5 Indecidibilidade

5.1 A TESE DE CHURCH-TURING

Até aqui, neste livro, temos discutido a seguinte questão: o que pode ser computado? (E, mais intrigantemente, o que *não pode* ser computado?) Apresentamos uma diversidade de modelos matemáticos para os processos computacionais que executam tarefas concretas nessa área – em particular, as tarefas de decidir, semidecidir ou gerar linguagens e de computar funções. No capítulo anterior, vimos que as máquinas de Turing podem realizar tarefas surpreendentemente complexas consideradas nessa classificação. Vimos também que certas extensões do modelo básico da máquina de Turing, incluindo a capacidade de acesso aleatório, não conseguem ampliar o conjunto das tarefas que podem ser realizadas. Além disso, seguindo um caminho completamente diferente (ao tentar generalizar as gramáticas livres de contexto), chegamos a uma classe de geradores de linguagens que apresentam precisamente o mesmo poder que as máquinas de Turing. Por fim, tentando formalizar nossas intuições sobre quais funções numéricas podem ser consideradas computáveis, definimos uma classe de funções que revelaram ser especificamente as funções recursivas.

Tudo isso sugere termos determinado um limite superior natural para a capacidade que se pode esperar de um dispositivo computacional: essa nossa pesquisa da noção matemática definitiva e mais geral de um processo computacional, de um *algoritmo*, foi concluída com sucesso – e a máquina de Turing é uma resposta para essa pesquisa. Entretanto, vimos também, no último capítulo, que não são todas as máquinas de Turing que merecem ser classificadas como "algoritmos": argumentamos que máquinas de Turing que semidecidem linguagens, e que, portanto, rejeitam suas entradas por nunca atingirem um estado de parada, não são dispositivos computacionais úteis, enquanto as máquinas de Turing que decidem linguagens e computam funções (e, deste modo, param para quaisquer entradas fornecidas) o são. Nossa noção de algoritmo deve excluir todas as máquinas de Turing que não param em reposta a algumas entradas.

Assim, *propomos adotar a máquina de Turing que pára em respostas a todas as entradas como sendo a noção formal precisa correspondente à intuitiva idéia de um "algoritmo"*. Nada será considerado como algoritmo se não puder ser expresso na forma de uma máquina de Turing, cuja parada é garantida em resposta a todas as possíveis entradas, e todas elas serão corretamente classificadas como algoritmos. Esse princípio é conhecido como a **tese de Church-Turing**. Ela é uma tese, não um teorema, porque não é um resultado matemático: ela simplesmente afirma que um certo conceito informal (algoritmo) corresponde a um certo objeto matemático (máquina de Turing). Não sendo uma proposição matemática, a tese de Church-Turing não pode ser provada. Teoricamente é possível, entretanto, que a tese de Church-Turing possa ser contradita em alguma ocasião no futuro, caso venha a ser proposto um modelo alternativo de computação que seja plausível e razoável e que, ainda, seja comprovadamente capaz de realizar computações que não possam ser realizadas por qualquer máquina de Turing. Apesar de possível, acredita-se que isso seja pouco provável.

Adotar para os algoritmos uma noção matemática rigorosa suscita a intrigante possibilidade de se provar formalmente que certos problemas computacionais *não podem* ser resolvidos por algoritmo algum. Já sabemos o suficiente para esperar isso. No Capítulo 1, constatamos que, se cadeias de símbolos são utilizadas para representar linguagens, nem toda linguagem pode ser representada, pois enquanto existe somente uma quantidade enumeravelmente infinita de cadeias sobre um alfabeto, existe por outro lado uma quantidade infinita, mas não enumerável de linguagens. Autômatos finitos, autômatos de pilha, gramáticas livres de contexto, gramáticas irrestritas e máquinas de Turing são exemplos de objetos finitos que podem ser utilizados na especificação de linguagens e que podem ser, eles próprios, descritos com a ajuda de cadeias de símbolos (na próxima seção, desenvolveremos em detalhes um particular modo de representar máquinas de Turing por meio de cadeias). Correspondentemente, existe uma quantidade enumeravelmente infinita de linguagens recursivas e de linguagens enumeráveis recursivamente sobre qualquer alfabeto. Assim, apesar de todos os esforços para estender ao máximo a capacidade dos dispositivos computacionais, elas podem ser utilizadas para semidecidir ou decidir somente uma fração infinitesimal do conjunto de todas as possíveis linguagens.

Utilizar argumentos baseados na cardinalidade dos conjuntos de linguagens para estabelecer a limitação de nossa técnica de representação das mesmas é relativamente trivial; descobrir exemplos particulares de tarefas computacionais que não podem ser realizadas por meio de um dado modelo é muito mais interessante e compensador. Em capítulos anteriores, tivemos sucesso em descobrir certas linguagens que não são regulares ou livres de contexto; neste capítulo faremos o mesmo em relação às linguagens recursivas. Todavia, notam-se duas importantes diferenças. Primeiro, esses novos resultados adversos não são apenas fracassos temporários que poderiam ser remediados em um capítulo posterior, por meio de algum dispositivo computacional mais poderoso a ser definido: de acordo com a tese de Church-Turing, tarefas computacionais que não podem ser executadas por máquinas de Turing são impossíveis, desanimadoras, *indecidíveis*. Segundo, para provar que linguagens não são recursivas deveriam ser usados métodos diferentes dos teoremas de "bombeamento", utilizados para explorar a fragilidade da gramática livre de contexto e autômatos finitos. Em vez disso, devemos desenvolver técnicas para explorar o considerável *poder* das máquinas de Turing, a fim de evidenciar suas limitações. O aspecto da potência das máquinas de Turing, que iremos explorar, é um tipo de capacidade intrínseca que elas possuem: mostraremos que as máquinas de Turing podem receber codificações delas mesmas como entradas, e manipular essas codificações de formas convenientes. Então, perguntamos o que acontece quando uma máquina de Turing manipula uma codificação de si própria – uma aplicação engenhosa e mais simples do princípio da diagonaliza-

ção. Como codificar uma máquina de Turing de modo que ela possa ser manipulada por outra (ou *por si própria!*) é, assim, nosso próximo assunto.

5.2 MÁQUINAS DE TURING UNIVERSAIS

Qual seria a base da computação? o *hardware* ou o *software*? Você pode ter uma opinião sobre esse assunto – e também deve ter uma posição acerca de sua significância. Mas o fato é que o formalismo que apresentamos e desenvolvemos no último capítulo para a representação de algoritmos – a máquina de Turing – é uma peça de *hardware* "não-programável", especializada em resolver um particular problema, usando instruções pré-fabricadas e imutáveis.

Iremos agora tomar o rumo oposto, discutindo o fato de que as máquinas de Turing também são *softwares*, ou seja, vamos mostrar que existe uma certa máquina de Turing de uso geral que pode ser programada, mais ou menos como é o caso de um computador de propósito geral, podendo assim solucionar *qualquer* problema passível de resolução através de máquinas de Turing. O "programa" que faz essa máquina genérica comportar-se como uma máquina específica M faz o papel de uma *descrição do comportamento* de M. Em outras palavras, o formalismo das máquinas de Turing pode ser interpretado como uma *linguagem de programação*, na qual podemos escrever programas. Programas escritos nessa linguagem podem então ser *interpretados* por uma *máquina de Turing universal* – ou seja, por um outro programa escrito na mesma linguagem. A idéia de que um programa, expresso em uma dada linguagem, pode interpretar qualquer programa escrito na mesma linguagem não é nada nova – nela se fundamentam as técnicas de *"bootstrapping"* empregadas na construção de compiladores e interpretadores de linguagem de programação.[1] Para prosseguir nosso projeto, neste livro, devemos esclarecer melhor esse ponto, no contexto das máquinas de Turing.

Vamos inicialmente propor uma forma geral para a especificação das máquinas de Turing, de modo que suas descrições possam ser utilizadas naturalmente como entradas para outras máquinas de Turing. Em outras palavras, iremos desenvolver uma linguagem cujas cadeias sejam todas representações legítimas de máquinas de Turing. Pode-se desde já notar a manifestação de um problema: independentemente do tamanho do alfabeto que escolhermos para nossa representação das máquinas de Turing, haverá sempre algumas máquinas com um maior número de estados ou de símbolos de fita. Evidentemente, será necessário codificar os estados e símbolos de fita como cadeias sobre um alfabeto fixo. Adotaremos para isso a seguinte convenção: a cadeia que representa um estado da máquina de Turing deverá ser da forma $\{q\}\{0,1\}^*$; isto é, a letra q, seguida por uma cadeia binária. Analogamente, um símbolo de fita será sempre representado como uma cadeia da forma $\{a\}\{0,1\}^*$.

[1] Os implementadores de linguagens de programação muitas vezes escrevem tradutores para tais linguagens usando a mesma linguagem da programação. Mas como um tradutor pode traduzir a *si própria*? Uma possibilidade é a seguinte: escrever um tradutor usando um subconjunto simples da mesma linguagem, sem explorar os recursos mais sofisticados (e difíceis de traduzir) da mesma. Escreve-se então um tradutor para tal subconjunto da linguagem – uma tarefa ainda mais simplificada – em uma versão ainda mais reduzida da linguagem. Continua desse modo até que a linguagem escolhida seja tão simples e explícita que se pareça com uma linguagem assembly, e assim ela possa ser traduzida de forma trivial.

Seja $M = (K, \Sigma, \delta, s, H)$ uma máquina de Turing. Sejam i e j os menores inteiros, tais que $2^i \geq |K|$ e $2^j \geq |\Sigma| + 2$. Nessas condições, cada estado em K será representado como um símbolo q seguido por uma cadeia binária de comprimento i; cada símbolo do conjunto Σ será, analogamente, representado como o símbolo a seguido de uma cadeia de j *bits*. Os símbolos $\leftarrow$ e $\rightarrow$, que representam o sentido do movimento do cabeçote de leitura e gravação de fita também serão tratados como se fossem símbolos de fita (esses dois símbolos extra foram a razão do aparecimento do termo "+2" na definição de j). Convencionam-se, para as representações dos símbolos especiais $\sqcup$, $\triangleright$, $\leftarrow$ e $\rightarrow$, os quatro menores símbolos em ordem lexicográfica, respectivamente: $\sqcup$ será sempre representado como $a0^j$, $\triangleright$ como $a0^{j-1}1$, $\leftarrow$ como $a0^{j-2}10$ e $\rightarrow$ como $a0^{j-2}11$. O estado inicial da máquina será sempre representado como o primeiro estado lexicográfico, $q0^i$. Note-se o uso obrigatório de zeros iniciais nas cadeias que sucedem os símbolos a e q, para compatibilizar o comprimento total com a convenção adotada.

