. Em todos os casos, $\Sigma = \{a, b\}$.

(a) Palavras que não contém exatamente dois a's.

(b) Palavras que não contém a subpalavra ba.

(c) Palavras que não contém 3 a's consecutivos.

CAPíTULO 6

Lema do Bombeamento para Linguagens 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.

EXEMPLO 6.1. Considere o autômato M da figura abaixo.

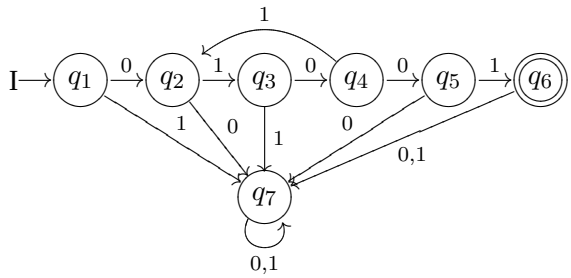


FIGURA 1

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 $y \neq \varepsilon$ 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) $y \neq \varepsilon$;

(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 $y = \varepsilon$ 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 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.

EXEMPLO 6.2. Para entender melhor este ponto, considere o autômato M' da figura 2.

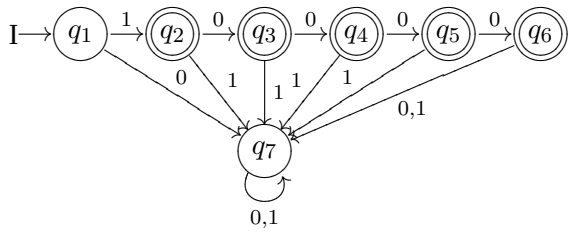


FIGURA 2

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 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 $L \subseteq \Sigma^*$ 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 mesma 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 ou igual ao 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) $y \neq \varepsilon (|y| > 0)$*;*

(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 1, 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

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

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

$$(q_1, w) = (q_1, \sigma_1 \cdots \sigma_n) \vdash^* (q_i, \sigma_i \cdots \sigma_n) \vdash^* (q_j, \sigma_j \cdots \sigma_n) \vdash^*$$

$$\vdash^* (q_n, \sigma_n) \vdash (q_{n+1}, \varepsilon).$$

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

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

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

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

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

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

Como $w = xyz \in L(M)$ 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.

Vamos fazer uma pausa e entender exatamente o que acabamos de provar. Se L é regular, L é aceita por algum AFD. Podemos não saber exatamente que AFD é esse, mas sabemos que ele existe e ele possuirá alguma quantidade positiva n de estados. O lema acima nos diz que, se tomarmos uma palavra de L com comprimento maior ou igual a esta quantidade n, garantidamente existirá para ela uma decomposição satisfazendo as propriedades do lema. Desta forma, o que o lema nos mostra é que, se L é regular, então existe um inteiro positivo n associado a L (que chamamos de comprimento de bombeamento) tal que, toda palavra $w \in L$ onde $|w| \geq n$ possui uma decomposição satisfazendo as propriedades do lema. Em resumo, se L é regular, L satisfaz o lema do bombeamento acima para algum valor inteiro positivo de n.

Podemos pensar nesta última afirmação na forma contrapositiva. Se L é regular, então L satisfaz o lema para algum valor inteiro positivo de n. Logo, se L *não* satisfaz o lema para *nenhum* valor de n, então L não pode ser regular. Será a afirmação nesta forma contrapositiva que utilizaremos para mostrar que algumas linguagens *não são* regulares.

OBSERVAÇÃO. 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 satisfazer o lema do bombeamento para algum inteiro positivo n *não* garante que L seja regular. Portanto, não é possível provar regularidade usando o lema do bombeamento. Voltaremos a discutir este ponto mais adiante em um exemplo.

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 ou igual a n, **existe** uma subpalavra $y \neq \varepsilon$ 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.

EXEMPLO 6.3. 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 for 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 ou igual a n terá que admitir uma subpalavra bombeável. Assim, para obter uma contradição, basta achar **uma** palavra satisfazendo esta condição de comprimento 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 6.4. 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 consequência 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 ou igual a n em L_{primos}. Para fazer isto, basta escolher um primo $q > n$. A existência de um tal primo é consequência 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 $y \neq \varepsilon$ 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

$$xy^kz = 0^i(0^j)^k0^{q-i-j} = 0^{i+jk+(q-i-j)} = 0^{q+(k-1)j}$$

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 suspeitas 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 6.5. Nosso próximo exemplo é a linguagem

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

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 sequência 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 ou igual a 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 6.6. 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

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

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 6.7. 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 ou igual a* n, de modo que as possibilidades para uma decomposição da forma $w = xyz$ sejam bastante limitadas.
- Bombeie y e mostre que se $xy^k z \in L$ então uma contradição é obtida para pelo menos um valor de $k \geq 0$.

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 $xy^k z \in L$ 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 panaceia infalível. Para ilustrar isto, vamos considerar mais um exemplo.

EXEMPLO 6.8. Seja L a linguagem no alfabeto $\{a, b, c\}$ formada pelas palavras da forma $a^i b^j c^r$ 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.

Como vimos no capítulo anterior, a interseção de duas linguagens regulares também é uma linguagem regular. Para provar que a linguagem acima não é regular, precisamos deste resultado em conjunto com o lema do bombeamento. Seja L_1 a linguagem acima. Vamos considerar L_2 a linguagem gerada pela expressão regular ab^*c^*. Logo, L_2 é regular. Se L_1 fosse regular, então, pelo fechamento das linguagens regulares por interseção, $L_1 \cap L_2$ também seria regular. Mas $L_1 \cap L_2 = \{ab^j c^j : j \geq 0\}$, que não é regular como pode ser mostrado utilizando-se o lema do bombeamento de forma análoga ao seu uso no exemplo 6.5. Desta forma, temos uma contradição e L_1 não é regular.

Finalizamos o capítulo com um exemplo final de aplicação do lema do bombeamento.

EXEMPLO 6.9. Considere a linguagem $L = \{0^{j^2} : j \geq 0\}$. Vamos mostrar que ela não é regular. Suponha, por contradição, que ela seja aceita por um autômato finito determinístico com n estados. Vamos considerar a palavra $w = 0^{n^2}$. $|w| = n^2 \geq n$, já que $n \geq 1$. Então, pelo lema do bombeamento, w admite uma decomposição $w = xyz$ satisfazendo as propriedades do lema.

Vamos considerar a palavra xy^2z. Temos $|xy^2z| = |xyz| + |y| = |w| + |y| = n^2 + |y|$. Como $|y| > 0$, temos que $|xy^2z| > |w| = n^2$.

Como $|x| \geq 0$, temos que $|y| \leq |xy|$. Por outro lado, pela hipótese do lema, temos $|xy| \leq n$. Assim, $|y| \leq |xy| \leq n$. Isto pode ser usado no seguinte cálculo. $|xy^2z| = |w| + |y| \leq n^2 + n = n(n+1) < (n+1)^2$.

Reunindo os resultados, temos $n^2 < |xy^2z| < (n+1)^2$. Isto significa que o comprimento de xy^2z está entre dois quadrados consecutivos. Isto é, o comprimento de xy^2z não é um quadrado perfeito, o que nos diz que $xy^2z \notin L$. Desta forma, temos uma contradição com o lema do bombeamento que afirma que $xy^kz \in L$ para todo $k \geq 0$. Assim, L não é uma linguagem regular.

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) $\{wxw^r : w, x \in \{0,1\}^* \setminus \{\varepsilon\}\}$.

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 7

Gramáticas Regulares

Neste capítulo, vamos estudar um novo tipo de estrutura, as *gramáticas*. Vamos então apresentar uma classe de gramáticas, as chamadas *gramáticas regulares* ou *gramáticas lineares à direita*, que podem ser usadas para descrever a classe de linguagens que estamos estudando, as linguagens regulares.

O estudo de gramáticas neste capítulo servirá como uma ponte entre o nosso estudo das linguagens regulares nesta parte das notas e o estudo da classe de linguagens seguinte, que se iniciará no próximo capítulo. Esta nova classe de linguagens é a classe das *linguagens livres de contexto* e ela será inicialmente definida através de gramáticas.

1. Gramáticas

Nesta seção, apresentaremos a definição de *gramática* com grande nível de generalidade. Na próxima seção e no próximo capítulo, estudaremos classes específicas de gramáticas.

DEFINIÇÃO 7.1. *Uma* gramática G *é uma quádrupla* $G = (T, V, S, R)$*, onde:*

- T *é um* conjunto finito *de símbolos, chamados de* símbolos terminais*;*
- V *é um* conjunto finito *de símbolos, chamados de* variáveis *ou* símbolos não-terminais*;*
- $S \in V$ *é o* símbolo inicial *e*
- R *é um* conjunto finito *de* regras.

DEFINIÇÃO 7.2. *Uma* regra *de uma gramática (um elemento do conjunto* R *da gramática) tem o formato* $u \to v$*, onde:*

- u *e* v *são palavras no alfabeto* $(T \cup V)$*, isto é,* $u, v \in (T \cup V)^*$ *e*
- u *contém pelo menos um símbolo do conjunto* V*, isto é,* $u \notin T^*$*.*

DEFINIÇÃO 7.3. *Dada uma gramática $G = (T, V, S, R)$ e dados $x, y \in (T \cup V)^*$, dizemos que y pode ser* derivado em um passo *a partir de x em G, notação $x \Rightarrow y$, ou que $x \Rightarrow y$ é uma* derivação em um passo *em G, se:*

(1) $x = x'.u.x''$*;*

(2) $y = x'.v.x''$ *E*

(3) $u \to v \in R$ *($u \to v$ é uma regra de G),*

onde $u, v, x', x'' \in (T \cup V)^*$.

O que a definição acima nos diz é que uma derivação em um passo substitui em x uma subpalavra que ocorre como lado esquerdo de uma regra de G pelo lado direito desta mesma regra.

OBSERVAÇÃO. Repare na diferença de notação. A seta $\to$ é utilizada em regras da gramática, enquanto a seta $\Rightarrow$ é utilizada em derivações em um passo.

DEFINIÇÃO 7.4. *Uma* derivação $x \Rightarrow^* y$ *é uma sequência de* zero ou mais *derivações em um passo de forma que*

$$x \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \ldots \Rightarrow y.$$

DEFINIÇÃO 7.5. *Definimos a* linguagem gerada *por uma gramática $G = (T, V, S, R)$, denotada por $L(G)$, como sendo o conjunto de palavras*

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

isto é, o conjunto das palavras formadas somente por símbolos terminais *para as quais existe alguma derivação em G a partir do símbolo inicial S.*

OBSERVAÇÃO. É importante lembrar que, pela definição das regras de uma gramática, o lado esquerdo de uma regra nunca contém apenas símbolos terminais. Isto significa que, quando uma palavra w formada apenas por terminais é produzida por uma derivação, a derivação necessariamente se encerra nesta palavra, já que nenhuma subpalavra de w irá casar com nenhum lado esquerdo de alguma regra para mais uma derivação em um passo.

2. Gramáticas Regulares

Nesta seção, apresentamos a primeira classe de gramática com que trabalharemos. São as chamadas *gramáticas regulares*. Elas possuem este nome pois, como veremos mais adiante, as linguagens que são geradas pelas gramáticas regulares são justamente as linguagens regulares que temos estudado. As gramáticas regulares também são chamadas de *gramáticas lineares à direita*.

A definição de qualquer classe de gramática é feita através de alguma restrição nos tipos de regra que podem aparecer no conjunto de regras da gramática. As gramáticas regulares são as que impõe as restrições mais fortes às regras que são permitidas.

DEFINIÇÃO 7.6. *Uma* gramática regular *é uma gramática* $G = (T, V, S, R)$ *onde todo elemento de* R *tem um dos seguintes formatos:*

(1) $X \to aY$*;*

(2) $X \to a$ *OU*

(3) $X \to \varepsilon$*,*

onde $X, Y \in V$ *e* $a \in T$.

De acordo com os formatos acima, concluímos que do lado esquerdo de qualquer regra de uma gramática regular pode aparecer apenas uma variável isolada e do lado direito de qualquer regra de uma gramática regular pode aparecer ou ε ou um terminal isolado ou um único terminal seguido de uma única variável.

EXEMPLO 7.7. A gramática $G = (T, V, S, R)$ onde $T = \{0, 1\}$, $V = \{X, Y, Z\}$, $S = X$ e $R = \{X \to 0X, X \to 1Y, Y \to 0; Y \to 1X, Y \to 1Z, Z \to \varepsilon\}$ é uma gramática regular, já que todas as suas regras se encaixam em um dos três formatos da definição acima.

Conforme vimos na seção anterior, para determinar que uma palavra $w \in T^*$ pertence a $L(G)$ precisamos construir uma derivação a partir do símbolo inicial S que termine na palavra w. No exemplo atual, o símbolo inicial é a variável X, logo

a seguinte derivação mostra que $0011 \in L(G)$:

$$X \Rightarrow 0X \Rightarrow 00X \Rightarrow 001Y \Rightarrow 0011Z \Rightarrow 0011.$$

No primeiro passo da derivação, casamos a variável X com o lado esquerdo da regra $X \to 0X$ e a substituímos pelo lado direito desta regra. No segundo passo, casamos a variável X que ocorre na palavra $0X$ com o lado esquerdo desta mesma regra e realizamos novamente a mesma substituição. No terceiro passo, casamos a variável X que ocorre na palavra $00X$ com o lado esquerdo da regra $X \to 1Y$ e a substituímos pelo lado direito desta regra. No quarto passo, casamos a variável Y que ocorre na palavra $001Y$ com o lado esquerdo da regra $Y \to 1Z$ e a substituímos pelo lado direito desta regra. Finalmente, no quinto passo, casamos a variável Z que ocorre na palavra $0011Z$ com o lado esquerdo da regra $Z \to \varepsilon$ e a substituímos pelo lado direito desta regra. Como a palavra obtida após este passo de derivação (0011) contém apenas terminais, não é mais possível casar nenhuma subpalavra dela com o lado direito de nenhuma regra e a derivação termina neste ponto. Assim, $0011 \in L(G)$.

Repare que, ao longo da derivação acima, alguma escolhas foram realizadas. Por exemplo, no primeiro passo, casamos a variável X com o lado esquerdo da regra $X \to 0X$, mas poderíamos igualmente ter casado esta variável com o lado esquerdo da regra $X \to 1Y$. O mesmo ocorre no segundo passo. Da mesma forma, outras escolhas de casamento e substituição eram possíveis nos demais passos. Cada conjunto de escolhas vai produzir uma palavra de T^* diferente no final do processo de derivação. Todas estas derivações distintas são corretas e todas as palavras resultantes pertencerão a $L(G)$.

Conforme podemos ver no exemplo de derivação acima, a cada passo da derivação a palavra contém uma única variável, que sempre ocorre na extremidade direita da palavra (a única exceção é a palavra após o último passo da derivação, que não contém nenhuma variável). É devido a essa característica das derivações em gramáticas regulares que elas também são conhecidas como *gramáticas lineares à direita*.

3. Gramáticas Regulares e Autômatos Finitos

Nesta seção, vamos mostrar que toda linguagem gerada por uma gramática regular é uma linguagem regular e que toda linguagem regular pode ser gerada por uma gramática regular.

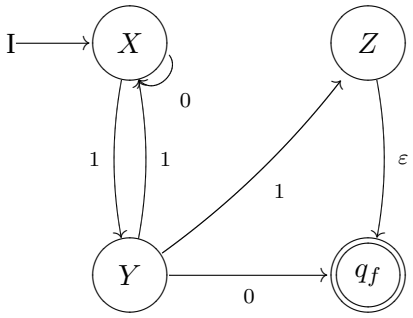
Para mostrar a primeira afirmativa, vamos mostrar que a linguagem gerada por uma gramática regular G é aceita por um AFND. Dada uma gramática regular $G = (T, V, S, R)$, queremos um método para construir um AFND $A = (\Sigma, Q, q_0, F, \Delta)$ tal que $L(A) = L(G)$, isto é, a linguagem aceita por A seja a mesma linguagem gerada por G. Em primeiro lugar, como G gera palavras de T^* e A aceita palavras de Σ^*, se quisermos que as linguagens coincidam, precisamos ter $\Sigma = T$.

Vamos agora aos outros componentes do AFND. Como vimos na seção anterior, a cada passo da derivação a palavra contém sempre uma única variável, com exceção do último passo da derivação, em que a palavra não contém nenhuma variável. De maneira semelhante, a cada passo da derivação um único novo terminal é acrescentado à palavra, com a possível exceção do último passo, que não acrescentará um novo terminal caso a regra utilizada possua ε como lado direito. Assim, podemos considerar os estados do nosso AFND como as variáveis que vão aparecendo na extremidade direita da palavra a cada passo da derivação, com um estado extra para representar o último passo em que nenhuma variável ocorre. O estado inicial será a variável que aparece no início da derivação, isto é, o símbolo inicial S da gramática. Como estado final, usaremos o estado extra que representa justamente o final da derivação, quando não ocorre mais nenhuma variável na palavra. Finalmente, a transição do autômato deve simular cada passo da derivação. Se temos uma palavra em que uma dada variável X aparece na extremidade direita e, após um passo da derivação, foi acrescentado o terminal a à palavra e a variável que agora aparece na extremidade direita da palavra é Y, então temos que fazer $Y \in \Delta(X, a)$. Listamos abaixo todos os componentes do AFND A de acordo com esta ideia:

- $\Sigma = T$;
- $Q = V \cup \{q_f\}$;

- $q_0 = S$;
- $F = \{q_f\}$;
- Δ:
  (1) Para cada regra no formato $X \to aY$, adiciono Y ao conjunto $\Delta(X, a)$;
  (2) Para cada regra no formato $X \to a$, adiciono q_f ao conjunto $\Delta(X, a)$;
  (3) Para cada regra no formato $X \to \varepsilon$, adiciono q_f ao conjunto $\Delta(X, \varepsilon)$.

EXEMPLO 7.8. Vamos construir o AFND equivalente à gramática regular do exemplo acima. O seu grafo é descrito abaixo.



Repare que este autômato possui realmente *não-determinismo*, então não poderíamos utilizar este método, conforme foi apresentado, com AFD's.

Vamos mostrar agora, lado a lado, a derivação que fizemos acima para a palavra 0011 e a computação desta palavra no autômato que acabamos de construir.

$$\begin{array}{rcl|rcl} X & \Rightarrow & 0X & (X, 0011) & \vdash & (X, 011) \\ & \Rightarrow & 00X & & \vdash & (X, 11) \\ & \Rightarrow & 001Y & & \vdash & (Y, 1) \\ & \Rightarrow & 0011Z & & \vdash & (Z, \varepsilon) \\ & \Rightarrow & 0011 & & \vdash & (q_f, \varepsilon) \end{array}$$

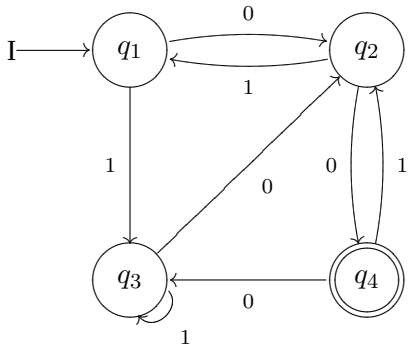
Podemos notar a "sincronia" entre as variáveis que aparecem nas palavras da derivação e nos estados do autômato na computação.

Vamos agora ao sentido inverso. Dado um AFD $A = (\Sigma, Q, q_0, F, \delta)$, quero um método para construir uma gramática regular $G = (T, V, S, R)$ tal que $L(G) = L(A)$. Neste caso, pelo mesmo argumento anterior, precisamos ter $T = \Sigma$.

Vamos agora aos outros componentes da gramática regular, seguindo a mesma ideia do argumento anterior. As variáveis da gramática que vão aparecendo na extremidade direita da palavra durante os passos de uma derivação devem acompanhar os estados do autômato ao longo de sua computação. Listamos abaixo todos os componentes da gramática regular G de acordo com esta ideia:

- $T = \Sigma$;
- $V = Q$;
- $S = q_0$;
- R:
  (1) Se $q' = \delta(q, a)$, adiciono $q \to aq'$ ao conjunto R;
  (2) Se $q \in F$, adiciono $q \to \varepsilon$ ao conjunto R.

EXEMPLO 7.9. Vamos construir a gramática regular ao autômato abaixo.



De acordo com nosso método, temos:

- $T = \Sigma = \{0, 1\}$;
- $V = Q = \{q_1, q_2, q_3, q_4\}$;
- $S = q_0 = q_1$;
- $R = \{q_1 \to 0q_2, q_1 \to 1q_3, q_2 \to 0q_4, q_2 \to 1q_1, q_3 \to 0q_2, q_3 \to 1q_3, q_4 \to 0q_3, q_4 \to 1q_2, q_4 \to \varepsilon\}$.

Vamos mostrar agora, lado a lado, a computação da palavra $w = 01100$ no AFD acima e a derivação desta palavra na gramática que construímos.

$$\begin{array}{rcl|rcl} (q_1, 01100) & \vdash & (q_2, 1100) & q_1 & \Rightarrow & 0q_2 \\ & \vdash & (q_1, 100) & & \Rightarrow & 01q_1 \\ & \vdash & (q_3, 00) & & \Rightarrow & 011q_3 \\ & \vdash & (q_2, 0) & & \Rightarrow & 0110q_2 \\ & \vdash & (q_4, \varepsilon) & & \Rightarrow & 01100q_4 \\ & & & & \Rightarrow & 01100 \end{array}$$

Podemos novamente notar a "sincronia" entre os estados do autômato na computação e as variáveis que aparecem nas palavras da derivação.

Com a conclusão deste método para transformar gramáticas regulares em autômatos finitos e vice-versa, obtemos a equivalência entre as quatro estruturas estudadas até agora: AFD's, AFND's, expressões regulares e gramáticas regulares. Isto pode ser resumido no teorema abaixo, que estende um teorema apresentado anteriormente.

TEOREMA 7.10. *As seguintes afirmações são equivalentes entre si:*

(1) L é uma linguagem regular.

(2) L é aceita por algum AFD.

(3) L é aceita por algum AFND.

(4) L pode ser gerada por uma expressão regular.

(5) L pode ser gerada por uma gramática regular.

4. Exercícios

(1) Determine gramáticas regulares 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 uma gramática regular que gere a seguinte linguagem

$$\{w \in \{0,1\}^* : w \text{ não contém a sequência } 00\}.$$

(3) Seja L uma linguagem regular no alfabeto Σ. Mostre que $L \setminus \{\varepsilon\}$ pode ser gerada por uma gramática regular 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.

(4) Considere uma nova classe de gramáticas que é uma pequena generalização das gramáticas regulares. Nesta classe, ao invés de permitirmos apenas um terminal do lado direito das regras, permitiremos uma palavra formada por terminais. Formalmente, nesta nova classe de gramáticas, as regras devem ter um dos seguintes formatos:

(a) $X \to wY$ OU

(b) $X \to w$,

onde $X, Y \in V$ e $w \in T^*$. Toda gramática regular é um caso particular de gramática desta nova classe. Entretanto, apesar dessa generalidade maior nos formatos das regras, mostre que, se G é uma gramática desta nova classe, sempre existe uma gramática regular G' tal que $L(G') = L(G)$.

(5) Ache autômatos finitos que aceitem as linguagens geradas pelas gramáticas 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) $S \to 0X$, $X \to 1101Y$, $X \to 11X$, $Y \to 11Y$, $X \to 1$.

Parte 2

Linguagens Livres de Contexto

CAPíTULO 8

Linguagens Livres de Contexto e Gramáticas Livres de Contexto

Neste capítulo, continuaremos o nosso estudo das gramáticas. Estudaremos aqui uma segunda classe de gramáticas, que é 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 regulares, as regras são extremamente rígidas: uma variável só pode ser levada na concatenação de algum terminal com alguma variável, sendo que a variável tem que estar à direita do terminal, ou então levada em um terminal isolado ou em ε.

Já as gramáticas livres de contexto, que estudaremos neste capítulo, admitem regras muito mais flexíveis. De fato, a única restrição é que à esquerda da seta só pode aparecer uma variável isolada. Esta é uma restrição que já existia nas gramáticas regulares. Por outro lado, as restrições que existiam nas gramáticas regulares sobre o que poderia ocorrer à direita da seta não existem mais nas gramáticas livres de contexto. Talvez você esteja se perguntando: mas o que mais poderia aparecer à esquerda de uma seta? Voltaremos a esta questão na seção 2, depois de considerar vários exemplos de gramáticas que *são* livres de contexto. Mas antes dos exemplos precisamos dar uma definição formal do que é uma gramática livre de contexto.

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

DEFINIÇÃO 8.1. *Seja G uma gramática com conjunto de terminais T, conjunto de variáveis V e símbolo inicial $S \in V$. Dizemos que G é* livre de contexto *se todas as suas regras são do tipo $X \to w$, onde $X \in V$ e $w \in (T \cup V)^*$.*

Observe que as regras de uma gramática regular se encaixam neste formato. Portanto, toda gramática regular é livre de contexto. Por outro lado, é evidente que nem toda gramática livre de contexto é regular.

EXEMPLO 8.2. Um exemplo simples é dado pela gramática G_1 com terminais $T = \{0, 1\}$, variáveis $V = \{S\}$, símbolo inicial S e conjunto de regras

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

Como do lado esquerdo de cada seta só há uma variável, G_1 é livre de contexto. Contudo, G_1 não é regular porque $S \to 0S1$ tem um terminal à direita de uma variável.

EXEMPLO 8.3. Outro exemplo simples é a gramática G_2 com terminais $T = \{0, 1\}$, variáveis $V = \{S, X\}$, símbolo inicial S e conjunto de regras

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

Mais uma vez, apesar de ser claramente livre de contexto, esta gramática não é regular, devido à 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 gramáticas regulares.

DEFINIÇÃO 8.4. *Seja G uma gramática livre de contexto com terminais T, variáveis V, símbolo inicial S e conjunto R de regras, e sejam $w, w' \in (T \cup V)^*$. Dizemos que w'* pode ser derivada em um passo *a partir de w em G, notação $w \Rightarrow_G w'$, 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 $\mathcal{G}$ que possua esta*

mesma variável no seu lado esquerdo. Em outras palavras, para que $w \Rightarrow_G w'$ *é preciso que:*

(1) exista uma decomposição da forma $w = uXv$*, onde* $u, v \in (T \cup V)^*$ *e* $X \in V$*;*

(2) exista uma regra $X \to \alpha$ *em* R *E*

(3) $w' = u\alpha v$*.*

Como já estamos acostumados a fazer, dispensaremos o 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 regulares, geramos palavras a partir de uma gramática livre de contexto pelo encadeamento de várias derivações em um passo.

