4 Máquinas de Turing

4.1 DEFINIÇÃO DA MÁQUINA DE TURING

Vimos nos últimos dois capítulos que nem os autômatos finitos, nem autômatos de pilha podem ser considerados modelos gerais de computação uma vez que não são sequer capazes de reconhecer linguagens simples, como $\{a^n b^n c^n : n \geq 0\}$. Neste capítulo, estudaremos dispositivos capazes de não apenas reconhecer essa como também outras linguagens mais complexas. Apesar de esses dispositivos, denominados **máquinas de Turing** em homenagem ao seu inventor, Alan Turing (1912—1954), serem mais genéricos do que os autômatos previamente estudados, seu aspecto básico é similar ao desses autômatos. Uma máquina de Turing consiste em um controle finito, uma fita e um cabeçote que pode ser utilizado para efetuar leituras e gravações na fita. A definição formal das máquinas de Turing e de sua operação será feita no mesmo estilo matemático utilizado para os autômatos finitos e de pilha. Assim, para conseguir um poder computacional adicional e a generalidade de função que as máquinas de Turing apresentam, não precisaremos migrar para um modelo computacional inteiramente diferente.

Contudo, as máquinas de Turing não constituem simplesmente mais uma classe de autômatos, a serem suplantados posteriormente por um outro tipo de autômato ainda mais poderoso. Veremos neste capítulo, que, por mais elementares que as máquinas de Turing pareçam ser, nenhuma tentativa para fortalecê-las se mostra eficaz. Por exemplo, estudaremos variantes das máquinas de Turing que se utilizam de muitas fitas, máquinas com dispositivos de memória mais sofisticados que podem ser lidos ou gravados em regime de *acesso aleatório*, semelhantes ao dos computadores reais. Intrigantemente, esses dispositivos não se mostram mais poderosos, em termos de poder computacional, que as máquinas de Turing básicas. Esses resultados são mostrados lançando-se mão de técnicas de simulação: podemos converter em máquina de Turing básica equivalente a qualquer máquina estendida. Portanto, qualquer computação que possa ser realizada em uma máquina de tipo mais sofisticado pode também ser realizada com máquinas de Turing básicas. Além disso, próximo ao final deste capítulo definimos um dispositivo gerador, que é uma generalização da gramática livre de contexto. Demonstra-se que esse dispositivo também é equivalente à máquina de Turing. De uma perspectiva totalmente diferente, discutimos também a questão da computabilidade de uma função numérica (tal como $2^x + x^2$) e acabamos concluindo que tal notação se mostra, mais uma vez, equivalente às máquinas de Turing!

Assim, as máquinas de Turing parecem formar uma classe *estável* e *máxima* de dispositivos computacionais quanto às computações que podem

realizar. De fato, no próximo capítulo ampliaremos esta visão, amplamente aceita, de que qualquer forma aceitável de expressão das idéias contidas em um "algoritmo" seja, em última instância, equivalente à que se pode obter empregando-se uma máquina de Turing.

Mas isso é um resultado que será obtido mais adiante. Neste momento, o ponto importante a ser frisado, nesta introdução, é que as máquinas de Turing foram idealizadas para satisfazerem simultaneamente os seguintes critérios:

(a) Elas devem operar como autômatos, isto é, sua construção e função devem aderir ao paradigma dos dispositivos reconhecedores previamente estudados.
(b) Elas devem ser simples de descrever, de definir e de entender.
(c) Elas devem apresentar a maior generalidade possível quanto às computações que podem realizar.

Analisemos mais de perto essas máquinas. Em essência, uma máquina de Turing consiste de um controle de estados finitos associado a uma unidade de fita (veja a Figura 4-1). A comunicação entre as duas é proporcionada por um simples cabeçote, responsável pela leitura dos símbolos da fita, e que é também utilizado para modificar os símbolos encontrados na fita. A unidade de controle opera em passos discretos; a cada passo, ela realiza duas operações, que dependem do seu estado atual e do símbolo da fita correntemente lido pelo cabeçote de leitura/gravação:

(1) Levar a unidade de controle para um novo estado.
(2) (a) Gravar um símbolo na célula corretamente apontada pelo cabeçote, substituindo algum símbolo lá encontrado, ou então
    (b) Mover o cabeçote de leitura/gravação para apontar uma célula à esquerda ou à direita na fita em relação à posição corrente.

A fita apresenta-se fisicamente delimitada em sua extremidade esquerda, mas estende-se indefinidamente para a direita. Para impedir que a máquina mova seu cabeçote até a extremidade esquerda da fita, vamos admitir que a extremidade esquerda da fita esteja sempre marcada por um símbolo especial, denotado por $\triangleright$; convencionamos também que todas as nossas máquinas de Turing sejam projetadas de tal modo que, sempre que o cabeçote encontrar um símbolo $\triangleright$, imediatamente a sua posição seja movida para a direita. Além disso, iremos utilizar os símbolos especiais $\leftarrow$ e $\rightarrow$ para denotar o movimento do cabeçote para a esquerda e para a direita, respectivamente, e também que esses dois símbolos não sejam membros de qualquer alfabeto a ser utilizado em nosso estudo.

Uma máquina de Turing é alimentada gravando-se previamente a cadeia de entrada nas células mais à esquerda da fita, imediatamente à direita do símbolo $\triangleright$. O restante da fita fica preenchido, nesse momento, com símbolos que representam **espaços em branco**, aqui denotados por $\sqcup$. A máquina é livre para modificar o conteúdo da sua fita de entrada, de qualquer maneira que se considere apropriada, bem como para gravar símbolos nos infinitos espaços em branco encontrados após a cadeia de entrada, na parte direita da fita. Dado que a máquina pode mover o seu cabeçote somente uma célula de cada vez, con-

clui-se que, após qualquer computação finita, apenas um número finito d células da fita terá sido visitado.

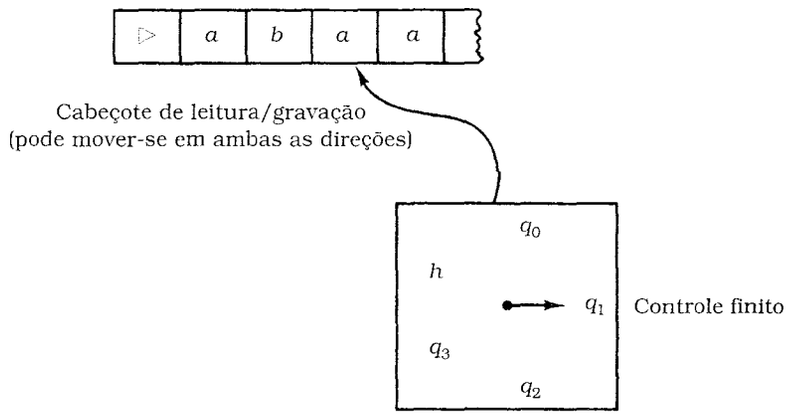


Figura 4-1

Podemos agora apresentar uma definição formal para a máquina de Turing

Definição 4.1.1: Uma máquina de Turing é uma quíntupla $(K, \Sigma, \delta, s, H)$, em

K é um conjunto finito de **estados**;

Σ é o alfabeto de entrada, que contém o símbolo de espaço em branco $\_$ o símbolo de extremidade esquerda $\triangleright$, mas que não contém os símbo $\leftarrow$ e $\rightarrow$;

$s \in K$ é o estado inicial;

$H \subseteq K$ é o conjunto de estados de parada;

δ, a função de transição, é uma função de $(K - H) \times \Sigma$ para $K \times (\Sigma \cup \{\leftarrow, \rightarrow$ tal que,

(a) para todos os $q \in K - H$, se $\delta(q, \triangleright) = (p, b)$, então $b = \rightarrow$

(b) para todos os $q \in K - H$ e $a \in \Sigma$, se $\delta(q, a) = (p, b)$, então $b \neq$

Se $q \in K - H$, $a \in \Sigma$ e $\delta(q, a) = (p, b)$, então M, quando no estado q e tend lido o símbolo corrente a, transitará para o estado p e, (1) se b for um símbo contido em Σ, M irá substituir na fita o símbolo corrente a pelo símbolo b ou, se b for um dos símbolos $\leftarrow$ ou $\rightarrow$, M moverá sua cabeça na direção convenc nada para o símbolo b. Uma vez que δ é uma função, a operação de M é deter minística, e irá parar somente quando M transitar para alguns dos estados d parada. Notem-se a exigências (a) para a função δ: Quando o final esquerd da fita $\triangleright$ é atingido, o cabeçote *deve* mover-se para a direita. Desse modo, o marc dor da extremidade esquerda da fita ($\triangleright$) nunca será apagado, e M nunca ultra passará a extremidade esquerda de sua fita. (b) M nunca grava um símbol fazendo dele um marcador inequívoco da extremidade esquerda da fita. Em outras palavras, podemos interpretar $\triangleright$ simplesmente como uma "barreira pro tetora" que impede que a cabeça de M caia inadvertidamente além da extrem dade esquerda da fita, o que não interfere de qualquer maneira na computaç realizada por M sob qualquer outro aspecto. Além disso, note-se que δ não definida nos casos em que o estado corrente pertence ao conjunto H. Dess maneira, sempre que a máquina alcançar um estado de parada, sua operação pára imediatamente.

Exemplo 4.1.1: Consideremos a máquina de Turing $M = (K, \Sigma, \delta, s, \{h\})$, onde

$$K = \{q_0, q_1, h\},$$
$$\Sigma = \{a, \sqcup, \triangleright\},$$
$$s = q_0,$$

sendo δ dado pela seguinte tabela:

q,	σ	$\delta(q, \sigma)$
q_0	a	$(q_1, \sqcup)$
q_0	$\sqcup$	$(h, \sqcup)$
q_0	$\triangleright$	$(q_0, \rightarrow)$
q_1	a	(q_0, a)
q_1	$\sqcup$	$(q_0, \rightarrow)$
q_1	$\triangleright$	$(q_1, \rightarrow)$

Quando M é iniciado em seu estado inicial q_0, move seu cabeçote de leitura para a direita, substituindo todos os a's por $\sqcup$'s à medida que vai avançando, até encontrar uma célula da fita que já contenha um simbolo $\sqcup$; então pára. (A ação de substituir símbolos que não sejam espaços em branco por espaços em branco é conhecida como a ação de **apagar** o símbolo original.) Mais especificamente, vamos supor que M seja iniciado com seu cabeçote varrendo os quatro primeiros a's, o último dos quais seguido por um $\sqcup$. Então o estado de M se alternará entre os estados q_0 e q_1, quatro vezes, substituindo cada a por um $\sqcup$ e movendo o cabeçote para a direita; a primeira e quinta linhas da tabela que define δ são as mais relevantes para essa seqüência de movimentos. Neste ponto, M encontrar-se-á no estado q_0 tendo como símbolo de entrada corrente $\sqcup$ e, de acordo com a segunda linha da tabela, irá portanto parar. Note que a quarta linha da tabela, isto é, o valor de $\delta(q_1, a)$, é irrelevante, uma vez que M nunca pode estar no estado q_1, ao encontrar um símbolo a, se a máquina tiver iniciada no estado q_0. Contudo, *algum* valor deve ser atribuído nessa tabela para $\delta(q_1, a)$, já que se exige que δ seja uma função, com domínio $(K-H) \times \Sigma$. ✧

Exemplo 4.1.2: Consideremos a máquina de Turing $M = (K, \Sigma, \delta, s, H)$, onde

$$K = \{q_0, h\},$$
$$\Sigma = \{a, \sqcup, \triangleright\},$$
$$s = q_0$$
$$H = \{h\},$$

e δ, uma função de transição especificada na tabela seguinte:

q,	σ	$\delta(q, \sigma)$
q_0	a	$(q_0, \leftarrow)$
q_0	$\sqcup$	$(h, \sqcup)$
q_0	$\triangleright$	$(q_0, \rightarrow)$

Essa máquina percorre para a esquerda a fita de entrada até encontrar um $\sqcup$, e então pára. Se cada célula da fita, a partir da posição do cabeçote

até a extremidade esquerda da fita, contiver um a, e, como, a extremidade esquerda da fita contém obrigatoriamente um $\triangleright$, então M atingirá o extremo esquerdo da fita e daí em diante irá alternar indefinidamente para trás e para a frente, entre a extremidade esquerda da fita e a célula imediatamente à sua direita. Diferentemente de outros dispositivos determinísticos que temos estudado eventualmente, uma máquina de Turing pode nunca parar.

Formalizaremos a seguir a operação de uma máquina de Turing.

Para caracterizar o andamento da computação de uma máquina de Turing, é necessário especificar o seu estado, o conteúdo da sua fita de entrada e a posição do cabeçote sobre a fita. Uma vez que, excetuando-se uma parcela finita do início da fita, toda ela estará repleta de espaços em branco, o conteúdo da fita pode ser descrito utilizando uma cadeia finita de símbolos. Resolvemos particionar a cadeia de entrada em duas partes: a parte à esquerda da célula que estiver sendo correntemente lida, incluindo o símbolo nela contido; e a parte (possivelmente vazia) dos símbolos à direita dessa célula corrente. Além disso, considerando que dois pares de cadeias irão corresponder à mesma combinação da posição e do cabeçote com o conteúdo da fita, é preciso convencionar que a segunda parte da cadeia nunca termine com um espaço em branco (todas as células da fita que estejam à direita do último símbolo explicitamente representado são pressupostamente preenchidas com espaços em branco). Essas considerações motivam as seguintes definições:

Definição 4.1.2: Uma **configuração** da máquina de Turing $M = (K, \Sigma, \delta, s, H)$ é algum membro de $K \times \triangleright\Sigma^* \times (\Sigma^*(\Sigma - \{\sqcup\}) \cup \{\varepsilon\})$.

Todas as configurações se iniciam, portanto, com o símbolo de extremidade esquerda e nunca terminam com um de espaço em branco – a não ser enquanto tais espaços em branco forem visitados durante a operação da máquina. Portanto, $(q, \triangleright a, aba)$, $(h, \triangleright \sqcup \sqcup \sqcup, \sqcup a)$ e $(q, \triangleright \sqcup a \sqcup \sqcup, \varepsilon)$ são configurações válidas (veja a Figura 4-2), mas $(q, \triangleright baa, abc\sqcup)$ e $(q, \sqcup aa, ba)$ não o são. Uma configuração cujo estado componente seja o estado H de parada será chamada **configuração de parada**.

Iremos utilizar uma notação simplificada para representar ao mesmo tempo o conteúdo da fita e a posição do cabeçote: devemos denotar como $w\underline{a}u$ o conteúdo da fita na configuração (q, wa, u); o símbolo sublinhado indica a posição do cabeçote. Para as três configurações ilustradas na Figura 4-2, o conteúdo da fita seria representada como $\triangleright\underline{a}aba$, $\triangleright\sqcup\sqcup\underline{\sqcup}\sqcup a$ e $\triangleright\sqcup a\underline{\sqcup}\sqcup$. Usando essa notação podemos simplificar a denotação de configurações. Assim, podemos reescrever (q, wa, u) como $(q, w\underline{a}u)$. Utilizando essa convenção, as três configurações mostradas na Figura 4-2 seriam reescritas como $(q, \triangleright\underline{a}aba)$, $(h, \triangleright\sqcup\sqcup\underline{\sqcup}\sqcup a)$ e $(q, \triangleright\sqcup a\underline{\sqcup}\sqcup)$, respectivamente.

Definição 4.1.3: Seja $M = (K, \Sigma, \delta, s, H)$ uma máquina de Turing. Sejam duas configurações de M, $(q_1, w_1 \underline{a_1} u_1)$ e $(q_2, w_2 \underline{a_2} u_2)$, onde $a_1, a_2 \in \Sigma$. Então,

$$(q_1, w_1 \underline{a_1} u_1) \vdash_M (q_2, w_2 \underline{a_2} u_2)$$

se, e somente se, para algum $b \in \Sigma \cup \{\leftarrow, \rightarrow\}$, $\delta(q_1, a_1) = (q_2, b)$ e também

1. $b \in \Sigma$, $w_1 = w_2$, $u_1 = u_2$ e a_2, $= b$, ou então
2. $b = \leftarrow$, $w_1 = w_2 a_2$ e
   (a) $u_2 = a_1 u_1$, se $a_1 \neq \sqcup$ ou $u_1 \neq \varepsilon$, ou
   (b) $u_2 = \varepsilon$ se $a_1 = \sqcup$ e $u_1 = \varepsilon$, ou ainda
3. $b = \rightarrow$, $w_2 = w_1 a_1$ e
   (a) $u_1 = a_2 u_2$, ou
   (b) $u_1 = u_2 = \varepsilon$, e $a_2 = \sqcup$

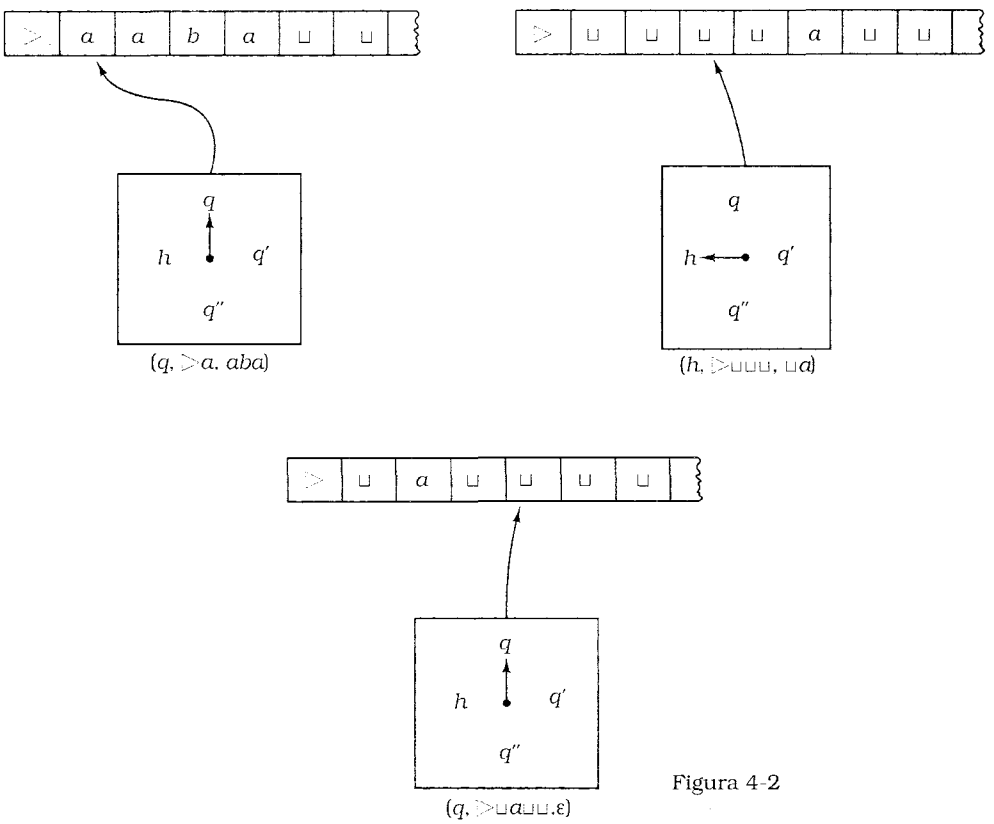


Figura 4-2

No Caso 1, M escreve um símbolo na fita sem mover o cabeçote. No Caso 2, M move o cabeçote uma célula para a esquerda; no caso em que o movimento da fita é para a esquerda, lendo espaços em branco no final da fita, os símbolos de espaço em branco da célula que acaba de ser lida desaparecem da configuração. No Caso 3, M move o cabeçote uma célula para a direita; se ele estiver se movendo sobre a parte vazia da fita, a cada célula consultada um novo símbolo de espaço em branco surge na configuração, representando explicitamente o novo símbolo lido. Note-se que todas as configurações, exceto as de parada, resultam exatamente em uma nova configuração.

Exemplo 4.1.3: Para ilustrar esses casos, sejam w, $u \in \Sigma^*$, onde u não termina em um espaço $\sqcup$, e sejam a, $b \in \Sigma$.

* N. de R.T. O original em inglês utiliza, inconsistemente, uma quádrupla, ao invés de uma tripla, para representar configurações na Figura 4-2.

Caso 1. $\delta(q_1, a) = (q_2, b)$.
Exemplo: $(q_1, w\underline{a}u) \vdash_M (q_2, w\underline{b}u)$.
Caso 2. $\delta(q_1, a) = (q_2, \leftarrow)$.
Exemplo para (a): $(q_1, wb\underline{a}u) \vdash_M (q_2, w\underline{b}au)$.
Exemplo para (b): $(q_1, wb\underline{\sqcup} \vdash_M (q_2, w\underline{b})$.
Caso 3. $\delta(q_1, a) = (q_2, \rightarrow)$.
Exemplo para (a): $(q_1, w\underline{a}bu) \vdash_M (q_2, wa\underline{b}u)$.
Exemplo para (b): $(q_1, w\underline{a}) \vdash_M (q_2, wa\underline{\sqcup})$.✧

Definição 4.1.4: Para uma máquina de Turing M arbitrária, seja $\vdash_M^*$ o fechamento transitivo reflexivo de $\vdash_M$; diz-se que a configuração C_1 **resulta** na configuração C_2 se $C_1 \vdash_M^* C_2$. Uma **computação** em M é uma seqüência de configurações $C_0, C_1, \ldots C_n$, para algum $n \geq 0$, tal que

$$C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \ldots \vdash_M C_n.$$

Diz-se, neste caso, que a computação é de comprimento n, ou que possui n passos, e denota-se $C_0 \vdash_M^n C_n$.

Exemplo 4.1.4: Considere-se a máquina de Turing M descrita no Exemplo 4.1.1. Se M for iniciada na configuração $(q_1, \triangleright\underline{\sqcup}aaaa)$, sua computação será representada formalmente como segue.

$$\begin{aligned}
(q_1, \triangleright\underline{\sqcup}aaaa) &\vdash_M (q_0, \triangleright \sqcup \underline{a}aaa) \\
&\vdash_M (q_1, \triangleright \sqcup \underline{\sqcup}aaa) \\
&\vdash_M (q_0, \triangleright \sqcup \sqcup\underline{a}aa) \\
&\vdash_M (q_1, \triangleright \sqcup \sqcup\underline{\sqcup}aa) \\
&\vdash_M (q_0, \triangleright \sqcup \sqcup \sqcup\underline{a}a) \\
&\vdash_M (q_1, \triangleright \sqcup \sqcup \sqcup \underline{\sqcup}a) \\
&\vdash_M (q_0, \triangleright \sqcup \sqcup \sqcup \sqcup\underline{a}) \\
&\vdash_M (q_1, \triangleright \sqcup \sqcup \sqcup \sqcup \underline{\sqcup}) \\
&\vdash_M (q_0, \triangleright \sqcup \sqcup \sqcup \sqcup \sqcup \underline{\sqcup}) \\
&\vdash_M (h, \triangleright \sqcup \sqcup \sqcup \sqcup \sqcup \underline{\sqcup})
\end{aligned}$$

Esta particular computação tem dez passos.✧

Uma notação para as máquinas de Turing

As máquinas de Turing que vimos até aqui são extremamente simples – pelo menos quando observados à luz dos objetivos declarados neste capítulo – e sua forma tabular já é bastante complexa e difícil de interpretar. Obviamente, torna-se conveniente que disponhamos de uma notação, para máquinas de Turing, que seja mais gráfica e transparente. Para autômatos finitos, utilizamos no Capítulo 2 uma notação que envolve círculos representando estados e setas denotando transições. Iremos, a seguir, adotar uma notação similar para as máquinas de Turing. Entretanto, nessa nova notação, os elementos interligados por setas serão, *eles próprios*, máquinas de Turing. Em outras palavras, utilizaremos uma notação *hierárquica* na qual máquinas de Turing cada vez mais complexas poderão ser construídas a partir de outras mais simples. Com isso em mente, vamos inicialmente definir um repertório muito simples de *máquinas de Turing básicas*, e um *conjunto de regras a serem empregadas para combinar máquinas já existentes*.