Devemos referenciar a representação de uma dada máquina de Turing M como "M". Esta representação consiste da tabela de transição δ, isto é, ela é uma seqüência de cadeias da forma (q, a, p, b), sendo q e p representações de estados e a, b representação de símbolos, todos esses elementos separados por vírgulas e incluídos entre parênteses. Adotamos a convenção de que as quádruplas são relacionadas em *ordem lexicográfica crescente*, começando com $\delta(s, \sqcup)$. O conjunto H de estados de parada fica implícito, mas pode ser determinado indiretamente pela não ocorrência de seus estados como primeiros componentes em qualquer quádrupla de "M". Se M decide uma linguagem e, portanto, $H = \{y, n\}$, adotaremos a convenção de que y seja, lexicograficamente, o menor dos dois estados de parada.

Qualquer máquina de Turing pode ser representada segundo essa convenção. O mesmo método será utilizado para representar quaisquer cadeias sobre o alfabeto da máquina de Turing. Qualquer cadeia $w \in \Sigma^*$ terá uma única representação, também referenciada como "w", e obtida pela justaposição das representações dos símbolos que a compõem.

Exemplo 5.2.1: Considere a máquina de Turing $M = (K, \Sigma, \delta, s, \{h\})$, onde $K = \{s, q, h\}$, $\Sigma = \{\sqcup, \triangleright, a\}$ e δ é dado pela tabela seguinte.

estado	símbolo	δ
s	a	$(q, \sqcup)$
s	$\sqcup$	$(h, \sqcup)$
s	$\triangleright$	$(s, \rightarrow)$
q	a	(s, a)
q	$\sqcup$	$(s, \rightarrow)$
q	$\triangleright$	$(q, \rightarrow)$

Como existem três estados em K e três símbolos em Σ, temos $i = 2$ e $j = 3$. Estes são os menores inteiros, tais que $2^i \geq 3$ e $2^j \geq 3 + 2$. Os estados e símbolos são representados como segue:

estado/símbolo	representação
s	$q00$
q	$q01$
h	$q11$
$\sqcup$	$a000$
.	$a001$
$\leftarrow$	$a010$
$\rightarrow$	$a011$
a	$a100$

Portanto, a representação da cadeia $\triangleright aa \sqcup a$ será:

"$\triangleright aa \sqcup a$" = $a001a100a100a000a100$.

A representação "M" da máquina de Turing M será a seguinte cadeia:

"M" = ($q00$, $a100$, $q01$, $a000$), ($q00$, $a000$, $q11$, $a000$),
($q00$, $a001$, $q00$, $a011$), ($q01$, $a100$, $q00$, $a011$),
($q01$, $a000$, $q00$, $a011$), ($q01$, $a001$, $q01$, 011).✧

Agora estamos prontos para discutir uma *máquina de Turing universal* U, que utiliza como programas as codificações de outras máquinas, para direcionar sua operação. Intuitivamente, U deve apresentar dois argumentos: uma descrição "M" de certa máquina M, e a descrição "w" de uma dada cadeia de entrada w,. Desejamos que U apresente a seguinte propriedade: U pára em resposta à entrada "M" "w" se e somente se M parar em resposta à entrada w. Para utilizar a notação funcional previamente estudada para as máquinas de Turing, escreveremos:

$$U(\text{"}M\text{"}\ \text{"}w\text{"}) = \text{"}M(w)\text{"}.$$

Na realidade, descrevemos não uma máquina U de uma única fita, mas uma máquina U' de 3 fitas, intimamente relacionada a U (assim, U será a máquina de Turing de uma única fita que simula U'). Especificamente, U' utiliza suas três fitas como segue: a primeira contém a codificação do conteúdo da fita de M; a segunda contém a codificação da própria máquina M; e a terceira fita contém a codificação do estado de M no ponto corrente da computação simulada.

A máquina U' é iniciada com alguma cadeia "M" "w" em sua primeira fita, estando as outras duas fitas preenchidas com espaços em branco. (É irrelevante como U' se comporta, se sua cadeia de entrada não estiver apresentada nessa forma.) Antes de mais nada, U' move "M" para a segunda fita e desloca "w" para a esquerda, ou seja, para a extremidade esquerda da primeira fita, precedendo-a com "$\triangleright\sqcup$". A essa altura, portando, a primeira fita contém "$\triangleright\sqcup w$". U' grava então na terceira fita a codificação do estado inicial s de M, que é sempre $q0^i$ (U' pode facilmente determinar i e j, examinando "M"). Agora U' passa a simular os passos da computação de M. Entre esses passos simulados, U' manterá os cabeçotes da segunda e da terceira fitas em suas extremidades esquerdas, e o cabeçote da primeira fita posiciona-se sobre o símbolo a associado à versão codificada do símbolo que M estaria lendo na ocasião.

U' simula então um passo da operação de M, como segue: percorre sua segunda fita até encontrar uma quádrupla cujo primeiro componente corresponda ao estado codificado gravado em sua terceira fita, e cujo segundo componente corresponda ao símbolo codificado apontado na primeira fita. Encontrando essa quádrupla, U' altera o estado corrente substituindo-o pelo terceiro componente dessa quádrupla e realiza, na primeira fita, a ação especificada pelo quarto componente. Se o quarto componente codifica um símbolo do alfabeto da fita de M, esse símbolo é gravado na primeira fita, substituindo aquele que estiver correntemente sendo apontado. Se o quarto componente for $a0^{j-2}10$, (ou seja, a codificação de $\leftarrow$), então U' move o primeiro cabeçote para o primeiro símbolo a à sua esquerda e, se for a codificação de $\rightarrow$, para a sua direita. Se um $\sqcup$ for encontrado, U' deve converte-lo para $a0^j$, o código associado a um espaço em branco de M.

Se em algum passo a combinação de estado-símbolo não for encontrada na segunda fita, isso significa que o estado corrente é um estado de parada. U' também pára em um estado de parada conveniente. Isso completa nossa descrição da operação de U'.

Problemas para a Seção 5.2

5.2.1 Lembre-se da máquina de Turing M no Exemplo 4.1.1. Para esta máquina:
(a) O que é a cadeia "M"?
(b) Qual seria a representação da cadeia aaa?
(c) Suponha que a máquina de Turing universal (de 3 fitas) U', descrita neste capítulo, simule a operação de M para a entrada aaa. Quais são os conteúdos das fitas de U' no começo da simulação? E no começo da simulação do terceiro passo da operação de M?

5.2.2 Por analogia com a máquina de Turing universal, podemos desejar projetar um *autômato finito universal* U, que aceite a linguagem {"M" "w": $w \in L(M)$}. Explique por que autômatos finitos universais não podem existir.

5.3 O PROBLEMA DA PARADA

Suponha que você escrevesse um programa, na sua linguagem de programação favorita, que realize o seguinte feito notável: tomando como entrada qualquer programa P, denotado na mesma linguagem, e uma entrada X desse programa, e utilizando alguma técnica engenhosa, esse programa sempre determine corretamente se o programa P irá parar em resposta à entrada X, retornando "sim" se isso acontecer, ou se ele irá permanecer eternamente em execução (nesse caso retorna "não"). Vamos dar a esse programa o nome de halts(P, X).

Este seria um programa de valor inestimável. Ele descobriria automaticamente todos os tipos de erros que levam outros programas a permanecer eternamente em execução em resposta a determinadas entradas. Utilizando esse programa, você poderia conduzir com êxito muitos projetos notáveis. Por exemplo, você poderia utilizá-lo para escrever outro programa, com o

nome de diagonal(X) (lembre-se da prova por diagonalização de que 2^N não é enumerável, na Seção 1.5):

```
    diagonal(X)
a:  se halts(X, X), então vá para a, senão pare
```

Observe-se como opera o programa diagonal(X): caso o programa halts decidir que o programa *X* parará caso lhe seja apresentado como entrada a codificação do próprio programa *X*, então diagonal(X) iniciará uma execução infinita, caso contrário, ele deverá parar.

E agora coloca-se uma questão inexplicável: a função diagonal(diagonal) *pára?* Ela pára se e somente se a chamada halts(diagonal, diagonal) retornar "não"; em outras palavras, *pára se e somente se ela não parar*. Isso é uma contradição: devemos concluir que a única hipótese, com que iniciamos esse raciocínio, é falsa, ou seja, que o programa halts(*P*, *X*) não existe. Isto é, *não pode haver programa ou algoritmo* para resolver o problema que o programa halts resolveria: decidir se programas arbitrários irão parar ou entrar em execução infinita.

Esse tipo de argumento deve ser familiar ao leitor não apenas pela sua experiência nas ciências da computação, mas também pela sua cultura geral do século XX. Dispomos agora de um arsenal de formalismo e métodos suficientes para com ele apresentarmos uma versão formal, matematicamente rigorosa, desse paradoxo. Temos uma notação completa para algoritmos, na forma de uma "linguagem de programação": a máquina de Turing. De fato, na última seção foi apresentada uma nova característica, de que precisamos: desenvolvemos uma estrutura que permite que nossos "programas" manipulem outros programas e suas entradas – exatamente como o nosso o fictício halts(*P*, *X*) faria. Já estamos prontos, portanto, para definir uma linguagem que não seja recursiva, e *provar* que ela não o é. Seja

H = {"M" "w": a máquina de Turing M pára em resposta à cadeia de entrada w}.

Note-se inicialmente que H é recursivamente enumerável: ela é precisamente a linguagem semidecidida por nossa máquina de Turing universal U na seção anterior. De fato, em resposta à entrada "M" "w", U pára precisamente quando essa entrada pertence ao conjunto H.

Além disso, sendo H recursiva, então *cada* linguagem recursivamente enumerável é recursiva. Em outras palavras, H tem a chave para a pergunta que fizemos na Seção 4.2, se todas as linguagens recursivamente enumeráveis são também Turing decidíveis: a resposta será afirmativa se e somente se H for recursiva. Para tanto, supondo que H seja, de fato, decidida por alguma máquina de Turing M_0, então, dada qualquer particular máquina de Turing M que semidecida uma linguagem $L(M)$, poderemos projetar uma máquina de Turing M' que *decida* $L(M)$ como segue: inicialmente, M' transforma sua fita de entrada de ▷ ␣ w ␣ para ▷ "M" "w" ␣ e, então, simula M_0 sobre essa entrada. Por hipótese, M_0 decidirá corretamente se M aceita w *ou não*. Antecipando as questões tratadas no Capítulo 7, podemos dizer que existirão *reduções* de todas as linguagens recursivamente enumeráveis para H, e, portanto, H é *completo* para toda a classe de linguagens recursivamente enumeráveis.

