Linguagens Formais e Autômatos

notas de aula – 2019.05.05.20.43

Jerônimo C. Pellegrini

id: 4e2626ca5ee2dc5536045a55e267b3bb2211b939

Sumário

Apresentação

O objetivo é um texto que cubra os tópicos usualmente tratados em um curso de Linguagens Formais e Autômatos.

Nomenclatura

Neste texto usamos marcadores para final de definições (⧫), exemplos (◄) e demonstrações (□).

$(\alpha)^m$ concatenação iterada de palavra, página 2

Δ relação de transição em autômato não-determinístico (notação usual), página 19

δ função de transição (notação usual), página 15

$\Omega(g)$ cresce assintoticamente mais que g, página 109

$\Rightarrow$ passo de derivação em gramática, página 7

Σ alfabeto (notação usual), página 1

Σ alfabeto (notação usual), página 15

Σ_ε alfabeto Σ aumentado com a cadeia vazia, página 1

$\Theta(g)$ $O(g)$ e também $\Omega(g)$, página 109

ε palavra vazia, página 1

$\vdash$ encadeamento de configurações de autômato com pilha durante reconhecimento de palavra, página 48

$\vdash$ encadeamento de configurações de autômato finito durante reconhecimento de palavra, página 17

$\vdash$ encadeamento de configurações de máquina de Turing durante reconhecimento de palavra, página 74

$|\alpha|$ comprimento da palavra α, página 2

a^*, A^* fecho de Kleene, página 4

a^+, A^+ fecho de Kleene sem palavra vazia, página 4

$E(q)$ estados alcançáveis a partir de q com transições vazias, página 22

F conjunto de estados finais (notação usual), página 15

$L(A)$ linguagem do autômato com pilha A, parando em estado final, página 48

$O(g)$ cresce assintoticamente menos que g, página 109

Q conjunto de estados (notação usual), página 15

q_0 estado inicial (notação usual), página 15

$V(A)$ linguagem do autômato com pilha A, parando com pilha vazia, página 48

Capítulo 1

Introdução

1.1 Linguagens

Definição 1.1 (alfabeto)**.** Um *alfabeto* é um conjunto finito de *símbolos*. ♦

Exemplo 1.2. $\{0, 1\}$ é o alfabeto binário, contendo os dígitos zero e um. ◄

Exemplo 1.3. $\{a, b, c, \ldots, z\}$ é o alfabeto romano usual. ◄

Exemplo 1.4. $\{a, b, c\}$ é um alfabeto contendo somente três símbolos. ◄

Exemplo 1.5. Podemos usar quaisquer conjunto de símbolos que queiramos como alfabeto. Um exemplo é $\{\diamondsuit, \clubsuit, \heartsuit, \spadesuit\}$. ◄

Definição 1.6 (cadeia, palavra)**.** Uma *cadeia* ou *palavra* sobre um alfabeto Σ é uma sequência finita (possivelmente vazia) de símbolos de Σ.

Denotamos por ε a palavra vazia. Se um alfabeto Σ não contém a palavra vazia, podemos denotar $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$. ♦

Embora seja comum denotar o alfabeto usado por Σ, esta é apenas uma convenção.

Exemplo 1.7. O alfabeto de bits é $\{0, 1\}$. Alguns exemplos de cadeias de bits são 0, 1, 000, 0110010101. ◄

Exemplo 1.8. Sobre o alfabeto $\{a, b, c, d\}$ são cadeias ε, a, b, aaadddcb, abbc. ◄

Exemplo 1.9. Sobre o alfabeto $\{\diamondsuit, \clubsuit, \heartsuit, \spadesuit\}$ (do Exemplo 1.5) podemos formar, por exemplo, as cadeias $\diamondsuit\heartsuit\diamondsuit\heartsuit$, $\diamondsuit\clubsuit\clubsuit$ e $\clubsuit\clubsuit\clubsuit$. ◄

Exemplo 1.10. Seja $\Sigma = \{a, b, c\}$. Então $\Sigma_\varepsilon = \{\varepsilon, a, b, c\}$. ◄

Exemplo 1.11. Seja $\Sigma = \{\text{um}, \text{dois}, \text{três}, \text{quatro}\}$. Embora possa parecer que este não deveria ser um alfabeto, ele é. Podemos chamar de "símbolo" a entidade que quisermos. Uma palavra sobre este alfabeto é "um, um, quatro, dois", por exemplo. ◀

Definição 1.12 (comprimento de palavra)**.** O *comprimento* de uma palavra é a quantidade de símbolos nela. Denotamos por $|\alpha|$ o comprimento da palavra α. ♦

Exemplo 1.13. Os comprimentos de algumas cadeias são dados a seguir.

$$|\varepsilon| = 0$$
$$|a| = 1$$
$$|b| = 1$$
$$|aaa| = 3$$
$$|baabb| = 4$$ ◀

Exemplo 1.14. Note que no exemplo 1.11, onde definimos palavras sobre o alfabeto $\Sigma = \{\text{um}, \text{dois}, \text{três}, \text{quatro}\}$, temos $|\textit{um}, \textit{um}, \textit{quatro}, \textit{dois}| = 4$, porque a cadeia (palavra) contém quatro símbolos do alfabeto. ◀

Definição 1.15 (concatenação de palavras)**.** Sejam $\alpha = \alpha_1\alpha_2\ldots\alpha_k$ e $\beta = \beta_1\beta_2\ldots\beta_n$ duas palavras. Então $\alpha\beta = \alpha_1\alpha_2\ldots\alpha_k\beta_1\beta_2\ldots\beta_n$ é a *concatenação* das palavras α e β.

Denotamos por $(\alpha)^n$ a concatenação de n cópias da palavra α. Para um símbolo isolado s, não usamos os parênteses, denotando s^n. Quando usamos o mesmo contador mais de uma vez na mesma expressão, significa que as quantidades devem ser iguais: em "a^nb^n", as quantidade de a's e b's são iguais; em a^kb^n, não. ♦

Evidentemente, $\varepsilon\alpha = \alpha\varepsilon = \alpha$ para qualquer palavra α.

Exemplo 1.16. A concatenação de abb com ca é $abbca$. ◀

Exemplo 1.17. Denotamos por $a^nb^nc^n$ as palavras contendo sequências de a, b e c, nesta ordem, sendo que cada palavra tem exatamente a mesma quantidade de a's, b's e c's: $\{\varepsilon, abc, aabbcc, aaabbbccc, \ldots\}$. ◀

Exemplo 1.18. $L = a^nb^kc^nd^k$ é o conjunto das sequências de n vezes a's, k vezes b's, n vezes c's e k vezes d's, nesta ordem:

$$\{\varepsilon, ac, bd, abcd, aabccd, abbcdd, aabbccdd, \ldots\}.$$

◀

Definição 1.19 (linguagem)**.** Uma *linguagem* é um conjunto, possivelmente vazio, de palavras. ♦

Exemplo 1.20. Conjuntos finitos de símbolos como $\emptyset, \{a\}, \{0\}, \{abcdef\}$ são linguagens. ◄

Exemplo 1.21. O conjunto de palavras representado por $a^n b^n c^n$ (no Exemplo 1.17) é uma linguagem, assim como o conjunto $a^n b^k c^n d^k$ do Exemplo 1.18. ◄

Exemplo 1.22. O conjunto de todas as sequências binárias representando números divisíveis por quatro, $\{0, 100, 1000, 1100, 10000, 10100, 11000, 11100, \ldots\}$, é uma linguagem. ◄

Exemplo 1.23. O conjunto de todas as palavras em um dicionário da Língua Portuguesa pode ser usado como alfabeto. Aqui é necessário ter em mente que há uma troca de nomenclatura: cada "palavra" do dicionário será, na linguagem formal que definirmos, um "símbolo"; e cada "sequência de palavras" do dicionário será uma "palavra" na linguagem construída.

Assim, sobre o alfabeto palavras da Língua Portuguesa, o conjunto de artigos definidos seguidos de substantivos é uma linguagem. Algumas palavras desta linguagem são

- o sapato
- a cebola
- a carro

Note que cada item é, para nós, *uma palavra*. No primeiro, a palavra "o sapato" tem dois símbolos, "o" e "sapato".

A última palavra pode parecer incorreta, mas essa incorretude existe somente em outro tipo de análise. Da maneira como definimos, está correta (faz parte da linguagem), porque determinamos que "artigos definidos" seguidos de "substantivos", *sem restringir gênero*, são parte da linguagem. ◄

Exemplo 1.24. Usando o alfabeto do Exemplo 1.11 ($\Sigma = \{\text{um}, \text{dois}, \text{três}, \text{quatro}\}$), podemos construir a linguagem das sequências de palavras um, dois, três, quatro, repetidas e terminando com "dois" ou com "quatro".

Pertencem a esta linguagem as cadeias

- um, um, dois
- dois, um, quatro
- dois

As seguintes palavras de Σ^* *não* pertencem à linguagem que definimos, porque não terminam em "dois" ou em "quatro".

- um

- dois, três
- dois, dois, três, um

◀

Definição 1.25 (união de linguagens)**.** Se A e B são linguagens, então $A \cup B$ é a união das duas linguagens, contendo todas as palavras presentes nelas. ⧫

Exemplo 1.26. Seja $\Sigma = \{0, 1\}$. Seja A a linguagem, sobre Σ, dos números binários ímpares (ou seja, aqueles terminando em 1). Seja B a linguagem dos números binários divisíveis por quatro. Então $A \cup B$ é a linguagem contendo as cadeias binárias que representam ímpares e *também* as cadeias que representam os divisíveis por quatro. Ou seja, $A \cup B$ contém as sequências binárias que terminam em um e as que terminam em dois zeros, mas *não* as que terminam em "10" (que são os pares não divisíveis por quatro). ◀

Exemplo 1.27. Se P a linguagem contendo as palavras da língua Portuguesa, e F é a linguagem contendo as palavras da língua Francesa, então $P \cup F$ é a linguagem das palavras existentes nas duas línguas. ◀

Definição 1.28 (concatenação de linguagens)**.** A *concatenação* de duas linguagens A e B, denotado AB, é o conjunto das concatenações de palavras, sendo a primeira de A e a segunda de B:

$$AB = \{ab \ : \ a \in A, \ b \in B\}.$$ ⧫

Definição 1.29 (fecho de Kleene)**.** Se A é um alfabeto, palavra ou linguagem, A^* é seu *fecho de Kleene,* ou simplesmente *fecho.* Pertencem ao fecho de Kleene todas as concatenações iteradas dos elementos de A, inclusive a palavra vazia.

Denotamos por A^+ o fecho transitivo de A, idêntico ao fecho de Kleene, exceto por não incluir a palavra vazia. ⧫

Exemplo 1.30. Se $\Sigma = \{a, b, c\}$, então

$$\Sigma^* = \{\varepsilon, a, b, c, aa, bb, cc, ab, ac, bc.ba, ca., aaa, bbb, ccc, aab, aac, bba, bbc, \ldots\}$$
$$\Sigma^+ = \{a, b, c, aa, bb, cc, ab, ac, bc.ba, ca., aaa, bbb, ccc, aab, aac, bba, bbc, \ldots\}$$

Se $L = \{abc, def, g\}$, então

$$L^* = \{\varepsilon, abc, def, g, abcdef, defabc, abcg, gabc, defg, gdef, \ldots\}$$
$$L^+ = \{abc, def, g, abcdef, defabc, abcg, gabc, defg, gdef, \ldots\}$$ ◀

Exemplo 1.31. Seja $\Sigma = \{0, 1\}$. Então

- $L = \{00, 01, 10, 11\}$ é uma linguagem finita com quatro palavras;

- $L^2 = \{0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1100, 1101, 1110, 1111\}$ é a linguagem das sequências binárias de comprimento quatro;

- $L \cup L^2$ é a linguagem das sequências binárias de comprimento dois ou quatro;

- L^* é a linguagem de todas as sequências binárias de comprimento par.

◀

1.2 Gramáticas gerativas

Usamos *gramáticas* como uma forma finita de especificar linguagens possivelmente infinitas. Uma gramática é a definição recursiva de uma linguagem, usando a concatenação de palavras na recursão. Esta definição recursiva é composta de *regras de produção*. Cada regra de produção é da forma $\alpha \to \beta$, indicando que a frase α pode ser trocada por β.

Exemplo 1.32. A seguir temos uma gramática que representa um pequeno conjunto de frases e Língua Portuguesa.

$$\begin{aligned}
\textbf{frase} &::= \textbf{sujeito}\ \ \textbf{predicado} \\
\textbf{sujeito} &::= \text{João} \mid \text{Paulo} \mid \text{Luíza} \\
\textbf{verbo} &::= \text{fez} \mid \text{comprará} \mid \text{come} \\
\textbf{predicado} &::= \textbf{verbo}\ \ \textbf{objeto} \\
\textbf{objeto} &::= \textbf{artigo}\ \ \textbf{substantivo} \\
\textbf{artigo} &::= \text{o} \mid \text{um} \\
\textbf{substantivo} &::= \text{montanha} \mid \text{peixe}
\end{aligned}$$

Os símbolos em negrito são variáveis auxiliares, que não fazem parte da linguagem – são chamados de *não-terminais*. Os outros são palavras da linguagem, e chamados de *terminais*.

A seguir usamos a gramática para gerar uma frase. Em cada linha, substituímos uma variável usando alguma regra de produção, até chegar a uma

palavra sem variáveis.

frase
sujeito predicado
sujeito verbo objeto
sujeito come **objeto**
Paulo come **objeto**
Paulo come **artigo substantivo**
Paulo come um **substantivo**
Paulo come um peixe ◀

Exemplo 1.33. Podemos também definir gramáticas que especificam a sintaxe de linguagens de programação:

```
  <comando>   ::= <atribuicao> | <condicional> | <repeticao>
 <atribuicao> ::= <id> = <expr>
      <expr>  ::= <id> | <num>
                | <expr> + <expr>
                | <expr> * <expr>
                | ( <expr> )
```

Esta gramática define a sintaxe de "<comando>", que pode ser "<atribuicao>", "<condicional>" ou "<repeticao>". Derivamos, usando esta sintaxe, uma palavra.

```
<comando>  ⇒ <atribuicao>
           ⇒ <id> = <expr>
           ⇒ a = <expr>
           ⇒ a = <expr> + <expr>
           ⇒ a = <num> + <expr>
           ⇒ a = 2 + <expr>
           ⇒ a = 2 + <id>
           ⇒ a = 2 + b
```

◀

Agora definimos rigorosamente o conceito de gramática. A definição com rigor é necessária porque, sem ela, não poderíamos elaborar enunciados claros a respeito das propriedades de gramáticas, nem demonstrar com rigor esses enunciados.

Definição 1.34 (gramática)**.** Uma *gramática* é composta de

- um conjunto T de símbolos terminais (um alfabeto)
- um conjunto Σ de símbolos não-terminais, diferentes dos terminais
- um conjunto de *regras de produção*, da forma

$$\alpha ::= \beta$$

As regras de produção são uma relação $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$

- um *símbolo inicial* $S \in N$, e também podem ser denotadas

$$\alpha \to \beta.$$ ♦

Note que as regras de produção podem transformar qualquer sequência de símbolos (terminais e não-terminais) em outra sequência.

Definição 1.35 (substituição, derivação)**.** Se $\alpha, \beta, \gamma, \delta$ são sequências de símbolos e $\alpha \to \beta$ é uma regra de produção de G, então

$$\gamma\alpha\delta \Rightarrow \gamma\beta\delta$$

é um *passo de derivação*.

Se existe uma sequência de passos de derivação que leve a uma cadeia de símbolos terminais, dizemos que esta é uma *derivação* de uma palavra da gramática; dizemos também que a gramática *gera* aquela palavra. ♦

Exemplo 1.36. Uma gramática simples é mostrada a seguir.

$$\begin{aligned} A &\to Ab \\ A &\to Bc \\ B &\to A \\ B &\to x \end{aligned}$$

Usamos | para agrupar regras, e podemos reescrever a gramática:

$$\begin{aligned} A &\to Ab \mid Bc \\ B &\to A \mid x \end{aligned}$$

Derivamos uma palavra da gramática.

$$\begin{aligned} A &\Rightarrow Ab \\ &\Rightarrow Bcb \\ &\Rightarrow xcb \end{aligned}$$ ◄

Exemplo 1.37. A derivação do Exemplo 1.32 é usualmente denotada da seguinte maneira.

frase $\Rightarrow$ **sujeito predicado** $\Rightarrow$ **sujeito verbo objeto**
$\Rightarrow$ **sujeito** come **objeto** $\Rightarrow$ Paulo come **objeto**
$\Rightarrow$ Paulo come **artigo substantivo** $\Rightarrow$ Paulo come um **substantivo**
$\Rightarrow$ Paulo come um peixe

Denotamos a derivação inteira por

$$\textbf{frase} \Rightarrow^* \text{Paulo come um peixe} \qquad \blacktriangleleft$$

Exemplo 1.38. Uma gramática pode ter regras com mais que apenas um não-terminal à esquerda, como na que segue.

$$\begin{aligned} A &\to BaB && (1)\\ B &\to bbb && (2)\\ B &\to Ab && (3)\\ Aba &\to c && (4) \end{aligned}$$

A seguir usamos esta gramática para derivar uma palavra da linguagem que ela representa.

$$\begin{aligned} A &\Rightarrow BaB && (1)\\ &\Rightarrow Babbb && (2)\\ &\Rightarrow Abababbb && (3)\\ &\Rightarrow cbbb. \end{aligned}$$

Neste texto não usaremos este tipo de gramática. ◀

Definição 1.39 (linguagem de uma gramática)**.** O conjunto de todas as palavras geradas por uma gramática G é a *linguagem de* G. ♦

Exemplo 1.40. A gramática a seguir gera a linguagem de parênteses balanceados.

S ::= (S) | SS | ε ◀

Exemplo 1.41. A seguir está uma gramática.

A ::= aB
A ::= bA
A ::= b
A ::= ε
B ::= bB
B ::= aA

A linguagem desta gramática é a das sequências de as e bs onde o número de as é par. ◀

1.3 "Autômatos" (robôs) e linguagens

Da mesma forma que podemos definir gramáticas que geram (ou permitem verificar) palavras de uma linguagem, podemos imaginar robôs (ou "autômatos", como eram chamados quando foram idealizados) que o façam.

Nesta seção tratamos muito superficialmente de autômatos, sem defini-los formalmente; eles são objeto de estudo mais detalhado no resto do texto.

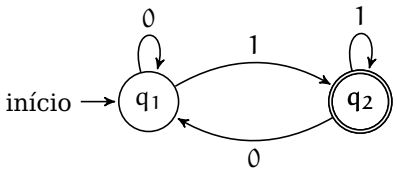
Trataremos de autômatos que recebem uma palavra e verificam se aquela palavra é parte de uma linguagem (a linguagem é fixa para cada autômato).

Os autômatos que usaremos são idéias abstratas de procedimento – semelhante a algoritmos, mas descritos como se fossem máquinas que lêem uma palavra escrita em uma fita.

Autômatos tem um *estado interno*, que os permite lembrar o que ocorreu até o momento; um destes estados é identificado como *estado inicial* (que significa que nada aconteceu ainda)

Um exemplo extremamente simples de autômato é dado a seguir. Este autômato lê uma sequência de símbolos do alfabeto $\Sigma = \{0, 1\}$ e aceita a palavra somente se a sequência representa um número binário ímpar (ou seja, se o último elemento é um). Presumimos que o autômato percebe quando chega o final da palavra.

O autômato tem dois estados: um para representar "último símbolo lido igual a zero" e outro para representar "último símbolo lido igual a um".



Nos Capítulos que seguem formalizaremos o conceito de autômato e de algumas variantes dele. Basicamente, descreveremos um autômato como um conjunto finito de estados Q; um alfabeto Σ de símbolos que podem estar na fita; uma função de transição ("programa") δ, que determina o próximo estado; um estado inicial $q_0 \in Q$; e um conjunto de estados finais $F \subseteq Q$. Como exemplo, o autômato da Figura anterior é descrito como $(Q, \Sigma, \delta, q_1, F)$, onde

$$
\begin{aligned}
Q &= \{q_1, q_2\} \\
\Sigma &= \{0, 1\} \\
\delta(q_1, 0) &= q_1 \\
\delta(q_1, 1) &= q_2 \\
\delta(q_2, 0) &= q_1 \\
\delta(q_2, 1) &= q_2 \\
F &= \{q_2\}
\end{aligned}
$$

Outros tipos de autômato terão descrições diferentes, claro. Um deles terá uma pilha acoplada, por exemplo; outro poderá gravar na fita, além de po-

der voltar a cabeça de leitura e gravação. Os diferentes tipos de autômatos descrevem diferentes classes de linguagem.

1.4 Problemas

O autômato na seção anterior mostra um outro aspecto do estudo de linguagens. Aquele autômato *resolve um problema de decisão*: dado um número (naquele caso representado em base dois, embora não seja necessário), o autômato lê sua descrição na fita e somente aceita o número se ele for ímpar. Isso significa que o autômato resolve o problema de identificar números ímpares.

A linguagem do problema ÍMPARES consiste do conjunto de sequências binárias que representam números ímpares. O autômato "decide" este problema ao decidir se uma sequência de símbolos pertence à linguagem ou não.

Autômatos mais complexos do que este são usados como modelos para algoritmos. Há também outras maneiras de formalizar o conceito de computação: duas delas são a Teoria das Funções Recursivas e o λ-Cálculo.

Há problemas que são *indecidíveis* – porque demonstramos que não existe autômato que possa decidi-los.

Dentre os problemas decidíveis, alguns problemas podem ser decididos em tempo "razoável" e outros não – a definição do que "razoável" significa neste contexto é dada no estudo da Complexidade de Tempo de algoritmos.

Exercícios

Ex. 1 — Derive cinco palavras usando cada uma das gramáticas a seguir.

a)

$$
\begin{aligned}
S &::= aAa \mid bBb \mid C \\
A &::= 1A \mid 0 \\
B &::= \varepsilon \mid b \mid bBb \\
C &::= cC \mid d
\end{aligned}
$$

b)

$$
\begin{aligned}
S &::= aAa \mid cBc \mid cCc \\
aA &::= 1A \mid 0 \\
A &::= 0A \mid 1 \\
B &::= \varepsilon \mid b \mid bBb \\
C &::= 0C \mid \varepsilon \\
00C &::= 00A
\end{aligned}
$$

Ex. 2 — Crie duas gramáticas diferentes para uma mesma linguagem finita.

Ex. 3 — Crie duas gramáticas diferentes para uma mesma linguagem infinita.

Ex. 4 — Crie uma gramática sobre o alfabeto $\Sigma = \{a, b, c\}$, com pelo menos três regras de produção, que gere uma linguagem infinita tendo somente palavras de comprimento par.

Ex. 5 — Modifique o autômato que verifica números ímpares, de forma que ele trabalhe com números na base dez.

Capítulo 2

Linguagens Regulares

Definição 2.1 (linguagem regular)**.** Uma linguagem é *regular* se é finita, ou se é a união, concatenação ou fecho de linguagens regulares. ♦

Exemplo 2.2.

- $\emptyset$ é linguagem regular (a linguagem sem nenhuma palavra), porque é um conjunto finito;
- $\{\varepsilon\}$ é linguagem regular, porque é finito (tem somente a palavra vazia, e portanto tem cardinalidade um)
- O conjunto $\{x\}$, onde $x \in \Sigma$, é uma linguagem regular sobre o alfabeto Σ, porque é finito. ◀

Exemplo 2.3. O conjunto de sequências binárias de tamanho no máximo dez é uma linguagem regular, porque é finito. ◀

Exemplo 2.4. O conjunto das cadeias formadas por vários zeros seguidos de vários uns é uma linguagem regular. Isto porque é a concatenação de duas linguagens regulares:

- 0^*, a linguagem das sequências de zeros (que é o fecho de $\{0\}$);
- 1^*, a linguagem das sequências de uns. ◀

Exemplo 2.5. A linguagem das sequências de as e bs de tamanho par é regular: ela é fecho da linguagem finita $\{aa, ab, ba, bb\}$. ◀

Exemplo 2.6. Seja $\Sigma = \{a, b\}$. A linguagem das palavras palíndromas sobre Σ *não* é regular. Ao contrário do que possa parecer inicialmente, não basta dizer que cada palavra da linguagem das palíndromas é a concatenação de palavras de Σ^*. Acontece que a concatenação de Σ^* com si mesma é $(\Sigma^*)(\Sigma^*)$, e esta linguagem não inclui apenas as palavras palíndromas! Ela também inclui $aabb$, por exemplo, porque $aa \in \Sigma^*$ e $bb \in \Sigma^*$. ◀

2.1 Expressões Regulares

A maneira mais natural de expressar linguagens regulares da maneira como as definimos é obtendo uma notação diretamente da definição. Isto resulta no que chamamos de *expressões regulares*.

Definição 2.7 (expressão regular)**.** Seja Σ um alfabeto.

- $a \in \Sigma$ é expressão regular sobre Σ
- a cadeia vazia, ϵ, é expressão regular sobre Σ
- a linguagem vazia, $\emptyset$, é expressão regular sobre Σ

Se R_1, R_2 são expressões regulares sobre Σ, então também são:

- $(R_1 + R_2)$
- $(R_1 R_2)$
- (R_1^*) ♦

Exemplo 2.8. A expressão regular a^*bc^* representa a linguagem das cadeias contendo um b com vários as à esquerda e vários cs à direita. *A linguagem não determina qualquer relação entre a quantidade de* a*'s e de* c*'s*, portanto a quantidade de as não precisa ser igual à quantidade de cs. ◄

Exemplo 2.9. A expressão regular $(ab+xy)(fg+jk)^*z$ representa a linguagem das palavras que começam com ab ou xy, seguida de uma sequência de fg's e jk's, seguida por um único z. ◄

Exemplo 2.10. A expressão regular

$$(0+1+2+3+4+5+6+7+8+9)^+.(0+1+2+3+4+5+6+7+8+9)^*$$

representa a linguagem dos números com ponto flutuante em base dez. ◄

Teorema 2.11. *As linguagens representadas por expressões regulares são exatamente as linguagens regulares.*

Demonstração. A definição de linguagem regular nos permite escrever linguagens regulares usando os símbolos $(A \cup B)$, $(A)^*$, e "AB" para concatenação; assim, uma linguagem regular pode ser descrita por uma fórmula; esta fórmula é exatamente uma expressão regular, exceto que, por convenção, em expressões regulares denotamos a união por $+$. □

2.2 Autômatos Finitos

Uma maneira de representar uma linguagem regular é através do autômato que a reconhece. Diferentes tipos de autômato servirão para reconhecer diferentes classes de linguagem. Apresentamos aqui os *autômatos finitos*, que reconhecem linguagens regulares.

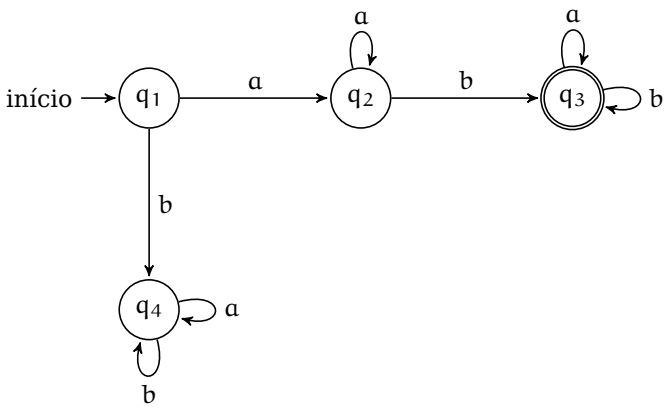
Em cada passo de computação, o autômato finito realiza o seguinte.

1. verifica o estado atual;
2. lê um símbolo da fita.
3. Se não existem mais símbolos a ler e o estado atual for marcado como "final", para aceitando a palavra; se o estado não for final para rejeitando a palavra;
4. dependendo do estado e do símbolo lido, determina um novo estado;
5. muda para o novo estado escolhido;
6. avança a cabeça de leitura uma posição para a direita na fita.

Definição 2.12 (autômato finito)**.** Um *autômato finito* é composto de

- um conjunto finito de estados Q;
- um alfabeto finito Σ;
- $\delta : Q \times \Sigma \to Q$, uma função de transição;
- $q_0 \in Q$, o estado inicial;
- $F \subseteq Q$, um conjunto de estados finais. ⧫

Exemplo 2.13. O seguinte diagrama mostra um autômato bastante simples, que reconhece a linguagem de palavras começando com a, e contendo pelo menos um b, sobre o alfabeto $\{a, b\}$.



O estado inicial (q_0) é marcado com uma seta vindo de nenhum outro estado. O único estado final (q_3) é indicado por duas circunferências concêntricas.

Ou seja,

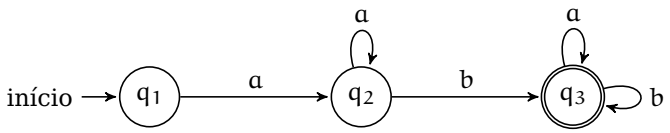
$$
\begin{aligned}
Q &= \{q_1, q_2, q_3\} \\
\Sigma &= \{a, b\} \\
\delta(q_1, a) &= q_2 \\
\delta(q_1, b) &= q_4 \\
\delta(q_2, a) &= q_2 \\
\delta(q_2, b) &= q_3 \\
\delta(q_3, a) &= q_3 \\
\delta(q_3, b) &= q_3 \\
\delta(q_4, a) &= q_4 \\
\delta(q_4, b) &= q_4 \\
q_0 &= q_1 \\
F &= \{q_3\}
\end{aligned}
$$

◀

Outra forma de representar o autômato é através de uma *tabela de transições*:

		a	b
▶	q_1	q_2	q_4
	q_2	q_2	q_3
	q_3	q_3	q_3
	q_4	q_4	q_4

Como δ é função, precisamos especificar transições para a e b saindo de cada estado. Podemos, no entanto, omitir estados como q_4, que só existem para oferecer um caminho que certamente não levará à aceitação da palavra.



O autômato é $(Q, \Sigma, \delta, q_1, F)$, com $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$ $F = \{q_3\}$ e

$$
\begin{aligned}
\delta(q_1, a) &= q_2, \\
\delta(q_2, a) &= q_2, \\
\delta(q_2, b) &= q_3, \\
\delta(q_3, a) &= q_3, \\
\delta(q_3, b) &= q_3
\end{aligned}
$$

Presumimos que, sendo um AFD, as transições faltantes levam a algum estado "inválido" (como q_4), que omitimos do diagrama.

Definição 2.14 (configuração de autômato finito)**.** Uma *configuração* de um autômato é (q, w), onde q é um estado do autômato e w é uma cadeia.

Se, estando em uma configuração $(q, a\alpha)$, o autômato lê o símbolo a, e termina no estado p, denotamos

$$(q, a\alpha) \vdash (p, \alpha)$$

Denotamos $\vdash^*$ quando há uma sequência de passos do autômato levando de uma configuração a outra. ♦

Exemplo 2.15. Se a palavra $aabaa$ estiver na fita, e o autômato do Exemplo 2.13 estiver no estado q_1, as três primeiras configurações pelas quais ele passará são

$$\begin{aligned} (q_1, aabaa) &\vdash (q_2, abaa) && (\delta(q_1, a) = q_2) \\ &\vdash (q_2, baa) && (\delta(q_2, a) = q_2) \end{aligned}$$

Podemos abreviar esta sequência de configurações por $(q_1, aabaa) \vdash^* (q_2, baa)$. ◄

Definição 2.16 (reconhecimento de linguagem por autômato)**.** Uma palavra w, composta pelos símbolos $w_1, w_2, \ldots$ é aceita por um autômato finito A se existe uma sequência de estados $r_1, r_2, \ldots, r_n$ tais que

- r_1 é estado inicial;
- $\delta(r_i, w_{i+1}) = r_{i+1}$;
- r_n é estado final.