Máquinas básicas. Para dar partida ao processo, vamos definir as máquinas mais simples: as *máquinas usadas para gravar símbolos na fita* e as *máquinas empregadas para deslocar o cabeçote* sobre a fita. Definido um alfabeto fixo Σ para nossas máquinas, para cada $a \in \Sigma \cup \{\rightarrow, \leftarrow\} - \{\triangleright\}$, definimos uma máquina de Turing $M_a = (\{s, h\}, \Sigma, \delta, s, \{h\})$, onde, para cada $b \in \Sigma - \{\triangleright\}$, $\delta(s, b) = (h, a)$. Naturalmente, $\delta(s, \triangleright)$ deve ser sempre $(s, \rightarrow)$, isto é, a única ação que essa máquina executa é a de gravar o símbolo a, se $a \in \Sigma$, movendo o cabeçote na direção indicada por a, se $a \in \{\leftarrow, \rightarrow\}$ e, em seguida, parar imediatamente. Naturalmente, há uma única exceção para esse procedimento: Se o símbolo lido é um $\triangleright$, então a máquina irá simplesmente mover seu cabeçote de leitura/gravação para a direita sobre a fita.

Como as máquinas usadas para gravar símbolos são empregadas com tanta freqüência, abreviamos seus nomes, escrevendo simplesmente a em lugar de M_a. Isto é, se $a \in \Sigma$, então a *máquina de gravação de a* será denotada simplesmente como a. As máquinas de movimentação do cabeçote $M_{\leftarrow}$ e $M_{\rightarrow}$ serão abreviadas como L (movimentação para a esquerda) e R (movimentação para a direita) – [*left* e *right*, respectivamente, em inglês].

Regras para combinar máquinas. As máquinas de Turing serão combinadas de uma forma que, por sua aparência, sugere a estrutura conhecida dos autômatos finitos. Máquinas individuais figuram, nessa notação, de uma forma similar à dos estados de um autômato finito, e podem ser conectadas umas às outras, da mesma forma que os estados de um autômato finito costumam ser conectados entre si. Entretanto, as conexões entre uma máquina e outra somente são percorridas no momento em que a primeira máquina termina sua execução; a outra máquina é então iniciada a partir de seu estado inicial, com a fita e o cabeçote exatamente na situação em que foi deixada pela primeira máquina. Assim, se M_1, M_2 e M_3 forem máquinas de Turing, a máquina mostrada na Figura 4-3 opera da seguinte maneira: *inicie no estado inicial de M_1; opere exatamente como M_1 operaria isoladamente, até M_1 parar; então, se o símbolo correntemente lido pelo cabeçote for um a, inicie M_2 e opere como M_2 operaria; caso contrário, se o símbolo corrente for b, então inicie a execução de M_3, e opere como M_3 operaria.*

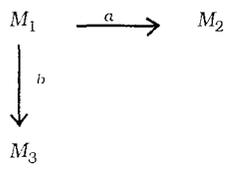


Figura 4-3

Como se pode notar, fica mais simples e direto definir, com essa notação, combinações de máquinas de Turing e de seus constituintes. Vamos estudar a máquina mostrada na Figura 4-3 acima. Sejam três máquinas de Turing M_1, M_2 e M_3 definidas como $M_1 = (K_1, \Sigma, \delta_1, s_1, H_1)$, $M_2 = (K_2, \Sigma, \delta_2, s_2, H_2)$ e $M_3 = (K_3, \Sigma, \delta_3, s_3, H_3)$. Para simplificar, vamos admitir, sem perda de generalidade que no contexto de máquinas combinadas *os conjuntos de estados dessas máquinas são disjuntos dois a dois*. A máquina combinada, mostrada na Figura 4-3 acima, representa, portando, $M = (K, \Sigma, \delta, s, H)$, onde

$K = K_1 \cup K_2 \cup K_3$,
$s = s_1$,
$H = H_2 \cup H_3$.
Para cada $\sigma \in \Sigma$ e $q \in K - H$, $\delta(q, \sigma)$ é definida como segue:
(a) Se $q \in K_1 - H_1$, então $\delta(q, \sigma) = \delta_1(q, \sigma)$.
(b) Se $q \in K_2 - H_2$, então $\delta(q, \sigma) = \delta_2(q, \sigma)$.
(c) Se $q \in K_3 - H_3$, então $\delta(q, \sigma) = \delta_3(q, \sigma)$.
(d) Por fim, se $q \in H_1$ – o único caso restante –, então $\delta(q, \sigma) = s_2$, se $\sigma = a$, $\delta(q, \sigma) = s_3$, se $\sigma = b$, caso contrário $\delta(q, \sigma) \in H$.

Todos os elementos de nossa notação estão agora devidamente posicionados. Podemos agora construir máquinas compondo as máquinas básicas e, então, temos a possibilidade de combinar adicionalmente as máquinas combinadas já construídas para obter outras máquinas mais complexas, e assim por diante. Sabemos que, se quisermos, podemos utilizar a forma de quíntupla para representar cada máquina que descrevermos, partindo das quíntuplas que definem as máquinas básicas, e realizando em cada passo a construção explícita acima exemplificada.

Exemplo 4.1.5: A Figura 4-4 (a) ilustra uma máquina que consiste de duas cópias de R. A máquina representada por esse diagrama desloca de uma posição o cabeçote para a direita; então, se na fita for encontrado um símbolo a, b, $\triangleright$ ou $\sqcup$, ela novamente deslocará o cabeçote para a direita, de mais uma posição.

(a) (b)

Figura 4-4

Mais uma vez, é possível simplificar a notação representando essa máquina conforme mostrado na Figura 4-4(b): uma seta rotulada com vários símbolos representa várias setas paralelas, uma para cada símbolo. Quando uma seta estiver rotulada por *todos os* símbolos do alfabeto Σ, então os rótulos podem ser omitidos. Portanto, se sabemos que $\Sigma = \{a, b, \triangleright, \sqcup\}$, então podemos representar a máquina acima como

$$R \rightarrow R,$$

onde a máquina mais à esquerda é sempre, por convenção, a máquina inicial. Uma seta não rotulada, conectando duas máquinas, pode ser inteiramente omitida pela justaposição das representações das duas máquinas. Usando mais essa convenção simplificadora, a máquina acima torna-se simplesmente RR ou, ainda, R^2. ✧

Exemplo 4.1.6: Se $a \in \Sigma$ é qualquer símbolo, podemos eliminar múltiplas setas e rótulos denotando como $\bar{a}$ *qualquer símbolo exceto* a. Usando essa convenção a máquina mostrada na Figura 4-5(a) percorre sua fita para a direita, até encontrar um espaço em branco. Essa máquina é bastante útil e é denotada como $R_\sqcup$.

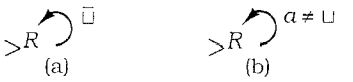


Figura 4-5

Outra abreviatura possível da mesma máquina é mostrada na Figura 4-5(b). Aqui $a \neq \sqcup$ é lido como "qualquer símbolo a que não seja $\sqcup$". A vantagem dessa notação é que a pode então ser utilizado em outra parte no diagrama como nome de uma máquina. Para ilustrar, a Figura 4-6 descreve uma máquina que percorre a fita para a direita até encontrar um símbolo não-branco, e então copia o símbolo na posição da fita imediatamente à esquerda daquela em que ele foi encontrado. ✧

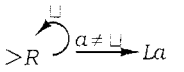


Figura 4-6

Exemplo 4.1.7: Máquinas que podem ser usadas para localizar posições marcadas ou desmarcadas da fita são ilustradas na Figura 4-7. São elas:

(a) $R_{\sqcup}$, o qual localiza na fita a primeira ocorrência de um espaço em branco à direita da posição corrente do cabeçote.

(b) $L_{\sqcup}$, o qual localiza na fita a primeira ocorrência de um espaço em branco à esquerda da posição corrente do cabeçote.

(c) $R_{\overline{\sqcup}}$, o qual localiza na fita a primeira ocorrência de um símbolo não-branco à direita da posição corrente do cabeçote.

(d) $L_{\overline{\sqcup}}$, o qual localiza na fita a primeira ocorrência de um símbolo não-branco à esquerda da posição corrente do cabeçote. ✧

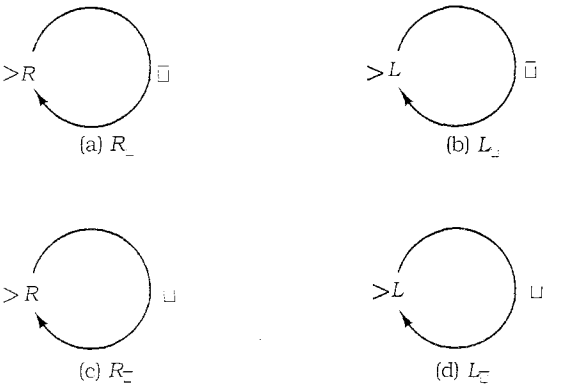


Figura 4-7

Exemplo 4.1.8: A *máquina de copiar C* realiza a seguinte função: iniciada com a entrada w, isto é, com uma cadeia w, eventualmente vazia, que contém somente símbolos não-brancos, colocados em uma fita inicialmente em branco, com uma célula em branco à esquerda, e com o cabeçote posicionado na primeira célula em branco à direita de w, então a máquina pára com $w \sqcup w$ em sua fita. Dizemos que C **transforma** $\sqcup w \sqcup$ **em** $\sqcup w \sqcup w \sqcup$.

Um diagrama para C é apresentado na Figura 4-8.✧

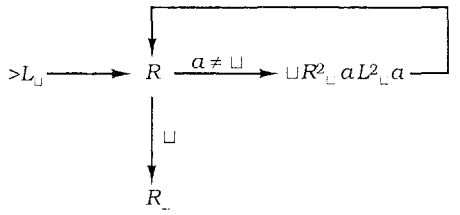


Figura 4-8

Exemplo 4.1.9: A máquina de deslocamento para a esquerda $S_{\leftarrow}$ transforma $\sqcup w \sqcup$, onde w não contém espaços em branco, em $w \sqcup$. Ela é apresentada na Figura 4-9. ✧

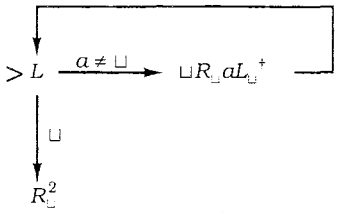


Figura 4-9

Exemplo 4.1.10: A Figura 4-10 é a máquina definida no Exemplo 4.1.1, que apaga todos os a's em sua fita substituindo-os por espaços em branco.

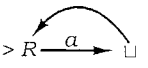


Figura 4-10

De fato, a tabela de transição completa dessa máquina irá diferir da tabela apresentada no Exemplo 4.1.1. Essas diferenças sutis são exploradas no Problema 4.1.8: a máquina da Figura 4-10 também conterá alguns estados a mais, correspondentes aos estados finais de suas máquinas constituintes. ✧

Problemas para a Seção 4.1

4.1.1 Seja $M = (K, \Sigma, \delta, s, \{h\})$, onde

$$K = \{q_0, q_1, h\},$$
$$\Sigma = \{a, b, \sqcup, \triangleright\}$$
$$s = q_0,$$

* N. de R.T. A máquina apresentada na Figura 4-9, no texto original em inglês, não corresponde à especificação do exemplo 4.1.9.

e δ é dado pela seguinte tabela:

q_i	σ	$\delta(q_i, \sigma)$
q_0	a	(q_1, b)
q_0	b	(q_1, a)
q_0	$\sqcup$	$(h, \sqcup)$
q_0	$\triangleright$	$(q_0, \rightarrow)$
q_1	a	$(q_0, \rightarrow)$
q_1	b	$(q_0, \rightarrow)$
q_1	$\sqcup$	$(q_0, \rightarrow)$
q_1	$\triangleright$	$(q_1, \rightarrow)$

(a) Descreva a computação de M partindo da configuração $(q_0, \triangleright \underline{a}abbba)$.
(b) Descreva informalmente o que M executa quando iniciado no estado q_0 com o cabeçote em qualquer posição da fita.

4.1.2 Repita o Problema 4.1.1 para a máquina $M = (K, \Sigma, \delta, s, \{h\})$, onde

$$K = \{q_0, q_1, q_2, h\},$$
$$\Sigma = \{a, b, \sqcup, \triangleright\}$$
$$s = q_0,$$

e δ é dado pela seguinte tabela (as transições em $\triangleright$ são $\delta(q, \triangleright) = (q, \rightarrow)$ e são omitidas).

q_i	σ	$\delta(q_i, \sigma)$
q_0	a	$(q_1, \leftarrow)$
q_0	b	$(q_0, \rightarrow)$
q_0	$\sqcup$	$(q_0, \rightarrow)$
q_1	a	$(q_1, \leftarrow)$
q_1	b	$(q_2, \rightarrow)$
q_1	$\sqcup$	$(q_1, \leftarrow)$
q_2	a	$(q_2, \rightarrow)$
q_2	b	$(q_2, \rightarrow)$
q_2	$\sqcup$	$(h, \sqcup)$

Inicie a operação da máquina a partir da configuração $(q_0, \triangleright \underline{a}bb \sqcup bb \sqcup \sqcup \sqcup aba)$.

4.1.3 Repita o Problema 4.1.1 para a máquina $M = (K, \Sigma, \delta, s, \{h\})$, onde

$$K = \{q_0, q_1, q_2, q_3, q_4, h\},$$
$$\Sigma = \{a, b, \sqcup, \triangleright\},$$
$$s = q_0,$$

e δ é dado pela seguinte tabela.

q,	σ	$\delta(q, \sigma)$
q_0	a	$(q_2, \rightarrow)$
q_0	b	(q_3, a)
q_0	$\sqcup$	$(h, \sqcup)$
q_0	$\triangleright$	$(q_0, \rightarrow)$
q_1	a	$(q_2, \rightarrow)$
q_1	b	$(q_2, \rightarrow)$
q_1	$\sqcup$	$(q_2, \rightarrow)$
q_1	$\triangleright$	$(q_1, \rightarrow)$
q_2	a	(q_1, b)
q_2	b	(q_3, a)
q_2	$\sqcup$	$(h, \sqcup)$
q_2	$\triangleright$	$(q_2, \rightarrow)$
q_3	a	$(q_4, \rightarrow)$
q_3	b	$(q_4, \rightarrow)$
q_3	$\sqcup$	$(q_4, \rightarrow)$
q_3	$\triangleright$	$(q_3, \rightarrow)$
q_4	a	$(q_2, \rightarrow)$
q_4	b	$(q_4, \rightarrow)$
q_4	$\sqcup$	$(h, \sqcup)$
q_4	$\triangleright$	$(q_4, \rightarrow)$

Inicie a operação da máquina a partir da configuração $(q_0, \triangleright \underline{a}aabbbaa)$.

4.1.4 Seja M a máquina de Turing $(K, \Sigma, \delta, s, \{h\})$, onde

$$K = \{q_0, q_1, q_2, h\},$$
$$\Sigma = \{a, \sqcup, \triangleright\},$$
$$s = q_0,$$

e δ é dado pela seguinte tabela.

Suponha que $n \geq 0$. Descreva cuidadosamente o que M executa quando iniciada na configuração $(q_0, \triangleright \sqcup a^n\underline{a})$.

q,	σ	$\delta(q, s)$
q_0	a	$(q_1, \leftarrow)$
q_0	$\sqcup$	$(q_0, \sqcup)$
q_0	$\triangleright$	$(q_0, \rightarrow)$
q_1	a	$(q_2, \sqcup)$
q_1	$\sqcup$	$(h, \sqcup)$
q_1	$\triangleright$	$(q_1, \rightarrow)$
q_2	a	(q_2, a)
q_2	$\sqcup$	$(q_0, \leftarrow)$
q_2	$\triangleright$	$(q_2, \rightarrow)$

4.1.5 Na definição de uma máquina de Turing, permitimos a gravação de uma célula da fita sem mover o cabeçote, bem como o movimento do cabeçote para a esquerda ou para a direita, sem nada gravar na fita. O que aconteceria se também permitíssemos deixar o cabeçote imóvel, sem nada gravar na fita?

4.1.6 (a) Qual ou quais das seguintes podem ser configurações de uma máquina de Turing?

(i) $(q, \triangleright a \sqcup a \sqcup \sqcup, \sqcup a)$
(ii) $(q, abcb, abc)$
(iii) $(p, \triangleright abca, \varepsilon)$
(iv) $(h, \triangleright \varepsilon, \varepsilon)$
(v) $(q, \triangleright a \sqcup abb, \sqcup aa \sqcup)$
(vi) $(p, \triangleright aab, \sqcup a)$
(vii) $(q, \triangleright \varepsilon, \sqcup aa)$
(viii) $(h, \triangleright aa, \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup a)$

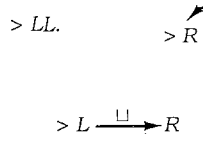
(b) Reescreva os itens (i) a (viii) acima que são configurações, utilizando a notação abreviada.

(c) Reescreva por extenso as seguintes configurações abreviadas:

(i) $(q, \triangleright a\underline{b}cd)$
(ii) $(q, \triangleright \underline{a})$
(iii) $(p, \triangleright aa \sqcup \underline{\sqcup})$
(iv) $(h, \triangleright \underline{\sqcup} abc)$

4.1.7 Projete e escreva por extenso uma máquina de Turing que percorre a fita para direita até encontrar dois a's consecutivos e, então, pára. O alfabeto dessa máquina de Turing deve ser $\{a, b, \sqcup, \triangleright\}$.

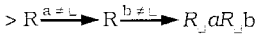
4.1.8 Descreva em todos os detalhes as máquinas de Turing seguintes.



4.1.9 As máquinas LR e RL sempre executam a mesma coisa? Explique.

ido

4.1.10 Explique o que executa a máquina seguinte:



4.1.11 Descreva a operação das máquinas de Turing do Exemplo 4.1.8, quando iniciadas com $\triangleright \underline{\sqcup} aabb$.

4.1.12 Descreva a operação da máquina de Turing do Exemplo 4.1.9 em $\triangleright \sqcup aabb \underline{\sqcup}$.

4.2 COMPUTAÇÕES USANDO MÁQUINAS DE TURING

As máquinas de Turing foram propostas com a promessa de que elas poderiam substituir, como reconhedores de linguagem, todos os tipos de autômatos apresentados nos capítulos anteriores. Até aqui, entretanto, foi nesta

somente a mecânica das máquinas de Turing, sem qualquer indicação de como elas devem ser utilizadas na realização de tarefas computacionais tais como reconhecer linguagens, por exemplo. Máquina de Turing são como computadores sem teclado, unidade de disco ou tela, isto é, sem um mecanismo para armazenar e recuperar informações nelas contidas. Torna-se oportuno, portanto, fixar algumas regras para a utilização das máquinas de Turing.

Primeiramente, vamos adotar a seguinte convenção para apresentar entradas às máquinas de Turing: a cadeia de entrada, isenta de espaços em branco, é gravada à direita do símbolo $\triangleright$, com um espaço em branco à sua esquerda e espaços em branco à sua direita; o cabeçote é colocado na posição da fita que contém o espaço em branco entre o $\triangleright$ e a cadeia de entrada; e a máquina é posicionada em seu estado inicial. Se $M = (K, \Sigma, \delta, s, H)$ é uma máquina de Turing, e $w \in (\Sigma - \{\sqcup, \triangleright\})^*$, então a **configuração inicial de M com entrada w** é $(s, \triangleright\sqcup w)$. Com essa convenção, podemos agora definir a utilização das máquinas de Turing como reconhecedores de linguagem.

Definição 4.2.1: Seja $M = (K, \Sigma, \delta, s, H)$ uma máquina de Turing, tal que $H = \{y, n\}$ consiste em dois estados distintos de parada (y e n simbolizam *sim* e *não* respectivamente). Qualquer configuração de parada cuja componente de estado é y é chamada **configuração de aceitação**, enquanto a configuração de parada cuja componente de estado é n é chamada **configuração de rejeição**. Dizemos que M aceita uma entrada $w \in (\Sigma - \{\sqcup, \triangleright\})^*$, se $(s, \triangleright\sqcup w)$ levar a uma configuração de aceitação; dizemos que M rejeita w se $(s, \triangleright\sqcup w)$ levar a uma configuração de rejeição.

Seja $\Sigma_0 \subseteq \Sigma - \{\sqcup, \triangleright\}$ um alfabeto, chamado **alfabeto de entrada** de M: fixando Σ_0 como um subconjunto de $\Sigma - \{\sqcup, \triangleright\}$, permitimos que nossas máquinas de Turing utilizem símbolos adicionais durante sua computação, além daqueles que aparecem em sua cadeia de entrada. Dizemos que M **decide** a linguagem $L \subseteq \Sigma_0^*$, se, para qualquer cadeia $w \in \Sigma_0^*$, se $w \in L$, então M aceita w; e se $w \notin L$, então M rejeita w.

Por fim, dizemos que a linguagem L é **recursiva** se houver uma máquina de Turing que a decide.

De acordo com essa definição, uma máquina de Turing decide uma linguagem L se, quando iniciada com a entrada w, ela sempre pára em um estado de parada que corresponde à resposta correta à entrada w: y, se $w \in L$, e n, se $w \notin L$. Note que nada se afirma quanto ao que acontece se a entrada w incluir espaços em branco ou o símbolo $\triangleright$ de final esquerdo.

Exemplo 4.2.1: Consideremos a linguagem $L = \{a^n b^n c^n : n \geq 0\}$, que, até aqui não pôde ser capturada por nenhum dos tipos de reconhecedores de linguagem. A máquina de Turing cujo diagrama é mostrado na Figura 4-11 decide L. Nesse diagrama, também utilizamos duas novas máquinas básicas, úteis para decidir linguagens: a máquina y leva ao estado de aceitação y, enquanto a máquina n transita para o estado n de rejeição.

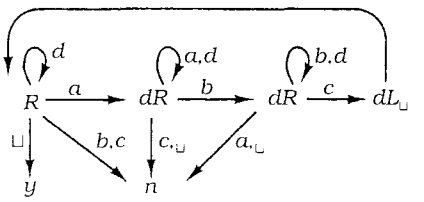


Figura 4-11

A estratégia empregada por M é simples: na entrada $a^n b^n c^n$, ela irá operar em n *estágios*. Em cada estágio, M inicia sua operação a partir do extremo esquerdo da cadeia e se move para a direita, à procura de um símbolo a. Encontrando um símbolo a, substitui-o por um símbolo d e, então, procura à direita um símbolo b adicional. Quando um b é encontrado, é substituído por um d, e então a máquina, nesse caso, procura um c. Quando um c é encontrado, é substituído por um d, e então esse estágio da análise é terminado e o cabeçote retorna para a extremidade esquerda da cadeia de entrada. O próximo estágio então tem início. Em cada estágio, a máquina substitui um a, um b e um c por d's. Se em qualquer ponto a máquina determinar que a cadeia não é da forma $a^*b^*c^*$, ou que há excesso de um certo símbolo (por exemplo, se for encontrado um b ou c, enquanto se procura um a), então a máquina entra no estado n e rejeita imediatamente a cadeia. Se, porém, ela atinge a extremidade direita da entrada enquanto procura um símbolo a, isso significa que toda a cadeia de entrada foi substituída por d's e, portanto, ela era de fato da forma $a^n b^n c^n$, para algum $n \geq 0$. A máquina, então, aceita a cadeia de entrada.✧

Há um ponto sutil em relação às máquinas de Turing que decidem linguagens. Com outros reconhecedores de linguagem até aqui estudados (mesmo os não-determinísticos), uma de duas condições podem ocorrer: ou a máquina aceita a cadeia de entrada ou então a rejeita. Uma máquina de Turing, por outro lado, mesmo que tenha somente dois estados de parada, y e n, sempre tem a opção de não responder "sim" nem "não", *simplesmente deixando de parar*. Dada uma máquina de Turing, ela pode ou não decidir uma linguagem – e não há um modo óbvio de dizer se ela faz isso. A abrangente importância – e *necessidade* – dessa deficiência deverá ser evidenciada mais adiante neste e no próximo capítulo.