Todavia, podemos mostrar, formalizando o argumento acima apresentado para a função halts(*P*, *X*), que H *não* é recursiva. Se H fosse recursiva, então

H_1 = {"M": a máquina de Turing M pára em resposta à cadeia de entrada "M"}

também seria recursiva. (H_1 corresponde à parte halts(*X*, *X*) do programa diagonal.) Se existisse uma máquina de Turing M_0 que pudesse decidir H, então uma máquina de Turing M_1 que decidisse H_1 necessitaria apenas transformar sua cadeia de entrada ▷ ␣ "M" ␣ em ▷ ␣ "M" "M" ␣ e, então, passar o controle para M_0. Isso é suficiente para mostrar que H_1 não é recursiva.

Adicionalmente, se H_1 fosse recursiva, então seu complemento também seria uma linguagem recursiva:

$\overline{H_1}$ = {w: ou w não é uma codificação legítima de uma máquina de Turing ou então w é a codificação "M" de uma máquina de Turing M que não pára em resposta à entrada "M"}.

Isso acontece porque a classe de linguagens recursivas é fechada em relação à operação de complementação (Teorema 4.2.2). Incidentalmente, $\overline{H_1}$ é a linguagem *diagonal*, análoga ao nosso programa diagonal e a última parte da prova.

Mas $\overline{H_1}$ não pode nem mesmo ser recursivamente enumerável – quanto menos recursiva. Seja M^* uma máquina de Turing que semidecide $\overline{H_1}$. Pertenceria "M^*" a $\overline{H_1}$? Por definição de $\overline{H_1}$, "M^*" $\in \overline{H_1}$ se e somente se M^* não aceitar a cadeia de entrada "M^*". Mas M^* por hipótese semidecide $\overline{H_1}$. Assim, "M^*" $\in \overline{H_1}$ se e somente se M^* aceitar "M^*". Concluímos assim que M^* aceita "M^*" se e somente se M^* não aceitar "M^*". Isso é absurdo, o que leva à conclução de que a existência de M_0 foi uma premissa incorreta.

Resumindo o desenvolvimento desta seção, queríamos descobrir se toda linguagem recursivamente enumerável é recursiva. Observamos que isso seria verdadeiro se e somente se a particular linguagem recursivamente enumerável H fosse recursiva. De H derivamos, em dois passos, a linguagem $\overline{H_1}$, a qual deverá ser recursiva para que H possa também ser recursiva. Mas a hipótese de que $\overline{H_1}$ é recursiva conduz a uma contradição lógica, por diagonalização. Provamos, desse modo, o seguinte importante teorema:

Teorema 5.3.1: *A linguagem H não é recursiva. Portanto, a classe das linguagens recursivas é um subconjunto estrito da classe das linguagens recursivamente enumeráveis.*

Mencionamos anteriormente que esse argumento é uma instância do princípio de diagonalização utilizado na Seção 1.5 para mostrar que 2^N não é enumerável. Para justificar e ressaltar novamente a essência da prova, vamos definir uma relação binária R em cadeias sobre o alfabeto utilizado na codificação de máquinas de Turing: (u, w) $\in R$ se e somente se u = "M" para alguma máquina de Turing M que aceita w. (R é uma versão de H.) Seja agora, para cada cadeia u,

$$R_u = \{w: (u, w) \in R\}$$

(os R_u's correspondem às linguagens recursivamente enumeráveis). Considere a *diagonal* de R, isto é,

$$D = \{w: (w, w) \notin R\}$$

(D corresponte a $\overline{H_1}$). Pelo princípio da diagonalização, $D \neq R_u$ para todo u, isto é, $\overline{H_1}$ é uma linguagem que difere de qualquer linguagem recursivamente enumerável.

Justifica-se que $D \neq R_u$ para qualquer u porque D difere, por construção, de cada R_u (e, portanto, de cada linguagem recursivamente enumerável) por, no mínimo, uma cadeia – a saber, u.

O Teorema 5.3.1 responde negativamente à primeira das duas questões que apresentamos ao final da Seção 4.2 ("cada linguagem recursivamente enumerável também é recursiva?" e "a classe de linguagens recursivamente enumeráveis é fechada em relação à operação de complementação?"). Mas a mesma prova fornece resposta para a outra questão. É fácil ver que H_1, assim como H, é recursivamente enumerável, e mostramos que $\overline{H_1}$ não é recursivamente enumerável. Portanto, ficou também provado o seguinte resultado:

Teorema 5.3.2: *A classe das linguagens recursivamente enumeráveis não é fechada em relação à operação de complementação.*

Problemas para a Seção 5.3

5.3.1 Podemos descobrir um exemplo particular de função não-recursiva sem utilizar um argumento baseado na diagonalização. A **função busy beaver** β: $\mathbf{N} \rightarrow \mathbf{N}$ é definida como segue: para cada inteiro n, $\beta(n)$ é o maior número m tal que haja uma máquina de Turing, com alfabeto $\{\triangleright, \sqcup, a, b\}$, e com exatamente n estados que, quando iniciada com espaços em branco na fita, acaba parando na configuração $(h, \triangleright\sqcup a^m)$.

(a) Mostre que, se f é uma função recursiva qualquer, então há um inteiro k_f, tal que $\beta(n + k_f) \geq f(n)$. (k_f é o número de estados na máquina de Turing M_f, que, quando iniciada com a entrada a^n, pára com $a^{f(n)}$ em sua fita.)

(b) Mostre que β não é recursiva. (Suponha que β seja recursiva; nesse caso, $f(n) = \beta(2n)$ também o seria. Aplique o resultado em (a) acima.)

5.3.2 Sabemos que a classe das linguagens recursivamente enumeráveis não é fechada em relação à operação de complementação. Mostre que, por outro lado, ela é fechada em relação às operações de união e intersecção.

5.3.3 Mostre que a classe de linguagens recursivas é fechada em relação às operações de união, complementação, intersecção, concatenação e estrela de Kleene.

5.4 PROBLEMAS INDECIDÍVEIS SOBRE MÁQUINAS DE TURING

Vamos retroceder um pouco e interpretar no nível intuitivo esse resultado que acabamos de provar, à luz da tese de Church-Turing. Como a prova estabelece que H não é recursiva e como aceitamos o princípio de que qualquer algoritmo pode ser convertido em uma máquina de Turing que pára em resposta a todas as suas entradas, devemos concluir que *não há algoritmo que decida, para uma dada máquina de Turing arbitrária M e uma cadeia de entrada w, se M aceita w ou não.* Os problemas para os quais não existem algoritmos são ditos **indecidíveis** ou **insolúveis**, e veremos muitos deles neste capítulo. O problema indecidível mais famoso e fundamental é o de responder se uma dada máquina de Turing pára ou não em resposta a uma dada entrada. Sua indecidibilidade foi o que acabamos de demonstrar. Esse problema é geralmente conhecido como o **problema da parada da máquina de Turing.**

Note-se que a indecidibilidade do problema de parada não implica de modo algum que não possa haver algumas circunstâncias em que seja possível prever se uma máquina de Turing irá parar em resposta a uma cadeia de entrada. No Exemplo 4.1.2, fomos capazes de concluir que uma certa máquina simples está limitada a nunca parar em resposta a uma determinada entrada. Alternativamente, podemos examinar a tabela de transições da máquina de Turing, por exemplo, para verificar se ela apresenta algum estado de parada; em caso negativo, tal máquina não poderá parar em resposta a qualquer cadeia de entrada. Essa e outras análises mais complexas podem dar algum resultado, o que pode ser útil em certos casos; mas nosso teorema afirma que qualquer análise desse tipo deve, em última instância, ser inconclusiva ou produzir um resultado incorreto: não há um método completamente geral que decida corretamente todos os casos.

Uma vez que constatamos, por diagonalização, que os problemas de parada são indecidíveis, decorre a indecidibilidade de uma grande variedade de problemas. Esses resultados são provados não mais por diagonalização, mas por *reduções*: mostramos, em cada caso, que se alguma linguagem L_2 for recursiva, então também o será alguma linguagem L_1, já conhecida como sendo não-recursiva, o que é uma contradição.

Definição 5.4.1: Sejam L_1, $L_2 \subseteq \Sigma^*$ duas linguagens. Uma **redução de L_1 para L_2** é uma função recursiva τ: $\Sigma^* \rightarrow \Sigma^*$, tal que $x \in L_1$ se e somente se $\tau(x) \in L_2$.

O leitor deve interpretar cuidadosamente o sentido da aplicação da redução. Para mostrar que uma linguagem L_2 não é recursiva, devemos identificar uma linguagem L_1 sabidamente não-recursiva e, então, reduzir L_1 a L_2. Reduzir L_2 a L_1 seria inócuo, pois simplesmente mostraria que L_2 só poderá ser decidido, se pudermos decidir L_1 – o que sabemos ser impossível.

Formalmente, o uso correto de reduções em provas de indecidibilidade é o seguinte:

Teorema 5.4.1: *Se L_1 é uma linguagem não-recursiva, e se houver uma redução de L_1 para L_2, então L_2 também não é recursiva.*

Prova: Seja L_2 uma linguagem recursiva. Seja M_2 uma máquina de Turing que decida L_2, e T, uma máquina de Turing que computa a redução τ. Nessas condições a máquina de Turing TM_2 deveria decidir L_1. Mas L_1 é indecidível, o que é uma contradição. ■

A seguir, empregaremos as reduções para provar a indecidibilidade de vários problemas sobre máquinas de Turing.

Teorema 5.4.2: *São indecidíveis os seguintes problemas acerca das máquinas de Turing:*

(a) Dada uma máquina de Turing M e uma cadeia de entrada w, M pára em resposta à entrada w?
(b) Dada uma máquina de Turing M, M pára em resposta a uma entrada vazia?
(c) Dada uma máquina de Turing M, há alguma cadeia em resposta à qual a máquina M pára?
(d) Dada uma máquina de Turing M, M pára em resposta a qualquer cadeia de entrada?
(e) Dadas duas máquinas de Turing M_1 e M_2, elas param em resposta às mesmas cadeia de entrada?
(f) Dada uma máquina de Turing M, a linguagem que M semidecide é regular? É livre de contexto? É recursiva?
(g) Existe alguma máquina fixa M, para a qual o seguinte problema é indecidível: dado w, M pára em resposta à entrada w?