♦

Exemplo 2.17. Executaremos o autômato do Exemplo 2.13 tendo como entrada a cadeia $aabba$.

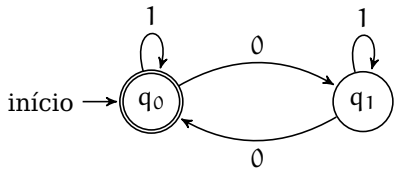
$$\begin{aligned} &(q_1, aabba) \\ \vdash &(q_2, abba) && (\delta(q_1, a) = q_2) \\ \vdash &(q_2, bba) && (\delta(q_2, a) = q_2) \\ \vdash &(q_3, ba) && (\delta(q_2, b) = q_3) \\ \vdash &(q_3, a) && (\delta(q_3, b) = q_3) \\ \vdash &(q_3, \varepsilon) && (\delta(q_3, a) = q_3) \end{aligned}$$

Novamente, podemos abreviar $(q_1, aabba) \vdash^* (q_3, \varepsilon)$.

A configuração quando a palavra termina de ser lida é (q_3, ε). Como q_3 é final e a cadeia restante é vazia, o autômato aceita a palavra. ◄

Damos mais alguns exemplos de autômatos finitos.

Exemplo 2.18. O autômato a seguir reconhece sequências binárias com quantidade par de zeros.

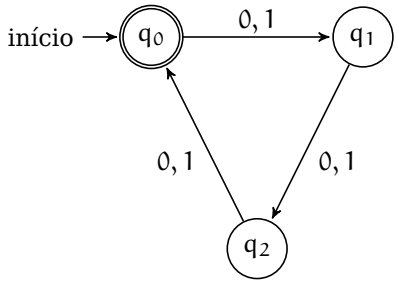


O autômato é (Q, Σ, δ, F), com $Q = \{q_0, q_1\}$, $\Sigma = \{0, 1\}$, $F = \{q_0\}$, e

$$\begin{aligned}
\delta(q_0, 0) &= q_1, \\
\delta(q_0, 1) &= q_0, \\
\delta(q_1, 0) &= q_0, \\
\delta(q_1, 1) &= q_1.
\end{aligned}$$

◀

Exemplo 2.19. O autômato a seguir reconhece sequências binárias com comprimento múltiplo de três. Nas arestas, "$0, 1$" é abreviação para "tanto zero como um".



O autômato é (Q, Σ, δ, F), com $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$, $F = \{q_0\}$, e

$$\begin{aligned}
\delta(q_0, 0) &= q_1, \\
\delta(q_0, 1) &= q_1, \\
\delta(q_1, 0) &= q_2, \\
\delta(q_1, 1) &= q_2, \\
\delta(q_2, 0) &= q_0, \\
\delta(q_2, 1) &= q_0.
\end{aligned}$$

◀

2.2.1 Autômatos Finitos Não-Determinísticos

Os autômatos finitos tem um inconveniente: eles tem uma *função* de transição. Especificá-la pode ser trabalhoso e sujeito a erros; podemos trocá-la

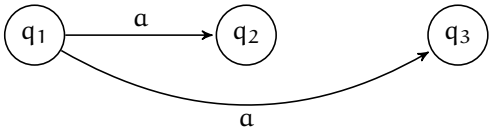
por uma *relação* de transição, de forma a especificar transições para estados diferentes usando o mesmo símbolo.

Definição 2.20 (autômato finito não-determinístico)**.** Um autômato *finito não determinístico (AFN ou NFA*[1]*)* é semelhante a um autômato finito, exceto que o AFN tem uma *relação de transição* $\Delta : Q \times \Sigma \cup \{\epsilon\} \times Q$ ao invés de uma função de transição.

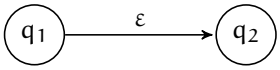
Pela semelhança e uso pretendido, poderemos abusar da notação e escrever $\Delta : Q \times \Sigma \cup \{\epsilon\} \to Q$, como se Δ fosse uma função:

$$\Delta(q_1, a) = q_2$$
$$\Delta(q_1, a) = q_3 \qquad \blacklozenge$$

O fato de Δ ser uma relação, e não uma função, significa que pode haver mais de um estado destino para cada par estado/símbolo (q, s).



Por termos usado $\Sigma \cup \{\epsilon\}$ ao invés de Σ, o AFN pode realizar transições sem ler símbolos da fita; equivalentemente, realiza transições ao ler ε (denotado $\Delta(q_1, \varepsilon) = q_2$).



Uma transição vazia é não-determinística da mesma forma que as transições múltiplas saindo de um estado.

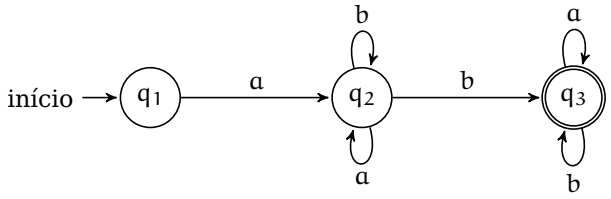
Há três interpretações para o não-determinismo.

- Interpretamos as transições não-determinísticas presumindo que o autômato "adivinhará" o caminho correto a usar para aceitar a palavra; assim, se houver *algum* caminho do estado inicial ao final, lendo a palavra e realizando transições, o autômato o fará e aceitará a palavra.
- Cada vez que há duas possibilidades de mudança de estado, o autômato cria uma cópia de si mesmo, e cada cópia segue em um estado diferente. Se alguma das cópias chegar ao estado final após ler a palavra da fita, a palavra é aceita.
- O autômato avança e retrocede: se havia uma escolha não-determinística a fazer, ele tenta uma possibilidade. Se não obtiver sucesso, volta e tenta a outra (ou seja, usa *backtracking*).

[1] *"Nondeterministic Finite Automata"* em Inglês.

Pode-se definir AFNs sem as ε-transições; também pode-se defini-los usando uma *função* de transição $\Delta : Q \times \Sigma \cup \{\varepsilon\} \to P(Q)$.

Exemplo 2.21. A linguagem do autômato do Exemplo 2.13 é a de as e bs, começando com a e contendo pelo menos um b. A seguir temos um autômato não-determinístico que a reconhece.



O autômato é não-determinístico porque tem uma *relação* de transição, e não uma função: há dois valores para (q_2, b), e não há valor definido para (q_1, b).

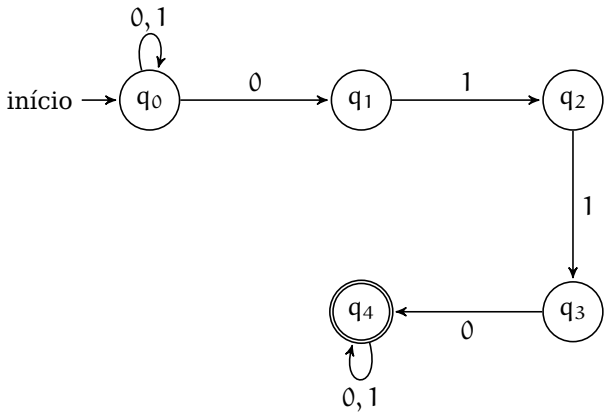
$$\begin{aligned} Q &= \{q_1, q_2\} \\ \Sigma &= \{a, b\} \\ \Delta(q_1, a) &= q_2 \\ \Delta(q_2, a) &= q_2 \\ \Delta(q_2, b) &= q_2 \\ \Delta(q_2, b) &= q_3 \\ \Delta(q_3, a) &= q_3 \\ \Delta(q_3, b) &= q_3 \\ q_0 &= q_1 \\ F &= \{q_3\} \end{aligned}$$

Uma tabela de transições deve listar todos os estados destino para cada par estado/símbolo:

		a	b
►	q_1	q_2	
	q_2	q_2	q_2, q_3
	q_3	q_3	q_3

◄

Exemplo 2.22. O autômato a seguir reconhece sequências binárias que contenham a subcadeia "0110".



O autômato é $(Q, \Sigma, \Delta, q_0, F)$, com $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{0, 1\}$, $F = \{q_4\}$, e

$$\begin{aligned}
\Delta(q_0, 0) &= q_0, \\
\Delta(q_0, 0) &= q_1, \\
\Delta(q_0, 1) &= q_0, \\
\Delta(q_1, 1) &= q_2, \\
\Delta(q_2, 1) &= q_3, \\
\Delta(q_3, 0) &= q_4, \\
\Delta(q_4, 0) &= q_4, \\
\Delta(q_4, 1) &= q_4.
\end{aligned}$$

◀

Teorema 2.23. *Seja A um autômato finito* determinístico *que aceita uma linguagem L. Então existe um autômato finito* não-determinístico *que também aceita L.*

Demonstração. Trivial: toda função é relação, portanto todo autômato finito determinístico é também não-determinístico. □

Teorema 2.24. *Se A é um autômato finito* não-determinístico *que aceita uma linguagem L, então é possível construir um autômato finito* determinístico *que também aceita L.*

Na demonstração a seguir usamos duas idéias:

- Se há mais de uma transição (q_i, a), indo a estados diferentes, então dizemos que o autômato foi para os dois estados simultaneamente, ou seja, está num conjunto de estados. Por isso cada estado do AFD será um elemento de $P(Q)$.
- Quando há uma transição vazia $(q_i, \varepsilon) \to q_j$, então qualquer transição que leve a q_i deve ser traduzida para "levando a q_i e *simultaneamente*

a todos os estados alcançáveis a partir de q_i *por zero ou mais transições vazias*[2]". Este conjunto de "estados alcançáveis a partir de q_i por transições ε" será denotado $E(q_i)$.

Demonstração. Construiremos um AFD a partir do AFN; o AFD poderá ter mais estados que o AFN.

Seja $(Q, \Sigma, \Delta, q_0, F)$ o AFN.

Quando o AFN tem duas transições para estados diferentes com o mesmo símbolo, criamos um estado no AFD representando *os dois* estados destino. Se $\Delta(r, a)$ leva tanto a s como a t, então, para todo estado R contendo r ($R = \{\ldots, r, \ldots\}$), criamos uma transição para o estado contendo s e t.

Um AFN pode ter transições vazias de um estado a outro. denotamos por $E(q)$ o conjunto de estados que podem ser alcançados por q através de zero ou mais transições vazias. Definimos recursivamente: $E(q)$ é o menor conjunto tal que

- $q \in E(q)$, e
- se $\delta(q, \varepsilon) = r$ então $E(r) \subseteq E(q)$.

Para cada transição, se o estado destino era um estado R, ele será $E(R)$.

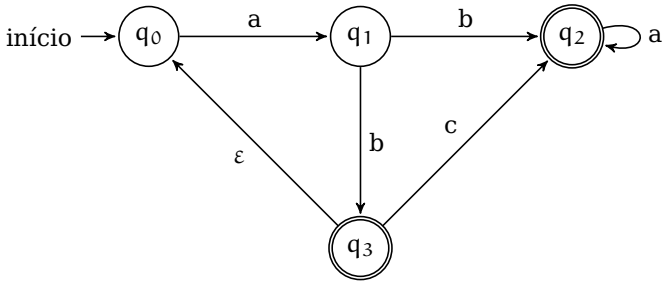
O AFD será $(Q', \Sigma, \delta, q_0', F')$:

- Q' é o conjunto das partes de Q, para que possamos representar, no AFD, múltiplos estados do AFN.
- No AFD, o estado $q' \in Q'$ é um *subconjunto dos estados do AFN*. Então, $\delta(q', a)$ leva a $\{q \ : \ q \in E(\Delta(p, a)), \text{para algum } \ p \in q'\}$
- $q_0' = \{q_0\}$.
- F contém todos os estados em Q' que incluem um estado final de Q.

O AFD construído claramente aceita a mesma linguagem que o AFN, porque simula seu comportamento. □

Exemplo 2.25. Usaremos o Teorema 2.24 para converter o AFN a seguir em AFD. Escolhemos um autômato simples, mas que tem uma transição vazia e uma escolha não-determinística para um símbolo lido (b, no estado q_1).

[2] Ou seja, se $\delta(q_1, \varepsilon) = q_2$, e $\delta(q_2, \varepsilon) = q_3$, então $q_3 \in E(q_1)$.



O autômato é $(Q, \Sigma, \delta, q_0, F)$, com $Q = \{q_0, q_1, q_2, q_3\}, \Sigma = \{a, b, c\}, F = \{q_2, q_3\}$, e

$$\begin{aligned}
\Delta(q_0, a) &= q_1 \\
\Delta(q_1, b) &= q_2 \\
\Delta(q_1, b) &= q_3 \\
\Delta(q_2, a) &= q_2 \\
\Delta(q_2, c) &= q_3 \\
\Delta(q_3, \varepsilon) &= q_0,
\end{aligned}$$

Ao invés de considerar todos os $P(Q)$ estados, tentaremos usar somente os estados alcançáveis a partir do inicial. Começamos então por $\{q_0\}$.

$$\delta(\{q_0\}, a) = \{q_1\}$$

Agora, em $\{q_1\}$ o autômato poderá ler b, indo tanto para q_2 como para q_3. Mas $E(q_3) = \{q_0, q_3\}$, logo

$$\delta(q_1, b) = \{q_0, q_2, q_3\}$$

Continuamos agora a partir de $\{q_0, q_2, q_3\}$. Neste estado, pode-se ler a (que era possível ler em q_0 e q_2) ou c (que era possível ler em q_3).

$$\begin{aligned}
\delta(\{q_0, q_2, q_3\}, a) &= \{q_1, q_2\} \\
\delta(\{q_0, q_2, q_3\}, c) &= \{q_2\}
\end{aligned}$$

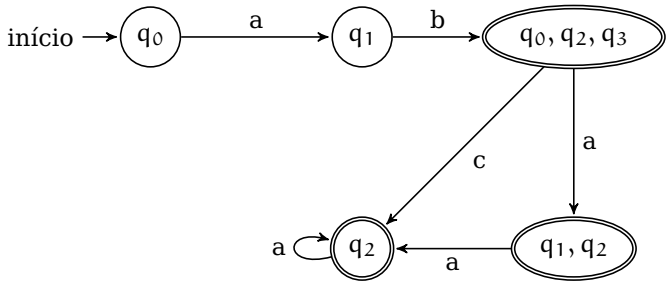
Tanto no estado q_0 como no estado q_2, a leva ao estado q_2, portanto a transição com a no novo estado $\{q_1, q_2\}$ será

$$\delta(\{q_1 q_2\}, a) = \{q_2\}.$$

Finalmente, estando em q_2 e lendo a, o autômato continua em q_2.

$$\delta(\{q_2\}, a) = \{q_2\}.$$

O autômato que resulta do processo é mostrado a seguir.



O autômato é $(Q, \Sigma, \delta, \{q_0\}, F)$, com $Q = \Big\{\{q_0\},\{q_1\},\{q_0, q_2, q_3\},\{q_1, q_2\},\{q_2\}\Big\}$ $\Sigma = \{a, b, c\}$, $F = \Big\{\{q_0, q_2, q_3\},\{q_1, q_2\},\{q_2\}\Big\}$, e

$$\begin{aligned}
\delta(\{q_0\}, a) &= \{q_1\},\\
\delta(\{q_1\}, b) &= \{q_0, q_2, q_3\},\\
\delta(\{q_0, a_2, q_3\}, a) &= \{q_1, q_2\},\\
\delta(\{q_1, q_2\}, a) &= \{q_2\},\\
\delta(\{q_0, q_2, q_3\}, c) &= \{q_2\},\\
\delta(\{q_2\}, a) &= \{q_2\}.
\end{aligned}$$

◀

2.2.2 Linguagem dos autômatos finitos

A classe de linguagens reconhecida por autômatos finitos é exatamente a das linguagens regulares. A demonstração é feita em dois Teoremas.

O primeiro Teorema mostra que qualquer linguagem regular é reconhecida por um autômato finito – o método consiste em mostrar que, dada uma expressão regular, existe um autômato finito que reconhece a linguagem dela.

O segundo Teorema mostra que a linguagem de todo autômato finito é regular. É uma demonstração por indução, mostrando que para qualquer autômato finito existe uma expressão regular equivalente.

Teorema 2.26. *Toda linguagem* regular *é reconhecida por um autômato finito.*

Demonstração. A demonstração é por indução na quantidade de operadores (união, concatenação e fecho) na expressão regular.

Para a base de indução, temos a linguagem vazia $\emptyset$; a linguagem contendo somente a cadeia vazia, $\{\varepsilon\}$, e as linguagens contendo símbolos isolados, $\{a\}$, com a pertencendo ao alfabeto da linguagem. Para estas, é trivial construir autômatos.

Procedemos ao passo de indução. A hipótese é de que toda linguagem representada por expressões regulares com menos que n operadores pode ser reconhecida por um autômato finito.

Há três casos a tratar:

- a expressão regular é uma união, $R = R_1 + R_2$. Neste caso, tanto R_1 como R_2 tem menos operadores que R, portanto há autômatos finitos A_1 e A_2 que as reconhecem. Assim, construímos um autômato A que tem um estado inicial isolado, levando aos estados iniciais de A_1 e A_2 com transições vazias.

- a expressão regular é uma concatenação, $R = R_1 R_2$. Novamente, tanto R_1 como R_2 tem menos operadores que R, portanto há autômatos finitos A_1 e A_2 que as reconhecem. Assim, construímos um autômato A composto pelos outros dois, mas modificando os estados finais de A_1 para que não sejam mais finais, e para que haja transição vazia deles para o estado inicial de A_2.

- a expressão regular é um fecho, $R = R_1^*$. A expressão R_1 tem menos operadores que R, então existe um autômato finito A_1 que a reconhece. Criamos um novo estado inicial, e o ligamos com transição ε ao estado inicial de A_1. Ligamos todos os estados finais de A_1 ao novo estado inicial, com transições ε.

Assim, há autômatos finitos que representam qualquer expressão regular. □

Teorema 2.27. *As linguagens reconhecidas por autômatos finitos são regulares.*

Demonstração. Usaremos indução na quantidade de estados usados para representar cadeias.

Suponha que uma linguagem seja aceita por um autômato finito determinístico M. Presuma, sem perda de generalidade, que os estados de M são $\{q_1, q_2, \ldots, q_n\}$, e que o estado inicial é q_1. Note que estamos intencionalmente evitando usar o estado "q_0".

Seja R_{ij}^k o conjunto das cadeias que seriam aceitas começando no estado q_i e terminando no estado q_j, mas usando estados intermediários entre q_1 e q_k (ou seja, sem usar estados intermediários com índice maior que k).

Uma definição recursiva de R_{ij}^k é dada a seguir.

$$
\begin{aligned}
R_{ij}^0 &= \{a \ : \ \delta(q_i, a) = q_j\} && (i \neq j)\\
R_{ii}^0 &= \{a \ : \ \delta(q_i, a) = q_i\} \cup \{\varepsilon\}\\
R_{ij}^k &= R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1} \ \cup \ R_{ij}^{k-1}
\end{aligned}
$$

A demonstração de que a linguagem reconhecida por um autômato equivale a uma expressão regular ser construída por indução no índice k. *Como temos uma definição indutiva, ela nos servirá de guia para a construção da demonstração.*

Para cada conjunto de cadeias R_{ij}^k, construiremos uma expressão regular equivalente, que denotaremos r_{ij}^k.

A base se dá com $k = 0$. R_{ij}^0 será um conjunto finito de cadeias unitárias, e possivelmente também a palavra vazia. Se não há transições saindo de q_i, então $R_{ij}^0 = \emptyset$. Então r_{ij}^0 será uma dentre as que seguem

$$r_{ij}^0 = a_1 + a_2 + \cdots + a_n$$
$$r_{ij}^0 = a_1 + a_2 + \cdots + a_n + \varepsilon$$

sendo possível que não haja nenhum a_l: R_{ij}^0 pode ser $\emptyset$ ou $\{\varepsilon\}$.

Para o passo indutivo, a hipótese é de que existe uma expressão regular para qualquer palavra gerada usando no máximo $k-1$ estados intermediários. A terceira linha da definição de R_{ij}^k nos diz que este pode ser construído usando

$$R_{ik}^{k-1}, R_{kk}^{k-1}, R_{kj}^{k-1} R_{ij}^{k-1},$$

todos com $k-1$ estados intermediários. Então a expressão regular para R_{ij}^k é

$$(r_{ik}^{k-1})(r_{kk}^{k-a})^*(r_{kj}^{k-1}) + r_{ij}^{k-1}.$$

A linguagem do autômato é

$$\bigcup_{q \in F} R_{1q}^n$$

portanto a expressão regular equivalente a ele será

$$+_{q \in F}\, r_{1q}^n \qquad \square$$

2.3 Gramáticas Regulares

Há uma classe de gramáticas, chamadas de *gramáticas regulares*, que gera a classe de linguagens regulares.

Definição 2.28 (gramática linear)**.** Uma gramática é *linear à direita* se suas regras de produção são uma função $p : N ::= TN$ – ou seja, todas as suas produções são da forma

$$A ::= \alpha B$$

Uma gramática é *linear à esquerda* se suas regras de produção são uma

função $p : N \to NT$ – ou seja todas as suas produções são de forma

$$A ::= B\alpha$$ ♦

Definição 2.29 (gramática regular)**.** Uma gramática é *regular* se é linear à direita ou linear à esquerda. ♦

Se uma gramática tem produções do tipo "linear à esquerda" misturadas com outras, do tipo "linear à direita", ela não é necessariamente regular. O Exercício 19 pede a demonstração disso.

Exemplo 2.30. A seguinte gramática é regular, porque é linear à direita.

$$\begin{aligned} S &::= xA \\ S &::= yB \\ A &::= aS \\ B &::= bS \\ B &::= b \end{aligned}$$

Dentre outras, esta gramática gera as palavras

$$yb, ybb, ybbb, ybbbbb, xayb, xaybb, xaxaybbbbb.$$ ◄

Exemplo 2.31. A seguinte gramática regular gera cadeias de dígitos representando números divisíveis por 25.

$$\begin{aligned} S &::= 0S \mid 1S \mid 2S \mid 3S \mid 4S \\ S &::= 5S \mid 6S \mid 7S \mid 8S \mid 9S \\ S &::= 00 \mid 25 \mid 50 \mid 75 \end{aligned}$$

Como exemplo, mostramos uma derivação.

$$\begin{aligned} S &\Rightarrow 3S \\ &\Rightarrow 37S \\ &\Rightarrow 3775 \end{aligned}$$ ◄

Teorema 2.32. *As linguagens representadas por gramáticas regulares são exatamente as linguagens regulares.*

Demonstração. **(Rascunho)** A demonstração consiste em observar que uma gramática regular pode ser vista como notação para autômatos finitos, sendo que as palavras geradas pela gramática são as mesmas aceitas pelo autômato.

Os símbolos não-terminais representam estados; o símbolo inicial representa o estado inicial.

Uma transição da forma

$$A ::= bC$$

representa uma transição: no estado A, a leitura do símbolo b leva ao estado C.

Uma transição da forma

$$A ::= b$$

representa uma transição para um estado final.

Pode haver mais de uma transição saindo de um estado:

$$A ::= bB$$
$$A ::= bC$$
$$A ::= dE$$ □

2.4 Propriedades

Diferentes tipos de linguagem apresentam diferentes propriedades. As linguagens regulares possuem várias propriedades interessantes. Demonstramos aqui duas delas, e enunciamos mais duas como exercício.

Teorema 2.33. *Linguagens regulares são fechadas sob complemento: se* L *é uma linguagem regular sobre um alfabeto* Σ *(e portanto* $L \in \Sigma^*$*), então* $\Sigma^* - L$ *é, também, uma linguagem regular.*

Demonstração. Seja $(Q, \Sigma, \delta, q_0, F)$ um AFD que reconhece a linguagem L. Então o AFD que reconhece $\overline{L}$ é $(Q, \Sigma, \delta, q_0, Q - F)$. Nas configurações onde o autômato de L pararia, o de $\overline{L}$ não para, e vice-versa. □

Teorema 2.34. *Linguagens regulares são fechadas sob interseção.*

Demonstração. Trivialmente: se L_1, L_2 são regulares, então são fechadas para complemento e união. Então

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

Exemplo 2.35. A linguagem L das sequências binárias cujas palavras contém quantidade par de zeros, quantidade ímpar de uns, e contém a subcadeia 0110 é regular.

Basta observar as seguintes linguagens:

- L_1, a linguagem das palavras que contém a subcadeia 0110 (definida pela expressão regular $(0+1)^*0110(0+1)^*$);
- L_2, a linguagem das palavras que contém quantidade par de zeros (definida pela expressão $((00)+1)^*$, logo é regular);

- L_3, a linguagem das palavras que contém quantidade ímpar de uns (complemento de $((11)+0)^*$, logo é regular).

Como L é a interseção de L_1, L_2 e L_3, e linguagens regulares são fechadas para interseção, então L é regular. ◀

Os Exercícios 17 e 18 pedem a demonstração dos Teoremas 2.36 e 2.37.

Teorema 2.36. *Linguagens regulares são fechadas sob diferença: Se* L *e* R *são regulares, e* $R \subseteq L$*, então* $L - R$ *é regular.*

Teorema 2.37. *Linguagens regulares são fechadas sob reversão: Se* L *é regular, então* L^R*, a linguagem das cadeias de* L *escritas de trás para a frente, também é regular.*

Definição 2.38 (homomorfismo de cadeias)**.** Um *homomorfismo de cadeias* sobre um alfabeto Σ é uma função $f : \Sigma \to \Sigma^*$. ♦

Teorema 2.39. *Linguagens regulares são fechadas sob homomorfismo: se uma linguagem* L *sobre um alfabeto* Σ *é regular, e* f *é um homomorfismo sobre* Σ*, então* $f(L)$ *é regular.*

Exemplo 2.40. Seja $\Sigma = \{a, b, c\}$, e considere a linguagem $L = a^*bc^*$.

Um exemplo de homomorfismo de cadeia sobre Σ é

$$f(a) = ac$$
$$f(b) = b$$
$$f(c) = c$$

Quando aplicamos o homomorfismo de cadeia sobre a linguagem L, obtemos a linguagem $(ac)^*bc^*$. ◀

2.5 Lema do Bombeamento

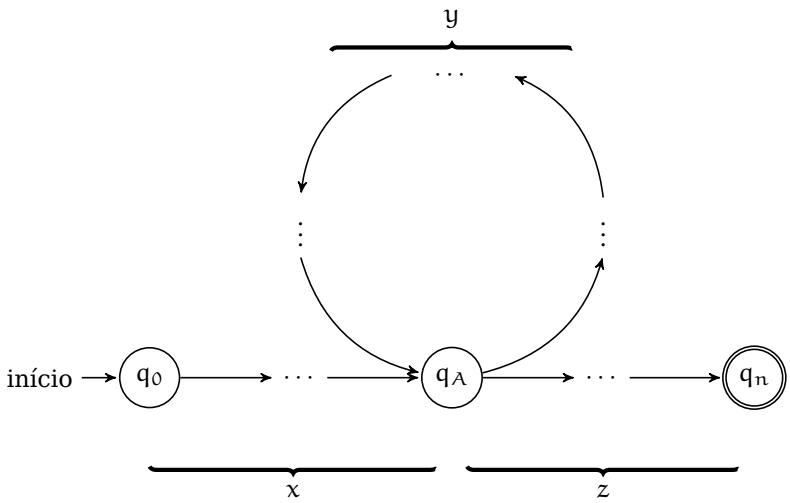
Para mostrar que uma linguagem é regular, podemos simplesmente construir um autômato, expressão regular ou gramática que a reconheça (ou gere).

Existe uma técnica para demonstrar que uma linguagem *não* é regular: o Lema do Bombeamento, baseando-se em uma propriedade de linguagens regulares, permite construir esse tipo de demonstração com relativa facilidade. O lema, como enunciado, descreve propriedades de linguagens que *são* regulares, e é usado, portanto, em demonstrações por contradição: mostramos que uma linguagem *não pode* ter aquelas propriedades, portanto não pode ser regular.

De maneira simplificada, o Lema do Bombeamento diz que se uma linguagem é regular, existe um tamanho máximo para as palavras da linguagem sem que haja repetição de trechos dentro da palavra.

Suponha que uma palavra tenha mais símbolos que a quantidade de estados no autômato que a reconhece. Então, para reconhecê-la, o autômato deve ter passado por alguns estados mais de uma vez – ou seja, passou por um ciclo. Mas se é possível fazer um ciclo ao reconhecer uma palavra, então esse ciclo pode ser repetido qualquer número de vezes (zero ou mais), gerando outras palavras da linguagem!

Uma representação visual do Lema do Bombeamento é dada a seguir. Se xyz pertence à linguagem, então $xy, xyz, xyyz, xyyyz$, e em geral, xy^iz também pertencem.



Por exemplo, para a linguagem $a(bc)^*d$, qualquer palavra com mais de quatro símbolos terá necessariamente o trecho "bc" repetido (concatenado com si mesmo).

$$a\,d$$
$$a\,bc\,d$$
$$a\,\underline{bc}\,\underline{bc}\,d$$
$$a\,\underline{bc}\,\underline{bc}\,\underline{bc}\,d$$

Lema 2.41 (do bombeamento, para linguagens regulares)**.** *Seja* L *uma linguagem regular. Então existe um número natural* p*, chamado de* comprimento de bombeamento*, tal que: para toda cadeia* $s \in L$*, com* $|s| \geq p$*, pode-se dividir* s *em partes,* $s = xyz$*, tal que*

- $\forall i \geq 0$, $xy^iz \in L$*;*
- $|y| > 0$*;*

- $|xy| < p$.

Podemos redigir a demonstração falando de estados de um autômato ou de símbolos não-terminais em uma gramática, já que são essencialmente a mesma coisa. Escolhemos a abordagem da gramática.

Demonstração. Seja G uma gramática linear à direita,e s uma palavra gerada por G, com $|s| \geq p$.

Qualquer derivação de s usando esta gramática precisará de pelo menos $|p|$ passos de derivação, porque somente um símbolo é gerado por passo. Isso significa que há pelo menos um não-terminal que será substituído mais de uma vez. Mas isso significa que ele pode ser substituído qualquer número de vezes, ainda gerando uma palavra da gramática.

Damos ao trecho que repete o nome de y. Ao trecho antes dele o nome de x, e ao trecho depois dele o nome de z. Claramente,

- $\forall i \geq 0$, $xy^iz \in L$;
- $|y| > 0$, porque $|p|$ é estritamente maior que $|N|$, logo deve haver pelo menos uma repetição;
- $|xy| < p$. □

O Exercício 20 pede a reconstrução dessa demonstração usando autômatos e expressões regulares.

Proposição 2.42. *A linguagem* a^nb^n *não é regular.*

Demonstração. Suponha que a^nb^n é regular, e seja s uma palavra desta linguagem. Como presumimos que a linguagem é regular, pelo Lema do Bombeamento, $s = xyz$, com as propriedades enunciadas no Lema. Dividimos agora o resto da demonstração em casos:

1. y só contém símbolos a. Se xyz pertence à linguagem, pelo lema, xy^2z também pertence. Mas como y só contém as, então xy^2z contém mais as do que bs, e não pode pertencer à linguagem. Assim, descartamos esta possibilidade.
2. y só contém bs. Este caso é análogo ao caso (1), e deve ser descartado.
3. Temos que admitir, portanto, que y deve conter as e bs. Mas pela natureza da linguagem y tem todos os as antes dos bs.