Funções recursivas

Uma vez que as máquinas de Turing podem alterar o conteúdo de suas fitas, elas podem produzir saídas mais elaboradas do que um simples "sim" ou "não":

Definição 4.2.2: Seja $M = (K, \Sigma, \delta, s, \{h\})$ uma máquina de Turing; $\Sigma_0 \subseteq \Sigma - \{\sqcup, \triangleright\}$, um alfabeto; e $w \in \Sigma_0^*$. Suponha que M pára ao operar sobre a entrada w e que $(s, \triangleright\sqcup w) \vdash_M^* (h, \triangleright\sqcup y)$ para algum $y \in \Sigma_0^*$. Então y é dito **saída de M para a entrada** w e é denotado $M(w)$. Note-se que $M(w)$ é definida somente se M pára ao operar sobre a entrada w e isso, de fato, acontece para configurações da forma $(h, \triangleright\sqcup y)$, com $y \in \Sigma_0^*$.

Agora seja f qualquer função de Σ_0^* para Σ_0^*. Dizemos que M **computa** a função f, se, para todos os $w \in \Sigma_0^*$, $M(w) = f(w)$. Assim, para todo $w \in \Sigma_0^*$, M acaba por parar na entrada w e, nessa ocasião, sua fita contém a cadeia $\triangleright \sqcup f(w)$. Uma função f é dita **recursiva**, se houver uma máquina de Turing M que computa f.

Exemplo 4.2.2: A função κ: $\Sigma^* \rightarrow \Sigma^*$ definida como $\kappa(w) = ww$ pode ser computada pela máquina $CS_{\leftarrow}$, isto é, a máquina de cópia seguida pela máquina de deslocamento à esquerda (ambas foram definidas ao final da última seção). ✧

Cadeias sobre o alfabeto binário, $\{0,1\}^*$, podem ser utilizadas para representar inteiros não-negativos na *notação binária* usual. Qualquer cadeia $w = a_1 a_2 \ldots a_n \in \{0, 1\}^*$ representa o número

$$\text{num}(w) = a_1 \cdot 2^{n-1} + a_2 \cdot 2^{n-2} + \ldots + a_n.$$

e qualquer número natural pode ser representado, de um único modo, por uma cadeia da forma $0 \cup 1\ (0 \cup 1)^*$ –, isto é, isenta de zeros à esquerda.

De acordo com isso, as máquinas de Turing que computam funções de $\{0, 1\}^*$ para $\{0, 1\}^*$ podem ser pensadas como funções que computam números naturais a partir de números naturais. Na verdade, as funções numéricas com múltiplos argumentos – como a adição e a multiplicação – podem ser executadas por máquinas de Turing projetadas para computar funções de $\{0,1,;\}^*$ para $\{0, 1\}^*$, onde ";" é um símbolo utilizado para separar argumentos binários.

Definição 4.2.3: Seja $M = (K, \Sigma, \delta, s, \{h\})$ uma máquina de Turing, tal que $0,1,; \in \Sigma$ e seja f qualquer função de $\mathbf{N}^k$ para $\mathbf{N}$, para algum $k \geq 1$. Dizemos que M **computa** a função f, se para todo $w_1,\ldots, w_k \in 0 \cup 1\ \{0, 1\}^*$ (isto é, para quaisquer k cadeia que sejam codificações binárias de inteiros), $\text{num}(M(w_1;\ldots;w_k)) = f(\text{num}(w_1),\ldots,\text{num}(w_k))$. Assim, se M *é iniciada com as representações binárias dos inteiros* $n_1, \ldots, n_k$ fornecidas como entrada, então, em algum momento, a máquina irá parar e, quando isso acontecer, sua fita conterá a cadeia que representa o número $f(n_1, \ldots, n_k)$, ou seja o valor da função. Uma função f: $\mathbf{N}^k \rightarrow \mathbf{N}$ é dita **recursiva** se houver uma máquina de Turing M que computa f.

De fato, o termo *recursiva*, utilizado para descrever tanto funções como linguagens computadas por máquinas de Turing, originou-se do estudo de funções numéricas recursivas. Ele antecipa um resultado que devemos provar ao nos aproximarmos do final deste capítulo: que as funções numéricas computáveis através de máquinas de Turing coincidem com aquelas que podem ser definidas *recursivamente* a partir de um conjunto adequado de funções básicas.

Exemplo 4.2.3: Podemos projetar uma máquina que computa a função *sucessora* $\text{succ}(n) = n + 1$ (na Figura 4.12; S_R é a *máquina de deslocamento para a direita*, similar a máquina no Exemplo 4.1.9). Essa máquina localiza inicialmente a extremidade direita da cadeia de entrada, e então move-se para a

esquerda, enquanto encontrar 1's, substituindo todos eles por 0's. Quando localizar um 0, transforma-o em um 1 e então pára. Se encontrar um ⊔ quando procura um 0, isso significa que o número fornecido como cadeia de entrada tem uma representação binária que é inteiramente composta de 1's (ou seja, é o predecessor de uma potência de dois) e assim a máquina novamente grava um 1 no lugar do ⊔ e pára, após toda a cadeia ser deslocada de uma posição para a direita na fita. Estritamente falando, a máquina mostrada não computa $n + 1$, porque ela falha sempre em parar com o cabeçote à direita do resultado; mas isso pode ser corrigido, adicionando-se-lhe uma cópia de $R_{\sqcup}$ (Figura 4-5). ✧

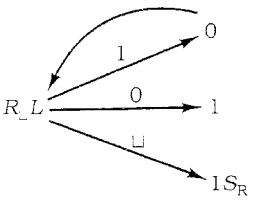


Figura 4-12

A última observação feita na subseção anterior, acerca de nossa incapacidade de determinar se uma máquina de Turing decide ou não uma linguagem, também se aplica à função de computação. O preço que devemos pagar pelo amplo espectro de funções que as máquinas de Turing são capazes de computar é que não podemos sequer afirmar se uma dada máquina de Turing de fato computa tal função – isto é, se ela pára qualquer que seja sua cadeia de entrada.

Linguagens recursivamente enumeráveis

Pelo fato de uma máquina de Turing decidir uma linguagem ou computar uma função, pode-se considerar razoável interpretá-la como um *algoritmo* que executa corretamente alguma tarefa computacional. A seguir, será apresentado um terceiro modo, mais sutil, pelo qual uma máquina de Turing pode definir uma linguagem:

Definição 4.2.4: Seja $M = (K, \Sigma, \delta, s, H)$ uma máquina de Turing. Seja $\Sigma_0 \subseteq \Sigma - \{\sqcup, \triangleright\}$ um alfabeto e $L \subseteq \Sigma_0^*$ uma linguagem. Dizemos que M **semidecide** L se, para qualquer cadeia $w \in \Sigma_0^*$, $w \in L$ se e somente se M pára em resposta à entrada w. Uma linguagem L é **recursivamente enumerável** se e somente se existir uma máquina de Turing M que semidecide L.

Portanto, quando M é acionada com a entrada $w \in L$, exige-se que pare ao final. Não importa a configuração precisa de parada que M atinge, bastando que M pare. Se, entretanto, $w \in \Sigma_0^* - L$, então M nunca deve atingir o estado de parada. Assim, como qualquer configuração não é de parada, M é levada sempre a alguma outra configuração que também não é de parada

(δ é uma função completamente definida), e a máquina irá, dessa maneira, continuar sua computação indefinidamente.

Estendendo a notação "funcional" de máquinas de Turing apresentada na subseção anterior (a qual permite escrever expressões tais como $M(w) = \uparrow$, escreve-se $M(w) = \nearrow$, se M falhar em parar em resposta à entrada w. Nessa notação, pode-se representar, novamente, a semidecisão de uma linguagem $L \subseteq \Sigma_0^*$ por máquina de Turing M como segue: para todo $w \in \Sigma_0^*$, $M(w) = \nearrow$ se e somente se $w \notin L$.

Exemplo 4.2.4: Seja $L = \{w \in \{a, b\}^*: w \text{ contém pelo menos um } a\}$. Então L é semidecidido pela máquina de Turing mostrada na Figura 4-13.

Figura 4-13

Essa máquina, iniciada na configuração $(q_0, \triangleright \sqcup w)$ para algum $w \in \{a, b\}^*$ simplesmente varre a cadeia da esquerda para a direita até que um a seja encontrado e então pára. Se nenhum a for encontrado, a máquina prossegue indefinidamente percorrendo os infinitos espaços em branco que se seguem à sua entrada, sem nunca parar. Assim, L será exatamente o conjunto de cadeias $w \in \{a, b\}^*$, tal que M pára em resposta à entrada w. Portanto, M semidecide L e, desta forma, L é recursivamente enumerável.✧

"Prosseguir, indefinidamente percorrendo os infinitos espaços em branco" é somente uma das maneiras pelas quais uma máquina de Turing pode deixar de parar. Por exemplo, qualquer máquina com $\delta(q, a) = (q, a)$ executará indefinidamente essa transição, sem efetuar qualquer progresso, se ela encontrar um a no estado q. Naturalmente, um procedimento mais complexo desse tipo pode ser projetado, com a máquina percorrendo cíclica e indefinidamente um número finito de diferentes configurações.

A definição de semidecisão por máquinas de Turing é uma extensão relativamente simples e direta da noção de aceitação definida para o autômato finito determinístico. Há, no entanto, uma diferença importante: um autômato finito *sempre pára* após ler toda sua entrada – a questão é se ele pára em um estado final ou não-final. Nesse sentido, ele é um dispositivo computacional útil, um *algoritmo* ao qual podemos confiantemente perguntar se uma entrada pertence à linguagem aceita. Para isso basta esperar até que toda a cadeia de entrada tenha sido lida e então observa-se o estado da máquina. Em contraste, uma máquina de Turing que semidecide uma linguagem L pode não ser adequada para dizer se uma cadeia w está ou não em L, porque, se $w \notin L$, então *nunca se saberá quando teremos esperado o suficiente para obter uma resposta.*† Máquinas de Turing que semidecidem linguagens não são algoritmos.

† Já encontramos a mesma dificuldade com os autômatos de pilha (ver a Seção 3.7). Um autômato de pilha pode, em princípio, rejeitar uma entrada manipulando para sempre sua pilha sem ler nenhuma entrada adicional – na Seção 3.7 esse procedimento foi removido a fim de obtermos autômatos de pilha computacionalmente úteis, para certas linguagens livres de contexto.

Sabemos, a partir do que foi estudado no Exemplo 4.2.1, que $\{a^n b^n c^n: n \geq 0\}$ é uma linguagem recursiva. Mas ela é recursivamente enumerável? A resposta é fácil: *qualquer linguagem recursiva também é recursivamente enumerável.* Tudo que é necessário para construir outras máquinas de Turing que semidecidam, em vez de decidir a linguagem, é converter o estado de rejeição n em um estado que não seja de parada, no qual se garanta que a máquina nunca irá parar. Especificamente, dada qualquer máquina de Turing $M = (K, \Sigma, \delta, s, \{y, n\})$ que decide L, podemos definir uma máquina M', que semidecide L, como segue: $M = (K, \Sigma, \delta', s, \{y\})$, onde δ' é apenas δ acrescido das seguintes transições referentes a n: $\delta'(n, a) = (n, a)$ para todos os $a \in \Sigma$. É claro que, se M decide L, então M' semidecide L, porque M' aceita a mesma entrada que M; além disso, se M rejeita uma entrada w, então M' não pára em resposta a w mas passa a executar um ciclo infinito no estado n. Em outras palavras, para toda entrada w, $M'(w) = \nearrow$ se e somente se $M(w) = n$.

Provamos a seguir os seguintes resultados importantes:

Teorema 4.2.1: *Se uma linguagem é recursiva, então ela é recursivamente enumerável.*

Naturalmente, a interessante (e difícil) questão que se coloca é a seguinte: será possível sempre transformar cada máquina de Turing que semidecide uma linguagem (com nossa definição de semidecisão, que a torna virtualmente inútil como um dispositivo computacional) em um *algoritmo* real para *decidir* a mesma linguagem? No próximo capítulo veremos que a resposta a esta pergunta é negativa, uma vez que: *há linguagens recursivamente enumeráveis que não são recursivas.*

Uma importante propriedade da classe das linguagens recursivas é que ela é fechada em relação à complementação:

Teorema 4.2.2: *Se L é uma linguagem recursiva, então seu complemento $\bar{L}$ também é recursivo.*

Prova: Se L é decidida pela máquina de Turing $M = (K, \Sigma, \delta, s, \{y, n\})$, então $\bar{L}$ é decidida pela máquina de Turing $M' = (K, \Sigma, \delta', s, \{y, n\})$, a qual é idêntica a M, *exceto que inverte os papéis dos dois estados especiais de parada y e n*, ou seja δ' é definida como segue:

$$\delta'(q, a) = \begin{cases} n & \text{se } \delta(q, a) = y, \\ y & \text{se } \delta(q, a) = n, \\ \delta(q, a) & \text{caso contrário.} \end{cases}$$

É claro que $M'(w) = y$ se e somente se $M(w) = n$ e, portanto, M' decide $\bar{L}$. ■

Seria a classe das linguagens recursivamente enumeráveis também fechada sob o complemento? Constataremos, no próximo capítulo, que *a resposta a esta questão também é negativa.*

Problemas para a Seção 4.2

4.2.1 *Forneça uma máquina de Turing (usando nossa notação abreviada* que compute a seguinte função de cadeias em $\{a, b\}^*$ para cadeias em $\{a, b\}^*$: $f(w) = ww^R$.

4.2.2 Apresente máquinas de Turing que decidam as seguintes linguagens sobre $\{a, b\}$:
(a) $\varnothing$
(b) $\{\varepsilon\}$
(c) $\{a\}$
(d) $\{a\}^*$

4.2.3 Forneça uma máquina de Turing que semidecida a linguagem a^*ba^*b.

4.2.4 (a) Forneça um exemplo de máquina de Turing com um estado de parada que não compute uma função de cadeia para cadeia.
(b) Forneça um exemplo de uma máquina de Turing com dois estados de parada, y e n, que não decida uma linguagem.
(c) Você pode dar um exemplo de uma máquina de Turing com um estado de parada que não *semidecide* uma linguagem?

4.3 EXTENSÕES DA MÁQUINA DE TURING

Os exemplos da seção anterior deixam claro que as máquinas de Turing podem executar cálculos bastante poderosos, embora de maneira lenta e desajeitada. A fim de melhor entender sua surpreendente potência, devemos considerar o efeito de estender o modelo da máquina de Turing em várias direções. Devemos ver que, em cada caso, os recursos adicionais nada acrescentam às classes de funções computáveis ou linguagens decidíveis: os "novos modelos melhorados" da máquina de Turing podem, em cada instância, ser *simulados* pelo modelo padrão. Tais resultados aumentam nossa confiança de que a máquina de Turing é, de fato, o dispositivo computacional definitivo, o fim de nossa progressão em direção a autômatos cada vez mais poderosos. Um efeito colateral benéfico desse resultado é que devemos nos sentir livres para utilizar o recurso adicional ao projetarmos máquinas de Turing para resolver problemas particulares, com base na certeza de que nossa dependência em relação a tal recurso pode, se *necessário, ser eliminada.*

Fitas múltiplas

Pode-se criar máquinas de Turing com várias fitas (veja a Figura 4-14). Cada fita é conectada ao controle finito por meio de um cabeçote de leitura/gravação correspondente. A máquina pode, em um passo de operação, ler os símbolos extraídos por todos esses cabeçotes e, então, dependendo dos símbolos lidos e do seu estado atual, regravar algumas das células lidas e mover alguns dos cabeçotes para a esquerda ou para a direita, além de mudar de estado. Para qualquer inteiro *fixo* $k \geq 1$, uma máquina de Turing de k *fitas* é uma máquina de Turing equipada, como foi descrito acima, com k fitas e correspondentes cabeçotes. Portanto, uma máquina de Turing "padrão", como a que foi estudada até aqui, neste capítulo, pode ser vista simplesmente como uma máquina de Turing de k fitas, com $k = 1$.

Definição 4.3.1: Seja $k \geq 1$ um inteiro. Uma **máquina de Turing de k fitas** é uma quíntupla $(K, \Sigma, \delta, s, H)$, onde K, Σ, s e H correspondem aos mesmos componentes da máquina de Turing normal e δ, a **função de transição**, é uma função de $(K - H) \times \Sigma^k$ para $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})^k$. Isto é, para cada estado q e cada k-tuplas de símbolos de fita $(a_1, \ldots, a_k)$, $\delta(q, (a_1, \ldots, a_k)) = (p, (b_1, \ldots, b_k))$, onde p continua significando o novo estado e b_j é, intuitivamente, a ação assumida por M na fita j. Novamente, se $a_j = \triangleright$ para algum $j \leq k$, então $b_j = \rightarrow$.

A computação ocorre em todas as k fitas de uma máquina de Turing de k fitas. Correspondentemente, uma *configuração* de tal máquina deve incluir informações sobre o conteúdo de todas as fitas:

Definição 4.3.2: Seja $M = (K, \Sigma, \delta, s, H)$ uma máquina de Turing de k fitas. Uma configuração de M é um membro do conjunto

$$K \times (\triangleright \Sigma^* \times (\Sigma^*(\Sigma - \{\sqcup\}) \cup \{\varepsilon\}))^k$$

Isto significa que a configuração identifica o estado, o conteúdo da fita e a posição dos cabeçotes em cada uma das k fitas.

Sendo $(q, (w_1 \underline{a_1} u_1, \ldots, w_k \underline{a_k} u_k))$ uma configuração de uma máquina de Turing de k fitas (na qual repetimos k vezes o uso da notação abreviada para configurações) e se $\delta(p, (a_1, \ldots, a_k)) = (b_1, \ldots, b_k)$, então em um movimento a máquina se moveria para a configuração $(p, (w'_1 \underline{a'_1} u'_1, \ldots, w'_k \underline{a'_k} u'_k))$, onde, para $i = 1, \ldots, k$, $w'_i \underline{a'_i} u'_i$, $w_i \underline{a_i} u_i$ é modificada pela ação b_i, precisamente como na Definição 4.1.3. Dizemos que a configuração $(q, (w_1 \underline{a_1} u_1, \ldots, w_k \underline{a_k} u_k))$ *produz como resultado, em um passo,* a configuração $(p, (w'_1 \underline{a'_1} u_1, \ldots, w'_k \underline{a'_k} u'_k))$.

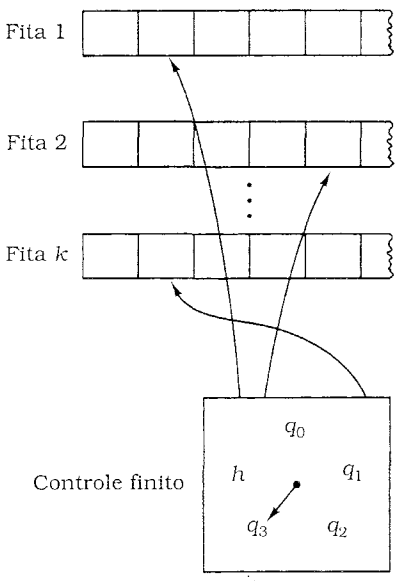


Figura 4-14

Exemplo 4.3.1: Máquinas de Turing de k fitas podem ser utilizadas para computar funções, decidir e semidecidir linguagens usando qualquer uma das maneiras discutidas anteriormente para máquinas de Turing convencionais. Adotamos a convenção de que a cadeia de entrada seja colocada na primeira fita, guardando as mesmas regras adotadas para uma máquina de Turing convencional. As demais fitas contêm inicialmente todas as células em branco, com o cabeçote posicionado no espaço em branco mais à esquerda de cada uma. No final de uma computação, uma máquina de Turing de k fitas deixa sua saída na primeira fita; o conteúdo das demais fitas é irrelevante.

A disponibilidade de múltiplas fitas muitas vezes facilita a construção de uma máquina de Turing para realizar uma particular função. Considere-se, por exemplo, a tarefa da máquina copiadora C do Exemplo 4.1.8: para transformar $\triangleright\sqcup w\underline{\sqcup}$ em $\triangleright\sqcup w\,\sqcup w\underline{\sqcup}$, onde $w \in \{a, b\}^*$, uma máquina de Turing de 2 fitas pode proceder da seguinte maneira:

(1) Movendo os cabeçote de ambas as fitas para a direita, copiando cada símbolo da primeira fita para a segunda fita, até que um espaço em branco seja encontrado na primeira fita. A primeira célula da segunda fita deve ser deixada em branco.
(2) Movendo o cabeçote da segunda fita para a esquerda até que um espaço em branco seja encontrado.
(3) Movendo novamente os cabeçotes de ambas as fitas para a direita, desta vez copiando os símbolos da segunda fita para a primeira. Pára quando um espaço em branco for encontrado na segunda fita.

Essa seqüência de ações pode ser representada como segue.

No início:	Primeira fita	$\triangleright\sqcup w$
	Segunda fita	$\triangleright\underline{\sqcup}$
Após (1):	Primeira fita	$\triangleright\sqcup w\underline{\sqcup}$
	Segunda fita	$\triangleright\sqcup w\underline{\sqcup}$
Após (2):	Primeira fita	$\triangleright\sqcup w\underline{\sqcup}$
	Segunda fita	$\triangleright\underline{\sqcup} w$
Após (3):	Primeira fita	$\triangleright\sqcup w\sqcup w\underline{\sqcup}$
	Segunda fita	$\triangleright\sqcup w\underline{\sqcup}$

Conforme comentamos, as máquinas de Turing com mais de uma fita podem ser descritas praticamente do mesmo modo que as máquinas de Turing de uma única fita. Simplesmente anexamos um índice superior, denotando para cada máquina o número da fita em que ela vai operar; todas as outras fitas não são afetadas. Por exemplo, $\sqcup^2$ grava um espaço em branco na segunda fita, $L^1_{\sqcup}$ procura à esquerda um espaço em branco na primeira fita e $R^{1,2}$ move para a direita os cabeçotes de *ambas* as fitas. Um rótulo a^1 em uma seta denota uma ação tomada se o símbolo lido na primeira fita for um a. E assim por diante. (Quando representamos máquinas de Turing de múltiplas fitas, evitamos empregar a abreviação M^2 para MM.) Utilizando essa convenção, a versão de 2 fitas da máquina copiadora pode ser como está ilustrado na Figura 4-15. Nessa figura, estão indicadas as submáquinas que executam as Funções 1 a 3 acima.✧

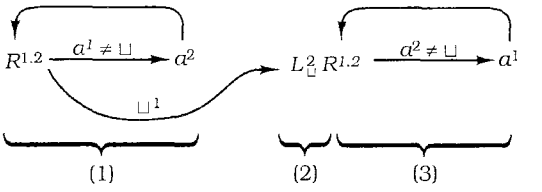


Figura 4-15

Exemplo 4.3.2: Vimos (Exemplo 4.2.3) que as máquinas de Turing podem adicionar uma unidade a qualquer inteiro binário. Não nos deve surpreender o fato de que as máquinas de Turing também possam somar *números binários arbitrários* (lembre-se do Problema 2.4.3, sugerindo que alguns autômatos finitos, em certo sentido, também o façam). Com duas fitas, essa tarefa pode ser realizada pela máquina descrita na Figura 4-16. Neste caso pares de bits, como 01, no rótulo de uma seta, são uma abreviação para, $a^1 = 0$, $a^2 = 1$.