Prova: A parte (a) foi provada na seção anterior.
(b) Propomos uma redução de H para a linguagem

$$L = \{\text{"}M\text{"}: M \text{ pára em resposta à cadeia vazia } \varepsilon\}.$$

Dada a descrição "M" de uma máquina de Turing M e uma entrada x, nossa redução simplesmente constrói a descrição de uma máquina de Turing M_w que opera da seguinte forma: quando acionada em sua fita de entrada vazia (isto é, na configuração $(s, \triangleright\sqcup)$), M_w grava w em sua fita e, então, inicia a simulação de M. Em outras palavras, se $w = a_1 \ldots a_n$, então M_w é simplesmente a máquina

$$Ra_1Ra_2R \ldots Ra_nL_{\sqcup}M.$$

É fácil constatar que a função τ, que mapeia "M" "w" em "M_w"; é, de fato, recursiva.

(c) Podemos reduzir a linguagem L, que provamos ser não-recursiva na Parte (b), para a linguagem $L' = \{\text{"}M\text{"}: M \text{ pára em resposta a alguma entrada}\}$, da seguinte forma: dada a representação "M" de uma máquina de Turing qualquer M, essa redução constrói a representação "M'" de uma máquina de Turing M' que apaga qualquer entrada que lhe seja fornecida e então simula M com a cadeia de entrada vazia. Claramente, M' pára em resposta a alguma cadeia se e somente se M' parar em resposta a todas as cadeias e se ainda, M parar em resposta à cadeia vazia.

(d) O argumento utilizado para provar a Parte (c) também se aplica neste caso, uma vez que M' foi construído para aceitar alguma entrada se e somente se aceitar cada possível entrada.

(e) Podemos reduzir ao presente problema o anterior (da Parte (d)). Dada a descrição "M" de uma máquina M, nossa redução constrói a cadeia

$$\tau(\text{"}M\text{"}) = \text{"}M\text{" "}y\text{"},$$

onde "y" é a descrição de uma máquina que aceita diretamente qualquer entrada. Claramente, as duas máquinas, M e y, aceitarão as mesmas entradas se e somente se M aceitar todas as entradas.

(f) Reduzimos o problema da Parte (b) acima ao presente problema. Mostramos como modificar qualquer máquina de Turing M para obtermos uma máquina de Turing M', tal que M' pára em resposta às cadeias em H, ou a nenhuma cadeia, dependendo de M parar ou não em resposta à cadeia vazia. Uma vez que não há algoritmo que responda se M pára em resposta à cadeia vazia, não pode haver um que responda se $L(M)$ é $\varnothing$ (que é L regular, livre de contexto e recursiva) ou H (que não é nenhuma das três). Inicialmente, M' guarda uma cópia de sua cadeia de entrada e inicia a simulação do comportamento de M em resposta à cadeia vazia. Quando (e se) M parar, M' restaura sua entrada e realiza sobre ela a operação da máquina de Turing universal U. Portanto, ou M' não pára em qualquer entrada, porque nunca termina de imitar M em resposta à entrada ε, ou pára precisamente em resposta às cadeias pertencentes ao conjunto H.

(g) A máquina fixa M_0, a que aludimos no enunciado do teorema, é precisamente a máquina de Turing universal U. ■

Problemas para a Seção 5.4

5.4.1 Dizemos que uma máquina de Turing M **utiliza** k células da fita para a cadeia de entrada w se e somente se houver uma configuração (q, ugv) da máquina M tal que $(s, \triangleright\sqcup w) \vdash_M^* (q, ugv)$ e $|uav| \geq k$.

(a) Mostre que o seguinte problema é solúvel: dada uma máquina de Turing M, uma cadeia de entrada w e um número k, M utiliza k células de fita para a cadeia de entrada w?
(b) Seja $f: \mathbf{N} \to \mathbf{N}$ uma função recursiva. Mostre que o seguinte problema é solúvel: dada uma máquina de Turing M e uma cadeia de entrada w, M utiliza $f(|w|)$ células de fita para a cadeia de entrada w?
(c) Mostre que o seguinte problema é indecidível: dada uma máquina de Turing M e uma cadeia de entrada w, existe um $k \geq 0$ tal que M não utiliza k células de fita para a cadeia de entrada w? (ou seja, M utiliza uma quantidade finita de fita para a cadeia de entrada w?)

5.4.2 Quais dos seguintes problemas sobre máquinas de Turing são solúveis e quais são indecidíveis? Explique cuidadosamente suas respostas.

(a) Determinar, dada uma máquina de Turing M, um estado q e uma cadeia w, se M sempre alcança o estado q quando acionada com a entrada w a partir do seu estado inicial.
(b) Determinar, dada uma máquina de Turing M e dois estados p e q, se existe alguma configuração com o estado p, que produza uma configuração com estado q, sendo p um estado particular de M.
(c) Determinar, dada uma máquina de Turing M e um estado q, se existe alguma configuração que produza uma configuração com o estado q.
(d) Determinar, dada uma máquina de Turing M e um símbolo a, se M sempre grava o símbolo a quando iniciada com sua fita de entrada vazia.

(e) Determinar, dada uma máquina de Turing M, se M sempre grava um símbolo não-branco quando iniciada com sua fita de entrada vazia.
(f) Determinar, dada uma máquina de Turing M e uma cadeia w, se M sempre move seu cabeçote para a esquerda quando iniciada com a fita de entrada w.
(g) Determinar, dadas duas máquinas de Turing, se uma delas semidecide o complemento da linguagem semidecidida pela outra.
(h) Determinar, dada duas máquinas de Turing, se existe alguma cadeia em resposta à qual ambas param.
(i) Determinar, dada uma máquina de Turing M, se a linguagem semidecidida por M é finita.

5.4.3 Mostre que é um problema indecidível determinar, dada uma máquina de Turing M, se existe ou não alguma cadeia w tal que M percorre cada um de seus estados durante o processamento dessa sua entrada w.

5.4.4 Mostre que o problema de parada para máquinas de Turing permanece indecidível mesmo quando restrito a máquinas de Turing com algum número fixo e pequeno de estados. (Quando exige-se que o número seja fixo mas não pequeno, a existência de uma máquina de Turing universal já estabelece o resultado. Mostre como qualquer máquina de Turing pode ser simulada, de alguma maneira, por uma máquina de Turing com cerca de meia dúzia de estados, mas um alfabeto muito maior. Na realidade, três estados são suficientes. Mais ainda, dois bastariam, se permitíssemos que nossas máquinas gravassem na fita e se movessem em um único passo.)

5.4.5 Mostre que qualquer máquina de Turing pode ser simulada por um autômato sem nenhuma fita mas com dois **contadores**. Um contador é um autômato de pilha com um único símbolo, e mais um marcador diferenciado de final, que nunca é removido. Portanto, um contador pode ser considerado como um registro destinado a conter um número em unário. As possíveis operações sobre contadores são as seguintes: adicionar 1; verificar se ele contém 0, e em caso contrário subtrair 1. Conclua que o problema da parada para essas máquinas com 2 contadores é indecidível. (Sugestão: comece mostrando como simular uma fita de máquina de Turing usando duas pilhas com dois símbolos; mostre que estas pilhas podem ser simuladas por *quatro* contadores, codificando em unário o conteúdo da pilha; por fim, simule quatro contadores com dois, codificando quatro números a, b, c, d como $2^a3^b5^c7^d$.)

5.5 PROBLEMAS INSOLÚVEIS ENVOLVENDO GRAMÁTICAS

Problemas insolúveis não ocorrem somente no domínio das máquinas de Turing, mas em virtualmente todos os campos da matemática. Por exemplo, há vários problemas indecidíveis relacionados às gramáticas, resumidos abaixo.

Teorema 5.5.1: *Cada um dos seguintes problemas é indecidível:*
(a) Para uma dada gramática G e uma cadeia w, determinar se $w \in L(G)$.
(b) Para uma dada gramática G, determinar se $\varepsilon \in L(G)$.
(c) Para duas gramáticas G_1 e G_2 dadas, determinar se $L(G_1) = L(G_2)$.
(d) Para uma gramática arbitrária G, determinar se $L(G) = \emptyset$.
(e) Além disso, existe uma certa gramática fixa G_0, para a qual é indecidível determinar se qualquer cadeia w está em $L(G_0)$.

Prova: Apresentando uma redução do problema de parada para (a), reduções muito similares podem ser aplicadas para provar os itens restantes. Dada qualquer máquina de Turing M e uma cadeia w, simplesmente aplicamos a M o raciocínio aplicado à prova do Teorema 4.6.1, para produzir uma gramática G, tal que $L(G)$ seja a linguagem semidecidida por M. Segue-se que $w \in L(G)$ se, e somente se, M parar em resposta à entrada w. ■

Sabendo que as gramáticas têm o mesmo poder que as máquinas de Turing (Teorema 4.6.1), a indecidibilidade que acabamos de constatar certamente não surpreende. No entanto, o mais intrigante é que questões similares a essas, aplicadas ao domínio das *gramáticas livres de contexto* e sistemas similares – um escopo muito mais simples e limitado – são também indecidíveis. Naturalmente, não se incluem, neste caso, os problemas que determinam se $w \in L(G)$ ou se $L(G) = \emptyset$ – estes podem ser resolvidos por algoritmos eficientes (ver Teorema 3.6.1). Vários outros problemas, porém, são insolúveis.

Teorema 5.5.2: *Cada um dos seguintes problemas é indecidível:*
(a) Dada uma gramática livre de contexto G, é $L(G) = \Sigma^*$?
(b) Dadas duas gramáticas livres de contexto G_1 e G_2, é $L(G_1) = L(G_2)$?
(c) Dados dois autômatos de pilha M_1 e M_2, eles aceitam precisamente a mesma linguagem?
(d) Dado um autômato de pilha M, determinar um autômato de pilha equivalente com o mínimo número de estados.

Prova: (a) A principal dificuldade técnica desde teorema está em provar a Parte (a); as demais são relativamente simples. Devemos reduzir à Parte (a) ao problema indecidível que analizamos na Parte (d) do Teorema 5.5.1, ou seja, o problema de decidir se uma dada gramática irrestrita gera, todas as possíveis cadeias sobre seu alfabeto.