Chegamos à conclusão de que a palavra não pode ser quebrada em xyz, com as propriedades dadas no Lema. A suposição que fizemos, de que é uma linguagem regular, deve portanto ser falsa. □

Proposição 2.43. *A linguagem* a^rb^s, *com* $r > s$ *não é regular.*

Demonstração. Usaremos o Lema do Bombeamento – mas neste caso precisaremos usar o valor p, definido no Lema.

Suponha que a linguagem seja regular, e que portanto satisfaça o Lema do Bombeamento para algum inteiro p. A cadeia $w = a^{p+1}b^p$ pertence à linguagem. Então esta cadeia pode ser dividida em xyz, satisfazendo $|y| > 0$ e $|xy| < p$. Observamos que y só pode conter as, pela condição $|xy| < p$. E toda cadeia xy^qz deve pertencer à linguagem. Escolhermos $q = 0$. A cadeia será xz, sem a parte y. Mas como y só continha as, e tínhamos exatamente um a a mais do que bs, chegamos a uma palavra da forma a^rb^s, com $r \leq s$, que não pertence à linguagem. □

Proposição 2.44. *Seja $\Sigma = \{a, b\}$. A linguagem ww, onde $w \in \Sigma^*$, não é regular.*

Demonstração. Suponha que a linguagem seja regular. Seja p o valor de bombeamento, como no lema. A cadeia a^pba^pb. claramente pertence à linguagem. Ela portanto deveria poder ser dividida em $a^pba^pb = xyz$, como no lema. Mas como $|y| > 0$ e $|xy| < p$ então xy deve conter somente as. Mas xy^iz deve pertencer à linguagem, e escolhemos $i = 0$. xy também deve estar na linguagem. Mas xy é

$$xy = a^{p-|y|}ba^pb,$$

que claramente não é da forma a^nba^nb. Chegamos a uma contradição, e rejeitamos a hipótese que diz que a linguagem é regular. □

Proposição 2.45. *A linguagem das sequências de símbolos "a" com comprimento primo não é regular.*

Demonstração. Suponha que a linguagem descrita no enunciado seja regular. Então, ela satisfaz o Lema do Bombeamento para um inteiro n. Escolha um primo $p > n + 2$, e seja s a cadeia com as de comprimento p.

Pelo Lema do Bombeamento, podemos dividir s em xyz, com $|y| > 1$, $|xy| \leq n$. Então, seja $|y| = k$. Pelo Lema do Bombeamento, a cadeia

$$xy^{p-k}z$$

deve pertencer à linguagem. Mas

$$|xy^{p-k}z| = |xz| + (p-k)|y| = p - k + (p-k)k = (k+1)(p-k).$$

O comprimento da cadeia pode, portanto, ser decomposto em dois fatores, e não é primo. A cadeia não pode fazer parte da linguagem, e chegamos a uma contradição. □

Exercícios

Ex. 6 — Apresente autômatos, gramáticas ou expressões regulares que reconheçam as linguagens a seguir.

a) Sobre o alfabeto $\{a, b, c\}$, a linguagem das palavras que terminam com a quando seu comprimento é divisível por dois; terminam com b quando seu comprimento é divisível por três; e terminam em c quando o comprimento é divisível por seis.

b) Sobre o alfabeto $\{1, 0\}$, a linguagem dos números binários que contenham a subcadeia "0101" **ou** a subcadeia "1010".

c) Sobre o alfabeto $\{x, y\}$, as palavras que nunca tem mais que duas ocorrências de x consecutivas.

d) Sobre o alfabeto $\{0, 1, 2, \ldots, 9\}$, as palavras que representam números **naturais estritamente menores que** 1003 e também os **maiores ou iguais a** 100000.

e) Sobre o alfabeto $\{a, b, c, \ldots, z\}$, a linguagem das palavras que contenham pelo menos duas das vogais "a", "e", "i" (em qualquer lugar e em qualquer quantidade).

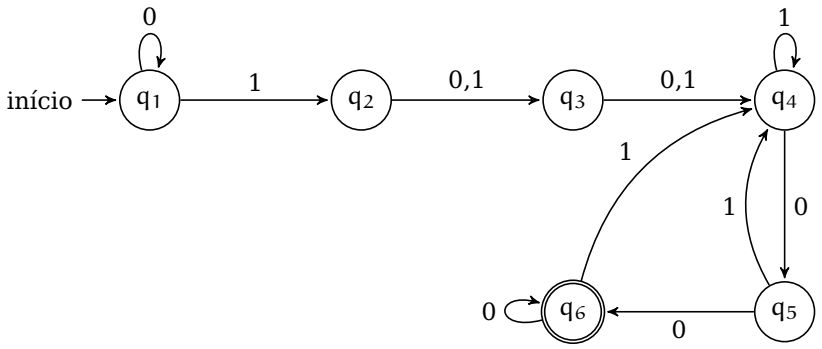
f) Sobre o alfabeto $\{a, b, c, d\}$, a linguagem das cadeias que, se começam com a tem comprimento par, e se começam com b tem comprimento divisível por 3.

g) Sobre o alfabeto $\{+, -, 1, 2, 3, \ldots\}$, a linguagem das expressões aritméticas sintaticamente válidas envolvendo números naturais. Note que "$4 - 10$" é uma expressão sintaticamente válida, mesmo que o resultado não seja natural.

h) Sobre o alfabeto $\{A, B, C, D\}$, a linguagem onde uma palavras só pode ter mais que um C se nela o primeiro A preceder o primeiro C; e além disso, um D só pode ocorrer imediatamente antes de um B.

i) Sobre o alfabeto $\{x, y, z\}$, a linguagem onde palavras tem comprimento no máximo 3, **exceto** se tiver a subcadeia "zz".

Ex. 7 — Descreva informalmente (mas **sucintamente** e com **precisão**) a linguagem do autômato a seguir (faça a descrição em termos de números binários).



Ex. 8 — A linguagem da gramática que você construiu no Exercício 4 é regular? Se não é, refaça o exercício, construindo uma linguagem regular.

Ex. 9 — Seria possível criar uma gramática semelhante à do Exercício 4-reg, que somente produza palavras de comprimento divisível por dois, três, ou por ambos?

Ex. 10 — Seria possível criar uma gramática semelhante à do Exercício 4-reg, que somente produza palavras de comprimento primo?

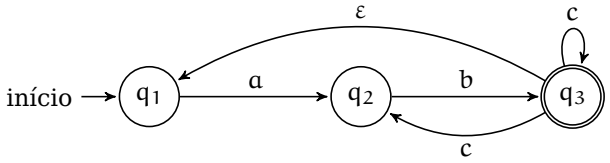
Ex. 11 — Seja Σ um alfabeto **finito**. A linguagem de todas as palavras sobre Σ que não repetem símbolo é regular? Porque? (Dica: "Sem perda de generalidade, seja $\Sigma = \{1, 2, 3, \ldots, k\}$")

Ex. 12 — Apenas para pensar: tente refazer o exercício anterior com Σ **infinito**. Que dificuldade encontra?

Ex. 13 — Construa um autômato **determinístico** sobre o alfabeto dos dígitos ($\Sigma = \{0, \ldots, 9\}$) que aceite a linguagem dos números divisíveis por 25. Observação: há várias maneiras de resolver esta questão, e vários autômatos corretos. Há uma solução com nove estados, e outra com mais estados (e possivelmente outras).

Ex. 14 — Prove que todo AFN pode ser convertido em um outro AFN que tem **um único** estado final.

Ex. 15 — Construa o AFD equivalente ao AFN a seguir.



Ex. 16 — Provamos que linguagens regulares são fechadas para interseção. Mostre como construir, a partir de AFDs A e B, um autômato C que

aceite a interseção das linguagens de A e B.

Ex. 17 — Prove o Teorema 2.36.

Ex. 18 — Prove o Teorema 2.37.

Ex. 19 — Mostre que uma gramática que tenha produções das formas $A ::= Bx$ e $A ::= xB$, onde A, B são não-terminais quaisquer e x é algum símbolo terminal, pode não ser regular.

Ex. 20 — Demonstre o Lema do Bombeamento (Lema 2.41) usando autômatos ao invés de gramáticas. Depois, tente usando expressões regulares.

Ex. 21 — Mostre que são ou que não são regulares as linguagens a seguir:

a) a linguagem sobre $\Sigma = \{a\}$, onde o tamanho das palavras é igual a $4k+3$, para $k \in \natural$.

b) a linguagem das sequências binárias de tamanho par.

c) a linguagem das sequências binárias de tamanho primo.

d) a linguagem das palavras binárias palíndromas.

e) a linguagem das palavras sobre $\Sigma = \{a, b, c, \ldots, z\}$, onde as palavras tem todas as vogais antes das consoantes.

f) $a^k b^m c^k$.

g) 1^k, onde k é quadrado perfeito.

h) a linguagem $L \subseteq \Sigma^*$, onde $\Sigma = \{a, b, c\}$, onde a quantidade de as é maior que a de bs.

i) $0^k 1^j$, com $k \neq j$.

Ex. 22 — Prove que a linguagem $L = \{wxw : x \in \{a, b\}^*, w \in \{0, 1\}^*\}$ não é regular. (L contém palavras que começam e terminam com a mesma subcadeia. Por exemplo, $000bababababaaab000$, $1baababa1$ e $01aaaabb01$ pertencem à linguagem)

Capítulo 3

Linguagens Livres de Contexto

Há uma classe de linguagens mais expressiva que a das linguagens regulares – a das *linguagens livres de contexto*. Este nome se justifica porque em gramáticas livres de contexto, as produções são da forma $A ::= \ldots$, onde não há restrição para o lado direito da produção, mas o lado esquerdo deve conter *um único não-terminal*, ao contrário das gramáticas *sensíveis ao contexto*, onde as regras podem ser da forma $\alpha A \beta ::= \ldots$ – ou seja, a substituição de A *depende do contexto*.

Linguagens livres de contexto são essenciais na construção de analisadores sintáticos para compiladores, e em outras áreas.

No início do Capítulo sobre linguagens regulares, as definimos a partir de suas propriedades. Para linguagens livres de contexto faremos de forma diferente: serão definidas como as linguagens geradas por gramáticas livres de contexto.

3.1 Gramáticas Livres de Contexto

Definição 3.1 (gramática livre de contexto)**.** Uma gramática é *livre de contexto* se suas regras de produção são da forma

$$S ::= \alpha$$

onde S é símbolo não-terminal, e α é concatenação de terminais e não terminais ($\alpha \in (N \cup \Sigma)^*$). ♦

Exemplo 3.2. A gramática a seguir tem uma regra "$S ::= aSb$", portanto não é regular. Mas como cada regra tem um único não-terminal no lado esquerdo, ela é livre de contexto.

$$\begin{aligned} S &::= aSb \\ S &::= \varepsilon \end{aligned}$$

A linguagem desta gramática é $a^n b^n$, que, como demonstramos no Capítulo 2, não é regular. Por exemplo, $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$. ◀

Definição 3.3 (linguagem livre de contexto)**.** Uma linguagem é *livre de contexto* se todas as suas palavras são geradas por uma gramática livre de contexto. ♦

Exemplo 3.4. A linguagem $a^n b^n$ é livre de contexto, porque é gerada pela gramática do Exemplo 3.2. ◀

Exemplo 3.5. A linguagem de parênteses balanceados, gerada pela gramática a seguir, é livre de contexto.

$$\begin{aligned} S &::= SS \mid (S) \\ S &::= \varepsilon \end{aligned}$$

Linguagens assim são chamadas de *Dyck languages*. ◀

Exemplo 3.6. Note que podemos trocar os parênteses por BEGIN e END, na gramática do Exemplo 3.5, e incluir a produção S ::= comando, obtendo a gramática

$$\begin{aligned} S &::= SS \mid \text{BEGIN } S \text{ END} \\ S &::= \text{comando} \mid \varepsilon \end{aligned}$$

Esta é a linguagem de sequências de "comandos", aninhados e agrupados por BEGIN e END. Por exemplo, a palavra

```
BEGIN
   BEGIN
   comando
   END
   BEGIN
      BEGIN
      comando
      END
   END
END
```

pertence a esta linguagem (a indentação serve apenas para facilitar a leitura; a palavra é uma sequência de símbolos). ◀

Exemplo 3.7. A linguagem $a^n b^* c^* d^n$, onde n é par também é livre de contexto, porque é gerada pela gramática

$$\begin{aligned} S &::= aaSdd \mid BC \\ B &::= bB \mid \varepsilon \\ C &::= cC \mid \varepsilon \end{aligned}$$ ◀

Teorema 3.8.

Definição 3.9 (derivação mais à esquerda)**.** Uma *derivação mais à esquerda* é feita escolhendo sempre o símbolo não-terminal mais à esquerda da palavra para aplicar uma regra de produção. ⧫

Exemplo 3.10. Considere a gramática

$$\begin{aligned} S &::= SAS \mid BSB \mid * \\ A &::= aA \mid a \\ B &::= bB \mid b \end{aligned}$$

A seguinte derivação *é* do tipo mais à esquerda:

$$\begin{aligned} S &\Rightarrow SAS \\ &\Rightarrow *AS \\ &\Rightarrow *aAS \\ &\Rightarrow *aaAS \\ &\Rightarrow *aaaS \\ &\Rightarrow *aaa* \end{aligned}$$

Note que sempre escolhemos o não-terminal mais à esquerda para substituir.

A seguinte derivação *não* é do tipo mais à esquerda:

$$\begin{aligned} S &\Rightarrow SAS \\ &\Rightarrow *AS \\ &\Rightarrow *A* && (\Leftarrow) \\ &\Rightarrow *aA* \\ &\Rightarrow *aaA* \\ &\Rightarrow *aaa* \end{aligned}$$

No passo indicado por $\Leftarrow$, o não-terminal A estava mais à esquerda que S, mas ainda assim, substituímos S por $*$.

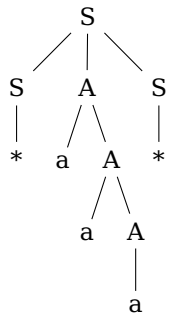
As palavras derivadas neste exemplo são iguais – "mais à esquerda" *não é uma propriedade de palavras, mas da derivação feita até chegar a uma palavra*. ◀

Definição 3.11 (árvore de derivação)**.** Seja G uma gramática livre de contexto com símbolo inicial S. Então uma *árvore da derivação* de G é uma árvore enraizada e rotulada, tal que

- a raiz da árvore tem como rótulo o símbolo inicial, S;
- se um nó interno tem rótulo A e filhos $B_1, \ldots, B_k$, então G tem uma produção
$$A ::= B_1 \ \cdots \ B_k \ ;$$
- folhas tem como rótulo símbolos terminais ou ε;
- cada nó interno pode ter no máximo uma folha ε.

Se $S \Rightarrow \beta_0 \Rightarrow \beta_1 \Rightarrow \cdots \Rightarrow \beta_k \Rightarrow \alpha$. é derivação de uma palavra α usando as regras de G, então cada passo da derivação corresponde a um nó interno e seus filhos. ♦

Exemplo 3.12. A derivação mais à esquerda do Exemplo 3.10 equivale à árvore a seguir.



As folhas, lidas da esquerda para a direita, contém a palavra $*aaa*$. ◀

Cada árvore de derivação corresponde a uma única derivação mais à esquerda – a demonstração é pedida no Exercício 28.

Teorema 3.13. *Existe uma correspondência um-para-um entre derivações mais à esquerda e árvores de derivação para uma palavra gerada por uma gramática.*

3.1.1 Ambiguidade

É possível, a partir da mesma gramática, derivar de formas diferentes a mesma palavra. Quando isto acontece, há consequências semânticas (é necessário, por exemplo, decidir se na expressão "$2+5*3$" qual das operações é realizada primeiro). Nesta Seção definimos o conceito de ambiguidade.

Definição 3.14 (gramática ambígua)**.** Uma gramática livre-de-contexto é *ambígua* se existe uma palavra em sua linguagem para a qual há mais de uma derivação mais à esquerda. ⧫

Exemplo 3.15. Criaremos uma gramática para expressões envolvendo variáveis, somas e multiplicações, incluindo parênteses.

$$\begin{aligned} E &::= E + E \\ E &::= E * E \\ E &::= \text{var} \\ E &::= (E) \end{aligned}$$

Nesta gramática, a palavra $a + b * c$ tem duas derivações mais à esquerda:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow a + E \\ &\Rightarrow a + E * E \qquad (*) \\ &\Rightarrow a + b * E \\ &\Rightarrow a + b * c \end{aligned}$$

Em $(*)$, trocamos o E por $E * E$.

Iniciamos a derivação trocando E *por* $E+E$*, e depois substituímos um dos "*E*"s restantes por* $E + E$*. Assim, encaramos a expressão como uma soma: de um lado,* a*; de outro,* $b*c$*. Isto significa que* $b*c$ *está sendo somado a* a *– de acordo com a intuição e as regras usuais de precedência de operadores.*

Tentamos outra derivação mais à esquerda.

$$\begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \qquad (*) \\ &\Rightarrow a + E * E \\ &\Rightarrow a + b * E \\ &\Rightarrow a + b * c \end{aligned}$$

Em $(*)$, trocamos o primeiro E por $E + E$.

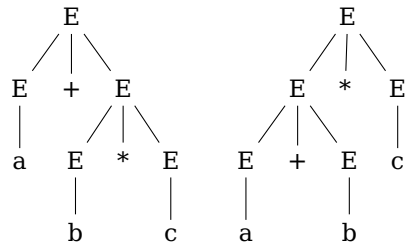
Iniciamos a derivação trocando E *por* $E * E$*. Neste contexto, isto significa que estamos encarando a expressão como uma multiplicação: de um lado, está* $a + b$*, e do outro,* c*. Significaria que a soma* $a + b$ *é que está sendo multiplicada, mas esta não é a forma usual de interpretar expressões! Normalmente damos prioridade à multiplicação.*[1]

Há duas derivações mais à esquerda para a mesma palavra, $a + b * c$. ◀

[1] Há pelo menos uma linguagem de programação onde "a+b*c" significa "a somado com b, depois multiplicado or c". Em Smalltalk, a cadeia "a+b*c" não é uma expressão matemática. Ela significa mande ao objeto "a" a mensagem "some a si mesmo com b"; depois, mande

Como há uma correspondência um-para-um entre derivações mais à esquerda e árvores de derivação, podemos dizer que uma gramática é ambígua se ela gera alguma palavra com mais de uma árvore de derivação.

Exemplo 3.16. As duas derivações mais à esquerda do Exemplo 3.15 correspondem às duas árvores a seguir.



◀

Exemplo 3.17. O trecho a seguir especifica a sintaxe do comando `if` em uma linguagem de programação.

$$\begin{aligned} \text{CMD} &::= \text{if BOOLEXP then CMD else CMD} \\ \text{CMD} &::= \text{if BOOLEXP then CMD} \\ \text{BOOLEXP} &::= \texttt{t} \mid \texttt{f} \end{aligned}$$

Esta gramática é ambígua. A palavra

$$\texttt{if t then if f then CMD}_1 \texttt{ else CMD}_2$$

tem duas derivações mais à esquerda. A ambiguidade está no fato de podermos escolher entre a primeira regra ou a segunda ao substituir CMD pela primeira vez. (O `else` pertence ao `if` interno ou ao externo?)

Esta ambiguidade existia em especificações da linguagem Pascal, e os compiladores sempre a resolvem associando o else ao `if` mais próximo.

É possível remover a ambiguidade da gramática – isto é pedido no exercício 27. ◀

Enunciamos a seguir, sem prova, um Teorema que afirma a existência de linguagens para as quais não há gramáticas não-ambíguas.

Teorema 3.18. *Há linguagens livres-de-contexto que são* inerentemente ambíguas *– ou seja, não existe gramática livre de contexto não-ambíguas que as gerem.*

No entanto, o próximo Teorema também é relevante.

---

ao objeto resultante a mensagem "multiplique a si mesmo por c". Isto é parte da filosofia fundamental de Smalltalk, onde *tudo* é realizado através de troca de mensagens entre objetos. Assim, em Smalltalk, `2+3*4` resulta em `20`.

Teorema 3.19. *Não existe algoritmo que possa verificar se uma linguagem é inerentemente ambígua.*

Apesar disso, existe um Lema (o "Lema de Ogden") que pode auxiliar a determinar que certas linguagens são inerentemente ambíguas.

Exemplo 3.20. A linguagem $\{a^i b^j c^k \ : \ i = j \text{ ou } j = k\}$ é inerentemente ambígua. ◀

Exemplo 3.21. A gramática do Exemplo 3.15 não é inerentemente ambígua. Podemos modificá-la, usando não-terminais diferentes para Expressão, Termo e Fator, forçando assim a derivação mais à esquerda que corresponde à interpretação usual de fórmulas, onde o operador $*$ tem precedência sobre $+$.

$$\begin{aligned} E &::= E + E \mid T \\ T &::= T * T \mid F \\ F &::= \text{var} \mid (E) \end{aligned}$$

Só há uma derivação mais à esquerda para "$a + b * c$":

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow T + E \\ &\Rightarrow F + E \\ &\Rightarrow a + E \\ &\Rightarrow a + T \\ &\Rightarrow a + T * T \\ &\Rightarrow a + F * T \\ &\Rightarrow a + b * T \\ &\Rightarrow a + b * F \\ &\Rightarrow a + b * c \end{aligned}$$ ◀

3.1.2 Forma Normal

Gramáticas livres de contexto podem ser postas em *formas normais*, que podem ser úteis no desenvolvimento de algoritmos para trabalhar com estas gramáticas, e também na demonstração de Teoremas (veja por exemplo o Teorema 3.60).

Antes de apresentar as formas normais, algumas definições são necessárias. Além do conceito de *produção vazia* (uma produção da forma $A ::= \varepsilon$), precisamos definir *símbolos úteis*.

Definição 3.22. Seja $G = (N, \Sigma, P, S)$ uma gramática. Um símbolo A não-terminal de G é *útil* se satisfaz duas condições: (i) A deve gerar uma cadeia

de terminais – A deve aparecer no lado esquerdo de uma regra, e deve ser possível gerar uma cadeia de terminais a partir de A (ou seja, $A \Rightarrow^* w$, com $w \in \Sigma^*$); (ii) A deve ser alcançável a partir da raiz de alguma árvore de derivação para alguma palavra ($S \Rightarrow^* \alpha A \beta$, com $\alpha, \beta \in (N \cup \Sigma)^*$). ♦

Lema 3.23. *Se* G *é uma gramática livre de contexto, é possível obter outra gramática* G'*, também livre de contexto, onde não há símbolos inúteis.*

Demonstração. (Rascunho) Seja $G = (N, \Sigma, P, S)$ uma gramática livre de contexto.

Primeiro, construímos uma nova gramática $G_1 = (N_1, \Sigma, P_1, S)$, onde todo não-terminal leva a cadeias de terminais, da seguinte forma:

Inicialmente, toda produção da forma $X ::= a$, com $a \in \Sigma$, ou $X ::= \varepsilon$ é incluída em P_1, e X é incluído em N_1.

Em seguida, para toda regra $Y ::= X_1 X_2 \dots X_k$, onde todos os X_i já foram incluídos em N', inclua também esta regra em P_1 e Y em N_1.

Depois disso, construímos $G_2 = (N_2, \Sigma, P_2, S)$, onde todos os não-terminais aparecem em alguma derivação começando pelo símbolo inicial.

Começamos com $N_2 = \{S\}$. Depois, repetimos o seguinte procedimento: para cada não-terminal A em N_2, consultamos as regras da forma $A ::= B_1 B_2 \dots B_k$, e incluímos os B_i em N_2, e a regra em P_2.

Desta forma, apenas os símbolos alcançáveis a partir da raiz permanecerão na gramática. □

Lema 3.24. *Se* $G = (N, \Sigma, P, S)$ *é uma gramática livre de contexto, é possível obter outra gramática* G'*, também livre de contexto, onde não há produções vazias, exceto por* $S ::= \varepsilon$*.*

Demonstração. Para cada regra $A ::= \varepsilon$, onde A não é o símbolo inicial, verifique cada regra $X ::= \alpha A \beta$ onde existem ocorrências de A, e crie uma nova regra $X ::= \alpha\beta$, removendo o A. Repita até não sobrar regras levando ao vazio, exceto por $S ::= \varepsilon$. □

Definição 3.25 (forma normal de Chomsky)**.** Uma gramática livre de contexto está na *forma normal de Chomsky* se suas regras são de uma das formas a seguir,

$$\begin{aligned} A &::= BC \\ A &::= a \end{aligned}$$

onde B e C não podem ser o símbolo inicial.

Também é permitida a regra $S ::= \varepsilon$, onde S é o símbolo inicial. ♦

Teorema 3.26. *Se* G *é uma gramática livre de contexto, é possível construir* G'*, na forma normal de Chomsky, que gera a mesma linguagem de* G*.*

Para levar uma gramática à forma normal de Chomsky, precisamos (i) eliminar as produções do tipo $A ::= B$; (ii) eliminar as produções vazias; e (iii) adequar as regras de produção com mais de dois símbolos no lado direito.

Demonstração. Dada uma gramática livre de contexto qualquer, o procedimento a seguir a transforma em uma gramática na forma normal de Chomsky, aceitando a mesma linguagem.

Primeiro, use o procedimento descrito na Demonstração do Lema 3.24.

Para cada regra da forma $A ::= B$, verifique as regras da forma "$B ::= \alpha$" e troque-as por "$A ::= \alpha$". Repetimos até não sobrar mais regras da forma $A ::= B$.

Para cada regra da forma $A ::= aB$, crie uma nova variável A' e a regra $A' ::= a$, e mude a regra para $A ::= A'B$.

Troque cada regra da forma $A ::= a_1a_2a_3 \cdots a_n$ por

$$\begin{aligned} A &::= a_1A_1 \\ A_1 &::= a_2A_2 \\ &\vdots \\ A_n &::= a_n \end{aligned}$$

A transformação claramente resulta em gramática equivalente (o que se pode notar em cada uma das transformações feitas), e na forma normal de Chomsky. □

Exemplo 3.27. A linguagem das palíndromas sobre o alfabeto $\{a, b\}$ é gerada pela gramática a seguir, que *não* está na forma normal de Chomsky:

$$S ::= aSa \mid bSb \mid \varepsilon$$

A mesma linguagem é gerada pela gramática a seguir.

$$\begin{aligned} S &::= AX \mid BY \mid \varepsilon \\ X &::= SA \\ Y &::= SB \\ A &::= a \\ B &::= b \end{aligned}$$

Note que usamos X e Y para quebrar as regras $S ::= aSa$ e $S ::= bSb$. ◄

Há outra forma normal, também relevante, chamada de *forma normal de Greibach*. O Exercício 43 pede a demonstração de que ela existe.

Definição 3.28 (forma normal de Greibach)**.** Uma gramática livre de contexto está na *forma normal de Greibach* se suas regras são de uma das formas a seguir,

$$\begin{aligned} A &::= aA_1A_2\ldots \\ A &::= a \end{aligned}$$

onde os A_i são símbolos não-terminais, diferentes do símbolo inicial.

Também é permitida a regra $S ::= \varepsilon$, onde S é o símbolo inicial. ♦

Teorema 3.29. *Para toda gramática livre de contexto existe uma gramática na forma normal de Greibach que gera a mesma linguagem.*

3.2 Autômatos com Pilha, Não-Determinísticos

Um autômato com pilha é semelhante a um autômato finito, exceto que tem uma *pilha* acoplada, e tem um alfabeto de símbolos que podem ser empilhados e desempilhados em cada transição.

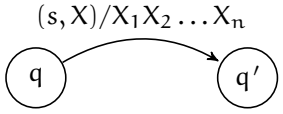
Ao contrário do que fizemos no estudo de autômatos finitos, começaremos com os autômatos com pilha não-determinísticos, para somente depois tratarmos dos determinísticos.

Em cada passo de computação, um autômato com pilha efetua o seguinte.

1. verifica o estado atual q;
2. lê o símbolo na posição atual da fita s;
3. lê o símbolo no topo da pilha e o desempilha x;
4. escolhe uma sequência de símbolos para empilhar $x_1x_2\ldots x_n$;
5. escolhe o próximo estado, q';
6. se o símbolo lido da fita foi o último, verifica se pode parar.

Para autômatos com pilha pode-se usar dois critérios de parada diferentes: podemos determinar que o autômato só possa parar quando a pilha estiver vazia, independente do estado em que se encontrar; ou podemos determinar que ele possa parar quando estiver em um estado final, não dependendo do conteúdo da pilha.

Usaremos a representação em diagrama como segue.



Neste diagrama, a transição

$$\delta(q, s, X) = \{\ldots, (q', X_1 X_2 \ldots X_n) \ldots\}$$

determina que, no estado q, tendo lido o símbolo s, e removendo X do topo da pilha, o autômato pode passar ao estado q' e empilhar a sequência $X_1, X_2, \ldots, X_n$.

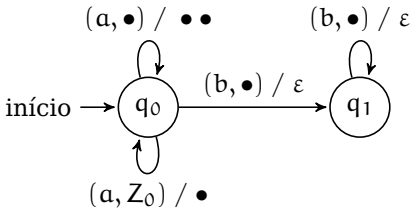
A seguir damos a definição.

Definição 3.30 (autômato com pilha)**.** Um *autômato com pilha*, (AFP ou PDA)[2] é composto de

- um conjunto finito de estados Q;
- um alfabeto finito Σ;
- um *alfabeto de fita* Γ;
- $\delta : Q \times \Sigma \times \Gamma_\varepsilon \to P(Q \times \Gamma^*)$, uma função de transição;
- $q_0 \in Q$, o estado inicial;
- $Z_0 \in \Gamma$, um símbolo inicial de pilha;
- $F \subseteq Q$, um conjunto de estados finais. ♦

A função de transição tem como domínio partes de $Q \times \Gamma^*$, portanto o autômato é não-determinístico.

Exemplo 3.31. O seguinte autômato com pilha opera sobre o alfabeto $\{a, b\}$.



O autômato é $(Q = \{q_0, q_1\}, \Sigma = \{a, b\}, \Gamma = \{Z_0, \bullet\}, \Delta, q_0, Z_0, F = \emptyset)$, com

$$\begin{aligned}
\delta(q_0, a, Z_0) &= \{(q_0, \bullet)\} \\
\delta(q_0, a, \bullet) &= \{(q_0, \bullet\bullet)\} \\
\delta(q_0, b, \bullet) &= \{(q_1, \varepsilon)\} \\
\delta(q_1, b, \bullet) &= \{(q_1, \varepsilon)\}.
\end{aligned}$$

◄

[2] Autômato Finito com Pilha, ou *PushDown Automaton*

Definição 3.32 (configuração de autômato com pilha)**.** A *descrição instantânea,* ou *configuração* de um autômato com pilha consiste em uma tupla contendo seu estado atual; o quanto falta ser lido; e o conteúdo da pilha. Se, estando em uma configuração $(q, s\sigma, X\alpha)$, o autômato lê o símbolo s, desempilha X, empilha Y, e termina no estado p, denotamos

$$(q, s\sigma, X\alpha) \vdash (p, \sigma, Y\alpha).$$

Querendo denotar que a sequência de configurações é de um autômato específico A, escrevemos $\vdash_A$. ♦

Exemplo 3.33. Estando a palavra $aabb$ na fita, o autômato do Exemplo 3.31 passa pelas configurações a seguir (além de outras).

$$\begin{aligned}(q_0, aabb, Z_0) &\vdash (q_0, abb, \bullet)\\ &\vdash (q_0, bb, \bullet\bullet)\\ &\vdash (q_1, b, \bullet)\end{aligned}$$ ◄

Lema 3.34. *Seja A um autômato com pilha. Se*

$$(q, a, \alpha) \vdash_A^* (r, b, \beta),$$

então, para quaisquer cadeias $w \in \Sigma^*$ e $\gamma \in \Gamma^*$,

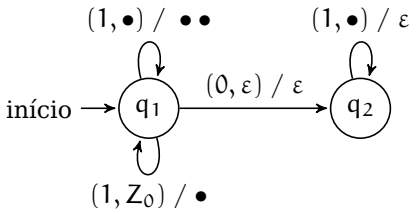
$$(q, aw, \alpha\gamma) \vdash_A^* (r, bw, \beta\gamma).$$

Definição 3.35 (critérios de parada para autômato com pilha)**.** Há dois critérios de parada possíveis para autômatos com pilha. Um deles é "quando o autômato chega a um estado final", ou seja, em uma configuração (q, ε, α), com $q \in F$; o outro é "quando a pilha estiver vazia", ou seja, em $(q, \varepsilon, \varepsilon)$, para qualquer $q \in Q$.