DEFINIÇÃO 8.5. *Dizemos que* w' *pode ser* derivada a partir de w em G *se existem palavras* $w_1, \ldots, w_{n-1} \in (T \cup V)^*$ *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* G *e escrevemos* $w \Rightarrow^* w'$.

OBSERVAÇÃO. É conveniente adotar a convenção de que w pode ser derivada dela própria em zero passos, de modo que faz sentido escrever $w \Rightarrow^* w$.

DEFINIÇÃO 8.6. *O conjunto de todas as palavras de* T^* *que podem ser derivadas a partir do símbolo inicial* S *na gramática* G *é a* linguagem gerada por G*. Denotando por* $L(G)$ *a linguagem gerada por* G *e usando a notação definida acima, temos que*

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

Nos capítulos anteriores, estivemos estudando as linguagens regulares. Agora, com as gramáticas livres de contexto, podemos definir uma nova classes de linguagens: as *linguagens livres de contexto*.

DEFINIÇÃO 8.7. *Dizemos que L é uma* linguagem livre de contexto *se existe uma gramática livre de contexto G tal que $L = L(G)$, isto é, se é possível construir uma gramática livre de contexto que gere a linguagem L.*

Vamos analisar agora alguns exemplos concretos.

EXEMPLO 8.8. Seja 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 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 G_1. Já $0^2S1^2 \Rightarrow 0^21^2$ foi obtida trocando-se S por ε. Podemos fazer isto por que $S \to \varepsilon$ também é uma regra de $\mathcal{G}_1$. Toda esta cadeia de derivações pode ser abreviada como

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

Por outro lado, $0^2S1^2 \Rightarrow 0^21^3$ *não* é uma derivação legítima em G_1 porque neste caso a regra usada foi $S \to 1$, que não pertence a G_1.

Do que fizemos acima, segue que $0^21^2 \in L(G_1)$. Mas podemos ser muito mais ambiciosos e tentar determinar todas as palavras de $L(G_1)$. Para começar, note que se $S \Rightarrow^n w$ em 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 G_1 que permite eliminar a variável S é $S \to \varepsilon$, 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,

$$L(G_1) = \{0^n1^n : n \geq 0\}.$$

Já vimos no capítulo 6 que esta linguagem não é regular. Em particular, não existe nenhuma gramática regular que gere $L(G_1)$.

Desta forma, mostramos que toda linguagem regular é uma linguagem livre de contexto, mas nem toda linguagem livre de contexto é uma linguagem regular. Formalizamos isto no teorema abaixo.

TEOREMA 8.9. *A classe das linguagens regulares é um* subconjunto próprio *da classe das linguagens livres de contexto.*

EXEMPLO 8.10. Passando à gramática 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. 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 a partir 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 G_2. Suponhamos que $S \Rightarrow^* w$ em G_2. É fácil ver que em w há um único 1 e 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

$$L(G_2) = \{0^n10^m : n, m \geq 0\}.$$

Mas esta é, na verdade, uma linguagem regular, que pode ser descrita na forma $L(G_2) = 0^*10^*$. Temos assim um exemplo de gramática livre de contexto que não é regular mas que gera uma linguagem regular.

Recapitulando, toda linguagem regular é livre de contexto. Isto ocorre porque uma linguagem regular sempre pode ser gerada por uma gramática regular. Mas, como já vimos, toda gramática regular é também livre de contexto. Por outro lado, nem toda linguagem livre de contexto é regular. Por exemplo, vimos que a linguagem $\{0^n 1^n : n \geq 0\}$ não é regular. No entanto, ela é gerada pela gramática 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 que é satisfeita por todas as linguagens livres de contexto, mas que não é satisfeita por L. Veremos como fazer isto mais adiante nestas notas.

2. Gramáticas que Não São Livres de 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.

EXEMPLO 8.11. Um exemplo simples, mas importante, é a gramática com terminais $\{a, b, c\}$, variáveis $\{S, X, Y\}$, símbolo inicial S e regras

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

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.

Mostraremos mais adiante nestas notas que esta linguagem não pode ser gerada por nenhuma gramática livre de contexto.

3. Mais Exemplos

Nesta seção, veremos mais exemplos de linguagens e gramáticas livres de contexto.

EXEMPLO 8.12. 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 G_{exp} , com conjunto terminais $\{+, *, (,), \mathrm{id}\}$, que gere todas as expressões aritméticas legítimas nos operadores $+$ e $*$. Note que 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:

$$E \to E + E$$

$$E \to E * E$$

$$E \to (E).$$

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 de sublinhar a instância da variável à qual uma regra está sendo aplicada:

$$\underline{E} \Rightarrow E + \underline{E} \Rightarrow \underline{E} + E * E \Rightarrow id + \underline{E} * E \Rightarrow \mathrm{id} + \mathrm{id} * \underline{E} \Rightarrow \mathrm{id} + \mathrm{id} * \mathrm{id}. \tag{3.3}$$

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.

DEFINIÇÃO 8.13. *Seja G uma gramática livre de contexto e $w \in L(G)$. Uma* derivação mais à esquerda *de w em 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 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 parênteses, 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 mais adiante nestas notas.

Analisando 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.

EXEMPLO 8.14. 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 consequência 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 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

ideia seria introduzir $S \to \varepsilon$. 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 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

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

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

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

4. Propriedades de Fechamento das Linguagens Livres de Contexto

Nesta seção, vamos discutir as propriedades de *fechamento* das linguagens livres de contexto com relação às seis operações de linguagens, de forma análoga ao que fizemos para as linguagens regulares.

Novamente, para o caso das operações de união, concatenação, interseção e diferença, consideramos apenas pares de linguagens definidas no mesmo alfabeto.

(1) **União:**

Queremos mostrar que as linguagens livres de contexto são fechadas com relação à operação de união. Para isso, precisamos mostrar que a união de duas linguagens livres de contexto quaisquer também é uma linguagem livre de contexto.

Até o momento, conhecemos apenas uma maneira de descrever linguagens livres de contexto: as gramáticas livres de contexto. Desta forma, procedemos da seguinte maneira. Se L_1 e L_2 são linguagens livres de contexto, então existem gramáticas livres de contexto $G_1 = (T, V_1, S_1, R_1)$ e $G_2 = (T, V_2, S_2, R_2)$ que geram L_1 e L_2, respectivamente (estamos assumindo que $V_1 \cap V_2 = \emptyset$). Queremos construir uma gramática livre de contexto $G_\cup = (T, V_\cup, S_\cup, R_\cup)$ que gere $L_1 \cup L_2$. A ideia para a construção desta gramática é muito semelhante a ideia que utilizamos anteriormente para a construção de um AFND que aceita a união das linguagens aceitas por dois AFND's dados.

Como $G_\cup$ deve gerar uma palavra que pertença a L_1 ou a L_2, o conjunto de regras de $G_\cup$ precisa conter R_1 e 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, $G_\cup$ fica definida pelos seguintes ingredientes:

- $V_\cup = V_1 \cup V_2 \cup \{S\}$;
- $S_\cup = S$ e
- $R_\cup = R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\}$.

Desta forma, como a linguagem $L_1 \cup L_2$ é gerada por uma gramática livre de contexto, ela também é uma linguagem livre de contexto.

(2) **Concatenação:**

Podemos proceder de maneira semelhante ao que fizemos no caso anterior da união para criar uma gramática $G_\bullet = (T, V_\bullet, S_\bullet, R_\bullet)$ 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 um novo símbolo inicial S e $S \to S_1 S_2$ às regras de R_1 e R_2. Os ingredientes da gramática $G_\bullet$ são:

- $V_\bullet = V_1 \cup V_2 \cup \{S\}$;
- $S_\bullet = S$ e
- $R_\bullet = R_1 \cup R_2 \cup \{S \to S_1 S_2\}$.

(3) **Estrela de Kleene:**

Novamente, a construção de uma gramática $G_* = (T, V_*, S_*, R_*)$ que aceite L_1^* é semelhante ao que fizemos nos casos anteriores. A operação estrela nos permite concatenar L_1 com ela própria quantas vezes desejarmos. Como as palavras de L_1 são geradas a partir de S_1, basta acrescentar um novo símbolo inicial S e as regras $S \to S_1 S$ e $S \to \varepsilon$, que nos permitem criar uma sequência de quantas cópias quisermos do símbolo S_1. Os ingredientes da gramática G_* são:

- $V_* = V_1 \cup \{S\}$;
- $S_* = S$ e
- $R_* = R_1 \cup \{S \to S_1 S, S \to \varepsilon\}$.

(4) **Interseção:**

No caso das linguagens regulares, vimos que elas são fechadas com relação a todas as seis operações de linguagens. Vamos começar agora, na operação de interseção, a mostrar que isto não é verdade para as linguagens livres de contexto. De fato, as linguagens livres de contexto são

fechadas apenas para as operações que já analisamos acima: união, concatenação e estrela. Elas não são fechadas para as operações de interseção, complemento e diferença. Isso significa que dadas duas linguagens livres de contexto, talvez a interseção delas ou a diferença delas ou ambas não sejam livres de contexto. Da mesma forma, dada uma linguagem livre de contexto, pode ser que o seu complemento não seja livre de contexto.

Para mostrar que as linguagens livres de contexto não são fechadas por interseção, basta exibirmos um exemplo de duas linguagens L_1 e L_2 tais que L_1 e L_2 são livres de contexto, mas $L_1 \cap L_2$ não é. É isto que faremos a seguir.

Sejam

$$L_1 = \{a^n b^n c^m : m, n \geq 0\} \text{ e}$$

$$L_2 = \{a^m b^n c^n : m, n \geq 0\}.$$

Estas duas linguagens são livres de contexto. Uma gramática livre de contexto $G = (T, V, S, R)$ que gera L_1 é a gramática com $T = \{a, b, c\}$, $V = \{X, Y, Z\}$, $S = X$ e $R = \{X \to YZ, Y \to aYb, Y \to \varepsilon, Z \to cZ, Z \to \varepsilon\}$. Uma gramática livre de contexto análoga gera L_2. Temos que a interseção destas duas linguagens é

$$L_1 \cap L_2 = \{a^n b^n c^n : n \geq 0\}.$$

Conforme já mencionamos anteriormente, esta linguagem não é livre de contexto. Isto será provado mais adiante nestas notas. Assim, as linguagens livres de contexto *não são* fechadas com relação à interseção.

(5) **Complemento:**

Vamos mostrar que as linguagens livres de contexto também não são fechadas com relação ao complemento.

Começamos relembrando uma igualdade entre conjuntos conhecida como *Lei de De Morgan*:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}.$$

Sejam L_1 e L_2 linguagens livres de contexto. Suponha, por contradição, que as linguagens livres de contexto são fechadas com relação ao complemento. Então, $\overline{L_1}$ e $\overline{L_2}$ também são linguagens livres de contexto. Como já vimos anteriormente, as linguagens livres de contexto são fechadas com relação à união, logo $\overline{L_1} \cup \overline{L_2}$ também é livre de contexto. Finalmente, como estamos supondo que as linguagens livres de contexto são fechadas com relação ao complemento, $\overline{\overline{L_1} \cup \overline{L_2}}$ também é livre de contexto. Mas isto é igual a $L_1 \cap L_2$, o que significa que $L_1 \cap L_2$ também é livre de contexto. Logo, se as linguagens livres de contexto são fechadas com relação ao complemento, então elas são fechadas com relação à interseção. Mas isto é uma contradição com o que mostramos acima no caso da interseção. Portanto, as linguagens livres de contexto *não são* fechadas com relação ao complemento.

(6) **Diferença:**

Vamos mostrar que as linguagens livres de contexto também não são fechadas com relação à diferença.

Seja L uma linguagem livre de contexto. Suponha, por contradição, que as linguagens livres de contexto são fechadas com relação à diferença. Temos que $\overline{L} = \Sigma^* - L$. Σ^* é uma linguagem regular (construa um AFD que aceita Σ^*). Logo, Σ^* é livre de contexto. Como estamos supondo que as linguagens livres de contexto são fechadas com relação à diferença, então $\Sigma^* - L$ também é livre de contexto. Mas isto é igual a $\overline{L}$ pela igualdade anterior, o que significa que $\overline{L}$ também é livre de contexto. Logo, se as linguagens livres de contexto são fechadas com relação à diferença, então elas são fechadas com relação ao complemento. Mas isto é uma contradição com o que mostramos acima no caso do complemento. Portanto, as linguagens livres de contexto *não são* fechadas com relação à diferença.

Vamos agora mostrar um exemplo em que utilizamos as construções acima para combinar gramáticas com as operações de união, concatenação e estrela.

EXEMPLO 8.15. 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 regulares bastante simples. Por outro lado, a gramática G_1 definida na seção 1 gera uma linguagem análoga a L_3 e L_4.

Resumimos os ingredientes destas várias gramáticas na tabela abaixo:

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

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

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

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: $\{a, b, c\}$;

Variáveis: $S, S', S'', S_1, S_2, S_3, S_4$;

Símbolo inicial: S;

Regras: $\{S \to S', S \to S'', S' \to S_1 S_4, S'' \to S_3 S_2, S_1 \to aS_1 b, S_1 \to \varepsilon, S_2 \to bS_2 c, S_2 \to \varepsilon, S_3 \to aS_3, S_3 \to \varepsilon, S_4 \to cS_4, S_4 \to \varepsilon\}$.

5. Exercícios

(1) Considere a gramática 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 $L(G)$ 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 G da palavra $b^m a b^n a b^p$.

(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) $\{w : w = w^r$ 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 G cujas regras são

$$S \to 0S1 \,|\, 0S0 \,|\, 1S0 \,|\, 1S1 \,|\, \varepsilon$$

não é regular. Entretanto, $L(G)$ é uma linguagem regular. Ache uma gramática regular G' que gere $L(G)$.

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

(6) Seja G uma gramática livre de contexto com conjunto de terminais T, conjunto de variáveis V e símbolo inicial S. Dizemos que 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 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 $\sigma = \varepsilon$.

(7) Seja G uma gramática livre de contexto que gere uma linguagem L. Mostre como construir, a partir de 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 9

Árvores de Análise Sintática

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 de análise sintática* ou *árvores gramaticais*. Esta noção tem origem na necessidade de diagramar a análise sintática de uma sentença em uma linguagem natural, como o português ou o javanês.

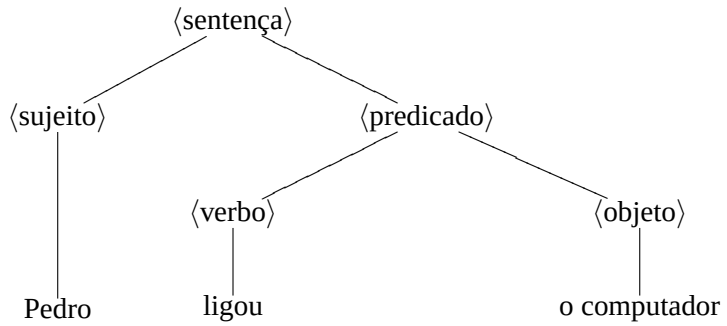
1. Análise Sintática e Linguagens Naturais

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 de análise sintática* ou *árvores gramaticais*.

EXEMPLO 9.1. 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 $\langle \quad \rangle$. Fizemos isto para diferenciar entre o conceito e a palavra da língua portuguesa que o denota. Em outras palavras, $\langle$ sujeito $\rangle$ 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:

$\langle$sentença$\rangle \to \langle$sujeito$\rangle\langle$predicado$\rangle$

$\langle$predicado$\rangle \to \langle$verbo$\rangle\langle$objeto direto$\rangle$

$\langle$sujeito$\rangle \to$ Pedro

$\langle$verbo$\rangle \to$ ligou

$\langle$objeto direto$\rangle \to$ 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 de Análise Sintática

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.

EXEMPLO 9.2. 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*.

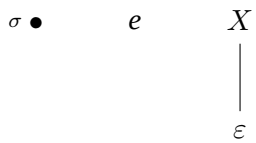
A ordenação entre irmãos, que faz parte da definição de árvore, pode ser estendida a uma ordenação de todas as folhas. Para ver como isto é feito, digamos que f' e f'' são duas folhas de uma árvore. Começamos procurando o seu primeiro ascendente comum, que chamaremos de v. Observe que f' e f'' têm que 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 necessária 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 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* ou *árvores de análise sintática*. 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.

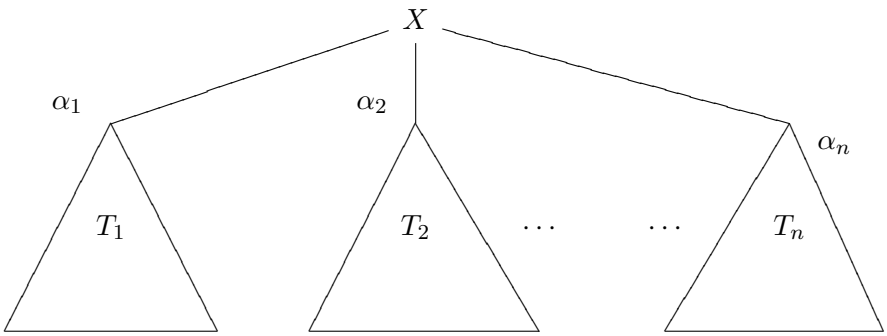
DEFINIÇÃO 9.3. *As* árvores de análise sintática *são definidas recursivamente da seguinte maneira:*

Árvores básicas: *se $\sigma \in T$, $X \in V$ e $X \to \varepsilon$ é uma regra de G, então*



são árvores gramaticais.

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.

EXEMPLO 9.4. Um exemplo simples é a árvore na gramática G_{exp} (definida no capítulo anterior) esboçada abaixo.

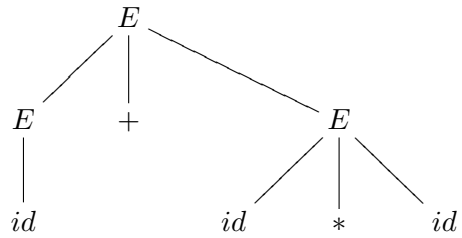


FIGURA 1. Árvore gramatical

Uma consequência 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 $T \cup \{\varepsilon\}$ 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 $\mathcal{T}$ (também chamada de *resultado* da árvore gramatical). Como a definição de árvore é recursiva, assim será a definição de colheita. As colheitas das árvores básicas são

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

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 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 consequência 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 nenhuma ambiguidade nesta maneira de expressar a colheita. Para uma demonstração formal deste fato veja o exercício 1. Portanto, a árvore da figura 1 tem colheita $\mathrm{id} + \mathrm{id} * \mathrm{id}$.

DEFINIÇÃO 9.5. *Digamos que w é uma palavra que pode ser derivada em uma gramática livre de contexto G com símbolo inicial S. Uma* árvore de derivação *para w é uma S-árvore de 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 anterior, vimos como gerar uma palavra formada por terminais a partir do símbolo inicial de uma gramática 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 sequência ordenada de árvores. Isto sugere a seguinte definição.

DEFINIÇÃO 9.6. *Seja 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 w-floresta $\mathcal{F}$ *é uma sequência 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 9.7. *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 $c(\mathcal{T})$.*

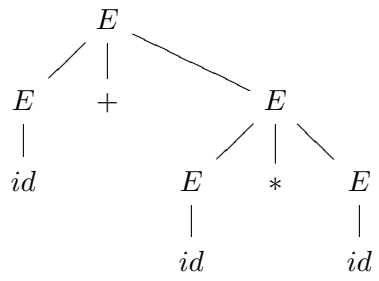
Etapa 1: *Inicialize $\mathcal{F}$ com $\mathcal{T}$.*

Etapa 2: *Se $\mathcal{F}$ é uma w-floresta, então imprima w. Pare 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.*

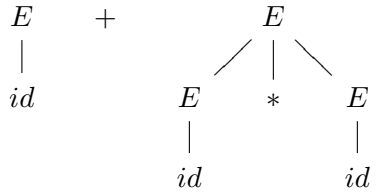
EXEMPLO 9.8. A seguir, você encontrará um exemplo da aplicação passo a passo deste algoritmo a uma árvore na gramática G_{exp}.

Inicialização



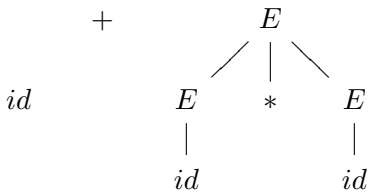
Imprime E

Passo 1



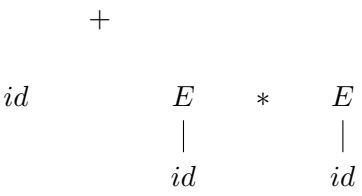
Imprime $E + E$

Passo 2

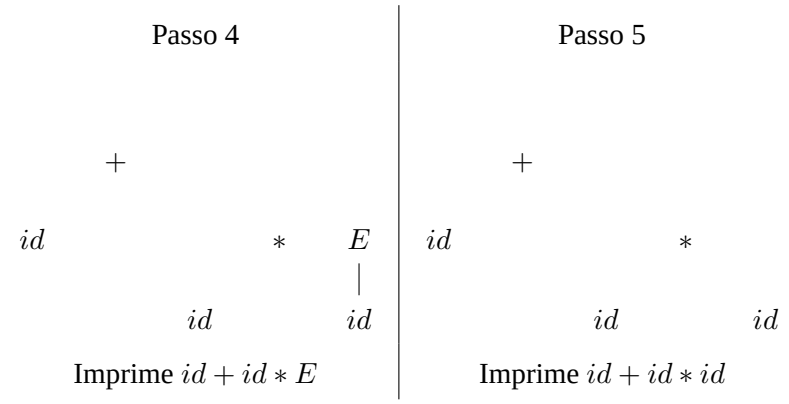


Imprime $id + E$

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 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 para 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 G é uma gramática livre de contexto então toda palavra que tem uma derivação em G é colheita de alguma árvore de derivação de 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 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 9.9. *Seja X uma variável da gramática livre de contexto G. Se existe uma derivação $X \Rightarrow^* w$, então w é colheita de uma X-árvore em G.*

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

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 $Y \in V$ e se existe uma derivação

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

em p passos, então u é colheita de uma Y-árvore de 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 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 9.10. *Seja G uma gramática livre de contexto e $w \in L(G)$. 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 consequência 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. Ambiguidade

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.

EXEMPLO 9.11. 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 G, formalizamos esta noção de dupla interpretação na seguinte definição.

DEFINIÇÃO 9.12. *A gramática G é* ambígua *se existe uma palavra $w \in L(G)$ que admite duas árvores de derivação distintas em 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.

EXEMPLO 9.13. Voltemos por um momento à gramática 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 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 ambígua. A ideia 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:

$$E \to E + F$$

$$E \to F$$

$$F \to F * F$$

$$F \to (E)$$

$$F \to \mathrm{id}$$

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 parênteses.

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

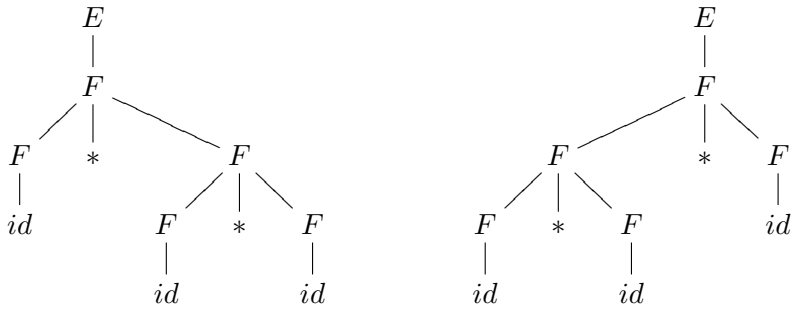


FIGURA 3. Outras duas árvores com mesma colheita

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

inicial E, e as seguintes regras:

$$E \to E + T \quad | \quad T$$

$$T \to T * F \quad | \quad F$$

$$F \to (E) | \quad \text{id}$$

A gramática G'_{exp} não é ambígua, 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 ambiguidade da gramática. Mas isto não é verdade, como mostra o seguinte exemplo.

EXEMPLO 9.14. A gramática G''_{exp} cujas regras são:

$$S \to E$$

$$E \to (E)R \quad | \quad VR$$

$$R \to +E \quad | \quad *E \quad | \quad A$$

$$V \to \text{id}$$

$$A \to \varepsilon$$

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

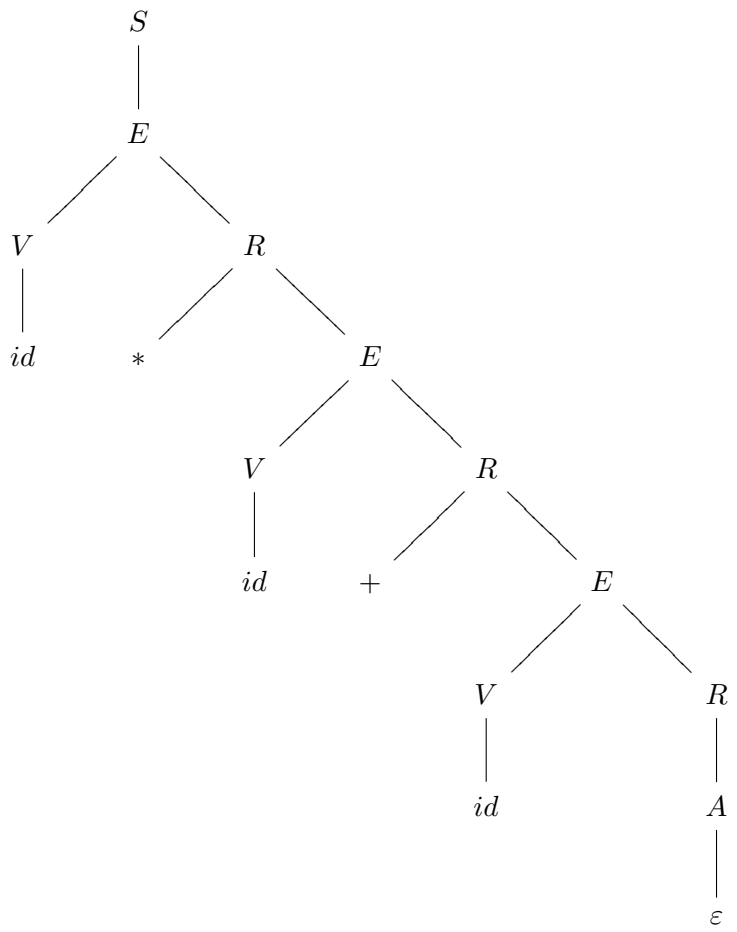
- não há mais de duas regras em G''_{exp} cujo lado esquerdo seja ocupado por uma mesma variável;
- se há duas regras a partir de uma mesma variável, o lado direito de uma começa com uma variável, e o da outra com um terminal;
- cada terminal só aparece uma vez como prefixo do lado direito de alguma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

Imagine, então, que estamos tentando calcular uma derivação à esquerda em 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 sequência de terminais da palavra que está sendo derivada.