Seja $G_1 = (V_1, \Sigma_1, R_1, S_1)$ uma gramática irrestrita. Inicialmente modificamos essa gramática da seguinte forma: sendo as regras de G_1 da forma $\alpha_i \rightarrow \beta_i$, para $i = 1, \ldots, |R_1|$, adicionamos, a V_1, $|R_1|$ novos símbolos não-terminais A_i, $i = 1, \ldots, |R_1|$, um para cada regra de R_1 e substituímos a i-ésima regra ($i = 1, \ldots, |R_1|$), pelas duas regras $\alpha_i \rightarrow A_i$ e $A_i \rightarrow \beta_i$, obtendo a gramática $G_1' = (V_1', \Sigma_1, R_1', S_1)$. É claro que $L(G_1') = L(G_1)$; qualquer derivação de uma cadeia por G_1 pode ser convertida em uma **derivação padrão** da mesma cadeia em G_1', na qual cada passo ímpar de derivação aplica uma regra da forma $u_i \rightarrow A_i$, enquanto o passo par subseqüente aplica uma regra da forma $A_i \rightarrow v_i$.

A partir de G_1' construímos uma gramática G_2 *livre de contexto* sobre o alfabeto Σ, tal que $L(G_2) = \Sigma^*$ se e somente *se* $L(G_1) = \emptyset$. Essa redução prova a Parte (a).

Seja agora uma derivação de uma cadeia de terminais pela gramática G_1'; pela observação acima, sabemos que ela é uma derivação padrão, a saber:

$$S_1 \Rightarrow x_1 \Rightarrow x_2 \Rightarrow x_3 \Rightarrow \ldots \Rightarrow x_n,$$

onde $x_i \in V_1'^*$ para todo i, $x_n \in \Sigma_1'^*$, n é par, e cada x_i com i ímpar contém exatamente um não-terminal A_j, enquanto cada x_i com i par não contém tal não-terminal. Essa derivação pode ser representada como uma cadeia sobre o alfabeto $\Sigma = V \cup \{\Rightarrow\}$, dada precisamente pela cadeia acima. De fato, por motivos que logo se tornarão claros, interessa-nos mais a **versão boustrophedon**[†] da cadeia que corresponde à derivação padrão, na qual os x_i's com índices ímpares são revertidos:

$$S_1 \Rightarrow x_1^R \Rightarrow x_2 \Rightarrow x_3^R \Rightarrow \ldots \Rightarrow x_{n-1}^R \Rightarrow x_n.$$

Consideremos agora a linguagem $D_{G_1'} \subseteq \Sigma^*$, formada por todas essas versões boustrophedon de derivações padrão em G_1' de cadeias de terminais. É claro que $D_{G_1'} = \emptyset$ se e somente se $L(G_1) = \emptyset$. Colocado em termos do complemento, $\overline{D_{G_1'}} = \Sigma^* - D_{G_1'}$.

$$\overline{D_{G_1'}} = \Sigma^* \text{ se e somente se } L(G_1) = \emptyset.$$

Portanto, tudo que temos de mostrar a fim de concluir a prova é que $\overline{D_{G_1'}}$ *é livre de contexto.*

Para tanto, vamos observar melhor a linguagem $\overline{D_{G_1'}}$. O que ela considera como cadeia w dessa linguagem? Em outras palavras, em quais situações uma cadeia w *falha* em ser uma versão boustrophedon de uma derivação padrão em G_1' de uma cadeia de terminais? Isso acontece se e somente se no mínimo uma das seguintes condições for verdadeira:

(1) w não se inicia com $S_1 \Rightarrow$.
(2) w não termina com $\Rightarrow v$, onde $v \in \Sigma_1'^*$.
(3) w contém um número ímpar de $\Rightarrow$'s.
(4) w está na forma $u \Rightarrow y \Rightarrow v$ ou $u \Rightarrow y$, onde
  (a) u contém um número par de ocorrências de $\Rightarrow$,
  (b) y contém exatamente uma ocorrência de $\Rightarrow$, e
  (c) y não está na forma $y = y_1 A_i y_2 \Rightarrow y_2^R \beta_i y_1^R$ para algum $i \leq |R_1|$ e $y_1, y_2 \in \Sigma_1'^*$, onde β_i é o lado direito da i-ésima regra de G_1.
(5) w está na forma $u \Rightarrow y \Rightarrow v$, onde
  (a) u contém um número ímpar de ocorrências de $\Rightarrow$,
  (b) y contém exatamente uma ocorrência de $\Rightarrow$, e
  (c) y não está na forma $y = y_1 \alpha_i y_2 \Rightarrow y_2^R A_i y_1^R$ para algum $i \leq |R_1|$ e $y_1, y_2 \in \Sigma_1'^*$, onde α_i é o lado esquerdo da i-ésima regra de G_1.

[†] *Boustrophedon*, do grego, designa um modo de escrever alternando as linhas em sentidos opostos, da esquerda para a direita e da direita para a esquerda.

Nota-se que w pertence a $\overline{D_{G_1'}}$ se e somente se w satisfizer qualquer uma dessas cinco condições, isto é, $\overline{D_{G_1'}}$ é a união das cinco linguagens L_1, L_2, L_3, L_4 e L_5 descritas nos itens (1) a (5) acima. *Postulamos que todas as cinco linguagens são livres de contexto* e, de fato, podemos construir gramáticas livres de contexto que as geram. Para as primeiras três, que são linguagens regulares, isso é trivial.

Para L_4 e L_5, nosso argumento é indireto: devemos discutir que podemos projetar um *autômato não-determinístico de pilha* para cada uma e aplicar a parte (b) do Teorema 3.6.1, que afirma existir um algoritmo para construir, a partir de qualquer autômato de pilha M, uma gramática livre de contexto G tal que $L(G) = L(M)$.

O autômato de pilha M_4 que aceita L_4 opera em duas fases. Na primeira, varre a entrada w da esquerda para a direita, sempre registrando se visitou um número ímpar ou um número par de $\Rightarrow$'s (isso pode ser feito alternando o autômato entre dois estados). Quando um $\Rightarrow$ de ordem par é encontrado, M_4 tem duas opções e escolhe entre elas, não-deterministicamente: continuar contando, em módulo 2, os símbolos $\Rightarrow$ ou entrar na segunda fase.

Na segunda fase de sua operação, M_4 armazena a cadeia de entrada em sua pilha até que um $\Rightarrow$ seja encontrado. Se nenhum símbolo A_i tiver sido empilhado, ou mais de um, então M_4 aceitará a entrada pois ela pertence a L_4. Caso contrário, M_4 compara o conteúdo da sua pilha (uma cadeia da forma $y_2^R A_i y_1^R$ quando lida do topo para o fundo da pilha) com a parte da entrada compreendida até o próximo símbolo $\Rightarrow$. Se uma não-correspondência for detectada antes de A_i ser encontrado, ou caso A_i não seja substituído, na entrada, por β_i^R (a cadeia registrada por M_4 em seu espaço de estados), ou se uma não-correspondência for encontrada após esse ponto, ou ainda se o próximo $\Rightarrow$ (ou o final da entrada) não for encontrado imediatamente após o esvaziamento da pilha, então M_4 aceita a cadeia. Caso contrário, rejeita-a.

Isso conclui a descrição de M_4. Obviamente, $L(M_4) = L_4$. A construção de L_5 é muito similar. Pudemos assim obter uma gramática livre de contexto que gera cada uma das linguagens L_1 a L_5; então, podemos projetar a gramática livre de contexto G_2 que realiza a sua união.

Portanto, dada qualquer gramática irrestrita G_1, podemos construir uma gramática livre de contexto G_2, tal que $L(G_2) = \overline{D_{G_1'}}$. Mas sabemos que $\overline{D_{G_1'}} = \Sigma^*$ se, e somente se, $L(G_1) = \emptyset$. Concluímos que se tínhamos um algoritmo capaz de decidir se qualquer dada gramática livre de contexto gera ou não o conjunto Σ^*, então podemos utilizar esse algoritmo para decidir se uma dada gramática irrestrita gera a linguagem vazia, o que sabemos ser impossível, logo a prova da Parte (a) está completa.

(b) Se pudéssemos dizer que duas gramáticas geram a mesma linguagem, então seríamos capazes de dizer se uma gramática livre de contexto gera Σ^*: considere a segunda gramática como sendo uma gramática trivial que realmente gera Σ^*.

(c) Se pudéssemos dizer que quaisquer dois autômatos de pilha são equivalentes, então seríamos capazes de dizer se duas dadas gramáticas livres de contexto são equivalentes, transformando-as em dois autômatos de pilha que aceitam as respectivas linguagens e, então, testá-las quanto à equivalência.

(d) Se houvesse um algoritmo para minimizar o número de estados de um autômato de pilha, similar ao que existe para autômatos finitos, então seria-

mos capazes de dizer se um dado autômato de pilha aceita Σ^*: isto ocorreria se e somente se o autômato de pilha otimizado tivesse um estado e aceitasse Σ^*. É decidível o problema de verificar se um autômato de pilha com um único estado aceita ou não Σ^* (isso é estudado no Problema 5.5.1). ■

Problemas para a Seção 5.5

5.5.1 Mostre que, dado um autômato de pilha *M com um estado*, é decidível se $L(M) = \Sigma^*$. (*Sugestão*: Mostre que tal autômato aceitará todas as possíveis cadeias se, e somente se, ele aceitar todas as cadeias de comprimento *um*.)

5.5.2 Um **sistema de correspondência de Post** é um conjunto finito P de pares ordenados de cadeias não-vazias; isto é, P é um subconjunto finito de $\Sigma^* \times \Sigma^{*\dagger}$. Uma **correspondência** de P é qualquer cadeia $w \in \Sigma^*$ tal que, para algum $n > 0$ e alguns (não necessariamente distintos) pares $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$, $w = u_1u_2\ldots u_n = v_1v_2\ldots v_n$.
(a) Mostre que é indecidível, dado um sistema de correspondência de Post, determinar se ele possui uma correspondência. (Iniciar com uma versão restrita, na qual as correspondências devem partir de um par distinto em P.)
(b) Utilize (a) acima para sugerir uma prova alternativa para o Teorema 5.5.2.