Se A é um autômato com pilha, usamos $V(A)$ para denotar a linguagem das palavras aceitas por A usando o critério de pilha vazia; e $L(A)$ para denotar a linguagem das palavras aceitas por A usando o critério de estado final. ♦

Adiante mostraremos que $L(A) = V(A)$ para todo autômato. O motivo para usar os dois critérios é que em diferentes demonstrações, um deles pode ser mais fácil de usar do que o outro.

Exemplo 3.36. O autômato a seguir aceita palavras da linguagem 1^n01^n (sequências de n uns, seguidas de um zero, seguido por n uns).



O autômato começa no estado q_1, onde pode ler símbolos 1, repetidamente. Para cada símbolo 1 lido, um símbolo • é empilhado. Quando ler um único zero, passa ao estado q_2. Ali, poderá ler uns, mas para cada um que ler, terá que desempilhar um •.

Definimos que este autômato só para *com a pilha vazia*, a quantidade de uns antes e depois do zero terá que ser a mesma. Nenhum estado é marcado como final, porque o conceito de estado final deixa de ser relevante com o critério da pilha vazia. A descrição do autômato é $(Q, \Sigma, \Gamma, \delta, q_1, Z_0, F)$, onde

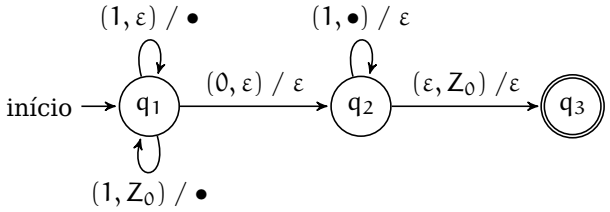
$$\begin{aligned} Q &= \{q_1, q_2\} \\ \Sigma &= \{0, 1\} \\ \Gamma &= \{\bullet\} \\ F &= \emptyset \\ \delta(q_1, 1, \varepsilon) &= \{(q_1, \bullet)\} \\ \delta(q_1, 0, \varepsilon) &= \{(q_2, \varepsilon)\} \\ \delta(q_2, 1, \bullet) &= \{(q_2, \varepsilon)\} \\ \delta(q_2, 1, Z_0) &= \{(q_2, \varepsilon)\} \end{aligned}$$

Verificamos agora a ação do autômato sobre a string 11011, listando a sequência de configurações pelas quais ele passa.

$$\begin{aligned} (q_1, 11011, Z_0) &\vdash (q_1, 1011, \bullet Z_0) \\ &\vdash (q_1, 011, \bullet\bullet Z_0) \\ &\vdash (q_2, 11, \bullet\bullet Z_0) \\ &\vdash (q_2, 1, \bullet Z_0) \\ &\vdash (q_2, \varepsilon, Z_0) \\ &\vdash (q_3, \varepsilon, \varepsilon) \end{aligned}$$

A última configuração é $(q_3, \varepsilon, \varepsilon)$, como esperado (os dois ε significam que não há mais símbolos a serem lidos da fita, e a pilha está vazia). ◄

Exemplo 3.37. Podemos modificar o autômato do Exemplo 3.36 para que aceite a mesma linguagem, mas usando o critério de parada de estado final ao invés do critério de pilha vazia.



O estado q_3 é marcado como final, para que o autômato possa parar. Ainda usamos o símbolo inicial da fita. Isso porque a definição de autômato com pilha o inclui, e mantê-lo não fará mal. A descrição do autômato é $(Q, \Sigma, \Gamma, \delta, q_1, Z_0, F)$, com

$$\begin{aligned} Q &= \{q_1, q_2, q_3\} \\ \Sigma &= \{0, 1\} \\ \Gamma &= \{Z_0, \bullet\} \\ F &= \{q_3\} \\ \delta(q_1, 1, \varepsilon) &= \{(q_1, \bullet)\} \\ \delta(q_1, 0, \varepsilon) &= \{(q_2, \varepsilon)\} \\ \delta(q_2, 1, \bullet) &= \{(q_2, \varepsilon)\} \\ \delta(q_2, \varepsilon, Z_0) &= \{(q_3, \varepsilon)\} \end{aligned}$$

Da mesma forma que fizemos no Exemplo 3.36, simulamos a ação do autômato sobre a string 11011, listando a sequência de configurações.

$$\begin{aligned} (q_1, 11011, \varepsilon Z_0) &\vdash (q_1, 1011, \bullet Z_0) \\ &\vdash (q_1, 011, \bullet\bullet Z_0) \\ &\vdash (q_2, 11, \bullet\bullet Z_0) \\ &\vdash (q_2, 1, \bullet Z_0) \\ &\vdash (q_2, Z_0,) \\ &\vdash (q_3, \varepsilon, \varepsilon) \end{aligned}$$

A última configuração é $(q_2, \varepsilon, \varepsilon)$, como esperado (os dois ε significam que não há mais símbolos a serem lidos da fita, e a pilha está vazia). ◀

Os dois critérios de parada (pilha vazia e estado final) são equivalentes – a classe de linguagens descrita pelos autômatos com pilha é a mesma, não dependendo do critério de parada escolhido.

Teorema 3.38. *Seja* L *a linguagem de um autômato com pilha usando critério de parada de pilha vazia. Então há um autômato com pilha, usando critério de parada de estado final, que reconhece* L*.*

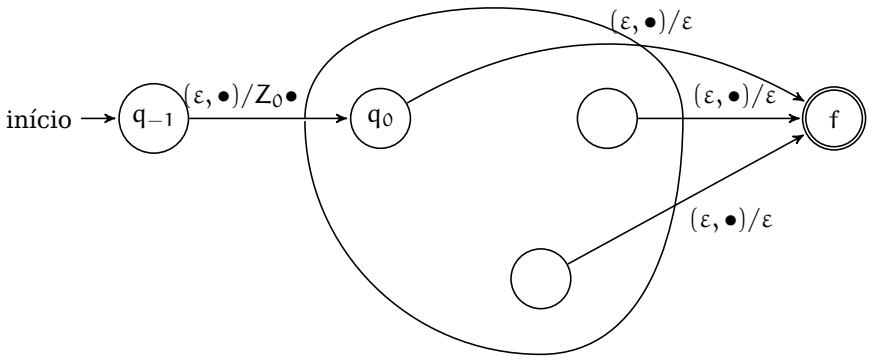
Demonstração. Seja $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ um autômato que para com a

pilha vazia. Construiremos outro autômato B que para quando chega a um estado final, e reconhece a mesma linguagem que A.

Criamos B a partir da descrição de A, inserindo um estado inicial extra, q_{-1} e um símbolo de pilha extra $\bullet$. Este novo símbolo ficará no fundo da pilha durante todo o trabalho, e só será retirado no último movimento.

Ao iniciar, para passar do novo estado inicial q_{-1} para o estado que era inicial em A (q_0), o autômato empilhará Z_0, porque o autômato A conta com este símbolo no fundo da pilha para trabalhar.

Também criamos um estado final f, para que o autômato possa parar.



Assim, $B = \big(Q \cup \{q_{-1}, f\},\ \Sigma,\ \Gamma \cup \{\bullet\},\ \delta',\ q_{-1},\ \bullet,\ \{f\}\big)$, onde δ' é a função de transição δ, de A, aumentada com as seguintes transições:

- Ao iniciar no estado q_{-1}, o autômato empilha Z_0 e passa para q_0:

$$\delta'(q_{-1}, \varepsilon, \bullet) = \{(q_1, Z_0\bullet)\}$$

- Em *todos* os estados do autômato A, incluímos uma transição vazia que desempilha $\bullet$ e vai para f, o único estado final:

$$\delta'(q_i, \varepsilon, \bullet) = \{(f, \varepsilon)\}$$

Não basta exibirmos esta construção. Precisamos mostrar que $V(A) = L(B)$. Mostramos que se A aceita uma palavra w, B também deve aceitar; e que se A *não* aceita w, B não pode aceitar[3].

Suponha que uma palavra w seja aceita por A. Isto significa que

$$(q_0, w, Z_0) \vdash_A^* (q, \varepsilon, \varepsilon), \tag{3.1}$$

[3]Poderíamos também mostrar que "se A aceita w, B aceita; e se B aceita w, A também aceita", e teríamos igualmente mostrado que $V(A) = L(B)$.

para algum estado q. O autômato B é idêntico ao autômato A, exceto por seus dois estados adicionais. Como temos $\bullet$ no fundo da pilha ao iniciar, e empilhamos Z_0 antes de entrar no estado q_0, temos

$$\begin{aligned}(q_{-1}, w, \bullet) &\vdash_B (q_0, w, Z_0\bullet) \\ &\vdash_B^* (q, \varepsilon, \bullet) && \text{(por 3.1 e Lema 3.34)} \\ &\vdash_B (f, \varepsilon, \varepsilon).\end{aligned}$$

O autômato B, portanto, chegará ao estado final.

Agora, suponha que *w não* seja aceita por A. Isto significa que ocorreu uma de duas possibilidades:

- a partir de q_0, o autômato A chega a um estado com elementos na pilha (ela não está vazia):

$$(q_0, w, Z_0) \vdash_A^* (q, \varepsilon, \alpha), \quad \alpha \in \Gamma^+.$$

- a partir de q_0, o autômato A chega a um estado onde não consegue continuar lendo, porque não há transições que permitam continuar:

$$(q_0, w, Z_0) \vdash_A^* (q, \beta, \alpha), \quad \alpha \in \Gamma^*.$$

No primeiro caso, o autômato B fará

$$\begin{aligned}(q_{-1}, w, \bullet) &\vdash_B (q_0, w, Z_0\bullet) \\ &\vdash_B^* (q, \varepsilon, \alpha\bullet),\end{aligned}$$

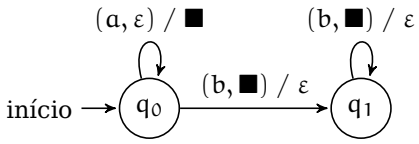
e o autômato B não poderá entrar no estado final f, porque isto só acontece quando é possível desempilhar $\bullet$; mas $\bullet$ só pode ser desempilhado depois de todos os outros símbolos, e a sequência α ainda está na pilha.

No segundo caso, B fará

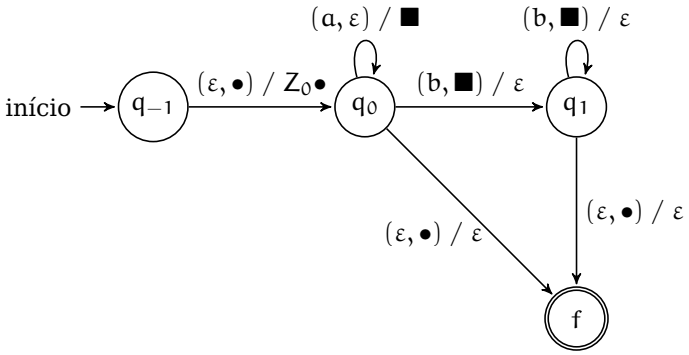
$$\begin{aligned}(q_{-1}, w, \bullet) &\vdash_B (q_0, w, Z_0\bullet) \\ &\vdash_B^* (q, \beta, \alpha\bullet), \quad \alpha \in \Gamma^*,\end{aligned}$$

mas não há transições em q para continuar (as únicas transições adicionadas em B são a inicial, que não pode ser usada, e as finais, que só podem ser usadas com a pilha vazia). Caso $\alpha = \varepsilon$, o autômato poderá passar para f, mas ali não poderá terminar de ler a palavra, porque não há transições possíveis saindo de f. □

Exemplo 3.39. A figura a seguir mostra um autômato que reconhece a linguagem $a^n b^n$, usando o critério de parada de pilha vazia.



O autômato equivalente, com os novos estados q_{-1} e f, e usando critério de parada de estado final, é mostrado a seguir. O símbolo ■ é usado para contar a quantidade de as, e o símbolo • é usado para marcar o fundo da pilha e mudar para o estado final.

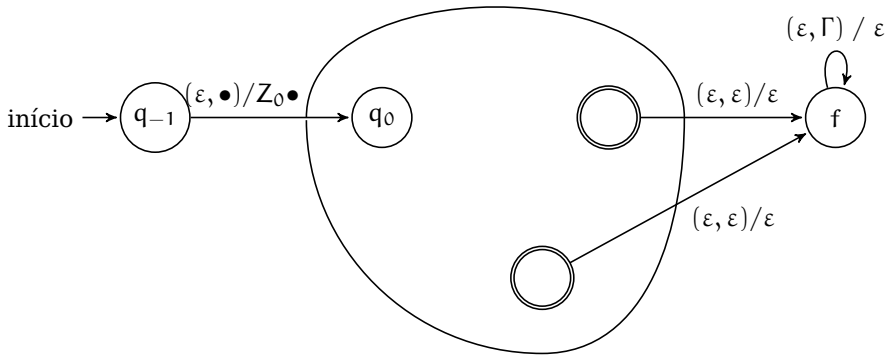


◀

Teorema 3.40. *Seja* L *a linguagem de um autômato com pilha usando critério de estado final. Então há um autômato com pilha, usando critério de parada de pilha vazia, que reconhece* L.

Demonstração. (incompleta!) Se B é um autômato com pilha que para em estado final, podemos construir A, que aceita a mesma linguagem, e para com pilha vazia.

- Criamos um novo estado f. Este será o estado onde o autômato parará com a pilha vazia.
- Incluímos em cada estado que era final em B uma transição vazia indo ao estado f (se o estado era final, o autômato pode ir para f).
- Em f, incluímos transições que permitem desempilhar, repetidamente, qualquer símbolo da pilha.
- Para evitar que o autômato pare antes do momento correto com a pilha vazia, criamos um novo estado inicial q_{-1}. Dali o autômato inicia com um símbolo especial • na pilha, empilha o símbolo inicial de B e passa para q_0.



Primeiro afirmamos que *se o autômato* B *para em estado final após ler uma cadeia* w*, o autômato* A *para no estado* f *com a pilha vazia*. Demonstramos. Suponha que $(q_0, w, Z_0) \vdash_B (q, \varepsilon, \alpha)$. Ou seja, B para no estado $q \in F$, e deixa uma sequência α de elementos na pilha (como o critério é estado final, a pilha pode não estar vazia). Então o autômato A, começando com $\bullet$ na pilha e no estado q_{-1}, realiza

$$\begin{aligned}(q_{-1}, w, \bullet) &\vdash_A^* (q_0, w, Z_0\bullet)\\ &\vdash_A^* (q, \varepsilon, \alpha\bullet)\\ &\vdash_A (f, \varepsilon, \alpha\bullet)\\ &\vdash_A^* (f, \varepsilon, \varepsilon),\end{aligned}$$

e A deverá parar (entrar na configuração de pilha vazia, $(f, \varepsilon, \varepsilon)$) depois de ler a mesma palavra.

Mais ainda, como $\bullet$ estará no fundo da pilha desde a entrada em q_0, o autômato não parará antes de entrar em f, o único lugar onde pode desempilhar $\bullet$.

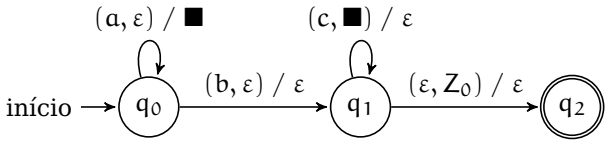
Agora afirmamos que *se o autômato* A *para com a pilha vazia depois de ler a cadeia* w*, então ele estava no estado* f*, e o autômato* B *teria parado em estado final se tivesse lido a mesma cadeia*.

Suponha que A tenha lido a palavra w e chegado a uma configuração de pilha vazia. Isto só é possível se o último estado for f, porque $\bullet$ é empilhado no primeiro passo, e só é desempilhado em f. Assim, os passos realizados por A foram necessariamente

$$\begin{aligned}(q_{-1}, w, \bullet) &\vdash_A^* (q_0, w, Z_0\bullet)\\ &\vdash_A^* (q, \varepsilon, \alpha\bullet) && \text{(algum } q \in F)\\ &\vdash_A (f, \varepsilon, \alpha\bullet)\\ &\vdash_A^* (f, \varepsilon, \varepsilon),\end{aligned}$$

Mas esta sequência de configurações inclui a sequência $(q_0, w, Z_0\bullet) \vdash_A^* (q, \varepsilon, \alpha\bullet)$, com $q \in F$, e como $\bullet$ nunca é empilhado ou desempilhado nas transições de B, podemos dizer também que $(q_0, w, Z_0) \vdash_A^* (q, \varepsilon, \alpha)$, e portanto B para em estado final depois de ler w. □

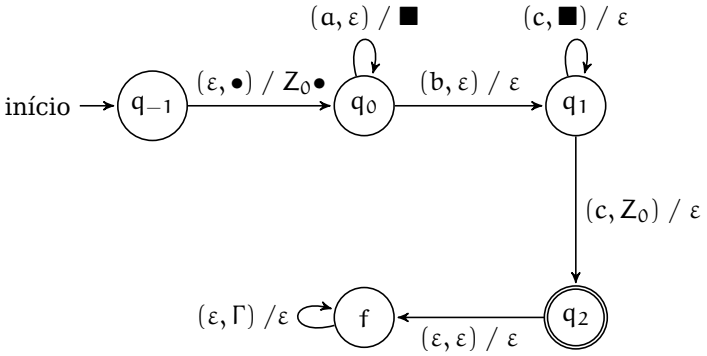
Exemplo 3.41. A figura a seguir mostra um autômato que reconhece a linguagem $a^n b c^n$, usando o critério de parada de estado final.



Ou seja, $(Q = \{q_0, q_1, q_2\}, \Sigma = \{a, b, c\}, \delta, q_0, \{q_2\})$, com

$$\begin{aligned}
\delta(q_0, a, \varepsilon) &= \{(q_0, \blacksquare)\} \\
\delta(q_0, b, \varepsilon) &= \{(q_1, \varepsilon)\} \\
\delta(q_1, c, \blacksquare) &= \{(q_1, \varepsilon)\} \\
\delta(q_1, \varepsilon, Z_0) &= \{(q_2, \varepsilon)\}
\end{aligned}$$

O autômato equivalente, com o novo estado e f, e usando critério de parada de pilha vazia, é mostrado a seguir. O símbolos $\bullet$ e Z_0são usados para garantir que a pilha só ficará vazia depois da entrada em q_2.



O autômato é, portanto, $(Q = \{q_{-1}, q_0, q_1, q_2, f\}, \Sigma = \{a, b, c\}, \delta, q_{-1}, \{q_2\})$, com

$$\begin{aligned}
\delta(q_{-1}, \varepsilon, \bullet) &= \{(q_0, Z_0\bullet)\} \qquad & \delta(q_1, \varepsilon, Z_0) &= \{(q_2, \varepsilon)\} \\
\delta(q_0, a, \varepsilon) &= \{(q_0, \blacksquare)\} & \delta(q_2, \varepsilon, \varepsilon) &= \{(f, \varepsilon)\} \\
\delta(q_0, b, \varepsilon) &= \{(q_1, \varepsilon)\} & \delta(f, \varepsilon, \Gamma) &= \{(f, \varepsilon)\} \\
\delta(q_1, c, \blacksquare) &= \{(q_1, \varepsilon)\}
\end{aligned}$$

Mesmo q_2 estando marcado como "final" (porque foi desenhado com duas circunferências concêntricas), isto não tem significado para este autômato, porque ele usa como critério de parada a pilha vazia. ◀

Teorema 3.42. *Toda linguagem livre de contexto é reconhecida por algum autômato não-determinístico com pilha.*

A demonstração deste Teorema é construtiva. O método que usamos na demonstração é um algoritmo usado internamente por compiladores: o projetista de linguagem de programação define uma gramática, e depois esta gramática é convertida em um autômato com pilha que reconhece a linguagem da gramática.

Depois de mostrar como construir o autômato, provaremos que se a gramática gera uma palavra, o autômato a aceita. Faltará mostrar que, se o autômato aceita uma palavra, ela é gerada pela gramática.

Demonstração. Primeiro mostramos como, a partir de uma gramática, construir um autômato com pilha. Depois demonstramos por indução no tamanho das derivações da gramática, que o autômato reconhece as palavras derivadas; em seguida, mostramos por indução na quantidade de passos do autômato que, se ele reconhece uma palavra, ela é derivável a partir da gramática.

Suponha, sem perda de generalidade, que a gramática está na forma normal de Greibach, $G = (N, \Sigma, P, S)$.

O autômato equivalente à gramática será

$$\big(Q = \{q\}, \Sigma, N, \delta, q, S, \emptyset\big)$$

O autômato tem um único estado, q, e inicialmente empilhará S (e não Z_0). O alfabeto da pilha contém exatamente os símbolos não-terminais.

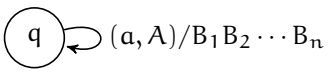
Para cada regra tendo A à esquerda,

$$A ::= aB_1B_2 \dots B_n$$

crie uma transição

$$\delta(q, a, A) = \{(q, B_1B_2 \dots B_n)\},$$

como no diagrama a seguir.



O autômato resultante terá várias transições indo do estado q para ele mesmo, mas aceitando, empilhando e desempilhando símbolos diferentes.

Passamos à demonstração de que esta construção de fato aceita as palavras da gramática..

Seja w uma palavra gerada pela gramática.

A base é com uma derivação de um único passo: somente uma regra foi usada, e ela só pode ter sido

$$S ::= a \quad \text{ou} \quad S ::= \varepsilon.$$

No primeiro caso, o autômato tem uma transição $\delta(q, a, S) = \{(q, \varepsilon)\}$; a pilha ficará vazia e o autômato aceitará a palavra. No segundo caso, há uma transição $\delta(q, \varepsilon, S) = \{(q, \varepsilon)\}$, e novamente, o autômato parará com a pilha vazia.

A hipótese de indução é de que, se a derivação mais à esquerda de uma palavra tem k passos, então o autômato da construção que fizemos reconhece a palavra.

Presuma agora que a derivação mais à esquerda tem $k+1$ passos. Como a gramática está na forma normal de Greibach, o primeiro passo da derivação é

$$S \Rightarrow aA_1A_2 \dots A_n.$$

Da forma como construímos, há uma transição no autômato que lê o símbolo a no estado S:

$$\delta(q, a, S) = \{(q, A_1A_2 \dots A_n)\}$$

Agora, a derivação do resto da palavra tem k passos, e pela hipótese de indução o autômato reconhecerá o resto da palavra. □

Exemplo 3.43. A gramática a seguir

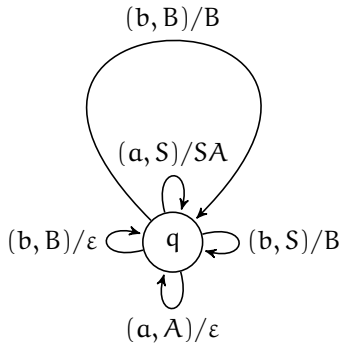
$$\begin{aligned} S &::= aSA \mid bB \\ A &::= a \\ B &::= bB \mid b \end{aligned}$$

gera a linguagem $a^n b^+ b + a^n$.

Para construir o autômato correspondente, criamos uma a uma as regras:

$$\begin{array}{ll} S ::= aSA & \delta(q, a, S) = (q, SA) \\ S ::= bB & \delta(q, b, S) = (q, B) \\ A ::= a & \delta(q, a, A) = (q, \varepsilon) \\ B ::= bB & \delta(q, b, B) = (q, B) \\ B ::= b & \delta(q, b, B) = (q, \varepsilon) \end{array}$$

O resultado é o autômato a seguir.



A gramática gera, por exemplo, a palavra aabbaa:

$$\begin{aligned} S &\Rightarrow aSA \\ &\Rightarrow aaSAA \\ &\Rightarrow aabBAA \\ &\Rightarrow aabbAA \\ &\Rightarrow aabbaA \\ &\Rightarrow aabbaa \end{aligned}$$

A mesma palavra é reconhecida pelo autômato. A seguir, em cada linha, a parte à esquerda representa os símbolos lidos, e a parte à direita, a pilha.

$$\begin{aligned} \varepsilon &\quad \boxed{S} \\ a &\quad \boxed{SA} \\ aa &\quad \boxed{SAA} \\ aab &\quad \boxed{BAA} \\ aabb &\quad \boxed{AA} \\ aabba &\quad \boxed{A} \\ aabbaa &\quad \boxed{\varepsilon} \end{aligned}$$

(mesmo que "$\boxed{\ }$")

Chegando ao final da entrada com a pilha vazia, o autômato aceitará a palavra. ◀

Teorema 3.44. *A linguagem de qualquer autômato não-determinístico com pilha é livre de contexto.*

Dado um autômato com pilha, construiremos uma gramática livre de contexto que reconhece a linguagem do autômato.

Demonstração. Um conceito central na demonstração será usado ao construir os não-terminais da gramática. Estes serão denotados entre colchetes,

e

$$[\, q, A, r \,] \Rightarrow^* w$$

significará

$$(q, w, A) \vdash^* (r, \varepsilon, \varepsilon).$$

Seja $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \emptyset)$ um autômato com pilha usando critério de parada de pilha vazia. Construiremos uma gramática $G = (N, \Sigma, P, S)$.

- além do símbolo inicial S, incluímos em N variáveis $[qXr]$, para todos os possíveis estados q, r e símbolos de pilha X.

- para todos os estados q, incluímos $S ::= [q_0 Z_0 q]$ nas regras, para garantir que a gramática gere todas as cadeias que começariam de q_0, desempilhariam o símbolo de pilha inicial Z_0 e terminariam em q.

- Se $\delta(p, a, X)$ contém $(q, Y_1 Y_2 \dots Y_k)$, com $a \in \Sigma_\varepsilon$ e $k \geq 0$, então para toda sequência possível de estados $q^1, q^2, \dots, q^k$ (note que o índice acima de cada estado significa que é i i-ésimo *desta sequência*, e nada tem a ver com o nome do estado), inclua a regra

  $$[pXq^k] ::= a[qY_1q^1][q^1Y_2q^2]\cdots[q^{k-1}Y_kq^k].$$

  Esta regra é usada para permitir gerar as palavras que, a partir de p, terminaria em q, desempilhando X. Se a função de transição inclui $\delta(p, a, X) = \{ \cdots \ (q, Y_1 Y_2 \dots Y_k) \ \cdots \}$, então o que a regra faz é *detalhar o caminho que o autômato faria* de p até q (se $\delta(p, a, X)$ empilha $Y_1 \dots Y_k$, consideramos todas as formas possíveis de desempilhar os símbolos $Y_1, \dots, Y_k$, chegando em q^k). .

Provaremos agora que as duas afirmações a seguir são equivalentes:

i) $[qXr] \Rightarrow^*_G w$

ii) $q, w, X \vdash^*_A (r, \varepsilon, \varepsilon)$

Primeiro, $(i) \Rightarrow (ii)$. A demonstração é na quantidade de passos de substituição na derivação.

A base se dá com um único passo: suponha que a gramática produz a palavra a partir de $[qXr]$ com uma única regra $[qXr] ::= w$. Mas uma regra assim só pode existir se havia uma transição

$$\delta(q, w, X) = \{\cdots \ (r, \varepsilon) \ \cdots\}$$

no autômato. Mas se esta transição existe, então

$$(q, w, X) \vdash (r, \varepsilon, \varepsilon).$$

A hipótese de indução determina que, se $[qXr] \Rightarrow^{<k} w$ (ou seja, com menos que k substituições), então $(q, w, X) \vdash^* (r, \varepsilon, \varepsilon)$.

Começamos o passo de indução: suponha que $[qXr] \Rightarrow^k w$. Observamos que

$$[q, X, r] \Rightarrow^k w$$

pode ser reescrita:

$$[q, X, r] \Rightarrow a[q^1 B_1 q^2][q^2 B_2 q^3] \cdots [q^k B_k q^{k+1}] \Rightarrow^{k-1} w, \tag{3.2}$$

para algum $a \in \Sigma$. Isto significa que podemos dividir w em subcadeias

$$w = a x_1 x_2 \cdots x_k,$$

de forma que a sequência $a x_1 x_2 \cdots x_k$ seja gerada por $[q, X, r]$, como em 3.2:

$$[q^j B_j q^{j+1}] \Rightarrow^{<K} x_j, \quad 1 \leq j \leq k.$$

Agora, pela hipótese de indução, para cada $1 \leq j \leq k$, temos que

$$[q^j B_j q^{j+1}] \Rightarrow^{<K} x_j \quad \text{implica em} \quad (q^j, x_j, B_j) \vdash^* (q^{j+1}, \varepsilon, \varepsilon)$$

Então, pelo Lema 3.34,

$$(q^j, x_j, B_j \cdots B_k) \vdash^* (q^{j+1}, \varepsilon, B_{j+1} \cdots B_k)$$

Voltamos à configuração (q, w, X). Temos

$$\begin{aligned}(q, w, X) &\vdash (q^1, x_1 x_2 \cdots x_n, B_1 B_2 \cdots B_k) \\ &\vdash (q^2, x_2 \cdots x_n, B_2 \cdots B_k) \\ &\vdots \\ &\vdash^* (q^{k+1}, \varepsilon, \varepsilon).\end{aligned}$$

Agora, $(ii) \Rightarrow (i)$. A demonstração é na quantidade de passos realizados pelo autômato ao reconhecer a palavra.

Para a base de indução, tratamos de palavras reconhecidas em um passo pelo autômato – ou seja, $(q_0, w, Z_0) \vdash (q, \varepsilon, \varepsilon)$. Isto só pode ocorrer se $w = \varepsilon$ ou $w = a \in \Sigma$ (ou seja, w não pode ter comprimento maior que um), e também se havia uma transição

$$\delta(q_0, a, Z_0) = \{\cdots \ (q, \varepsilon) \ \cdots\}$$

no autômato. Mas se esta transição existia, então a construção da gramática

incluiu as regras

$$S ::= [q_0 Z_0 q]$$
$$[q_0, Z_0, q] ::= a$$

que geram a palavra.

Como hipótese de indução, presumimos que

$$(q, w, X) \vdash^{<k} (r, \varepsilon, \varepsilon) \quad \text{implica em} \quad [qXr] \Rightarrow^* w.$$

Seja w uma palavra reconhecida pelo autômato a partir do estado q, com X na pilha, em k passos: $(q, w, X) \vdash^k (r, \varepsilon, \varepsilon)$. Podemos separar w em

$$w = w_0 w'.$$

onde w_0 é a subcadeia lida no primeiro passo.