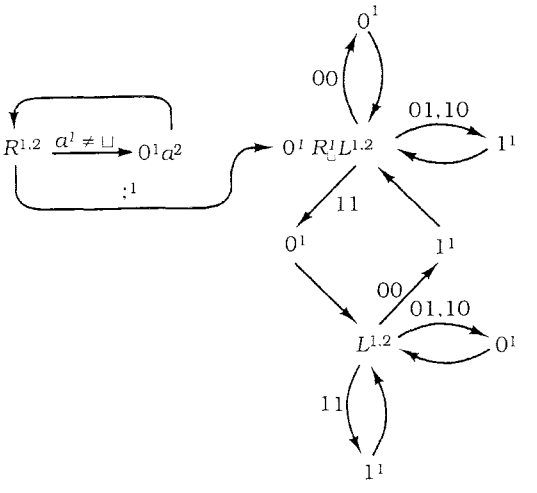


Figura 4-16

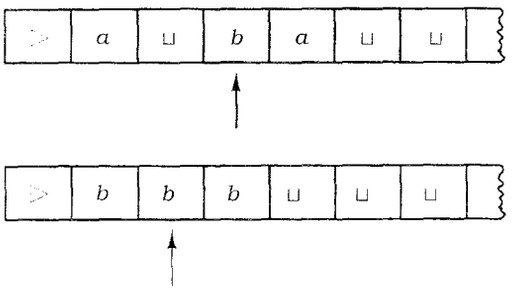
Essa máquina copia inicialmente o primeiro inteiro binário para sua segunda fita, gravando zeros em seu lugar (e no lugar do ";" que separa os dois inteiros) na primeira fita; desse modo, a primeira fita contém o segundo inteiro, com zeros adicionados na frente. A máquina então realiza uma adição binária pelo "método escolar", iniciando do bit menos significativo de ambos os inteiros, somando os bits correspondentes, gravando o resultado na primeira fita e considerando o «vai-um» em seu estado.✧

Adicionalmente, é possível construir uma máquina de Turing de 3 fitas capaz de *multiplicar* dois números; seu projeto é deixado como um exercício para o leitor (Problema 4.3.5).

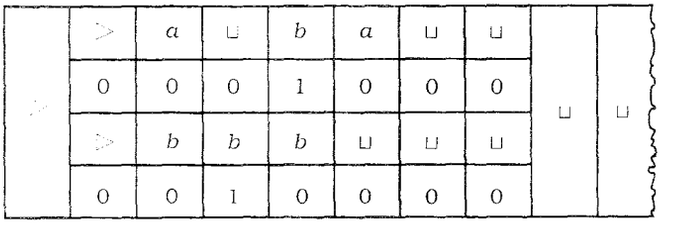
Evidentemente, máquinas de Turing de *k fitas* são capazes de computar tarefas bastante complexas. Mostramos a seguir que qualquer máquina de Turing de k fitas pode ser *simulada* por uma máquina de uma única fita.

Com isso queremos dizer que, a partir de uma máquina de Turing arbitrária de k fitas, podemos projetar uma máquina de Turing convencional que executa a mesma operação – decidir ou semidecidir a mesma linguagem, ou computar a mesma função. Essas *simulações* são importantes ferramentas que utilizamos para estudar, neste livro, o poder dos dispositivos computacionais. Em geral, elas procuram, de alguma forma, imitar o comportamento de cada passo da máquina simulada executando um ou mais passos na máquina simuladora. Nosso primeiro resultado dessa natureza, bem como sua prova, são ilustrativos dessa linha de raciocínio.

Teorema 4.3.1: *Seja $M = (K, \Sigma, \delta, s, H)$ uma máquina de Turing de k fitas, para algum $k \geq 1$. Então, há uma máquina de Turing convencional $M' = (K', \Sigma', \delta', s', H')$, onde $\Sigma \subseteq \Sigma'$, para o qual vale o seguinte: para qualquer cadeia de entrada $x \in \Sigma^*$, recebendo M a entrada x, pára com saída y na primeira fita se e somente se recebendo M' a entrada x, esta última terminar o processamento no mesmo estado de parada e com a mesma saída y em sua fita. Além disso, se M pára em resposta à entrada x, após t passos, então M' também pára, em resposta à entrada x, após um número de passos que é $O(t \cdot (|x| + t))$.*



(a)



(b)

Figura 4-17

Prova: A fita de M' deve de alguma forma dispor de todas as informações contidas em todas as fitas de M. Um modo simples de conseguir isso é imaginar que a fita de M' esteja dividida em várias *trilhas* (veja a Figura 4-18(b)), com suas "trilhas" dedicadas à simulação das diferentes fitas de M. Em particular, exceto quanto à célula mais à esquerda, a qual contém, como de

costume, o símbolo de final esquerdo ▷ e os infinitos espaços em branco no restante direito da fita, a única fita de M' é dividida, horizontalmente, em $2k$ trilhas. A primeira, terceira,..., $(2k - 1)$-ésimas trilhas da fita de M' correspondentes à primeira, segunda,..., k-ésimas fitas de M. A segunda, a quarta, ..., $2k$-ésima trilhas da fita de M' são utilizadas para gravar as posições dos cabeçotes sobre a primeira, segunda,..., k-ésimas fitas de M, do seguinte modo: se o cabeçote da i-ésima fita de M estiver posicionado sobre a n-ésima célula da fita, então as $2i$-ésimas trilhas da fita de M' contêm um 1 na $(n + 1)$-ésima célula da fita e um 0 em todas as células da fita, exceto a $(n + 1)$-ésima. Por exemplo, se $k = 2$, então as fitas e cabeçotes de M, mostrados na Figura 4-18(a), corresponderão à fita de M' mostrada na Figura 4-18(b).

Naturalmente, a divisão da fita de M' em trilhas é um recurso puramente conceitual; formalmente, esse efeito é obtido lendo

$$\Sigma' = \Sigma \cup (\Sigma \times \{0, 1\})^k.$$

Isto é, o alfabeto de M' consiste no alfabeto de M (isso permite que M' receba as mesmas entradas que M e recupere a mesma saída), mais todas as $2k$-tuplas da forma $(a_1, b_1, \ldots, a_k, b_k)$ com $a_1, \ldots, a_k \in \Sigma$ e $b_1, \ldots, b_k \in \{0, 1\}$. A interpretação desse alfabeto para a versão de $2k$ trilhas é simples: lemos quaisquer dessas $2k$-tuplas como se a primeira trilha de M' contivesse a_1, a segunda, b_1, e assim por diante para as $2k$-ésima trilhas contendo b_k. Isso, por sua vez, significa que o correspondente símbolo da i-ésima fita de M contém a_i e que esse símbolo é lido pelo i-ésimo cabeçote se e somente se $b_i = 1$ (lembre-se da Figura 4-17(b)).

Dada uma entrada $w \in \Sigma^*$, M' opera conforme descrito a seguir.

(1) Desloca a entrada uma célula da fita para a direita. Retorna para a célula imediatamente à direita do símbolo ▷, e grava o símbolo (▷, 0, ▷, 0,... ▷, 0) nessa célula – isso representará o final esquerdo das k fitas. Desloca o cabeçote uma célula para a direita, e grava o símbolo (␣, 1, ␣, 1,..., ␣, 1) – isso significa que as primeiras células de todas as k fitas contêm um ␣ e são todas lidas pelos cabeçotes correspondentes. Desloca novamente para a direita. Em cada célula, se um símbolo $a \neq$ ␣ for encontrado, grava-se em sua posição o símbolo $(a, 0, ␣, 0, \ldots, ␣, 0)$. Se um ␣ for encontrado, a primeira fase estará concluída. O conteúdo da fita de M' representa fielmente a configuração inicial de M.

(2) Simula a computação por M' até que M pare (caso isto ocorra). Para simular um passo da computação de M, M' terá de realizar as operações na seguinte seqüência (assumimos que ele inicia a simulação em cada passo com seu cabeçote posicionado no primeiro "espaço em branco verdadeiro", isto é, a primeira célula de sua fita, que não tenha sido ainda subdividida em trilhas):

   (a) Varre a parte esquerda da fita, reunindo informações sobre os símbolos lidos nas k fitas pelos cabeçotes de M. Após todos os símbolos lidos terem sido identificados (pelos 1's nas correspondentes trilhas pares), reposiciona-se no espaço em branco verdadeiro mais à esquerda. Nenhuma gravação ocorre na fita durante essa parte da operação de M', mas quando o cabeçote retorna para a extremidade direita da cadeia,

o estado do controle finito é alterado para refletir as k-tuplas de símbolos de Σ, nas k trilhas das posições marcadas do cabeçote.

(b) Percorre o lado esquerdo, e então, o lado direito da fita para atualizar as trilhas de acordo com o movimento (de M) que será simulado. Em cada par de trilhas, move-se o marcador de posição do cabeçote uma célula para a direita ou para a esquerda, ou então regrava-se a célula com um símbolo de Σ.

(3) No momento em que a máquina simulada M for parar, M' converte primeiramente sua fita de trilhas para um formato de seqüência de símbolo, ignorando-se todas as trilhas, exceto a primeira; posiciona o cabeçote onde M teria posicionado seu primeiro cabeçote, e, por fim, pára no mesmo estado em que M teria parado.

Muitos detalhes foram omitidos nessa descrição. A Fase 2, embora não contenha nada que seja conceitualmente complexo, é relativamente confusa para ser especificada explicitamente, e, de fato, há várias escolhas quanto ao modo como as operações descritas podem, realmente, ser realizadas. Convém descrever um pequeno detalhe. Ocasionalmente, para algum $n > |w|$, M pode ter de mover inicialmente um de seus cabeçotes para a n-ésima célula da fita correspondente. Para simular tal operação, M' teria de estender a parte da sua fita, que é dividida em $2k$ trilhas, e regravar o primeiro $\sqcup$ à direita usando $2k$-tuplas $(\sqcup, 0, \sqcup, 0, \ldots, \sqcup, 0) \in \Sigma'$.

É claro que M' pode simular o procedimento de M conforme indicado no enunciado do teorema. Ainda falta justificar que o número de passos necessários para M' simular t passos de M na entrada x é $O(t \cdot (|x| + t))$. A Fase 1 da simulação requer $O(|x|)$ passos de M'. Então, para cada passo de M, M' deve realizar a operação descrita na Fase 2, (a) e (b). Isso requer que M' percorra a parte de $2k$ trilhas da sua fita duas vezes; em outras palavras, M' precisa de um número de passos que é proporcional ao comprimento das $2k$ trilhas que compõem a fita de M. A questão é: que comprimento pode ter essa parte da fita de M'? Ela inicia tendo $|x| + 2$ e posteriormente aumenta o seu comprimento por não mais de uma unidade para cada passo simulado de M. Portanto, se t passos de M são simulados para a entrada x, o comprimento da parte de $2k$ trilhas da fita de M' será no máximo de $|x| + 2 + t$ e portanto, cada passo de M pode ser simulado por $O(|x| + t)$ passos de M', completando a demonstração. ■

Utilizando as convenções descritas para a entrada e saída de uma máquina de Turing de k fitas, o seguinte resultado decorre facilmente do teorema anterior.

Corolário: *Qualquer função computada ou linguagem decidida ou semidecidida por uma máquina de Turing de k fitas, também é respectivamente computada, decidida ou semidecidida, por uma* máquina de Turing convencional.

Fita infinita em ambas as direções

Suponha agora que nossa máquina disponha de uma fita que seja infinita em ambas as direções. Todas as células contêm inicialmente espaços em branco,

exceto aquelas que contêm a cadeia de entrada; o cabeçote fica inicialmente, por exemplo, imediatamente à esquerda da cadeia de entrada. Nossa convenção quanto ao símbolo $\triangleright$ é desnecessária e sem sentido para essas máquinas.

Não é difícil constatar que, conforme ocorre no caso das fitas múltiplas, fitas infinitas em ambas as direções não proporcionam poder adicional às máquinas de Turing. Uma fita infinita em ambas as direções pode ser facilmente simulada por uma máquina de duas fitas: uma fita sempre contém a parte da fita à direita da célula que contém o primeiro símbolo da cadeia de entrada, enquanto outra contém a parte da fita à esquerda dessa, porém ao reverso do original. A máquina de duas fitas, por sua vez, pode ser simulada por uma máquina de Turing convencional. De fato, a simulação gasta apenas um tempo *linear*, em lugar de um tempo quadrático, já que, a cada passo, somente uma das trilhas estará ativa. Obviamente, as máquinas com *várias* fitas infinitas em ambas as direções também podem ser simuladas do mesmo modo.

Múltiplos cabeçotes

O que ocorre se uma máquina de Turing tem uma só fita, mas vários cabeçotes? Todos os cabeçotes lêem os símbolos correspondentes e podem mover-se ou escrever independentemente. (Alguma convenção deve ser adotada sobre o comportamento da máquina quando dois cabeçotes se posicionam na mesma célula da fita, ao mesmo tempo que se tentar gravar simultaneamente diferentes símbolos. Uma possibilidade seria dar preferência ao cabeçote de número mais baixo. Além disso, vamos admitir que os cabeçotes não têm a capacidade de detectar a presença simultânea com outros cabeçotes na mesma célula da fita, exceto talvez indiretamente, devido a gravações malsucedidas.)

É natural que uma simulação similar àquela que utilizamos para máquinas de k fitas possa ser realizada também para máquinas de Turing com vários cabeçotes em uma fita. A idéia básica é, novamente, dividir a fita em trilhas, das quais todas, exceto uma, são utilizadas somente para gravar as posições do cabeçote. Para simular um passo computacional executado pelas máquinas de múltiplos cabeçotes, a fita deve ser lida duas vezes: uma para localizar os símbolos contidos nas posições dos cabeçotes e outra, para alterar esses símbolos ou mover apropriadamente os cabeçotes. O número de passos necessários neste caso é novamente *quadrático*, como no Teorema 4.3.1.

O uso de múltiplos cabeçotes, bem como o de múltiplas fitas, pode, às vezes, simplificar drasticamente a construção de uma máquina de Turing. Uma versão de 2 cabeçotes da máquina de copiar C, no Exemplo 4.1.8, poderia funcionar de um modo muito mais natural que a versão de um só cabeçote (ou mesmo as versões de duas fitas, do Exemplo 4.3.1); veja o Problema 4.3.3.

Fita bidimensional

Outro tipo de generalização da máquina de Turing pode permitir que sua "fita" seja uma grade bidimensional infinita, e até permitir um espaço de armazenamento de dimensão mais alta. Tal dispositivo pode ser muito mais útil que as máquinas de Turing convencionais para resolver problemas do tipo "quebra-cabeça" (veja o problema do *ladrilhamento* no próximo capítulo). Partiremos

dele como um exercício (Problema 4.3.6) para definir em detalhes a operação de tais máquinas. Mais uma vez, entretanto, não é obtido nenhum aumento fundamental no poder computacional. Curiosamente, o número de passos necessários para simular t passos das máquinas de Turing bidimensionais na entrada x pela máquina de Turing normal é novamente *polinomial* em t e $|x|$.

As extensões do modelo da máquina de Turing, feitas acima, podem ser *combinadas*: pode-se pensar em máquinas de Turing com várias fitas, todas ou algumas das quais são de algum modo infinitas nas duas direções e têm mais de um cabeçote ou, ainda, podem ser multidimensionais. Novamente, é bastante simples notar que as capacidades da máquina de Turing, em última instância, não variam de um caso para outro.

Resumimos a seguir essa discussão sobre as diversas variantes das máquinas de Turing:

Teorema 4.3.2: *Qualquer linguagem decidida ou semidecidida e qualquer função computada por máquinas de Turing com várias fitas, vários cabeçotes, fitas infinitas nas duas direções ou fitas multidimensionais, podem ser decididas, semidecididas ou computadas, respectivamente, por uma máquina de Turing convencional.*

Problemas para a Seção 4.3

4.3.1 Defina formalmente:
(a) M semidecide L, sendo M uma máquina de Turing de fitas infinitas em ambas as direções;
(b) M computa f, sendo M uma máquina de Turing de k fitas, e sendo f uma função de cadeia para cadeia.

4.3.2 Defina formalmente:
(a) uma máquina de Turing com k cabeçotes (com uma fita infinita unidirecional);
(b) uma configuração para uma máquina assim definida;
(c) o resultado, em um passo, da relação entre configurações de uma máquina assim definida. (Há mais de um conjunto de definições corretas).

4.3.3 Descreva (através de uma extensão da notação adotada para máquinas de Turing de k fitas) uma máquina de Turing de 2 cabeçotes que computa a função $f(w) = ww$.

4.3.4 A pilha de um autômato de pilha pode ser interpretada como se fosse uma fita que se pode gravar e apagar somente em sua extremidade direita; nesse sentido, uma máquina de Turing generaliza o autômato de pilha determinístico. Neste problema, consideramos uma generalização em outra direção, conhecido como *autômato determinístico de pilha com duas pilhas*.
(a) Defina informal, mas criteriosamente, a operação de tal máquina. Defina o que significa tal máquina decidir uma linguagem.
(b) Mostre que a classe de linguagens decididas por tais máquinas é, precisamente, a classe das linguagens recursivas.

4.3.5 Apresente uma máquina de Turing de três fitas que, recebendo em sua primeira fita dois inteiros binários separados por um ";", efetua o cálculo do seu produto. (*Sugestão*: Utilize como "sub-rotina" a máquina de adição do Exemplo 4.3.2.

4.3.6 Defina formalmente uma máquina de Turing com uma fita bidimensional, suas configurações e sua operação. Defina o que significa tal máquina decidir uma linguagem L. Mostre que t passos dessa máquina, iniciada com uma célula de entrada de comprimento n, podem ser simulados por uma máquina de Turing convencional em um tempo que é *polinomial* em relação a t e n.

4.4 MÁQUINAS DE TURING DE ACESSO ALEATÓRIO

A despeito da evidente potência e versatilidade exibidas pelas variantes das máquinas de Turing discutidas até este ponto, todas elas se apresentam com um aspecto bastante restritivo: sua memória é *seqüencial*, isto é, para acessar a informação armazenada em algum local dessa memória, a máquina deve acessar inicialmente, uma por uma, todas as posições da fita compreendidas entre a célula atual e aquela que se deseja acessar. Em contraste, computadores reais empregam *memórias de acesso aleatório*, em que cada elemento pode ser acessado em um passo único, desde que adequadamente endereçado. O que aconteceria se equipássemos as memórias das máquinas de Turing com essa capacidade de acesso aleatório, permitindo o acesso a qualquer célula desejada em um único passo? Para obter essa capacidade, devemos também equipar nossas máquinas com *registradores*, capazes de armazenar e manipular *endereços* das células de sua memória. Nesta subseção, definimos tal extensão da máquina de Turing, e constatamos também que ela é equivalente, em potência, à máquina de Turing clássica apresentando, porém, com uma redução polinomial em sua eficiência.

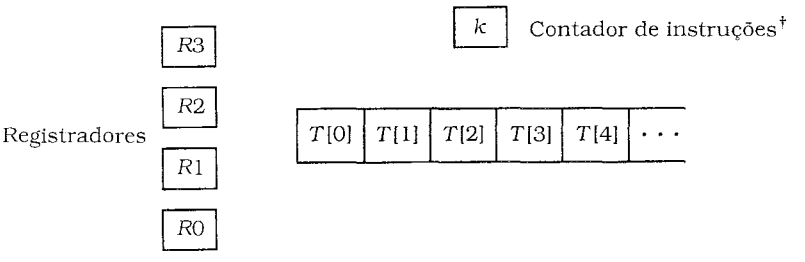


Figura 4-18

Uma máquina de Turing *de acesso aleatório* apresenta um número fixo de *registradores* e uma fita infinita unidirecional (veja a Figura 4-18; continuamos a chamar de "fita" a memória dessa máquina por razões de compatibilidade e para facilitar a comparação com o modelo padrão, apesar do fato de que, como veremos, ela funcionar de maneira muito mais parecida com um *chip* de memória semicondutora de acesso aleatório). Cada registrador e cada célula da fita comportam qualquer número natural arbitrário. A máquina age sobre as célu-

† N. de R.T. Program counter, em inglês.

las da sua fita e sobre seus registradores conforme determinado por um *programa* fixo – o análogo da função de transição das máquinas de Turing convencionais. Os programas de uma máquina de Turing com acesso aleatório são seqüências de *instruções*, que evocam as instruções usuais dos computadores. Os tipos permitidos de instruções estão enumerados na Figura 4-19.

Inicialmente, os registradores estão preenchidos com 0, o contador de programa aponta a instrução 1 e a fita contém a cadeia de entrada, codificada de uma maneira simples, brevemente especificada adiante. Nessas condições, a máquina executa a primeira instrução de seu programa. Isso irá alterar o conteúdo dos registradores ou o conteúdo da fita, conforme definido na Figura 4-19; além disso, o valor do *contador de instruções* K é um inteiro que especifica em cada momento a instrução que estiver sendo executada, e é modificado por algumas instruções conforme indicado na figura. Note o papel especial do Registrador 0: ele é um *acumulador*, no qual todas as operações aritméticas e lógicas são efetuadas. A *K*-ésima instrução do programa será a próxima a ser executada, e assim por diante, até que uma instrução halt seja executada – nesse ponto a operação de máquina de Turing de acesso aleatório termina.

Instrução	Operando	Semântica
read	j	$R_0 := T[R_j]$
write	j	$T[R_j] := R_0$
store	j	$R_j := R_0$
load	j	$R_0 := R_j$
load	$= c$	$R_0 := c$
add	j	$R_0 := R_0 + R_j$
add	$= c$	$R_0 := R_0 + c$
sub	j	$R_0 := \max\{R_0 - R_j, 0\}$
sub	$= c$	$R_0 := \max\{R_0 - c, 0\}$
half		$R_0 := \lfloor R_0/2 \rfloor$
jump	s	$K := s$
jpos	s	se $R_0 > 0$ então $K := s$
jzero	s	se $R_0 = 0$ então $K := s$
halt		$K := 0$

Notas: j significa o número de um registrador, $0 \leq j < k$. $T[i]$ denota o conteúdo atual da célula i da fita. R_j denota o conteúdo corrente do Registrador j. $s \leq p$ denota qualquer número de instrução no programa. c é qualquer número natural. Todas as instruções alteram K para $K + 1$, a não ser que explicitamente mencionado em contrário.

Figura 4-19

Podemos agora definir formalmente a máquina de Turing de acesso aleatório, suas configurações e sua operação.

Definição 4.4.1: Uma máquina de Turing de acesso aleatório é um par $M = (k, \Pi)$, onde $k > 0$ é o número de registradores e $\Pi = (\pi_1, \pi_2, \ldots, \pi_p)$, o **programa**, é a seqüência finita de **instruções**, em que cada instrução π_i é de um dos

tipos definidos na Figura 4-19. Admitimos que a última instrução, π_p, é sempre a instrução halt (o programa pode conter outras instruções halt também).

A **configuração** da máquina de Turing de acesso aleatório (k, Π) é a $k + 2$-tupla $(K, R_0, R_1, \ldots R_{k-1}, T)$, onde

> $K \in \mathbf{N}$ é o **contador de instruções**, um inteiro entre 0 e p.
> Uma configuração é dita a **configuração de parada** se K é zero.
> Para cada j, $0 \leq j < k$, $R_j \in \mathbf{N}$ é o **valor atual do Registrador** j.
> T, o conteúdo da fita, é um conjunto finito de pares de inteiros positivos – isto é, um subconjunto finito de $(\mathbf{N} - \{0\}) \times (\mathbf{N} - \{0\})$ – tal que, para todos os $i \geq 1$ há, no máximo, um par na forma $(i, m) \in T$.

Intuitivamente, $(i, m) \in T$ significa que a i-ésima célula da fita realmente contém o inteiro $m > 0$. Todas as células de fita que não aparecem como primeiros componentes de um par pertencente a T, serão consideradas preenchidas com 0.

Definição 4.4.1 (continuação): Seja $M = (k, \Pi)$ uma máquina de Turing de acesso aleatório. Dizemos que a configuração $C = (K, R_0, R_1, \ldots, R_{k-1}, T)$ de M **produz em um passo** a configuração $C' = (K', R'_0, R'_1, \ldots, R'_{k-1}, T')$ (denotada $C \vdash_M C'$) se, intuitivamente, os valores de K', os R'_js e o T' refletem corretamente a aplicação da "semântica" (como na Figura 4-19) da atual instrução π_K à configuração C. Devemos para isso seguir precisamente a semântica definida para os quatorze tipos de instruções, na Figura 4-19.

Se π_K tiver o formato read j, onde $j < k$, então a execução dessa instrução terá o seguinte efeito: o valor contido no Registrador 0 é substituído pelo valor armazenado na célula número R_j – a célula "endereçada" pelo Registrador j. Em outros termos, $R'_0 = T[R_j]$, onde $T[R_j]$ é o único valor m, tal que $(R_j, m) \in T$, se tal m existir, e 0, em caso contrário. Além disso, $K' = K + 1$. Todos os outros componentes da configuração C' permanecer idênticos aos de C.