5.5.3 Um **autômato finito com dois cabeçotes** (não-determinístico) contém um controle finito, uma fita de entrada e dois cabeçotes que podem ler mas não podem gravar na fita, e se movem somente da esquerda para a direita. A máquina é iniciada em seu estado inicial com ambos os cabeçotes na célula mais à esquerda da fita. Cada transição é da forma (q, a, b, p), onde q e p são estados e a e b são símbolos ou ε. Essa transição significa que M pode, quando no estado q, ler a com seu primeiro cabeçote e b com seu segundo cabeçote, e transitar para o estado p. M aceita uma cadeia de entrada movendo ambos os cabeçotes para a extremidade direita da fita, transitando ao mesmo tempo para um estado final diferenciado.
(a) Mostre que é decidível, dado um autômato finito de dois cabeçotes M e uma cadeia de entrada w, determinar se M aceita w.
(b) Mostre que é indecidível, dado um autômato finito de dois cabeçotes M, determinar se existe alguma cadeia que seja aceita por M. (Utilize o problema anterior.)

5.6 UM PROBLEMA INSOLÚVEL DE LADRILHAMENTO

Recebemos um conjunto finito de *ladrilhos*, cada qual ocupa uma célula unitária. Pede-se dispor lado a lado, no primeiro quadrante do plano, cópias

† N. de R.T. Mais rigorosamente, como as cadeias são obrigatoriamente não-vazias, P é subconjunto de $\Sigma^* \times \Sigma^*$ (o que não invalida a afirmação em questão).

desses ladrilhos, conforme mostra a Figura 5-1. Temos um suprimento infinito de cópias dos ladrilhos.

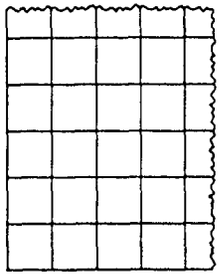

Figura 5-1

As únicas restrições são de que um ladrilho especial, o *ladrilho inicial*, deve ser posicionado no canto inferior esquerdo; que somente certos pares de ladrilhos podem se justapor horizontalmente; e que somente certos pares de ladrilhos podem se justapor verticalmente. (Os ladrilhos não podem ser girados nem empilhados.) Existe um algoritmo para determinar se o primeiro quadrante pode ser ladrilhado, dado um conjunto finito de ladrilhos, o ladrilho inicial e as regras de justa posição.

Esse problema pode ser formalizado como segue. Um **sistema de ladrilhamento** (*tiling system*) é uma quádrupla $\mathcal{D} = (D, d_0, H, V)$, onde D é um conjunto finito de **ladrilhos**, $d_0 \in \mathcal{D}$ e $H, V \subseteq D \times D$. Um **ladrilhamento efetuado** por $\mathcal{D}$ é uma função f: $\mathbf{N} \times \mathbf{N} \to D$, tal que:

$$f(0, 0) = d_0,$$
$$(f(m, n), f(m + 1, n)) \in H \quad \text{para todo } m, n \in \mathbf{N},$$
$$(f(m, n), f(m, n + 1)) \in V \quad \text{para todo } m, n \in \mathbf{N}.$$

Teorema 5.6.1: *É indecidível o problema de determinar, dado um sistema de ladrilhamento, se existe algum ladrilhamento possível para esse sistema.*

Prova: Reduzimos a esse problema de ladrilhamento o problema de determinar, dada uma máquina de Turing M, se esta *falha em parar* em resposta à entrada ε. Isso corresponde simplesmente ao *complemento* do problema de parada e, portanto, é um problema indecidível. Se esse problema puder ser reduzido ao problema de ladrilhamento, então o problema de ladrilhamento será seguramente indecidível.

A idéia básica é construir, a partir de qualquer máquina de Turing M, um sistema de ladrilhamento $\mathcal{D}$ tal que um ladrilhamento efetuado por $\mathcal{D}$, se existir, representa uma computação infinita executada por M, executada sobre uma fita em branco. As configurações de M são representadas horizontalmente em um ladrilhamento; configurações sucessivas aparecem listadas verticalmente: a dimensão horizontal representa a fita de M, enquanto

a dimensão vertical representa o tempo. Se M nunca pára em resposta à entrada vazia, sucessivas linhas podem ser ladrilhadas *ad infinitum*; mas se M pára após k passos, será impossível ladrilhar mais que k linhas.

Ao construir as relações H e V, é útil considerar que as bordas dos ladrilhos estejam marcadas com certas informações; permitimos que dois ladrilhos se justaponham horizontal ou verticalmente somente se as marcas das bordas justapostas forem idênticas. Nas bordas horizontais, essas marcas ou são um símbolo do alfabeto de M ou uma combinação estado-símbolo. O sistema de ladrilhos é organizado de tal modo que, se um ladrilhamento for possível, então, procurando nas marcas das bordas horizontais entre a n-ésima e $(n + 1)$-ésima linhas de ladrilhos, podemos ler a configuração de M após $n - 1$ passos de sua computação. Portanto, somente uma aresta ao longo dessa borda é marcada com um par estado-símbolo; as outras bordas são marcadas com símbolos apenas.

A marcação em uma aresta vertical de um ladrilho ou não existe (ela somente corresponde a arestas verticais, também sem marcações) ou consiste de um estado de M, associado a um indicador "direcional", que indicamos por uma ponta de seta. (Duas exceções são indicadas no item (e) abaixo.) Essas marcações nas arestas verticais são utilizadas para comunicar um movimento do cabeçote para a esquerda ou para a direita, de um ladrilho para o próximo.

Para ser mais específico, seja $M = (K, \Sigma, \delta, s, H)$. Então $\mathcal{D} = (D, d_0, H, V)$, onde D contém os seguintes ladrilhos:

(a) Para cada $a \in \Sigma$ e $q \in K$, os ladrilhos ilustrados na Figura 5-2, simplesmente repassam inalterados de uma configuração para outra quaisquer símbolos encontrados no lado inferior do ladrilho.

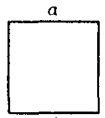

Figura 5-2

(b) Para cada $a \in \Sigma$ e $q \in K$ tal que $\delta(q, a) = (p, b)$, onde $p \in K$ e $b \in \Sigma$, o ladrilho mostrado na Figura 5-3 repassa a posição do cabeçote para o lado superior do ladrilho e altera apropriadamente o estado e o símbolo correntes.

Figura 5-3

(c) Para cada $a \in \Sigma$ e $q \in K$, tal que $\delta(q, a) = (p, \rightarrow)$ para algum $p \in K$ e para cada $b \in \Sigma - \{\ \}$, os ladrilhos mostrados na Figura 5-4 repassam movimentos do cabeçote um ladrilho da esquerda para a direita, enquanto mudam o estado apropriadamente. Note que, naturalmente, não é permitido que nenhum estado avance o símbolo de final esquerdo ▷ a partir da esquerda.

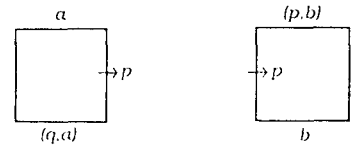

Figura 5-4

(d) Ladrilhos similares aos definidos em (c) para o caso em que $\delta(q, a) = (p, \leftarrow)$ são ilustrados na Figura 5-5. O símbolo ▷ não é exceção neste caso.

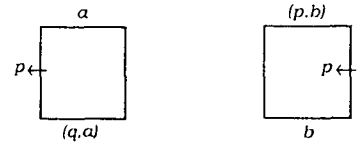

Figura 5-5

Esses ladrilhos fazem a maior parte da simulação de M por $\mathcal{D}$. Resta especificar alguns ladrilhos que iniciam a computação e asseguram que a linha inferior esteja corretamente ladrilhada.

(e) O ladrilho inicial d_0 é ilustrado na Figura 5-6(a). Ele especifica, na sua aresta superior, o estado inicial s de M e o símbolo ▷, isto é, em vez de ter M iniciado na configuração $(s, \triangleright\sqcup)$, consideramos que ele inicia em $(s, \triangleright)$; por nossa convenção relativa a ▷, sua próxima configuração será obrigatoriamente $(s, \triangleright\sqcup)$. A aresta direita do ladrilho inicial é marcada com o símbolo de espaço em branco; essa aresta pode justapor-se somente à aresta esquerda de nosso último ladrilho, mostrada na Figura 5-6(b), o qual, por sua vez, propaga para a direita a informação de que a aresta superior de cada ladrilho na linha inferior é marcada com branco.

Figura 5-6

Isso completa a construção de $\mathcal{D}$. O conjunto de ladrilhos do tipo (e) assegura que a aresta entre as primeiras duas linhas é marcada $(s, \triangleright)$ ⊔ ⊔ ⊔...; os outros ladrilhos forçam cada aresta subseqüente a ser marcada de modo correto. Note que nenhum ladrilho menciona qualquer estado de parada, de modo que se M parar após n passos, somente n linhas podem ter sido ladrilhados.

Exemplo 5.6.1: Considere-se a máquina de Turing $(K, \Sigma, \delta, s, \{h\})$, onde $\Sigma = \{\triangleright, \sqcup\}$, $K = \{s, h\}$ e δ é dado por

$$\delta(s, \triangleright) = (s, \rightarrow),$$
$$\delta(s, \sqcup) = (s, \leftarrow).$$

Essa máquina simplesmente alterna a posição do cabeçote da esquerda para a direita e vice-versa, nunca se movendo além da primeira célula da fita. O ladrilhamento do plano, correspondente às infinitas computações de M, é mostrado na Figura 5-7. ✧

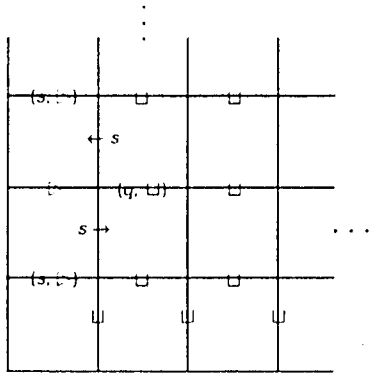

Figura 5-7

Problemas para a Seção 5.6

5.6.1 Seja $M = (\{s\}, \{a, \sqcup, \triangleright\}, \delta, s)$, onde $\delta(s, \sqcup) = (s, a)$ e $\delta(s, a) = (s, \rightarrow)$. Apresente um conjunto de ladrilhos associado a M usando o método utilizado nesta seção e ilustre as primeiras quatro linhas do ladrilhamento do plano utilizando tais ladrilhos.

5.6.2 Mostre que há algum conjunto fixo de ladrilhos D e regras de adjacência H e V, tais que o seguinte problema fica indecidível: dado um ladrilhamento parcial, isto é, um mapeamento $f: S \rightarrow D$ para algum subconjunto finito $S \subseteq \mathbf{N} \times \mathbf{N}$, tal que f obedeça às regras de adjacência, pode f ser estendido para um ladrilhamento de todo o plano?