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



6. Removendo Ambiguidade

O que fizemos na seção anterior parece indicar que, ao se deparar com uma gramática ambígua, tudo o que temos que fazer é adicionar algumas variáveis e alterar um pouco as regras de maneira a remover a ambiguidade. É verdade que criar novas variáveis e regras para remover ambiguidade pode não ser muito fácil, e é bom não esquecer que não chegamos a provar que a gramática G'_{exp} não é ambígua.

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 gramática é ambígua. De fato, *não pode existir um algoritmo para determinar se uma dada gramática livre de contexto é ou não ambígua.* 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 ambígua. Tais linguagens são chamadas de *inerentemente ambíguas*. Note que ambiguidade é uma propriedade da gramática, mas ambiguidade inerente é uma propriedade da própria linguagem.

EXEMPLO 9.15. Um exemplo bastante simples de linguagem inerentemente ambígua é a linguagem que já vimos no capítulo anterior

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

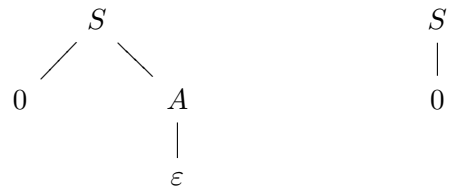
Com isso você descobre porque demos este nome a L_{in}. Não é fácil mostrar que esta linguagem é inerentemente ambígua, e por isso não vamos fazer a demonstração aqui. Os detalhes podem ser encontrados em [**1**, theorem 7.2.2, p. 236].

Para não encerrar o capítulo num clima pessimista, vamos analisar o problema da ambiguidade 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 regulares. Assim a primeira pergunta é: uma gramática regular pode ser ambígua? A resposta é sim, como mostra o seguinte exemplo.

EXEMPLO 9.16. Considere a gramática com terminal $\{0\}$, que tem S e X como variáveis (S é o símbolo inicial) e cujas regras são

$$S \to 0A \quad | \quad 0 \text{ e } A \to 0 \quad | \quad \varepsilon.$$

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



Felizmente, no caso de gramáticas regulares, é sempre possível remover a ambiguidade. Em outras palavras, não existem linguagens regulares inerentemente ambíguas. Além disso, existe um algoritmo simples que, tendo como entrada uma gramática regular, constrói uma outra que gera a mesma linguagem regular mas não é ambígua. O algoritmo é consequência do seguinte fato: a gramática regular construída a partir de um autômato finito determinístico pelo algoritmo do capítulo 7 *não* é ambígua.

Para entender porque isto é verdade, suponha que M seja um autômato finito determinístico e G seja a gramática construída a partir de M pelo algoritmo do capítulo 7. Vimos que as derivações em 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 $L(G) = L(M)$ só tem uma derivação possível. Como G é regular, toda derivação em G é mais à esquerda. Logo G não é ambígua pelo teorema da seção 4.

Diante deste resultado, é claro que, ao receber uma gramática regular G como entrada, o algoritmo procede da seguinte maneira:

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

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

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

Como já vimos, G' não pode ser uma gramática ambígua.

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 G uma gramática livre de contexto e seja X uma variável de G. Seja w uma palavra somente em terminais e que pode ser derivada em 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 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 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 G''_{exp} não é ambígua.

   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.

CAPíTULO 10

Lema do Bombeamento para Linguagens 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 mencionamos anteriormente que a linguagem

$$L_{abc} = \{a^n b^n c^n : n \geq 0\}$$

não é livre de contexto, mas 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 comprimento (em número de arestas) do 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 G uma gramática livre de contexto. A *amplitude* $\alpha(G)$ de uma gramática livre de contexto G é o comprimento máximo das palavras que aparecem à direita de uma seta em uma regra de G.

EXEMPLO 10.1. Para as gramáticas G_{exp} e G''_{exp} definidas no capítulo anterior, temos $\alpha(G_{exp}) = 3$ e $\alpha(G''_{exp}) = 4$.

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 10.2. *Seja G uma linguagem livre de contexto. Se X é uma variável de G e se w é colheita de uma X-árvore de G de altura h então*

$$|w| \leq \alpha(G)^h.$$

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 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.

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 G, sua colheita é um elemento de $L(G)$ no qual os trechos v 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.

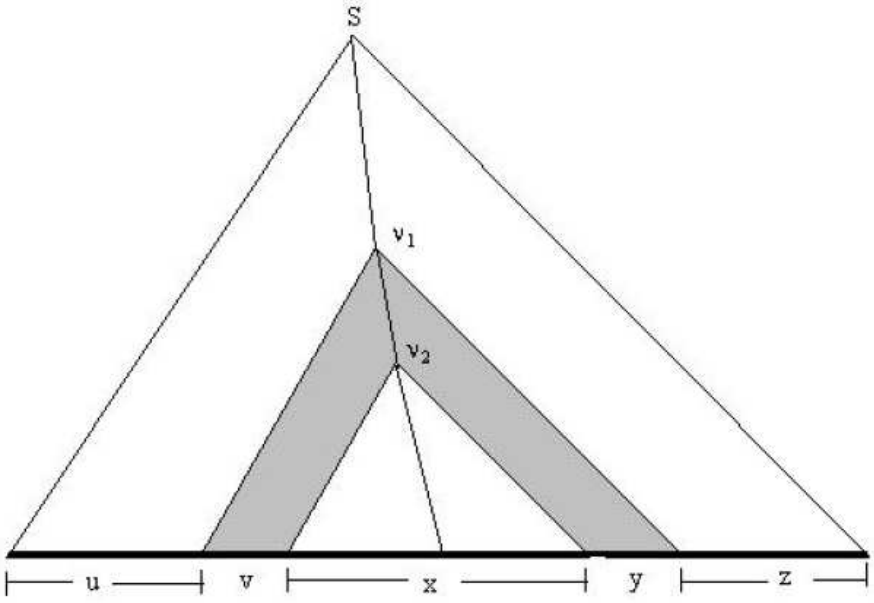


FIGURA 1. Decompondo a palavra para bombear

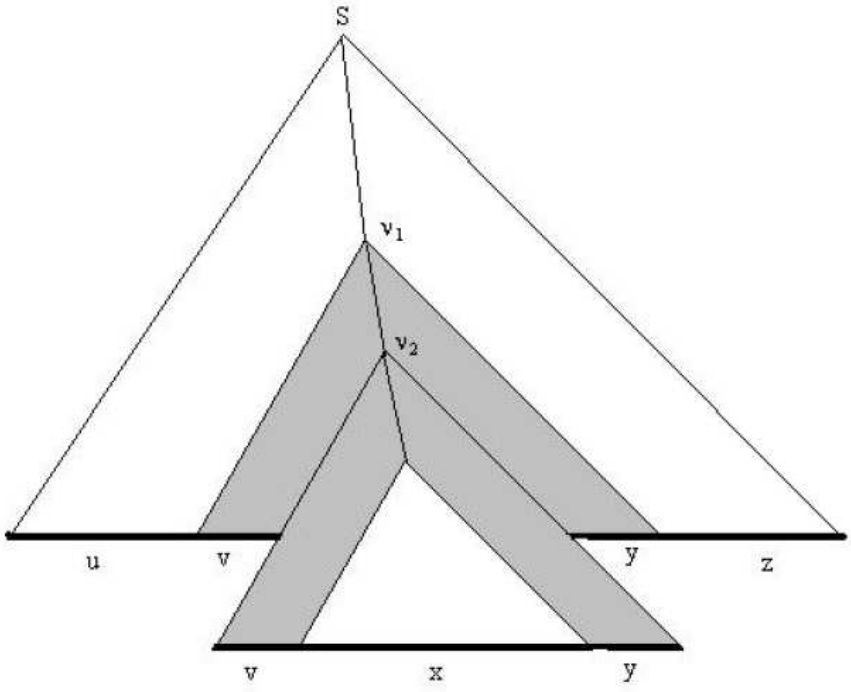


FIGURA 2. Bombeando uma vez

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 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. Como exemplo, uma versão mais sofisticada é discutida em [**2**, Exercício 7.2.3, p. 300].

LEMA DO BOMBEAMENTO. *Seja G uma gramática livre de contexto. Existe um número inteiro ρ, que depende de G, tal que, se $w \in L(G)$ e $|w| \geq \rho$, então existe uma decomposição de w na forma $w = uvxyz$, onde*

(1) $vy \neq \varepsilon$*;*

(2) $|vxy| \leq \rho$*;*

(3) $uv^nxy^nz \in L(G)$*, para todo* $n \geq 0$*.*

DEMONSTRAÇÃO. Suponhamos que G tem k variáveis e que $\alpha(G) \geq 2$; neste caso escolheremos $\rho = \alpha(G)^{k+1}$ (no caso em que $\alpha(G) = 1$, o lema funcionará com a escolha de $\rho = 2$). Seja $w \in L(G)$ uma palavra com comprimento maior ou igual a ρ. Já sabemos, pelo teorema do capítulo anterior, que deve existir ao menos uma árvore de derivação com colheita w. Entre todas as árvores com esta colheita, 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 G.

Como a colheita de $\mathcal{T}$ tem comprimento maior ou igual a $\rho = \alpha(G)^{k+1}$, que é maior do que $\alpha(G)^k$, 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

$$|vxy| \leq \alpha(G)^{k+1} = \rho.$$

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 $vy \neq \varepsilon$. 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, $vy \neq \varepsilon$ 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 ou igual a 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 10.3. Já havíamos mencionado anteriormente que a linguagem

$$L_{abc} = \{a^m b^m c^m : m \geq 0\}$$

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 $m \geq \rho$, então é possível decompor $w = a^m b^m c^m$ na forma $w = uvxyz$, onde:

(1) $vy \neq \varepsilon$;

(2) $|vxy| \leq \rho$;

(3) $uv^n xy^n z \in L_{abc}$ para todo $n \geq 0$.

Como $m \geq \rho$ 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 m, ao passo que o número de cs não foi alterado, e continua sendo m. 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_{abc} não é uma linguagem livre de contexto. Isto conclui também a demonstração, que deixamos indicada anteriormente, de que as linguagens livres de contexto não são fechadas com relação à interseção.

EXEMPLO 10.4. Anteriormente, 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) $vy \neq \varepsilon$;

(2) $|vxy| \leq \rho$;

(3) $uv^nxy^nz \in L_{primos}$ 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

$$u = 0^m,\ v = 0^r,\ x = 0^s,\ y = 0^t \text{ e } z = 0^{p-m-r-s-t}.$$

Além disso, segue de (1) que $r + t > 0$, e de (3) que

$$uv^nxy^nz = 0^m(0^r)^n0^s(0^t)^n0^{p-m-r-s-t} = 0^{p+(r+t)(n-1)}$$

pertence a L_{primos} para todo $n \geq 0$. 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 10.5. Considere, agora, a linguagem

$$L_{rr} = \{rr : r \in (0 \cup 1)^*\}.$$

Já vimos anteriormente 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 ideia já usada quando provamos que esta linguagem não é regular, escolhemos $w = 0^m10^m1$ como nossa primeira tentativa. Entretanto, escolhendo $\rho = 3$ e

decompondo 0^m10^m1 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 $r = 0^m1$.

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 $r = 0^m1^m$.

Supondo, por contradição, que L_{rr} é livre de contexto e escolhendo $w = 0^m1^m0^m1^m$ temos, pelo lema do bombeamento, que se $m \geq \rho$ então w pode ser decomposta na forma

$$0^m1^m0^m1^m = uvxyz$$

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^m1^m. Neste caso, ao tomarmos $n = 2$ no bombeamento, transformaremos o primeiro 0^m1^m em uma palavra maior enquanto mantemos o segundo 0^m1^m inalterado. Precisamos mostrar então que, ao dividirmos a nova palavra uv^2xy^2z ao meio, as duas metades não serão mais iguais. Isto significará que $uv^2xy^2z \notin L_{rr}$ e chegaremos à contradição desejada.

A palavra w original possui comprimento $4m$, logo $2m$ símbolos estão em cada uma das suas metades. Ao bombearmos com $n = 2$ o primeiro 0^m1^m, a palavra uv^2xy^2z passará a ter mais do que $4m$ símbolos. De fato, ela terá $4m + |vy|$ símbolos. Para que uv^2xy^2z tenha alguma chance de pertencer à linguagem, é necessário que $|vy|$ seja par, pois se a palavra possuir comprimento ímpar, ela certamente não possuirá duas metades iguais. Vamos então assumir como hipótese a partir de agora que $|vy|$ é par.

Com estes novos símbolos que são acrescentados pelo bombeamento no primeiro 0^m1^m, os últimos símbolos desta parte da palavra, que pertenciam à primeira

metade da palavra original, serão "empurrados" para a segunda metade da palavra uv^2xy^2z. Como $|vy| \leq \rho \leq m$, no máximo m novos símbolos serão bombeados no primeiro 0^m1^m, logo uma quantidade de símbolos que é certamente menor do que m será "empurrada" da primeira metade da palavra original para a segunda metade da palavra uv^2xy^2z. Como os últimos m símbolos da primeira metade da palavra original são todos 1s, garantidamente apenas 1's serão "empurrados" para o início da segunda metade da palavra uv^2xy^2z. Mas então temos um problema: a primeira metade de uv^2xy^2z começa com 0 e a segunda metade começa com 1. Assim, elas certamente não são iguais e $uv^2xy^2z \notin L_{rr}$, o que é uma contradição.

O caso em que vxy é subpalavra 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 contém 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 10.6. Vamos mostrar que a linguagem

$$L_{a\leq b\leq c} = \{a^ib^jc^k : 0 \leq i \leq j \leq k\}$$

não é livre de contexto.

A prova é bem semelhante à do exemplo da linguagem L_{abc} acima, com apenas uma diferença importante, que iremos destacar.

Suponha, por contradição, que $L_{a\leq b\leq c}$ é livre de contexto. Repare que as palavras $w = a^mb^mc^m$, com $m \geq 0$, pertencem à linguagem acima. Pelo lema do bombeamento, existe um inteiro positivo ρ tal que, se $m \geq \rho$, então é possível decompor $w = a^mb^mc^m$ na forma $w = uvxyz$, onde:

(1) $vy \neq \varepsilon$;

(2) $|vxy| \leq \rho$;

(3) $uv^nxy^nz \in L_{a\leq b\leq c}$ para todo $n \geq 0$.

Como $m \geq \rho$ 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^nxy^nz tem que ser maior que m, ao passo que o número de cs não foi alterado, e continua sendo m. Assim, neste caso, $uv^nxy^nz \notin L_{a \leq b \leq c}$, quando $n > 1$.

Até aqui, tudo está caminhando exatamente como no caso da linguagem L_{abc} que fizemos anteriormente. A diferença entre os dois casos surge agora. Para a linguagem L_{abc}, o caso em que vxy só contém bs e cs podia ser tratado de maneira inteiramente análoga. Neste exemplo, isto não é mais verdade.

No caso em que vxy só contém bs e c, quando $n > 1$, o número de bs ou cs em uv^nxy^nz tem que ser maior que m, ao passo que o número de as não foi alterado, e continua sendo m. No entanto, a palavra obtida neste formato não necessariamente deixa de pertencer a $L_{a \leq b \leq c}$, já que o número de bs e cs pode ser maior que o de a's nas palavras desta linguagem. Desta forma, não obtemos uma contradição neste segundo caso com valores $n > 1$.

O que salva a situação e nos permite obter a contradição é um caso muitas vezes esquecido do lema do bombeamento. A palavra uv^nxy^nz também deve pertencer à linguagem quando $n = 0$, isto é, quando v e y são retirados da palavra original. Este caso do bombeamento é informalmente conhecido como *"bombear para baixo"*.

Quando $n = 0$, o número de bs ou de cs diminui, se tornando menor do que m, ao passo que o número de as não é alterado e continua sendo m. Assim, neste caso, $uv^nxy^nz \notin L_{a \leq b \leq c}$, quando $n = 0$.

Desta forma, obtemos uma contradição para os dois casos em que dividimos a prova. Concluímos, então, que $L_{a \leq b \leq c}$ não é uma linguagem livre de contexto.

4. Exercícios

(1) Mostre que nenhuma das linguagens abaixo é livre de contexto usando o lema do bombeamento.

a) $\{a^{2^n} : n$ 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) $\{0^k 1^k 0^k : k \geq 0\}$;

j) $\{wct : w$ é uma subpalavra de t e $w, t \in \{a, b\}^*\}$;

k) O conjunto de todos os palíndromos no alfabeto $\{0, 1\}$ que contém o mesmo número de 0's e 1's.

CAPíTULO 11

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.

EXEMPLO 11.1. 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 sequência de as e bs que apareceu antes do c. Comparamos, então, esta sequência 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 pare 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 $aabcba$. 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.

EXEMPLO 11.2. Considere a linguagem

$$L_2 = \{vv^R : v \in \{a, b\}^*\},$$

no alfabeto $\{a, b\}$. Ela 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 adivinhar 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, as atitudes 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 com 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.

DEFINIÇÃO 11.3. *Um autômato de pilha não-determinístico (abreviado como AP)* A *é uma 6-upla* $A = (\Sigma, \Gamma, Q, q_0, F, \Delta)$*, onde:*

- Σ *é o* alfabeto da entrada*;*
- Γ *é o* alfabeto da pilha*;*
- Q *é um* conjunto finito *de* estados*;*
- $q_0 \in Q$ *é o* estado inicial*;*
- $F \subseteq Q$ *é o conjunto de* estados finais *e*

- Δ *é a* função de transição *não-determinística. Esta função tem o formato*

$$\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to \mathcal{P}_f(Q \times \Gamma^*).$$

isto é, para cada tripla formada por um estado do conjunto Q, um símbolo do alfabeto Σ ou o símbolo ε e um símbolo do alfabeto Γ ou o símbolo ε, a função de transição fornece como resposta um conjunto finito *de pares formados por um estado do conjunto Q e uma palavra de Γ^*.*

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 porque, 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

$$Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}).$$

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 não-determinístico" 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 acima. Suponhamos que

$$q \in Q, \quad \sigma \in \Sigma \cup \{\varepsilon\} \quad \text{e} \quad \gamma \in \Gamma \cup \{\varepsilon\}.$$

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 (com a convenção de que u é empilhado de forma que seu símbolo *mais à esquerda* fique no topo da pilha).

Além disso, se $\sigma = \varepsilon$ temos que a entrada não é consultada. Já quando $\gamma = \varepsilon$, 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
$\sigma = \varepsilon$	a entrada não é consultada
$\gamma = \varepsilon$	o topo da pilha não é consultado
$\sigma = \gamma = \varepsilon$	nem a entrada, nem o topo da pilha são consultados
$\gamma \neq \varepsilon$ e $u = \varepsilon$	remove γ da pilha
$\gamma = \varepsilon$ e $u \neq \varepsilon$	empilha u
$\gamma = \varepsilon$ e $u = \varepsilon$	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.

EXEMPLO 11.4. 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

$$\Delta(q_1, a, \varepsilon) = \{(q_1, a)\},$$
$$\Delta(q_1, b, \varepsilon) = \{(q_1, b)\}.$$

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

$$\Delta(q_1, c, \varepsilon) = \{(q_2, \varepsilon)\}.$$

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,

$$\Delta(q_2, a, a) = \{(q_2, \varepsilon)\},$$

$$\Delta(q_2, b, b) = \{(q_2, \varepsilon)\}.$$

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.

EXEMPLO 11.5. 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 no topo da pilha e o desempilha
q_2	b	b	(q_2, ε)	acha b na entrada e no topo da pilha e o desempilha

EXEMPLO 11.6. 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 no topo da pilha e o desempilha
q_2	b	b	(q_2, ε)	acha b na entrada e no topo da 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 para 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.

DEFINIÇÃO 11.7. *Seja M um autômato de pilha cujos elementos são dados pelo vetor $(\Sigma, \Gamma, Q, q_0, F, \Delta)$. 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*.

DEFINIÇÃO 11.8. *Suponhamos que $C = (q, \sigma w, \gamma u)$ é uma configuração de M, onde $\sigma \in \Sigma \cup \{\varepsilon\}$, $\gamma \in \Gamma \cup \{\varepsilon\}$. Dizemos que $C' = (q', w, vu)$ é uma das configurações* seguintes *a C se*

$$(q', v) \in \Delta(q, \sigma, \gamma).$$

Neste caso, escrevemos $C \vdash C'$.

DEFINIÇÃO 11.9. *Uma* computação *de M é uma sequência de configurações $C_0, \cdots, C_k$ tais que C_{i+1} é uma das configurações seguintes a C_i. Frequentemente abreviaremos a computação acima na forma $C_0 \vdash^* C_k$.*

Como no caso de autômatos finitos, convencionaremos que se C é uma configuração, então $C \vdash^* C$. Note, contudo, que $C \vdash^* C$ não é equivalente a uma transição com entrada vazia. Por exemplo, o autômato $\mathcal{M}_2$ satisfaz

$$(q_1, \varepsilon, \varepsilon) \vdash (q_2, \varepsilon, \varepsilon).$$

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.

EXEMPLO 11.10. Partindo da configuração $(q_1, abcba^2, a)$, temos a seguinte computação no autômato $\mathcal{M}_1$ da seção 2:

$$(q_1, abcba^2, a) \vdash (q_1, bcba^2, a^2) \vdash (q_1, cba^2, ba^2) \vdash (q_2, ba^2, ba^2) \vdash$$
$$(q_2, a^2, a^2) \vdash (q_2, a, a) \vdash (q_2, \varepsilon, \varepsilon).$$

Esta é, essencialmente, a única computação possível partindo de $(q_1, abcb\,a^2, a)$. A única coisa que podemos fazer para obter uma computação diferente é interromper a computação acima antes de chegar a $(q_2, \varepsilon, \varepsilon)$. Entretanto, isto está longe de ser sempre verdade, como mostra o exemplo seguinte.

EXEMPLO 11.11. 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

$$(q_1, a^2b, \varepsilon) \vdash (q_1, ab, a) \vdash (q_1, b, a^2) \vdash (q_2, b, a^2).$$

Mas há muitas outras possibilidades, como:

$$(q_1, a^2b, \varepsilon) \vdash (q_2, a^2b, \varepsilon)$$

ou ainda

$$(q_1, a^2b, \varepsilon) \vdash (q_1, ab, a) \vdash (q_2, ab, a).$$

A diferença é que o primeiro autômato era basicamente determinístico, ao passo que este último é claramente não-determinístico.

DEFINIÇÃO 11.12. *Supondo, como antes, que M é um autômato de pilha cujos elementos são dados pelo vetor $(\Sigma, \Gamma, Q, q_0, F, \Delta)$, diremos que uma palavra $w \in \Sigma^*$ é* aceita *por M se existe uma computação*

$$(q_1, w, \varepsilon) \vdash^* (p, \varepsilon, \varepsilon),$$

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 inspeção 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 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 11.13. 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 $\varepsilon \neq w \in \{a, b\}^*$ 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 segunda 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 anômala 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 para. A conclusão é que o autômato continua aceitando o que devia apesar da alteração na tabela de transição.

EXEMPLO 11.14. A linguagem deste segundo exemplo é uma generalização da que aparece no primeiro. Seja L_2 a linguagem formada pelas palavras $w \in \{a, b\}$ para as quais o número de as e bs é o mesmo. Assim, $abaaabbabb \in L_2$, mas $aaabaa \notin L_2$. Observe que a linguagem do exemplo anterior está contida em L_2.

À primeira vista, podemos construir um autômato de pilha que aceite L_2 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

$$\Delta(q_1, \varepsilon, \varepsilon) = \{(q_2, \beta)\}.$$

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_2 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_2, 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 $\{a, b, \beta\}$ como alfabeto 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 11.15. 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 adivinhar 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 alfabeto de entrada igual a $\{a, b\}$ e alfabeto da pilha igual a $\{a, b, \beta\}$; 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

EXEMPLO 11.16. Neste exemplo, construímos um autômato de pilha não-determinístico que aceita o *complemento* da linguagem

$$L = \{ww^R : w \in (0 \cup 1)^*\}.$$

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, ε)	tenta adivinhar o meio da entrada (palavras de comprimento par)
q_2	0	ε	(q_4, ε)	tenta adivinhar o símbolo central da entrada (palavras de comprimento ímpar)
q_2	1	ε	(q_4, ε)	tenta adivinhar o símbolo central da entrada (palavras de comprimento ímpar)
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

Precisamos considerar dois casos de palavras que não pertencem a L: todas as palavras de comprimento ímpar e as palavras de comprimento par que não apresentam o padrão ww^R.

No caso das palavras de comprimento ímpar, depois de marcar o fundo da pilha, o autômato empilha 0s e 1s até decidir, de maneira não-determinística, que o símbolo central da palavra de entrada foi encontrado. Então, muda de estado para q_4 e passa a desempilhar. Precisamos fazer isto para ter certeza de que o autômato adivinhou o símbolo central da palavra corretamente e que a palavra realmente tem comprimento ímpar. Se não sobrarem nem 0s, nem 1s na pilha, o autômato remove o marcador do fundo e esvazia a pilha.

No caso das palavras de comprimento par, 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 para q_3 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 q_4 e esvazia a pilha, comparando-a símbolo a símbolo com a entrada. Precisamos fazer isto para ter certeza de que o autômato adivinhou o meio da palavra corretamente e que a palavra não apresenta o padrão ww^R (falha na correspondência em q_3). 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 adivinhar incorretamente o meio da palavra ou o símbolo central, então sobrarão símbolos na entrada (se adivinhar cedo demais) ou na pilha (se adivinhar tarde demais). Em nenhum destes casos a palavra será aceita.