Dividimos a computação realizada pelo autômato em um primeiro passo e o resto dos passos. No primeiro passo, o autômato reconhece w_0. Nos outros, reconhece w'.

$$(q, w, X) \vdash (q^1, w', B_1 B_2 \cdots B_n) \vdash^{k-1} (r, \varepsilon, \varepsilon),$$

onde $B_1 \cdots B_n$ é uma sequência, possivelmente vazia, de símbolos de pilha.

Separemos a computação em duas partes, e podemos usar a hipótese de indução nas duas:

- $(q, w, X) \vdash (q^1, w', \alpha)$ implica que $[qXq'] \Rightarrow^* w_0$.

- No início da segunda parte da computação, a configuração é

  $$(q^1, w', B_1 B_2 \cdots B_n)$$

  Como cada um dos B_i será removido da pilha na ordem em que aparecem, podemos chamar de q^i o estado em que o autômato estava quando B_i foi desempilhado, e w_i a subcadeia aceita entre q^i e q^{i+1}. Portanto, reescrevemos:

  $$(q^1, w_1 \ldots w_n, B_1 B_2 \cdots B_n) \vdash^{k-1} (r, \varepsilon, \varepsilon).$$

  A computação foi dividida em n pedaços, cada um com menos de k

passos. Observamos que

$$\begin{aligned}(q^1, w_1 \dots w_n, B_1 B_2 \cdots B_n) &\vdash^{<k} (q^2, w_2 \dots w_n, B_2 \cdots B_n) \\ &\vdash^{<k} (q^3, w_3 \dots w_n, B_3 \cdots B_n) \\ &\vdots \\ &\vdash^{<k} (r, \varepsilon, \varepsilon).\end{aligned}$$

Isto implica, pela hipótese de indução, que

$$\begin{aligned}[q^1 B_1 q^2] &\Rightarrow^* w_1, \\ [q^2 B_2 q^3] &\Rightarrow^* w_2, \\ &\vdots \\ [q^n B_n q^{n+1}] &\Rightarrow^* w_n\end{aligned}$$

E claramente a gramática gera $w_0 w_1 \dots w_n = w$.

Finalmente, observamos que, como

$$[q_0, Z_0, p] \Rightarrow^* w \quad \text{se e somente se} \quad (q_0, w, Z_0) \vdash^* (p, \varepsilon, \varepsilon),$$

as linguagens representadas pelo autômato e pela gramática são as mesmas. □

3.2.1 Determinismo

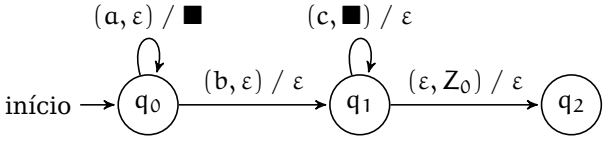
Ao contrário do que ocorre com autômatos finitos, os autômatos determinísticos com pilha são *menos* expressivos que os não-determinísticos.

Definição 3.45 (autômato com pilha determinístico)**.** Um autômato com pilha é *determinístico* quando

i) para cada tripla $(q, a, x) \in Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$, a função de transição leva a no máximo um único elemento;

ii) $\delta(q, \varepsilon, \alpha) \neq \emptyset$ implica em $\delta(q, a, \alpha) = \emptyset$ para todo $a \in \Sigma$ (ou seja, o autômato não precisa decidir entre fazer uma transição vazia ou seguir consumindo um símbolo; ele sempre pode decidir olhando para o topo da pilha). ♦

Exemplo 3.46. O autômato com pilha a seguir é determinístico.



A função de transição obedece as restrições impostas para autômatos determinísticos:

$$\begin{aligned}\delta(q_0, a, \varepsilon) &= \{(q_0, \blacksquare)\}\\ \delta(q_0, b, \varepsilon) &= \{(q_1, \varepsilon)\}\\ \delta(q_1, c, \blacksquare) &= \{(q_1, \varepsilon)\}\\ \delta(q_1, \varepsilon, Z_0) &= \{(q_2, \varepsilon)\}\end{aligned}$$

As duas transições saindo de q_0 são com símbolos diferentes (a, b); só há uma saindo de q_1 com c; e como há uma para (q_1, ε, Z_0), não existe nenhuma outra para (q_1, x, Z_0), para nenhum $x \in \Sigma$. ◀

Definição 3.47. Uma linguagem livre de contexto é *determinística* se é reconhecida por um autômato determinístico com pilha. ♦

Exemplo 3.48. A linguagem $a^n b c^n$ é livre de contexto e determinística, porque é reconhecida pelo autômato do Exemplo 3.46. ◀

Teorema 3.49. *Há linguagens livres de contexto que não são determinísticas.*

Exemplo 3.50. A linguagem das palíndromas sobre qualquer alfabeto com mais de um símbolo é não-determinística. A demonstração exige mais do que pretendemos para este texto, mas a indução pode ser dada: um autômato com pilha poderia empilhar os mesmos símbolos que encontra ao ler a cadeia w, e depois desempilhar para ler w^R. No entanto, ele não saberia quando começar a desempilhar, porque não sabe onde fica o meio da palavra. ◀

Teorema 3.51. *Se um autômato com pilha aceita uma linguagem* L *usando critério de parada de pilha vazia, então a linguagem* L *é gerada por uma gramática não-ambígua (ou seja,* L *não é inerentemente ambígua).*

Demonstração. (Idéia apenas)

O método de conversão de autômato em gramática usado no Teorema 3.44 produz gramáticas não-ambíguas quando usados em autômatos determinísticos. □

Teorema 3.52. *Se um autômato com pilha aceita uma linguagem* L *usando critério de parada de estado final, então a linguagem* L *é gerada por uma gramática não-ambígua (ou seja,* L *não é inerentemente ambígua).*

Teorema 3.53. *Não existe autômato determinístico com pilha que aceite uma linguagem livre de contexto inerentemente ambígua.*

Teorema 3.54. *O complemento de uma linguagem livre de contexto determinística é, também, uma linguagem livre de contexto determinística.*

Teorema 3.55. *Linguagens livres de contexto determinísticas* não *são fechadas para interseção, união ou reversão.*

Demonstração. Para interseção, considere

$$\begin{aligned} A &= \{a^i b^j c^k \ : \ i = j\} \\ B &= \{a^i b^j c^k \ : \ j = k\} \\ A \cap B &= \{a^n b^n c^n\} \end{aligned}$$

Mas a última, $A \cap B$, não é livre de contexto.

Para união, considere duas linguagens sobre o alfabeto $\Sigma = \{a, b, c\}$. $L_1 = a^i b^j c^k, i \neq j$ e $L_2 = a^i c^j b^k, j \neq k$. As duas são livres de contexto e determinísticas, mas a união delas seria $a^n b^n c^n$, que sequer é livre de contexto.

Para reversão, considere a linguagem

$$L = \{1a^i b^j c^k \ : \ i = j\} \ \cup \ \{2a^i b^j c^k \ : \ j = k\}$$

Esta linguagem é livre de contexto determinística, porque primeiro símbolo de cada palavra pode ser usado por um autômato para decidir se o resto da palavra é do primeiro tipo ($a^i b^j c^k \ : \ i = j$) ou do segundo ($a^i b^j c^k \ : \ j = k$), mas a linguagem reversa não é. □

O não-determinismo não implica necessariamente em ambiguidade.

Teorema 3.56. *Há linguagens livres de contexto que são não ambíguas e também são não determinísticas.*

Demonstração. Por exemplo, linguagens de palavras palíndromas sobre dois símbolos, como $\{s \in \{a, b\}^* \ : \ s = s^R\}$.

A linguagem não é inerentemente ambígua: considere a gramática a seguir, que claramente a gera.

$$\begin{aligned} S &::= aSa \\ S &::= bSb \\ S &::= \varepsilon \end{aligned}$$

Existe somente uma derivação possível para cada palavra, porque existe um único não-terminal na linguagem, e ele aparece no máximo uma única vez no lado direito de cada produção.

No entanto, não existe autômato determinístico com pilha que a reconheça. □

Teorema 3.57. *A linguagem* $(a+b)^* - ww$*, onde* $w \in (a+b)^*$ *(ou seja, cadeias de* a*s e* b*s sem restrição, a não ser que a primeira metade da palavra não pode ser igual à segunda)* não é *livre de contexto determinística.*

Demonstração. Se a linguagem fosse determinística, pelo Teorema 3.54 seu complemento também seria. O complemento da linguagem é o conjunto de cadeias de as e bs onde a primeira metade é exatamente igual à segunda. Mas esta linguagem não é determinística. □

Teorema 3.58. *A interseção de uma linguagem livre de contexto determinística com uma linguagem regular é uma linguagem livre de contexto determinística.*

3.3 Propriedades de Linguagens Livres de Contexto

Teorema 3.59. *Linguagens livres de contexto não são fechadas para interseção, complemento ou diferença.*

Demonstração. Sejam $L_1 = \{a^n b^n c^m\}$ e $L_1 = \{a^n b^m c^m\}$, ambas livres de contexto; então

$$L_1 \cap L_2 = \{a^k b^k c^k\},$$

que não é livre de contexto.

Suponha que valha o fecho para complemento. Então, para quaisquer L_1, L_2 livres de contexto,

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

e teríamos obtido a interseção de operações fechadas. Isso contradiz a primeira parte.

Para a diferença, considere alguma linguagem L livre de contexto sobre um alfabeto Σ. Então

$$\overline{L} = \Sigma^* - L,$$

e obteríamos o complemento usando a diferença, que supusemos fechada. □

Teorema 3.60. *Linguagens livres de contexto são fechadas para união, concatenação, fecho e reversão.*

Demonstração. Sejam duas linguagens livres de contexto quaisquer, geradas por duas gramáticas. Suponha, sem perda de generalidade, que A é o símbolo inicial da primeira gramática, e B o da segunda, e suponha também que elas não tem símbolos não-terminais em comum.

União: a gramática $S ::= A \mid B$, com as regras de A e B, gera a união das linguagens.

Concatenação: a gramática $S ::= AB$ gera a linguagem de A concatenada com a de B.

Fecho: a gramática $S ::= SA \mid \varepsilon$ gera o fecho de A.

Reversão: Há uma gramática na forma normal de Chomsky que aceita a linguagem da gramática A. Reescreva $X ::= YZ$ como $X ::= ZY$ em todas as regras. □

Teorema 3.61. *Sejam* L *uma linguagem livre de contexto e* R *uma linguagem regular. Então* $L \cap R$ *é livre de contexto.*

Damos apenas um rascunho da demonstração; o Exercício 42 pede o detalhamento.

Demonstração. (Rascunho) Sejam P um autômato com pilha e A um autômato finito. Construa um autômato cujos estados sejam o produto cartesiano dos estados dos dois; os estados finais são os pares (p, q) onde p é estado final de P e q é estado final de A. O resultado será um autômato com pilha, que aceita as palavras que os dois aceitariam. □

Teorema 3.62. *Linguagens livres de contexto são fechadas para diferença com linguagem regular: se* L *é livre de contexto e* R *regular, então* $L - R$ *é livre de contexto.*

Demonstração. Se L é livre de contexto e R é regular, então

$$L - R = L \cap \overline{R}.$$

Como $\overline{R}$ é regular, e a interseção de L com linguagem regular é livre de contexto, e está demonstrado o Teorema. □

3.4 Lema do Bombeamento

Há um Lema do Bombeamento também para linguagens livres de contexto, semelhante em espírito àquele para linguagens regulares.

O Lema 3.63 é usado na demonstração do Lema do Bombeamento; sua demonstração é pedida no Exercício 29.

Lema 3.63. *Seja* G *uma gramática livre de contexto na forma normal de Chomsky. Se* G *gera uma palavra* s *e o maior caminho, na árvore de derivação, da raiz a uma folha é* n*, então o comprimento de* s *é menor ou igual a* 2^{n-1}.

O lema do bombeamento diz que, se uma palavra é longa o suficiente, houve a necessidade de repetir não-terminais com duas produções diferentes, $A \to BC$, $A \to DE$. Mas isto significa que ou B ou C levam a A (para que ele pudesse ser repetido), e podemos portanto gerar palavras repetindo

aquele trecho quantas vezes quisermos: basta, ao invés de usar $A \to DE$, usar novamente $A \to BC$.

Lema 3.64 (do bombeamento, para linguagens livres de contexto)**.** *Seja* L *uma linguagem livre de contexto. Então existe um número natural* p*, chamado de* comprimento de bombeamento*, tal que: para toda cadeia* $s \in L$*, com* $|s| \geq p$*, pode-se dividir* s *em partes,* $s = uvxyz$*, tal que*

i) $|vxy| < p$*;*

ii) $|vy| > 0$*;*

iii) $\forall i \geq 0$, $uv^i xy^i z \in L$.

Demonstração. Seja G uma gramática com m variáveis, e s uma palavra de comprimento maior ou igual que 2^m. Presuma, sem perda de generalidade, que G está na forma normal de Chomsky.

Pelo Lema 3.63, a árvore de derivação de s terá altura maior ou igual que $m + 1$. Além disso, em um caminho com $m + 1$ arestas há $m + 2$ nós. Neste caminho, exatamente um (a folha) pertence a Σ, e os nós internos pertencem a N, portanto há $m + 1$ nós com rótulos em N. Mas $|N| = m$, portanto, pelo princípio da casa dos pombos, o caminho contém dois nós com o mesmo rótulo.

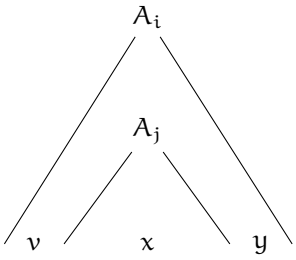
Denotaremos o caminho da raiz até a folha por $A_0, A_1, \ldots, A_{m+1}, x$, onde A_0 é a raiz e x é a folha. Sabemos, portanto, que existem dois rótulos iguais, $A_i = A_j, i \neq j$, e A_j está abaixo de A_i na árvore.

O caminho de A_i até a folha tem no máximo $m + 1$ arestas. Isso porque, se seguirmos o caminho da folha para cima, no máximo na $m + 1$-ésima aresta teremos um nó repetido. Este é A_i.

Sejam T_i e T_j as subárvores de A_i e A_j.

Seja w a cadeia gerada por A_j. Então a cadeia gerada por A_i contém x.

$$\begin{aligned} A_j &\Rightarrow^* x \\ A_i &\Rightarrow^* vA_jy \\ &\Rightarrow^* vxy \end{aligned}$$



Temos **(i)** $|vxy| \leq 2^m$, porque não existe caminho com mais de $k+1$ arestas em T_i (já selecionamos o maior caminho da árvore).

Agora, como a gramática está na forma normal de Chomsky, a produção usada em A_i era da forma $A_i \to BC$, portanto A_j está completamente contido na subárvore de B ou na de C. Como nenhuma das duas é vazia, temos **(ii)** $|vy| > 0$.

Observando as árvores e usando A para denotar tanto A_i como A_j, concluímos que

$$A \Rightarrow^* vAy$$
$$A \Rightarrow^* x$$

e portanto concluímos **(iii)**

$$A \Rightarrow^* v^i x y^i. \qquad \square$$

A seguir usamos o Lema do Bombeamento para mostrar que algumas linguagens não são livres de contexto.

Proposição 3.65. *A linguagem* $a^n b^n c^n$ *não é livre de contexto.*

Demonstração. Seja $s = a^n b^n c^n$. Então $s = uvxyz$, e

- $|v| > 0, |y| > 0$, e
- $|vxy| \leq k$
- $uv^p xy^p z \in L$, para todo $p \geq 0$.

Há dois casos a considerar:

- v e y contém um só tipo de símbolo. Mas então, em uv^2xy^2z teremos aumentado a quantidade desse símbolo, e não dos outros.
- v ou y contém dois símbolos diferentes. Mas em uv^2xy^2z haverá símbolos fora de ordem.

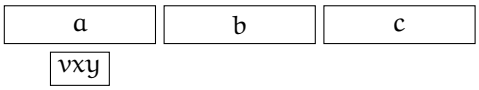
Assim, os dois únicos casos levam a contradição, e a linguagem não pode ser livre de contexto, porque não satisfaz o enunciado do Lema do Bombeamento. $\square$

Proposição 3.66. *A linguagem* $L = a^i b^j c^k$*, com* $i \leq j \leq k$ *não é livre de contexto.*

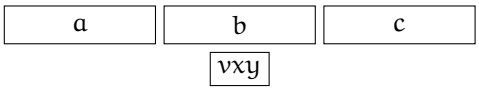
Demonstração. Suponha que a linguagem seja livre de contexto, e que, portanto, vale o Lema do Bombeamento. Seja p o comprimento de bombeamento para esta linguagem.

Agora, $s = a^p b^p c^p$ pertence à linguagem. De acordo com o Lema do Bombeamento, podemos dividir s em partes, $s = uvxyz$, valendo as afirmativas do enunciado do Lema.

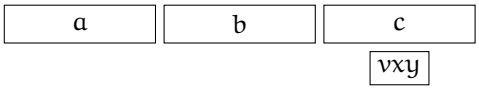
- Suponha que vxy esteja inteiramente em uma região da palavra onde só existe um tipo de símbolo. Então,
  - se vxy só contém as, então uv^2xy^2z é $a^{p+k}b^pc^p \notin L$;



  - se vxy só contém bs, então uv^2xy^2z é $a^pb^{p+k}c^p \notin L$;

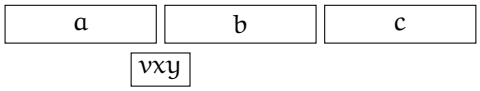


  - se vxy só contém cs, então uxz é $a^pb^pc^{p-k} \notin L$.

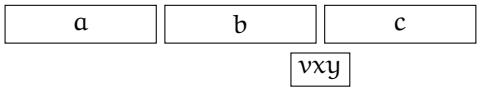


- Se vxy tem mais de um tipo de símbolo, então pode ter no máximo dois, porque $|vxy| = p$, e não há como conter todos os bs do meio com as à esquerda e bs à direita (só os bs já são p símbolos!) Assim, presumimos que vxy contém dois tipos de símbolos, e um deles deve ser b.

  Suponha que $vxy = aaa\cdots abbb\cdots b$ (não contém cs). Então a palavra bombeada uv^2xy^2z terá mais as ou bs do que cs, em desacordo com a definição da linguagem.



  Se vxy não contém as, de maneira similar, $vxy = bbb\cdots bccc\cdots c$, e a palavra bombeada uxz terá menos bs ou cs do que as.



Como não há como dividir a palavra de forma que possa ser bombeada produzindo outras palavras da linguagem, concluímos que L não é livre de contexto. □

Exercícios

Ex. 23 — Construa uma gramática e um autômato que reconheçam a linguagem $\{a^ib^jc^k \mid i > j \text{ ou } j = 2k\}$

Ex. 24 — Construa uma gramática para palavras sobre o alfabeto $\Sigma = \{a, b, c, \ldots, z\}$, que comecem e terminem com a mesma letra.

Ex. 25 — Construa uma gramática livre de contexto que gere a linguagem $L = \{a^n b^m c^k \mid k = m + n\}$.

Ex. 26 — Mostre que a gramática a seguir é ambígua.

$$S \to 0A \mid 1B$$
$$A \to 0AA \mid 1S \mid 1$$
$$B \to 1BB \mid 0S \mid 0$$

Ex. 27 — Modifique a gramática do Exemplo 3.17 para reconhecer a mesma linguagem, mas removendo a ambiguidade.

Ex. 28 — Prove o Teorema 3.13.

Ex. 29 — Prove o Lema 3.63.

Ex. 30 — Considere as linguagens $a^n b^m c^m d^n$ e $a^n b^m c^n d^m$. Uma delas é livre de contexto e a outra não – prove! (Ou seja, dê uma gramática livre de contexto para uma e use lema do bombeamento para provar que para a outra não há gramática livre de contexto). Depois explique, de forma intuitiva, a diferença entre elas.

Ex. 31 — Considere a linguagem $L = \{a^i b^j c^k \mid j = k = 2i\}$. L é regular? Livre de contexto? (Prove que sim ou que não)

Ex. 32 — Seja L a linguagem com zeros e uns, onde em toda palavra truncada (ou seja, toda subpalavra começando do caracter zero e indo até um pedaço da palavra), há mais zeros que uns. L é regular? Livre de contexto? (Prove que sim ou que não)

Ex. 33 — A linguagem sobre o alfabeto $\{0, 1\}$ onde a quantidade de zeros é *exatamente* duas vezes a quantidade de uns, e onde não há dois uns consecutivos, é regular? Livre de contexto? (Prove que sim ou que não)

Ex. 34 — A linguagem $a^i b^j c^k$, com $i < j$, e $i < k$ é livre de contexto? (Prove que sim ou que não)

Ex. 35 — A linguagem $a^i b^j c^k$, com $i < j$, e $i > k$ é livre de contexto? (Prove que sim ou que não)

Ex. 36 — A linguagem $\alpha\alpha^R\beta\beta^R$, com $\alpha, \beta \in \{0, 1\}^*$, é livre de contexto? (Prove que sim ou que não)

Ex. 37 — A linguagem ww, onde $s \in \{0,1\}^*$ é livre de contexto? (Prove que sim ou que não)

Ex. 38 — Tente obter o equivalente de expressões regulares para linguagens livres de contexto. Prove que suas expressões de fato representam as linguagens livres de contexto.

Ex. 39 — Se eu remover um número finito de cadeias de uma linguagem livre de contexto qualquer, ela continua livre de contexto?

Ex. 40 — Construa um autômato com pilha que reconheça números divisíveis por três. Seu autômato deve ter somente dois estados. (Mas ele fará muito trabalho na pilha!)
Dica: comece com uma transição que não lê símbolos da entrada, mas que empilha o número zero.

Ex. 41 — Seja Σ um alfabeto, com $|\Sigma| > 1$. Sabemos que a linguagem das palíndromas sobre Σ é livre de contexto.

a) Mostre que o complemento desta linguagem também é livre de contexto.

b) O resultado que você provou no item (a) significa que a linguagem das palíndromas é determinística (já que tanto ela como seu complemento são livres de contexto)?

Ex. 42 — Redija a demonstração do Teorema 3.61 com detalhes.

Ex. 43 — Prove o Teorema 3.29.

Ex. 44 — Seja X_n a linguagem de as e bs, onde cada palavra tem comprimento no máximo n, e tem quantidade de as igual à quantidade de bs. Determine a cardinalidade de X_n.

Ex. 45 — Seja G uma gramática livre de contexto na forma normal de Chomsky, tendo o conjunto de não-terminais N. Mostre que se G pode gerar alguma palavra de comprimento maior que $2^{|N|-1}$, então a linguagem de G é infinita.

Ex. 46 — Mostre uma gramática livre de contexto que gere o *complemento* da linguagem das palíndromas sobre $\Sigma = \{a, b\}$.

Ex. 47 — Prove que a classe de linguagens de autômatos com pilha que somente empilham *um* símbolo por transição é a mesma dos autômatos que empilham *vários* símbolos por transição.

Ex. 48 — Prove que toda linguagem livre de contexto sobre um alfabeto contendo um único símbolo é regular.

Capítulo 4

Linguagens Recursivas e Recursivamente Enumeráveis

4.1 Máquinas de Turing

A Máquina de Turing é um modelo de computação que generaliza os outros autômatos já estudados. Foi concebida por Alan Turing como uma forma de expressar o conceito de "computação".

Em cada passo de computação, uma máquina de Turing

1. verifica o estado atual. Se o estado for final, a máquina para aceitando a palavra;
2. lê um símbolo da fita;
3. dependendo do estado e do símbolo lido, escolhe outro símbolo, um novo estado e uma direção a andar. Se não existe escolha possível, para rejeitando a palavra;
4. escreve o novo símbolo sobre o símbolo lido;
5. muda para o novo estado;
6. move a cabeça de leitura e gravação na direção escolhida.

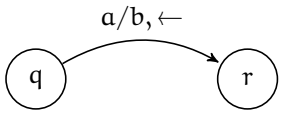
O único critério de parada para máquinas de Turing é, portanto, "estar em estado final", não dependendo do conteúdo da fita. Isso porque para máquinas de Turing não faz sentido "chegar ao final da entrada": como ela pode movimentar-se nas duas direções, pode chegar ao final da entrada várias vezes.

Definição 4.1 (máquina de Turing)**.** Uma *Máquina de Turing* é composta de

- um conjunto de estados Q;
- um alfabeto de entrada Σ;
- um alfabeto de fita Γ, tal que $\Sigma \subset \Gamma$. O alfabeto de fita contém um símbolo especial # (branco), que *não* pertence a Σ;
- uma função de transição $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow\}$;
- um estado inicial $q_0 \in Q$;
- um conjunto de estados finais, $F \subseteq Q$. ⧫

Usaremos diagramas para representar máquinas de Turing, assim como nos Capítulos anteriores. O diagrama a seguir representa a transição

$$\delta(q, a) = (r, b, \leftarrow)$$

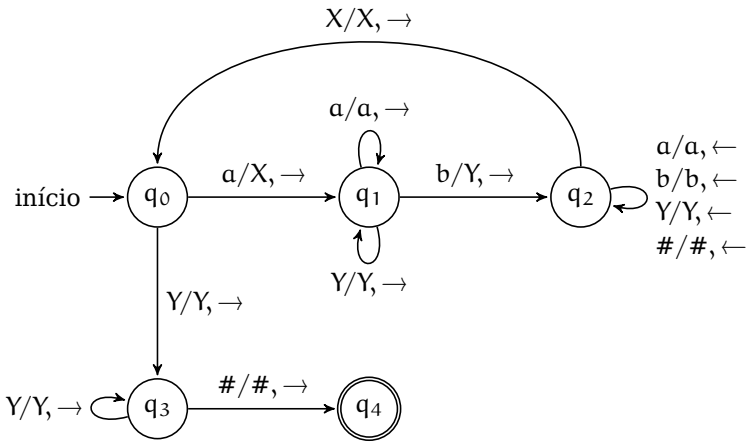


Da mesma forma que fizemos com autômatos finitos e autômatos com pilha, definiremos configuração para máquinas de Turing.

Definição 4.2 (configuração de máquina de Turing)**.** Se uma máquina de Turing está no estado q, o conteúdo de sua fita é $\alpha\beta$, e a cabeça de leitura e gravação está no primeiro símbolo de β, sua *configuração* é (α, q, β), ou $\alpha q \beta$, sem os parênteses. ⧫

Exemplo 4.3. Construímos a seguir uma máquina de Turing que reconhece cadeias da forma $a^n b^n$. Não precisaríamos de uma máquina de Turing para isso: no Capítulo 3 mostramos um autômato com pilha que reconhece esta linguagem. No entanto, a partir desta máquina de Turing construiremos outra, que reconhece $a^n b^n c^n$, que não é livre de contexto.

O autômato com pilha que reconhece $a^n b^n$ é bastante simples, mas a máquina de Turing equivalente será ligeiramente mais complexa, porque não tem uma pilha, e terá que escrever na própria fita. De maneira simplificada, a máquina percorrerá a entrada, da esquerda para a direita, sobrescrevendo em cada passada um a com um X e um b com um Y. Quando não houver mais as e bs, a máquina aceita a palavra.



A descrição da máquina de Turing é dada a seguir.

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2, q_3, q_4\} \\
\Sigma &= \{a, b\} \\
\Gamma &= \Sigma \cup \{\#, X, Y\} \\
F &= \{q_4\}
\end{aligned}
$$

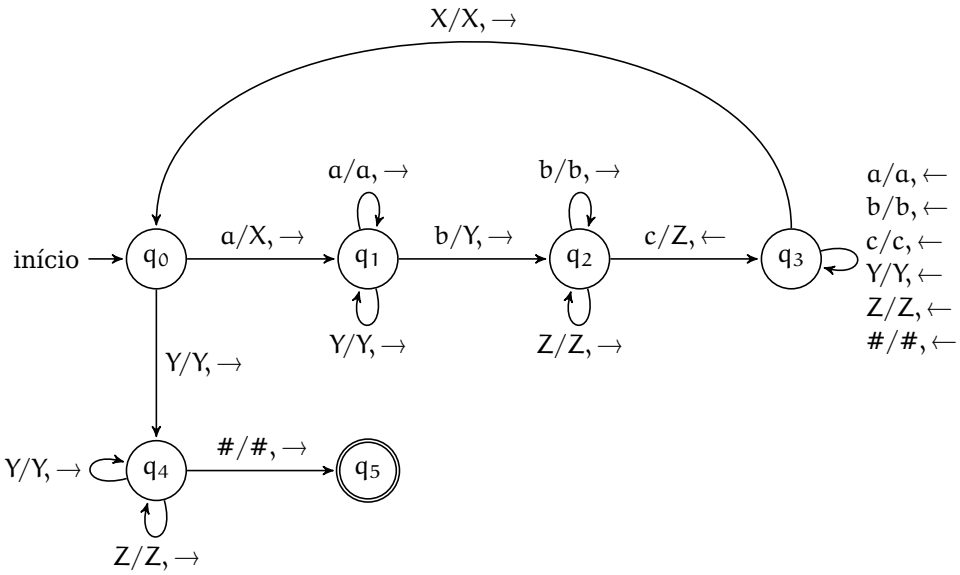
$$
\begin{aligned}
\delta(q_0, a) &= (q_1, X, \rightarrow) & \qquad \delta(q_2, b) &= (q_2, b, \leftarrow) \\
\delta(q_0, Y) &= (q_3, Y, \rightarrow) & \qquad \delta(q_2, Y) &= (q_2, Y, \leftarrow) \\
\delta(q_1, a) &= (q_1, a, \rightarrow) & \qquad \delta(q_2, X) &= (q_0, X, \rightarrow) \\
\delta(q_1, Y) &= (q_1, Y, \rightarrow) & \qquad \delta(q_2, \#) &= (q_2, \#, \leftarrow) \\
\delta(q_1, b) &= (q_2, Y, \rightarrow) & \qquad \delta(q_3, Y) &= (q_3, Y, \rightarrow) \\
\delta(q_2, a) &= (q_2, a, \leftarrow) & \qquad \delta(q_3, \#) &= (q_4, \#, \rightarrow)
\end{aligned}
$$

Exemplificamos o funcionamento com a cadeia $aabb$. Para maior clareza, representaremos os estados dentro das configurações em caixas (por exemplo, $\boxed{q_j}$).

$$
\begin{array}{ll}
\boxed{q_0}\,aabb\# & \vdash XXY\boxed{q_1}\,b\# \\
\vdash X\boxed{q_1}\,abb\# & \vdash XXYY\boxed{q_2}\,\# \\
\vdash Xa\boxed{q_1}\,bb\# & \vdash XXY\boxed{q_2}\,Y\# \\
\vdash XaY\boxed{q_2}\,b\# & \vdash XX\boxed{q_2}\,YY\# \\
\vdash Xa\boxed{q_2}\,Yb\# & \vdash X\boxed{q_2}\,XYY\# \\
\vdash X\boxed{q_2}\,aYb\# & \vdash XX\boxed{q_0}\,YY\# \\
\vdash \boxed{q_2}\,XaYb\# & \vdash XXY\boxed{q_3}\,Y\# \\
\vdash X\boxed{q_0}\,aYb\# & \vdash XXYY\boxed{q_3}\,\# \\
\vdash XX\boxed{q_1}\,Yb\# & \vdash XXYY\#\boxed{q_4}
\end{array}
$$

A máquina de Turing percorreu a fita três vezes da esquerda para a direita: na primeira, marcou o primeiro a e o primeiro b; na segunda, mais um a e um b; na terceira, verificou que não havia mais símbolos a e b para marcar, e aceitou a palavra. ◀

Exemplo 4.4. A máquina de Turing do Exemplo 4.3 pode ser facilmente modificada para aceitar a linguagem $a^n b^n c^n$, como mostramos a seguir. Um novo estado foi adicionado, onde a máquina troca cs por Zs. O funcionamento é semelhante ao da máquina original.