5.6.3 Suponha que as regras do jogo de ladrilhamento sejam mudadas, de tal modo que, em vez de fixar um particular ladrilho a ser posicionado na origem, fixamos arbitrariamente um particular conjunto de ladrilhos, e estipulamos que somente esses ladrilhos podem ser utilizados para ladrilhar a primeira linha. Mostre que o problema do ladrilhamento permanece indecidível.

5.6.4 Suponha que as regras do jogo do ladrilho sejam mudadas da seguinte forma: os ladrilhos não são obrigatoriamente quadrados, mas podem ter várias elevações e entalhes ao longo de suas arestas. Dois desses ladrilhos podem ser justapostos somente se suas arestas se encaixem perfeitamente, como as peças de um quebra-cabeças; somente ladrilhos com lados perfeitamente retos podem ser utilizados nas arestas. Mostre que o problema do ladrilhamento permanece indecidível, mesmo que agora nos fosse permitido girar os ladrilhos ou empilhá-los. (Não há "ladrilho inicial" específico nessa versão.)

5.6.5. Suponha agora que consideramos o espaço de ladrilhos como sendo determinado pelas cores de suas quatro arestas e que duas arestas podem justapor-se desde que tenham a mesma cor. Mostre que, se pudermos girar os ladrilhos e empilhá-los, então qualquer conjunto não-vazio de ladrilhos pode ser utilizado para ladrilhar todo o primeiro quadrante (mesmo se continuarmos a exigir que um ladrilho especial seja posicionado na origem).

* N. de R.T. Há um erro no original em inglês, que troca o estado s por q.

5.7 PROPRIEDADES DAS LINGUAGENS RECURSIVAS

Já vimos que toda linguagem recursiva é recursivamente enumerável, mas as duas classes não são idênticas: a linguagem H evidencia a diferença. Quais linguagens recursivamente enumeráveis são recursivas? Há muitas maneiras de responder a essa questão; apresentamos uma a seguir.

Teorema 5.7.1: *Uma linguagem é recursiva se e somente se tanto ela como seu complemento forem recursivamente enumeráveis.*

Prova: Se L é recursiva, então L é recursivamente enumerável pelo Teorema 4.2.1; além disso, $\bar{L}$ é recursiva e, portanto, recursivamente enumerável, pelo Teorema 4.2.2.

Por outro lado, suponha que L seja semidecidida por M_1 e que $\bar{L}$ seja semidecidida por M_2. Podemos então construir uma máquina de Turing M que decide L. Por conveniência, descrevemos M como uma máquina de duas fitas; pelo Teorema 4.3.1, M pode ser simulado por uma máquina de uma fita. A máquina M inicia seu processamento copiando a cadeia de entrada w em ambas as fitas e posicionando seus cabeçotes na extremidade direita de ambas as cópias da entrada. Então, M simula M_1 e M_2 em paralelo: a cada passo da operação de M, um passo da computação M_1 é realizado na primeira fita e um passo da computação de M_2 é realizado na segunda fita. Uma vez que M_1 ou M_2 deve parar em w, mas não ambos, M acaba por atingir uma situação na qual ou a versão simulada de M_1 ou a de M_2 esteja para parar. Quando isso acontecer, M determina qual das duas máquinas estava para parar e pára com y ou n, correspondentemente. ■

Há uma interessante caracterização alternativa das linguagens recursivamente enumeráveis: elas são exatamente aquelas que podem ser *enumeradas* por alguma máquina de Turing.

Definição 5.7.1: Dizemos que uma máquina de Turing M **enumera** uma linguagem L se e somente se para algum estado fixo q de M,

$$L = \{w: (s, \triangleright\sqcup) \vdash_M^* (q, \triangleright\sqcup w)\}.$$

Uma linguagem é **Turing - enumerável** se e somente se existir uma máquina de Turing que a enumera.

Em outras palavras, M enumera L partindo dos espaços em branco da fita e prosseguindo a computação, passando periodicamente por um estado especial q (que *não* seja um estado de parada). Atingir o estado q sinaliza que a cadeia corrente da fita de M é um membro da linguagem L; M pode então partir do estado q e atingi-lo novamente mais tarde com algum outro membro da linguagem L em sua fita. Note-se que os membros de L podem estar relacionados nessa fita em qualquer ordem, e com repetições.

Teorema 5.7.2: *Uma linguagem é recursivamente enumerável se e somente se ela for Turing-enumerável.*

Prova: Seja L uma linguagem que é semidecidida pela máquina de Turing M. Nesse caso, é possível projetar uma máquina M', a qual, em lugar de utilizar uma cadeia de entrada, opera com a fita de entrada inicialmente vazia, e sistematicamente gera (por exemplo, em ordem lexicográfica) todas as cadeias possíveis sobre o alfabeto de L, realizando sobre cada uma dessas cadeias a mesma computação que M iria realizar. Infelizmente, a maneira óbvia de realizar essa tarefa não tem como funcionar, pois nossa nova máquina M' não pode aguardar o término da sua computação sobre cada cadeia antes de começar a operar sobre a segunda, uma vez que ela poderia ficar eternamente ocupada processando alguma cadeia para a qual a operação de M não termina, mesmo que haja outras cadeias que M poderia aceitar, mas que ainda não foram geradas. (Essa estratégia seria bem sucedida se L fosse *decidida* por M. Veja a prova do próximo teorema.)

A solução é baseada em uma versão do procedimento "leque" que utilizamos na Seção 1.4 para mostrar que uma união de um número finito de conjuntos é emumerável. Em lugar de aguardar o término da computação sobre cada cadeia à medida que ela vai sendo gerada, M' realiza a seguinte seqüência de operações:

(Fase 1) Inicialmente M' realiza um passo da computação de M sobre a (lexicograficamente) primeira cadeia sobre o alfabeto de M.
(Fase 2) Depois, ela realiza dois passos da computação de M sobre cada uma das primeiras *duas cadeias*.
(Fase 3) Em seguida, ela realiza três passos da computação de M sobre cada uma das três primeiras cadeias, e assim por diante.

Na primeira vez em que M' concluir que M iria aceitar uma cadeia, digamos w_1, M' grava w_1 em sua fita e vai para o estado q, para sinalizar que $w_1 \in L$.

No caso geral, após w_i, a i-ésima cadeia de linguagem L, ter sido descoberta, M' a exibe inicialmente no estado q e reinicia o processamento a partir da Fase 1, e assim por diante, *mantendo w_i em sua fita*. Sempre que M' encontra uma nova cadeia aceita por M, M' primeiro a compara com w_i; se forem diferentes, M' continua. Se M' constatar que a cadeia recém-descoberta coincide com w_i, então M' procura *a próxima* cadeia que seja aceita por M.

Essa próxima cadeia será w_{i+1}. Novamente, w_{i+1} é exibida no estado q, memorizada e comparada. É claro que qualquer membro de L irá ser exibido em algum momento do processamento.

A segunda parte da prova é relativamente simples. Se M enumera L, então podemos modificar M para semidecidir L como segue: reprojetamos M, de tal modo que memorize qualquer entrada a ela fornecida antes de iniciar seu processo de enumeração. Além disso, para cada vez que M fosse transitar para seu estado especial q, a máquina modificada compara o conteúdo corrente da fita com a cadeia de entrada memorizada. Se uma correspondência for detectada, a cadeia de entrada é aceita; caso contrário, o processo de enumeração prossegue. Dessa forma, a nova máquina semidecide exatamente a linguagem enumerada por M. ■

E quanto às *linguagens recursivas*? Revela-se que elas podem ser enumeradas de um modo mais ordenado.

Definição 5.7.2: Seja M uma máquina de Turing que enumera a linguagem L. Sendo q um estado diferenciado "de exibição", dizemos que M **enumera lexicograficamente** L, se sempre que $(q, \triangleright\sqcup w) \vdash_M^+ (q, \triangleright\sqcup w')$, então w' vem lexicograficamente após w. Uma linguagem é dita **lexicograficamente Turing enumerável** se e somente se existe uma máquina de Turing que a enumera lexicograficamente.

Teorema 5.7.3: *Uma linguagem é recursiva se e somente se ela for lexicograficamente Turing enumerável.*

Prova: Seja M uma máquina de Turing que decide L. Então, a seguinte máquina de Turing M' (a qual foi utilizada em nossa primeira tentativa de provar a primeira parte do Teorema 5.7.2) enumera lexicograficamente L: M' gera, uma após a outra, em uma ordem lexicográfica, todas as cadeias sobre o alfabeto de L, e executa M sobre cada uma dessas cadeias. Sempre que M aceitar a cadeia, M' exibe a mesma e prossegue para a próxima. Se M a rejeita, M' simplesmente prossegue para a próxima cadeia sem passar pelo estado de exibição.

Por outro lado, caso L seja enumerada lexicograficamente por uma máquina de Turing M, há dois casos a considerar: se L é *finita*, então não há nada a provar, uma vez que nesse caso L é certamente recursiva (bem como livre de contexto, regular, etc.). Assim, suponhamos que L seja infinita. A seguinte máquina M' decide L: operando sobre a cadeia de entrada w, M' começa a enumerar de acordo com a máquina M. Aguarda até que ou w seja exibida ou que qualquer cadeia lexicograficamente posterior a w venha a ser exibida. No primeiro caso, aceita w, e no segundo, rejeita-a. Como existe um número finito de cadeias lexicograficamente anteriores a w (menos que $|\Sigma + 1|^{|w|}$) e como L é infinita, podemos ter certeza que uma das duas irá certamente acontecer. ■

Cada máquina de Turing M semidecide uma única linguagem $L(M)$, que é, o conjunto de todas as cadeias de entrada para as quais ela pára. Mas

$L(M)$ é semidecidida por muitas outras máquinas de Turing, variando desde triviais modificações de M (por exemplo, uma versão de M com seus estados renumerados, ou uma máquina que percorre uma seqüência arbitrária desnecessária de novos estados imediatamente antes de parar, sendo nos outros casos idêntica a M) passando por muitas variantes sutis (convidamos o leitor a apresentar algumas). Em outras palavras, essa função do conjunto de todas as máquinas de Turing para a classe de linguagens recursivamente enumeráveis está longe de ser um isomorfismo, uma vez que mapeia uma mesma linguagem em uma infinidade de máquinas bastante diferentes. De fato, sabemos que é indecidível determinar se duas máquinas são mapeadas para a mesma linguagem por esse mecanismo de mapeamento. O teorema abaixo sugere que esse mapeador é tão complicado que, em um certo sentido, abaixo esclarecido, *todos os problemas concebíveis acerca dele são indecidíveis.*

Teorema 5.7.4 (Teorema de Rice): Seja $\mathcal{C}$ um subconjunto próprio não-vazio da classe das linguagens recursivamente enumeráveis. Então, o seguinte problema é indecidível: dada uma máquina de Turing M, é $L(M) \in \mathcal{C}$?