Finalmente, se a palavra da entrada estiver em L, isto é, apresentar o padrão ww^R, e se o autômato adivinhar 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, já que não encontra nenhuma falha de correspondência, o marcador do fundo da pilha não é removido, de modo que a pilha não será esvaziada e a palavra não será aceita.

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 sequências 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) $L = \{w \in \{a, b\}^* :$ o número de as é diferente do de bs$\}$;

d) $L = \{a^n b^m : m, n \geq 0 \text{ e } m \neq n\}$;

e) $L = \{w_1 c w_2 : w_1, w_2 \in \{a, b\}^* \text{ e } w_1 \neq w_2^r\}$;

(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, isto é, a linguagem das palavras formadas pelos símbolos (e) tais que, se w é uma palavra desta linguagem, então a quantidade total de símbolos (é igual à

quantidade total de símbolos) em w e, se w pode ser decomposta como $w = w'w''$, então a quantidade total de símbolos (é maior ou igual à quantidade total de símbolos) em w'.

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 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 $\Delta(i, \varepsilon, \varepsilon) = \{(q, \beta)\}$, 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) 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:

$$N(M) = \{w \in \Sigma^* : \text{ existe } (q_1, w, \varepsilon) \vdash^* (q, \varepsilon, \varepsilon) \text{ onde } q \in Q\}.$$

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 $(q, \varepsilon, \varepsilon)$ 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.

CAPíTULO 12

Relação entre Gramáticas Livres de Contexto e Autômatos de Pilha

Nosso objetivo neste capítulo é dar uma demonstração do seguinte resultado fundamental.

TEOREMA 12.1. *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 G precisamos de uma receita para construir um autômato de pilha não-determinístico que aceite $L(G)$. 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 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 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 G.

Digamos que a gramática livre de contexto 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 G. Como $\mathcal{M}$ deve aceitar $L(G)$, 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 ideia delineada acima consiste em escolher o alfabeto da pilha como sendo $T \cup V$, e fazer corresponder a cada derivação em um passo de G uma transição do autômato. Entretanto, como a derivação de palavras de $L(G)$ 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

$$\Delta(i, \varepsilon, \varepsilon) = \{(f, S)\},$$

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 G fazemos corresponder a transição

$$(f, \alpha) \in \Delta(f, \varepsilon, X) \tag{1.1}$$

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.

EXEMPLO 12.2. Suponhamos que G_1 é a gramática com terminais $\{a, b, c\}$, variável $\{S\}$, símbolo inicial S, e regras

$$S \to Sc \quad |aSb \quad | \quad \varepsilon.$$

De acordo com a discussão acima o autômato $\mathcal{M}_1$ correspondente a 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 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 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)

$$(i, abc^2, \varepsilon) \vdash (f, abc^2, S) \vdash (f, abc^2, Sc) \vdash (f, abc^2, Sc^2) \vdash (f, abc^2, aSbc^2).$$

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

$$(1.3) \qquad (f, abc^2, aSbc^2) \vdash (f, bc^2, Sbc^2) \vdash (f, bc^2, bc^2) \vdash (f, c^2, c^2) \vdash$$

$$\vdash (f, c, c) \vdash (f, \varepsilon, \varepsilon).$$

Observe que a passagem da segunda para a terceira configuração em (1.3) corresponde à regra $S \to \varepsilon$. Todas as outras etapas são aplicações das transições da segunda tabela.

2. A Receita e Mais um Exemplo

Podemos agora descrever de maneira sistemática a receita usada para construir um autômato de pilha não-determinístico $\mathcal{M}$ cuja linguagem aceita é $L(G)$. Se a gramática livre de contexto 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

$$\Delta : \{i, f\} \times (T \cup \{\varepsilon\}) \times (T \cup V \cup \{\varepsilon\}) \to \{i, f\} \times (T \cup V)^*$$

que é definida por

$$\Delta(q, \sigma, \gamma) = \begin{cases} (f, S) & \text{se } q = i \text{ e } \sigma = \gamma = \varepsilon \\ \{(f, u) : X \to u \in R\} & \text{se } q = f, \sigma = \varepsilon \text{ e } \gamma = X \in V \\ (f, \varepsilon) & \text{se } q = f \text{ e } \sigma = \gamma \in T \end{cases}$$

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 de 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 esquerdo.

Vejamos mais um exemplo de autômato de pilha construído a partir de uma gramática livre de contexto pela receita acima.

EXEMPLO 12.3. Considere a gramática G'_{exp}, definida anteriormente, 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.

TEOREMA 12.4. *Se 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 $L(G) = L(\mathcal{M})$.*

A demonstração deste resultado consiste em formalizar a relação entre derivações mais à esquerda em 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 G'_{exp} com uma computação no autômato que acabamos de descrever.

EXEMPLO 12.5. 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 G'_{exp}

$$(i, \mathrm{id} + \mathrm{id}, \varepsilon) \vdash (f, \mathrm{id} + \mathrm{id}, E),$$

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:

$$(f, \mathrm{id}, T) \vdash (f, \mathrm{id}, F) \vdash (f, \mathrm{id}, \mathrm{id}) \vdash (f, \varepsilon, \varepsilon).$$

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 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 $L(G) \subseteq L(\mathcal{M})$ e $L(G) \supseteq L(\mathcal{M})$.

Seja G é uma gramática livre de contexto formada pelos elementos (T, V, S, R) e seja w uma palavra gerada por G. Queremos mostrar que w é aceita por $\mathcal{M}$. Suponhamos que conhecemos uma derivação mais à esquerda $S \Rightarrow^* w$ em G. Devemos ser capazes de usar esta derivação para obter uma computação

$$(i, w, \varepsilon) \vdash^* (f, \varepsilon, \varepsilon).$$

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

$$S \Rightarrow \ldots \Rightarrow \alpha X v,$$

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

$$(i, w, \varepsilon) \vdash (f, w, S) \vdash^* (f, \beta, Xv),$$

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

$$S \Rightarrow \ldots \Rightarrow \alpha X v \Rightarrow \alpha u v.$$

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 $(f, u) \in \Delta(f, \varepsilon, X)$, que nos dá

$$(i, w, \varepsilon) \vdash (f, w, S) \vdash^* (f, \beta, Xv) \vdash (f, \beta, uv).$$

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 $\alpha uv = \alpha\gamma Y v'$, $w = \alpha\beta$ 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

$$(i, w, \varepsilon) \vdash^* (f, \beta, uv) = (f, \gamma\beta', \gamma Y v') \vdash^* (f, \beta', Y v'),$$

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

$$(i, w, \varepsilon) \vdash^* (f, \varepsilon, \varepsilon),$$

demonstrando, portanto, que $w \in L(\mathcal{M})$.

Provamos, assim, que $L(G) \subseteq L(\mathcal{M})$. 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 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 derivação 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 sequência de palavras de $(T \cup V)^*$ é de fato uma derivação de w a partir de S. Há três coisas a verificar:

(1) a sequência começa com S;

(2) para passar de uma palavra da sequência para a próxima trocamos sua variável mais à esquerda pelo lado direito de uma regra de G;

(3) a última palavra da sequência é w.

Mas (1) e (2) são consequências imediatas da maneira como $\mathcal{M}$ foi definido, e (3) segue do fato de que a computação acaba na configuração $(f, \varepsilon, \varepsilon)$. Provamos, portanto, que $w \in L(G)$.

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 G tal que $L(G) = L(\mathcal{M})$.

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 $\Delta(i, \varepsilon, \varepsilon) = \{(q, \beta)\}$, 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 $\Delta(q, \sigma, \beta) = (p, \varepsilon)$ 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 $(\Sigma, \Gamma, Q, q_0, F, \Delta)$. 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, onde estamos supondo que $\sigma \in \Sigma \cup \{\varepsilon\}$ e que $\gamma \in \Gamma \cup \{\varepsilon\}$:

estado	entrada	topo da pilha	transição
i	ε	ε	(q_0, β)
$q \neq i, f$	σ	$\gamma \neq \beta$	$\Delta(q, \sigma, \gamma)$
$q \neq i, f$	$\sigma \neq \varepsilon$	β	$\Delta(q, \sigma, \varepsilon)$
$q \neq i, f$ e $q \notin F$	ε	β	$\Delta(q, \varepsilon, \varepsilon)$
$q \neq i, f$ e $q \in F$	ε	β	$\Delta(q, \varepsilon, \varepsilon) \cup \{(f, \varepsilon)\}$

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 sucintamente 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_0 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

$$(q_0, w, \varepsilon) \vdash^* (p, \varepsilon, \varepsilon) \quad \text{em } \mathcal{M}.$$

Esta computação dará lugar a uma computação em $\mathcal{N}$ da forma

$$(i, w, \varepsilon) \vdash (q_0, w, \beta) \vdash^* (p, \varepsilon, \beta) \vdash (f, \varepsilon, \varepsilon).$$

Portanto, se w for aceita por $\mathcal{M}$ será aceita por $\mathcal{N}$.

Por outro lado, uma computação

$$(i, w, \varepsilon) \vdash^* (f, \varepsilon, \varepsilon), \tag{4.1}$$

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

$$(i, w, \varepsilon) \vdash (q_0, w, \beta) \vdash^* (p, \varepsilon, \beta) \vdash (f, \varepsilon, \varepsilon).$$

Temos assim que $(q_0, w, \varepsilon) \vdash^* (p, \varepsilon, \varepsilon)$ 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

$$\Delta(i, \varepsilon, \varepsilon) = \{(i', \beta)\},$$

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 $(r_1, \ldots, r_k) \in Q^k$ construímos 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: $u = \varepsilon$

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) $\quad (i, w, \varepsilon) \vdash (i', w, \beta) \vdash^* (f, \varepsilon, \varepsilon)$ 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

$$(q, \sigma v, \gamma u) \vdash (p, v, \gamma_1 \cdots \gamma_k u),$$

onde $v \in \Sigma^*$ e $u \in \Gamma^*$. Evidentemente para que esta etapa seja legítima é preciso que

(5.2) $\quad (p, \gamma_1 \cdots \gamma_k) \in \Delta(q, \sigma, \gamma).$

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, substituímos 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 12.6. *Sejam p e q estados do autômato de pilha não-determinístico $\mathcal{N}$. Então*

$$(q, w, X) \vdash^* (p, \varepsilon, \varepsilon) \text{ se, e somente se } (q, X, p) \Rightarrow^* w$$

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

$$\textbf{(A}(m)\textbf{)} \qquad \text{se } (q, w, X) \vdash^m (p, \varepsilon, \varepsilon), \text{ então } (q, X, p) \Rightarrow^* w.$$

por indução em m.

Se $m = 1$, então

$$w = \sigma \in \Sigma \cup \{\varepsilon\} \text{ e } (p, \varepsilon) \in \Delta(q, \sigma, X).$$

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

$$(q, w, X) \vdash^s (p, \varepsilon, \varepsilon) \tag{5.3}$$

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

$$(q^k, v_k, Y_k) \vdash^* (p, \varepsilon, \varepsilon).$$

Como cada uma destas computações tem menos de s passos, segue da hipótese de indução, que

$$(q^i, v_i, Y_i) \vdash^* (q^{i+1}, \varepsilon, \varepsilon),$$

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

(**B**(m)) se $(q, X, p) \Rightarrow^m w$ então $(q, w, X) \vdash^* (p, \varepsilon, \varepsilon)$

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 $\sigma \in \Sigma \cup \{\varepsilon\}$. Além disto, esta regra só pode ocorrer se houver uma transição da forma

$$(p, \varepsilon) \in \Delta(q, \sigma, X)$$

em $\mathcal{N}$. Mas esta transição dá lugar à computação

$$(q, \sigma, X) \vdash (p, \varepsilon, \varepsilon)$$

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

(5.7) $$(q, X, p) \Rightarrow^s w$$

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

$$(s_i, v_i, Y_i) \vdash^* (s_{i+1}, \varepsilon).$$

para $i = 1, \ldots, k$. Além disso, se pusermos $Y_{i+1}, \ldots, Y_k$ abaixo de Y_i, no fundo da pilha, obteremos

$$(s_i, v_i, Y_i Y_{i+1} \cdots Y_k) \vdash^* (s_{i+1}, \varepsilon).$$

Agora, (5.8) provém da transição

$$(p, Y_1 \cdots Y_k) \in \Delta(q, \sigma, X).$$

Lembrando que $w = \sigma v_1 \cdots v_k$, vemos que $\mathcal{N}$ admite uma computação da forma

$$(q, w, X) \vdash (s_1, v_1 \cdots v_k, Y_1 \cdots Y_k) \vdash^* (s_2, v_2 \cdots v_k, Y_2 \cdots Y_k) \vdash^* \cdots \vdash^* (s_k, v_k, Y_k) \vdash (p, \varepsilon, \varepsilon),$$

como queríamos mostrar.

Podemos então resumir estes resultados de equivalência entre autômatos de pilha não-determinísticos e gramáticas livres de contexto no teorema abaixo.

TEOREMA 12.7. *As seguintes afirmações são equivalentes entre si:*

(1) L *é uma linguagem livre de contexto.*

(2) L *pode ser gerada por uma gramática livre de contexto.*

(3) L *é aceita por algum autômato de pilha não-determinístico.*

OBSERVAÇÃO. Ao contrário do que ocorria com os autômatos finitos, no caso dos autômatos de pilha não é mais verdade que os autômatos determinísticos e não-determinísticos aceitam a mesma classe de linguagens. Existem linguagens livres de contexto que não são aceitas por nenhum autômato de pilha determinístico. Isto significa que os resultados de equivalência que mostramos neste capítulo não se mantém quando consideramos apenas autômatos de pilha determinísticos. É por essa razão que, no caso dos autômatos de pilha, nos restringimos apenas ao estudo dos autômatos não-determinísticos.

6. De Volta às Propriedades de Fechamento

Vimos anteriormente que a interseção e a diferença de duas linguagens livres de contexto não são necessariamente linguagens livres de contexto. Entretanto, mostraremos agora que, se L é uma linguagem livre de contexto e R é uma *linguagem regular*, então $L \cap R$ e $L - R$ são sempre linguagens livres de contexto.

Seja $A_1 = (\Sigma, \Gamma, Q, q_0, F, \Delta)$ um autômato de pilha não-determinístico que aceita a linguagem L e seja $A_2 = (\Sigma, Q', q_0', F', \delta')$ um AFD que aceita a linguagem R. Queremos construir um autômato de pilha não-determinístico $A_\cap = (\Sigma, \Gamma, Q_\cap, q_{0_\cap}, F_\cap, \Delta_\cap)$ que aceite $L \cap R$. A ideia que utilizaremos na construção do autômato é a mesma que utilizamos anteriormente para construir um autômato que aceitava a interseção de duas linguagens regulares: o nosso autômato deverá simular em paralelo as computações dos dois autômatos originais com uma mesma palavra, realizando as transições nos dois de maneira sincronizada. O novo autômato aceitará a palavra se os dois autômatos originais aceitarem.

Repare que o autômato $A_\cap$ que vamos construir possui apenas uma pilha (como qualquer autômato de pilha). Esta pilha poderá ser usada exclusivamente para a simulação do autômato A_1, já que o autômato A_2 não utiliza uma pilha. É por

esta razão que esta construção funciona no caso em que uma linguagem é livre de contexto e a outra é regular mas não funcionaria para provar que a interseção de duas linguagens livres de contexto é livre de contexto: teríamos que simular as duas pilhas dos autômatos originais em apenas uma pilha do autômato que estamos construindo, o que não é possível.

Seguindo esta ideia, temos então:

- $Q_\cap = Q \times Q'$;
- $q_{0\cap} = (q_0, q'_0)$;
- $F_\cap = F \times F'$ (A_1 e A_2 terminam em estado final);
- $((p, p'), u) \in \Delta_\cap((q, q'), \sigma, \gamma)$ se $(p, u) \in \Delta(q, \sigma, \gamma)$ e $p' = \delta'(q', \sigma)$ (execução sincronizada das transições).

Para a diferença $L - R$ das linguagens, conseguimos mostrar que ela é também uma linguagem livre de contexto utilizando a igualdade $L - R = L \cap \overline{R}$. Já vimos anteriormente que, se R é uma linguagem regular, $\overline{R}$ também é necessariamente regular. Além disso, acabamos de mostrar acima que a interseção de uma linguagem livre de contexto com uma linguagem regular é uma linguagem livre de contexto. Logo, $L \cap \overline{R}$ é uma linguagem livre de contexto, o que significa que $L - R$ é uma linguagem livre de contexto.

7. 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 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) 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) $L = \{a^i b^j c^j d^i e^3 : i, j \geq 0\}$.

(4) Seja G uma gramática livre de contexto cujo símbolo inicial é S, e seja $\mathcal{M}$ o autômato de pilha construído a partir de G pela receita da seção 2. Suponhamos que w é uma palavra de comprimento k em $L(G)$. Determine o número de passos da computação de $\mathcal{M}$ que corresponde à derivação mais à esquerda $S \Rightarrow^n w$.

Parte 3

Computabilidade e Complexidade

CAPíTULO 13

Um Modelo Formal de Computação: A Máquina de Turing

Nos capítulos anteriores, estudamos dois modelos formais de computação que apresentam limitações significativas: os autômatos finitos e os autômatos de pilha. Neste capítulo, vamos iniciar o estudo de um terceiro modelo, conhecido como *Máquina de Turing*, que possui maior poder computacional do que os dois estudados anteriormente. De fato, como veremos mais adiante, este modelo possui o mesmo poder computacional dos computadores reais que utilizamos.

1. A Máquina de Turing

Uma *Máquina de Turing* fornece um modelo matemático (um modelo teórico) simples, porém preciso, de computabilidade. Ela é útil para estudar os limites do que pode ser resolvido algoritmicamente, para mostrar que existem (muitos) problemas sem solução algorítmica e para estudar os requisitos de tempo e espaço (memória) necessários para resolver algoritmicamente um dado problema.

A partir do modelo da Máquina de Turing, desenvolvemos a *Teoria da Computabilidade* e a *Teoria da Complexidade*. A Teoria da Computabilidade estuda a distinção entre problemas *solúveis* algoritmicamente (também conhecidos como *decidíveis*) e problemas *insolúveis* (*indecidíveis*). Já a *Teoria da Complexidade* se preocupa com a eficiência concreta de um dado algoritmo para resolver um problema, distinguindo os problemas entre *tratáveis* (algoritmos eficientes para a solução do problema são conhecidos) e *intratáveis*.

A Máquina de Turing possui este nome porque foi inicialmente proposta pelo matemático inglês *Alan Turing*, em um artigo de 1936 (quando ele tinha 24 anos de idade).

Alan Turing nasceu em 1912 e faleceu em 1954. Além do desenvolvimento da Máquina de Turing, ele também se destacou no trabalho como criptoanalista para os britânicos na Segunda Guerra Mundial, tendo papel central na quebra da *Máquina Enigma*, que implementava a criptografia utilizada pelos alemães.

No mesmo artigo em que Turing descreveu a sua máquina, ele também mostrou a existência de um problema indecidível, isto é, de um problema que não admite solução algorítmica. Este problema exibido por Turing passou a ser conhecido como *Problema da Parada*.

Começamos a apresentação do modelo da Máquina de Turing com uma descrição informal (Figura 1).

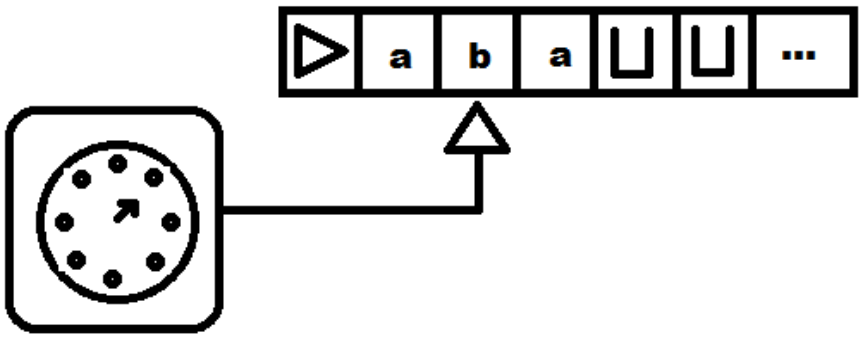


FIGURA 1. Ilustração de uma Máquina de Turing

A máquina possui os seguintes componentes:

(1) Uma *fita*, que é *infinita à direita*, dividida em *casas*.
   - Cada casa pode conter um *símbolo* ou estar em *branco* (representado por $\sqcup$).
   - A casa na extremidade esquerda da fita contém o símbolo $\triangleright$. A máquina não pode escrever outro símbolo nesta casa durante a computação e este símbolo ($\triangleright$) não pode ser escrito em nenhuma outra casa da fita. Assim, o símbolo $\triangleright$ sempre funciona como um marcador da extremidade esquerda da fita.
   - Apesar da fita ser infinita à direita, temos que, em qualquer momento da computação, apenas uma quantidade *finita* de casas *não está em*

branco. Dito de outra maneira, sempre existe uma casa na fita a partir da qual *todas as casas mais à direita* estão em branco.

(2) Um *cabeçote*, que se encontra sempre sobre uma das casas da fita. O cabeçote é capaz de realizar tanto operações de *leitura* quanto de *escrita* na fita e é capaz de se mover tanto para a *esquerda* quanto para a *direita*. A única restrição de movimentação do cabeçote diz respeito à extremidade esquerda da fita. Se o cabeçote está sobre esta casa, ele não pode se movimentar para a esquerda.

(3) Um conjunto *finito* de estados.

A *transição* de uma Máquina de Turing se dá da seguinte forma: dado um estado atual da máquina e o símbolo que o cabeçote está lendo na fita (o símbolo que se encontra na casa da fita onde está o cabeçote), a máquina muda de estado e realiza uma ação na fita. A cada transição, a máquina pode realizar uma das seguintes três ações na fita:

(1) Mover o cabeçote uma casa para a esquerda;

(2) Mover o cabeçote uma casa para a direita;

(3) Manter o cabeçote parado e escrever um novo símbolo na casa atual da fita.

Vamos prosseguir agora para a descrição *formal* de uma Máquina de Turing.

DEFINIÇÃO 13.1 (Máquina de Turing). *Uma máquina de Turing (determinística) é uma* 6*-upla* $M = (\Sigma_0, \Sigma, Q, q_0, F, \delta)$*, onde:*

- Σ_0 *é o* alfabeto da entrada*;*
- Σ *é o* alfabeto da fita*;*
- Q *é o conjunto* finito *de* estados*;*
- $q_0 \in Q$ *é o* estado inicial*;*
- $F \subseteq Q$ *é o conjunto de* estados finais *ou* estados de parada *e*
- $\delta : (Q - F) \times \Sigma \to Q \times ((\Sigma - \{\triangleright\}) \cup \{\leftarrow, \rightarrow\})$ *é a função de* transição.

Vamos analisar cuidadosamente a entrada e a saída da função de transição. A função δ possui duas entradas: um estado que não é final (um elemento de $Q - F$)

e um símbolo do alfabeto da fita (um elemento de Σ), que representa o símbolo que o cabeçote está lendo na fita. Por outro lado, ela possui duas saídas: um estado (um elemento de Q) e a ação que é realizada na fita. A ação de mover o cabeçote uma casa para a esquerda é representada pelo símbolo $\leftarrow$ e a ação de mover o cabeçote uma casa para a direita é representada pelo símbolo $\rightarrow$. Por fim, a ação de escrever um novo símbolo na posição atual da fita é representada pelo símbolo de $(\Sigma - \{\triangleright\})$ que será escrito. O símbolo a ser escrito é um elemento de $(\Sigma - \{\triangleright\})$ e não de Σ porque o elemento $\triangleright$ deve servir sempre de marcador da extremidade esquerda da fita, de forma que ele não pode ser escrito em nenhuma outra posição.

Podemos reparar que $\delta(q, \sigma)$ só está definida se $q \notin F$. Logo, a máquina sempre para quando alcança um estado final. Por outro lado, como a transição $\delta(q, \sigma)$ sempre está definida para todo $q \notin F$, a máquina somente para se alcançar um dos estados finais. Como a máquina para se e somente se alcança um dos estados finais, estes estados também são conhecidos como *estados de parada*. É importante observar então que uma computação em uma Máquina de Turing pode nunca parar, caso nenhum estado final seja alcançado durante a computação.

É importante observarmos também que o comportamento esperado para a máquina quando o cabeçote está sobre a casa na extremidade esquerda da fita impõe restrições à saída da função δ quando o símbolo lido é $\triangleright$. Se estamos na extremidade esquerda da fita, não podemos mover o cabeçote para a esquerda. Por outro lado, também não podemos escrever um novo símbolo nesta casa, senão perderíamos a marcação fornecida pelo símbolo $\triangleright$. Assim, após a exclusão destas duas ações, a única ação que resta é mover o cabeçote para a direita. Portanto, para todo $q \in Q - F$, temos

$$\delta(q, \triangleright) = (q', \rightarrow),$$

para algum $q' \in Q$.

O alfabeto da entrada (Σ_0) contém os símbolos que podem ser utilizados para formar a palavra que será fornecida como entrada para a máquina, de forma análoga ao que temos também nos casos dos autômatos finitos e dos autômatos de pilha. Já o alfabeto da fita (Σ) contém os símbolos que podem ser colocados nas casas da fita

da máquina, lembrando que cada casa da fita não pode conter mais de um símbolo. Ao contrário do caso dos autômatos de pilha, em que o alfabeto da entrada e o alfabeto da pilha são independentes entre si e sem restrições, no caso das Máquinas de Turing existem algumas restrições para os alfabetos Σ_0 e Σ:

(1) $\Sigma_0 \cap \{\triangleright\} = \emptyset$;
(2) $\Sigma_0 \cup \{\triangleright, \sqcup\} \subseteq \Sigma$ e
(3) $\Sigma \cap \{\leftarrow, \rightarrow\} = \emptyset$.

A primeira restrição diz que o símbolo $\triangleright$ não faz parte do alfabeto da entrada. Esta restrição é condizente com a condição que impomos anteriormente de que este símbolo não pode ser escrito em nenhuma outra casa da fita que não seja a casa da extremidade esquerda.