Se π_K tiver a forma add = c, onde $c > 0$ é um inteiro fixo tal como, por exemplo, 5, então temos $R'_0 = R_0 + c$ e $K' = K + 1$, com todos os outros componentes permanecendo os mesmos.

Se π_K assumir a forma write j, onde $j < k$, então temos $K' = K + 1$. T' será T com o par da forma (R_j, m) excluído se existir, e, se $R_0 > 0$, o par (R_j, R_0) será adicionado; todos os outros componentes permanecerão invariáveis.

Se π_K for encontrado na forma jpos s, onde $1 < s < p$, então temos $K' = s$, se $R_0 > 0$, e $K' = K + 1$, caso contrário; todos os outros componentes permanecem os mesmos.

Para os outros tipos de instruções a operação é similar. A relação $\vdash_M^*$, o fechamento transitivo reflexivo de $\vdash_M$, é lida "produz".

Exemplo 4.4.1: O conjunto de instruções de nossa máquina de Turing de acesso aleatório (definido na Figura 4-19) não tem nenhuma instrução de *multiplicação* mply. Se desejamos incluir essa instrução como operação primitiva, nossa máquina de Turing de acesso aleatório, apesar de continuar sendo equivalente à máquina de Turing convencional, consumiria muito mais tempo em sua execução simulada (veja Problema 4.4.4).

No entando, a omissão da instrução de multiplicação não representa uma grande perda, porque essa instrução pode ser emulada pela execução do pro-

grama mostrado na Figura 4-20. Se† o Registrador 0 inicialmente contém um número natural x e o Registrador 1 inicialmente contém y, então essa máquina de Turing de acesso aleatório irá parar e o Registrador 0 conterá o produto xy. A multiplicação é realizada por adições sucessivas, onde a instrução **half** é utilizada para obter a representação binária de y (de fato, nosso conjunto de instruções contém essa instrução não-usual precisamente para essa finalidade).

```
 1. store 2
 2. load 1
 3. jzero 19
 4. half
 5. store 3
 6. load 1
 7. sub 3
 8. sub 3
 9. jzero 13
10. load 4
11. add 2
12. store 4
13. load 2
14. add 2
15. store 2
16. load 3
17. store 1
18. jump 2
19. load 4
20. halt
```

Figura 4-20

Eis uma seqüência típica de configurações (como essa máquina não interage com as células da fita, a parte T dessas configurações é vazia; há $k = 5$ registradores):

(1; 5, 3, 0, 0, 0; Ø) ⊢ (2; 5, 3, 5, 0, 0; Ø) ⊢ (3; 3, 3, 5, 0, 0; Ø) ⊢ (4; 3, 3, 5, 0, 0; Ø) ⊢ (5; 1, 3, 5, 0, 0; Ø) ⊢ (6; 1, 3, 5, 1, 0; Ø) ⊢ (7; 3, 3, 5, 1, 0; Ø) ⊢ (8; 2, 3, 5, 1, 0; Ø) ⊢ (9; 1, 3, 5, 1, 0; Ø) ⊢ (10; 1, 3, 5, 1, 0; Ø) ⊢ (11; 0, 3, 5, 1, 0; Ø) ⊢ (12; 5, 3, 5, 1, 0; Ø) ⊢ (13; 5, 3, 5, 1, 5; Ø) ⊢ (14; 5, 3, 5, 1, 5; Ø) ⊢ (15; 10, 3, 5, 1, 5; Ø) ⊢ (16; 10, 3, 10, 1, 5; Ø) ⊢ (17; 1, 3, 10, 1, 5; Ø) ⊢ (18; 1, 1, 10, 1, 5; Ø) ⊢ (2; 1, 1, 10, 1, 5; Ø) ⊢* (18; 0, 0, 20, 0, 15; Ø) ⊢ (2; 0, 0, 20, 0, 15; Ø) ⊢ (3; 0, 0, 20, 0, 15; Ø) ⊢ (19; 0, 0, 20, 0, 15; Ø) ⊢ (20; 15, 0, 20, 0, 15; Ø)]

† A computação de uma máquina de Turing de acesso aleatório é iniciada com zeros em todos os registradores. Uma vez que o presente programa se destina a ser utilizado como uma parte de outras máquinas de Turing de acesso aleatório, faz sentido investigar o que aconteceria se ela fosse iniciada com uma configuração arbitrária.

Suponha que x e y sejam inteiros não-negativos encontrados nos Registradores 0 e 1, respectivamente, ao início da execução desse programa. Afirmamos que a máquina pára ao final com o produto $x \cdot y$ armazenado no Registrador 0 – como se tivesse executado a instrução "mply 1". O programa prossegue em várias *iterações*. Uma iteração é uma execução completa da seqüência de instruções π_2 a π_{18}. Na k-ésima iteração, $k \geq 1$, observam-se as seguintes condições:

(a) O registrador 2 contém $x\,2^k$.
(b) O registrador 3 contém $\lfloor y/2^k \rfloor$,
(c) O registrador 1 contém $\lfloor y/2^{k-1} \rfloor$,
(d) O registrador 4 contém o "resultado parcial" $x \cdot (y \bmod 2^k)$.

A iteração procura manter essas "invariantes". Assim, a sequência das instruções π_2 a π_5 implementam a invariante (b), assumindo que (c) tenha sido armazenado na iteração anterior. As instruções π_6 a π_8 computam o k-ésimo bit menos significativo de y, e, se esse bit *não* for zero, as instruções π_9 a π_{12} adicionam x^{2k-1} ao Registrador 4, conforme determina a invariante (d). Então, o conteúdo do Registrador 2 é dobrado pela sequência de instruções π_{13} a π_{15}, implementando a invariante (a) e, por fim, o conteúdo do Registrador 3 é transferido para o do Registrador 1, implementando (c). A iteração é então repetida. Se, em algum ponto, for verificado que $\lfloor y/2^{k-1} \rfloor = 0$, então o processo termina, e o resultado final é copiado do Registrador 4 para o acumulador.

Podemos abreviar esse programa como "mply 1", isto é, podemos representá-lo como se fosse uma instrução com a semântica $R_0 := R_0 \cdot R_1$. Assim, podemos nos sentir livres para utilizar a instrução "mply *j*" ou "mply = *c*" em nossos programas, sabendo que é possível simulá-las usando o programa acima. Naturalmente, o número de instruções será diferente, refletindo o tamanho do programa que implementa essa instrução mply. Se uma máquina de Turing de acesso aleatório utiliza essa instrução, então assume-se implicitamente que, além dos seus registradores explicitamente mencionados, ela deve ter três registradores adicionais, que desempenham o papel dos Registradores 2, 3 e 4 no programa acima. ✧

Podemos eliminar a aparência desajeitada dos programas nas máquinas de Turing de acesso aleatório, como a do exemplo anterior, adotando algumas abreviações convenientes. Por exemplo, denotando o valor armazenado no Registrador 1 por R_1, no Registrador 2 por R_2 e assim por diante. Dessa forma, podemos escrever, por exemplo,

$$R_1 := R_2 + R_1 - 1$$

como uma abreviatura da seqüência

```
1.   load 1
2.   add 2
3.   sub =1
4.   store 1
```

Uma vez que tenhamos adotado essa simplificação, podemos utilizar nomes adequados que representam significativamente as quantidades ar-

mazenadas nos registradores 1 e 2 e assim expressar essa seqüência de instruções simplesmente como

$$x := y + x - 1.$$

Aqui x e y são apenas nomes que foram associados aos conteúdos dos Registradores 1 e 2. Podemos, adicionalmente, utilizar abreviações como

$$\text{enquanto } x > 0 \text{ faça } x := x - 3,$$

onde x denota o valor do Registrador 1, em lugar da seqüência

1. load 1
2. jzero 6
3. sub =3
4. store 1
5. jump 1

Exemplo 4.4.1 (continuação): Aqui está uma forma abreviada muito mais legível do programa mply da Figura 4-20, no qual estamos assumindo que x e y serão multiplicados, e que o resultado será denominado w:

$w := 0$
enquanto $y > 0$ faça
 início
 $z :=$ half (y)
 se $y - z - z \neq 0$ então $w := w + x$
 $x := x + x$
 $y := z$
 fim
pare

A correspondência entre a forma acima e o programa original na Figura 4-20 é esta: y representa R_1, x representa R_2, z representa R_3 e w representa R_4. Note também que omitimos, por questão de clareza, os números de instrução explícitos; se instruções **goto** forem necessárias, podemos identificar instruções com rótulos simbólicos tais como a e b sempre que for necessário.

É, naturalmente, um processo relativamente mecânico partir de um programa abreviado, como o mostrado acima, para chegar a um programa completo, equivalente ao original, da máquina de Turing de acesso aleatório.✧

Apesar de termos detalhado o funcionamento das máquinas de Turing de acesso aleatório, não dissemos de que maneira elas recebem suas entradas e retornam suas saídas. Para facilitar comparações com os modelos clássicos de máquina de Turing, devemos assumir que as convenções de entrada e saída das máquinas de Turing de acesso aleatório seguem as mesmas convenções para entrada e saída das máquinas de Turing convencionais: a entrada é apresentada como uma seqüência de símbolos na fita, e apesar de a fita de uma máquina de Turing de acesso aleatório poder conter números naturais arbitrários, assumimos que inicialmente tais números representam, codificadamente, símbolos de alguma cadeia de entrada.

Definição 4.4.2: Vamos escolher um alfabeto Σ – o alfabeto do qual nossa máquina de Turing de acesso aleatório obterá sua cadeia de entrada – com $\sqcup \in \Sigma$ e $\triangleright \notin \Sigma$ ($\triangleright$ neste caso é desnecessário, uma vez que a máquina de Turing de acesso aleatório não corre o risco de ultrapassar a extremidade esquerda de sua fita). Além disso, seja **E** uma bijeção fixa entre Σ e $\{0, 1, \ldots, |\Sigma|-1\}$; é assim que estarão representadas as cadeias de entrada e de saída de máquinas de Turing de acesso aleatório. Assumimos que $\mathbf{E}(\sqcup) = 0$. A **configuração inicial** da máquina de Turing de acesso aleatório $M = (K, \Pi)$ com cadeia de entrada $w = a_1 a_2 \ldots a_n \in (\Sigma - \{\sqcup\})^*$ é $(K, R_0, \ldots, R_k, T)$, onde $K = 1$, $R_j = 0$ para todos os j e $T = \{(1, \mathbf{E}(a_1)), (2, \mathbf{E}(a_2)), \ldots, (n, \mathbf{E}(a_n))\}$.

Dizemos que M **aceita** a cadeia $x \in \Sigma^*$ se a configuração inicial que apresenta a entrada x produzir uma configuração de parada em que $R_0 = 1$. Dizemos que ela **rejeita** x se a configuração inicial que apresenta a cadeia de entrada x produzir uma configuração de parada na qual $R_0 = 0$. Em outras palavras, uma vez que M pára, podemos ler sua decisão no registrador 0; se esse valor for 1, a máquina aceitará x, e se for 0, ela o rejeitará.

Seja $\Sigma_0 \subseteq \Sigma - \{\sqcup\}$ um alfabeto, e $L \subseteq \Sigma_0^*$ uma linguagem. Dizemos que M **decide** L, se sempre que $x \in L$, M aceitar x e sempre que $x \notin L$, M rejeite x. Dizemos que M **semidecide** L quando $x \in L$, se e somente se para a cadeia de entrada x, M produzir alguma configuração de parada.

Por fim, seja f: $\Sigma_0^* \to \Sigma_0^*$ uma função. Dizemos que M **computa** f se, para todo $x \in \Sigma_0^*$, se a configuração inicial da máquina M apresentar a cadeia de entrada x, M produzir uma configuração de parada com os seguintes conteúdos de fita: $\{(1, \mathbf{E}(a_1)), (2, \mathbf{E}(a_2)), \ldots, (n, \mathbf{E}(a_n))\}$, onde $f(x) = a_1 \ldots a_n$.

Exemplo 4.4.2: Apresenta-se abaixo um programa para uma máquina de Turing de acesso aleatório, na forma abreviada, que decide a linguagem $\{a^n b^n c^n : n \geq 0\}$.

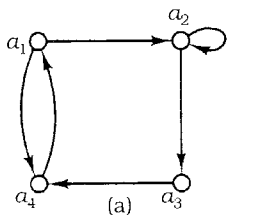
acount:= *bcount*:= *ccount*:= 0, *n*:= 1
enquanto $T[n] = 1$ faça: *n*:= *n* + 1, *acount*:= *acount* + 1
enquanto $T[n] = 2$ faça: *n*:= *n* + 1, *bcount*:= *bcount* + 1
enquanto $T[n] = 3$ faça: *n*:= *n* + 1, *acount*:= *acount* + 1
se *acount* = *bcount* = *ccount* e $T[n] = 0$, então aceitar, senão rejeitar

Assumimos aqui que $\mathbf{E}(a) = 1$, $\mathbf{E}(b) = 2$, $\mathbf{E}(c) = 3$ e utilizamos as variáveis *acount*, *bcount* e *ccount* para guardar o número de *a*'s, *b*'s e *c*'s, respectivamente, encontrados até então. Também usamos a abreviatura aceitar para "load =1, halt" e rejeitar para "load =0, halt". ✧

Exemplo 4.4.3: Para um exemplo mais completo, descrevemos em seguida uma máquina de Turing de acesso aleatório que determina o *fechamento reflexivo transitivo* de uma relação binária finita (ver Seção 1.6). É fornecido um grafo orientado $R \subseteq A \times A$, onde $A = \{a_0, \ldots, a_{n-1}\}$ e desejamos determinar R^*.

Surge então uma questão importante: como *representar* na forma de uma cadeia uma relação $R \subseteq A \times A$? R pode ser representada em termos de sua *matriz de adjacências* A_R, uma matriz $n \times n$ cujos elementos pertencem a $\{0, 1\}$ tais que o (i, j)-ésimo elemento vale 1 se e somente se $(a_i, a_j) \in R$ (ver exemplo na Figura 4-21). A matriz de adjacências, por sua vez, pode ser representada como uma cadeia de forma $\{0, 1\}^*$ de comprimento n^2, conten-

do primeiramente os elementos da primeira linha da matriz, em seguida os da segunda linha, e assim por diante. Denotamos a cadeia que representa a matriz de adjacências de uma relação R como x_R. Por exemplo, se R é a relação binária mostrada na Figura 4-21(a), então A_R e x_R têm a aparência mostrada na Figura 4-21(b) e (c), respectivamente.



(a)

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

(b)

0 1 0 1 0 1 1 0 0 0 0 1 1 0 0 0

(c)

Figura 4-21: O grafo, sua matriz de adjacências e a cadeia que representa essa matriz

Devemos, portanto, projetar uma máquina de Turing de acesso aleatório M que compute a função f definida como segue: para qualquer relação R sobre algum conjunto finito $\{a_1, \ldots, a_n\}$, $f(x_R) = x_{R^*}$. Note-se que não estamos interessados na maneira como M responde às cadeias de entrada que não estão em $\{0, 1\}^{n^2}$, ou seja, que não representam matrizes de adjacências de grafos orientados.

O programa de M é mostrado abaixo; assumimos que $\mathbf{E}(0) = 1$ e $\mathbf{E}(1) = 2$. Como sempre, $\mathbf{E}(\sqcup) = 0$.

As primeiras três instruções determinam o número n de elementos do conjunto A (um a menos que o menor número cujo quadrado aponta um espaço em branco na fita de entrada). Desse ponto em diante a (i, j)-ésima entrada da matriz, com $0 \leq i, j < n$, pode ser localizada como o $(i.n + j)$- ésimo símbolo na fita de M. Uma vez que esse programa é uma simples e direta implementação do algoritmo $O(n^3)$ da Seção 1.6, fica evidente que ele de fato determina o fechamento reflexivo transitivo da relação representada por sua cadeia de entrada. Naturalmente, uma versão não abreviada do programa da máquina de Turing pode ser mecanicamente obtida a partir do programa abaixo.✧

```
n := 1
enquanto T[n · n] ≠ 0 faça n := n + 1
n := n − 1
i := 0
enquanto i < n faça i := i + 1, T[i · n + i] := 2
i := j := k := 0
enquanto j < n faça j := j + 1,
    enquanto i < n faça i := i + 1,
        enquanto k < n faça k := K + 1,
            se T[i.n+j] = 2 e T[j · n + k] = 2 então, T[i · n + k]: = 2
pare
```

Evidentemente, a máquina de Turing de acesso aleatório é um modelo notavelmente poderoso e ágil. Como se pode comparar sua potência com

da máquina de Turing convencional? É muito fácil constatar, sem grande surpresa, que *a máquina de Turing de acesso aleatório* seja, pelo menos, *tão poderosa quanto a máquina de Turing convencional.* Seja $M = (K, \Sigma, \delta, s, H)$ uma máquina de Turing; podemos projetar uma máquina de Turing de acesso aleatório M' que simula M. M' tem um registrador. Vamos denominá-lo n. Este registrador informa permamentemente a posição do cabeçote nas trilhas de M. Inicialmente n aponta para o início da cadeia de entrada. Cada estado $q \in K$ é simulado por uma seqüência de instruções no programa de M'. Por exemplo, seja $\Sigma = \{\sqcup, a, b\}$, $\mathbf{E}(a) = 1$, $\mathbf{E}(b) = 2$ e q um estado de M tal que $\delta(q, \sqcup) = (p, \rightarrow)$, $\delta(q, a) = (p, \leftarrow)$, $\delta(q, b) = (r, \sqcup)$ e $\delta(q, \triangleright) = (s, \rightarrow)$. Uma seqüência de instruções que simula o estado q é a seguinte:

q: se $T[n] = 0$ então, $n := n + 1$, desviar para p
se $T[n] = 1$ então, se $n > 0$ então $n := n - 1$, desviar para p
senão desviar para s
se $T[n] = 2$ então, $T[n] := 0$, desviar para r

A parte **senão**, na terceira linha (a qual deve estar presente em qualquer linha que simule um movimento $\leftarrow$) faz o mesmo efeito do símbolo $\triangleright$, assegurando que o cabeçote nunca se desloque além da extremidade esquerda da fita de M'.

Dessa forma, mostramos:

Teorema 4.4.1: *Qualquer linguagem recursiva ou recursivamente enumerável, bem como qualquer função recursiva, podem ser decididas, semidecididas e computadas, respectivamente, por uma máquina de Turing de acesso aleatório.*

O notável é o oposto:

Teorema 4.4.2: *Qualquer linguagem decidida ou semidecidida por uma máquina de Turing de acesso aleatório e qualquer função computável por uma máquina de Turing de acesso aleatório, podem ser decididas, semidecididas e computadas, respectivamente, por uma máquina de Turing convencional. Adicionalmente, se tais máquinas param em resposta a uma entrada, então o número de passos executados pela máquina de Turing convencional é limitado polinomialmente pelo número de passos da máquina de Turing de acesso aleatório operando sobre a mesma entrada.*

Prova: Seja $M = (k, \Pi)$ uma máquina de Turing de acesso aleatório que decide ou semidecide uma linguagem $L \subseteq \Sigma^*$, ou que computa uma função de Σ^* para Σ^*. Esboçaremos o projeto de uma máquina de Turing M' convencional que simula M. Devemos descrever M' como uma máquina de $(k + 3)$ fitas, onde k é o número de registradores de M; sabemos, pelo Teorema 4.3.1, que tal máquina pode, por sua vez, ser simulada pelo modelo convencional.

A máquina de Turing M' monitora a configuração atual da máquina de Turing de acesso aleatório M e, repetidamente, determina a sua configuração seguinte. A primeira fita é utilizada somente para promover a leitura da cadeia de entrada de M e, possivelmente, para conter, o resultado da computação ao final da operação de M. A segunda fita é utilizada para comportar a parte T da configuração – o conteúdo da fita de M. A relação T é manti-

da como uma seqüência de cadeias da forma (111, 10), ou seja, a representação binária de um inteiro, seguido por uma vírgula, seguida por outro inteiro binário, tudo isso entre parênteses. Convenciona-se que a cadeia acima represente que a sétima célula da fita de M contém o inteiro 2. Os pares de inteiros da seqüência que representa T não estão em qualquer ordem particular, e podem ser separados arbitrariamente por longas seqüências de espaços em branco (para tanto, é necessário convencionar um *marcador de final*, tal como $, ao final da representação de T, para ajudar M' a decidir quando viu todos esses pares). Cada uma das demais k fitas de M' mantém o conteúdo de um dos registradores de M, também em binário. O valor corrente do contador de instruções K de M é permamentemente mantido no estado de M', conforme detalhado a seguir.

A simulação tem três fases. Durante a primeira fase, M' recebe em sua primeira fita a entrada $x = a_1a_2...a_n \in \Sigma^*$ e a converte na cadeia $(1, \mathbf{E}(a_1)) \ldots (n, \mathbf{E}(a_n))$, que é gravada na segunda fita. Portanto, M' pode iniciar a segunda fase, que corresponde à simulação de M, a partir da sua configuração inicial, com a cadeia de entrada x.

Durante a segunda fase, M' repetidamente simula um passo de M através da execução de diversos passos seus. A natureza exata do passo a ser simulado depende muito do contador de instruções K de M. Como foi mencionado anteriormente, K registra sempre o estado corrente de M'. O conjunto de estados de M' que são utilizados durante essa fase são separados em p conjuntos disjuntos $K_1 \cup K_2 \cup \ldots \cup K_p$, onde p é o número de instruções contidas no programa Π de M. O conjunto de estados K_j "especializa-se" em simular a instrução π_j de Π. A natureza precisa dessa parte de M' depende naturalmente do tipo de instrução π_j. Três exemplos ilustram esse mecanismo.

Primeiro suponhamos que π_j seja a instrução add 4, requerendo que o conteúdo do registrador 4 seja adicionado àquele contido no registrador 0. Então, M' realizará uma adição binária (ver o Exemplo 4.3.2) entre suas duas fitas, representando os registradores 4 e 0, deixará o resultado na fita associada ao registrador 0 e então mover-se-á para o primeiro estado de K_{j+1} para iniciar a simulação da próxima instrução. Se a instrução for, por exemplo, add = 33, então M' iniciará gravando o inteiro 33 em sua forma binária na $(k + 3)$-ésima fita (aquela até aqui não associada a qualquer parte de M); a representação binária da constante 33 é registrada pelos estados de K_j. Adicionará assim 33 ao conteúdo do registrador 0 e, por fim, apagará a última fita e se moverá para K_{j+1}.

Suponhanos, em seguida, que π_j seja a instrução write 2, que determina que o conteúdo do acumulador seja copiado para a célula da fita apontada pelo registrador 2. Então, M' acrescentará à extremidade direita da segunda fita – na qual o conteúdo da fita de M é mantido no formato (x, y) – o par (x, y), em que x é o conteúdo do registrador 2 e y o conteúdo do registrador 0; x e y são copiados das fitas correspondentes de M'. M' então percorrerá todos os outros pares (x', y) na segunda fita, comparando cada x', bit por bit, com o conteúdo do registrador i. Se uma correspondência for detectada, o par será apagado, mantendo assim a integridade da tabela. Então, o estado move-se para K_{j+1} e a próxima instrução é executada.

Suponhamos agora que π_j seja a instrução jpos 19, a qual exige que a instrução 19 seja executada caso o registrador 0 contenha um inteiro posi-

tivo. A máquina de Turing M' simplesmente percorrerá a fita que representa o registrador 0; se um 1 for encontrado na representação binária do inteiro nele contido, M move-se para K_{19}; caso contrário, move-se para K_{j+1}.

É simples e direto simular, de uma maneira muito semelhante, todos os demais tipos de instruções da tabela da Figura 4-19. Ao final, M poderá alcançar uma instrução halt. Quando isso acontecer, M' entrará em sua terceira fase, traduzindo a saída de M de tal forma que atenda as convenções de saída da máquina de Turing. Caso M esteja decidindo uma linguagem, então M' lerá o registrador 0. Se seu conteúdo não for 1, ele irá parar no estado y, se for 0 ele irá parar no estado n. Se M estiver semidecidindo uma linguagem, então M' simplesmente irá parar no estado h. Por fim, se M estiver computando uma função, então M' deverá traduzir o conteúdo da fita de M para a forma de uma cadeia de Σ^*, invertendo a bijeção **E**, e então parar.