Fica claro que a máquina pode também ser modificada para aceitar palavras da forma $a^n b^n c^n d^n e^n \cdots$.

A descrição da máquina de Turing é dada a seguir.

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2, q_3, q_4, q_5\} \\
\Sigma &= \{a, b, c\} \\
\Gamma &= \Sigma \cup \{\#, X, Y, Z\} \\
F &= \{q_5\}
\end{aligned}
$$

$$
\begin{aligned}
\delta(q_0, a) &= (q_1, X, \rightarrow) & \delta(q_3, b) &= (q_3, b, \leftarrow) \\
\delta(q_0, Y) &= (q_4, Y, \rightarrow) & \delta(q_3, c) &= (q_3, c, \leftarrow) \\
\delta(q_1, a) &= (q_1, a, \rightarrow) & \delta(q_3, Y) &= (q_3, Y, \leftarrow) \\
\delta(q_1, Y) &= (q_1, Y, \rightarrow) & \delta(q_3, Z) &= (q_3, Z, \leftarrow) \\
\delta(q_1, b) &= (q_2, Y, \rightarrow) & \delta(q_3, \#) &= (q_3, \#, \leftarrow) \\
\delta(q_2, b) &= (q_2, b, \rightarrow) & \delta(q_3, X) &= (q_0, X, \rightarrow) \\
\delta(q_2, Z) &= (q_2, Z, \rightarrow) & \delta(q_4, Y) &= (q_4, Y, \rightarrow) \\
\delta(q_2, c) &= (q_3, Z, \leftarrow) & \delta(q_4, Z) &= (q_4, Z, \rightarrow) \\
\delta(q_3, a) &= (q_3, a, \leftarrow) & \delta(q_4, \#) &= (q_5, \#, \rightarrow)
\end{aligned}
$$

O funcionamento é semelhante ao da máquina do Exemplo 4.3. ◄

4.1.1 Linguagens recursivas e recursivamente enumeráveis

Os autômatos estudados nos Capítulos anteriores movem-se apenas em uma direção, e sempre param quando chegam ao final da entrada. No entanto, não temos garantia de que uma máquina de Turing em algum momento terminará sua computação, já que ela pode permanecer indo e voltando na fita indefinidamente. Isto nos força a definir *duas* classes diferentes de linguagem para máquinas de Turing: as linguagens das máquinas de Turing que sempre param, e as linguagens das máquinas de Turing que podem entrar em ciclo e nunca parar.

Definição 4.5 (linguagem recursiva e recursivamente enumerável)**.** Uma linguagem L é *recursivamente enumerável* se existe uma máquina de Turing A que a reconhece: quando tem como entrada uma palavra de L, A em algum momento para aceitando a palavra; quando tem como entrada uma palavra que não pertence a L, A pode parar rejeitando a palavra, ou pode nunca parar.

Uma linguagem L é *recursiva* se existe uma máquina de Turing B que a *decide*: tendo como entrada uma palavra de L, B para aceitando a pala-

vra. Quando tem como entrada uma palavra que não pertence a L, B para rejeitando a palavra. A máquina B sempre para. ⧫

Toda linguagem recursiva é, por definição, recursivamente enumerável também.

Exemplo 4.6. A linguagem $a^n b^n c^n$ é recursivamente enumerável, porque é reconhecida pela máquina de Turing do Exemplo 4.4. Esta linguagem é também recursiva, porque claramente a máquina de Turing que a reconhece sempre para.

O mesmo vale para $a^n b^n$, que é reconhecida pela máquina do Exemplo 4.3. ◀

4.2 Variantes

É natural questionar se a máquina de Turing não pode ser modificada para que se torne, de alguma maneira, mais poderosa. Discutimos nesta seção algumas modificações, e mostramos que elas não mudam o poder computacional das máquinas de Turing (as linguagens aceitas continuam sendo as mesmas).

4.2.1 Múltiplas fitas

Uma máquina de Turing com múltiplas fitas funciona da mesma forma que uma máquina de Turing comum, exceto que tem várias fitas, com uma cabeça de leitura e gravação em cada uma. A decisão tomada a cada passo é, dado o estado atual e os símbolos em cada fita, qual símbolo escrever em cada uma, e para qual lado cada cabeça se move. A única modificação na representação do autômato é em sua função de transição: em uma máquina com 3 fitas, por exemplo,

$$\delta(q_i, s^1, s^2, s^3) = (q_j, r^1, r^2, r^3, \leftarrow, \leftarrow, \rightarrow)$$

significa que, estando no estado q_i, e lendo s^1, s^2, s^3 nas três fitas, a máquina muda para o estado q_j; grava r^1, r^2, r^3; e move as três cabeças de leitura e gravação para a esquerda, esquerda e direita.

Assim, em uma máquina de Turing com k fitas, a função de transição será

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\leftarrow, \rightarrow\}^k.$$

Teorema 4.7. *Se uma linguagem é reconhecida por uma máquina de Turing com k fitas, é também reconhecida por uma máquina de Turing com uma única fita.*

Demonstração. Seja M_k uma máquina de Turing com k fitas. Podemos construir uma máquina de Turing com uma única fita que aceita a mesma linguagem de M_k.

Primeiro, representamos as k fitas e as k cabeças de leitura e gravação em uma única fita. É possível simular várias *trilhas* em uma fita, usando um alfabeto maior: suponha que $\Sigma = \{a, b\}$. Se em cada posição queremos escrever três símbolos, podemos usar um alfabeto $\Sigma = \{0, \ldots, 7\}$, de forma que bbb corresponda a 7, bba corresponda a 6, e assim por diante.

Para cada fita de M_k, representaremos duas trilhas na nova máquina. Uma trilha simulará o conteúdo da i-ésima fita, e a outra manterá a posição da i-ésima cabeça de leitura e gravação.

						▼		
x	y	y	y	y	y	x	x	x
	▼							
a	a	a	b	a	c	a	b	c
				▼				
1	0	0	0	1	1	0	1	0

Para cada transição de M_k, crie um trecho de programa na nova máquina de Turing que varre a fita da esquerda para a direita, simulando a transição. Isto inclui ajustar as posições dos marcadores de cabeça de leitura e gravação e realizar as escritas na fita. □

4.2.2 Não-determinismo

Da mesma forma que autômatos finitos e com pilha, definimos máquinas de Turing não-determinísticas. Para cada estado atual e símbolo lido, a máquina poderá escolher entre zero ou mais possibilidades de próximo estado, símbolo gravado e direção a seguir:

$$\delta : Q \times \Gamma \to P\big(Q \times \Gamma \times \{\leftarrow, \rightarrow\}\big).$$

Teorema 4.8. *Se uma linguagem é reconhecida por uma máquina de Turing não-determinística, ela também é reconhecida por uma máquina de Turing determinística.*

Demonstração. (Rascunho) Um autômato determinístico pode simular um autômato não-determinístico usando a técnica de *backtracking*. □

Embora a linguagem das máquinas de Turing não-determinísticas seja a mesma das determinísticas, elas são diferentes, e as duas tem papel fundamental no estudo da Complexidade de Algoritmos.

4.2.3 Autômatos Finitos com duas pilhas

Um autômato finito com duas pilhas pode desempilhar e empilhar símbolos em duas pilhas em cada passo de computação. Pode-se usar os critérios de parada definidos para autômatos com pilha. A função de transição desses autômatos é

$$\delta : Q \times \Sigma \times \Gamma_\varepsilon^2 \to Q \times (\Gamma^*)^2$$

Um autômato finito com duas pilhas pode simular uma máquina de Turing, usando as duas pilhas como se fossem uma fita. Sejam A e B as duas pilhas do autômato.

- inicialmente, empilha na pilha A toda a entrada;
- para andar à direita, desempilhe da pilha B um símbolo e o empilhe na pilha A;
- para escrever na fita e andar à direita, desempilhe da pilha B um símbolo e o empilhe o novo símbolo na pilha A.

Mais pilhas não aumentam o poder computacional do autômato.

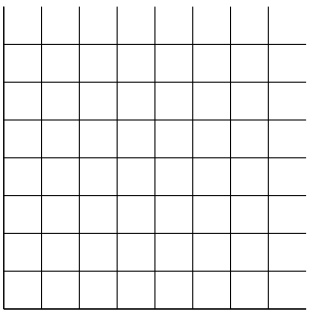
4.2.4 Máquinas com contadores

Uma máquina com contador é semelhante a um autômato finito, exceto que, além da fita de entrada, tem contadores. Além de ler símbolos da fita e mudar de estado, o autômato pode incrementar ou decrementar o valor de um de seus contadores a cada passo.

4.2.5 Outras variantes

Há outras modificações que podem ser feitas em máquinas de Turing; nenhuma delas aumenta o poder computacional.

- **Fita Multidimensional.** Uma fita multidimensional é semelhante a um mapa de células, onde cada uma é referenciada por dois índices.



A função de transição de uma máquina como esta é semelhante a de uma máquina de Turing comum, exceto que no contradomínio temos quatro possibilidades de movimento ($\leftarrow, \uparrow, \downarrow, \rightarrow$) ao invés de somente uma. A função de transição dessa máquina é

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \uparrow, \downarrow, \rightarrow\}.$$

- **Múltiplas cabeças.** Pode-se definir uma máquina de Turing com única fita e múltiplas cabeças de leitura e gravação. A função de transição é semelhante àquela para múltiplas fitas.

$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{\leftarrow, \rightarrow\}^k.$$

- **Três movimentos na fita.** Da maneira como definimos, uma máquina de Turing é obrigada a mudar de posição na fita a cada passo. Poderíamos modificá-la para que também seja possível "permanecer na posição atual", que representaremos por $\circ$. Máquinas de Turing deste tipo tem função de transição da seguinte forma.

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \circ, \rightarrow\}.$$

4.3 Mais que reconhecer linguagens

Por poderem gravar na fita, as Máquinas de Turing podem ser usadas não apenas para reconhecer linguagens, mas também para computar funções, enumerar linguagens, e simular outras máquinas.

4.3.1 Máquinas de Turing que computam funções

Se presumirmos que a palavra deixará inicialmente na fita contém argumentos numéricos, podemos ver a Máquina de Turing como uma máquina que computa funções. O resultado da computação é o que for deixado pela máquina no final da computação.

O exemplo a seguir ilustra esta idéia.

Exemplo 4.9. Construiremos uma máquina de Turing que some dois números em representação binária. A máquina aplicará o algoritmo usual para soma de binários. Presumimos que

- A máquina tem três fitas: duas para os dois números da entrada, e uma terceira fita onde a soma será escrita.

- Apesar de ser possível usar uma máquina que anda em cada uma das fitas independentemente, isto não será necessário, e *esta máquina*

sempre fará movimentos para a direita ou esquerda nas três fitas ao mesmo tempo.

- Presumimos que os dois números começam com um dígito zero. Isto nos permitirá fazer soma com *carry* ("vai-um").

A máquina terá quatro estados.

- q_0: vá para a direita até encontrar um branco; volte uma casa para a esquerda, para ficar em cima do último símbolo. vá para q_1
- q_1: se a posição atual contém um branco, vá para q_3. Caso contrário, some os símbolos das duas fitas e coloque o resultado na terceira. Ande uma posição à esquerda. Se o resultado da soma era menor ou igual que 1, então vá para q_2, senão, permaneça em q_1.
- q_2: se a posição atual contém um branco, vá para q_3. Caso contrário, some 1 com os símbolos das duas fitas e coloque o resultado na terceira. Ande uma posição à esquerda. Se o resultado da soma era maior que 1, então permaneça q_2, senão, vá para q_1.
- q_3: pare.

Não há transições saindo de q_3. Por termos representado a máquina com três fitas, seu diagrama é visualmente carregado. Ao invés dele, construímos uma tabela representando a função de transição.

	q_0	q_1	q_2	q_3
$(\#,\#\,\#)$	$(\#,\#\,\#), q_1, \leftarrow$	$(\#,\#\,\#), q_3, \rightarrow$	$(\#,\#\,\#), q_3, \rightarrow$	
$(0,0,z)$	$(0,0,z), q_0, \rightarrow$	$(0,0,0), q_1, \leftarrow$	$(0,0,1), q_1, \leftarrow$	
$(0,1,z)$	$(0,1,z), q_0, \rightarrow$	$(0,1,1), q_1, \leftarrow$	$(0,1,0), q_2, \leftarrow$	
$(1,0,z)$	$(1,0,z), q_0, \rightarrow$	$(1,0,1), q_1, \leftarrow$	$(1,0,0), q_2, \leftarrow$	
$(1,1,z)$	$(1,1,z), q_0, \rightarrow$	$(1,1,0), q_2, \leftarrow$	$(1,1,1), q_2, \leftarrow$	

◀

No exemplo dado, presumimos que a máquina tem três fitas, e que os argumentos foram deixados em duas delas. É comum, ao descreve máquinas de Turing como computadoras, o uso de outra forma de codificar os argumentos e o resultado: opta-se pela representação *unária* de números naturais: o número n é representado por q^n, e os argumentos são separados por zeros. Por exemplo, se uma Máquina de Turing foi concebida para calcular uma função f em três variáveis, então para calcular $f(2,3,1)$, damos à máquina uma fita contendo

$$11011101\#$$

A máquina deixará na fita o resultado, na forma 1^k.

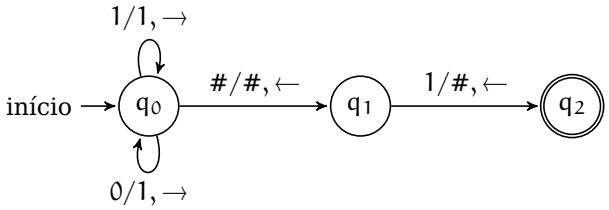
Exemplo 4.10. Construiremos uma máquina de Turing que soma dois números na representação unária. Quando a entrada for

$$1^a 0 1^b$$

A saída deverá ser

$$1^{a+b}$$

Resulta que a construção da máquina é bastante simples: ela deve andar para a direita procurando um zero. Ao encontrá-lo, *desloca o segundo argumento para a esquerda*. Para realizar isto, grava um 1 onde estava o zero e segue sem mudar nada até encontrar um branco. Chegando ao branco, volta e grava outro branco na posição anterior.



A máquina é $(Q, \{0,1\}, \delta, q_0, \{q_2\})$, com $Q = \{q_0, q_1, q_2\}$ e

$$\begin{aligned}
\delta(q_0, 1) &= (q_0, 1, \rightarrow) \\
\delta(q_0, 0) &= (q_0, 1, \rightarrow) \\
\delta(q_0, \#) &= (q_1, \#, \leftarrow) \\
\delta(q_1, 1) &= (q_2, \#, \leftarrow)
\end{aligned}$$

◀

Claramente, é possível construir máquinas de Turing que computam funções mais elaboradas. Em particular, é possível

- computar funções com inteiros (usando alguma marca para sinalizar números menores que zero) e racionais, que podem ser representados como pares de inteiros;
- computar funções com mais de um número como resultado, bastando para isso usar, na saída, a mesma codificação da entrada; e
- computar funções com elementos não numéricos (grafos, por exemplo), desde que sejam enumeráveis.

4.3.2 Máquinas de Turing que emulam Máquinas de Turing (a *Máquina de Turing Universal*)

seção incompleta

Uma vez que é possível descrever formalmente máquinas de Turing (e que a descrição de qualquer máquina de Turing é finita), podemos codificar essa descrição de alguma maneira em uma fita, para que sirva de entrada para outra máquina de Turing. Desta maneira, é possível construir máquinas que simulam o comportamento de outras.

Há diversas maneiras de codificar uma máquina de Turing; desenvolvemos uma delas. Presumiremos o seguinte.

- $Q = \{q_1, \ldots, q_n\}$
- q_1 é o estado inicial.
- $\Sigma = \{0, 1\}$ (qualquer alfabeto pode ser codificado em binário, portanto presumimos que o alfabeto da máquina é binário)
- $\Gamma - \Sigma = \{\#\}$ (além do alfabeto Σ, somente o branco será usado).
- $F = \{q_n\}$ (só é necessário um estado final).

Já temos o conjunto de estados enumerado, sendo q_i o i-ésimo estado. Enumeramos também os símbolos do alfabeto e as direções, para que possamos identificá-los por seus índices:

$$\begin{aligned} s_1 &= 0 & d_1 &= \leftarrow \\ s_2 &= 1 & d_2 &= \rightarrow \\ s_3 &= \# \end{aligned}$$

Como já definimos o alfabeto, e fixamos os estados inicial e final, não precisamos inclui-los explicitamente na codificação da máquina. Apenas listamos a função de transição. Cada transição

$$\delta(q_i, s_j) = (q_m, s_n, d_o)$$

pode ser representada como

$$0^i 1 0^j 1 0^m 1 0^n 1 0^o.$$

Como a representação usa somente zeros para cada um dos elementos, e uns isolados para separá-los, podemos usar uns consecutivos para separar as transições. Uma máquina de Turing pode então ser codificada da seguinte forma

$$111 \text{ transição}_1 \, 11 \text{transição}_2 \, 11 \, \cdots \, 11 \text{ transição}_k \, 111$$

Exemplo 4.11. A máquina desenvolvida no Exemplo 4.10 tem quatro transições:

$$\delta(q_0, 1) = (q_0, 1, \rightarrow)$$
$$\delta(q_0, 0) = (q_0, 1, \rightarrow)$$
$$\delta(q_0, \#) = (q_1, \#, \leftarrow)$$
$$\delta(q_1, 1) = (q_2, \#, \leftarrow)$$

Primeiro, mudamos a codificação dos estados, para que o primeiro seja "q_1": renomeamos, $q_0 \rightarrow q_1$, $q_1 \rightarrow q_2$, $q_2 \rightarrow q_3$):

$$\delta(q_1, 1) = (q_1, 1, \rightarrow)$$
$$\delta(q_1, 0) = (q_1, 1, \rightarrow)$$
$$\delta(q_1, \#) = (q_2, \#, \leftarrow)$$
$$\delta(q_2, 1) = (q_3, \#, \leftarrow)$$

Tivemos que renomear os estados desta maneira porque o estado q_0 em unário seria representado pela cadeia vazia, que poderia gerar problemas.

A primeira regra é $\delta(q_1, 1) = (q_1, 1, \rightarrow)$. A regra lista q_1, 1, q_1, 1, $\rightarrow$, ou seja, q_1, s_2, q_1, s_2, d_2. Sua codificação será, portanto,

$$\underbrace{0}_{q_1}\ 1\ \underbrace{00}_{s_2}\ 1\ \underbrace{0}_{q_1}\ 1\ \underbrace{00}_{s_2}\ 1\ \underbrace{00}_{d_2}$$

Aplicamos a tradução a todas as transições e obtemos a codificação a seguir.

$$\begin{aligned}
\delta(q_1, 1) = (q_1, 1, \rightarrow) &\quad\rightarrow\quad 0\,1\,00\,1\,0\,1\,00\,1\,00 \\
\delta(q_1, 0) = (q_1, 1, \rightarrow) &\quad\rightarrow\quad 0\,1\,0\,1\,0\,1\,00\,1\,00 \\
\delta(q_1, \#) = (q_2, \#, \leftarrow) &\quad\rightarrow\quad 0\,1\,000\,1\,00\,1\,000\,1\,0 \\
\delta(q_2, 1) = (q_3, \#, \leftarrow) &\quad\rightarrow\quad 00\,1\,00\,1\,000\,1\,000\,1\,0
\end{aligned}$$

A máquina de Turing é, portanto, codificada da seguinte maneira.

$$\begin{aligned}
&111\ 0\,1\,00\,1\,0\,1\,00\,1\,00\ 11\ 0\,1\,0\,1\,0\,1\,00\,1\,00\ 11 \\
&0\,1\,000\,1\,00\,1\,000\,1\,0\ 11\ 00\,1\,00\,1\,000\,1\,000\,1\,0\ 111
\end{aligned}$$

Uma entrada para a máquina de Turing que simula esta máquina seria $< M > 0 < 2 > 0 < 3 >$, se quisermos somar 2 com 3. $< M >$ é a codificação de M; $< 2 >$ e $< 3 >$ são as codificações dos argumentos em unário:

$$< 2 > = 11$$
$$< 3 > = 111$$

Assim, $< M > 0 < 2 > 0 < 3 >$ é

$$111\,0\,1\,00\,1\,0\,1\,00\,1\,00\,11\,0\,1\,0\,1\,0\,1\,00\,1\,00\,11$$
$$0\,1\,000\,1\,00\,1\,000\,1\,0\,11\,00\,1\,00\,1\,000\,1\,000\,1\,0\,111$$
$$0\,11\,0\,111$$ ◀

Definição 4.12 (máquina de Turing universal)**.** Uma *máquina de Turing universal* é uma máquina de Turing U que recebe como entrada uma descrição codificada de outra máquina M; uma sequência de símbolos w; e simula a execução de M com a entrada w:

- Se M aceita a palavra w, deixando um conteúdo r na fita, U aceita $(< M >, w)$ e deixa r em alguma posição da fita.

- Se M rejeita a palavra w, U rejeita $(< M >, w)$. ⧫

4.3.3 Máquinas de Turing que enumeram linguagens

seção incompleta

Uma vez que máquinas de Turing podem escrever em uma fita e emular outras máquinas de Turing, elas podem ser usadas para enumerar linguagens. Podemos, por exemplo, idealizar uma máquina de Turing que escreva, em uma fita, palavras separadas por espaços. Dizemos que esta máquina de Turing *gera* aquelas palavras (ou seja, gera uma linguagem).

Definição 4.13 (conjunto enumerável)**.** Um conjunto A é *enumerável* se é finito ou se tem a mesma cardinalidade que $\mathbb{N}$, ou seja, se existe uma bijeção entre A e $\mathbb{N}$. ⧫

Exemplo 4.14. Os inteiros são enumeráveis, dado que existe uma bijeção $f : \mathbb{N} \to \mathbb{Z}$:

n	0	1	2	3	4	5	6	$\cdots$
$f(n)$	0	1	-1	2	-2	3	-3	$\cdots$

◀

É interessante que a bijeção nos dá uma *ordem* em $\mathbb{Z}$, diferente da usual: $0 \prec 1 \prec -1 \prec 2 \prec \cdots$.

Teorema 4.15. *$\mathbb{N} \times \mathbb{N}$ é enumerável.*

Demonstração. Enumeramos $\mathbb{N} \times \mathbb{N}$ da seguinte maneira: primeiro os pares

que somam 2; depois os pares que somam 3; e assim por diante.

$$\begin{array}{c} (1,1) \\ \hline (1,2) \\ (2,1) \\ \hline (1,3) \\ (2,2) \\ (3,1) \\ \hline (1,4) \\ \vdots \end{array}$$

A próxima figura ilustra este processo. Cada diagonal passa por pares que somam um número natural maior ou igual a dois.

$$\begin{array}{cccccc} (1,1) & (1,2) & (1,3) & (1,4) & (1,5) & (1,6) \\ (2,1) & (2,2) & (2,3) & (2,4) & (2,5) & (2,6) \\ (3,1) & (3,2) & (3,3) & (3,4) & (3,5) & (3,6) \\ (4,1) & (4,2) & (4,3) & (4,4) & (4,5) & (4,6) \\ (5,1) & (5,2) & (5,3) & (5,4) & (5,5) & (5,6) \\ (6,1) & (6,2) & (6,3) & (6,4) & (6,5) & (6,6) \\ & & & \vdots & & \end{array} \quad \cdots \quad \ddots$$

Há outras maneiras de percorrer a tabela, cada uma delas resultando em uma ordem diferente, mas nos interessa apenas que ela é enumerável – o que já provamos. □

Teorema 4.16. *Seja* $\Sigma = \{0,1\}$. *Então* Σ^* *é enumerável.*

Demonstração. Σ^*, o conjunto de todas as cadeias de zeros e uns, pode ser enumerado de maneira semelhante à enumeração de pares de inteiros: enumerando primeiro as cadeias de tamanho zero, as de tamanho um; as de tamanho dois, etc.

Ao enumerar cadeias de tamanho k, listamos as sequências na ordem crescente de valor, interpretando as cadeias como binários.

tamanho	cadeias
0	ε
1	0, 1
2	00, 01, 10, 11
3	000, 001, 010, 011, 100, 101, 110, 111
⋮	⋮

Chamamos esta ordem de *ordem canônica* ou *lexicográfica*.

A bijeção entre $\mathbb{N}$ e Σ^* é, portanto,

0	1	2	3	4	5	6	7	$\cdots$
ε	0	1	00	01	10	11	100	$\cdots$

□

Os dois Lemas a seguir serão usados na demonstração do Lema 4.20. A prova deles é pedida no Exercício 58.

Lema 4.17. *Existe uma máquina de Turing que gera Σ^* em sua fita, na ordem canônica.*

Lema 4.18. *Existe uma máquina de Turing que gera em sua fita $\mathbb{N} \times \mathbb{N}$ na mesma ordem explicitada na demonstração do Teorema 4.15.*

Lema 4.19. *Seja M uma máquina de Turing que enumera uma linguagem L. Então existe uma máquina M_D, que decide a mesma linguagem.*

Lema 4.20. *Seja M uma máquina de Turing. Então existe uma outra máquina de Turing M_E, que enumera a linguagem de M.*

Demonstração. Toda máquina de Turing pode ser codificada como uma sequência *finita* de zeros e uns. O conjunto $\mathcal{T}$, de todas as máquinas de Turing, é, portanto, um conjunto de cadeias de zeros e uns, e portanto enumerável. Da mesma forma, qualquer linguagem, sendo subconjunto de Σ^*, é enumerável.

A máquina M_E simulará M, portanto tem as mesmas fitas de M, *mais uma*, onde escreverá as palavras (sua *fita de saída*).

M_E gera $\mathbb{N}\times\mathbb{N}$, e para cada par (i, j), simula j passos de M com a i-ésima palavra como entrada. Quando M aceitar, M_E escreve a palavra em sua fita de saída. □

Teorema 4.21. *Uma linguagem é recursivamente enumerável se e somente se é gerada por uma máquina de Turing.*

Demonstração. Segue dos Lemas 4.20 e 4.19. □

Teorema 4.22. *Uma linguagem é recursiva se e somente se é gerada por uma máquina de Turing,* na ordem canônica.

Demonstração. Se uma máquina M gera L em ordem lexicográfica, então podemos construir uma máquina M_D, que decide se uma palavra pertence a L: a máquina M_D lê sua entrada (a palavra w), e simula M. Quando M gerar w, M_D aceita. Quando M já tiver enumerado todas as palavras do tamanho de $|w|$, M_D para rejeitando a palavra.

Agora, suponha que exista uma máquina N que decide uma linguagem L. Então podemos construir uma máquina M_E que enumera L: a máquina M_E enumera Σ^*, na ordem lexicográfica, e para cada palavra w gerada, simula

N com a entrada w. Se N aceitar, M_E escreve a palavra em sua fita de saída. Se N rejeitar, M_E não escreve. Como N sempre para, M_E gerará todas as palavras da linguagem. □

4.4 Gramáticas Irrestritas

As linguagens recursivamente enumeráveis são, assim como as outras de que tratamos, geradas por uma classe de gramáticas, chamadas de *gramáticas irrestritas*.

Definição 4.23 (gramática irrestrita)**.** Uma gramática é *irrestrita* se suas produções são da forma $(\Sigma \cup N)^+ \to (\Sigma \cup N)^*$. A única restrição imposta, portanto, é que o lado esquerdo das produções não seja vazio. ⧫

Exemplo 4.24. A gramática a seguir gera a linguagem $a^n b^n c^n$.

$$\begin{aligned} S &::= abc \mid A \\ A &::= aABc \mid abc \\ cB &::= Bc \\ bB &::= bb \end{aligned}$$

Como exemplo, derivamos a palavra aaabbbccc.

$$\begin{aligned} S &\Rightarrow \underline{A} && \\ &\Rightarrow a\underline{A}Bc && (A \to aABc) \\ &\Rightarrow aaAB\underline{cB}c && (A \to aABc) \\ &\Rightarrow aa\underline{A}BBcc && (cB \to Bc) \\ &\Rightarrow aaab\underline{cB}Bcc && (A \to abc) \\ &\Rightarrow aaa\underline{bB}cBcc && (cB \to Bc) \\ &\Rightarrow aaabb\underline{cB}cc && (bB \to bb) \\ &\Rightarrow aaab\underline{bB}ccc && (cB \to Bb) \\ &\Rightarrow aaabbbccc && (bB \to bb) \end{aligned}$$

◀

Exemplo 4.25. A seguir está uma gramática que gera a linguagem sobre o alfabeto $\Sigma = \{a, b, c\}$, tendo a mesma quantidade para cada símbolo (mas

em qualquer ordem).

$$
\begin{aligned}
S &::= ABCS \mid \varepsilon \\
A &::= a \\
B &::= b \\
C &::= c \\
AB &::= BA \\
BC &::= CB \\
AC &::= CA \\
BA &::= AB \\
CA &::= AC \\
CB &::= BC
\end{aligned}
$$

Na primeira linha, as regras determinam que uma palavra seja o fecho estrela de ABC. As próximas regras, para A, B e C, determinam que cada não-terminal pode ser trocado por exatamente um terminal equivalente. Se deixássemos as regras assim, a linguagem seria $(abc)^*$, e só conteria palavras da forma $abcabcabc\cdots abc$. As seis últimas regras, no entanto, permitem trocar a ordem dos símbolos, resultando na linguagem $\{a+b+c\}^*$ com quantidades iguais de as, bs e cs. ◀

Teorema 4.26. *Toda linguagem gerada por uma gramática irrestrita é recursivamente enumerável.*

Demonstração. (Rascunho) Seja G uma gramática. Construiremos uma máquina de Turing com duas fitas. Na primeira estará a entrada (esta servirá apenas para leitura). Na segunda, a máquina tentará simular a derivação da palavra. Se conseguir terminar a derivação e as duas fitas tiverem conteúdo idêntico, a máquina para aceitando. Se sua entrada for qualquer palavra não aceita por G, a máquina nunca para.

A primeira fita é inicializada com a palavra a ser verificada; a segunda, com o símbolo inicial apenas.

A máquina começa escolhendo não-deterministicamente uma posição na segunda fita. Em seguida, escolhe não-deterministicamente uma regra $\alpha ::= \beta$. Se a cadeia α estiver na posição escolhida, a máquina troca a essa subcadeia por β.