A segunda restrição diz que o alfabeto da fita deve necessariamente conter todos os símbolos do alfabeto de entrada e também o símbolo de marcação da extremidade esquerda da fita ($\triangleright$) e o símbolo de marcação de uma casa em branco na fita ($\sqcup$). Além destes símbolos, o alfabeto da fita poderá conter ainda outros símbolos adicionais que poderão ser escritos na fita ao longo da computação da máquina.

Finalmente, a terceira restrição diz que os símbolos $\leftarrow$ e $\rightarrow$ não podem pertencer ao alfabeto da fita. O objetivo desta restrição é evitar ambiguidades na descrição da transição da máquina, já que estes dois símbolos são utilizados justamente para denotar as ações na fita que não envolvem a escrita de símbolos na fita: a movimentação do cabeçote uma casa para a esquerda e uma casa para a direita, respectivamente.

Uma Máquina de Turing inicia uma computação no seu estado inicial q_0, com o símbolo $\triangleright$ escrito na primeira casa da fita e as demais casas em branco (isto é, ocupadas pelo símbolo $\sqcup$) ou ocupadas por símbolos do alfabeto da entrada Σ_0. É importante lembrarmos mais uma vez que apenas um número *finito* de casas da fita pode ser preenchido com símbolos de Σ_0.

No momento de iniciar a computação, precisamos especificar em qual casa da fita o cabeçote começa. A computação que será realizada pela máquina é dependente da posição inicial escolhida para o cabeçote. Adotamos como convenção que, se nenhuma posição for especificada, o cabeçote começa na primeira casa à direita do símbolo $\triangleright$.

Após o início da computação, a Máquina de Turing irá então realizar uma sequência de transições dadas pela função δ. A computação irá prosseguir até que a máquina alcance um estado final (estado de parada). Se a máquina nunca alcançar um estado de parada ao longo da computação, esta computação nunca irá se encerrar (a máquina nunca parará).

Para descrever a computação da máquina, precisamos de uma maneira compacta de descrever as configurações da fita. Como exemplo, suponha que a fita, em algum momento da computação, apresente a configuração abaixo.

$\triangleright$	σ_1	$\ldots$	σ_k	$\sqcup$	$\sqcup$	$\ldots$
	$\Uparrow$					

Nesta configuração, o cabeçote se encontra sobre a segunda casa da fita, que contém o símbolo σ_1. Além disso, após a casa com o símbolo σ_k, o restante da fita se encontra em branco. Podemos representar de forma compacta esta configuração da fita com a seguinte palavra w no alfabeto da fita ($w \in \Sigma^*$):

$$w = \triangleright \underline{\sigma_1} \sigma_2 \ldots \sigma_k.$$

Nesta notação, o símbolo da palavra que se encontra sublinhado indica a posição atual do cabeçote. Além disso, nesta notação, podemos sempre assumir que todas as casas após a casa que contém o último símbolo de w estão em branco.

Podemos então descrever uma computação em uma Máquina de Turing como uma sequência de pares (q, w), onde $q \in Q$ é um estado da máquina e $w \in \Sigma^*$ é uma descrição da fita conforme a convenção acima. O estado que aparece no primeiro par da sequência é sempre o estado inicial q_0. Caso apareça um estado

final da máquina em algum par da sequência, ele deve aparecer no último par, uma vez que a computação sempre termina quando a máquina alcança um estado final.

Podemos ter então uma computação em um dos dois padrões abaixo:

$$(q_0, w_0) \vdash (q_1, w_1) \vdash \ldots \vdash (q_t, w_t),$$

onde $q_i \in (Q - F)$, para todo $0 \leq i < t$, e $q_t \in F$, ou

$$(q_0, w_0) \vdash (q_1, w_1) \vdash \ldots \vdash (q_t, w_t) \vdash \ldots,$$

onde $q_i \in (Q - F)$, para todo $i \in \mathbb{N}$.

EXEMPLO 13.2. Seja $M_1 = (\Sigma_0, \Sigma, Q, q_0, F, \delta)$, onde:

- $\Sigma_0 = \{a\}$
- $\Sigma = \{a, \triangleright, \sqcup\}$
- $Q = \{q_0, q_1, h\}$
- $F = \{h\}$
- δ :

δ	a	$\triangleright$	$\sqcup$
q_0	$(q_1, \sqcup)$	$(q_0, \rightarrow)$	$(h, \sqcup)$
q_1	(q_0, a)	$(q_1, \rightarrow)$	$(q_0, \rightarrow)$

Vamos analisar a computação desta máquina se colocarmos nas primeiras casas da fita a palavra $w = \triangleright aa \sqcup aaa$. Esta computação vai depender da posição inicial do cabeçote.

No primeiro caso, vamos supor que o cabeçote começa sobre a casa com o símbolo $\triangleright$. Teremos então a seguinte computação:

$$(q_0, \underline{\triangleright} aa \sqcup aaa) \vdash (q_0, \triangleright \underline{a} a \sqcup aaa) \vdash (q_1, \triangleright \underline{\sqcup} a \sqcup aaa) \vdash (q_0, \triangleright \sqcup \underline{a} \sqcup aaa) \vdash$$

$$\vdash (q_1, \triangleright \sqcup \underline{\sqcup} \sqcup aaa) \vdash (q_0, \triangleright \sqcup \sqcup \underline{\sqcup} aaa) \vdash (h, \triangleright \sqcup \sqcup \underline{\sqcup} aaa).$$

Caso o cabeçote comece na segunda casa da fita, o resultado final da computação será o mesmo, bastando remover o primeiro par da sequência acima.

No caso em que o cabeçote começa na terceira casa, temos a seguinte computação:

$$(q_0, \triangleright a\underline{a} \sqcup aaa) \vdash (q_1, \triangleright a\underline{\sqcup} \sqcup aaa) \vdash (q_0, \triangleright a \sqcup \underline{\sqcup} aaa) \vdash (h, \triangleright a \sqcup \underline{\sqcup} aaa).$$

Finalmente, se o cabeçote começa na quarta casa, temos a seguinte computação:

$$(q_0, \triangleright aa\underline{\sqcup} aaa) \vdash (h, \triangleright aa\underline{\sqcup} aaa).$$

Olhando para as operações que a máquina realizou na fita nos casos acima, podemos concluir o que esta máquina faz: ela "apaga" (substitui pelo símbolo $\sqcup$ de casa em branco) todos os símbolos "a" a partir da posição inicial do cabeçote, prosseguindo para a direita até achar a primeira casa em branco. Ao encontrar esta casa, a máquina para. Como a fita é infinita para a direita e apenas uma quantidade finita de casas pode não estar em branco, esta máquina sempre irá parar eventualmente.

EXEMPLO 13.3. Seja $M_2 = (\Sigma_0, \Sigma, Q, q_0, F, \delta)$, onde:

- $\Sigma_0 = \{a\}$
- $\Sigma = \{a, \triangleright, \sqcup\}$
- $Q = \{q_0, h\}$
- $F = \{h\}$
- δ :

δ	a	$\triangleright$	$\sqcup$
q_0	$(q_0, \leftarrow)$	$(q_0, \rightarrow)$	$(h, \sqcup)$

Vamos analisar a computação desta máquina se colocarmos nas primeiras casas da fita a mesma palavra do exemplo anterior: $w = \triangleright aa \sqcup aaa$. Novamente, a computação vai depender da posição inicial do cabeçote.

No primeiro caso, vamos supor que o cabeçote começa sobre a casa com o símbolo $\triangleright$. Teremos então a seguinte computação:

$$(q_0, \underline{\triangleright} aa \sqcup aaa) \vdash (q_0, \triangleright \underline{a} a \sqcup aaa) \vdash (q_0, \underline{\triangleright} aa \sqcup aaa) \vdash \ldots$$

O terceiro par da computação é igual ao primeiro. Como a máquina é determinística, a computação irá repetir para sempre os dois primeiros pares. Desta forma, neste caso, a máquina não para.

Caso o cabeçote comece na segunda casa da fita, o resultado final da computação será o mesmo, sendo que esta computação irá se iniciar pelo segundo par da computação acima. O caso em que o cabeçote começa na terceira casa também é completamente análogo aos dois anteriores.

No caso em que o cabeçote começa na quarta casa, temos a seguinte computação:

$$(q_0, \triangleright aa\underline{\sqcup}aaa) \vdash (h, \triangleright aa\underline{\sqcup}aaa).$$

Nos três primeiros casos, a máquina entra em um laço infinito. No quarto caso, ela encerra a computação imediatamente sem realizar qualquer operação na fita. Parece então que esta máquina não realiza nenhuma tarefa útil. Para observarmos o que a máquina faz, precisamos colocar o cabeçote inicialmente em alguma posição à direita da primeira casa em branco. Vamos colocar o cabeçote sobre a sexta casa da fita. Temos então a seguinte computação:

$$(q_0, \triangleright aa \sqcup a\underline{a}a) \vdash (q_0, \triangleright aa \sqcup \underline{a}aa) \vdash (q_0, \triangleright aa\underline{\sqcup}aaa) \vdash (h, \triangleright aa\underline{\sqcup}aaa).$$

Esta máquina "rebobina" a fita para a esquerda até achar a primeira casa em branco e então para. É por isso que ela parou imediatamente quando o cabeçote já começava sobre uma casa em branco. Também é por isso que ela não parava nos primeiros casos: nem a casa onde o cabeçote estava no início da computação nem nenhuma casa à esquerda da posição inicial do cabeçote era uma casa em branco naqueles casos.

Nós podemos utilizar as Máquinas de Turing como dispositivos de reconhecimento de linguagens, de forma análoga ao uso que fazíamos dos autômatos finitos e dos autômatos de pilha.

Dado um alfabeto Σ_0, uma linguagem $L \subseteq \Sigma_0^*$ e uma palavra $w \in \Sigma_0^*$, gostaríamos de construir uma Máquina de Turing com alfabeto da entrada Σ_0 que

pudesse determinar se $w \in L$ ou se $w \notin L$. Relembrando, sabemos que se L é uma linguagem regular, esta determinação pode ser feita por um autômato finito e que se L é uma linguagem livre de contexto, esta determinação pode ser feita por um autômato de pilha.

Para utilizar uma Máquina de Turing para determinar se $w \in L$ ou $w \notin L$, escrevemos a palavra $w = \sigma_1\sigma_2\ldots\sigma_k$ nas primeiras casas da fita da máquina, logo após o símbolo $\triangleright$ e posicionamos o cabeçote sobre o primeiro símbolo de w (σ_1). Desta forma, a configuração inicial da fita da máquina pode ser descrita por $\triangleright\underline{\sigma_1}\sigma_2\ldots\sigma_k$.

DEFINIÇÃO 13.4 (Decisor). *Um* decisor *é uma Máquina de Turing* $M = (\Sigma_0, \Sigma, Q, q_0, F, \delta)$ *que satisfaz duas propriedades:*

(1) M *possui dois estados finais (dois estados de parada): um de* aceitação *e um de* rejeição. *O estado de aceitação é denotado por* s *(de "Sim") e o de rejeição por* n *(de "Não"). Desta forma, em um decisor temos* $F = \{s, n\}$.

(2) M *sempre para (alcança um estado final) com qualquer palavra* $w \in \Sigma_0^*$ *colocada como entrada no início da fita da maneira descrita no parágrafo anterior.*

Dizemos que um decisor M com alfabeto da entrada Σ_0 *decide* uma linguagem $L \subseteq \Sigma_0^*$ (ou dizemos que L é decidida por M) se, para todo $w \in \Sigma_0^*$, temos que

$$w \in L \quad \Leftrightarrow \quad M \text{ para no estado } s \text{ com entrada } w.$$

Repare que, como M é um decisor, M possui apenas dois estados de parada (s e n) e M sempre para em um deles para qualquer palavra $w \in \Sigma_0^*$ fornecida como entrada. Desta forma, caso M não pare em s, irá parar em n. Portanto, a condição acima também implica que

$$w \notin L \quad \Leftrightarrow \quad M \text{ para no estado } n \text{ com entrada } w.$$

DEFINIÇÃO 13.5. *Dizemos que uma linguagem L é uma* linguagem recursiva *ou uma* linguagem decidível *se existe algum decisor M que* decide *L.*

Definimos assim a terceira classe de linguagens com que vamos trabalhar. É simples mostrar que toda linguagem regular e toda linguagem livre de contexto são também linguagens recursivas. Para isso, basta mostrarmos como simular um autômato finito e um autômato de pilha com a utilização de uma Máquina de Turing que seja um decisor.

Utilizar um decisor para simular um autômato finito é bastante simples. Podemos nos ater ao caso dos autômatos finitos determinísticos, uma vez que já sabemos que todo autômato finito não-determinístico possui um autômato finito determinístico equivalente.

Seja $A = (\Sigma_f, Q_f, q_{0f}, F_f, \delta_f)$ um autômato finito determinístico. Queremos simulá-lo em um decisor. Para isso, construímos o decisor $M = (\Sigma_0, \Sigma, Q, q_0, F, \delta)$ da seguinte forma:

- $\Sigma_0 = \Sigma_f$;
- $\Sigma = \Sigma_0 \cup \{\triangleright, \sqcup, X\}$, onde $X \notin \Sigma_f$;
- $Q = Q_f \cup \{s, n\}$, onde $s, n \notin Q_f$;
- $q_0 = q_{0f}$;
- $F = \{s, n\}$;
- $\delta(q, \sigma) = (\delta_f(q, \sigma), X)$, se $\sigma \notin \{\triangleright, \sqcup, X\}$;
- $\delta(q, X) = \delta(q, \triangleright) = (q, \rightarrow)$;
- $\delta(q, \sqcup) = (s, \sqcup)$, se $q \in F_f$, e
- $\delta(q, \sqcup) = (n, \sqcup)$, se $q \in Q_f - F_f$.

Nesta máquina, colocamos a palavra $w \in \Sigma_f^*$ no início da fita, logo após o símbolo $\triangleright$ e colocamos o cabeçote da máquina no primeiro símbolo de w. O símbolo extra X que pertence ao alfabeto da fita da máquina é usado para simbolizar que o símbolo daquela casa já foi consumido durante a simulação. A simulação então realiza as transições de acordo com a orientação da função δ_f do autômato finito, marcando os símbolos já consumidos com X e avançando o cabeçote para a direita. Quando o cabeçote alcança a primeira casa em branco após a palavra w, a máquina

vai para o estado final de aceitação se a simulação terminou em um estado final do autômato finito (um estado de F_f) e para o estado final de rejeição caso contrário. Ao chegar em qualquer destes estados, a máquina para.

Também é possível simular um autômato de pilha utilizando uma Máquina de Turing (determinística) que é um decisor. Entretanto, como os autômatos de pilha são não-determinísticos, é mais simples construir esta simulação após o estudo das Máquinas de Turing não-determinísticas (Exercício 21).

Com isso, podemos concluir que a classe das linguagens regulares e a classe das linguagens livres de contexto estão contidas na classe das linguagens recursivas. Esta inclusão é própria, isto é, existe ao menos uma linguagem que é recursiva, mas não é livre de contexto. Um exemplo de tal linguagem é

$$L = \{a^n b^n c^n : n \geq 0\},$$

que já mostramos, utilizando o Lema do Bombeamento, que não é livre de contexto. Ao final do capítulo, estaremos aptos a construir uma Máquina de Turing que *decide* esta linguagem (Exercício 15).

Além do conceito de *decisão*, podemos também definir uma outra maneira mais fraca de como utilizar uma Máquina de Turing para determinar se $w \in L$ ou $w \notin L$. Para isto, vamos considerar uma máquina M genérica, isto é, M não é um decisor.

Dizemos que uma máquina M com alfabeto da entrada Σ_0 *aceita* uma linguagem $L \subseteq \Sigma_0^*$ (ou dizemos que L é aceita por M) se, para todo $w \in \Sigma_0^*$, temos que

$$w \in L \quad \Leftrightarrow \quad M \text{ para (alcança um estado final) com entrada } w.$$

Esta definição também implica que

$$w \notin L \quad \Leftrightarrow \quad M \text{ não para (nunca alcança um estado final) com entrada } w.$$

DEFINIÇÃO 13.6. *Dizemos que uma linguagem L é uma* linguagem recursivamente enumerável *ou uma* linguagem semi-decidível *se existe alguma Máquina de Turing M que* aceita *L.*

Definimos assim a quarta e última classe de linguagens com que vamos trabalhar. Toda linguagem recursiva é também recursivamente enumerável, pois sempre podemos transformar um decisor M que decide uma linguagem L em uma Máquina de Turing M' que aceita L. Para isto, substituímos as transições de M que levam o decisor para o estado final n por transições que levam a máquina de volta ao estado atual, sem realizar alterações na fita. Desta forma, se $w \in L$, então M para no estado s e M' também para no estado s. Já se $w \notin L$, então M para no estado n e M' não para nunca (nunca alcança um estado final). Desta forma, enquanto M decide L, M' aceita L.

Mais adiante, vamos mostrar que existem linguagens recursivamente enumeráveis que não são recursivas. Obtemos então uma hierarquia entre as classes de linguagens, onde todas as inclusões são próprias. Esta hierarquia, conhecida como *Hierarquia de Chomsky*, pode ser vista de forma esquemática no diagrama abaixo (Figura 2). LR denota a classe das linguagens regulares, LLC a classe das linguagens livres de contexto, Rec a classe das linguagens recursivas e RE a classe das linguagens recursivamente enumeráveis.

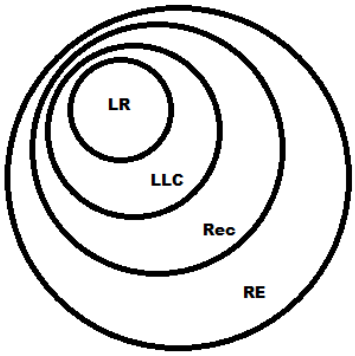


FIGURA 2. Hierarquia de Chomsky

Podemos utilizar também Máquinas de Turing para resolver problemas que não envolvem determinar se $w \in L$ ou $w \notin L$, isto é, não estão no formato sim/não.

Seja f uma função $f : \Sigma_0^* \to \Sigma_0^*$. Dizemos que uma Máquina de Turing M *computa* a função f se, para todo $w \in \Sigma_0^*$, quando a máquina M recebe como entrada w no início da fita, logo após o símbolo $\triangleright$, e inicia com o cabeçote sobre o primeiro símbolo de w, ela termina a computação parando com $f(w)$ no início da fita (também logo após o símbolo $\triangleright$).

Dizemos que $f : \Sigma_0^* \to \Sigma_0^*$ é uma *função computável* se existe uma Máquina de Turing M com alfabeto da entrada Σ_0 que *computa* f.

Encerramos esta seção com dois exemplos "de carne e osso" de Máquinas de Turing:

(1) Máquina de Turing de carretel e caneta:

`http://aturingmachine.com/examples.php`

(2) Máquina de Turing de Lego:

`http://legoofdoom.blogspot.com`

2. Diagramas de Composição

Máquinas de Turing que executam tarefas que não são tão simples quanto as exemplificadas na seção anterior podem apresentar tabelas de transição bastante longas e complexas. Desta forma, construir uma Máquina de Turing descrevendo a sua tabela de transição pode ser uma tarefa muito difícil.

Para termos uma maneira mais simples e rápida de construir Máquinas de Turing que executem as tarefas que queremos, apresentamos os *Diagramas de Composição* para Máquinas de Turing. Um diagrama de composição descreve de forma gráfica o funcionamento de uma Máquina de Turing. Um diagrama é construído a partir de dois tipos de componentes:

(1) Máquinas básicas

(2) Regras de composição

As *máquinas básicas* são as seguintes:

- **D** = Máquina que move o cabeçote uma casa para a direita e para.
- **E** = Máquina que move o cabeçote uma casa para a esquerda (se possível) e para.

- $\mathbf{W}_a$ = Máquina que escreve o símbolo a na posição atual da fita e para.
- **P** = Máquina que para sem realizar nenhuma ação na fita.

Para construir *decisores*, existem ainda duas outras máquinas básicas úteis:

- **S** = Máquina que para no estado final de aceitação
- **N** = Máquina que para no estado final de rejeição

As regras de composição são descritas utilizando-se arestas entre máquinas básicas, como exemplificado no diagrama abaixo (Figura 3).

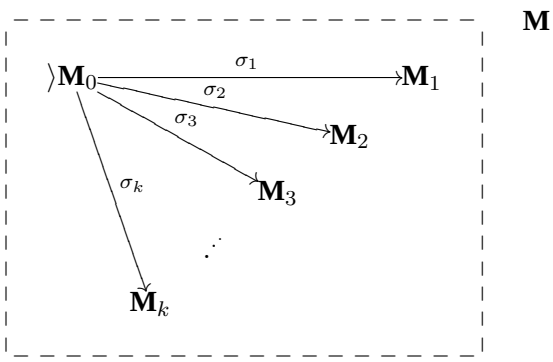


FIGURA 3. Esquema Básico de um Diagrama de Composição

Neste diagrama, o símbolo $\rangle$ indica que a máquina **M** inicia a sua computação de acordo com a computação de $\mathbf{M}_0$. Enquanto $\mathbf{M}_0$ não parar, **M** continua sua computação de forma idêntica a $\mathbf{M}_0$. Quando $\mathbf{M}_0$ para, a máquina **M** verifica que símbolo está enxergando na fita. Se o símbolo for σ_1, ela continua sua computação de acordo com a computação de $\mathbf{M}_1$, se o símbolo for σ_2, ela continua sua computação de acordo com a computação de $\mathbf{M}_2$, e assim por diante. Caso o símbolo que **M** está enxergando na fita não seja nenhum dos símbolos $\sigma_1, \sigma_2, \ldots, \sigma_k$ que rotulam as arestas que saem de $\mathbf{M}_0$ no diagrama, então **M** para quando $\mathbf{M}_0$ para.

Vemos então que um diagrama de composição nos permite descrever o comportamento de uma Máquina de Turing "por partes", compondo máquinas mais simples de forma a obter máquinas mais complexas.

Algumas convenções de notação são muito úteis na elaboração de diagramas de composição:

(1) Um diagrama no formato

$$\mathbf{M}_1 \overset{a}{\underset{b}{\rightrightarrows}} \mathbf{M}_2$$

pode ser escrito de forma mais compacta como

$$\mathbf{M}_1 \xrightarrow{a,b} \mathbf{M}_2$$

(2) Um diagrama no formato

$$\mathbf{M}_1 \longrightarrow \mathbf{M}_2$$

denota que a máquina executa a computação de $\mathbf{M}_2$ após a computação de $\mathbf{M}_1$ terminar independentemente de qual símbolo a máquina está enxergando na fita quando $\mathbf{M}_1$ para. Este diagrama pode ser escrito de forma mais compacta como

$$\mathbf{M}_1\mathbf{M}_2$$

(3) Um diagrama no formato

$$\mathbf{M}_1 \xrightarrow{a \neq \sigma} \mathbf{M}_2$$

denota que a máquina executa a computação de $\mathbf{M}_2$ após a computação de $\mathbf{M}_1$ terminar se o símbolo que a máquina está enxergando na fita quando $\mathbf{M}_1$ para for qualquer símbolo diferente de σ. Repare que, nesta situação, o símbolo a que aparece na aresta não é o símbolo que efetivamente está escrito na fita. Este a pode ser pensado como uma "variável de programação", que armazena *qualquer* símbolo que esteja na casa atual da fita, desde que este símbolo não seja σ. Uma outra forma de escrever este diagrama é

$$\mathbf{M}_1 \xrightarrow{\overline{\sigma}} \mathbf{M}_2$$

Além das notações acima, temos também mais algumas máquinas bastante úteis:

- $\mathbf{E}_{\sigma}$ = Máquina que move o cabeçote para a esquerda *até* achar o símbolo σ. Esta máquina é uma abreviação do diagrama

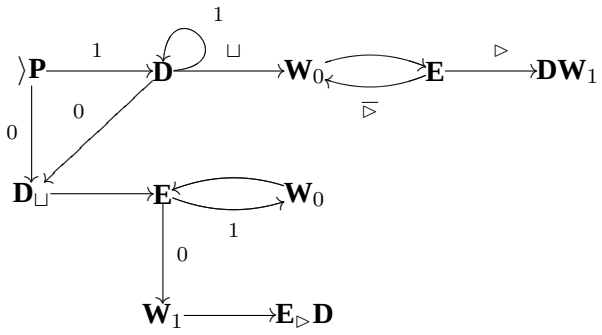
- $\mathbf{D}_{\sigma}$ = Máquina que move o cabeçote para a direita *até* achar o símbolo σ. Esta máquina é uma abreviação do diagrama

- $\mathbf{E}_{\overline{\sigma}}$ = Máquina que move o cabeçote para a esquerda *até* achar um símbolo diferente de σ, isto é, *enquanto* acha σ. Esta máquina é uma abreviação do diagrama

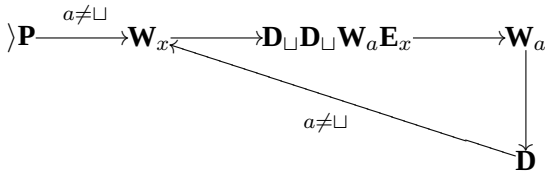
- $\mathbf{D}_{\overline{\sigma}}$ = Máquina que move o cabeçote para a direita *até* achar um símbolo diferente de σ, isto é, *enquanto* acha σ. Esta máquina é uma abreviação do diagrama

EXEMPLO 13.7. Vamos construir o diagrama de composição para a Máquina de Turing que computa a função $f(x) = x + 1$, onde a entrada e a saída da função são codificadas em binário na fita. A máquina deve varrer o número da direita para esquerda, trocando os uns por zeros, até encontrar o primeiro zero. Este zero é então trocado por um e a máquina para. O único caso que não é coberto por este procedimento é o de um número formado apenas por uns. Neste caso, se o número possui k uns, a resposta deve ser um algarismo um seguido de k algarismos zero.

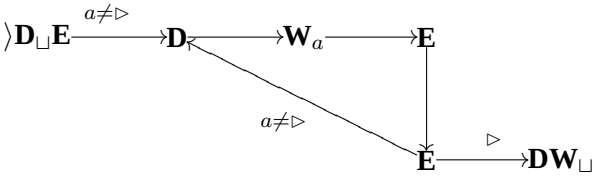
O cabeçote começa na primeira posição à direita do símbolo $\triangleright$.