Prova: Podemos assumir que $\emptyset \notin \mathcal{C}$ (caso contrário, repetimos o restante do argumento para a classe de todas as linguagens recursivamente enumeráveis *não* pertencentes a $\mathcal{C}$, a qual é também um subconjunto não-vazio próprio das linguagens recursivamente enumeráveis). A seguir, sendo $\mathcal{C}$ não vazio, podemos assumir que existe uma linguagem $L \in \mathcal{C}$ que seja semidecidida pela máquina M_L.

Devemos reduzir o problema da parada ao problema de decidir se a linguagem semidecidida por uma dada máquina de Turing está ou não em $\mathcal{C}$. Suponha, então, que nos seja apresentada uma máquina de Turing M e a entrada w, e que desejamos decidir se M pára em resposta à cadeia de entrada w. Para tanto, construímos uma máquina de Turing $T_{M,w}$, tal que a linguagem semidecidida por $T_{M,w}$ ou é a linguagem L fixada acima ou então é a linguagem $\emptyset$. Ao processar a entrada x, $T_{M,w}$ simula a máquina de Turing universal U com a cadeia de entrada "M" "w". Se ela pára, então, em vez de parar, $T_{M,w}$ continua a simular M_L com a entrada x: ou ela pára e aceita a cadeia ou nunca pára, dependendo do comportamento de M_L em relação a x. Naturalmente, se U("M" "w") =↗, então $T_{M,w}(x) =$ ↗ também. Resumindo, $T_{M,w}$ é dada por:

$T_{M,w}(x)$: se U("M" "w") $\neq$ ↗ então $M_L(x)$ senão ↗

Afirmação: A linguagem semidecidida por $T_{M,w}$ pertence à classe $\mathcal{C}$ se e somente se M parar em resposta à entrada w.

Esse enunciado declara que a construção de $T_{M,w}$ a partir de M e w corresponde a uma *redução* do problema de parada da máquina de Turing, para o problema de responder, dada uma máquina de Turing, se a linguagem semidecidida por ela está ou não em $\mathcal{C}$. Isso concluiria a prova do teorema.

Prova: Suponha que M pare em relação à entrada w. Então, $T_{M,w}$ com a entrada x, detecta esta parada aceita x se e somente se $x \in L$. Portanto, nesse caso, a linguagem semidecidida por $T_{M,w}$ é L, que é um elemento de $\mathcal{C}$.

Suponha então que $M(w) =$ ↗. Nesse caso $T_{M,w}$ *nunca* pára e portanto, M_x semidecide a linguagem $\emptyset$, que, como se sabe, não pertence a $\mathcal{C}$. ■

A indecidibilidade de muitos problemas decorre do Teorema de Rice: dada uma máquina de Turing M, a linguagem semidecidida por ela, $L(M)$, é regular? Livre de contexto? Finita? vazia? Σ^*? Recursiva? – e assim por diante.

Problemas para a Seção 5.7

5.7.1 Mostre que L é recursivamente enumerável se e somente se, para alguma máquina de Turing não-determinística M, $L = \{w: (s \sqcup w) \vdash_M^* (h \sqcup w)\}$, onde s é o estado inicial de M.

5.7.2 Mostre que se uma linguagem é recursivamente enumerável, então há uma máquina de Turing que a enumera sem jamais repetir qualquer elemento da linguagem.

5.7.3 (a) Seja Σ um alfabeto que não contém o símbolo ";" e seja $L \subseteq \Sigma^*;\Sigma^*$ recursivamente enumerável. Mostre que a linguagem $L' = \{x \in \Sigma^*: x;y \in L \text{ para algum } y \in \Sigma^*\}$ é recursivamente enumerável.
(b) L' em (a) é necessariamente recursiva se L for recursiva?

5.7.4 Dizemos que uma gramática é **sensível ao contexto** se e somente se cada regra estiver na forma $u \rightarrow v$, onde $|v| \geq |u|$. Uma **linguagem sensível ao contexto** é gerada por uma gramática sensível ao contexto.
(a) Mostre que toda linguagem sensível ao contexto é recursiva. (*Sugestão*: que comprimento as derivações podem apresentar?)
(b) Mostre que uma linguagem é sensível ao contexto se e somente se ela for gerada por uma gramática tal que cada regra esteja na forma $uAv \rightarrow uwv$, onde A é um não-terminal e $w \neq \varepsilon$.
(c) Um **autômato linearmente limitado (in-place acceptor)** é uma máquina de Turing não-determinística tal que o cabeçote da fita nunca visita uma célula em branco exceto quanto às duas células imediatamente à direita e à esquerda da célula corrente da entrada. Mostre que uma linguagem é sensível ao contexto, se e somente se ela for semidecidida por um autômato linearmente limitado. (*Sugestão*: Ambas as partes da demonstração são especializações da prova do Teorema 4.6.1.)

5.7.5 A classe das linguagens sensíveis ao contexto, apresentadas no problema anterior e caracterizadas alternativamente por autômato linearmente limitado na Parte (c) do problema 5.7.4 acima, completa a **hierarquia de Chomsky**, uma importante estrutura de pensamento para classificar a expressividade dos geradoras de linguagens e o poder dos autômatos, proposta pelo lingüista Noam Chomsky em 1960. A hierarquia de Chomsky consiste nessas cinco classes de linguagens, na ordem em que foram introduzidas neste livro: *linguagens regulares, linguagens livres de contexto, linguagens recursivas, linguagens recursivamente enumeráveis e linguagens sensíveis ao contexto.* Organize essas classes de linguagens em ordem de generalidade crescente (de modo que cada classe da lista inclua apropriada-

mente todas as anteriores) e escreva ao lado de cada uma a classe correspondente de autômatos e/ou gramáticas.

5.7.6 Seja f: $\Sigma_0^* \to \Sigma_1^*$ uma função recursiva bijetora. Mostre que há uma função recursiva g: $\Sigma_1^* \to \Sigma_0^*$, tal que $f(g(w)) = w$ para cada $w \in \Sigma_0^*$.

5.7.7 Mostre que é indecidível responder se a linguagem semidecidida por uma dada máquina de Turing é
(a) finita.
(b) regular.
(c) livre de contexto.
(d) recursiva.
(e) igual à linguagem $\{ww^R: w \in \{a, b\}^*\}$.

5.7.8 As linguagens não-recursivas L, exibidas neste capítulo, têm a propriedade de que ou L ou $\bar{L}$ é recursivamente enumerável.
(a) Mostre, através da utilização de contagem, que existem linguagens L tais que nem L e nem $\bar{L}$ seja recursivamente enumerável.
(b) Dê um exemplo de tal linguagem.

5.7.9 Esse problema é continuação do problema 3.7.10: estenda o diagrama de Venn
(a) das linguagens regulares,
(b) das linguagens livres de contexto,
(c) das linguagens determinísticas livres de contexto e
(d) do complemento das linguagens livres de contexto para incluir as três novas classes seguintes:
(e) recursivas,
(f) recursivamente enumeráveis,
(g) complementos das linguagens recursivamente enumeráveis.
Dê um exemplo de uma linguagem em cada região desse diagrama de Venn.

5.7.10 Descreva uma máquina de Turing M que tem a propriedade de, em resposta a qualquer entrada, gerar "M" – *sua própria descrição*. (*Sugestão*: escreva inicialmente um programa em uma linguagem de programação que imprima a si próprio. Ou, para um argumento mais complicado, veja o próximo problema.)

5.7.11 (a) Discuta a existência de uma máquina de Turing G que, acionada tendo como entrada a descrição de uma máquina de Turing M, computa a descrição de outra máquina de Turing M_2, a qual, qualquer que seja sua entrada x, primeiramente simula M com a entrada "M" e, então, se M parar com uma descrição válida de uma máquina de Turing N em sua fita, simula N usando a entrada x.
(b) Discuta a existência de uma máquina de Turing C que, acionada tendo como entrada a descrição de duas máquinas de Turing M_1 e M_2, computa a descrição de uma máquina de Turing que é a *composição* de M_1 e M_2. Em outras palavras a máquina que, para qualquer entrada x, primeiro simula M_2 para a entrada x e, então, simula M_1 usando como entrada o resultado anteriormente obtido.
(c) Seja M uma máquina de Turing que, ativada tendo como entrada uma descrição válida de máquina de Turing, produz uma descrição válida de outra máquina de Turing. Mostre que qualquer M desse tipo tem um *ponto fixo*, isto é, uma máquina de Turing F com a propriedade de que F e a máquina representada por M("F") comportam-se de maneira exatamente igual para todas as possíveis entradas. (Suponha que a composição – lembre-se da Parte (b) – de M e G – lembre-se da Parte (a) – é apresentada como uma entrada para G. Prove que a máquina cuja descrição é enviada para a saída corresponde ao ponto fixo F exigido.)
(d) Suponha que M seja uma máquina de Turing qualquer. Discuta a existência de outra máquina de Turing M' com a seguinte propriedade: para qualquer entrada x, se M parar em resposta à entrada x, com saída y, então M' também pára em resposta à entrada x e sua saída será y"M'" – isto é, M' preenche a saída de M com sua própria descrição, ou "assinatura."

REFERÊNCIAS

A indecidibilidade do problema da parada foi provada por A. M. Turing em seu artigo de 1936, a que fizemos referência no capítulo anterior. A indecidibilidade de problemas relacionados a gramáticas livres de contexto foi provada no artigo de Bar-Hillel, Perles e Shamir citado no fim do Capítulo 3. O problema da correspondência de Post (veja o Problema 5.5.2) é de:

- E. L. Post "A variant of a recursively unsolvable problem", *Bulletin of the American Math. Society*, 52, pp. 264-268, 1946.

A hierarquia Chomsky (veja o Problema 5.7.5) é creditada a Noam Chomsky:

- N. Chomsky "Three models for the description of language", *IRE Transactions on Information Theory*, 2, 3, pp. 113-124, 1956.

O problema do ladrilhamento (Seção 5.6) é de:

- H. Wang "Proving theorems by pattern recognition", *Bell System Technical Journal*, 40, pp. 1-141, 1961.

Para vários outros problemas insolúveis, veja as referências do último capítulo, especialmente os livros de Martin Davis.