Fica evidente da precedente discussão, que uma máquina de Turing de $k+3$ fitas M' pode ser projetada de forma que possa realizar as tarefas acima – e, portanto, pelo Teorema 4.3.1, uma máquina de Turing convencional pode, de fato, efetuar essas mesmas tarefas.

Para provar a segunda parte do teorema, devemos ainda provar que t passos de M, operando sobre uma cadeia de entrada de tamanho n, podem ser simulados em tempo $O(t + n)^3$. Naturalmente, as constantes na notação O dependerão, como sempre ocorre, da máquina simulada M; por exemplo, dependerão do valor da maior constante, tal como em add = 314159, referenciada no programa de M.

O limite $O(t + n)^3$ é baseado nas seguintes três observações:

(a) Cada passo de M, incluindo os passos de adição e subtração, (Problema 4.4.3), pode ser simulado em $O(m)$ passos de M', onde m é o comprimento total das partes de todas as fitas de M' contendo valores não-brancos – isto é, o comprimento total das codificações binárias de todos os inteiros na configuração corrente de M.

(b) O parâmetro m, definido acima, pode em cada passo aumentar de, no máximo, $O(r)$, onde r é o comprimento da mais longa das representações binárias de qualquer inteiro armazenado nos registradores ou nas células das fitas de M. Isso ocorre porque o aumento decorre da execução de uma instrução add ou de uma instrução store, e em ambos os casos é trivial ver que esse aumento só pode ser linear em r.

(c) Por fim, é fácil ver que $r = O(t)$; isto é, o comprimento do maior dos inteiros representados em M pode aumentar somente por uma constante a cada passo. O limite acima declarado decorre da junção desses três fatos. ■

Problemas para a Seção 4.4

4.4.1 Apresente explicitamente os detalhes completos do programa para a máquina de Turing de acesso aleatório do Exemplo 4.4.2. Apresente a seqüência de configurações dessa máquina em resposta à entrada *aabccc*.

4.4.2 Apresente (em notação abreviada) um programa para a máquina de Turing de acesso aleatório que decide a linguagem $\{wcw: w \in \{a, b\}^*\}$.

4.4.3 Mostre que, na simulação da prova do Teorema 4.4.2, cada passo de M pode ser simulado pelos $O(m)$ passos de M', onde m é o compri-

mento total das fitas *de M'*. (Você deve estabelecer que a adição [illegible] máquina de Turing de 2 fitas do Exemplo 4.3.2, opera em tem[illegible] linear.) Você pode estimar a constante em $O(m)$?

4.4.4 Suponha que nossa máquina de Turing de acesso aleatório tenh[illegible] uma instrução explícita mply. O que sai errado agora na segun[illegible] parte da prova do Teorema 4.4.2?

4.5 MÁQUINAS DE TURING NÃO-DETERMINÍSTICAS

Adicionamos a nossas máquinas de Turing muitos recursos aparentemen[illegible] poderosos – múltiplas fitas e cabeçotes, até mesmo acesso aleatório – sem aumento significativo na potência. Há, entretanto, um importante e familia[illegible] recurso que ainda não experimentamos: *o não-determinismo*.

Vimos que, ao permitirmos que autômatos finitos ajam não-deterministicamente, nenhuma variação ocorre no seu poder computacional (exceto que um número exponencialmente menor de estados podem ser necessários para executar a mesma tarefa). Não obstante, os autômatos de pilha não-determinísticos mostram-se mais poderosos que os determinísticos. Podemos também imaginar máquinas de Turing operando não-deterministiamente: essas máquinas podem ter, para certas combinações de símbolos de estado e símbolo de entrada lido, mais que uma escolha de procedimento possível. Formalmente uma máquina de Turing **não-determinística** é uma quíntupla $(K, \Sigma, \Delta, s, H)$, onde K, Σ, s e H são similares aos utilizados nas máquinas de Turing convencionais e Δ é um *subconjunto* de $((K - H) \times \Sigma) \times (K \times (\Sigma \cup \{\leftarrow, \rightarrow\}))$, em vez de uma função de $(K - H) \times \Sigma$ para $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$. As configurações e as relações $\vdash_M$ e $\vdash_M^*$ são definidas da maneira usual. Entretando, $\vdash_M$ não precisa mais ter um único valor: uma configuração pode produzir várias outras em um passo.

Perguntamos se, permitindo que máquinas de Turing operem não-deterministicamente, pode ou não ocorrer algum aumento no seu poder computacional? Devemos inicialmente definir o que significa dizer que uma máquina de Turing não-determinística computa alguma coisa. Uma vez que uma máquina pode, não-deterministicamente, eventualmente produzir dois ou mais diferentes estados finais a partir de uma mesma entrada, devemos ser cuidadosos sobre o que deve ser considerado como sendo o resultado final da computação efetuada por tal máquina. Por causa disso, é mais fácil, para começar, considerar máquinas de Turing não-determinísticas que *semidecidem* linguagens.

Definição 4.5.1: Seja $M = (K, \Sigma, \Delta, s, H)$ uma máquina de Turing não-determinística. Dizemos que M **aceita** uma entrada $w \in (\Sigma - \{\triangleright, \sqcup\})^*$ se $(s, \triangleright\sqcup w) \vdash_M^* (h, u\underline{a}w)$ para algum $h \in H$ e $a \in \Sigma$, $u, w \in \Sigma^*$. Note que a máquina não-determinística aceita uma dada entrada – ainda que haja muitas possíveis computações que sejam diferentes de um comando de parada, em resposta a essa entrada – até que ocorra pelo menos uma instrução de parada. Dizemos que M **semidecide a linguagem** $L \subseteq (\Sigma - \{\triangleright, \sqcup\})^*$ quando, para todos os $w \in (\Sigma - \{\triangleright, \sqcup\})^*$: $w \in L$ se e somente se M aceita w.

É um pouco mais sutil definir o que significa, para uma máquina de Turing não-determinística, decidir uma linguagem ou computar uma função.

Definição 4.5.2 Seja $M = (K, \Sigma, \Delta, s, \{y, n\})$ uma máquina de Turing não-determinística. Dizemos que M **decide** a linguagem $L \subseteq ((\Sigma - \{\triangleright, \sqcup\})^*$ se as duas condições seguintes se aplicam para todos os $w \in (\Sigma - \{\triangleright, \sqcup\})^*$:

(a) Há um número natural N, dependente de M e w, tal que não haja nenhuma configuração C que satisfaça $(s, \triangleright\sqcup w) \vdash_M^N C$.

(b) $w \in L$ se e somente se $(s, \triangleright\sqcup w) \vdash_M^* (y, u\underline{a}v)$ para algum $u, v \in \Sigma^*$, $a \in \Sigma$.

Por fim, dizemos que M computa a função f: $(\Sigma - \{\triangleright, \sqcup\})^* \rightarrow (\Sigma - \{\triangleright, \sqcup\})^*$ se as seguintes duas condições se aplicam a todos os $w \in (\Sigma - \{\triangleright, \sqcup\})^*$:

(a) Há um N, dependente de M e w, tal que não haja nenhuma configuração C satisfazendo $(s, \triangleright\sqcup w) \vdash C$.

(b) $(s, \triangleright\sqcup w) \vdash_M^* (h, u\underline{a}v)$, se, e somente se, $ua = \triangleright\sqcup$ e $v = f(w)$.

Essas definições refletem as dificuldades associadas à computação que se realiza com máquinas de Turing não-determinísticas. Note-se inicialmente que, para uma máquina não-determinística decidir uma linguagem e computar uma função, exigimos que *todas* as suas computações parem; chegamos a isso impondo que não haja nenhuma computação em andamento após N passos, onde N é um "limite superior", dependente da máquina e da entrada (essa é a condição (a) acima). Em segundo lugar, para M decidir uma linguagem, exigimos somente que *pelo menos* uma de suas possíveis computações finais aceite a entrada. Algumas, a maioria ou todas as demais computações até podem acabar eventualmente com uma rejeição – a máquina aceita a sua entrada pela força dessa simples computação de aceitação. Essa é uma convenção muito rara, assimétrica e não-intuitiva. Por exemplo, para criar uma máquina que decida o *complemento* da linguagem, não é suficiente inverter os papéis dos estados y e n (uma vez que a máquina pode ter ambas as computações, de aceitação e de rejeição, para algumas entradas). Como no caso dos autômatos finitos não-determinísticos, para mostrar que a classe de linguagens decididas por máquinas de Turing não-determinísticas é fechada sob a operação de complementação, devemos passar por uma máquina de Turing *determinística* equivalente – e nosso principal resultado nesta seção (Teorema 4.5.1) afirma que uma máquina de Turing determinística equivalente deve existir.

Por fim, para uma máquina de Turing não-determinística computar uma *função*, exigimos que *todas as* possíveis computações estejam de acordo entre si quanto ao resultado obtido. Se não formos capazes de decidir qual é o valor correto da função (nos casos de decidir ou semidecidir uma linguagem, resolvemos essa incerteza arbitrando que a resposta positiva prevalece).

Antes de mostrar que o não-determinismo, assim como ocorre com os demais recursos considerados nas seções anteriores, pode ser eliminado das máquinas de Turing, vamos considerar um exemplo clássico que demonstra o poder, que tem o não-determinismo das máquinas de Turing, como dispositivo conceitual.

Exemplo 4.5.1 Um **número é dito composto** quando pode ser representado pelo produto de dois números naturais, maiores que 1; por exemplo, 4, 6, 8, 9, 10 e 12 são compostos, mas 1, 2, 3, 5, 7 e 11 não são. Em outras palavras, um número composto é um não-primo diferente de 1 ou de zero.

Seja C = {100, 110, 1000, 1001, 1010,..., 1011011 ...} o conjunto de todas as representações binárias de números compostos. Projetar um algoritmo "eficiente" que decida C é um problema antigo, importante e difícil. Para projetar tal algoritmo, parece ser necessário inventar algum modo engenhoso de descobrir os *fatores*, de um número (se é que existem) – uma tarefa visivelmente bastante complexa. Naturalmente, ao procurarmos exaustivamente todos os números menores que um número dado (de fato, menores que sua *raiz quadrada*) acabaríamos descobrindo seus fatores; a questão é que *nenhum método mais direto que esse é evidente.*

Entretanto, se o não-determinismo for admitido, podemos projetar uma máquina para semidecidir C de maneira relativamente simples, adivinhando os fatores, se houver algum. Essa máquina opera conforme descrito a seguir, quando recebe como entrada a representação binária do número inteiro n:

(1) Escolhe não-deterministicamente dois números binários p e q maiores que um, e escreve bit a bit sua representação binária ao lado da entrada.
(2) Utiliza a máquina de multiplicação do Problema 4.3.5 (na realidade, a máquina de uma única fita que a simula) para substituir as representações binárias de p e q pela representação do seu produto.
(3) Verifica se os dois inteiros, n e $p \cdot q$, são os mesmos. Isso pode ser feito facilmente comparando-os bit a bit. Pára se os dois inteiros forem iguais; caso contrário, continua indefinidamente de algum modo (observe-se que, no momento, estamos interessados apenas em uma máquina que semidecide C).

Essa máquina, operando sobre a entrada 1000010 (que é a representação binária de 66) efetua muitas rejeições nas computações correspondentes à fase 1 acima, escolhendo pares de inteiros binários como 101; 11101, cujo produto não é 66. Entretanto, sendo 66 um número composto, haverá pelo menos *uma* computação de M que acabará aceitando – e isso é tudo de que precisamos. A rigor, haverá quatro computações válidas (correspondendo a $2 \cdot 33 = 6 \cdot 11 = 11 \cdot 6 = 33 \cdot 2 = 66$). Se a entrada fosse 1000011, entretanto, nenhuma das computações acabaria sendo aceita, já que 67 é um número primo.

Essa máquina pode ser convertida em uma máquina não-determinística que *decide* a linguagem C. Tal máquina tem a mesma estrutura básica, exceto que, na Fase (1), a nova máquina nunca escolhe arbitrariamente um inteiro com mais bits que os de n – obviamente, tal inteiro nunca poderia ser um fator de n. E, na Fase 3, após comparar a entrada com o produto, a nova máquina irá parar no estado y, se houver igualdade, e no estado n, caso contrário. Conseqüentemente, *todas as computações acabarão parando*, após algum número finito de passos.

O limite superior N, imposto pela Definição 4.5.2, é agora fácil de ser calculado explicitamente. Suponha-se que um dado inteiro n tem ℓ bits. Seja N_1 o número máximo de passos, executados em qualquer computação, pela máquina multiplicadora quando esta opera sobre qualquer entrada de comprimento $2\ell + 1$ ou menos; tal número é finito, e é o máximo dentre os elementos de um conjunto finito de naturais. Seja N_2 o número de passos necessários para comparar duas cadeias; cada qual de comprimento máximo 3ℓ. Nessas condições, qualquer computação executada por M' terminará

após $N_1 + N_2 + 3l + 6$ passos, que certamente é um número finito dependente exclusivamente da máquina e da sua entrada.✧

Pode parecer que o não-determinismo seja um recurso muito poderoso, e não facilmente eliminável. De fato, parece ser sempre difícil simular passo a passo uma máquina de Turing não-determinística de forma determinística, como foi feito nos outros casos de máquinas de Turing complexas que examinamos até este ponto. Entretanto, veremos que, de fato, as linguagens que são semidecididas, ou então decididas por máquinas de Turing não-determinísticas não são diferentes daquelas que são respectivamente semidecididas ou decididas, por máquinas de Turing determinísticas.

Teorema 4.5.1: *Se uma máquina de Turing não-determinística M semidecide ou decide uma linguagem, ou se ela computa uma função, então existe uma máquina de Turing convencional M' que semidecide, ou decide a mesma linguagem, ou computa a mesma função, respectivamente.*

Prova: Devemos descrever a construção de M' para o caso no qual M semidecide uma linguagem L; as construções para o caso de M decidir uma linguagem ou computar uma função são muito similares. Assim, seja $M = (K, \Sigma, \Delta, s, H)$ uma máquina de Turing não-determinística que semidecide uma linguagem L. Dada uma entrada w, M' irá tentar executá-la sistematicamente, para todas as possíveis computações de M procurando uma configuração de parada de M. Uma vez atingida tal configuração, M' também irá parar. Assim M' pára se e somente se M parar, como se esperaria de uma simulação.

Mas M pode ter um grande número de computações diferentes partindo da mesma configuração inicial; como M poderia explorar todas elas? Isso se faz utilizando um *procedimento do leque* (lembre-se do argumento ilustrado na Figura 1-8). A observação crucial é a seguinte: apesar de, para qualquer configuração C de M, poder haver várias configurações C' tais que $C \vdash C'$, o *número* de tais configurações C' é fixo, e limitado de um modo que depende somente de M, e não de C. Especificamente, o número de quádruplas $(q, a, p, b) \in \Delta$ que podem ser aplicados em qualquer ponto é finito; de fato, ele não pode exceder $|K| \cdot (|\Sigma|+2)$, uma vez que esse é o número máximo de possíveis combinações (p, b) com $p \in K$ e $b \in \Sigma \cup \{\leftarrow, \rightarrow\}$. Vamos chamar de r o número máximo de quádruplas que podem ser aplicadas em qualquer ponto; o número r pode ser determinado por inspeção de M. De fato, devemos considerar que, para cada combinação estado-símbolo (q, a) e para cada inteiro $i \in \{1, 2, \ldots, r\}$, há uma bem definida i-ésima quádrupla aplicável (q, a, p_i, b_i). Se para alguma combinação estado-símbolo (q, a) há menos que r quádruplas relevantes em Δ, então algumas podem ser repetidas.

Uma vez que M é não-determinística, ela não dispõe de uma forma definida para decidir em cada passo como decidir entre suas r alternativas disponíveis. Mas suponhamos que *a ajudemos a decidir*. Para sermos mais precisos, suponhamos que M_d, a *versão determinística* de M, seja um dispositivo com o mesmo conjunto de estados de M, mas com duas fitas. A primeira delas é a fita de M, contendo inicialmente a entrada w, enquanto a segunda contém inicialmente uma cadeia de n inteiros no intervalo $1, \ldots, r$, por

exemplo $i_1 i_2 \ldots i_n$. M_d então opera por n passos conforme descrito a seguir: nos primeiros passos, entre as próximas r combinações possíveis estão a combinação $(p_1, b_1), \ldots, (p_r, b_r)$ aplicadas à configuração inicial, M_d escolhe a i_1-ésima –isto é, (p_{i_1}, b_{i_1}), indicada pelo símbolo atualmente lido na segunda fita, i_1. M_d também move a cabeça da sua segunda fita para a direita de modo que, a seguir, ele leia i_2. No próximo passo, M_d selecionará a i_2-ésima combinação, depois a i_3-ésima e assim por diante. Quando M_d encontrar um espaço em branco na sua segunda fita, isso significa que ele esgotou todas as alternativas, então ela pára.

M_d é um importante ingrediente em nosso projeto da máquina de Turing determinística M' que simula M. Devemos descrever M' como uma máquina de Turing de 3 fitas; sabemos, pelo Teorema 4.3.1, que M' pode ser convertido em uma máquina de Turing equivalente, de uma fita. As três fitas de M' são utilizadas como segue:

(1) A primeira fita nunca é modificada; ela sempre contém a entrada original w, de modo que cada computação simulada de M pode ser iniciada usando sempre a mesma entrada.
(2) A segunda e a terceira fita são utilizadas para simular as computações de M_d, a versão determinística de M, testando todas as cadeias em $\{1, 2, \ldots, r\}^*$. A entrada é copiada da primeira fita para a segunda antes que M' comece a simular cada nova computação. Inicialmente, a terceira fita contém ε, a cadeia vazia (e, portanto, a simulação de M_d não irá nem mesmo iniciar sua primeira execução).
(3) Entre duas simulações de M_d, M' usa uma máquina de Turing N para gerar a cadeia *seguinte, na ordem lexicográfica*, em $\{1, 2, \ldots, r\}^*$. Em outras palavras, N irá gerar as cadeias $1, 2, \ldots, r, 11, 12, \ldots, rr, 111, \ldots$. Para $r = 2$, N é precisamente a máquina de Turing que computa a função binária sucessora (Exemplo 4.2.3); *sua* generalização para $r > 2$ é relativamente simples e direta.

M' é a máquina de Turing dada na Figura 4-22. Por $C^{1\to 2}$ queremos entender uma máquina de Turing simples que apaga a segunda fita e copia a primeira fita na segunda. B^3 é a máquina que gera a cadeia lexicograficamente seguinte na terceira fita. Por fim, M_d é a versão determinística de M operando sobre as fitas 2 e 3. Isso completa a descrição de M'.

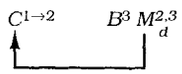


Figura 4-22

Afirmamos que M' pára em resposta a uma entrada w se e somente se alguma computação de M o faz. Suponha que M', de fato, pára em resposta à entrada w. Observando a Figura 4-22, vemos que M_d pára com seu terceiro cabeçote na fita, não varrendo um espaço em branco. Isso implica que em resposta a alguma cadeia $i_1 i_2 \ldots i_n \in \{1,2, \ldots, r\}^*$, quando iniciada com w em sua primeira fita e $i_1 i_2 \ldots i_n$ em sua segunda, M_d pára antes de alcançar os espaços em branco de sua segunda fita. Entretanto, isso significa que há uma computação de M que pára em resposta à entrada w. Inversamente, se

há uma parada na computação de M para a entrada w, digamos com n passos, então, após no máximo $r + r^2 + ... + r^n$ tentativas fracassadas, a cadeia em $\{1, 2, ..., r\}^*$ que corresponde às escolhas de parada na computação de M' serão geradas por B^3 e M_d deverá percorrer inclusive o último símbolo dessa cadeia. Portanto, M' irá parar, e assim, esta prova se completa. ■

Como esperávamos, a simulação de uma máquina de Turing não-determinística por outra determinística, não é uma simulação passo a passo, como foram todas as outras simulações que vimos anteriormente neste capítulo. Em lugar disso, ela segue um a um, por todas as possíveis computações da máquina de Turing não-determinística. Como resultado, é necessário *um número de passos exponencial* em n para simular uma computação de n passos da máquina não-determinística – enquanto todas as outras simulações descritas anteriormente são de fato *polinomiais*. Determinar se essa longa e indireta simulação é uma característica intrínseca do não-determinismo ou um artefato de nosso pobre entendimento sobre ele é uma profunda e importante questão em aberto, explorada nos Capítulos 6 e 7 deste livro.

Problemas para a Seção 4.5

4.5.1 Forneça (em notação abreviada) máquinas de Turing não-determinísticas que aceitem as linguagens seguintes:
(a) $a^*abb^*baa^*$
(b) $\{wvw^Ruu^R: w, u \in \{a, b\}^*\}$

4.5.2 Seja $M = (K, \Sigma, \delta, s, \{h\})$ a seguinte máquina de Turing não-determinística:

$K = \{q_0, q_1, h\}$
$\Sigma = \{a, \triangleright, \sqcup\}$
$s = q_0$
$\Delta = \{(q_0, \sqcup, q_1, a), (q_0, \sqcup, q_1, \sqcup), (q_1, \sqcup, q_1, \sqcup), (q_1, a, q_0, \rightarrow), (q_1, a, h, \rightarrow)\}$

Descreva todas as possíveis computações de M com cinco ou menos passos iniciando-se na configuração $(q_0, \triangleright\sqcup)$. Explique em palavras como M opera quando iniciada a partir dessa configuração. O que é o número r (da prova do Teorema 4.5.1) para essa máquina?

4.5.3 Apesar de as máquinas de Turing não-determinísticas não serem úteis para mostrar fechamento em relação à complementação das linguagens recursivas, elas são muito convenientes para mostrar outras propriedades de fechamento. Utilize máquinas de Turing não-determinísticas para mostrar que a classe de linguagens recursivas é fechada em relação à união, concatenação e estrela de Kleene. Repita isso para a classe das linguagens recursivamente enumeráveis.

4.6 GRAMÁTICAS

Neste capítulo temos apresentado vários dispositivos computacionais – precisamente, a máquina de Turing e suas diversas extensões – e demonstra-

mos que *todas elas são equivalentes em poder computacional.* Razoavelmente, todas essas várias espécies de máquinas de Turing podem ser chamadas *autômatos*, assim como seus parentes mais fracos – os autômatos finitos – os autômatos de pilha – estudados em capítulos anteriores. Assim como esses autômatos, as máquinas de Turing e suas extensões operam basicamente como *aceitadores de linguagem*, recebendo uma entrada, examinando-a e expressando de várias maneiras a sua aprovação ou desaprovação em relação a essa entrada. Como resultado, surgem duas importantes famílias de linguagens: as recursivas e as recursivamente enumeráveis.

Mas em capítulos anteriores vimos que há outras importantes famílias de dispositivos – conceitualmente muito diferentes dos aceitadores de linguagens – que podem ser utilizados para definir interessantes classes de linguagens: os *geradores de linguagem*, tais como as expressões regulares e as gramáticas livres de contexto. De fato, demonstramos que esses dois formalismos oferecem valiosas *caracterizações alternativas* das classes de linguagens definidas pelos aceitadores de linguagens. Este capítulo não ficaria completo sem tal manobra devemos agora introduzir um novo tipo de linguagem geradora que é uma generalização da gramática livre de contexto, chamada **gramática** (ou **gramática irrestrita**, para distingui-la da gramática livre de contexto) e mostrar que a classe das linguagens geradas por tal gramática é, precisamente, a classe das linguagens recursivamente enumeráveis.

Lembre-se do funcionamento essencial de uma gramática livre de contexto. Ela tem um alfabeto V dividido em duas partes: o conjunto de símbolos terminais Σ e o conjunto de símbolos não-terminais, $V - \Sigma$. Ela também tem um conjunto finito de regras, da forma $A \rightarrow u$, onde A é um símbolo não-terminal e $u \in V^*$. Uma gramática livre de contexto opera a partir do seu símbolo inicial S, que é um não-terminal, e vai repetidamente substituindo ocorrências do lado esquerdo de uma regra pelo lado direito correspondente, até que nenhuma substituição adicional possa ser feita.