EXEMPLO 13.8. Vamos agora construir uma máquina de copiar, isto é, uma máquina que começa com uma palavra $w = \sigma_1 \ldots \sigma_k$ escrita no início da fita (a configuração inicial da fita é $\triangleright\underline{\sigma_1}\sigma_2 \ldots \sigma_k$) e termina a computação com a configuração $\triangleright\sigma_1\sigma_2 \ldots \sigma_k \sqcup \sigma_1\sigma_2 \ldots \sigma_k$, tendo escrito uma segunda cópia da palavra na fita.

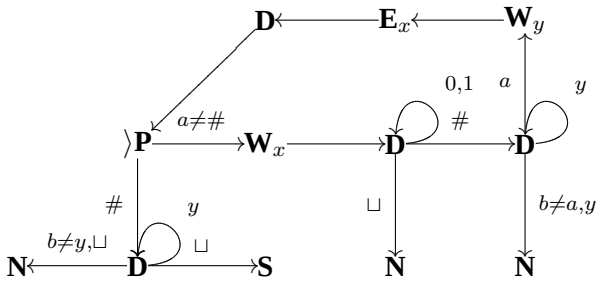


EXEMPLO 13.9. Vamos agora construir uma máquina que realiza um shift-right na fita.



EXEMPLO 13.10. Vamos construir uma máquina que *decide* a linguagem $L = \{w\#w : w \in \{0,1\}^*\}$. Vamos considerar que a entrada é formada apenas pelos

símbolos $\{0, 1, \#\}$, isto é, os símbolos x e y pertencem apenas ao alfabeto da fita.



3. Generalizações da Máquina de Turing

Um fato muito interessante e importante a respeito do poder computacional das máquinas de Turing diz respeito ao que acontece quando tentamos aumentar seus recursos em busca de maior poder de computação. Se, ao invés de utilizarmos a máquina de Turing padrão (conforme definida no início do capítulo), optarmos por utilizar uma máquina com uma fita infinita nos dois sentidos ou com múltiplas fitas ou com múltiplas cabeças de leitura e escrita ou com uma movimentação da(s) cabeça(s) de leitura e escrita menos restrita (ao invés da movimentação de apenas uma casa para a esquerda ou para a direita de cada vez), não iremos obter nenhum ganho relevante de poder computacional. Qualquer linguagem decidida ou semi-decidida e qualquer função computada por máquinas de Turing com várias fitas, cabeças de leitura e escrita, fitas infinitas nos dois sentidos ou políticas menos restritas de movimentação da(s) cabeça(s) de leitura e escrita pode ser decidida, semi-decidida ou computada, respectivamente, por uma máquina de Turing padrão. Além disso, o número de passos necessário para que a máquina de Turing padrão realize a tarefa é apenas polinomialmente maior do que o número de passos executado pela máquina de Turing com recursos adicionais para realizar a mesma tarefa.

Vamos analisar em mais detalhes o caso da Máquina de Turing de múltiplas fitas. Um diagrama esquemático de uma Máquina de Turing com três fitas pode ser visto na ilustração a seguir (Figura 4).

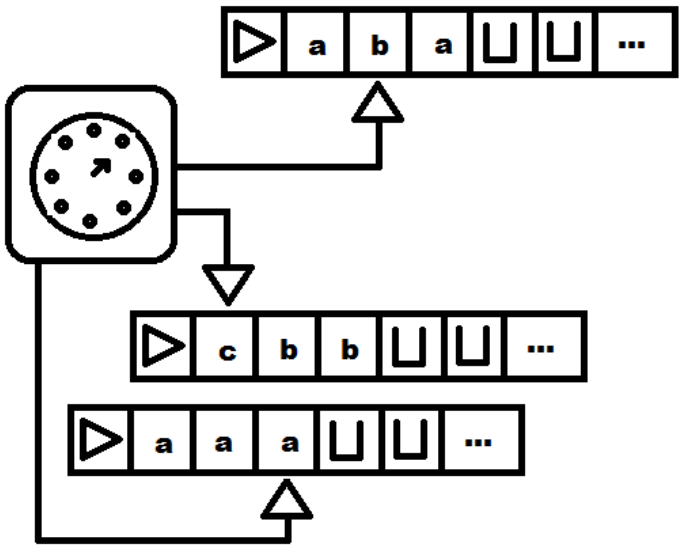


FIGURA 4. Ilustração de uma Máquina de Turing de Múltiplas Fitas

A função de transição de uma Máquina com k fitas é definida da seguinte forma:

$$\delta : (Q - F) \times \Sigma^k \to Q \times ((\Sigma - \{\triangleright\}) \cup \{\leftarrow, \rightarrow\})^k.$$

Isto significa que a ação realizada em cada uma das fitas depende dos símbolos lidos em todas as fitas. Em outras palavras, a ação realizada em uma fita não depende apenas do símbolo lido nesta fita, mas também de todos os outros símbolos lidos.

Vamos mostrar agora, em termos gerais, como uma Máquina de Turing de múltiplas fitas pode ser simulada por uma Máquina de Turing padrão.

A primeira etapa consiste em representar o conteúdo das k fitas da máquina original na fita única da máquina padrão. Para esta etapa, é muito importante lembrarmos que cada fita de uma Máquina de Turing possui apenas uma quantidade *finita* de casas que não está em branco em qualquer momento da computação. Isto significa que, em qualquer fita, existe sempre uma casa a partir da qual todas as casas à direita estão em branco. Ao representarmos as fitas da máquina original na fita da máquina padrão, iremos descartar estes trechos das fitas em que não há mais nenhuma casa que não esteja em branco.

Finalmente, para viabilizarmos a representação do conteúdo das várias fitas da máquina original na fita da máquina padrão, precisamos adicionar ao alfabeto da fita da máquina padrão um símbolo novo, que não ocorre no alfabeto das fitas da máquina original. Este símbolo funcionará como separador do conteúdo das diversas fitas. Utilizaremos como símbolo separador o $\#$.

Assim, a representação das 3 fitas da máquina exibida na Figura 4 na fita única da máquina padrão fica da seguinte forma:

$\triangleright$	$\#$	a	b	a	$\#$	c	b	b	$\#$	a	a	a	$\#$	$\sqcup$	$\ldots$

Já representamos as k fitas da máquina original na fita única da máquina padrão, mas agora precisamos de uma maneira de representar os k *cabeçotes* da máquina original dispondo do cabeçote único da máquina padrão. Para isto, duplicamos o alfabeto da fita da máquina padrão (com exceção do $\triangleright$), criando uma segunda cópia de cada símbolo deste alfabeto, com a diferenciação de que estas cópias terão um ponto sobre o símbolo. Por exemplo, na máquina da Figura 4, o alfabeto das fitas é o conjunto $\{\triangleright, \sqcup, a, b, c\}$. Na máquina padrão, este alfabeto é inicialmente expandido para $\{\triangleright, \sqcup, a, b, c, \#\}$. Após a duplicação do alfabeto que acabamos de descrever, o alfabeto se torna $\{\triangleright, \sqcup, a, b, c, \#, \dot{\sqcup}, \dot{a}, \dot{b}, \dot{c}, \dot{\#}\}$.

Os símbolos com ponto podem ser usados para especificar, na fita da máquina padrão, onde estariam os cabeçotes da máquina original. Assim, voltando ao exemplo da Figura 4, como o cabeçote da primeira fita está sobre o b, o símbolo b na primeira parte da fita da máquina padrão é substituído por $\dot{b}$. O mesmo acontece com o símbolo c na segunda parte da fita e o último símbolo a na terceira parte da fita, pois estas são as posições dos cabeçotes da segunda e terceira fitas na máquina original. Desta forma, a representação completa do conteúdo das fitas e das posições dos cabeçotes da máquina original fica da seguinte maneira:

$\triangleright$	$\#$	a	$\dot{b}$	a	$\#$	$\dot{c}$	b	b	$\#$	a	a	$\dot{a}$	$\#$	$\sqcup$	$\ldots$

Agora que já sabemos descrever a configuração da máquina original na fita da máquina padrão, precisamos simular as transições da máquina original. Vamos

considerar que a máquina original tem k fitas. Em uma primeira etapa, o cabeçote da máquina padrão deve varrer toda a fita desde o $\triangleright$ até encontrar o $(k+1)$-ésimo símbolo $\#$, pois este é o símbolo que indica o final da parte escrita da fita. Durante esta varredura, a máquina deve armazenar em seu estado os símbolos com ponto que ela encontrar, pois estes são os símbolos sobre os quais os cabeçotes da máquina original estariam.

Uma vez concluída a primeira etapa, a máquina tem as informações necessárias para simular a transição da máquina de múltiplas fitas. Para realizar esta transição, o cabeçote da máquina padrão retorna até o $\triangleright$ e novamente varre toda a fita até encontrar o $(k+1)$-ésimo símbolo $\#$. Durante esta segunda varredura, a máquina deve executar, em cada parte da fita, a ação definida pela função de transição da máquina original para cada uma das suas fitas.

Se a ação em uma das fitas da máquina original é manter o cabeçote parado e escrever um novo símbolo, então, na parte equivalente da fita da máquina padrão, um novo símbolo com ponto deve ser escrito no lugar do símbolo com ponto anterior (lembrando que a marcação do ponto indica que um dos cabeçotes da máquina de múltiplas fitas estaria naquela posição). Se a ação for mover o cabeçote uma casa para a esquerda na máquina original, então, na parte equivalente da fita da máquina padrão, o símbolo com ponto deve ser substituído pelo seu análogo sem ponto e o símbolo sem ponto da casa imediatamente à esquerda deve ser substituído pelo seu análogo com ponto. Finalmente, se a ação for mover o cabeçote uma casa para a direita na máquina original, o procedimento é inteiramente similar ao que descrevemos anteriormente para a movimentação do cabeçote para a esquerda.

Existem dois casos particulares de transição que precisam ser pensados com cuidado. O primeiro é o caso de, após o movimento de um dos cabeçotes da máquina original para a esquerda, o ponto marcador na parte equivalente da fita da máquina padrão ser colocado sobre o $\#$ que marca a extremidade *esquerda* deste setor da fita (isto é, um $\dot{\#}$ na extremidade esquerda do setor da fita). Neste caso, a marcação $\dot{\#}$ significa que, na máquina original, o cabeçote da fita correspondente está sobre o $\triangleright$ na extremidade esquerda da fita.

O segundo caso é quando, após o movimento de um dos cabeçotes da máquina original para a direita, o ponto marcador na parte equivalente da fita da máquina padrão é colocado sobre o $\#$ que marca a extremidade *direita* deste setor da fita (isto é, um $\dot{\#}$ na extremidade direita do setor da fita). Neste caso, esta marcação significa que, na máquina original, o cabeçote da fita correspondente se moveu para a primeira casa em branco ($\sqcup$) da parte vazia da fita. Assim, se na próxima transição da máquina original o cabeçote desta fita escrever um novo símbolo nesta casa em branco, haverá problemas na máquina padrão que está realizando a simulação, pois este novo símbolo será escrito sobre o símbolo $\#$, removendo a separação entre dois setores da fita da máquina padrão. Desta forma, quando o ponto marcador é colocado sobre o $\#$ que marca a extremidade *direita* de um setor da fita, um procedimento auxiliar deve ser adotado: ao invés de colocar o ponto marcador sobre este $\#$, deve ser realizado um shift-right na fita de todas as casas a partir deste $\#$ (inclusive) e deve-se escrever o símbolo $\dot{\sqcup}$ (com o ponto) na casa vazia criada pela operação de shift-right.

Agora que discutimos em detalhes a máquina de Turing de múltiplas fitas e sua transformação para a máquina de Turing padrão, vamos analisar uma outra generalização da máquina de Turing: a máquina de Turing *não-determinística*.

Uma máquina de Turing não-determinística é análoga a uma máquina de Turing padrão, tendo como única diferença a sua função de transição, que é da seguinte forma:

$$\Delta : (Q \setminus F) \times \Sigma \to \mathcal{P}(Q \times ((\Sigma \setminus \{\triangleright\}) \cup \{\rightarrow, \leftarrow\})).$$

Se a máquina se encontra em um estado q e a sua cabeça de leitura está sobre uma casa que contém o símbolo σ, então, após a leitura deste símbolo, a máquina escolherá não-deterministicamente um elemento $(q', a) \in \Delta(q, \sigma)$, executará a ação a e irá para o estado q'. Isto é, assim como autômatos finitos não-determinísticos e autômatos de pilha não-determinísticos, uma máquina de Turing não-determinística oferece um *conjunto* de possíveis transições a cada momento, ao invés de uma única possibilidade de transição.

DEFINIÇÃO 13.11. *Dizemos que uma máquina de Turing não-determinística* aceita *(ou semi-decide)* $L \in \Sigma_0^*$ *se, para qualquer palavra* $w \in \Sigma_0^*$*, o seguinte é verdadeiro:* $w \in L$ *se e somente se existe ao menos uma computação em* M *que termina (para) em um estado final.*

DEFINIÇÃO 13.12. *Dizemos que uma máquina de Turing não-determinística* decide $L \in \Sigma_0^*$ *se, para qualquer palavra* $w \in \Sigma_0^*$*,* M *sempre para com entrada* w *e se* $w \in L$*, então existe ao menos uma computação em* M *que para no estado de aceitação (*s*). Repare que isto implica que, se* $w \notin L$*, então todas as computações em* M *param no estado de rejeição (*n*).*

DEFINIÇÃO 13.13. *Finalmente, dizemos que uma máquina de Turing não-determinística* computa *uma função* ϕ *se, para todo conjunto de argumentos da função fornecidos como entrada, a máquina sempre para e, para todas as suas possíveis computações, sempre termina com o valor correto da função na fita.*

Apesar de ser um recurso bastante poderoso, no caso das máquinas de Turing (assim como já havia acontecido com os autômatos finitos), a adição do não-determinismo não representa um ganho de poder computacional. Qualquer linguagem decidida ou semi-decidida e qualquer função computada por máquinas de Turing não-determinísticas pode ser decidida, semi-decidida ou computada, respectivamente, por uma máquina de Turing padrão.

Precisamos mostrar, então, como podemos simular uma máquina de Turing não-determinística utilizando uma máquina de Turing determinística. Podemos utilizar uma máquina de Turing de múltiplas fitas para realizar esta simulação, uma vez que já mostramos acima que é sempre possível, a partir de uma máquina de Turing de múltiplas fitas, obter uma máquina de Turing padrão equivalente.

De fato, para simular uma máquina de Turing não-determinística, vamos construir uma máquina de Turing determinística de 3 fitas. Primeiramente, precisamos garantir que, para todas as entradas da função de transição da máquina de Turing não-determinística original, está estabelecida uma ordenação (e uma enumeração) para todas as possíveis transições que a função de transição permite para aquela

entrada. Como todos os conjuntos de possíveis transições oferecidos pela função de transição de uma máquina de Turing não-determinística são *finitos*, construir esta ordenação e esta enumeração é sempre possível. De fato, suponhamos que, para uma dada máquina de Turing não-determinística, o número máximo de transições que a função de transição oferece para alguma de suas possíveis entradas seja β. Então, os elementos de todos os conjuntos de possíveis transições oferecidos por esta função podem ser ordenados e enumerados por elementos do conjunto $C_\beta = \{1, 2, \ldots, \beta\}$.

A máquina de Turing determinística deverá simular, de maneira ordenada, todas as possíveis computações da máquina de Turing não-determinística original. Entretanto, isto deve ser feito de forma cuidadosa, uma vez que algumas destas computações podem ser infinitas (nunca alcançam um estado final). Caso a máquina determinística simule, em alguma ordem determinada, as possíveis computações da máquina não-determinística original até o fim de cada uma, isto é, prosseguindo para a próxima possível computação apenas quando a simulação da possível computação atual acabar, ela poderá nunca encontrar uma computação da máquina original que alcança um estado final porque corre o risco de ficar presa simulando uma computação da máquina original que nunca termina.

Assim, a maneira com que as simulações das possíveis computações da máquina original são realizadas na máquina determinística precisa ser mais refinada. Primeiramente, verificamos se o estado inicial da máquina original é um estado final. Se for, esta máquina original pode ser simulada por qualquer máquina determinística que pare sem executar ação nenhuma. Assim, este caso não é interessante. Vamos assumir então que o estado inicial da máquina original não é um estado final. Iremos simular, na máquina determinística, as possíveis computações da máquina original, porém com uma limitação no número de passos que a máquina determinística pode executar. Primeiramente, permitimos que a máquina determinística execute a simulação de apenas 1 transição da máquina original. Nesta etapa, vamos simular então, de acordo com a ordem que definimos inicialmente, todas as

possíveis transições a partir do estado inicial. Quando esta etapa acabar, permitimos então que a máquina determinística execute a simulação de 2 transições da máquina original, depois a simulação de 3 transições da máquina original e assim por diante.

Para implementar esta ideia, utilizamos uma máquina determinística de três fitas. Inicialmente, tanto a primeira fita quanto a segunda irão conter a entrada w sobre a qual desejamos realizar a simulação. A terceira fita irá conter símbolos no alfabeto $C_\beta = \{1, 2, \ldots, \beta\}$. Cada símbolo escrito nesta terceira fita indica para a máquina determinística que escolha de transição realizar dentro do conjunto de transições oferecidos pela função de transição da máquina original naquele momento (o símbolo 1 indica a escolha do primeiro elemento do conjunto de transições, o símbolo 2 indica a escolha do segundo elemento do conjunto, e assim por diante). Inicialmente, a terceira fita contém o símbolo 1 na casa imediatamente à direita do $\triangleright$ e nada mais.

Uma vez preenchidas as três fitas com suas configurações iniciais, a máquina determinística opera de acordo com o seguinte procedimento:

(1) A máquina simula, *na segunda fita*, uma das possíveis computações da máquina original, seguindo a escolha de transições de acordo com as orientações codificadas na terceira fita (conforme explicação acima). A cada transição da máquina original que é simulada, o cabeçote da terceira fita deve se mover uma casa para a direita. Caso uma transição deva ser simulada, mas o cabeçote da terceira fita esteja sobre uma casa em branco (o número máximo de transições foi alcançado), ou o símbolo lido na terceira fita não corresponda a uma transição possível na configuração atual (por exemplo, o símbolo na terceira fita é β, sendo que a função de transição da máquina original oferece menos do que β possíveis transições a partir da configuração atual), então a simulação atual deve ser encerrada e a máquina prossegue para o próximo passo.

(2) Se a máquina alcançou um estado correspondente a um estado final da máquina original, ela também termina a sua computação em um estado

final (este estado final deve ser correspondente em termos de aceitação/rejeição ao estado da máquina original, no caso da máquina original ser um decisor). Caso contrário, a máquina prossegue para o próximo passo.

(3) A sequência de símbolos na terceira fita é incrementada (de 1 para 2, de 2 para 3, ..., de $\beta - 1$ para β, de β para 11, de 11 para 12, ..., de $\beta\beta$ para 111, etc.).

(4) O conteúdo da primeira fita é copiado para a segunda fita (a fita onde ocorrem as simulações) e o procedimento é executado novamente, retornando ao primeiro passo.

Entretanto, apesar de podermos simular qualquer máquina de Turing não-determinística com uma máquina de Turing determinística, esta máquina determinística pode ter a necessidade de executar um número de passos exponencialmente maior para realizar uma tarefa do que o número de passos utilizado pela máquina original para realizar a mesma tarefa. Suponha que exista uma computação da máquina de Turing original que termine de forma bem sucedida após n passos. No pior caso, até simular esta computação particular, a máquina determinística deverá simular antes uma transição de cada possível computação a partir do estado inicial (são, no máximo, β possibilidades), depois simular duas transições de cada possível computação (no máximo, β^2 possibilidades) e assim por diante, até executar n transições de cada possível computação (no máximo, β^n possibilidades, considerando que, no pior caso, a computação bem sucedida de n passos será a última a ser simulada). Assim, o número de passos que a máquina determinística irá executar é da ordem de

$$\sum_{i=1}^{n} \beta^i = \frac{\beta(\beta^n - 1)}{\beta - 1} \approx \beta^n,$$

o que efetivamente representa um aumento exponencial do número de passos.

Não se sabe se este aumento exponencial no número de passos é uma característica intrínseca ao não-determinismo ou se ele é derivado do nosso entendimento restrito dele. Este é um dos principais problemas em aberto na teoria da computabilidade e complexidade computacional e voltaremos a discutí-lo no capítulo final.

4. Propriedades de Fechamento das Linguagens Recursivas e Recursivamente Enumeráveis

Nesta seção, vamos analisar o fechamento das Linguagens Recursivas com relação às operações de união, interseção, complemento e diferença e das Linguagens Recursivamente Enumeráveis com relação às operações de união e interseção.

Começamos pelo complemento das Linguagens Recursivas. A estratégia é bem semelhante à que utilizamos para as Linguagens Regulares. Naquele caso, tomamos um autômato finito determinístico para a linguagem regular e invertemos os seus estados finais com os estados não-finais. Agora, no caso das Linguagens Recursivas, basta tomarmos uma máquina de Turing que *decida* uma linguagem recursiva L e invertermos o seu estado final de aceitação (s) com seu estado final de negação (n). Fazendo isto, obtemos uma máquina de Turing que *decide* $\overline{L}$. Logo, a classe das Linguagens Recursivas é fechada por complemento.

Para analisarmos as operações de união e interseção, o argumento é o mesmo tanto para a classe das Linguagens Recursivas quanto para a classe das Recursivamente Enumeráveis. A estratégia também é semelhante ao que fizemos para a união e interseção de Linguagens Regulares, onde construímos um autômato finito determinístico que simulava em paralelo outros dois autômatos finitos determinísticos.

Seja M_1 uma máquina de Turing que decide (no caso do argumento para Linguagens Recursivas) ou aceita (no caso do argumento para Linguagens Recursivamente Enumeráveis) L_1 e M_2 uma máquina de Turing que decide ou aceita (também dependendo se estamos construindo o argumento para Linguagens Recursivas ou Recursivamente Enumeráveis) L_2. Queremos construir máquinas de Turing que decidam/aceitem $L_1 \cup L_2$ e $L_1 \cap L_2$. Para isto, vamos construir máquinas que simulem M_1 e M_2 em paralelo. Estas máquinas serão máquinas de duas fitas, onde em cada fita será simulada uma das máquinas M_1 e M_2. A ideia geral destas máquinas é descrita graficamente na Figura 5.

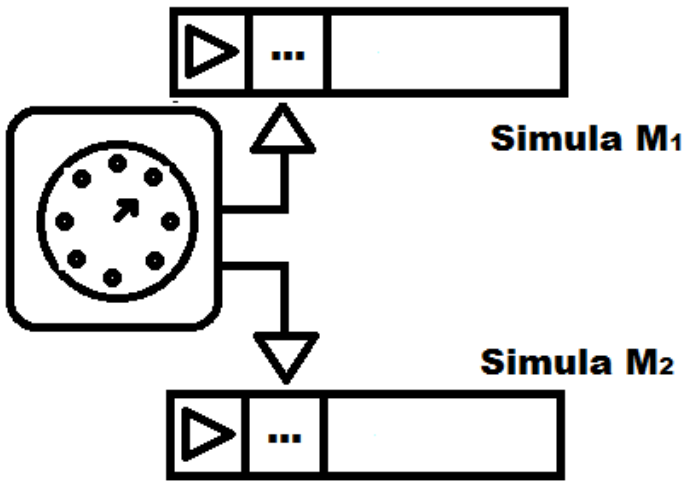


FIGURA 5. Máquina de Turing com duas fitas utilizada na demonstração de propriedades de fechamento

A máquina M de duas fitas irá simular M_1 e M_2 em paralelo em cada uma das suas fitas. Para a operação de união, no caso das Linguagens Recursivas, queremos que a máquina M pare em um estado de aceitação caso pelo menos uma das máquinas M_1 e M_2 parem em um estado de aceitação e que a máquina M pare em um estado de rejeição caso ambas as máquinas M_1 e M_2 parem em seus estados de rejeição. Já no caso das Linguagens Recursivamente Enumeráveis, queremos que a máquina M pare se pelo menos uma das máquinas M_1 e M_2 parar e continue a computação enquanto as duas máquinas continuarem.

Para a operação de interseção, no caso das Linguagens Recursivas, queremos que a máquina M pare em um estado de aceitação caso ambas as máquinas M_1 e M_2 parem em seus estados de aceitação e que a máquina M pare em um estado de rejeição caso pelo menos uma das máquinas M_1 e M_2 parem em um estado de rejeição. Já no caso das Linguagens Recursivamente Enumeráveis, queremos que a máquina M pare se ambas as máquinas M_1 e M_2 pararem e continue a computação enquanto pelo menos uma das duas máquinas continuar.

Vemos então que tanto a classe das Linguagens Recursivas quanto a das Recursivamente Enumeráveis são fechadas tanto para a operação de união quanto para a operação de interseção.

Dado que as Linguagens Recursivas são fechadas por interseção e por complemento, então elas também são fechadas por diferença. Sejam L_1 e L_2 Linguagens Recursivas. Então, como as Linguagens Recursivas são fechadas por complemento, $\overline{L_2}$ também é recursiva. Por outro lado, como elas também são fechadas por interseção, então $L_1 \cap \overline{L_2}$ também é recursiva. Mas $L_1 \cap \overline{L_2} = L_1 - L_2$, logo a classe das Linguagens Recursivas é fechada por diferença.

No próximo capítulo, iremos mostrar que a classe das Linguagens Recursivamente Enumeráveis não é fechada por complemento. Isto, por sua vez, implica que ela também não é fechada por diferença. Suponha, por contradição, que ela seja fechada por diferença. Seja $L \subseteq \Sigma^*$ uma Linguagem Recursivamente Enumerável. Temos que Σ^* é uma Linguagem Regular, logo também é Recursivamente Enumerável. Assim, se as Linguagens Recursivamente Enumeráveis são fechadas por diferença, $\Sigma^* - L$ também é uma Linguagem Recursivamente Enumerável. Mas $\Sigma^* - L = \overline{L}$, o que significaria que a classe das Linguagens Recursivamente Enumeráveis é fechada por complemento, o que contradiz a afirmação que fizemos no início do parágrafo.

Para finalizar esta seção, apresentamos um resultado que relaciona as Linguagens Recursivas e Recursivamente Enumeráveis.