Em seguida, compara as duas fitas. Se forem iguais, para aceitando. Se não forem, volta ao passo em que escolhe uma posição. □

Teorema 4.27. *Toda linguagem recursivamente enumerável é gerada por alguma gramática irrestrita.*

Demonstração. (Rascunho) A partir de uma máquina de Turing M, constrói-se uma gramática (irrestrita) que primeiro gera não-deterministicamente

uma palavra de Σ^*; depois disso, a gramática permite gerar uma cópia da palavra, para manter duas cópias dela antes de terminar a derivação; em seguida, as regras da gramática permitem simular a execução das regras de M sobre uma das cópias. Se M aceitar, as regras permitem apagar a cópia em que a simulação foi feita, e terminar com a outra cópia, intacta. Se M não aceitar a palavra, a gramática não gera palavra alguma, gerando uma variável para a qual não há expansão possível. □

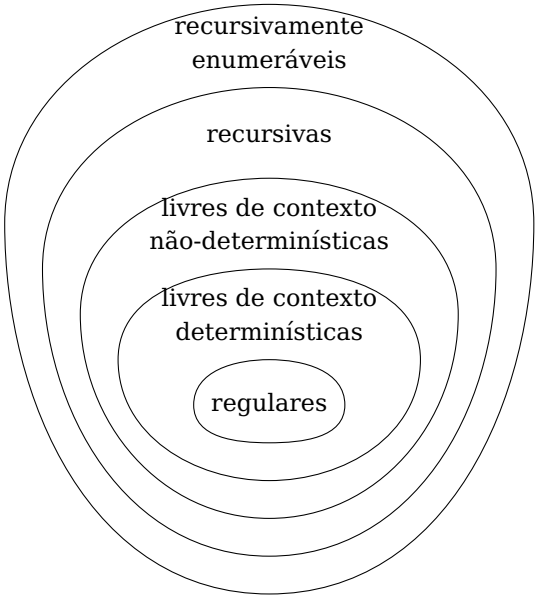
Corolário 4.28. *Linguagens livres de contexto e regulares são recursivamente enumeráveis.*

Demonstração. Gramáticas regulares e livres de contexto são, por definição, irrestritas também. □

Teorema 4.29. *Linguagens livres de contexto e regulares são recursivas.*

Demonstração. (Rascunho) São recursivamente enumeráveis. O fato de serem recursivas decorre de que seus autômatos percorrem a palavra apenas da esquerda para a direita, uma única vez. Quando uma máquina de Turing simular estes autômatos, também fará o percurso pela palavra somente uma vez, necessariamente parando ao final do processo. □

As inclusão entre classes de linguagens é, portanto, como na próxima figura.



4.5 Linguagens que não são Recursivamente Enumeráveis

Autômatos finitos e com pilha, por lerem sua entrada uma única vez, da esquerda para a direita, não podem aceitar palavras com tamanho maior que um certo número. Máquinas de Turing são bastante diferentes desses autômatos – diferentes o suficiente para que um "lema do bombeamento" não tenha sido enunciado e elaborado para elas. Nesta Seção mostraremos, usando a técnica de *diagonalização*, que há linguagens que não são recursivamente enumeráveis. No Capítulo 5, trataremos de linguagens que são recursivamente enumeráveis, mas não são recursivas (as linguagens que são reconhecidas, mas não decididas, por máquinas de Turing).

A demonstração que faremos, de que há linguagens não recursivamente enumeráveis, usa a técnica de diagonalização, usada por Georg Cantor para mostrar que a cardinalidade dos Reais é maior que a dos Naturais; pode isso, apresentamos brevemente esta demonstração.

Teorema 4.30. $|\mathbb{R}| > |\mathbb{N}|$*, ou seja, o conjunto dos números reais não é enumerável.*

Demonstração. Provamos que $|[0,1]| > |\mathbb{N}|$, que implica que $|\mathbb{R}| > |\mathbb{N}|$.

Presuma que é possível enumerar os números no intervalo $[0,1]$. Então existe uma bijeção entre esses números e os naturais, e os números em $[0,1]$, portanto, podem ser listados em uma tabela, na ordem dada pela bijeção. Por exemplo, se representarmos somente os números após a vírgula, e os primeiros reais enumerados tiverem seus primeiros dígitos como a seguir,

$$\begin{aligned} &0,10040\cdots\\ &0,85002\cdots\\ &0,75700\cdots\\ &0,40000\cdots\\ &0,03030\cdots \end{aligned}$$

então a tabela começaria da seguinte maneira.

$$\begin{array}{c|ccccc} 0 & 1 & 0 & 0 & 4 & 0 \\ 1 & 8 & 5 & 0 & 0 & 2 \\ 2 & 7 & 5 & 7 & 0 & 0 \\ 3 & 4 & 0 & 0 & 0 & 0 \\ 4 & 0 & 3 & 0 & 3 & 0 \\ \vdots & & & & & \ddots \end{array} \tag{4.1}$$

De maneira mais geral, a tabela será

0	$d_{0,0}$	$d_{0,1}$	$d_{0,2}$	$d_{0,3}$	$d_{0,4}$
1	$d_{1,0}$	$d_{1,1}$	$d_{1,2}$	$d_{1,3}$	$d_{1,4}$
2	$d_{2,0}$	$d_{2,1}$	$d_{2,2}$	$d_{2,3}$	$d_{2,4}$
3	$d_{3,0}$	$d_{3,1}$	$d_{3,2}$	$d_{3,3}$	$d_{3,4}$
4	$d_{4,0}$	$d_{4,1}$	$d_{4,2}$	$d_{4,3}$	$d_{4,4}$
$\vdots$					$\ddots$

Agora construímos um número que não está listado na tabela: tome cada elemento d_{ii} da diagonal da tabela e troque por outro, diferente de d_{ii}. O número resultante não está em nenhuma linha (não pode estar na i-ésima linha, porque difere justamente na i-ésima posição).

Por exemplo, se $[0,1]$ fosse enumerado de acordo com a tabela 4.1, a diagonal da tabela seria um número começando por $0,15700$:

0	$\boxed{1}$	0	0	4	0
1	8	$\boxed{5}$	0	0	2
2	7	5	$\boxed{7}$	0	0
3	4	0	0	$\boxed{0}$	0
4	0	3	0	3	$\boxed{0}$
$\vdots$					$\ddots$

Podemos portanto construir um número que está em $[0,1]$, mas que não está na tabela; este número poderia, portanto, começar com $0,26811\cdots$, tendo sempre seu i-ésimo dígito diferente do i-ésimo elemento da diagonal.

De fato, podemos construir infinitos números de $[0,1]$ que estão fora da tabela. □

Teorema 4.31. *Existe uma linguagem que não é recursivamente enumerável.*

Demonstração. Usando um argumento enumeratório: o conjunto $\mathcal{L}$, de todas as linguagens possíveis sobre Σ, é $|\mathcal{L}| = |P(\Sigma^*)|$. Mas a cardinalidade das partes de um conjunto é sempre estritamente maior que a do próprio conjunto, e $P(\Sigma)$ não é enumerável.

Podemos também construir a demonstração de outra forma: o mesmo argumento de diagonalização usado para mostrar que $|\mathbb{R}| > |\mathbb{N}|$ pode ser usado para mostrar que existe uma linguagem que nenhuma máquina de Turing reconhece.

Uma linguagem construída desta forma é chamada de *linguagem diagonal*.

Presumimos que as linguagens são enumeráveis. Construímos, portanto, uma tabela. Cada linha representa uma linguagem sobre $\{0,1\}^*$: a posição

(i, j) da tabela significa que a i-ésima linguagem contém a j-ésima palavra. Por exemplo, na linha abaixo a linguagem L_i contém as palavras $\varepsilon, 00, 01$, mas não contém as palavras $0, 1, 10, 11000$.

								0	
				0	1	0	1	0	$\cdots$
	ε	0	1	0	0	1	1	0	
L_i	•			•	•				$\cdots$

Agora podemos obter, por diagonalização, uma linguagem que não está na tabela – e portanto *o conjunto de todas as linguagens sobre* $\{0, 1\}$ *não é enumerável.*

Como o conjunto das linguagens não é enumerável e o das máquinas de Turing é enumerável, há linguagens que não são reconhecidas por máquinas de Turing. □

Não existe apenas uma linguagem diagonal: há infinitas delas. E assim como acontece com números reais e naturais, *existe uma quantidade não-enumerável de linguagens que não é reconhecida por qualquer máquina de Turing.*

4.6 Propriedades

Teorema 4.32. *Linguagens recursivamente enumeráveis são fechadas para união, interseção, concatenação e fecho estrela.*

Demonstração. Sejam G_1 e G_2 duas gramáticas irrestritas, com símbolos iniciais S_1 e S_2, e sem símbolos não-terminais em comum.

Usando as regras das duas gramáticas, podemos criar outras três gramáticas, com símbolos iniciais U, C e E, cada uma com uma nova regra:

- $U ::= S_1 \mid S_2$, a união das duas linguagens;
- $C ::= S_1 S_2$, a concatenação das duas linguagens;
- $E ::= S_1 E \mid \varepsilon$, o fecho estrela de G_1.

Para a interseção, observamos que existem máquinas de Turing M_1 e M_2 que reconhecem as duas linguagens. Podemos construir uma máquina de Turing M que simula o funcionamento simultâneo de M_1 e M_2, aceitando somente quando os dois aceitarem. Se uma das máquinas entrar em um ciclo sem fim, a máquina que as simula também o fará. Isto faz sentido, porque queremos que M pare somente quando *as duas* máquinas, M_1 e M_2 pararem. □

Teorema 4.33. *Linguagens recursivamente enumeráveis* não *são fechadas para diferença de conjuntos nem para complemento.*

Teorema 4.34. *Linguagens recursivas são fechadas para união, interseção, concatenação e fecho estrela.*

Demonstração. **incompleta!** Sejam M_1 e M_2 duas máquinas de Turing *que sempre param*. As linguagens de M_1 e M_2 são, portanto, recursivas.

Construa uma máquina de Turing M a partir do produto cartesiano dos estados de M_1 e M_2. Para verificar que M sempre para, observe que M segue os passos de M_1 e de M_2, simultaneamente, e como estas duas sempre param, M também o fará.

Se fizermos $F = F_1 \cup F_2$, então M deverá parar aceitando quando qualquer uma das duas máquinas aceitam, e a linguagem de M é $L_1 \cup L_2$.

Se fizermos $F = F_1 \cap F_2$, M só vai parar quando M_1 e M_2, ambas, pararem. Assim teríamos $L = L_1 \cap L_2$.

□

Teorema 4.35. *Linguagens recursivas são fechadas para diferença de conjuntos e para complemento.*

Demonstração. Para complemento, basta trocar, na máquina de Turing que aceita a linguagem, os estados finais com os não-finais (usar $F' = Q - F$). *Como a máquina sempre para*, a máquina resultante reconhecerá o complemento da linguagem da máquina original.

Para diferença entre duas linguagens reconhecidas por máquinas M_1 e M_2, podemos construir uma terceira máquina M, que simula as duas outras, parando quando M_1 para e M_2 não para. □

Teorema 4.36. *Linguagens recursivas são fechadas para homomorfismo livre de ε.*

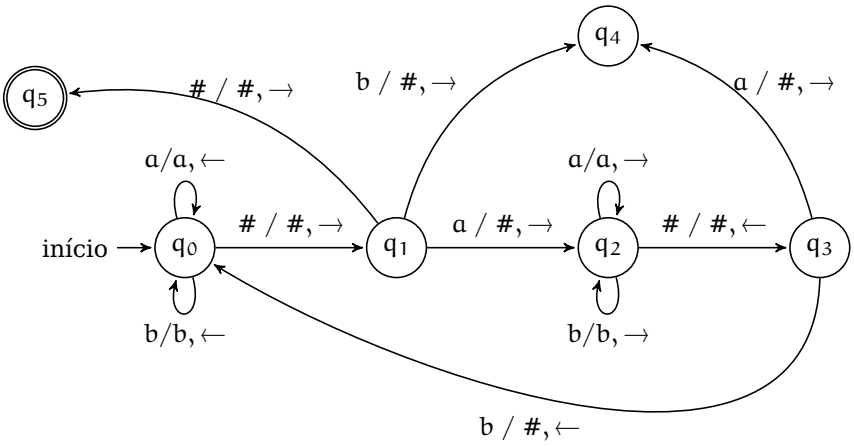
Exercícios

Ex. 49 — Descreva uma máquina de Turing que leia um número natural em representação binária e deixe na fita sua divisão por dois. Se a cadeia de entrada não for um número binário, a máquina deve rejeitar a palavra.

Ex. 50 — Descreva uma máquina de Turing que leia duas cadeias sobre o alfabeto $\{a, b\}$ e decida se elas são iguais. As cadeias devem ser separadas por um símbolo "!". Por exemplo, a máquina deve aceitar a entrada "aaba!aaba", e deve rejeitar "aa!aab".

Ex. 51 — Descreva uma máquina de Turing que leia uma palavra na forma $a^i b^j$, com $j < i$, e complete a quantidade de bs para ficar igual à quantidade de as.

Ex. 52 — Determine a linguagem da máquina de Turing a seguir.



Ex. 53 — Modifique a máquina de Turing do Exemplo 4.4 para que aceite

i) $a^n b^{n+1} c^n$

ii) $a^n b^{2n} c^n$

Ex. 54 — Descreva uma máquina de Turing que reconheça a linguagem $a^n b^m c^n d^m$.

Ex. 55 — Determine a linguagem gerada pela gramática a seguir (demonstre que a linguagem é realmente a que você diz ser).

$$\begin{aligned} S &::= aSBC \mid aBC \\ CB &::= BC \\ aB &::= ab \\ bB &::= bb \\ bC &::= bc \\ cC &::= cc \end{aligned}$$

Ex. 56 — Prove que um autômato finito com fila é equivalente a uma máquina de Turing. (O autômato tem uma fita de entrada, somente de leitura; somente pode mover à direita na fita; ele tem uma *fila* ao invés de pilha – pode enfileirar símbolos de um *alfabeto de fila*, que são retirados na ordem "primeiro a entrar é o primeiro a sair".)

Ex. 57 — Prove que as variantes de máquina de Turing na Seção 4.2.5 são equivalentes, em poder computacional, a máquinas de Turing comuns.

Ex. 58 — Demonstre os Lemas 4.17 e 4.18.

Capítulo 5

Decidibilidade

5.1 Problemas de decisão e de busca

Definição 5.1 (Problemas de decisão e de busca)**.** Um *problema computacional* é uma pergunta que pode ser feita a respeito de um conjunto.

Um *problema de decisão* é um problema computacional que admite somente respostas "sim" ou "não".

Um *problema de busca* é um problema computacional para o qual a solução pode ser uma cadeia qualquer de símbolos. Para um problema de busca pode haver mais de uma solução. ♦

Exemplo 5.2. Seja G um grafo. Determinar se G tem um caminho que passe por todos seus vértices é um problema de decisão, porque a resposta pode ser "sim" ou "não". Determinar o caminho é um problema de busca. Estes dois problemas estão evidentemente relacionados. Dizemos que são duas versões (uma "de decisão" e outra "de busca") do mesmo problema. ◄

Definição 5.3 (Linguagem de um problema)**.** A *linguagem de um problema* de decisão é o conjunto de entradas para as quais o problema tem "sim" como resposta. ♦

Exemplo 5.4. A linguagem do problema descrito no Exemplo 5.2 é o conjunto de todos os grafos em que há um caminho passando por todos os vértices. ◄

5.2 Decidibilidade e o Problema da Parada

Definição 5.5 (Problema decidível)**.** Dizemos que um problema é indecidível se sua linguagem não é decidível (recursiva), o que equivale a dizer que não existe uma máquina de Turing que leia uma instância do problema, e

sempre pare, aceitando quando a instância tem solução e rejeitando quando não tem. ⧫

Exemplo 5.6. O problema de determinar se um número natural maior que um é primo é decidível, porque é possível construir uma máquina de Turing que determine se um número é primo ou não: se n é par, responda "não"; se n é ímpar, enumere os números ímpares entre 3 e $\lfloor\sqrt{n}\rfloor$, e verifique se algum deles divide n. Se n tem divisor entre 3 e $\lfloor\sqrt{n}\rfloor$, responda "não", caso contrário responda "sim". ◀

Definição 5.7 (problema da parada)**.** Dada a descrição de uma máquina de Turing M e uma entrada E, decidir se M para com a entrada E. ⧫

Teorema 5.8. *O problema da parada é indecidível.*

Demonstração. Suponha que existe uma máquina de Turing M_P, que decida se outra máquina deverá parar quando iniciada com determinada entrada:

$$M_P(M, e) = \begin{cases} \text{aceita} & \text{se } M \text{ para com entrada } e \\ \text{rejeita} & \text{se } M \text{ não para com entrada } e \end{cases}$$

Construa a seguinte máquina de Turing, M_C, que usa M_p para verificar se M, quando alimentada com sua própria descrição, para. No entanto, o comportamento de M_C será o de *parar aceitando* se M nunca para com entrada M; ou *nunca parar* se M para com entrada M.

$$M_C(M) \begin{cases} \text{não para} & \text{se } M_P(M, M) \text{ aceita} \\ \text{aceita} & \text{se } M_P(M, M) \text{ rejeita} \end{cases}$$

Ou seja, M_C para se M nunca para com entrada M. E nunca para, se M para com entrada M.

Agora aplicamos M_C a si mesma. Ao examinar a execução de $M_C(M_C)$, chegamos a uma contradição, porque

- se M_C para, então M_C não pode parar, e o resultado deve ser rejeição;
- se M_C não para, então M_C deve parar, e o resultado deve ser aceitação. □

5.3 Reduções. Outros problemas indecidíveis

Definição 5.9 (Problema da Correspondência de Post (PCP))**.** Seja Σ um alfabeto, com $|\Sigma| \geq 2$, e sejam $A = (\alpha_1, \cdots, \alpha_n)$, $B = (\beta_1, \ldots, \beta_n)$ duas listas de cadeias sobre Σ.

Determinar se existe uma sequência de índices $i_1, i_2, \ldots, i_k$ tal que

$$\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\cdots\beta_{i_k}.$$

É comum denotar uma instância do PCP por

$$\binom{\alpha_1}{\beta_1}, \binom{\alpha_2}{\beta_2}, \cdots, \binom{\alpha_n}{\beta_n},$$

Uma solução para uma instância é, portanto, uma sequência de pares onde a concatenação das cadeias na parte superior é igual à concatenação das cadeias na parte inferior. ♦

Exemplo 5.10. Se

$$A = (cca, bab, abc, ab, c, ab, aaa),$$
$$B = (c, bc, b, abba, aca, b, a),$$

então podemos representar esta instância do PCP como

$$\binom{cca}{c}, \binom{bab}{bc}, \binom{abc}{b}, \binom{ab}{abba}, \binom{c}{aca}, \binom{ab}{b}, \binom{aaa}{a}$$

Esta instância tem a solução $(4, 2, 1, 5, 7, 5, 6)$,

$$\binom{ab}{abba}\binom{bab}{bc}\binom{cca}{c}\binom{c}{aca}\binom{aaa}{a}\binom{c}{aca}\binom{ab}{b}$$

porque

$$\begin{aligned} &ab\ bab\ cca\ c\ aaa\ c\ ab \\ =\ &abba\ bc\ c\ aca\ a\ aca\ b \end{aligned}$$

◄

Exemplo 5.11. Sobre o alfabeto $\{a, b, c, d\}$, temos outra instância do PCP:

$$\binom{aa}{ca}, \binom{ab}{cc}, \binom{bab}{daa}, \binom{acd}{cc}, \binom{bdd}{cb}, \binom{ac}{dbaa}$$

Para esta instância não existe correspondência possível, porque todos os α_i começam com a ou b, e todos os β_i começam com c ou d. ◄

Teorema 5.12. *O problema da correspondência de Post é indecidível.*

Demonstração. Idéia apenas Mostraremos que, se o PCP for decidível, então o problema da parada também é.

Suponha que o PCP seja decidível. Então existe uma máquina de Turing (ou um algoritmo, tanto faz) que para respondendo sim ou não para qualquer instância do PCP. Nossa demonstração consiste em usar esse algoritmo para resolver o problema da parada.

Uma instância do problema da parada é (M, s), onde M é uma máquina de Turing e s é uma cadeia. Se M para com a entrada s, então a sequência de configurações de M com entrada s é *finita*. Se M não para com entrada s, a sequência de configurações será *infinita*.

Para cada instância (M, s) do problema da parada, obteremos uma instância do PCP, onde as cadeias formadas pelas peças representam as configurações de M com a entrada s, na ordem em que aparecem na execução. Ou seja, se (M, s) passa pelas configurações

$$\begin{aligned} q_0 s &\vdash \alpha \\ &\vdash \beta \\ &\vdots \\ &\vdash \zeta \end{aligned}$$

então será possível, com as peças da instância do PCP, obter

$$\binom{\#}{\#q_0 s\#}\binom{\cdots}{\cdots}\cdots\binom{\cdots}{\cdots}$$

de forma que as cadeias (superior e inferior – que são idênticas, porque são uma solução para o PCP) sejam iguais a $\# \; q_0 s \; \# \; \alpha \; \# \; \beta \; \# \; \cdots \; \# \; \zeta \; \#$. Se houver uma solução para o PCP, então *existe uma sequência finita de configurações de* M *levando a estado final*, e portanto M para com a entrada s. Desta forma podemos tomar o algoritmo decisório para PCP e usá-lo para resolver o problema da parada. Como o problema da parada é indecidível, o algoritmo de decisão para o PCP não pode existir, e *o problema de correspondência de Post é indecidível*.

Para a cadeia s, a instancia do PCP terá a peça

$$\binom{\#}{\#q_0 s\#}$$

Outras peças serão criadas de forma que as duas cadeias concatenadas (acima e abaixo) formem configurações válidas na execução de M com a entrada s.

Suponha que a máquina de Turing esteja em uma configuração $\alpha q a \beta$ (a cabeça de leitura e gravação está sobre a), e que haja uma regra $\delta(q, a) = \{(r, b, \rightarrow)\}$. Então a mudança na configuração será como descrito a seguir.

$$\alpha \, q \, a \, \beta \quad \underbrace{\vdash}_{\delta(q,a)=\{(r,b,\rightarrow)\}} \quad \alpha \, b \, r \, \beta$$

O efeito, na configuração, foi o de *trocar* qa *por* br.

Da mesma forma, se a configuração for $\alpha \, a \, q \, b \, \beta$ e a máquina usar uma

regra $\delta(q, b) = \{(r, c, \leftarrow)\}$, então a configuração mudará da seguinte maneira.

$$\alpha\, a\, q\, b\, \beta \quad \underbrace{\vdash}_{\delta(q,b)=\{(r,c,\leftarrow)\}} \quad \alpha\, r\, a\, c\, \beta$$

A mudança realizada na configuração foi a de *trocar* aqb *por* rac.

$$\text{para toda regra } \delta(q_i, a) = (q_j, b, \rightarrow), \quad \begin{pmatrix} q_i a \\ b q_j \end{pmatrix}$$
$$\text{para toda regra } \delta(q_i, b) = (q_j, c, \leftarrow), \quad \begin{pmatrix} a q_i b \\ q_j a c \end{pmatrix}$$

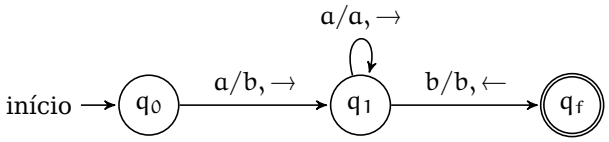
Além destas peças, também precisaremos de peças que insiram símbolos individuais iguais acima e abaixo:

$$\text{para todo } a \in \Gamma, \quad \begin{pmatrix} a \\ a \end{pmatrix}$$

Quando uma configuração contém um estado final, *a máquina de Turing já parou*. No entanto, os estados finais surgirão na cadeia inferior, em uma última configuração, que ficará "sobrando". As peças a seguir servirão para terminar o encaixe.

$$\begin{pmatrix} q_f \# \# \\ \# \end{pmatrix}$$
$$\text{e para todo } a \in \Gamma, \quad \begin{pmatrix} a\, q_f \\ q_f \end{pmatrix}, \quad \begin{pmatrix} q_f\, a \\ q_f \end{pmatrix}$$

A máquina de Turing a seguir é bastante simples, e será usada como exemplo.



Para esta máquina de Turing com a entrada aab, criamos uma instância

do PCP tem as seguintes peças:

$$\begin{pmatrix} \# \\ \#q_0aab\# \end{pmatrix},$$
$$\begin{pmatrix} q_0a \\ bq_1 \end{pmatrix}, \begin{pmatrix} q_1a \\ aq_1 \end{pmatrix}, \begin{pmatrix} aq_1b \\ q_fab \end{pmatrix}, \begin{pmatrix} bq_1b \\ q_fbb \end{pmatrix},$$
$$\begin{pmatrix} a \\ a \end{pmatrix}, \begin{pmatrix} b \\ b \end{pmatrix}, \begin{pmatrix} \# \\ \# \end{pmatrix},$$
$$\begin{pmatrix} a\,q_f \\ q_f \end{pmatrix}, \begin{pmatrix} b\,q_f \\ q_f \end{pmatrix}, \begin{pmatrix} q_f\,a \\ q_f \end{pmatrix}, \begin{pmatrix} q_f\,b \\ q_f \end{pmatrix}, \begin{pmatrix} q_f\#\# \\ \# \end{pmatrix}$$

Usamos a peça inicial,

$$\begin{pmatrix} \# \\ \#q_0aab\# \end{pmatrix}$$

Para termos uma solução para o PCP, as cadeias superior e inferior devem ser iguais. A cadeia superior é #, e a inferior é $\#q_0aab\#$. Para obter um encaixe, é necessário inserir outra peça, *que comece com os mesmos símbolos da cadeia inferior*. A única peça que pode ser usada é $\binom{q_0a}{bq_1}$:

$$\begin{pmatrix} \# \\ \#q_0aab\# \end{pmatrix}\begin{pmatrix} q_0a \\ bq_1 \end{pmatrix}$$

Agora as cadeias são $\#q_0a$ e $\#q_0aab\#bq_1$. Falta completar o final da cadeia superior, então usamos as peças de único símbolo:

$$\begin{pmatrix} \# \\ \#q_0aab\# \end{pmatrix}\begin{pmatrix} q_0a \\ bq_1 \end{pmatrix}\begin{pmatrix} a \\ a \end{pmatrix}\begin{pmatrix} b \\ b \end{pmatrix}\begin{pmatrix} \# \\ \# \end{pmatrix}$$

As duas cadeias são $\#q_0aab\#$ e $\#q_0aab\#bq_1ab\#$. Mais alguns passos e chegamos a

$$\begin{pmatrix} \# \\ \#q_0aab\# \end{pmatrix}\begin{pmatrix} q_0a \\ bq_1 \end{pmatrix}\begin{pmatrix} a \\ a \end{pmatrix}\begin{pmatrix} b \\ b \end{pmatrix}\begin{pmatrix} \# \\ \# \end{pmatrix}\begin{pmatrix} b \\ b \end{pmatrix}\begin{pmatrix} q_1a \\ aq_1 \end{pmatrix}\begin{pmatrix} b \\ b \end{pmatrix}\begin{pmatrix} \# \\ \# \end{pmatrix}$$

As duas cadeias agora são

$$\#q_0aab\#bq_1ab\#$$
$$\#q_0aab\#bq_1ab\#baq_1b\#$$

Continuando, chegaremos a

$$\begin{pmatrix} \# \\ \#q_0aab\# \end{pmatrix}\begin{pmatrix} q_0a \\ bq_1 \end{pmatrix}\begin{pmatrix} a \\ a \end{pmatrix}\begin{pmatrix} b \\ b \end{pmatrix}\begin{pmatrix} \# \\ \# \end{pmatrix}\begin{pmatrix} b \\ b \end{pmatrix}\begin{pmatrix} q_1a \\ aq_1 \end{pmatrix}\begin{pmatrix} b \\ b \end{pmatrix}\begin{pmatrix} \# \\ \# \end{pmatrix}\begin{pmatrix} b \\ b \end{pmatrix}\begin{pmatrix} aq_1b \\ q_fab \end{pmatrix}\begin{pmatrix} \# \\ \# \end{pmatrix}$$

Temos um estado final, e sabemos que a máquina para. Mas as cadeias são

$$\#q_0aab\#bq_1ab\#baq_1b\#$$
$$\#q_0aab\#bq_1ab\#baq_1b\#bq_fab\#,$$

e a cadeia inferior ainda tem um trecho a mais que a superior. Usamos as peças com estado final.

$$\binom{\#}{\#q_0aab\#}\binom{q_0a}{bq_1}\binom{a}{a}\binom{b}{b}\binom{\#}{\#}\binom{b}{b}\binom{q_1a}{aq_1}\binom{b}{b}\binom{\#}{\#}\binom{b}{b}\binom{aq_1b}{q_fab}\binom{\#}{\#}$$
$$\binom{b}{b}\binom{q_fa}{q_f}\binom{b}{b}\binom{\#}{\#}\binom{bq_f}{q_f}\binom{b}{b}\binom{\#}{\#}\binom{q_fb}{q_f}\binom{\#}{\#}\binom{q_f\#\#}{\#}$$

É possível demonstrar, por indução na quantidade de configurações, que a construção da instância do PCP sempre funciona como descrevemos, e que de fato a instância do PCP terá solução se e somente se a máquina de Turing parar. □

A seguir definimos o problema do ladrilhamento, que consiste em encaixar peças de quebra-cabeça de forma a cobrir o primeiro quadrante no plano cartesiano.

Há um conjunto X de peças quadradas, e cada lado de uma peça tem uma cor diferente. Só é permitido encaixar as peças de maneira que cores iguais fiquem adjacentes. Uma peça é posta encaixada na origem.

Uma solução para este problema é uma função que leva posições do plano em tipos de peça, $f : \mathbb{N} \times \mathbb{N} \to X$.

Definição 5.13 (Problema do Ladrilhamento)**.** ⧫

Teorema 5.14. *O problema do ladrilhamento é indecidível.*

Teorema 5.15. *Determinar se uma gramática livre de contexto é ambígua é um problema indecidível.*

Demonstração. Reduziremos o PCP ao problema de determinar a ambiguidade de gramáticas livres de contexto.

Seja

$$\binom{\alpha_1}{\beta_1}, \binom{\alpha_2}{\beta_2}, \cdots, \binom{\alpha_n}{\beta_n}$$

uma instância qualquer do PCP usando um alfabeto Σ. Sejam $a_1, \ldots, a_n$ símbolos não pertencentes a Σ. Construímos, a partir da instância do PCP, uma gramática.

$$S ::= A \mid B$$
$$A ::= \alpha_i A a_i \mid \alpha_i a_i$$
$$B ::= \beta_i B a_i \mid \beta_i a_i$$

Se a instância do PCP tiver uma solução $j_1, j_2, \ldots, j_k$, as duas derivações diferentes a seguir resultam na mesma palavra.

$$S \Rightarrow A \Rightarrow^* \alpha_{j_1} \alpha_{j_2} \ldots \alpha_{j_k} a_{j_k} \ldots a_{j_2} a_{j_1}$$
$$S \Rightarrow B \Rightarrow^* \beta_{j_1} \beta_{j_2} \ldots \beta_{j_k} a_{j_k} \ldots a_{j_2} a_{j_1}$$

Isso acontece porque, sendo $j_1, j_2, \ldots, j_k$ solução para a instância do PCP, as duas palavras serão iguais, porque

$$\alpha_{j_1} \alpha_{j_2} \ldots \alpha_{j_k} = \beta_{j_1} \beta_{j_2} \ldots \beta_{j_k}$$

Se a instância do PCP não tiver solução, a gramática não será ambígua, porque as regras para A e para B gerarão conjuntos diferentes de palavras. □

Teorema 5.16. *Sejam L_1 e L_2 duas linguagens livres de contexto. Determinar se $L_1 \cap L_2 = \emptyset$ é indecidível.*

Teorema 5.17 (de Rice)**.** *Seja L uma linguagem reconhecida por alguma máquina de Turing. Seja P uma propriedade não-trivial sobre L. Então P é indecidível.*

Exercícios

Ex. 59 — Prove que o Problema da Correspondência de Post é decidível se usarmos um alfabeto com um único símbolo.