Em uma gramática, todas essas mesmas convenções se aplicam, *exceto que os lados esquerdos das regras não consistem precisamente de simples não-terminais.* Em lugar disso, o lado esquerdo de uma regra pode consistir de qualquer cadeia de terminais e não-terminais, que contenha pelo menos um não-terminal. Um passo único, ou uma derivação, remove a ocorrência da subcadeia do lado esquerdo de uma regra e a substitui pelo lado direito correspondente. O produto final é, como na gramática livre de contexto, uma cadeia contendo somente terminais.

Definição 4.6.1: Uma **gramática** (ou **gramática irrestrita**, ou **sistema de regravação**) é a quádrupla $G = (V, \Sigma, R, S)$, onde

V é um alfabeto;
$\Sigma \subseteq V$ é o conjunto de símbolos **terminais**;
$V - \Sigma$ é o conjunto de símbolos **não-terminais**;
$S \in V - \Sigma$ é o símbolo **inicial**; e
R, o conjunto de **regras**, é um subconjunto finito de $(V^*(V - \Sigma)V^*) \times V^*$.

Escrevemos $u \rightarrow v$ se $(u, v) \in R$; escrevemos $u \Rightarrow_G v$ se e somente se para algum $w_1, w_2 \in V^*$ e alguma regra $u' \rightarrow v' \in R$, $u = w_1 u' w_2$ e $v = w_1 v' w_2$. Como sempre, $\Rightarrow_G^*$ é o fechamento transitivo reflexivo de $\Rightarrow_G$. Dizemos que uma

cadeia $w \in \Sigma^*$ é gerada por G se e somente se $S \Rightarrow_G^* w$; $L(G)$, a linguagem **gerada por** G, é o conjunto de todas as cadeias de Σ^* geradas por G.

Usa-se também aqui uma terminologia introduzida originalmente para gramáticas livre de contexto: **derivação** é uma seqüência da forma $w_0 \Rightarrow_G w_1 \Rightarrow_G \ldots \Rightarrow_G w_n$.

Exemplo 4.6.1: Qualquer gramática livre de contexto é uma gramática; de fato, uma gramática livre de contexto é uma gramática tal que o lado esquerdo de cada regra é um membro de $V - \Sigma$ além de ser membro de $V^*(V - \Sigma)V^*$. Em uma gramática, uma regra pode assumir a forma $uAv \rightarrow uwv$, a qual pode ser lida "substituir A por w no *contexto de u e v*". Naturalmente, as regras de uma gramática podem até ter uma forma mais geral que essa; entretanto, qualquer linguagem que possa ser gerada por uma gramática qualquer pode ser gerada por uma gramática na qual todas as regras sejam do tipo "substituição dependente do contexto" (Problema 4.6.3).✧

Exemplo 4.6.2: A seguinte gramática G gera a linguagem $\{a^n b^n c^n: n \geq 1\}$. $G = (V, \Sigma, R, S)$, onde

$$
\begin{aligned}
V &= \{S, a, b, c, A, B, C, T_a, T_b, T_c\}, \\
\Sigma &= \{a, b, c\}, \text{ e} \\
R &= \{S \rightarrow ABCS\}, \\
&\quad S \rightarrow T_c, \\
&\quad CA \rightarrow AC, \\
&\quad BA \rightarrow AB, \\
&\quad CB \rightarrow BC, \\
&\quad CT_c \rightarrow T_c c, \\
&\quad CT_c \rightarrow T_b c, \\
&\quad BT_b \rightarrow T_b b, \\
&\quad BT_b \rightarrow T_a b, \\
&\quad AT_a \rightarrow T_a a, \\
&\quad T_a \rightarrow \varepsilon\}.
\end{aligned}
$$

As primeiras três regras geram uma cadeia na forma $(ABC)^n T_c$. As três regras seguintes permitem que os A's, B's e C's na cadeia sejam reordenados corretamente, de tal modo que a cadeia se torne $A^n B^n C^n T_c$. Por fim, as demais regras permitem que T_c, seja deslocado para a esquerda, transformando todos os C's em c's e, então, convertendo-se em T_b. T_b, por sua vez, migra para a esquerda, transformando todos os B's em b's e tornando-se T_a e, por fim, T_a transforma todos os A's em a's, e é apagado.

É relativamente óbvio que qualquer cadeia da forma $a^n b^n c^n$ pode ser produzida dessa maneira. Naturalmente, muitas outras cadeias que contêm não-terminais podem ser produzidas; entretanto, não é difícil ver que o único modo de apagar todos os não-terminais é seguir o procedimento acima delineado. Portanto, somente as cadeias sobre $\{a, b, c\}$ que podem ser geradas por G são aquelas pertencentes ao conjunto $\{a^n b^n c^n: n \geq 1\}$. ✧

Evidentemente, a classe das linguagens geradas por gramáticas contém algumas linguagens que não são livres de contexto. Mas, precisamente, qual seria a amplitude dessa classe de linguagens? Sobretudo, de que forma ela se relaciona com as outras duas importantes extensões das linguagens

livres de contexto que vimos neste capítulo, ou seja, as linguagens recursivas e as recursivamente enumeráveis? De fato, a gramática desempenha em relação às máquinas de Turing, precisamente o mesmo papel que a gramática livre de contexto desempenha em relação aos autômatos de pilha e as expressões regulares, em relação aos autômatos finitos:

Teorema 4.6.1: *Uma linguagem é gerada por uma gramática se e somente se ela for recursivamente enumerável.*

Prova: *Somente se*. Seja $G = (V, \Sigma, R, S)$ uma gramática. Devemos projetar uma máquina de Turing M que semidecida a linguagem gerada por G. De fato, M será *não-determinística*; sua conversão para forma de uma máquina determinística que semidecide a mesma linguagem é garantida pelo Teorema 4.5.1.

M tem três fitas. A primeira fita contém a entrada w, que nunca é modificada. Na segunda fita, M tenta construir uma derivação de w a partir de S usando a gramática G; portanto M começa gravando S na segunda fita. Então, M continua executando ciclos correspondentes aos passos da derivação que está sendo efetuada. Cada ciclo se inicia com uma transição não-determinística, escolhendo arbitrariamente um entre $|R| + 1$ estados possíveis. Cada um dos primeiros $|R|$ desses $|R|+1$ estados é o início de uma seqüência de transições que aplica a regra gramatical que corresponde ao conteúdo corrente da segunda fita. Suponhamos que a regra escolhida seja $u \rightarrow v$. Nesse caso, M percorre não-deterministicamente sua segunda fita da esquerda para a direita, parando em algum símbolo. Verifica que os próximos $|u|$ símbolos correspondem a u, apaga u, desloca o resto da cadeia apropriadamente, para criar área suficiente para v, e então grava v no lugar de u. Se a verificação falhar, M inicia uma computação em ciclo interminável – a tentativa corrente de gerar w falhou.

A sequência de transições iniciada no estado $|R| + 1$ de M verifica se a cadeia atual é igual à entrada w. Sendo esse o caso, M pára e aceita w: w pode de fato ser gerado por G. E, se as cadeias foram diferentes, M novamente inicia um ciclo infinito.

É claro que as únicas computações possíveis de parada de M são aquelas que correspondem a uma derivação da cadeia w pela gramática G. Portanto, M aceita w se e somente se $w \in L(G)$ e a direção *somente se* foi provada.

Se. Seja agora $M = (K, \Sigma, \delta, s, \{h\})$ uma máquina de Turing. Convém assumir que Σ e K sejam disjuntos e que nenhum deles contenha o novo marcador de final $\triangleleft$. Também assumimos que se M pára, isso sempre ocorre na configuração $(h, \triangleright\sqcup)$ – isto é, após ter apagado sua fita. Qualquer máquina de Turing que semidecide uma linguagem pode ser transformada em outra equivalente que satisfaz as condições acima. Devemos construir uma gramática $G = (V, \Sigma - \{\sqcup, \triangleright\}, R, S)$ que gera a linguagem $L \subseteq (\Sigma - \{\sqcup, \triangleright\})^*$ *semidecidida* por M.

O alfabeto V consiste de todos os símbolos de Σ, todos os estados de K, mais o símbolo inicial S e o marcador de final $\triangleleft$. Intuitivamente, as derivações de G simularão as *computações de M em ordem inversa*. Devemos simular configurações $(q, \triangleright u\underline{a}w)$ usando a cadeia $\triangleright uaqw \triangleleft$ – isto é, usando o conteúdo da fita, com o estado atual inserido imediatamente após o símbolo na posição corrente do cabeçote. As regras de G simulam

os *movimentos* de M às avessas, ou seja, para cada $q \in K$ e $a \in \Sigma$, G tem estas regras, dependendo de $\delta(q, a)$.

(1) Se $\delta(q, a) = (p, b)$ para algum $p \in K$ e $b \in \Sigma$, então G deve apresentar uma regra $bp \rightarrow aq$.
(2) Se $\delta(q, a) = (p, \rightarrow)$ para algum $p \in K$, então G deve ter uma regra $abp \rightarrow aqb$ para cada um dos $b \in \Sigma$ e também a regra $a \sqcup p \triangleleft \rightarrow aq \triangleleft$ (a última regra simula às avessas a extensão da fita para a direita, por um novo espaço em branco).
(3) Se $\delta(q, a) = (p, \leftarrow)$ para algum $p \in K$ e $a \neq \sqcup$, então G deve possuir uma regra $pa \rightarrow aq$.
(4) Se $\delta(q, \sqcup)$ $(p, \leftarrow)$ para algum $p \in K$, então G deve conter uma regra $pab \rightarrow aqb$ para cada um dos $b \in \Sigma$, e também a regra $p \triangleleft \rightarrow \sqcup q \triangleleft$ que simula às avessas a remoção de espaços em branco excedentes.

Por fim, G deve incluir certas transições para representar o início da computação (no caso, isto representará o final da derivação) e também para o final da computação (o início da derivação). A regra

$$S \rightarrow \triangleright \sqcup h \triangleleft$$

força a derivação a iniciar-se exatamente onde uma computação de aceitação seria finalizada. As outras regras são $\triangleright \sqcup s \rightarrow \varepsilon$, apagando a parte do final da cadeia à esquerda da entrada, e $\triangleleft \rightarrow \varepsilon$, apagando o marcador de final da cadeia de entrada.

O seguinte resultado torna precisa a nossa noção de que G simula, em ordem inversa, as computações de M:

Para quaisquer duas configurações $(q_1, u_1\underline{a_1}w_1)$ *e* $(q_2, u_2\underline{a_2}w_2)$ *de* M, *temos que* $(q_1, u_1\underline{a_1}w_1) \vdash_M (q_2, u_2\underline{a_2}w_2)$ *se e somente se* $u_2a_2q_2w_2 \triangleleft \Rightarrow_G u_1a_1q_1w_1 \triangleleft$.

A prova dessa afirmação é uma análise simples e direta da natureza dos movimentos de M e é deixada como um exercício.

Agora completamos a prova do teorema, mostrando que, para todos os $w \in (\Sigma - \{\triangleright, \sqcup\})^*$, M pára em resposta à cadeia w se e somente se $w \in L(G)$. Por outro lado, $w \in L(G)$ se e somente se

$$S \Rightarrow_G \triangleright \sqcup h \triangleleft \Rightarrow_G^* \triangleright sw \triangleleft \Rightarrow_G w \triangleleft \Rightarrow_G w,$$

como $S \rightarrow \triangleright \sqcup h \triangleleft$ é a única regra que envolve S e as regras $\triangleright \sqcup s \rightarrow \varepsilon$ e $\triangleleft \rightarrow \varepsilon$ são as únicas regras que permitem a remoção do estado e o marcador de final $\triangleleft$. Agora, pelo que foi mostrado acima, $\triangleright \sqcup h \triangleleft \Rightarrow_G^* \triangleright \sqcup sw \triangleleft$ se e somente se $(s, \triangleright \underline{\sqcup} w) \vdash_M^* (h, \triangleright \underline{\sqcup})$, o que acontece se e somente se M pára em resposta a w. Isso completa a prova do Teorema. ■

O Teorema 4.6.1 identifica gramáticas com um aspecto das máquinas de Turing que julgamos irreais – a semidecisão, com sua definição unilateral que não provê informações quando a entrada não está na linguagem. Isso é consistente com as propriedades das gramáticas: se uma cadeia puder ser gerada pela gramática, podemos pacientemente procurar todas as possíveis derivações, iniciando das menores em direção às maiores, até localizarmos

a correta. Mas se não existir nenhuma derivação, esse processo continuará indefinidamente, sem nos dar qualquer informação útil.

Nota-se, no entando, que é também possível identificar as gramáticas com instrumentos de computação mais úteis, baseados, em máquinas de Turing.

Definição 4.6.2: Seja $G = (V, \Sigma, R, S)$ uma gramática e $f: \Sigma^* \to \Sigma^*$ uma função. Dizemos que G **computa** f se, para todo $w, v \in \Sigma^*$, verifica-se que

$$SwS \Rightarrow_G^* v \text{ se e somente se } v = f(w).$$

Em outras palavras, a cadeia que forma a entrada w, acrescida dos delimitadores direito e esquerdo, produz exatamente uma cadeia em Σ^*: que é o valor correto de $f(w)$.

Uma função $f: \Sigma^* \to \Sigma^*$ é dita **gramaticalmente computável** se e somente se houver uma gramática G que a compute.

Deixaremos como exercício a demonstração do teorema seguinte, que é uma variante da demonstração do Teorema 4.6.1 – veja o Problema 4.6.4.

Teorema 4.6.2: *A função $f: \Sigma^* \to \Sigma^*$ é recursiva se e somente se ela for gramaticalmente computável.*

Problemas para a Seção 4.6

4.6.1 (a) Apresente uma derivação da cadeia *aaabbbccc* na gramática do Exemplo 4.6.2.
(b) Prove criteriosamente que a gramática no Exemplo 4.6.2 gera a linguagem $L = \{a^n b^n c^n: n \geq 1\}$.

4.6.2 Apresente gramáticas que gerem as seguintes linguagens:
(a) $\{ww: w \in \{a, b\}^*\}$
(b) $\{a^{2^n} n \geq 0\}$
(c) $\{a^{n^2} n \geq 0\}$

4.6.3 Mostre que qualquer gramática pode ser convertida em uma gramática equivalente em que todas as regras têm a forma $uAv \to uwv$, com $A \in V - \Sigma$ e $u, v, w \in V^*$.

4.6.4 Prove o Teorema 4.6.2. (Para a direção *somente se*, dada uma gramática G, mostre como construir uma máquina de Turing que, para a entrada w, produz uma cadeia $u \in \Sigma^*$, tal que $SwS \Rightarrow_G^* u$, se tal cadeia u existir. Para a direção *se*, utilize uma simulação semelhante à que foi utilizada na prova do Teorema 4.6.1, porém não às avessas.

4.6.5 Uma excentricidade no uso de gramáticas para computar funções é que a ordem em que as regras são aplicadas não é determinada. Nas alternativas seguintes, creditadas a A. A. Markov (1903-1979), essa indeterminação é evitada. Um **Sistema de Markov** é uma quádrupla $G = (V, \Sigma, R, R_1)$, onde V é um alfabeto; $\Sigma \subseteq V$; R é uma *seqüência* finita (não um conjunto) de regras $(u_1 \to v_1, \ldots, u_k \to v_k)$, onde $u_i, v_i \in V^*$; e R_1 é um subconjunto das regras de R. A relação

$w \Rightarrow_G w'$ é definida como segue: se há um i tal que u_i é uma subcadeia de w, sendo i o menor desses números e w_1 a cadeia mais curta tal que $w = w_1 u_i w_2$; então $w \Rightarrow_G w'$ desde que $w' = w_1 v_i w_2$. Portanto, há apenas uma regra a ser aplicada em cada caso, e o ponto de aplicação é sempre único. Dizemos nesse caso que G **computa** uma função f: $\Sigma^* \to \Sigma^*$, se para todos os $u \in \Sigma^*$

$$u = u_0 \Rightarrow_G u_1 \Rightarrow_G \ldots \Rightarrow_G u_{n-1} \Rightarrow_G u_n = f(u),$$

e na primeira vez que uma regra de R_1 for utilizada, essa vez será a última, $u_{n-1} \Rightarrow_G u_n$. Mostre que uma função é computável por um sistema de Markov se e somente se ela for recursiva. (A prova é similar à do Teorema 4.6.2.)

4.7 FUNÇÕES NUMÉRICAS

Vamos agora observar a computação de um ponto de vista completamente diferente, que não seja baseado em qualquer formalismo computacional explícito ou de manipulação de informação, como ocorre com as máquinas de Turing ou com as gramáticas. Em lugar disso, focalizaremos o objetivo da computação a realizar: as *funções de números para números*. Por exemplo, é natural que o valor da função

$$f(m,n) = m \cdot n^2 + 3 \cdot m^{2 \cdot m+17}$$

possa ser computado para quaisquer valores dados de m e n, porque se trata de uma composição de funções – adição, multiplicação e exponenciação, mais algumas poucas constantes – que por sua vez podem ser computadas. E como poderemos saber se uma operação de potenciação pode ser computada? Porque ela pode ser *recursivamente definida* em termos de uma função mais simples (no caso, a multiplicação). Afinal de contas, m^n vale 1 se $n = 0$ e, caso contrário, valerá $m \cdot m^{n-1}$. A própria multiplicação pode ser definida recursivamente em termos da operação de adição – e assim por diante.

No princípio, devemos ser hábeis para iniciar o processo com funções de números naturais para números naturais, que sejam tão simples que possam ser inequivocamente provadas como computáveis (por exemplo, a função identidade e a função sucessor $\text{succ}(n) = n + 1$) e então combiná-las, vagarosa e pacientemente, através de combinadores que sejam, também, por sua vez, muito elementares e obviamente computáveis – tais como os operadores de composição e de definição recursiva – para então, por fim, obter uma classe de funções de números para números que sejam completamente gerais e não triviais. Nesta seção, vamos nos incumbir desse exercício. Significativamente, a noção de computação assim definida será então provada como idêntica às noções que nos chegam através das conclusões tiradas por outros caminhos neste capítulo – máquinas de Turing, suas variantes e gramáticas – caminhos esses que se mostram muito diferentes em espírito, extensão e detalhes.

Definição 4.7.1: Iniciamos definindo algumas funções extremamente simples de $\mathbf{N}^k$ para $\mathbf{N}$, para valores de $k \geq 0$ (a função 0-ária é, naturalmente, uma constante, uma vez que não depende de nada). As **funções básicas** são as seguintes:

(a) Para qualquer $k \geq 0$, a **função nula** k**-ária** é definida como $\text{zero}_k(n_1,\ldots, n_k) = 0$ para todo $n_1,\ldots, n_k \in \mathbf{N}$.
(b) Para qualquer $k \geq j > 0$, a j-ésima **função identidade** k-**ária** - simplesmente a *função* $\text{id}_{k,j}(n_1,\ldots,n_k) = n_j$ para todo $n_1,\ldots,n_k \in \mathbf{N}$
(c) A **função sucessor** é definida como $\text{succ}(n) = n + 1$ para todo $n \in \mathbf{N}$.

A seguir, apresentaremos duas maneiras simples de combinar funções para obter funções ligeiramente mais complexas.

(1) Seja $k, \ell \geq 0$, g: $\mathbf{N}^k \to \mathbf{N}$ uma função k-ária, e sejam $h_1,\ldots$ funções ℓ-árias. Nessas condições, define-se a **composição de** g **com** $h_1,\ldots, h_k$ como sendo a função ℓ-ária seguinte.

$$f(n_1,\ldots,n_\ell) = g(h_1(n_1,\ldots,n_\ell),\ldots,h_k(n_1,\ldots,n_\ell)).$$

(2) Suponha que $k \geq 0$, que g seja uma função k-ária e que h seja uma função $(k + 2)$-ária. Então, a **função definida recursivamente** a partir de g **e** h é a função $(k + 1)$-ária f definida como:

$$f(n_1,\ldots,n_k, 0) = g(n_1,\ldots, n_k),$$
$$f(n_1,\ldots, n_k, m + 1) = h(n_1,\ldots, n_k, m, f(n_1,\ldots,n_k, m))$$

para todos $n_1,\ldots, n_k, m \in \mathbf{N}$.

São **funções recursivas primitivas** todas as funções básicas e também todas as funções que podem ser obtidas a partir delas por qualquer número de aplicações sucessivas das operações de composição e de definição recursiva.

Exemplo 4.7.1: A função plus2, definida como $\text{plus2}(n) = n + 2$ é recursiva primitiva, pois pode ser obtida a partir da função básica succ por uma composição com ela mesma. Em particular, seja $k = \ell = 1$ em (1) da definição 4.7.1 e $g = h_1 = \text{succ}$.

Analogamente, a função binária plus, definida como $\text{plus}(m, n) = m + n$, é recursiva primitiva, porque ela pode ser definida do mesmo modo a partir de combinações, das funções identidade, nula e sucessor. Em particular, na Parte 2 da definição 4.7.1, fazendo $k = 1$, e considerando g como a função $\text{id}_{1,1}$ e h como a função $h(m, n, p) = \text{succ}(\text{id}_{3,3}(m, n, p))$ – a composição de succ com $\text{id}_{3,3}$. A função assim obtida, definida recursivamente, é precisamente a função plus:

$$\text{plus}(m, 0) = m$$
$$\text{plus}(m, n + 1) = \text{succ}(\text{plus}(m, n)).$$

Continuando, a função de multiplicação $\text{mult}(m, n) = m \cdot n$ pode ser definida recursivamente como

$$\text{mult}(m, 0) = \text{zero}(m),$$
$$\text{mult}(m, n + 1) = \text{plus}(m, \text{mult}(m, n)),$$

e a função $\exp(m, n) = m^n$, como

$$\exp(m, 0) = \text{succ}(\text{zero}(m)),$$
$$\exp(m, n + 1) = \text{mult}(m, \exp(m, n)).$$

Portanto, todas essas funções são recursivas primitivas.

Todas as funções *constantes* da forma $f(n_1, \ldots, n_k) = 17$ são recursivas primitivas, pois podem ser obtidas pela composição de uma função nula apropriada com a função succ, nesse exemplo, dezessete vezes. Além disso, a função *sinal* sgn(n), que vale zero se $n = 0$ e, um nos demais casos, também é recursiva primitiva: sgn(0) = 0 e sgn($n + 1$) = 1.

Para melhor legibilidade, escreveremos $m + n$ em vez de plus(m, n); $m \cdot n$, em vez de mult(m, n), e $m \uparrow n$, em lugar de exp(m, n). Todas as funções numéricas, tais como

$$m \cdot (n + m^2) + 178^m$$

são, portanto, recursivas primitivas, uma vez que elas podem ser obtidas a partir das funções anteriormente definidas, por meio de composições sucessivas.

Uma vez que estamos limitados aos números naturais, não podemos ter as operações de subtração e divisão convencionais. Entretanto, podemos definir certas funções úteis, como $m \sim n = \max\{m - n, 0\}$ e as funções div(m, n) e rem(m, n) (o quociente inteiro e o resto da divisão de m por n; convencionando que ambos são nulos quando $n = 0$). Primeiro, definimos a função *predecessora*:

$$\begin{aligned} \text{pred}(0) &= 0, \\ \text{pred}(n + 1) &= n, \end{aligned}$$

da qual obtemos nossa função de "subtração não-negativa"

$$\begin{aligned} m \sim 0 &= m, \\ m \sim n + 1 &= \text{pred}(m \sim n). \end{aligned}$$

As funções quociente e resto serão definidas posteriormente em um exemplo.✧

É relativamente claro que podemos calcular o valor de qualquer função recursiva primitiva dados os valores de seus argumentos. Também é igualmente evidente que podemos determinar a validade de *asserções* sobre números, como por exemplo:

$$m \cdot n > m^2 + n + 7,$$

para quaisquer valores fornecidos de m e n. É adequado, para isso, definir um **predicado recursivo primitivo** como sendo uma função recursiva primitiva que somente assume os valores 0 e 1. Intuitivamente, um predicado recursivo primitivo, como greater-than (m, n), irá representar uma *relação* que pode ou não ser aplicável aos valores m e n. Se a relação se aplicar, então o predicado recursivo primitivo assumirá o valor 1, caso contrário, será nulo.