TEOREMA 13.14. *Se tanto L quanto $\overline{L}$ são Linguagens Recursivamente Enumeráveis, então L e $\overline{L}$ são Linguagens Recursivas.*

DEMONSTRAÇÃO. Se L e $\overline{L}$ são Linguagens Recursivamente Enumeráveis, então existem máquinas de Turing M_1 e M_2 que aceitam, respectivamente L e $\overline{L}$. Se construirmos uma máquina M que simula M_1 e M_2 em paralelo, da mesma forma que descrevemos na Figura 5, podemos utilizar esta máquina para decidir L.

Seja w uma palavra no alfabeto em que L está definida. Se $w \in L$, então M_1 irá parar com entrada w e M_2 não irá parar. Por outro lado, se $w \notin L$, então M_2 irá parar com entrada w e M_1 não irá parar. Vemos então que, em qualquer caso, exatamente uma das máquinas irá parar com qualquer entrada w. Assim, a máquina M pode decidir L da seguinte maneira. Se a simulação na primeira fita parar, então

$w \in L$. Já se a simulação na segunda fita parar, então $w \notin L$. Já que M decide L, então L é uma Linguagem Recursiva. Finalmente, como as Linguagens Recursivas são fechadas por complemento, $\overline{L}$ também é uma Linguagem Recursiva.

Vemos então que temos apenas três cenários possíveis com relação a um par de linguagens L e $\overline{L}$. Ou ambas as linguagens são recursivas, ou nenhuma delas é recursivamente enumerável, ou uma delas é recursivamente enumerável (porém não é recursiva) e a outra não é recursivamente enumerável.

5. Exercícios

(1) Considere a máquina de Turing M cujo alfabeto da fita 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 $(q_0, \triangleright\underline{0}0\ 1110)$.

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 com alfabeto da fita $\{a, b, \sqcup, \triangleright\}$, que se move para a esquerda até encontrar três as na fita e então para.

(3) Explique porque as máquinas DE e ED nem sempre têm a mesma saída.

(4) Forneça uma descrição formal $M = (\Sigma_0, \Sigma, Q, q_0, F, \delta)$ de cada uma das máquinas básicas utilizadas para a construção de diagramas de composição. Em particular, descreva a tabela de transição de cada uma delas.

(5) Descreva o diagrama de composição 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) $\{0^{n^2} : n \geq 1\}$.

(6) Descreva o diagrama de composição e a tabela de transições da máquina de Turing 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.

(7) Construa uma máquina que realiza a operação de rotate-right na fita.

(8) Construa uma máquina que realiza a operação de rotate-left na fita.

(9) 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)
$$f(n,m) = \begin{cases} n - m & \text{se } n - m \geq 0 \\ 0 & \text{se } n < m. \end{cases}$$

(10) Construa uma máquina que computa a função $f(w) = w^r$, onde $w \in \{0,1\}^*$.

(11) Descreva uma máquina de Turing que, tendo como entrada uma palavra $w \in \{0,1\}^*$ encontra o símbolo do meio da palavra (se existir!).

(12) 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 $w_2 \in \{c,d\}^*$.

(13) Utilizando a máquina de Turing da questão anterior construa uma máquina de Turing que aceita a linguagem $\{ww : w \in \{0,1\}^*\}$.

(14) Mostre que a linguagem $\{ww : w \in \{0,1\}^*\}$ é recursiva.

(15) Construa uma máquina que *decida* a linguagem $L = \{a^n b^n c^n : n \geq 0\}$. A construção de tal máquina prova que a classe das linguagens recursivas é mais ampla que a classe das linguagens livres de contexto, já que provamos com o Lema do Bombeamento que esta linguagem L não é livre de contexto.

(16) Dê a definição formal de uma máquina de Turing cuja fita é duplamente infinita (isto é, infinita nos dois sentidos). 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?

(17) 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.

(18) 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.

(19) 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 $L_1 \cup L_2$.

(20) A interseção de linguagens recursivas é recursiva? Explique sua resposta.

(21) Mostre que toda linguagem livre de contexto também é uma linguagem recursiva descrevendo como podemos simular um autômato de pilha em um decisor (potencialmente não-determinístico).

(22) Um autômato com duas pilhas é análogo a um autômato de pilha, porém ele possui duas pilhas independentes como memória. A transição é então da forma

$$\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to \mathcal{P}_f(Q \times \Gamma^* \times \Gamma^*),$$

isto é, a transição depende dos símbolos presentes no topo das duas pilhas e o resultado da transição produz ações independentes em cada uma das pilhas. Mostre que podemos simular uma Máquina de Turing em um autômato com duas pilhas e vice-versa.

(23) Um autômato de fila é análogo a um autômato de pilha, porém, enquanto símbolo consultado para a transição (e retirado da memória) continua sendo aquele no topo, os novos símbolos inseridos pela transição são inseridos no fundo. Mostre que podemos simular uma Máquina de Turing em um autômato de fila e vice-versa.

CAPíTULO 14

Poder e Limitações das Máquinas de Turing

Neste capítulo, continuamos o estudo das Máquinas de Turing que iniciamos no capítulo anterior, discutindo primordialmente o poder e as funcionalidades das Máquinas de Turing de um lado e as suas limitações inerentes de outro. Este estudo nos permite estabelecer parâmetros formais para os poderes e limitações dos computadores.

1. Tese de Church-Turing

Como vimos no capítulo anterior, uma Máquina de Turing é um modelo computacional simples. Ao longo de todo o texto, já vínhamos estudando também outros modelos computacionais mais simples. Este estudo nos permite abordar a questão do que pode ser computado por uma máquina e a questão ainda mais intrigante do que *não pode* ser computado.

No capítulo anterior, vimos que a Máquina de Turing é um modelo mais poderoso do que os estudados anteriormente e vimos também que a adição de novas capacidades e funcionalidades a ela, como múltiplas fitas, múltiplos cabeçotes, fita infinita em ambos os sentidos e até mesmo o não-determinismo não aumentam o seu poder computacional. Mesmo que adicionemos a ela a capacidade de acessar qualquer casa da fita através de um sistema de endereçamento, tornando-a mais parecida com os computadores reais, ainda assim o seu poder computacional não irá aumentar.

É interessante recordarmos neste momento o que queremos dizer com “poder computacional”. Não estamos falando de velocidade. Certamente, uma máquina com múltiplas fitas, com não-determinismo ou com endereçamento de memória realizará computações mais rapidamente do que a Máquina de Turing tradicional. Entretanto, qualquer problema que a Máquina de Turing for incapaz de resolver

continuará insolúvel também com o uso de Máquinas de Turing com quaisquer das funcionalidades extras.

Além dos modelos baseados na Máquina de Turing, outros pesquisadores desenvolveram, de forma independente, outros modelos formais de computação. Como exemplos, podemos citar o λ-Cálculo, proposto por Alonzo Church, e as Funções μ-Recursivas, propostas por Stephen Kleene. Estes modelos são todos equivalentes entre si e equivalentes ao modelo da máquina de Turing, no sentido de que a classe de funções computáveis definida por qualquer destes modelos é a mesma classe das funções computáveis por uma máquina de Turing.

Isto sugere que as máquinas de Turing são efetivamente um limite superior natural para o que pode ser computado. Desta forma, podemos adotar a noção de uma máquina de Turing que decide uma linguagem ou que computa uma função como modelo teórico da noção de "algoritmo". Esta noção de que um problema de decisão ou uma função são "algoritmicamente computáveis" se e somente se existe uma máquina de Turing que o decide ou que a computa, respectivamente, é conhecida como *Tese de Church-Turing*.

Esta tese não pode ser provada e pode, de fato, ser refutada se eventualmente for desenvolvido um modelo de computação mais poderoso do que uma máquina de Turing. Entretanto, isto é considerado altamente improvável, devido a todos os modelos de computação independentemente desenvolvidos até o momento terem se mostrado equivalentes ao modelo da máquina de Turing.

Como curiosidade, nem mesmo o novo modelo de computação que vem surgindo como potencial alternativa nos últimos anos, a Computação Quântica, refutaria a Tese de Church-Turing. Em um computador quântico, algumas tarefas podem ser realizadas de forma mais rápida, mas, novamente, tudo que um computador tradicional é incapaz de resolver permanece insolúvel para um computador quântico.

Adotar uma noção precisa de algoritmo nos abre a possibilidade de formalmente provarmos que certos problemas computacionais *não podem* ser resolvidos

por nenhum algoritmo, impondo limites concretos ao que um computador é capaz de fazer. Iremos tratar desta questão na última seção deste capítulo.

2. 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 $\mathcal{U}$.

Assim, se podemos enxergar uma máquina de Turing como um modelo formal de um algoritmo, podemos enxergar a máquina de Turing universal como um modelo formal de um computador programável, isto é, um computador que pode executar qualquer algoritmo, desde que receba o conjunto de instruções deste algoritmo.

Há muitas maneiras diferentes de construir a máquina $\mathcal{U}$. Na construção que faremos, o alfabeto da entrada da máquina será

$$\Sigma_{\mathcal{U}} = \{0, 1, \sigma, q, X, Y, Z, \#, a, b\}$$

e o alfabeto da fita será $\Sigma_{\mathcal{U}} \cup \{\triangleright, \sqcup\}$.

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[1]. 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 da fita Σ_M 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,

[1]Em unário, o número n é representado por $000\cdots0 = 0^n$.

reservaremos $m+3$ casas para representar cada símbolo, preenchendo as casas redundantes com 1's. 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 extras. A outra casa é usada para pôr o prefixo σ, que indica que a sequência 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 $\leftarrow$. Adotaremos a convenção de que $\triangleright$, $\rightarrow$ e $\leftarrow$ serão sempre os três primeiros símbolos a serem enumerados. Assim,

símbolo	código
$\triangleright$	$\sigma 01^{m+1}$
$\rightarrow$	$\sigma 0^2 1^m$
$\leftarrow$	$\sigma 0^3 1^{m-1}$

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 $\delta(q_i, \sigma_j) = (q_r, \sigma_s)$. Como $\sigma_s \in \Sigma_M \cup \{\rightarrow, \leftarrow\}$, 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, o mesmo valendo para todos os outros símbolos com expoentes no esquema acima.

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 X's.

$\triangleright$	X	S_1	X	S_2	$\cdots$	X	S_t	Y	Q	Z	σ	u_1	σ	u_2	$\#$	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 $\#$ e não de σ. A $\#$ 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 $\#$ ocupa o lugar do símbolo σ no início da codificação do símbolo da fita de M sobre o qual o cabeçote de M começaria. Os símbolos do trecho da fita de $\mathcal{U}$ que contém a descrição da máquina M (o trecho à direita de $\triangleright$ e à esquerda de Z) formam uma palavra no alfabeto $\Sigma_{\mathcal{U}}$. Vamos denotar esta palavra por $c(M)$. Por outro lado, a palavra formada pelos símbolos do trecho da fita à direita de Z, que contém a codificação da entrada w fornecida à máquina M, será denotada por $c(w)$.

EXEMPLO 14.1. 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 6 casas para cada símbolo (já que há 3 estados e 3 símbolos), sendo que as casas redundantes devem ser preenchidas com 1's.

estado	código	símbolo	código
q_0	$q01^2$	$\triangleright$	$\sigma 01^4$
q_1	$q0^21$	$\rightarrow$	$\sigma 0^2 1^3$
h	$q0^3$	$\leftarrow$	$\sigma 0^3 1^2$
		$\sqcup$	$\sigma 0^4 1$
		0	$\sigma 0^5$

Para ilustrar como codificar uma transição na fita de $\mathcal{U}$, a primeira transição descrita na tabela que aparece no início deste exemplo ($\delta(q_0, 0) = (q_1, \sqcup)$) é codificada como

X	$q01^2$	$\sigma 0^5$	$q0^21$	$\sigma 0^4 1$	X

O processamento 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, de acordo com o padrão exposto acima; 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 $\triangleright$ e verifica que os X's, q's, σ's, Y's e Z's estão posicionados na ordem correta. À medida que faz isto, $\mathcal{U}$ deve comparar se as representações de todos os estados possuem o mesmo comprimento, fazendo também uma comparação análoga para as representações de todos os símbolos. 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 descrita como um ciclo com 3 etapas:

Primeira Etapa: No início, o cabeçote está parado logo à direita do $\triangleright$. Então, a máquina deve varrer o trecho entre o primeiro X e o Y, buscando localizar uma quádrupla que comece com a representação do par (Q, τ), onde τ é o símbolo representado no setor da fita à direita do Z que se inicia com a marcação $\#$ e Q é o estado representado entre Y e Z. À medida que vai tentando achar este par, a máquina vai trocando os 0's e 1's (da esquerda para a direita) por a's e b's, respectivamente. Isto permite identificar até onde já foi feita a busca. Ao achar o par desejado, a máquina também troca os 0's e 1's deste par por a's e b's. Com isto, o $q0^i1^{n-i}$ mais à esquerda da fita corresponde ao estado seguinte a Q na transição de M por τ e o $\sigma 0^j 1^{m+2-j}$ mais à esquerda da fita corresponde à ação na fita que a transição de M por τ realizará. 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, o estado seguinte ao atual na transição de M a partir de Q e τ deve ser copiado para o campo que corresponde ao estado atual de M, entre Y e Z. À medida que a cópia do estado é realizada, os 0's e 1's da representação deste estado devem ser trocados por a's e b's, para que a máquina saiba quais símbolos já foram copiados e quais ainda faltam. Consideramos agora o σ seguido de algum número que fica mais à esquerda da fita. Como explicado na etapa anterior, este trecho nos diz o que M faria com sua fita de entrada nesta transição. A máquina $\mathcal{U}$ irá simular então a ação na fita que M realizaria, fazendo uma ação na

região da fita após o Z. A ação feita por $\mathcal{U}$ está descrita na tabela abaixo, de acordo com cada possibilidade.

Símbolo	Ação
$\sigma 0^1 1^{m+1}$ (representa $\rightarrow$)	move o $\#$ para o lugar do σ seguinte (simula a movimentação do cabeçote de M para a direita)
$\sigma 0^2 1^m$ (representa $\leftarrow$)	move o $\#$ para o lugar do σ anterior (simula a movimentação do cabeçote de M para a esquerda)
$\sigma 0^j 1^{m+2-j}$ e $j \geq 3$	substitui o segmento entre $\#$ e σ por $0^j 1^{m+2-j}$ (simula a escrita de um novo símbolo)

Terceira Etapa: Agora, $\mathcal{U}$ rebobina o cabeçote até $\triangleright$, trocando todos os as e bs por 0s e 1s, respectivamente. Finalmente, o cabeçote retorna à casa logo à direita de $\triangleright$. Assim, $\mathcal{U}$ está pronta para começar um novo ciclo.

3. O Problema da Parada

Nesta seção, vamos demonstrar três resultados importantes a partir de um mesmo raciocínio:

(1) Existem problemas indecidíveis, isto é, existem problemas que não podem ser "algoritmicamente resolvidos" ou, de maneira equivalente, que não podem ser decididos por uma máquina de Turing (isto impõe limites àquilo que um computador pode fazer);

(2) A classe de linguagens recursivas é um subconjunto próprio da classe de linguagens recursivamente enumeráveis, isto é, existem linguagens que são recursivamente enumeráveis, mas não são recursivas e

(3) A classe das linguagens recursivamente enumeráveis não é fechada por complemento.

Suponha que exista um algoritmo P que receba como entrada o código de um outro algoritmo A e uma entrada e para A e responda sempre corretamente se o algoritmo A para ou não quando é executado com entrada e. Isto é, P responde "Sim" se o algoritmo A para com a entrada e e responde "Não" se o algoritmo P não para (entra em loop) com a entrada e. Dizemos que este algoritmo hipotético P resolveria o *Problema da Parada*.

Vamos mostrar que o algoritmo P não pode existir. Em outras palavras, vamos mostrar que o Problema da Parada não é algoritmicamente solúvel.

Inicialmente, consideramos a linguagem $H \subseteq \Sigma_{\mathcal{U}}^*$ definida da seguinte forma:

$$H = \{c(M)Zw : \text{ a máquina de Turing } M \text{ para com a entrada } w \in \Sigma_{\mathcal{U}}^*\},$$

onde $c(M)$ é a codificação de M no alfabeto $\Sigma_{\mathcal{U}}$ que é utilizada para representar M na fita da Máquina de Turing Universal $\mathcal{U}$. Repare em uma hipótese implícita na definição acima: não são todas as máquinas de Turing que contribuem para a formação de palavras em H, apenas aquelas máquinas que são capazes de processar uma palavra w formada por símbolos do alfabeto $\Sigma_{\mathcal{U}}$.

Primeiramente, notamos que esta é uma linguagem recursivamente enumerável. Em primeiro lugar, sabendo que o alfabeto das palavras w que contribuem para a formação das palavras em H está fixado como $\Sigma_{\mathcal{U}}$, podemos facilmente construir uma máquina de Turing $\mathcal{T}_{\Sigma_{\mathcal{U}}}$ que transforme uma entrada $\triangleright c(M)Zw$ em $\triangleright c(M)Zc(w)$. Deixamos a construção desta máquina como exercício (Exercício 4). Com a máquina $\mathcal{T}_{\Sigma_{\mathcal{U}}}$ em mãos, não é difícil concluir que a máquina M_H definida como $\mathcal{T}_{\Sigma_{\mathcal{U}}} \to \mathcal{U}$ aceita a linguagem H. Assim, uma vez que há uma máquina de Turing que aceita H, ela é uma linguagem recursivamente enumerável.

Vamos agora considerar a linguagem $H' \subseteq \Sigma_{\mathcal{U}}^*$ definida da seguinte forma:

$$H' = \{c(M) : \text{ a máquina de Turing } M \text{ para com entrada } c(M)\}.$$

Esta é certamente uma linguagem bastante exótica, mas algumas máquinas podem processar entradas construídas sobre o mesmo alfabeto que a palavra $c(M)$, isto é, o alfabeto $\Sigma_{\mathcal{U}}$. Nestes casos, faz sentido perguntar o que aconteceria com a

máquina M ao receber a sua própria codificação como entrada. Aquelas máquinas que param neste caso, são as máquinas que aparecem na linguagem H'.

Esta situação não é tão absurda quanto parece inicialmente. Como analogia, podemos imaginar um compilador para programas escritos na linguagem C. Este compilador também foi, ele mesmo, escrito em alguma linguagem de programação, que pode ter sido a própria linguagem C. Neste caso, faria sentido re-compilar o código fonte deste compilador utilizando o próprio compilador.

Sabendo que H é recursivamente enumerável, podemos concluir que H' também é recursivamente enumerável. Seja $\mathcal{C}$ uma máquina de Turing que transforma uma entrada $\triangleright c(M)$ em $\triangleright c(M)Zc(M)$. Tendo a máquina M_H que aceita H, é simples concluir que a máquina $M_{H'}$ definida como $\mathcal{C} \rightarrow M_H$, aceita a linguagem H'. Assim, uma vez que há uma máquina de Turing que aceita H', ela também é recursivamente enumerável.

Sabemos agora que H' é recursivamente enumerável. Colocamo-nos então a seguinte pergunta: H' é uma linguagem recursiva?

Pela propriedade de fechamento das linguagens recursivas pela operação de complemento, se H' for recursiva, então $\overline{H'}$ também será, necessariamente, recursiva. Surpreendentemente, isto está longe de ser verdade, como veremos em breve.

Antes de prosseguirmos, vale a pena analisarmos com calma que palavras compõem a linguagem $\overline{H'}$. Em primeiro lugar, temos que $H' \subseteq \Sigma_{\mathcal{U}}^*$, logo $\overline{H'} = \Sigma_{\mathcal{U}}^* - H'$. Sendo assim, $\overline{H'}$ conterá as palavras formadas por símbolos do alfabeto $\Sigma_{\mathcal{U}}$ que não satisfazem a propriedade de H'. Podemos então dividir as palavras de $\overline{H'}$ em três grupos:

(1) Palavras w tais que $w = c(M)$, onde M é uma Máquina de Turing que *não para* com entrada $c(M)$;
(2) Palavras w tais que $w = c(M)$, onde M é uma Máquina de Turing que não processa entradas no alfabeto $\Sigma_{\mathcal{U}}$ (lembre-se que $c(M) \in \Sigma_{\mathcal{U}}^*$, para qualquer máquina M) e

(3) Palavras w que são formadas por símbolos de $\Sigma_{\mathcal{U}}$, mas que não representam a codificação de nenhuma Máquina de Turing.

Veremos agora que a linguagem $\overline{H'}$ e, por consequência, as linguagens H' e H não são recursivas.

TEOREMA 14.2. *A linguagem $\overline{H'}$ não é sequer recursivamente enumerável.*

DEMONSTRAÇÃO. Suponhamos, por contradição, que $\overline{H'}$ é recursivamente enumerável. Então existe uma máquina de Turing N que aceita $\overline{H'}$. Dada esta máquina N, podemos então perguntar se $c(N)$ pertence a H' ou a $\overline{H'}$ (qualquer palavra $c(M)$, onde M é uma Máquina de Turing, deve pertencer a exatamente uma das duas linguagens, uma vez que elas são complementares).

Suponhamos inicialmente que $c(N) \in H'$ (e $c(N) \notin \overline{H'}$). Isto implica, pela definição de H', que

(3.1) $$N \text{ para com entrada } c(N).$$

Por outro lado, recapitulando, $\overline{H'}$ é a linguagem aceita pela máquina N. Assim, se $c(N)$ não pertence à linguagem aceita por N, então, pela definição de aceitação por uma Máquina de Turing,

(3.2) $$\text{a máquina } N \text{ não para com entrada } c(N).$$

As afirmativas (3.1) e (3.2) formam uma contradição. Logo, não é possível que $c(N) \in H'$.

Suponhamos então que $c(N) \in \overline{H'}$ (e $c(N) \notin H'$). Se $c(N) \in \overline{H'}$, precisamos inicialmente determinar a qual dos três grupos de palavras descritos anteriormente a palavra $c(N)$ pertence. Em primeiro lugar, $c(N)$ é uma codificação de uma Máquina de Turing, logo esta palavra não pode pertencer ao grupo (3). Em segundo lugar, como N é a máquina que aceita $\overline{H'}$ e $\overline{H'} \in \Sigma_{\mathcal{U}}^*$, então N processa entradas no alfabeto $\Sigma_{\mathcal{U}}$. Assim, $c(N)$ também não pode pertencer ao grupo (2). Por exclusão, temos então que, se $c(N) \in \overline{H'}$, então $c(N)$ deve pertencer ao grupo (1), o que

significa que

(3.3) N não para com entrada $c(N)$.

Por outro lado, $\overline{H'}$ é a linguagem aceita pela máquina N. Logo, se $c(N)$ pertence à linguagem aceita por N, então, pela definição de aceitação por uma Máquina de Turing,

(3.4) a máquina N para com entrada $c(N)$.

As afirmativas (3.3) e (3.4) também formam uma contradição. Logo, não é possível que $c(N) \in \overline{H'}$.

Parece que chegamos a absurda conclusão de que $c(N)$ não pertence nem a H' nem a $\overline{H'}$. Se não é possível que $c(N) \in H'$ e também não é possível que $c(N) \in \overline{H'}$, concluímos então que a máquina N que aceitaria $\overline{H'}$ não pode existir. Isto significa que $\overline{H'}$ não é recursivamente enumerável.

Dado que $\overline{H'}$ não é recursivamente enumerável, ela também não é recursiva. Podemos concluir então que a linguagem H' não é recursiva. Esta linguagem H' nos fornece um exemplo de uma linguagem que é recursivamente enumerável, mas não é recursiva.

Obtemos assim dois resultados muito importantes, que complementam os resultados que havíamos obtido no capítulo anterior.

COROLÁRIO 14.3. *Existem linguagens recursivamente enumeráveis que não são recursivas.*

DEMONSTRAÇÃO. A linguagem H' é um exemplo de linguagem recursivamente enumerável que não é recursiva.

COROLÁRIO 14.4. *O complemento de uma linguagem recursivamente enumerável não é necessariamente recursivamente enumerável, isto é, a classe das linguagens recursivamente enumeráveis não é fechada pela operação de complemento.*

DEMONSTRAÇÃO. A linguagem H' é recursivamente enumerável, mas o seu complemento $\overline{H'}$ não é.

Deixamos como exercício (Exercício 5), a demonstração, a partir do corolário acima, de que a classe das linguagens recursivamente enumeráveis também não é fechada pela operação de diferença.

Podemos enunciar novamente o Problema da Parada que descrevemos no início da seção, agora de uma maneira mais formal, com a utilização de máquinas de Turing:

PROBLEMA 14.5. Vamos fixar um alfabeto Σ. Dada uma máquina de Turing M neste alfabeto e uma palavra $w \in \Sigma^*$, existe uma máquina de Turing $\mathcal{P}_\Sigma$ que, tendo $c(M)Zw$ como entrada decide se M para 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 $\mathcal{A}_M$ a máquina que transforma a entrada $\triangleright w$ em $\triangleright c(M)Zw$. Se $w \in L$, então a máquina M para na entrada w; portanto $\mathcal{A}_M \to \mathcal{P}_\Sigma$ para no estado *sim*. Do contrário, M não para com entrada w, mas neste caso $\mathcal{A}_M \to \mathcal{P}_\Sigma$ também para, só que no estado *não*. Assim, enquanto M *aceita* L, $\mathcal{A}_M \to \mathcal{P}_\Sigma$ *decide* L.

Diante disto, podemos concluir que o problema da parada não pode ter uma resposta afirmativa porque, como vimos anteriormente, existem linguagens recursivamente enumeráveis que não são recursivas.

A resposta negativa para o problema da parada tem importantes consequências, tanto de natureza teórica quanto prática. Por exemplo, seria certamente desejável ter algoritmo que, recebendo um programa A e uma entrada e de A, decidisse se A para 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 consequência 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:

(1) Dadas duas linguagens livres de contexto, determinar se sua interseção é livre de contexto;
(2) Dada uma linguagem livre de contexto, determinar se ela é regular;
(3) Dada uma linguagem livre de contexto, determinar se ela é inerentemente ambígua.