Ex. 60 — Se não permitirmos repetições de índices na solução do PCP, ele continua indecidível?

Ex. 61 — Prove que EQ_{GLC} é indecidível. Use o problema $TODAS_{GLC}$:

$$TODAS_{GLC} = \{G \mid G \text{ aceita todas as palavras sobre o alfabeto definido}\}$$

Ou seja, $TODAS_{GLC} = \{G \mid L(G) = \Sigma^*\}$

Ex. 62 — (Um pouco difícil) Seja

$$R = \{M \mid M \text{ é máquina de Turing, e M aceita } w^R \text{ se e somente se M aceita } w\}$$

– ou seja, R é a linguagem das máquinas de Turing que sempre aceitam palavras e suas reversas (as máquinas de R **não** aceitam uma palavra w sem aceitarem w^R também). Mostre que a linguagem R é indecidível.
Dica: tente reduzir A_{MT}: construa um algoritmo para A_{MT}, usando a máquina de Turing que supostamente existe para R. (Seu algoritmo começaria

lendo M, w, e construindo uma máquina que aceita os reversos das palavras de M. Depois, concatene M com sua nova máquina...)

Capítulo 6

Complexidade

Dentre os problemas decidíveis, há muitos para os quais conhecemos somente algoritmos muito pouco eficientes – tão pouco eficientes que os chamamos de "intratáveis". Neste Capítulo classificamos problemas de acordo com os algoritmos que temos para solucioná-los.

6.1 Crescimento assintótico de funções

Para classificar problemas em fáceis ou intratáveis, será necessário definir rigorosamente estes conceitos. Diremos que um problema é fácil quando a quantidade de passos que ele executa, *para uma entrada de tamanho* n*, é dada por uma função* $T(n)$ *que não cresce muito rápido*. Começamos, portanto, estabelecendo uma linguagem usada para expressar fatos sobre o crescimento de funções.

Definição 6.1 (O, Ω, Θ)**.** Sejam $f, g : \mathbb{N} \to \mathbb{R}$ duas funções.

Dizemos que f é $O(g)$, e que g é $\Omega(f)$, se existem constantes k e Z tais que

$$|f(n)| \leq k|g(n)|, \quad \text{para todo } n > Z.$$

Se f é $O(g)$ e f é $\Omega(g)$, então as duas funções têm crescimento assintótico equivalente, e dizemos que f é $\Theta(g)$, e que g é $\Theta(f)$. ♦

Exemplo 6.2. Sejam

$$f(n) = 2n + 1$$
$$g(n) = 3n^2$$
$$h(n) = 2^n - 4$$

Então

- $f(n)$ é $O(g(n))$, porque para $n \geq 1$, $f(n) \leq g(n)$.
- $g(n)$ é $O(h(n))$, porque para para qualquer $n \geq 8$, $g(n) \leq h(n)$. ◀

Teorema 6.3. *A relação* O *é*

i) reflexiva: f *é* $O(f)$*;*

ii) e transitiva: se f *é* $O(g)$*, e* g *é* $O(h)$*, então* f *é* $O(h)$*.*

Como consequência, Ω *também é.*

Exemplo 6.4. Sejam

$$f(n) = n \log(n)$$
$$g(n) = n^2$$
$$h(n) = 2^n$$

$f(n)$ é $O(g(n))$, porque $n \log(n) < n^2$ para $n \geq 1$.
$g(n)$ é $O(h(n))$, porque $n^2 < 2^n$ para $n \geq 5$.
Como consequência, $f(n)$ é $O(h(n))$. ◀

6.2 Complexidade de espaço e de tempo

Definimos agora complexidade de espaço e de tempo.

Definição 6.5 (Complexidade de espaço)**.** Seja M uma máquina de Turing. Seja $S(n)$ a função que, dado o tamanho da entrada, determina quantas posições da fita, no máximo, serão usadas pela máquina de Turing. Dizemos que S é a função que dá a *complexidade de espaço* desta máquina de Turing. ⧫

Exemplo 6.6. A máquina de Turing que realiza a multiplicação de um número binário por dois tem complexidade de espaço $n + 1$, porque quando sua entrada tiver n dígitos, a saída terá $n + 1$, e *em nenhum momento a máquina usa mais que* $n + 1$ *posições para realizar a computação.*

Como $f(n) = n + 1$ é $O(n)$, dizemos que a complexidade de espaço dessa máquina de Turing é $O(n)$. ◀

Definição 6.7 (Complexidade de tempo)**.** Seja M uma máquina de Turing. Seja $T(n)$ a função que, dado o tamanho da entrada, determina a quantidade máxima de passos que M executará. Dizemos que T é a função que dá a *complexidade de tempo* desta máquina de Turing. ⧫

Exemplo 6.8. A máquina de Turing que reconhece palavras da forma $a^n b^n$ percorre a fita repetidas vezes até parar.

- Na primeira vez, anda n posições para a direita, marcando um X no primeiro a e um Y no primeiro b, depois anda mais n para voltar
- Na segunda vez, anda n posições para a direita até chegar ao próximo b, e volta n;
- Continua, andando n posições em cada passo.

Assim, a máquina realiza $2n^2$ deslocamentos, mais n deslocamentos à direita no último passo, saindo do último X e chegando ao branco no final da fita. A complexidade de tempo dessa máquina de Turing é, portanto,

$$T(n) = 2n^2 + n.$$

Dizemos que a complexidade de tempo da máquina é $O(n^2)$. É claro que poderíamos também dizer que a complexidade é $O(n^3)$, ou ainda, $O(n!)$ – mas escolhemos sempre usar a medida de complexidade mais justa possível. ◀

As definições dadas nesta seção são conhecidas como complexidade de espaço e de tempo *para o pior caso*, porque as funções que estabelecem devem funcionar para todas as instâncias, contabilizando a quantidade máxima possível de posições de fita ou de movimentos. É possível também definir a complexidade *no caso médio*, ou *no melhor caso*, mas não o faremos neste texto.

Passamos a partir de agora a usar algoritmos ao invés de máquinas de Turing.

Exemplo 6.9. Calcularemos a complexidade do problema de multiplicar uma matriz por um vetor.

Para calcular $M \cdot v$, onde M é uma matriz $n \times n$ e v é um vetor com n componentes, usamos o algoritmo a seguir

```
para i de 1 a n:
    para j de 1 a n:
        para k de 1 a n:
            w_i ← w_i + M_{i,j} v_j
```

O laço externo (i) será executado uma vez, e executará n vezes o laço j. O laço j executará n vezes, e em cada vez, executará o laço k, que executa a última a linha n vezes.

O total de vezes que a última linha será executada é n^3, que é a complexidade de tempo deste algoritmo. ◀

Exemplo 6.10. O problema de soma de subconjuntos tem como entrada um número $s \in \mathbb{N}$ e um conjunto finito A de naturais. Resolver o problema é

determinar se existe um subconjunto de A cuja soma seja exatamente igual a s.

Um algoritmo ingênuo e bastante simples para resolver este problema consiste em enumerar todas as combinações possíveis de números em A, verificando, para cada seleção de números, se a soma é igual a s.

- enumerar os subconjuntos de tamanho 1 (complexidade igual a $|A|$), e para cada um deles verificar se a soma é s. No total a complexidade será $|A|$
- enumerar os subconjuntos de tamanho 2 (complexidade igual a $\binom{|A|}{2}$), e verificar a soma. A complexidade será $2\binom{|A|}{2}$
- ...
- enumerar os subconjuntos de tamanho k (complexidade igual a $\binom{|A|}{k}$), e verificar a soma. A complexidade será $k\binom{|A|}{k}$
- ...
- enumerar os subconjuntos de tamanho s (complexidade igual a $\binom{|A|}{s}$). A complexidade será $s\binom{|A|}{s}$

Assim, a complexidade de tempo deste algoritmo será

$$T(|A|, s) = \sum_{j=1}^{s} j\binom{|A|}{s}.$$

Claramente, $T(|A|, s)$ não é polinomial em $|A|$. ◀

6.3 Classes de complexidade P e NP

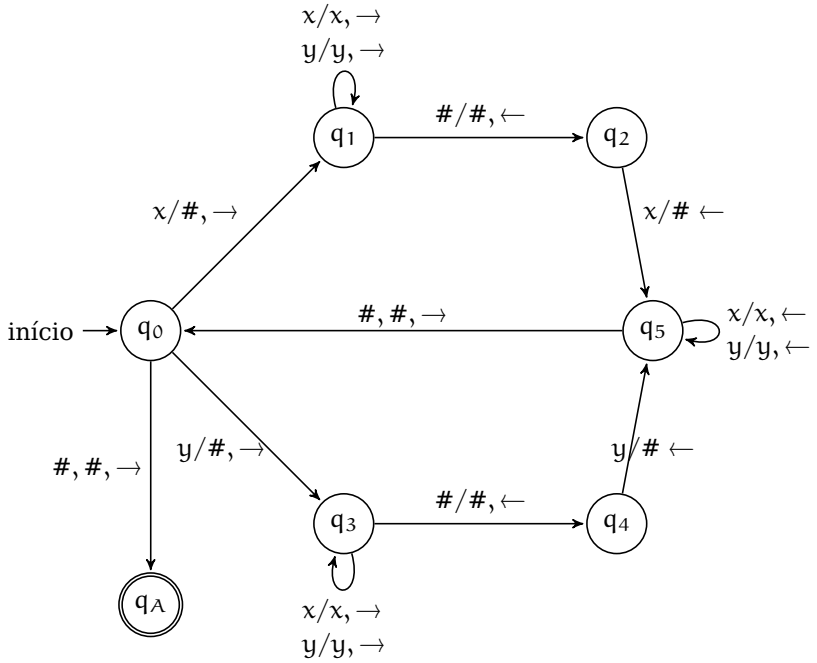
Os problemas computacionais decidíveis são classificados em *classes de complexidade*, de acordo com suas complexidades de tempo espaço. Neste texto trataremos somente de complexidade de tempo.

Definição 6.11 (Classe P de problemas)**.** Um problema de decisão está na classe P se existe máquina de Turing *determinística* (ou algoritmo) que o decida, com complexidade de tempo $O(n^k)$, para algum k – o que é equivalente a dizer que tem complexidade de tempo *dada por um polinômio*, porque é $O(p(n))$, para algum polinômio n. ♦

Exemplo 6.12. O problema de verificar se uma cadeia é palíndroma de comprimento par está em P, porque existe um algoritmo determinístico que o resolve em tempo polinomial:

```
palindroma_par?(w):
    para i de 1 até |w|/2:
        se w_i ≠ w_{|w|−i} REJEITE
    ACEITE
```

De acordo com a tese de Church-Turing, há também uma máquina de Turing que resolve este problema:



Há algumas observações importantes sobre este exemplo:

- tanto o algoritmo como a máquina de Turing são determinísticos;
- tanto o algoritmo como a máquina de Turing são *decisores* (sempre param);
- *não* apresentamos uma gramática irrestrita, porque as linguagens delas são recursivamente enumeráveis, e não recursivas;
- o algoritmo tem complexidade de tempo $O(n)$, mas a máquina de Turing tem complexidade de tempo $O(n^2)$. No entanto, *os dois têm complexidade de tempo polinomial.* ◀

O exemplo deixa também claro porque usamos algoritmos, e não máquinas de Turing, para discutir problemas computacionais – as máquinas de Turing são grandes demais. Elas são úteis na demonstração de teoremas e discussão dos conceitos fundamentais relacionados à computabilidade, mas não em discussões sobre problemas computacionais específicos.

Exemplo 6.13. O problema de verificar se um número é quadrado perfeito está em P, porque há um algoritmo determinístico polinomial para resolvê-lo:

```
quadrado(n):
    i ← 2
    repita:
        q ← i · i
        se q = n ACEITE
    até q > n
    REJEITE
```

O algoritmo é determinístico, e sua complexidade de tempo é polinomial (é $O(\sqrt{n})$). ◀

Definição 6.14 (Classe NP de problemas)**.** Um problema de decisão está na classe NP se existe máquina de Turing *não-determinística* (ou algoritmo) que o decida, com complexidade de tempo $O(p(n))$, onde p é algum polinômio. ♦

Exemplo 6.15. *SOMA_DE_SUBCONJUNTO:* dado um conjunto de números naturais S e um número $k \in \mathbb{N}$, determinar se existe um subconjunto $X \subseteq S$ com soma igual a k.

Não sabemos dizer se o problema da soma de subconjunto está em P, mas ele certamente está em NP: se tivermos um computador não-determinístico, ele poderá escolher não-deterministicamente o subconjunto, e resta somente verificar se a soma desse subconjunto é igual a s.

```
soma_subconjunto(S, k)
    encontre não-deterministicamente X ⊆ S
    se ∑_{x∈X} x = k ACEITE
    senão REJEITE
```

Essa verificação pode ser feita em tempo proporcional ao tamanho do conjunto, e portanto o algoritmo é *não-determinístico e tem complexidade de tempo polinomial.* ◀

Exemplo 6.16. *CICLO_HAMILTONIANO:* dado um grafo, determinar se ele tem um ciclo hamiltoniano – um caminho que passa por todos os vértices, sem repetir vértices (a não ser pelo primeiro e último, que são iguais). ◀

Exemplo 6.17. *SAT:* dada uma fórmula booleana com n variáveis sem quantificadores, usando apenas conectivos **e** e **ou**, determinar se existe uma valoração das variáveis (ou seja, uma atribuição de valor verdadeiro ou falso a cada uma) que torne a fórmula verdadeira. ◀

Exemplo 6.18. *PARTIÇÃO:* dado um conjunto S e uma função que atribui um valor a cada elemento de S, $z : S \to \mathbb{N}_+$, decidir se existe como particioná-lo em dois, de forma que a soma dos valores de cada uma das duas partições seja igual à da outra.

Este problema está na classe NP, porque pode ser resolvido por um algoritmo não-determinístico polinomial.

```
partição(S, z):
    não-deterministicamente esolha X ⊆ S
    se ∑_{x∈X} z(x) = ∑_{y∈S−X} z(y) ACEITE
    senão REJEITE
```

A complexidade de tempo do algoritmo é $O(|S|)$, porque as somas são realizadas sobre todos os elementos de S. ◀

Exemplo 6.19. *MOCHILA:* dados uma mochila com capacidade C, um conjunto A de objetos, uma função $v : A \to \mathbb{R}_+$ que atribui valor aos objetos, e uma função $w : A \to \mathbb{R}_+$, que atribui peso aos objetos, e um valor mínimo $K \in \mathbb{R}$, determinar se existe uma lista de objetos que caiba na mochila com valor mínimo V, ou seja, se existe $Z \subseteq A$, tal que

$$\sum_{z \in Z} v(z) \geq K$$
$$\sum_{z \in Z} w(z) \leq C$$

O problema da mochila está em NP, porque o algoritmo não-determinístico a seguir, que tem complexidade de tempo polinomial, o resolve.

```
mochila(A, C, K, v, w):
    não-deterministicamente obtenha Z ⊆ A
    se ∑_{z∈Z} v(z) ≥ K e ∑_{z∈Z} w(z) ≤ C ACEITE
    senão REJEITE
```

O algoritmo tem complexidade $O(|A|)$. ◀

Exemplo 6.20. *CLIQUE:* Uma *clique* em um grafo G é um subgrafo de G onde todos os vértices estão ligados a todos (ou seja, uma clique é um subgrafo completo). O problema de determinar se um grafo tem uma clique de tamanho maior ou igual a k está na classe NP, porque pode ser resolvido pelo algoritmo não-determinístico a seguir, que tem complexidade de tempo polinomial.

```
clique(V, E, C, k)
    não-deterministicamente obtenha os vértices C ⊆ V
    se |C| < k REJEITE
```

```
para todo v ∈ C:
    para todo u ∈ C:
        se (u, v) ∉ E REJEITE
ACEITE
```

O algoritmo tem complexidade de tempo $O(|V|^2)$, porque no pior caso $C = V$, e todos os pares de vértices serão verificados. ◀

6.4 Propriedades

O Exercício 68 pede a demonstração de propriedades de P, enunciadas no Teorema 6.21.

Teorema 6.21. *A linguagem* P *é fechada para união, concatenação e complemento.*

6.5 Reduções e classe NP-completo

A classe P é conhecida como a dos problemas "fáceis", por ser possível implementar algoritmos determinísticos polinomiais em dispositivos reais (e não hipotéticos, como no caso de algoritmos não-determinísticos). Quando um problema computacional é no mínimo tão difícil quanto a classe NP, dizemos que este problema é NP-*difícil*. Se este problema também estiver em NP, ele é chamado de NP-completo.

Definição 6.22 (problema NP-completo)**.** Um problema Π é NP-completo se

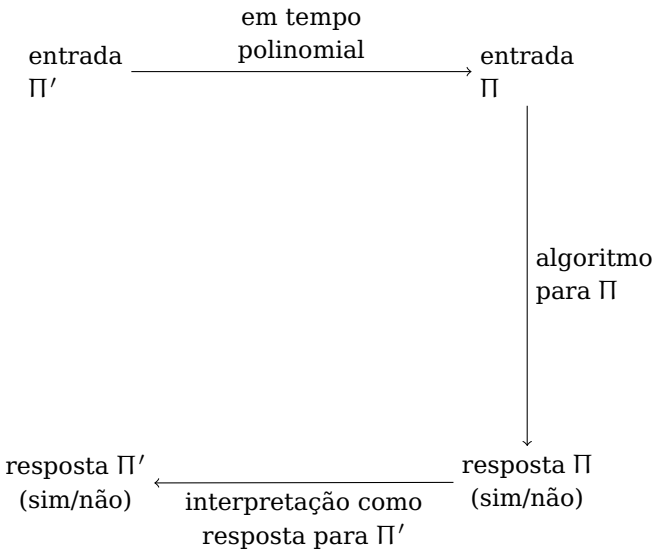
i) Π está em NP;

ii) Todo problema $\Pi' \in$ NP pode ser reduzido a Π, *sendo a redução (transformação da entrada de* Π' *em uma entrada de* Π*) deve, ela mesma, ter complexidade de tempo polinomial.* ⧫

De acordo com a definição, para provar que um problema Π é NP-completo são necessários dois passos: primeiro, provar que o problema está em NP. Já vimos, na Seção 6.3, como demonstrar que um problema está nessa classe. O segundo passo é mostrar que todo problema em NP se reduz a Π. A maneira simples de fazer isso é demonstrar que *um* problema Π', NP-completo se reduz a Π. Isso porque todo problema em NP já se reduz a Π' (ele é NP-completo), portanto todos também serão reduzidos a Π.

$$\Pi \longleftarrow \Pi' \longleftarrow \text{outros / NP}$$

A redução é de Π' para Π é um método (algoritmo), determinístico, e que use tempo polinomial, que receba uma entrada de Π; e a transforme em entrada de Π, de forma que seja possível usar um algoritmo que resolve Π. A saída desse algoritmo, depois, deve ser interpretada como solução para $'Pi'$.



O Exemplo 6.23 mostra detalhadamente uma demonstração de NP-completude para um problema computacional (o problema da mochila).

Exemplo 6.23. Dado que o problema da partição é NP-completo, provamos que o da mochila também é.

i) *MOCHILA* $\in$ NP – já verificamos no Exemplo 6.19.

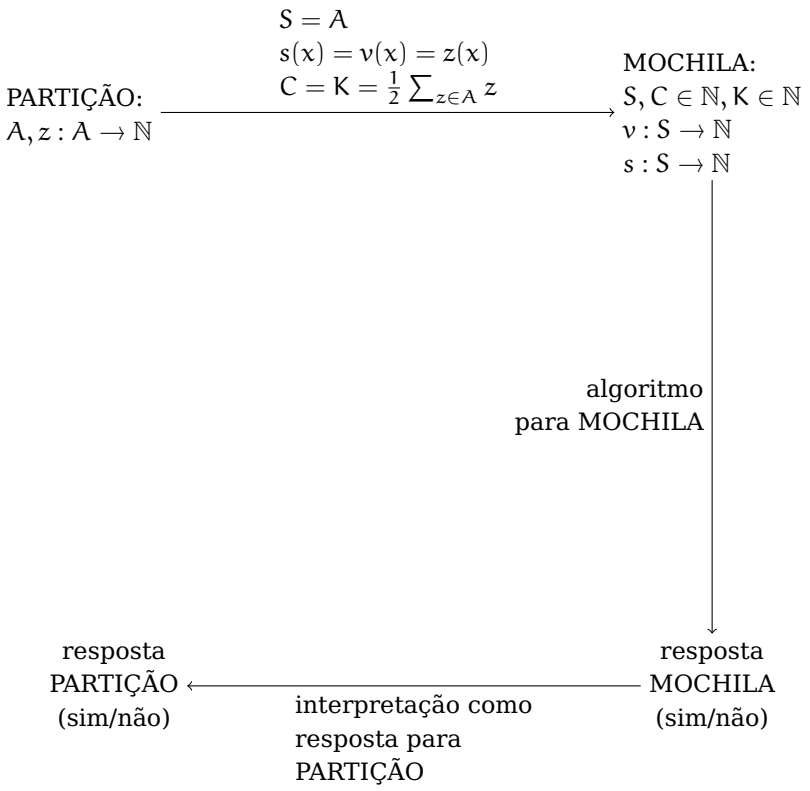
ii) Todo problema de NP se reduz, em tempo polinomial, ao da mochila. Será suficiente reduzir *um* problema NP-completo ao da mochila, porque se como o problema escolhido é NP-completo, todos os outros em NP já se reduzem a ele.

   Transformamos uma instância do problema da partição em uma instância do problema da mochila:

$$s(a) = v(a) = z(a), \quad \forall a \in A$$
$$C = K = \frac{1}{2} \sum_{a \in A} z(a)$$

   As restrições exigem que incluamos na mochila elementos com valor igual ou maior que K, mas também com peso igual ou menor que K.

Como peso e valor são iguais, teremos que tanto o peso como o valor deverão ser exatamente K, que é a metade do total no conjunto A.



É importante que a redução – ou seja, calcular $1/2 \sum_{a \in A} z(a)$, e copiar A como S, e copiar a função z em v e s, pode ser feito deterministicamente e em tempo polinomial.

A idéia subjacente é: se houver um algoritmo determinístico (e que portanto possa ser implementado em dispositivos reais) que resolve o problema da mochila em tempo polinomial, então haveria um outro para o problema da partição:

```
partição(A, z):
    soma ← 1/2 Σ_{a∈A} z(a)
    S ← A
    C ← soma
    K ← soma
    v ← z
    s ← z
    resposta ← mochila(S, C, K, v, s)  /* em tempo polinomial */
```

```
se (resposta = ACEITE) então ACEITE
senão REJEITE
```

Isso implica que, se houver um algoritmo determinístico polinomial para o problema da mochila, haverá também para todos os problemas na classe NP. A diferença entre eles estará na redução, antes de usar o algoritmo da mochila. ◀

Exercícios

Ex. 63 — Quais afirmações são verdadeiras?

a) $3n$ é $O(n)$

b) $3n$ é $\Theta(n)$

c) n^3 é $O(n^2)$

d) 2^{3n} é $O(2^n)$

e) n^2 é $O(n \log^2 n)$

f) $n!$ é $O(3^n)$

g) $n!$ é $\Omega(3^n)$

h) $n!$ é $\Omega(n!)$

Ex. 64 — Determine a complexidade de tempo dos algoritmos a seguir.

a) Para comparar duas sequências de números de tamanho n, comparo um a um seus elementos. Quando encontrar dois elementos diferentes na mesma posição ($a_i \neq b_i$), respondo "não". Se chegar ao final das duas sequências sem que diferenças sejam encontradas, respondo "sim".

b) Em um grafo com m arestas e n nós, para verificar se existe caminho entre dois nós x e y, gero todas as sequências possíveis de vértices com arestas levando de um a outro (todos os possíveis caminhos). Se uma sequência tiver x e y nas duas extremidades, respondo "sim". Se tiver gerado todas as sequências sem encontrar caminho entre x e y, respondo "não".

c) Para o problema no item (b):

```
alcança(x, y):
    se x = y pare com resposta SIM
    senão
        para todo vizinho z de x:
            se alcança(z, y) pare com resposta SIM
    pare com resposta NÃO
```

Ex. 65 — Tendo como entrada um vetor de n posições, quero ordená-lo. Determine a complexidade de fazê-lo usando os algoritmos a seguir.

a) Enumero as maneiras diferentes de ordená-lo, e para cada uma, verifico se está ordenado.

b) Uso o "selection sort":

```
para i de 1 a n − 1
    para j de i + 1 a n
        se v_j < v_i então
            troque v_i ↔ v_j
```

c) Uso o "bubble sort":

```
trocou ← false
repita até trocou = true:
    trocou ← false
    para j de i a n − 1
        se v_j < v_i então
            troque v_i ↔ v_j
            trocou ← true
```

Ex. 66 — Mostre que os problemas a seguir estão em NP. Se conseguir, mostre também que estão em P (mas não é esperado que seja possível mostrar que todos estão em P). Dê atenção aos detalhes.

a) Determinar se um número binário n de k bits é composto.

b) Calcular o máximo divisor comum de dois inteiros positivos.

c) Encontrar uma clique de tamanho k em um grafo com n vértices e m arestas. (Uma *clique* é um subconjunto dos vértices do grafo, sendo que cada vértice na clique ligam-se a todos os outros que também estão nela)

d) Determinar se um vértice x alcança outro vértice y em um grafo.

Ex. 67 — Um *conjunto independente* em um grafo é um subconjunto de vértices sendo que cada vértice no subconjunto não tem ligação com nenhum outro que também esteja no conjunto independente.

a) Mostre uma redução do problema de algum problema NP-completo ao problema de encontrar um conjunto independente de tamanho k. Mostre um diagrama ilustrando sua redução.

b) Mostre que o problema do conjunto independente está em NP.

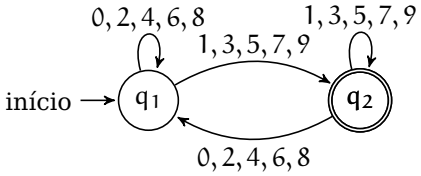
c) O que significam as respostas dos itens (a) e (b), juntas?

Ex. 68 — Prove o Teorema 6.21.

Apêndice A

Dicas e Respostas

Resp. (Ex. 5) — Basta exigir que o número termine em dígito ímpar.



Resp. (Ex. 17) — (Dica) Comece trocando estados finais com não finais em um autômato finito que representa a linguagem.

Resp. (Ex. 18) — (Dica) Há mais de uma maneira de demonstrar.

(i) Seja A um autômato reconhecendo uma linguagem regular. Modifique-o para que tenha um estado inicial isolado q_A e um único estado final isolado q_Z. Reverta as arestas e faça q_Z ser inicial e q_A ser final.

(ii) Seja r uma expressão regular representando a linguagem. Prove, por indução, que reverter as concatenações de r leva a uma expressão regular que representa a linguagem reversa.

(iii) Mude a gramática linear à direita para que seja linear à esquerda, trocando as produções $A ::= xB$ por $A ::= Bx$. Prove que a nova gramática representa a linguagem reversa.

Resp. (Ex. 19) — A gramática a seguir representa a linguagem 0^n1^n, que

não é regular.

$$S ::= 0A$$
$$S ::= \varepsilon$$
$$A ::= S1$$

Resp. (Ex. 28) — Tente indução no tamanho (quantidade de substituições) na derivação.

Resp. (Ex. 29) — (Rascunho) Tente indução em i: para $i = 1$, a única derivação possível é $S \Rightarrow a$, para algum a na linguagem da gramática. Depois, veja que, para $i > 1$ a árvore de derivação tem uma raiz com dois filhos (porque a gramática está na forma normal de Chomsky). As subárvores esquerda e direita tem caminhos de tamanho no máximo $i-1$, e derivam palavras de tamanho 2^{i-2} cada uma. Concatenado as palavras geradas por cada subárvore, obteremos uma palavra de tamanho no máximo $2\left(2^{i-2}\right) = 2^{i-1}$.

Resp. (Ex. 36) — Sim! Sabemos que a linguagem das palíndromas sobre qualquer alfabeto é livre de contexto. Agora, $\alpha\alpha^R$ é uma palíndroma; $\beta\beta^R$ é outra. A linguagem proposta é a concatenação das duas, portanto livre de contexto.

Mais ainda, é simples construir uma gramática que gere esta linguagem:

$$S ::= AA$$
$$A ::= 1A1 \mid 0A0 \mid \varepsilon$$

Resp. (Ex. 39) — Sim! A "quantidade finita de cadeias" é uma linguagem regular, e a diferença será livre de contexto.

Resp. (Ex. 44) — Tente primeiro com tamanho fixo, depois com tamanho menor ou igual a n. Para palavras com comprimento exatamente k e quantidade igual de as e bs, escolhemos as k/s posições dos as, e o resto será ocupado por bs:

$$\binom{k}{k/2}.$$

Para tamanho máximo n, contamos todos os *pares* de 2 até n.

$$|X_n| = \sum_{q=1}^{\lfloor n/2 \rfloor} \binom{2q}{q}.$$

Resp. (Ex. 46) — O símbolo inicial S precisa garantir que haverá pelo menos um par de símbolos que *não* casam antes de terminar a palavra.

$$S ::= aSa \mid bSb \mid aRb \mid bRa$$
$$R ::= aRa \mid bRb \mid aRb \mid bRa \mid a \mid b \mid \varepsilon$$

Resp. (Ex. 55) — Prove que as, bs e cs sempre ficarão em ordem, e que a quantidade de cada letra gerada é a mesma. Terá provado que a linguagem é $a^n b^n c^n$.

Resp. (Ex. 60) — Não, porque haverá uma quantidade finita de soluções ($\sum_{j=1}^{n} j!$, onde n é a quantidade de elementos).

Resp. (Ex. 63) — (a), (b), (g), (h)

Resp. (Ex. 65) — (a) $O(n!)$. (b) $O(n^2)$. (c) $O(n^2)$ – no caso (c) é necessário pensar em quantas passadas são necessárias para levar um elemento fora de ordem ao seu lugar, no pior caso.

Resp. (Ex. 66) — (Dicas apenas)
(a) Sim, porque, tendo os fatores, podemos em tempo polinomial multiplicá-los e verificar se o resultado é n.
(b) MDC está em P, portanto também está em NP. Mas para mostrar que está em P é necessário mostrar que o algoritmo que o calcula sempre para, e que esse algoritmo para em tempo polinomial.
(c) Está em NP, porque pode-se verificar em tempo polinomial se cada vértice do subconjunto está ligado a todos os outros.
(d) O algoritmo do Exercício 64.b é determinístico e executa em tempo polinomial, logo o problema está em P. Como está em P, também está em NP.

Resp. (Ex. 67) — Tome a descrição de um grafo G. Crie outro grafo $\overline{G}$ (chamado de complemento de G), invertendo a presença de arestas (se havia aresta (x, y) em G, não inclua esta aresta em $\overline{G}$; se não havia aresta (x, y) em G, inclua esta aresta em $\overline{G}$). Uma clique em G é um conjunto independente em $\overline{G}$, e vice-versa. Como a redução (a *tradução da entrada para o problema da clique em* G *para o problema do conjunto independente em* $\overline{G}$) pode se feita em tempo polinomial, podemos usar um algoritmo que resolva conjunto independente em $\overline{G}$ para resolver o problema da clique em G.

Ficha Técnica

Este texto foi produzido inteiramente em LaTeX em sistema Debian GNU/Linux. Os diagramas foram criados sem editor gráfico, usando diretamente o pacote Ti*k*Z. O ambiente Emacs foi usado para edição do texto LaTeX.

Índice Remissivo