Exemplo 4.7.2: A função iszero, que vale 1 se $n = 0$ e zero se $n > 0$, é um predicado recursivo primitivo, definido recursivamente da seguinte maneira:

$$\begin{aligned} \text{iszero}(0) &= 1, \\ \text{iszero}(m + 1) &= 0. \end{aligned}$$

Analogamente podemos definir, isone(0) = 0 e isone(n + 1) = iszero(n). O predicado positive(n) é idêntico ao já definido sgn(n). Além disso, greater-than-or-equal(m, n), denotado $m \geq n$, pode ser definido como iszero($n \sim m$). Sua *negação*, less-than(m, n) é naturalmente 1 ~ greater-than-or-equal (m, n). Em geral, a negação de qualquer predicado recursivo primitivo também é um predicado recursivo primitivo. De fato, assim são a *disjunção* e a *conjunção* de dois predicados recursivos primitivos: $p(m, n)$ or $q(m, n)$ é 1 ~ iszero($p(m, n) + q(m, n$ e $p(m, n)$ and $q(m, n)$ é 1 ~ iszero($p(m, n) \cdot q(m, n)$). Por exemplo, equals(m, n pode ser definido como a conjunção de greater-than-or-equal (m, n) e greater-than-or-equal (n, m).

Além disso, se f e g são funções recursivas primitivas e p é um predicado recursivo primitivo, todos os três com o mesmo número de argumentos k, então a **função definida por casos**

$$f(n_1,\ldots, n_k) = \begin{cases} g(n_1,\ldots, n_k), & \text{se } p(n_1,\ldots,n_k); \\ h(n_1,\ldots, n_k), & \text{caso contrário} \end{cases}$$

também é recursiva primitiva, uma vez que pode ser reescrita como:

$$f(n_1,\ldots, n_k) = p(n_1,\ldots, n_k) \cdot g(n_1,\ldots, n_k) + (1 \sim p(n_1,\ldots, n_k)) \cdot h(n_1,\ldots, n_k)$$

Como se pode facilmente constatar, a definição por casos constitui uma abreviação de grande utilidade.✧

Exemplo 4.7.3: Podemos finalmente definir div e rem.

$$\text{rem}(0, n) = 0,$$

$$\text{rem}(m + 1, n) = \begin{cases} 0 & \text{se equal(rem}(m, n), \text{pred}(n)); \\ \text{rem}(m, n) + 1 & \text{caso contrário,} \end{cases}$$

e

$$\text{div}(0, n) = 0,$$

$$\text{div}(m + 1, n) = \begin{cases} \text{div}(m + 1, n)+1, & \text{se equal (rem } (m, n), \text{pred}(n)); \\ \text{div}(m, n) & \text{caso contrário.} \end{cases}$$

Outra função interessante, que se revela como sendo recursiva primitiva, é digit(m, n, p), que representa *o m-ésimo dígito menos significativo da representação do número n em base p.* (Como uma ilustração do uso de digit, o predicado odd(n), representando o fato de n ser ímpar, pode ser escrito alternativamente como digit (1, n, 2).) É fácil verificar que digit(m, n, p) pode ser definido como div(rem(n, $p \uparrow m$), $p \uparrow (m \sim 1)$).✧

Exemplo 4.7.4: Se $f(n, m)$ é uma função recursiva primitiva, então a *função*

$$\text{sum}_f(n, m) = f(n, 0) + f(n, 1) + \ldots + f(n, m)$$

também é recursiva primitiva, já que ela pode ser definida como $\text{sum}_f(n, 0) = 0$ e $\text{sum}_f(n, m + 1) = \text{sum}_f(n, m) + f(n, m + 1)$. Também podemos definir desse modo as *conjunções e disjunções ilimitadas* dos predicados. Por exemplo, se $p(n, m)$ é um predicado, então a disjunção

$$p(n, 0) \text{ or } p(n, 1) \text{ or } p(n, 2) \text{ or } \ldots \text{ or } p(n, m)$$

pode ser reescrito como $\mathsf{sgn}(\mathsf{sum}_p(n, m))$. ✧

Evidentemente, partindo dos elementos primitivos da definição 4.7.1, é possível mostrar que várias funções extremamente complexas são também recursivas primitivas. Entretanto, funções recursivas primitivas não são capazes de representar todas as funções que poderíamos considerar como computáveis. Isso pode ser melhor colocado usando um argumento baseado na *diagonalização*:

Exemplo 4.7.5: O conjunto infinito de funções recursivas primitivas é *enumerável*. Essa é a razão por que cada função recursiva primitiva pode, a princípio, ser definida nos termos das funções básicas e, portanto, ser representada como uma cadeia sobre um alfabeto finito; o alfabeto deve conter símbolos para representar as funções de identidade, sucessor e nula, para as recursões e composições primitivas, além de parênteses e os símbolos 0 e 1 utilizados para indexar, em binário, funções básicas como $\mathsf{id}_{17,11}$ (veja a Seção 5.2 para outro uso dessa indexação, desta vez para representar todas as possíveis máquinas de Turing). Poderemos então enumerar todas as cadeias sobre o alfabeto, e manter somente aquelas que sejam definições corretas de funções recursivas primitivas – de fato, podemos decidir manter apenas as funções recursivas primitivas *unárias*, definidas com um argumento apenas.

Suponha, então, que listamos todas as funções recursivas primitivas unárias, como cadeias, em ordem lexicográfica

$$f_0, f_1, f_2, f_3, \cdots$$

Nessas condições, dado qualquer número $n \geq 0$, podemos localizar f_n, a n-ésima função unária recursiva primitiva nessa lista, e então utilizar sua definição para computar o número $g(n) = f_n(n) + 1$. Claramente, $g(n)$ é uma função computável – apenas delineamos uma forma de computá-la. Adicionalmente, g não é uma função recursiva primitiva. Porque, caso fosse, existiria algum $m \geq 0$ tal que $g = f_m$, e em conseqüência, deveríamos ter $f_m(m) = f_m(m) + 1$, o que é absurdo.

Esse é um argumento de diagonalização. Ele depende da disponibilidade de uma lista seqüencial contendo todas as funções recursivas primitivas; a partir dessa lista é possível definir alguma função que difira de todas aquelas pertencentes à lista e que, portanto, não pode ela própria estar na lista. Compare esse argumento com a prova do Teorema 1.5.2, afirmando que $2^{\mathbf{N}}$ é não enumerável. Naquele caso, iniciamos com uma lista que deveria abranger todos os membros de $2^{\mathbf{N}}$, e obtivemos um membro de $2^{\mathbf{N}}$ que não estava na lista. ✧

Evidentemente, qualquer forma de definir funções, de modo que elas abranjam tudo o que podemos chamar de "computável", não pode estar baseada somente em operações simples tais como composição e definição recursiva, as quais produzem funções que podem sempre e seguramente ser reconhecidas como tal e, portanto, enumeradas. Descobrimos, assim, um interessante fato sobre formalismos de computação: qualquer formalismo dessa natureza, cujos membros (dispositivos computacionais) sejam *auto-evidentes* (isto é, são tais que, dada uma cadeia, é possível decidir facil-

mente se ela, codifica um dispositivo computacional desse formalismo [illegible] ser ou pouco poderoso (como autômatos de estado finito e funções recursivas primitivas) ou tão genérico que são inúteis na prática (como as máquinas de Turing que podem não parar em resposta a uma entrada). Qualquer formalismo que possa representar todas as funções computáveis, e somente essas, deve incluir *funções que não são auto-evidentes* (exatamente como não é auto-evidente se uma máquina de Turing pára em resposta a todas as entradas e, portanto, decide uma linguagem). Definimos a seguir uma operação mais sutil sobre funções, correspondente à conhecida primitiva computacional da *iteração ilimitada* – algo similar a um comando **While**. Como veremos adiante, a iteração ilimitada realmente propicia a possibilidade de que o resultado possa *não* ser uma função.

Definição 4.7.2: Seja g uma função $(k+1)$-ária, para algum $k \geq 0$. Chama-se **minimização** de g a função k-ária f definida como segue:

$$f(n_1,\ldots,n_k) = \begin{cases} \text{o menor } m \text{ tal que } g(n_1,\ldots,n_k, m) = 1, \\ \quad \textit{se} \text{ tal } m \text{ existir;} \\ 0, \text{ caso contrário.} \end{cases}$$

Denotaremos a minimização de g por $\mu\, m[g(n_1,\ldots, n_k,m) = 1]$.

Apesar de a minimização de uma função g ser sempre bem-definida não há um método óbvio para computá-la – mesmo que saibamos como se computa g. O método óbvio

m: = 0;
enquanto $g(n_1,\ldots,n_k, m) \neq 1$ faça $m := m + 1$;
emitir m

não é um algoritmo, porque ele pode, eventualmente, não terminar nunca.

Vamos então chamar uma função g **minimizável**, se o método acima sempre terminar. Em outras palavras, uma função $(k+1)$-ária g é *minimizável* se ela apresentar a seguinte propriedade: para cada $n_1,\ldots,n_k \in \mathbf{N}$, há um $m \in \mathbf{N}$ tal que $g(n_1, \ldots, n_k,m) = 1$.

Por fim, dizemos que uma função é **μ-recursiva** se ela puder ser obtida a partir das funções básicas pela aplicação das operações de composição, definição recursiva e *minimização* de *funções minimizáveis*.

Note-se que, agora, não mais podemos repetir o argumento de diagonalização do exemplo 4.7.5 para apresentar uma função computável que não seja μ-recursiva. A questão é que, dada uma definição apropriada de uma função μ-recursiva, *não é claro se de fato ela define uma função μ-recursiva ou não* — isto é, se todas as aplicações da minimização nessa definição de fato atuam sobre funções minimizáveis!

Exemplo 4.7.6: Definimos o inverso da operação de adição (a função ~) e o inverso da multiplicação (a função div); mas e quanto ao inverso da exponencia-

ção – o *logaritmo*? Utilizando a minimização† podemos definir a função *logaritmo*: log(m, n) é a menor potência à qual devemos elevar $m + 2$ para obter um inteiro pelo menos tão grande quanto $n + 1$ (isto é, log(m, n) = $\lceil \log_{m+2} (n + 1) \rceil$; temos utilizado $m + 2$ e $n + 1$ como argumentos para evitar as armadilhas matemáticas na definição de $\log_m n$, quando $m < 1$ ou $n = 0$). A função log é definida como segue:

$$\log (m, n) = \mu p[\text{greather-than-or-equal}((m + 2) \uparrow p, n + 1)].$$

Note que essa é uma definição apropriada de uma função μ-recursiva, uma vez que a função $g(m, n, p)$ = greather-than-or-equal $((m + 2) \uparrow p, n + 1)$ é minimizável: de fato, para qualquer m, $n \geq 0$ há $p \geq 0$, tal que $(m + 2)^p \geq n$ – porque, elevando um inteiro ≥ 2 a potências cada vez maiores, podemos obter inteiros arbitrariamente grandes.✧

Podemos agora provar o principal resultado desta seção:

Teorema 4.7.1: *A função f: $\mathbf{N}^k \rightarrow \mathbf{N}$ é μ-recursiva se e somente se ela for recursiva (isto é, computável por uma máquina de Turing).*

Prova: *Somente se*: Seja f uma função μ-recursiva. Então, ela é definida a partir das funções básicas pela aplicação das operações de composição, definição recursiva e minimização de funções minimizáveis. Devemos mostrar que, nessas condições, f é computável por máquina de Turing.

É fácil constatar que as funções básicas são todas recursivas: vimos que a execução da função sucessor (Exemplo 4.2.3) e o restante das funções envolve apenas apagar alguma ou todas as entradas.

Assim, seja f: $\mathbf{N}^k \rightarrow \mathbf{N}$ a composição das funções g: $\mathbf{N}^l \rightarrow \mathbf{N}$ e $h_1, \ldots, h_l$: $\mathbf{N}^k \rightarrow \mathbf{N}$, onde, por indução, sabemos como computar g e os h_i's. Então, podemos computar f como segue (nesse e em outros casos apresentaremos programas, no estilo dos programas para máquinas de Turing de acesso aleatório, que computam essas funções; é relativamente fácil ver que o mesmo efeito pode ser obtido com o auxílio da máquinas de Turing padrão):

$m_1 := h_1(n_1, \ldots, n_k)$;
$m_2 := h_2(n_1, \ldots, n_k)$;
⋮
$m_l := h_l(n_1, \ldots, n_k)$;
emitir $g(m_1, \ldots, m_l)$

Analogamente, se f for definido recursivamente a partir de g e h (lembre-se da definição), então $f(n_1, \ldots, n_k, m)$ pode ser computado pelo seguinte programa:

$v := g(n_1, \ldots, n_k)$;
se $m = 0$ então emitir v
senão para $i := 1, 2, \ldots, m$ faça
    $v := h(n_1, \ldots, n_k, i - 1, v)$;
emitir v.

† A função logaritmo pode ser definida sem o auxílio da operação de minimização, como pode ser visto no Problema 4.7.2. Nosso uso de minimização aqui é somente para fins de ilustração e conveniência.

Por fim, seja f definida como $\mu\, m[g(n_1,\ldots,n_k, m)]$, onde g é minimizável e computável. Então, f pode ser computado pelo programa

$m := 0$;
enquanto $g(n_1,\ldots,n_k, m) \neq 1$ faça $m := m + 1$;
emitir m

Caso g seja minimizável, o algoritmo acima sempre irá terminar e emitir um número.

Provamos, assim, que todas as funções básicas são recursivas e que a composição e as definições recursivas de funções recursivas e a minimização de funções recursivas minimizáveis são recursivas; devemos, então, concluir que todas as funções μ-recursivas são recursivas. Isso completa a direção *somente se* desta prova.

Se: Suponha que uma máquina de Turing $M = (K, \Sigma, \delta, s, \{h\})$ computa uma função f: $\mathbf{N} \rightarrow \mathbf{N}$ – assumimos, para simplificar a apresentação, que f é unária; os demais casos podem ser obtidos mediante uma simples extensão (veja o Problema 4.7.5). Devemos mostrar que f é μ-recursiva. Devemos, para isso, aos poucos definir certas funções μ-recursivas que representam o comportamento de M e sua operação, até termos acumulado material suficiente para definir a própria função f.

Sem perda de generalidade, vamos admitir que K e Σ sejam disjuntos. Seja $b = |\Sigma| + |K|$, e vamos definir um mapeamento $\mathbf{E}$ de $\Sigma \cup K$ para $\{0, 1,\ldots, b-1\}$, tal que $\mathbf{E}(0) = 0$ e $\mathbf{E}(1) = 1$ – lembre-se de que, uma vez que M computa uma função numérica, seu alfabeto deve conter os símbolos 0 e 1. Utilizando esse mapeamento, poderemos representar as configurações de M como inteiros na base b. A configuração $(q, a_1a_2 \ldots \underline{a_k} \ldots a_n)$, onde os a_i's são símbolos de Σ, será representada como um inteiro, em base b, $a_1a_2\ldots a_kqa_{k+1}\ldots a_n$, isto é, como o inteiro

$$\mathbf{E}(a_1)b^n + \mathbf{E}(a_2)b^{n-1} + \ldots + \mathbf{E}(a_k)b^{n-k+1} + \mathbf{E}(q)b^{n-k} + \mathbf{E}(a_{k+1})b^{n-k-1} + \ldots + \mathbf{E}(a_n)$$

Agora estamos prontos para estabelecer a definição de f como uma função μ-recursiva. Em última instância, f será definida como

$$f(n) = \text{num}(\text{output}(\text{last}(\text{comp}(n)))).$$

num é uma função que recebe um inteiro cuja representação em base b, é uma cadeia de 0's e 1's e produz o valor binário dessa cadeia. output leva o inteiro representado em base b em uma configuração de parada na forma $\triangleright \sqcup hw$ e omite os primeiro três símbolos $\triangleright \sqcup h$. comp(n) é o número cuja representação em base b é a justaposição da única seqüência de configurações que se inicia com $\triangleright \sqcup sw$, onde w é a representação binária codificada de n, e termina com $\triangleright \sqcup hw'$, onde w' é a representação binária codificada de $f(n)$; tal seqüência existe, uma vez que assumimos que M computa alguma função – em particular, f. Finalmente, last recebe um inteiro representando a justaposição de configurações e extrai a última configuração da seqüência (a parte entre o último $\triangleright$ e o final).

Naturalmente, temos de definir todas essas funções. Faremos mais algumas definições abaixo, deixando o restante como um exercício (Problema 4.7.4). Vamos iniciar pela função last. Podemos definir lastpos(n), a última

(mais à direita, menos significativa) posição na cadeia codificada de n em base b onde um ▷ ocorre:

$$\text{lastpos}(n) = \mu\, m[\text{equal}(\text{digit}(m, n, b), \mathbf{E}(\triangleright)) \text{ or } \text{equal}(m, n)].$$

Note-se que, a fim de fazermos a função entre colchetes minimizável, permitimos que lastpos(n) seja o próprio n se nenhum ▷ for encontrado em n. Esse é outro uso superficial da minimização, uma vez que essa função pode ser facilmente redefinida sem utilizar minimização. Podemos então definir último(n) como rem(n, $b \uparrow$ lastpos(n)). Também podemos definir rest(n), a seqüência que permanece após a exclusão da last(n), como div(n, $b \uparrow$ lastpos(n)).

output(n) é simplesmente rem(n, $b \uparrow \log(b \sim 2, n \sim 1) \sim 2$) —lembre-se de nossa convenção de que os argumentos de log devem ser reduzidos em 2 e 1 unidade, respectivamente.

A função num(n) pode ser escrita como sendo a soma

$$\text{digit}(1, n, b) \cdot 2 + \text{digit}(2, n, b) \cdot 2 \uparrow 2 + \ldots + \text{digit}(\log(b \sim 2, n \sim 1), n, b) \cdot 2 \uparrow \log(b \sim 2, n \sim 1).$$

Essa é uma função μ-recursiva uma vez que tanto o comando sum como o limite log (b ~ 2, n ~ 1) o são. Sua função inversa, bin(n), que mapeia qualquer inteiro para a cadeia que é sua codificação binária, codificada novamente como um inteiro em base b, é muito similar, com os papéis de 2 e b invertidos.

A função mais interessante (e difícil de expressar) na definição de $f(n)$ acima é comp(n), a qual mapeia n para a seqüência de configurações de M que realizam a computação de $f(n)$ – de fato, o inteiro em base b que codifica essa seqüência. No nível mais alto, trata-se simplesmente de:

$$\text{comp}(n) = \mu\, m[\text{iscomp}(m, n) \text{ and } \text{halted}(\text{last}(m))], \qquad (1)$$

onde iscomp(m, n) é um predicado afirmando que m é a seqüência de configurações em uma computação, não necessariamente de parada, iniciando a partir de ▷$\underline{s}$**b**(n). (Esse é *o único ponto dessa prova, na qual a minimização é real e inerentemente necessária.*) Note que a função entre colchetes em (1) é de fato minimizável: uma vez que M, por hipótese, computa f, tal seqüência m de configurações existirá para todos os valores de n. halted(n) é simplesmente equal(digit(log(b – 2, n ~ 1) ~ 2, n, b), **E**(h)).

Deixaremos a definição precisa de iscomp como um exercício (Problema 4.7.4).

Segue-se que f é de fato uma função μ-recursiva, o que completa a prova do teorema. ■

Problemas para a Seção 4.7

4.7.1 Seja f: **N** → **N** uma função recursiva primitiva. Defina-se F: **N** → **N** como

$$F(n) = f(f(f(\ldots f(n) \ldots))),$$

através de n composições de função. Mostre que F é recursiva primitiva.

4.7.2 Mostre que as seguintes funções são recursivas primitivas:
(a) fatorial(n) = $n!$.
(b) gcd (m, n), o máximo divisor comum de m e n.
(c) prime(n), o predicado que é 1 se é somente n é um número primo.
(d) p(n), o n-ésimo número primo, onde p(0) = 2, p(1) = 3 e assim por diante.
(e) A função log definida no texto.

4.7.3 Seja f uma função bijetora μ-recursiva de **N** para **N**. Mostre que sua inversa, f^{-1}, é também μ-recursiva.

4.7.4 Mostre que a função iscomp, descrita na prova do Teorema 4.7.1, é recursiva primitiva.

4.7.5 Que modificações devem ser feitas na construção da prova da direção *se* do Teorema 4.7.1, na hipótese de M computar uma função f: $\mathbf{N}^k \rightarrow \mathbf{N}$ com $k > 1$?

4.7.6 Desenvolva uma representação de funções recursivas primitivas com cadeias sobre um alfabeto Σ à sua escolha (veja o próximo capítulo para tal representação de máquinas de Turing). Formalize o argumento no Exemplo 4.7.5 de que nem todas as funções computáveis podem ser recursivas primitivas.

REFERÊNCIAS

As máquinas de Turing foram originalmente concebidas por Alan M. Turing:

A. M. Turing "On computable numbers, with an application to the Entscheidungsproblem", *Proceedings, London Mathematical Society*, 2, 42 pp. 230-265, e n° 43, pp. 544-546, 1936.

Turing introduziu esse modelo a fim de argumentar que todos os conjuntos detalhados de instruções que podem ser realizados por um calculador humano também podem ser realizados por uma máquina simples adequadamente definida. Para guardar dados, a máquina de Turing original tem uma fita infinita de duas vias e um cabeçote (veja a Seção 4.5). Um modelo similar foi independentemente projetado por Post, veja:

E. L. Post "Finite Combinatory Processes. Formulation I", *Journal of Symbolic Logic*, 1, pp. 103-105, 1936.

Os seguintes livros contêm interessantes introduções às máquinas de Turing:

M. L. Minsky *Computation: Finite and Infinite Machines*, Englewood Cliffs, N. J.: Prentice-Hall, 1967.

F. C. Hennie *Introduction to Computability*, Reading, Mass.: Addison-Wesley, 1977.

As seguintes publicações são outros livros avançados sobre máquinas de Turing e conceitos correlatos introduzidos neste capítulo e nos três seguintes:

M. Davis, ed., *The Undecidable*, Hewlett, N. Y.: Raven Press, 1965. (Este livro contém muitos dos artigos originais sobre diversos aspectos do assunto, incluindo os artigos de Turing e Post citados acima.)

M. Davis, ed., *Computability and Unsolvability* New York: McGraw-Hill, 1958.

S. C. Kleene, *Introduction to Metamathematics*, Princeton, N. J.: D. Van Nostrand, 1952,

W. S. Brainerd and L. H. Landweber, *Theory of Computation*, New York: John Wiley, 1974,

M. Machtey and P. R. Young, *An Introduction to the General Theory of Algorithms*, New York: Elsevier North-Holland, 1978,

- H. Rogers, Jr., *The Theory of Recursive Functions and Effective Computability*, New York: McGraw-Hill, 1967.
- M. Sipser, *Introduction to the Theory of Computation*, Boston, Mass.: PWS Publishers, 1996.
- J. E. Hopcroft and J. D. Ullman *Introduction to Automata Theory, Languages, and Computation*, Reading, Mass.: Addison Wesley, 1979.
- C. H. Papadimitriou, *Computational Complexity*, Reading, Mass.: Addison Wesley, 1994.
- H. Hermes, *Enumerability, Decidability, Computability*, New York: Springer Verlag, 1969 (traduzido da edição alemã, 1965).

A noção e notação utilizadas na presente obra para combinar máquinas de Turing (Seção 4.3) foram influenciadas por este último livro.

Máquinas de acesso aleatório, similares em espírito à nossa "máquina de Turing de acesso aleatório" na Seção 2.4, foram estudadas em:

- S. A. Cook and R. A. Reckhow "Time-bounded random-access machines", Journal *of* Computer and Systems Sciences, 7, 4, pp. 354-375, 1973.

Funções primitivas e μ-recursivas são creditadas a Kleene:

- S. C. Kleene "General recursive functions of natural numbers", *Mathematische Annalen*, 112, pp. 727-742, 1936,e

Os algoritmos de Markov (Problema 2.6.5) são devidos a:

- A. A. Markov *Theory of Algorithms*, Trudy Math. Inst. V. A. Steklova, 1954. Tradução para o inglês: Israel Program for Scientific Translations, Jerusalem, 1961.