Sabendo que um dado problema, como o Problema da Parada por exemplo, é indecidível, podemos provar que outros problemas são também indecidíveis através de um método conhecido como *redução*. Sejam $L_1, L_2 \subseteq \Sigma_0^*$ duas linguagens. Dizemos que existe uma redução de L_1 para L_2 se existe uma função computável $\tau : \Sigma_0^* \to \Sigma_0^*$ tal que $w \in L_1$ se e somente se $\tau(w) \in L_2$. Se sabemos que L_1 não é recursiva, podemos mostrar que L_2 também não é recursiva exibindo uma redução de L_1 para L_2. Suponha, por contradição, que existe uma redução de L_1 para L_2, L_1 não é recursiva, mas L_2 é. Então, existe uma função $\tau : \Sigma_0^* \to \Sigma_0^*$ tal que $w \in L_1$ se e somente se $\tau(w) \in L_2$ computada por uma máquina de Turing M_1. Além disso, como L_2 é recursiva, existe uma máquina de Turing M_2 que a decide. Desta forma, uma máquina de Turing que simule o funcionamento de M_1 seguida de M_2 decidiria L_1, o que é uma contradição. Desta forma, L_2 também é indecidível.

4. Exercícios

(1) Explique a tese de Church-Turing e dê dois argumentos a seu favor.
(2) Construa a fita de entrada para que a máquina de Turing universal simule a computação da máquina de Turing do exercício 1 do capítulo 13 partir da configuração $(q_0, \triangleright\underline{0}01110)$.

(3) Descreva a fita de $\mathcal{U}$ correspondente à máquina do exemplo 14.1 com entrada $000\sqcup$.

(4) Fixando um alfabeto Σ, construa uma máquina de Turing que transforme uma entrada $\triangleright w$, onde $w \in \Sigma^*$ em $\triangleright c(w)$, onde $c(w)$ é a codificação da palavra w no alfabeto $\Sigma_{\mathcal{U}}$ da máquina de Turing universal $\mathcal{U}$.

(5) Mostre que a classe das linguagens recursivamente enumeráveis não é fechada pela operação de diferença.

(6) Enuncie o Problema da Parada e explique por que ele é indecidível.

CAPíTULO 15

Introdução à Teoria da Complexidade

A classificação dos problemas em decidíveis e indecidíveis não nos fornece toda a informação necessária a respeito de um problema. Muitos problemas são decidíveis, mas, na prática, eles precisam de um tempo muito alto para serem resolvidos. Tais problemas são chamados de *intratáveis*. Para determinar quais problemas decidíveis são tratáveis na prática e quais são intratáveis e para estabelecer uma hierarquia entre os problemas com relação à quantidade de tempo e de memória necessários para resolvê-los, desenvolveu-se a teoria da complexidade de algoritmos e de classes de complexidade.

1. Complexidade de Tempo

DEFINIÇÃO 15.1. *Uma máquina de Turing é dita* polinomialmente limitada *se há uma função polinomial $p(n)$ tal que, para toda entrada $w \in \Sigma_0^*$, qualquer computação a partir do estado inicial da máquina sempre alcança um estado de parada em, no máximo, $p(|w|)$ passos.*

DEFINIÇÃO 15.2. *A classe P é definida como a classe de todas as linguagens que podem ser decididas por uma máquina de Turing* determinística *polinomialmente limitada.*

Em geral, consideramos um problema como sendo *tratável* se e somente se ele pertence a P. O tempo de execução de um problema em P aumenta polinomialmente com relação ao tamanho da entrada, por isso, mesmo para entradas de tamanho grande, o algoritmo ainda é capaz de retornar uma resposta em uma quantidade de tempo razoável.

DEFINIÇÃO 15.3. *Analogamente, a classe NP é definida como a classe de todas as linguagens que podem ser decididas por uma máquina de Turing* não-determinística *polinomialmente limitada.*

Problemas em NP tem um perfil de "adivinhar e verificar": o não-determinismo é utilizado para adivinhar, dentro de um espaço de busca finito, uma "testemunha" ou "certificado" de que uma dada entrada pertence à linguagem, que depois é verificada deterministicamente em tempo polinomial. Caso exista uma tal testemunha, uma das computações da máquina de Turing irá parar no estado de aceitação (a computação que escolhe a testemunha correta), o que significa que a máquina de Turing aceita a entrada. Caso não haja nenhuma testemunha para a entrada, então todas as computações irão parar no estado de rejeição e a máquina de Turing rejeita a entrada. Para que a linguagem pertença a NP é necessário que a verificação de que a testemunha é correta para a entrada em questão possa ser feita em tempo polinomial.

Em outras palavras, um problema em NP pode não possuir um algoritmo eficiente para calcular sua solução, mas ele possui um algoritmo eficiente para verificar se uma solução efetivamente satisfaz o problema. Desta forma, em NP, resolver o problema pode ser difícil, mas verificar se uma solução dada é correta pode ser feito de forma eficiente.

EXEMPLO 15.4. Como exemplo de linguagem em NP, podemos citar a linguagem formada pela codificação dos números compostos. Uma testemunha ou certificado neste caso é um número maior do que 1 e menor do que a entrada que seja um fator do número codificado na entrada. Para verificar se um dado número é uma testemunha correta de que a entrada pertence à linguagem, basta dividir a entrada por este número e verificar se o resto da divisão é igual a zero. Se for, este número é um fator da entrada maior do que 1 e menor do que a entrada. A existência de tal fator estabelece que o número da entrada é composto. Desta forma, esta linguagem está em NP.

EXEMPLO 15.5. Outro exemplo de linguagem em NP é a linguagem formada pela codificação dos grafos hamiltonianos. Não se conhece, até o momento, nenhum algoritmo eficiente (nenhum algoritmo que seja executado em tempo polinomial no tamanho da entrada) para determinar se um grafo é hamiltoniano. Entretanto, dado um certificado de que o grafo é hamiltoniano, na forma de um caminho supostamente hamiltoniano neste grafo, podemos verificar de forma eficiente se este certificado é válido ou não para mostrar que o grafo é hamiltoniano.

TEOREMA 15.6. *$P \subseteq NP$.*

DEMONSTRAÇÃO. Este resultado segue diretamente do fato de que toda Máquina de Turing determinística pode ser considerada como um caso particular de uma Máquina de Turing não-determinística.

DEFINIÇÃO 15.7. *Uma máquina de Turing é dita* exponencialmente limitada *se há uma constante inteira $c > 1$ e uma função polinomial $p(n)$ tal que, para toda entrada $w \in \Sigma_0^*$, qualquer computação a partir do estado inicial da máquina sempre alcança um estado de parada em, no máximo, $c^{p(|w|)}$ passos.*

DEFINIÇÃO 15.8. *A classe EXPTIME é definida como a classe de todas as linguagens que podem ser decididas por uma máquina de Turing* determinística *exponencialmente limitada.*

DEFINIÇÃO 15.9. *A classe NEXPTIME é definida como a classe de todas as linguagens que podem ser decididas por uma máquina de Turing* não-determinística *exponencialmente limitada.*

Analogamente à classe NP, problemas em NEXPTIME também tem um perfil de “adivinhar e verificar”. No entanto, no caso de NEXPTIME, a verificação de que a testemunha é correta para a entrada em questão deve ser feita em tempo exponencial.

TEOREMA 15.10. *$EXPTIME \subseteq NEXPTIME$.*

DEMONSTRAÇÃO. A prova deste resultado é inteiramente análoga à prova de que P $\subseteq$ NP.

TEOREMA 15.11. *NP $\subseteq$ EXPTIME.*

DEMONSTRAÇÃO. Como vimos anteriormente, uma Máquina de Turing não-determinística pode ser transformada em uma Máquina de Turing determinística. Entretanto, esta transformação tem um preço. A Máquina de Turing determinística pode precisar executar um número exponencialmente maior de passos do que a Máquina de Turing não-determinística original. Assim, ao transformarmos uma Máquina de Turing não-determinística polinomialmente limitada em uma Máquina de Turing determinística, devido a este aumento exponencial do número de passos executado pela máquina, poderemos obter uma Máquina de Turing determinística exponencialmente limitada. Logo, NP $\subseteq$ EXPTIME.

Temos então a seguinte hierarquia entre as classes de complexidade estudadas até o momento:

$$\text{P} \subseteq \text{NP} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME}.$$

2. Complexidade de Espaço (Memória)

Na seção anterior, analisamos a quantidade de passos que uma Máquina de Turing executa, isto é, o tempo de execução dos algoritmos. Nesta seção, vamos analisar os requisitos de espaço, isto é, memória dos algoritmos. Para isto, vamos observar a quantidade de casas distintas da fita que o cabeçote da Máquina de Turing visita durante a sua computação.

DEFINIÇÃO 15.12. *Uma máquina de Turing é dita* espacialmente polinomialmente limitada *se há um polinômio $p(n)$ tal que, para toda entrada $w \in \Sigma_0^*$, qualquer computação a partir do estado inicial da máquina sempre alcança um estado de parada tendo visitado, no máximo, $p(|w|)$ casas distintas da fita da máquina.*

DEFINIÇÃO 15.13. *A classe PSPACE é definida como a classe de todas as linguagens que podem ser decididas por uma máquina de Turing* determinística *espacialmente polinomialmente limitada.*

DEFINIÇÃO 15.14. *A classe NPSPACE é definida como a classe de todas as linguagens que podem ser decididas por uma máquina de Turing* não-determinística *espacialmente polinomialmente limitada.*

Da mesma maneira que nos casos anteriores em que P $\subseteq$ NP e EXPTIME $\subseteq$ NEXPTIME, temos que PSPACE $\subseteq$ NPSPACE. Entretanto, enquanto nos casos de P e NP e de EXPTIME e NEXPTIME não há nenhum resultado conclusivo que nos diga se as inclusões de P em NP e de EXPTIME em NEXPTIME são próprias ou não, conforme discutiremos mais adiante, no caso das classes PSPACE e NPSPACE tal resultado conclusivo existe. Ele é conhecido como Teorema de Savitch.

TEOREMA 15.15 (Teorema de Savitch). *PSPACE $=$ NPSPACE.*

O que o teorema acima nos diz é que, enquanto a transformação de uma Máquina de Turing não-determinística em uma Máquina de Turing determinística pode acarretar em um aumento exponencial do número de passos que a máquina precisará executar, este aumento exponencial não ocorre com relação à quantidade de memória (quantidade de fita) que a máquina irá utilizar. Assim, a quantidade de fita que a máquina determinística irá utilizar é apenas polinomialmente maior do que a quantidade de fita utilizada pela máquina não-determinística original.

TEOREMA 15.16. *P $\subseteq$ PSPACE.*

DEMONSTRAÇÃO. A cada passo da computação, uma Máquina de Turing visita no máximo uma casa nova (uma casa que ainda não foi visitada) da fita. Então, se a máquina é *polinomialmente limitada* e executa no máximo $f(|w|)$ passos, então ela visita no máximo $f(|w|) + 1$ casas distintas da fita, o que significa que ela também é *espacialmente polinomialmente limitada*. A partir desta implicação, temos que P $\subseteq$ PSPACE.

Com o mesmo argumento acima, podemos concluir também que NP $\subseteq$ NPSPACE. Mas como NPSPACE $=$ PSPACE, pelo Teorema de Savitch, temos então o resultado a seguir.

TEOREMA 15.17. *NP $\subseteq$ PSPACE.*

Vamos agora mostrar outra inclusão relacionada à classe PSPACE.

TEOREMA 15.18. *PSPACE $\subseteq$ EXPTIME.*

DEMONSTRAÇÃO. Seja $L \in$ PSPACE. Então, existe uma Máquina de Turing determinística $M = (\Sigma_0, \Sigma, Q, q_0, F, \delta)$ que decide L visitando no máximo $f(|w|)$ casas distintas da fita, para todo $w \in \Sigma_0^*$ (máquina espacialmente polinomialmente limitada).

Vamos analisar quantas configurações diferentes esta máquina pode apresentar durante a computação com uma dada entrada $w \in \Sigma_0^*$. Suponha que $|Q| = n$ e $|\Sigma| = m$. Como a máquina é espacialmente polinomialmente limitado, o cabeçote está restrito a uma região da fita formada por $f(|w|)$ casas. Temos então que, em um dado momento, a máquina pode estar em um de seus n estados, com o seu cabeçote posicionado em uma das $f(|w|)$ casas da fita e com um de m possíveis símbolos do alfabeto em cada uma das $f(|w|)$ casas da fita. Assim, existe um total de

$$n \times f(|w|) \times m^{f(|w|)}$$

possíveis configurações em que a máquina pode estar ao longo da computação.

A máquina não pode repetir uma configuração durante a sua computação, pois isto a faria entrar em um loop, já que ela é determinística. Mas a máquina não pode entrar em loop, já que ela é um decisor. Assim, esta máquina executa no máximo $n \times f(|w|) \times m^{f(|w|)}$ passos durante a computação com a entrada w. Portanto, a máquina é exponencialmente limitada, o que significa que $L \in$ EXPTIME.

Novamente, apresentamos a hierarquia entre as classes de complexidade estudadas até o momento:

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME}.$$

3. Co-Classes

DEFINIÇÃO 15.19. *Se $\mathcal{C}$ é uma classe de complexidade, definimos a classe co$-\mathcal{C}$, chamada de* complemento *da classe $\mathcal{C}$, como a classe de todas as linguagens cujos complementos pertencem a $\mathcal{C}$. Em outras palavras, $L \in co-\mathcal{C}$ se e somente se $\overline{L} \in \mathcal{C}$.*

Se L é decidida por uma Máquina de Turing determinística, então, pela simples inversão dos estados de aceitação e rejeição, podemos obter uma Máquina de Turing determinística que decide $\overline{L}$ e possui as mesmas características no que diz respeito à quantidade de tempo e memória utilizados em relação à máquina original. Desta forma, para todas as classes definidas a partir de máquinas de Turing determinísticas, temos que co-$\mathcal{C} = \mathcal{C}$. Logo, temos que:

(1) P = co-P,

(2) PSPACE = co-PSPACE e

(3) EXPTIME = co-EXPTIME.

Por outro lado, pelo Teorema de Savitch, que vimos anteriormente, temos que PSPACE = NPSPACE. Logo, o complemento das duas classes também deve ser igual, isto é, temos que co-PSPACE = co-NPSPACE. Mas como co-PSPACE = PSPACE, então podemos concluir que co-NPSPACE = PSPACE.

Permanecem como questões em aberto até o presente momento se co-NP é igual ou diferente de NP e se co-NEXPTIME é igual ou diferente de NEXPTIME. Acredita-se fortemente que a resposta para ambas as perguntas seja que tais conjuntos são diferentes.

O lema a seguir nos auxilia a situar estas novas classes na hierarquia de classes de complexidade.

LEMA 15.20. *Sejam C_1 e C_2 classes de complexidade. Se $C_1 \subseteq C_2$, então co-$C_1 \subseteq$ co-C_2.*

DEMONSTRAÇÃO. Seja $L \in$ co-C_1. Então, $\overline{L} \in C_1$. Como $C_1 \subseteq C_2$, temos que $\overline{L} \in C_2$. Isto, por sua vez, nos permite concluir que $L \in$ co-C_2. Portanto, co-$C_1 \subseteq$ co-C_2.

Dado que P $\subseteq$ NP, temos, pelo lema acima, que co-P $\subseteq$ co-NP. Mas como co-P $=$ P, então P $\subseteq$ co-NP, o que significa que P $\subseteq$ NP $\cap$ co-NP. Por um raciocínio inteiramente análogo, temos também que EXPTIME $\subseteq$ NEXPTIME $\cap$ co-NEXPTIME.

Além disso, dado que NP $\subseteq$ PSPACE, temos, novamente pelo lema acima, que co-NP $\subseteq$ co-PSPACE. Mas como co-PSPACE $=$ PSPACE, então co-NP $\subseteq$ PSPACE.

Podemos então reunir todos os resultados que estudamos sobre as diversas classes de complexidade na seguinte hierarquia:

Acredita-se fortemente que todas estas inclusões apresentadas na hierarquia acima são próprias (em particular, acredita-se que P $\neq$ NP, isto é, que existem problemas na classe NP que são inerentemente intratáveis no caso geral) e que NP $\neq$ co-NP e NEXPTIME $\neq$ co-NEXPTIME, mas nenhum destes resultados foi provado até hoje e todos permanecem como problemas em aberto. O único resultado já provado é que P $\neq$ EXPTIME.

4. Redução Polinomial entre Linguagens

De forma análoga ao nosso uso de reduções entre linguagens para mostrar que uma linguagem é indecidível a partir do conhecimento de que outra linguagem é indecidível, podemos utilizar uma noção refinada deste conceito para inserir linguagens na hierarquia das classes de complexidade a partir do conhecimento do posicionamento de outras linguagens nesta hierarquia.

DEFINIÇÃO 15.21. *Dizemos que uma função* $f : \Sigma_0^* \to \Sigma_0^*$ *é* polinomialmente computável *se existe uma máquina de Turing determinística polinomialmente limitada que a computa.*

DEFINIÇÃO 15.22. *Considere duas linguagens* $L_1, L_2 \subseteq \Sigma_0^*$. *Dizemos que existe uma* redução polinomial *de* L_1 *para* L_2*, denotada por* $L_1 \leq_P L_2$*, se existe uma função polinomialmente computável* $f : \Sigma^* \to \Sigma^*$ *tal que* $w \in L_1$ *se e somente se* $f(w) \in L_2$.

Seja $\mathcal{C}$ uma classe de complexidade. Suponha que $L_2 \in \mathcal{C}$ e que exista uma redução polinomial de L_1 para L_2 ($L_1 \leq_P L_2$). Então, podemos decidir se $w \in L_1$ da seguinte forma:

(1) Como existe uma redução polinomial de L_1 para L_2, existe uma função polinomialmente computável f tal que $w \in L_1$ se e somente se $f(w) \in L_2$. Por sua vez, como f é polinomialmente computável, ela pode ser computada por uma Máquina de Turing determinística polinomialmente limitada.
(2) Como $L_2 \in \mathcal{C}$, existe uma Máquina de Turing que atende aos requisitos de tempo e espaço impostos pela classe $\mathcal{C}$ que nos permite decidir se $f(w) \in L_2$.
(3) Podemos compor as Máquinas de Turing dos dois itens anteriores para obter uma Máquina de Turing que decide se $w \in L_1$. A máquina obtida por esta composição também atende aos requisitos de tempo e espaço impostos pela classe $\mathcal{C}$, uma vez que a máquina do item 1 apresenta o menor grau de complexidade dentro da hierarquia (máquina determinística polinomialmente limitada).

Podemos concluir então que, se $L_2 \in \mathcal{C}$ e $L_1 \leq_P L_2$, então $L_1 \in \mathcal{C}$. Desta forma, podemos dizer que, se existe uma redução polinomial de L_1 para L_2, então L_1 é "não mais complexa" do que L_2, ou, em outras palavras, L_2 é "ao menos tão complexa" quanto L_1.

DEFINIÇÃO 15.23. *Seja $\mathcal{C}$ uma classe de complexidade e L uma linguagem. Dizemos que L é* $\mathcal{C}$-Difícil *se, para toda linguagem $L' \in \mathcal{C}$, existe uma redução polinomial de L' para L. Desta forma, L ser $\mathcal{C}$-Difícil significa que L é ao menos tão complexa quanto* qualquer *linguagem de $\mathcal{C}$.*

DEFINIÇÃO 15.24. *Seja $\mathcal{C}$ uma classe de complexidade e L uma linguagem. Dizemos que L é* $\mathcal{C}$-Completa *se:*

(1) L é $\mathcal{C}$-Difícil e

(2) $L \in \mathcal{C}$.

Desta forma, uma linguagem $\mathcal{C}$-Completa é uma linguagem de $\mathcal{C}$ que é ao menos tão complexa quanto qualquer linguagem de $\mathcal{C}$, o que nos permite pensar nas linguagens $\mathcal{C}$-Completas como as linguagens mais complexas dentro da classe $\mathcal{C}$, ou o "limite superior" de complexidade dentro da classe.

Se sabemos que uma linguagem L é $\mathcal{C}$-Completa, podemos provar que uma outra linguagem L' também é $\mathcal{C}$-Completa mostrando que $L' \in \mathcal{C}$ e que existe uma redução polinomial f de L para L'. Como L é $\mathcal{C}$-Completa, existe uma redução polinomial de qualquer linguagem em $\mathcal{C}$ para L. Logo, compondo estas reduções polinomiais com f, temos reduções polinomiais de qualquer linguagem em $\mathcal{C}$ para L', o que significa que L' é $\mathcal{C}$-Completa.

Acredita-se fortemente que P $\neq$ NP, mas isto não foi provado. Um indício forte da validade desta desigualdade vem do estudo dos problemas NP-Completos.

TEOREMA 15.25. *$P = NP$ se e somente se existe uma linguagem L NP-Completa tal que $L \in P$.*

DEMONSTRAÇÃO. ($\Rightarrow$) Suponha que P $=$ NP e que L é uma linguagem NP-Completa. Queremos mostrar que $L \in$ P. Pela condição (2) da definição 15.24, se L é NP-Completa, então $L \in$ NP. Mas como estamos assumindo a hipótese de que NP $=$ P, então $L \in$ P, como queríamos mostrar.

($\Leftarrow$) Seja L uma linguagem NP-Completa tal que $L \in$ P. Queremos mostrar que P $=$ NP. Como $L \in$ P, L pode ser decidida por uma máquina de Turing

determinística polinomialmente limitada M. Por outro lado, pela condição (1) da definição 15.24, se L é NP-Completa, então L é NP-Difícil. Por sua vez, se L é NP-Difícil, então, para toda linguagem $L' \in$ NP, existe uma redução polinomial de L' para L, que pode ser computada por uma Máquina de Turing determinística polinomialmente limitada M'. Mas então a composição das Máquinas de Turing M' e M também resulta em uma Máquina de Turing determinística polinomialmente limitada. Esta máquina decide L', o que significa que $L' \in P$. Como esta argumentação vale para qualquer $L' \in NP$, temos então que $P = NP$.

O primeiro problema que foi determinado como sendo NP-Completo foi o problema de determinar se uma dada fórmula da lógica proposicional é satisfatível, no teorema de Cook. A partir de então, através do uso de reduções polinomiais, muitos outros problemas foram caracterizados como NP-Completos. Até hoje, nenhum algoritmo polinomial foi descoberto para nenhum dos problemas NP-Completos, o que nos faz acreditar fortemente que os problemas NP-Completos não estão em P, o que é equivalente a P $\neq$ NP.

Outro resultado relacionado aos problemas NP-Completos diz respeito à igualdade ou diferença entre as classes NP e co-NP.

LEMA 15.26. *Seja C uma classe de complexidade. Se $C \subseteq$ co-C, então $C =$ co-C.*

DEMONSTRAÇÃO. De acordo com o lema 15.20, se $C \subseteq$ co-C, então co-$C \subseteq$ co-(co-C) $= C$. Assim, a partir de $C \subseteq$ co-C e co-$C \subseteq C$, podemos concluir que $C =$ co-C.

TEOREMA 15.27. *NP $=$ co-NP se e somente se existe uma linguagem L NP-Completa tal que $L \in$ co-NP.*

DEMONSTRAÇÃO. ($\Rightarrow$) Suponha que NP $=$ co-NP e que L é uma linguagem NP-Completa. Pela condição (2) da definição 15.24, se L é NP-Completa, então $L \in$ NP. Mas como estamos assumindo a hipótese de que NP $=$ co-NP, então $L \in$ co-NP, como queríamos mostrar.

($\Leftarrow$) Seja L uma linguagem NP-Completa tal que $L \in$ co-NP. Pela condição (1) da definição 15.24, se L é NP-Completa, então L é NP-Difícil. Por sua vez, se L é NP-Difícil, então, para toda linguagem $L' \in$ NP, existe uma redução polinomial de L' para L, que pode ser computada por uma Máquina de Turing determinística polinomialmente limitada M'. Esta redução é dada por uma função polinomialmente computável f tal que $w \in L' \Leftrightarrow f(w) \in L$. Esta função f também oferece então uma redução polinomial entre $\overline{L'}$ e $\overline{L}$, da seguinte forma:

$$w \in \overline{L'} \Leftrightarrow w \notin L' \Leftrightarrow f(w) \notin L \Leftrightarrow f(w) \in \overline{L}.$$

Como temos, por hipótese, que $L \in$ co-NP, então $\overline{L} \in$ NP. Assim, existe uma Máquina de Turing não-determinística polinomialmente limitada que decide $\overline{L}$. Mas então a composição das Máquinas de Turing M' e M também resulta em uma máquina de Turing não-determinística polinomialmente limitada. Esta máquina decide $\overline{L'}$, o que significa que $\overline{L'} \in$ NP. Desta forma, podemos concluir que $L' \in$ co-NP. Como esta argumentação vale para qualquer $L' \in$ NP, temos então a seguinte conclusão: se $L' \in$ NP, então $L' \in$ co-NP. Isto, por sua vez, significa que NP $\subseteq$ co-NP. Este resultado, em conjunto com o lema acima, nos permite concluir que NP $=$ co-NP.

5. Exercícios

(1) Justifique cada uma das inclusões abaixo:
   (a) P $\subseteq$ NP
   (b) NP $\subseteq$ EXPTIME
   (c) P $\subseteq$ PSPACE
   (d) PSPACE $\subseteq$ EXPTIME

(2) Explique por que P $= co -$ P, PSPACE $= co -$ PSPACE e EXPTIME $= co -$ EXPTIME.

(3) Seja $\mathcal{C}$ uma classe de complexidade. Explique o que é uma linguagem $\mathcal{C}$-Difícil e o que é uma linguagem $\mathcal{C}$-Completa.

(4) Mostre que P $=$ NP se e somente se existe uma linguagem NP-Completa L tal que $L \in$ P.

(5) Mostre que NP $=$ co-NP se e somente se existe uma linguagem L NP-Completa tal que $L \in$ co-NP.

Referências Bibliográficas

[1] M. A. Harrison, *Introduction to formal language theory*, Addison-Wesley (1978).

[2] J. E. Hopcroft, J. D. Ullman e R. Motwani, *Introdução à Teoria de Autômatos, Linguagens e Computação*, Tradução da 2a edição americana, Campus/Elsevier (2002).

[3] H. R. Lewis e C. H. Papadimitriou, *Elementos de Teoria da Computação*, 2a edição revisada, Bookman (2004)

[4] M. Sipser, *Introdução à Teoria da Computação*, Tradução da 2a edição norte-americana, Cengage Learning (2007).